UNIVERSIDADE da MADEIRA

Faculdade de Ciências Exatas e da Engenharia

Engenharia Informática

Programação Orientada por Objetos

Funchal, 28 de dezembro de 2024

Trabalho elaborado por:

Nelson Macedo nº 2143723
Leandro Rodrigues nº2104123
Manuel Gama nº2106723

Índice

1.Introdução

1.1.Objetivos

O projeto "POO Civilizations" tem como objetivo proporcionar uma experiência prática e consolidar o entendimento dos princípios fundamentais da programação orientada a objetos (POO), aplicados no desenvolvimento de um jogo de estratégia. Implementado em Java, o jogo simula a expansão de uma civilização, desafiando os jogadores a gerenciar recursos, fundar e desenvolver cidades, e comandar unidades como militares, colonos e construtores. A principal ênfase do projeto é o uso dos princípios SOLID, buscando uma arquitetura de software bem estruturada e modular, com foco no encapsulamento, abstração, herança e polimorfismo.

1.2.Visão Geral

O jogo é ambientado em um mapa dinâmico e estratégico, onde diferentes tipos de terrenos com propriedades distintas afetam o movimento das unidades e a produção de recursos. O jogador deve administrar o crescimento de sua civilização, otimizar a produção de alimentos, ouro e recursos industriais, e expandir seu território de forma estratégica. As cidades, que são o centro do desenvolvimento, geram recursos e unidades, mas a sua expansão deve respeitar restrições de espaço e distância, promovendo um planejamento territorial cuidadoso.

Além de construir cidades, o jogador precisa desenvolver unidades específicas para explorar, construir e proteger o território. A jogabilidade envolve decisões estratégicas, como escolher onde alocar trabalhadores, otimizar a produção e o uso de ouro, e gerenciar o crescimento populacional das cidades. O objetivo final do jogo é atingir uma condição de vitória definida, como a conquista de cidades inimigas ou a acumulação de uma quantidade significativa de ouro.

O projeto é desenvolvido de forma modular, permitindo a adição de novas funcionalidades e tipos de unidades ou terrenos sem a necessidade de modificações extensas no código existente. A interface gráfica será limitada à consola Java, com prints personalizados para facilitar a interação do jogador com o sistema e proporcionar uma visualização clara e acessível das ações e status no jogo. Dessa forma, o "POO Civilizations" oferece uma plataforma para a aplicação de conceitos de POO em um contexto desafiador e envolvente, ao mesmo tempo que proporciona uma experiência de jogo estratégica e dinâmica.

2. Descrição Geral

2.1. Escopo do Projeto

1. Mapa do Jogo

O mapa é a base territorial onde o jogo se desenrola. Foi implementado como uma matriz bidimensional representando diferentes tipos de terreno. O mapa tem as seguintes características:

Tipos de Terreno: Cada célula do mapa é representada por um tipo específico de terreno, como florestas e pedreiras. Esses terrenos possuem propriedades distintas que afetam o movimento das unidades e a produção de recursos.

Terrenos Inacessíveis: Certas áreas do mapa, como água e terrenos montanhosos, foram implementadas como áreas inacessíveis, onde não é possível mover unidades ou fundar cidades.

Mapa Circular no Eixo X: O mapa foi projetado para ser circular no eixo X, permitindo que as unidades que se movem para o leste reapareçam no lado oposto, criando uma experiência contínua e imersiva.

2. Gestão da Civilização

A gestão das civilizações é uma parte crucial do jogo. Implementamos menus e funcionalidades para que o jogador possa:

Gerenciar Unidades e Cidades: O jogador pode mover unidades, atacar inimigos, construir unidades, melhorar a infraestrutura das cidades e expandir o território.

Recursos: O jogador deve gerenciar recursos como comida, produção e ouro, que são gerados pelas cidades e utilizados para sustentar a população e construir unidades e edifícios.

Tesouro da Civilização: O ouro é acumulado no tesouro da civilização, sendo utilizado como multiplicador de produção para acelerar a construção de unidades e edifícios.

3. Cidades

As cidades são a base do desenvolvimento da civilização e possuem diversas funcionalidades:

Colocação e Distância entre Cidades: Foi implementada uma verificação que garante que as cidades estão sempre a uma distância mínima de duas células uma da outra.

Geração de Recursos: Cada cidade gera recursos como comida, produção industrial e ouro, que são distribuídos de acordo com as células de trabalho alocadas aos cidadãos.

Gestão de Recursos: Os recursos gerados são utilizados para alimentar a população e construir unidades e edifícios. O sistema de crescimento populacional é gerido pelo excedente de comida acumulado.

4. Unidades

O projeto inclui a implementação de diferentes tipos de unidades, cada uma com suas funções e características:

Construtores: Responsáveis por construir melhorias no terreno, como estradas, minas e fazendas, que aumentam a produção de recursos.

Militares: Unidades com a capacidade de atacar outras unidades e defender o território da civilização. Implementamos uma mecânica simples de combate entre unidades e entre unidades e cidades.

Colonos: Unidades responsáveis por fundar novas cidades, contribuindo para a expansão territorial da civilização.

5. Condição de Vitória

A condição de vitória foi definida como a conquista de todas as cidades inimigas ou o alcance de um determinado valor de ouro. A vitória pode ser alcançada ao longo do jogo, dependendo da estratégia adotada pelo jogador.

6. Interface Gráfica e Interação

O jogo foi desenvolvido para ser executado na consola Java, com prints personalizados que facilitam a visualização das informações e a interação do jogador. A interface gráfica foi projetada para ser simples, mas funcional, permitindo que o jogador controle as ações de sua civilização por meio de menus interativos.

2.2. Requisitos Funcionais

O projeto foi concluído, cumprindo todos os requisitos obrigatórios. O mapa foi implementado com vários tipos de superfícies e com um sistema de mapa circular. A gestão da civilização abrange opções para deslocamento, controle de cidades, recursos e atividades de combate. As cidades possuem limitações de localização, gerenciam recursos e têm um sistema de aumento populacional. Diversas unidades, como construtores, militares e colonizadores, exercem papeis fundamentais para a ampliação e proteção da cidade. A condição de vitória foi implementada com foco em metas estratégicas.

2.3. Requisitos Opcionais - Não implementado

2.4. Tecnologias utilizadas

Para o desenvolvimento do projeto "POO Civilizations", foram utilizadas diversas tecnologias, linguagens de programação, bibliotecas e ferramentas que possibilitaram a implementação das funcionalidades do jogo de forma eficiente e organizada. A seguir, são apresentadas as principais tecnologias e ferramentas utilizadas:

1. Linguagem de Programação: Java

A principal linguagem utilizada no desenvolvimento do projeto foi Java, que oferece uma plataforma robusta para a implementação de sistemas orientados a objetos, sendo ideal para a construção de jogos como este. Java foi escolhida devido à sua simplicidade, portabilidade e forte suporte à programação orientada a objetos (POO), que é a abordagem principal do projeto. O Draw.io foi utilizado para criar o diagrama UML do projeto, especificamente o Diagrama de Classes, devido à sua interface intuitiva, biblioteca de elementos UML, e funcionalidades como conexões automáticas, exportação em diversos formatos e integração com serviços de nuvem, permitindo a representação clara da estrutura do sistema e facilitando a comunicação e o planejamento entre os membros da equipa.

Características do Java utilizadas:

Programação Orientada a Objetos (POO): Utilização de conceitos como classes, objetos, herança, polimorfismo, encapsulamento e abstração.

Métodos Abstratos e Interfaces: Utilização de métodos abstratos e interfaces para garantir modularidade e flexibilidade, permitindo a fácil extensão do código com novos tipos de unidades, terrenos ou outras funcionalidades.

2. Ambientes de Desenvolvimento: NetBeans e VSCode

Durante o desenvolvimento, foram utilizados dois ambientes de desenvolvimento integrados (IDEs) para facilitar o processo de codificação, depuração e execução do jogo.

NetBeans: Foi utilizado para a maioria do desenvolvimento, especialmente por sua excelente integração com Java. O NetBeans oferece recursos poderosos, como autocompletar código, depuração, e visualização de projetos, que tornam a codificação mais ágil.

VSCode: Embora o NetBeans tenha sido o principal ambiente utilizado, o VSCode também foi utilizado em algumas partes do projeto. O VSCode é um editor de código leve e altamente configurável, ideal para projetos menores ou para explorar rapidamente novas ideias e funcionalidades.

3. Estruturas de Dados

O projeto faz uso de arrays e ArrayLists para a gestão de dados dinâmicos e a manipulação do mapa, das unidades e das cidades. Essas estruturas permitem uma organização eficiente dos dados e oferecem flexibilidade para expandir ou modificar o jogo conforme necessário.

Arrays: Utilizados para representar o mapa, onde cada célula contém informações sobre o tipo de terreno ou a presença de unidades e cidades.

ArrayList: Utilizados para armazenar coleções de objetos como unidades e cidades, permitindo o ajuste dinâmico no número de elementos armazenados.

4. Abstração e Interfaces

Foi utilizado um sistema baseado em métodos abstratos e interfaces para promover a flexibilidade e modularidade do código. As interfaces e classes abstratas são usadas para definir comportamentos comuns, como o movimento das unidades, a produção de recursos pelas cidades, e as ações dos jogadores, permitindo que novos tipos de unidades, recursos ou terrenos possam ser adicionados facilmente no futuro sem a necessidade de reescrever o código.

Métodos Abstratos: Utilizados para garantir que as subclasses implementem comportamentos específicos, como a movimentação de unidades ou a produção de recursos pelas cidades.
Interfaces: Definiram contratos comuns entre diferentes tipos de objetos, como as interfaces para unidades, o que permite que a lógica do jogo seja generalizada e extensível.

5. Bibliotecas

Embora o projeto tenha sido desenvolvido principalmente com funcionalidades nativas de Java, o uso de bibliotecas externas foi evitado para manter o foco na implementação dos conceitos de POO e estruturas de dados. No entanto, a utilização de bibliotecas padrão da linguagem Java, como java.util para manipulação de listas e arrays, foi essencial para o gerenciamento eficiente de dados.

6. Consola Java

A interface do jogo foi implementada utilizando a consola do Java, onde todas as interações do jogador, como a movimentação de unidades, construção de cidades e visualização do mapa, acontecem por meio de prints personalizados. Embora o jogo tenha uma interface gráfica limitada, a escolha pela consola foi uma decisão estratégica para focar na lógica do jogo e na implementação dos conceitos de POO.

3. Arquitetura do Sistema

3.1. Diagrama de Classes

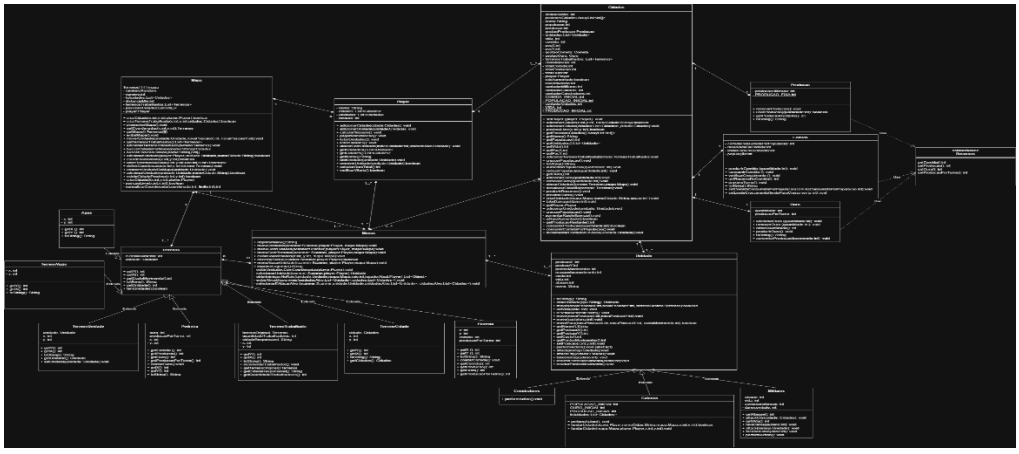


Fig.1-Diagrama UML
Nota: Em anexo está o link do draw.io para poder ver melhor o diagrama

3.2.Descrição das Classes

Classe Agua

Responsabilidade: A classe Agua representa um tipo específico de terreno no contexto de um jogo. Esse tipo de terreno é caracterizado por ser um espaço no mapa que não pode ser explorado ou trabalhado pelos jogadores, simbolizando águas que são inacessíveis. Ela serve para ser visualizada no mapa do jogo, mas não interfere nas ações de exploração, coleta ou construção, por exemplo. A principal função dessa classe é fornecer uma representação visual e lógica de áreas aquáticas dentro do jogo.

Atributos:
x (int): Representa a coordenada X do terreno de água no mapa.
y (int): Representa a coordenada Y do terreno de água no mapa.
Métodos:
Construtor Agua(int x, int y): Inicializa um terreno de água com as coordenadas específicas fornecidas (x, y). O custo de atravessar esse terreno é definido como um valor alto (9999), representando a impossibilidade de acesso.
getX(): Retorna a coordenada X do terreno de água.
getY(): Retorna a coordenada Y do terreno de água.
toString(): Retorna uma string "A", que representa visualmente a água no mapa, facilitando a visualização da água no jogo.

Relações com Outras Classes

Herança: A classe Agua herda da classe Terrenos, que é uma classe base para diferentes tipos de terrenos no jogo. Isso permite que Agua compartilhe comportamentos gerais de terrenos (como o custo de atravessar) e os personalize com atributos específicos, como o valor das coordenadas e a representação visual.

Associação: Agua interage com outras classes no sistema, através de terrenos, como o mapa onde os terrenos são armazenados e manipulados, ou classes de jogo que tratam da movimentação ou interação com os terrenos.

Classe Cidades

A classe Cidades representa uma cidade em um jogo, onde cada cidade possui vários atributos, como população, comida, produção, unidades e recursos (ouro, comida). A classe também inclui métodos para gerenciar o crescimento da cidade, produção de recursos e criação de unidades. Aqui está uma explicação detalhada da funcionalidade da classe:

Atributos:
distanciaMin (int): Representa a distância mínima que deve ser mantida entre as cidades.
posicoesCidades (ArrayList<int[]>): Lista que armazena as posições das cidades no mapa, com cada posição representada por um array de inteiros [x, y].
nome (String): Este nome é gerado automaticamente a partir de um contador estático, o qual garante que cada cidade tenha um nome único.
populacao (int): Quantidade de habitantes da cidade. Inicialmente, o valor é definido pela constante POPULACAO_INICIAL.
producao(int): Quantidade de produção por turno da cidade, que afeta a produção de recursos.
gestaoProducao (Producao): Objeto responsável pela gestão da produção de recursos na cidade.
unidades (List<Unidade>): Lista que contém as unidades criadas na cidade, como militares, colonos, e construtores.
vida (int):A quantidade de vida da cidade, que pode ser reduzida por ataques.
comida (int):A quantidade de comida disponível na cidade. A comida é essencial para o crescimento da população e a manutenção da cidade.
posX (int): Coordenada X da cidade no mapa. Define a posição horizontal da cidade no mundo virtual.
posY (int): Coordenada Y da cidade no mapa. Define a posição vertical da cidade no mundo virtual.
gestaoComida (Comida): Objeto responsável pela gestão da comida na cidade. Controla a quantidade de comida disponível e sua produção.
gestaoOuro (Ouro): Objeto responsável pela gestão do ouro da cidade. Controla a quantidade de ouro disponível e sua produção.terrenosTrabalhados (List<Terrenos>):
Lista de terrenos nos quais os cidadãos da cidade estão trabalhando, como florestas e pedreiras.
comidainicial (int):A quantidade inicial de comida disponível na cidade, definida pela constante COMIDA_INICIAL.
totalComida (int):A quantidade total de comida produzida pela cidade, com base nos terrenos trabalhados.
totalProducao (int):O valor total de produção da cidade, o qual influencia a produção de recursos em cada turno.
totalOuro (int):A quantidade total de ouro produzido pela cidade.
player (Player): Referência ao jogador responsável pela cidade.
POPULACAO_INICIAL (int): Valor constante que define a população inicial de uma cidade quando ela é criada.
PRODUCAO_INICIAL (int): Valor constante que define a produção inicial de uma cidade por turno.
COMIDA_INICIAL (int): Valor constante que define a quantidade inicial de comida disponível na cidade.
contadorCidades (static int): Contador estático usado para garantir que o nome de cada cidade seja único
raioAumentado(boolean) Indica se o raio de influência da cidade foi expandido devido ao crescimento populacional.

raioInfluencia(int) Define o alcance máximo onde a cidade pode alocar cidadãos para trabalhar terrenos.
contadorMilitares(int) Número de unidades do tipo Militares produzidas pela cidade.
contadorColonos(int) Número de unidades do tipo `Colonos` produzidas pela cidade.
contadorConstrutores(int) Número de unidades do tipo `Construtores` produzidas pela cidade.
Métodos:
setPlayer(Player player): Define o jogador responsável pela cidade.
adicionarCidade(int x, int y, String nomeCidade):Método para adicionar uma cidade no mapa, verificando se a posição está livre.
adicionarCidade(List<Cidades> cidades, Cidades cidade):Método para adicionar uma cidade na lista de cidades
posicaoLivre(int x, int y): Verifica se uma posição no mapa está livre para adicionar uma nova cidade.
adicionarTerrenoTrabalhado(TerrenoTrabalhado terreno): Adiciona o Terreno trabalhado à lista de terrenos trabalhados
crescerPopulacao(): Aumenta a população da cidade, desde que haja comida suficiente para sustentá-la e evoliu a cidade caso a sua população chegue a 10.
aumentarPopulacao(int quantidade): Aumenta a população em uma quantidade específica.
reduzirPopulacao(int quantidade): Diminui a população de acordo com a quantidade informada.
getOuro(): Retorna a quantidade de ouro disponível na cidade.
adicionarOuro(int quantidade) e removerOuro(int quantidade):Métodos para adicionar ou remover ouro da cidade.
alocarCidadao(Terrenos terreno, Mapa mapa): Aloca um cidadão para trabalhar em um terreno específico (como florestas ou pedreiras).
desalocarCidadão(Terrenos terreno): Desaloca um cidadão de um terreno específico.
produzirRecursos(): Produz recursos (comida, ouro) com base nos terrenos trabalhados.
proximoTurno(): Avança o turno, produzindo recursos e permitindo o crescimento da população.
criarUnidade(Player player, String tipo, String nomeCidade, Mapa mapa): Cria uma nova unidade na cidade (militar, colono ou construtor).
takeDamage(int dano): Reduz a vida da cidade com base no dano recebido.
adicionarUnidade(Unidade unidade): Adiciona uma unidade à cidade.
toString():Método que retorna uma string contendo informações sobre a cidade, como nome, população, comida, ouro, etc.
aumentarRaioDeInfluencia(): Aumenta o raio de influência da cidade para 4 espaços em vez de 3
isRaioAumentado():Método que verifica se o raio está aumentado.
consumirProducao(int quantidade):Método que verifica se a produção foi consumida.
consumirComidaPorPopulacao():Método que consome a comida com base na população da cidade e caso não haja comida suficiente para a população toda morrem as unidades que não comeram.

incrementarContadorUnidade(Unidade unidade): incrementa o contador correspondente ao tipo de unidade fornecida.

Associações e Herança

A classe Cidades não possui herança direta de outras classes, mas ela estabelece várias associações importantes dentro do sistema.

Primeiramente, a classe Cidades possui uma associação com a classe Producao, já que ela contém um objeto do tipo Producao para gerenciar a produção de recursos da cidade. Da mesma forma, a cidade também interage com a classe Comida, através do objeto gestaoComida, para controlar a quantidade de comida disponível e sua produção. A classe Ouro é outra associação, sendo gerida pela cidade através de um objeto gestaoOuro, permitindo o controle sobre o ouro produzido e disponível na cidade.

Além disso, a classe Cidades mantém uma associação com a classe Player, pois cada cidade está associada a um jogador específico, indicado pelo atributo player, que controla a cidade. A cidade também tem uma lista de Unidade, que representa os diversos tipos de unidades que podem ser criadas e alocadas na cidade, como militares, colonos e construtores.

No que diz respeito aos Terrenos, a classe Cidades mantém uma relação com esses objetos, especificamente com instâncias de Floresta e Pedreira, que são tipos específicos de terrenos que podem ser trabalhados para produção de recursos. A cidade pode alocar cidadãos para trabalhar nesses terrenos, o que está diretamente relacionado com a classe Terrenos e suas subclasses, como Floresta e Pedreira.

Finalmente, a classe Cidades se associa à classe Mapa, já que as cidades são posicionadas em um mapa e interagem com ele para definir a localização dos terrenos e das unidades. O mapa armazena as cidades e os terrenos, e a cidade interage com o mapa para alocar e gerenciar os recursos e cidadãos no contexto do mundo do jogo.

Classe Colonos

Responsabilidade: Representar uma unidade especializada no jogo, responsável por fundar novas cidades no mapa.
Principais Atributos:
POPULACAO_INICIAL(int): Define a população inicial da cidade que será fundada.
OURO_INICIAL(int): Define a quantidade inicial de ouro gerada pela cidade.
PRODUCAO_INICIAL(int): Define a produção inicial da cidade.
listcidades(List<Cidades>): Lista que armazena as cidades associadas a esta unidade (aparentemente não utilizada ou inicializada diretamente no construtor).

Métodos:
Colonos(String nome): Construtor da classe que inicializa a unidade com um nome e pontos de movimento padrão (10).
fundarCidade(Player player, String nomeCidade, Mapa mapa, int x, int y): Permite a unidade fundar uma nova cidade no mapa.
performAction(): Realiza uma ação específica da unidade Colono, que neste caso é a fundação de uma nova cidade. Exibe uma mensagem indicando que o colono está em ação.

Relações:Herdeira de: Unidade
Reaproveita atributos e métodos básicos como posição no mapa e pontos de movimento.
Interage com: Player, para remover a unidade e associar a cidade ao jogador. Mapa, para verificar posições disponíveis e criar cidades.

Classe Comida

Responsabilidade: A classe Comida gerencia os aspectos relacionados à comida dentro de um jogo ou simulação. Ela controla a produção, consumo e crescimento populacional com base na quantidade de comida disponível. A classe permite que a população consuma comida, cresça de acordo com a reserva de comida e lida com cenários onde a comida é insuficiente, afetando a população.

Atributos:
comidaConsumidaPorPopulacao (int): Quantidade de comida consumida por cada unidade de população por turno.
reservaDeComida (int): Quantidade total de comida disponível na reserva.
limiteParaCrescimento (int): Quantidade de comida necessária para aumentar a população.
populacao (int): Número de habitantes na população.

Métodos:

Construtor Comida(): Inicializa os valores com configurações padrão. A comida produzida e a reserva inicial são 20, a população começa com 10, e o limite de comida para crescimento também é 50.
Construtor Comida(int comidaInicial): Inicializa a quantidade de comida produzida com um valor específico fornecido como parâmetro.
produzirComida(): Aumenta a reserva de comida pela quantidade definida em **comidaProduzida():** Esse método é chamado a cada turno para adicionar comida à reserva.
consumirComida(): Deduz a quantidade de comida consumida pela população (baseado em populacao e comidaConsumidaPorPopulacao). Se a reserva de comida for insuficiente, a população é reduzida em 1 e a reserva é zerada.
verificarCrescimento(): Verifica se a comida disponível é suficiente para permitir o crescimento populacional. Se a comida for suficiente (superior ao limiteParaCrescimento), a população aumenta e a comida necessária para o crescimento é retirada da reserva.
getReservaDeComida(): Retorna a quantidade atual de comida na reserva.
proximoTurno(): Avança para o próximo turno realizando as seguintes ações em sequência:
Produzir comida.
Consumir comida com base na população.
Verificar se a população pode crescer com base na comida disponível.
toString(): Retorna uma representação textual do estado atual da comida e da população, no formato "População: X, Reserva de Comida: Y".
setComidaConsumidaPorPopulacao(int comidaConsumidaPorPopulacao): Define a quantidade de comida consumida por unidade de população por turno.

setLimiteParaCrescimento(int limiteParaCrescimento): Define a quantidade de comida necessária para o crescimento populacional.

Relações com Outras Classes

Associação com a População: A classe Comida tem uma associação indireta com a Cidades, uma vez que o crescimento populacional depende da quantidade de comida disponível. A classe também afeta a população diretamente ao reduzir o número de habitantes quando a comida é insuficiente.
Associação com o Jogo ou Sistema de Turnos: A classe Comida interage com o ciclo de turnos do jogo, pois as ações de produção, consumo e crescimento ocorrem a cada turno, conforme indicado pelo método proximoTurno().

Classe Construtores

Responsabilidade: A classe Construtores é responsável por realizar melhorias no terreno, como a construção de estradas, e por reparar infraestruturas danificadas, desempenhando um papel vital no desenvolvimento e manutenção do território no jogo.

Métodos:
performAction(): Executa a ação da unidade. No caso dos construtores, a ação padrão é ficar em construir uma melhoria no terreno.

Classe Floresta

Responsabilidade: A classe Floresta representa um tipo de terreno no jogo que fornece recursos como comida e produção por turno. As florestas podem ser exploradas para coletar comida, mas não produzem ouro ou produção acumulada diretamente. A classe implementa a interface Recursos, o que implica que ela deve fornecer os métodos necessários para acessar os recursos associados a ela, como comida e produção.

Atributos:
comida (int): Quantidade de comida coletada da floresta. Inicialmente, está definida como zero e é aumentada conforme a coleta.
producaoPorTurno (int): Quantidade de produção que a floresta gera a cada turno (definido como 10 por padrão).
x (int): Coordenada X do terreno no mapa.
y (int): Coordenada Y do terreno no mapa

Métodos:

Construtor Floresta(int x, int y): Inicializa uma instância de Floresta com coordenadas específicas no mapa (x, y). Também define um custo fixo de 4 para movimentação neste tipo de terreno, utilizando o construtor da classe pai Terrenos.
getComida(): Retorna a quantidade de comida acumulada na floresta. O valor é inicialmente 0 e é incrementado conforme a coleta de comida.

getProducao(): Retorna 0, pois a floresta não gera produção acumulada direta.
getOuro(): Retorna 0, já que florestas não geram ouro.
getProducaoPorTurno(): Retorna o valor de producaoPorTurno, que é 10, representando a quantidade de produção gerada pela floresta a cada turno.
toString(): Retorna uma representação textual do terreno. A letra "F" é usada para representar a floresta no mapa.
coletarComida(): Aumenta a quantidade de comida acumulada na floresta com base na produção por turno (10 unidades de comida a cada coleta).
getX(): Retorna a coordenada X da floresta.
getY(): Retorna a coordenada Y da floresta.

Relações com Outras Classes

Herança: A classe Floresta herda de Terrenos, o que significa que ela possui as propriedades e comportamentos básicos de um terreno, como o custo de movimentação.
Implementação da Interface Recursos: A classe Floresta implementa a interface Recursos, o que a obriga a fornecer implementações para os métodos relacionados aos recursos, como comida, produção e ouro. Embora florestas não gerem ouro nem produção acumulada diretamente, a interface é cumprida ao retornar valores adequados (0 para ouro e produção acumulada).

Classe Mapa

A classe Mapa representa o mapa do jogo e oferece funcionalidades para manipulação de terrenos, cidades e unidades. A seguir, estão as descrições dos principais atributos e métodos desta classe.

Atributos:
mapa(Terrenos[][]): Um array bidimensional que armazena os terrenos presentes no mapa. Cada posição pode conter diferentes tipos de terrenos como florestas, pedreiras, cidades ou terrenos vazios.
random(Random): Um objeto da classe Random, usado para gerar números aleatórios, especialmente na criação aleatória de terrenos.
numero(int): Um número gerado aleatoriamente para determinar o tipo de terreno a ser colocado no mapa.
listcidades(List<Cidades>): Uma lista que mantém as cidades criadas no mapa.
distanciaMin(int): Define a distância mínima entre duas cidades no mapa.
terrenosTrabalhados(List<Terrenos>): Uma lista que contém os terrenos que foram trabalhados (como florestas e pedreiras) pelas cidades.
posicoesCidades(List<int[]>): Armazena as coordenadas das cidades para evitar a criação de cidades muito próximas.
player(Player): Representa o jogador associado ao mapa, permitindo a interação do jogador com o mundo do jogo.

Métodos:

criarCidade(int x, int y, Player player): Este método cria uma nova cidade no mapa, verificando se a posição é válida (se o terreno é adequado e se está suficientemente distante de outras cidades). Ele adiciona a cidade à lista de cidades e altera o terreno na posição correspondente para um TerrenoCidade.

criarTerrenoTrabalhado(int x, int y, Cidades cidade): Cria um TerrenoTrabalhado (floresta ou pedreira) em uma posição específica, caso o terreno seja adequado e a posição esteja livre.

preencherMapa(): Preenche o mapa com terrenos aleatórios, que podem ser florestas, pedreiras, águas ou terrenos vazios.

posicaoLivre(int x, int y): Verifica se uma posição no mapa está livre para ser ocupada por uma cidade ou unidade, com base em uma distância mínima de outras cidades.

exibirMapa(): Exibe o mapa no console, representando cada tipo de terreno com caracteres específicos, como "T" para terrenos trabalhados, "F" para florestas, "P" para pedreiras, "C" para cidades e outros para diferentes unidades.

moverUnidade(Unidade unidade, int novaPosicaoX, int novaPosicaoY): Move uma unidade do mapa de uma posição para outra, verificando se o movimento é válido com base nos pontos de movimento restantes da unidade e o custo de movimento do terreno de destino.

adicionarUnidade(Unidade unidade, String nomeCidade): Adiciona uma unidade ao mapa em uma posição ao redor de uma cidade, verificando se há espaço disponível.

buscarCidadePorNome(String nome): Busca uma cidade pelo nome, retornando a cidade correspondente, ou null se não encontrada.

buscarPosicaoCidade(String nome): Retorna as coordenadas de uma cidade a partir de seu nome.

coordenadaValida(int x, int y): Verifica se as coordenadas fornecidas estão dentro dos limites do mapa.

obterTerrenosNoRaio(int x, int y, int raio): Retorna uma lista de terrenos em um raio especificado ao redor de uma posição no mapa.

definirCoordenadas(int x, int y, Terrenos terreno): Define um terreno específico em uma posição no mapa, desde que a coordenada seja válida.

removerUnidadeDoMapa(Unidade unidade): Remove uma unidade do mapa, alterando a posição da unidade para um TerrenoVazio.

adicionarUnidade(Player player, Unidade unidade, String nomeCidade)(boolean): Adiciona uma unidade ao mapa próximo a uma cidade especificada.

existeCidadeProxima(int x, int y)(boolean): Verifica se há uma cidade próxima a uma posição específica.

criarCidade2(int x, int y, Player player)(boolean): Verifica se o terreno está ocupado por uma unidade específica (colono) e se é adequado para construção.

posicaoLivre(int x, int y)(boolean): Verifica se uma posição específica está livre, considerando uma distância mínima entre cidades.

normalizarCoordenada(int coordenada, int limite)(int): Normaliza uma coordenada, garantindo que ela esteja dentro dos limites do mapa.

Relações com Outras Classes:

Cidades: A classe Mapa tem uma relação de associação com a classe Cidades, pois contém uma lista de cidades e oferece métodos para criá-las, localizar suas posições e interagir com elas.
Player: A classe Mapa está associada ao Player, uma vez que o mapa pertence a um jogador e contém métodos para adicionar unidades e cidades controladas por esse jogador.
Terrenos: Mapa interage com diferentes tipos de Terrenos (como Floresta, Pedreira, TerrenoVazio, TerrenoCidade), os quais são manipulados para definir o tipo de cada posição no mapa.
Unidade: Mapa gerencia a movimentação de Unidade, associando-as a terrenos e controlando seus movimentos no mapa.
TerrenoTrabalhado: A classe também tem associação com o TerrenoTrabalhado, que representa os terrenos trabalhados pelas cidades, como florestas e pedreiras.

Classe Menus

Responsabilidade:

A classe Menus é responsável por exibir e gerenciar os menus de interação com o jogador, oferecendo opções de movimentação de unidades, ataques, gerenciamento de cidades e terrenos, e encerramento de turnos. Ela implementa a lógica de navegação entre as diferentes opções do jogo e interação com o jogador através de entradas no console.

Atributos: Não há atributos declarados diretamente na classe, pois ela é focada em fornecer menus e funcionalidades interativas.

Métodos:

ImprimirMenu(): Exibe o menu principal com várias opções.
menuUnidades(): Exibe o menu para movimentar unidades, permitindo ao jogador escolher e mover uma unidade no mapa.
selecionarUnidade(): Auxilia na escolha de uma unidade para ataque.
obterInimigosNoRaio(): Calcula inimigos ao redor da unidade no raio de ataque.
exibirAlvosDisponiveis(): Exibe os alvos possíveis para um ataque (unidades ou cidades inimigas).
selecionarEAtacarAlvo(): Permite que o jogador selecione e ataque um alvo (unidade ou cidade).
menuGerirCidades(): Exibe o menu para gerenciar cidades, onde o jogador pode criar unidades ou gerenciar terrenos.
menuGerirTerrenos(): Exibe o menu para gerenciar terrenos das cidades, incluindo a alocação e desalocação de cidadãos.
exibirCoordenadas(): Exibe as coordenadas ao redor de uma cidade para o jogador.
encerrarTurno(): Pergunta ao jogador se deseja encerrar o turno.
menuAtacarUnidade(): Gerencia o processo de ataque de uma unidade, escolhendo alvos e executando o ataque.
exibirUnidadesComCoordenadas(Player player): Exibe as unidades do jogador e suas respectivas coordenadas.

selecionarUnidade(Scanner scanner, Player player) Permite ao jogador selecionar uma unidade entre as disponíveis.
obterInimigosNoRaio(Unidade unidade, Mapa mapa, int raio, Player jogadorAtual): Retorna uma lista de inimigos (unidades ou cidades) dentro do raio de ataque da unidade.
exibirAlvosDisponiveis(List<Unidade> unidadesAlvo, List<Cidades> cidadesAlvo): Exibe uma lista de alvos disponíveis para ataque.
selecionarEAtacarAlvo(Scanner scanner, Unidade unidade, List<Unidade> unidadesAlvo, List<Cidades> cidadesAlvo): Permite ao jogador selecionar e atacar um alvo.

Herança: A classe Menus não utiliza herança diretamente, pois ela não estende outra classe.

Interação com outras classes:

A classe Menus interage com várias outras classes, como Player, Unidade, Cidades, Mapa e Terrenos, para fornecer ao jogador uma interface para movimentar unidades, atacar inimigos, gerenciar cidades e terrenos, e criar novas cidades. Embora não utilize herança, ela depende dessas classes para executar suas funções, como listar unidades, validar movimentos, atacar inimigos e alocar cidadãos em cidades, centralizando a lógica de interação do jogador com o jogo.

Classe Militares

Responsabilidade: A classe Militares representa uma unidade militar em um jogo, com a capacidade de atacar cidades e outras unidades, bem como sofrer danos. Ela herda da classe Unidade e implementa o comportamento de combate, com atributos como ataque, vida e dano de combate. Além disso, a classe permite que a unidade execute ações durante o combate e defina sua estratégia (como ficar em posição defensiva).

Atributos:
ataque (int): O valor base de ataque da unidade militar. Inicialmente é 10, e pode ser modificado conforme necessário.
vida (int): A quantidade de vida da unidade. Inicialmente é 5.
customanutencao (int): Não utilizado diretamente no código, mas pode ser interpretado como o custo de manutenção da unidade.
danocombate (int): O valor de dano base que a unidade pode causar em combate, configurado como 10 por padrão
.
Métodos:
Construtor Militares(String nome): Inicializa a unidade militar com o nome fornecido e atribui valores fixos para os atributos ataque, vida e danocombate. Também chama o construtor da classe pai Unidade para definir atributos como o custo de movimento, velocidade e outros valores básicos.
getAtaque(): Retorna o valor de ataque da unidade.
getVida(): Retorna a quantidade de vida da unidade.

attackCity(Cidades cidade): Método para atacar uma cidade. O valor de dano (definido em danocombate) é causado à cidade, que recebe esse dano.
takeDamage(int dano): Método que recebe dano e reduz a vida da unidade. Caso a vida chegue a 0 ou menos, a unidade é considerada destruída e a mensagem "foi destruída" é exibida. Caso contrário, a quantidade de vida restante é informada.
attack(Unidade inimigo): Método que permite atacar outra unidade. Se a unidade alvo for do tipo Militares, o dano causado é calculado com base no ataque da unidade atual e metade do ataque da unidade inimiga. Se o alvo não for válido (ou seja, não for uma instância de Militares), uma mensagem de erro é exibida.
receberDano(int dano): Este método parece ser redundante, pois também reduz a vida da unidade. A principal diferença é que é um método separado e pode ser usado para outros tipos de dano além dos ataques normais.
performAction(): Executa a ação da unidade. No caso dos militares, a ação padrão é ficar em posição defensiva.

Relações com Outras Classes

Herança: A classe Militares herda de Unidade, o que significa que ela tem as propriedades e comportamentos de uma unidade básica, como nome, movimento, e ações.

Associação com Cidades: A classe tem uma associação com a classe Cidades no método attackCity, onde ela pode atacar uma cidade e causar dano à mesma.

Associação com outras Unidades: No método attack, a classe Militares interage com outras instâncias da classe Unidade, permitindo que uma unidade militar ataque outra.

Classe Ouro

Responsabilidade: A classe Ouro representa o sistema de gestão de ouro no jogo. Ela permite adicionar, remover, produzir e aumentar a produção de ouro ao longo dos turnos do jogo. O ouro é uma das moedas ou recursos que os jogadores podem acumular e utilizar em várias ações dentro do jogo. A classe tem funcionalidades para modificar a quantidade de ouro disponível e para exibir seu estado atual.

Atributos:
quantidade (int): A quantidade atual de ouro no tesouro.
producaoPorTurno (int): A quantidade de ouro que é produzida a cada turno.

Métodos:

Construtor Ouro(int quantidadeInicial, int producaoPorTurno): Inicializa a classe com uma quantidade de ouro inicial e uma produção por turno.
adicionarOuro(int quantidade): Adiciona uma quantidade de ouro ao tesouro, desde que o valor seja maior que zero.

removerOuro(int quantidade): Remove uma quantidade de ouro do tesouro. Se a quantidade a ser removida for maior que a quantidade disponível no tesouro, uma exceção IllegalArgumentException é lançada.
obterQuantidade(): Retorna a quantidade atual de ouro no tesouro.
produzirOuro(): Aumenta a quantidade de ouro no tesouro de acordo com a produção definida para o turno.
toString(): Retorna uma representação textual da quantidade de ouro no tesouro.
aumentarProducao(int incremento): Aumenta a produção de ouro por turno. O valor de incremento deve ser positivo, caso contrário, é exibida uma mensagem de erro.

Relações com Outras Classes

Associação com o jogo: A classe Ouro está associada ao jogo como um recurso que pode ser usado por unidades, cidades ou jogadores para realizar diversas ações. Ela interage com o ciclo de turnos, gerando e modificando a quantidade de ouro disponível a cada iteração do jogo.

Classe Pedreira

Responsabilidade: A classe Pedreira representa um tipo de terreno no jogo que produz ouro. Ela simula uma pedreira, que é um recurso valioso que pode ser extraído para fornecer ouro aos jogadores. Este terreno tem a capacidade de gerar uma quantidade de ouro a cada turno, e também fornece informações sobre sua localização no mapa.

Atributos:
ouro (int): A quantidade atual de ouro acumulado na pedreira. Inicialmente, é 0.
producaoPorTurno (int): A quantidade de ouro gerada pela pedreira a cada turno, que é 20 por padrão.
x (int): A coordenada X da pedreira no mapa.
y (int): A coordenada Y da pedreira no mapa.

Métodos:

Construtor Pedreira(int x, int y): Inicializa a pedreira com coordenadas específicas e define o custo fixo para movimentação no terreno.
x: A coordenada X onde a pedreira está localizada.
y: A coordenada Y onde a pedreira está localizada.
getComida(): Retorna 0, pois a pedreira não gera comida.
getProducao(): Retorna 0, pois a pedreira não gera produção acumulada diretamente.
getOuro(): Retorna a quantidade de ouro atualmente acumulado na pedreira.
getProducaoPorTurno(): Retorna a quantidade de ouro gerada pela pedreira a cada turno, que é 20.
toString(): Retorna uma representação textual da pedreira no mapa, representando-a com a letra "P".
coletarOuro(): Aumenta a quantidade de ouro acumulado na pedreira, adicionando o valor de producaoPorTurno à quantidade de ouro.
getX(): Retorna a coordenada X da pedreira.

getY(): Retorna a coordenada Y da pedreira.

Relações com Outras Classes

Associação com o mapa: A Pedreira é um tipo de terreno no jogo, então ela está associada ao mapa onde é colocada. Ela fornece recursos (ouro) ao ser "coletada" durante os turnos do jogo.
Associação com Recursos: A classe Pedreira implementa a interface Recursos, que a obriga a fornecer métodos para coletar comida, produção e ouro. No caso da pedreira, ela oferece ouro como recurso e não comida ou produção.

Classe Player

Responsabilidade: A classe Player representa um jogador no contexto de um jogo de estratégia, como um jogo de conquista ou construção de império. Ela gerencia as cidades, unidades e tesouro do jogador e contém métodos para adicionar e remover unidades e cidades, pagar a manutenção das unidades, calcular o tesouro, entre outras funcionalidades.

Atributos:
nome (String): O nome do jogador.
cidades (List<Cidades>): Lista de cidades controladas pelo jogador.
unidades (List<Unidade>): Lista de unidades controladas pelo jogador.
tesouro (int): Quantidade de ouro disponível no tesouro do jogador.

Métodos:

Construtor: Player(String nome): Inicializa um jogador com um nome e listas vazias para cidades e unidades. O tesouro começa com 0.
adicionarCidade(Cidades cidade): Adiciona uma cidade à lista de cidades do jogador.
adicionarUnidade(Unidade unidade): Adiciona uma unidade à lista de unidades do jogador.
calcularTesouro(): Calcula o tesouro total do jogador, somando o ouro de todas as suas cidades.
pagarManutencao(): Paga a manutenção das unidades, descontando o custo das unidades do tesouro.
listarUnidades(): Lista as unidades do jogador, exibindo suas posições no mapa.
listarCidades(): Lista as cidades do jogador, exibindo o nome e a população.
atacarComUnidade(int indiceUnidade, Unidade unidadeAlvo): Permite que uma unidade do jogador ataque outra unidade.
addUnidades(Unidade unidade) - Método que adiciona unidades de uma lista associada a um jogador, com uma mensagem de confirmação ao adicionar uma unidade.
removerUnidade(Unidade unidade): Remove uma unidade da lista de unidades do jogador.
calcularOuroTotal(): Este método calcula o total de ouro acumulado por um jogador somando o ouro produzido por todas as suas cidades.
verificarVitoria():Este método verifica se o jogador alcançou o objetivo de 320 unidades de ouro, determinando uma condição de vitória.

Relações com Outras Classes:

Associação com Cidades: O jogador pode controlar várias cidades, e essas cidades são mantidas na lista cidades.

Associação com Unidade: O jogador pode controlar várias unidades militares, e essas unidades são mantidas na lista unidades.

Associação com Ouro: O jogador possui um tesouro (ouro) que é utilizado para pagar a manutenção das unidades e realizar outras ações no jogo.
Associação com Recursos: Embora não herde diretamente da interface Recursos, o jogador interage com as classes que implementam essa interface (como Pedreira, Floresta) para coletar recursos como ouro, comida e produção.

Classe Producao

Responsabilidade:A classe Producao gerencia a produção de recursos. Ela mantém o controle da quantidade de produção gerada por turno e da quantidade de produção utilizada

Atributos:
PRODUCAO_FIXA (int): Quantidade fixa de produção que é gerada a cada turno, definida como 10. Este valor é constante para todas as instâncias da classe.
producaoPorTurno (int): A quantidade de produção gerada a cada turno, que pode ser configurada no construtor.
producaoUtilizada (int): A quantidade de produção que já foi utilizada durante o turno.

Métodos:

Construtor: Producao(int producaoPorTurno) - Inicializa a produção com a quantidade fornecida e define a produção utilizada como 0.
reiniciarProducao(): Reinicia a produção no início de um novo turno, zerando o valor de producaoUtilizada.
usarProducao(int quantidade): Permite consumir uma quantidade específica de produção. Retorna true se a produção foi consumida com sucesso, e false se não houver produção suficiente.
getProducaoRestante(): Retorna a quantidade restante de produção que pode ser utilizada no turno.
setProducaoPorTurno(int producaoPorTurno): Atualiza a quantidade de produção gerada por turno.
toString(): Retorna uma representação textual da produção disponível, mostrando a quantidade restante.

Relações com Outras Classes:

Associação com Unidade ou Player: A produção gerida por esta classe provavelmente é usada por unidades ou jogadores no sistema para realizar ações ou construir novos elementos, embora a classe não esteja diretamente associada a essas outras classes no código fornecido.

Classe Recursos

Responsabilidade: A interface Recursos define os métodos que todas as classes que gerenciam recursos (como comida, produção e ouro) devem implementar. Ela padroniza a forma como diferentes tipos de terrenos ou entidades podem fornecer recursos no jogo. Assim, qualquer classe que implemente esta interface estará comprometida em fornecer métodos para acessar diferentes tipos de recursos.

Métodos:

int getComida(): Retorna a quantidade de comida disponível.
int getProducao(): Retorna a quantidade de produção acumulada (não necessariamente por turno).
int getOuro(): Retorna a quantidade de ouro disponível.
int getProducaoPorTurno(): Retorna a quantidade de produção gerada a cada turno.

Relações com Outras Classes:

Associação com Classes de Recursos: Qualquer classe que gerencia recursos, como Floresta, Pedreira, ou outras entidades que forneçam comida, ouro ou produção, deve implementar esta interface. Isso garante que essas classes tenham métodos padronizados para fornecer recursos ao jogo.

Polimorfismo: Como é uma interface, ela permite que diferentes tipos de terrenos ou entidades sejam tratados de forma polimórfica. Por exemplo, uma classe pode usar a interface Recursos para acessar dados de qualquer objeto (como Floresta ou Pedreira) sem se preocupar com o tipo específico de terreno ou entidade.

Classe TerrenoCidade

Responsabilidade: A classe TerrenoCidade representa um tipo específico de terreno no jogo onde uma cidade está localizada. Ela herda de Terrenos e associa uma Cidade a um conjunto de coordenadas no mapa. O principal objetivo desta classe é vincular uma cidade a um terreno específico, permitindo que o jogo associe cidades a posições no mapa e forneça informações sobre a cidade, como sua localização.

Atributos:
cidade (Cidades): A cidade que está localizada nesse terreno específico.
x (int): A coordenada X da cidade no mapa.
y (int): A coordenada Y da cidade no mapa.

Métodos:

TerrenoCidade(Cidades cidade, int x, int y): Construtor que inicializa o terreno com a cidade associada e suas coordenadas no mapa.
toString(): Retorna uma representação textual da cidade no mapa, representando-a com a letra "C".

getX(): Retorna a coordenada X onde a cidade está localizada.
getY(): Retorna a coordenada Y onde a cidade está localizada.
getCidade(): Retorna a cidade associada ao terreno.

Relações com Outras Classes:

Herança de Terrenos: A classe TerrenoCidade herda de Terrenos, o que a faz parte da hierarquia de terrenos no jogo. Ela pode ter características comuns a outros tipos de terrenos, como custo de movimentação, além de ter seus próprios atributos e métodos específicos para representar a cidade.

Associação com Cidades: A classe tem uma forte associação com a classe Cidades, já que ela contém um objeto do tipo Cidades (representando a cidade localizada nesse terreno). A classe Cidades define o comportamento e os atributos da cidade em questão, enquanto TerrenoCidade lida com a localização e representação da cidade no mapa.

Uso no Jogo: A classe TerrenoCidade pode ser usada em conjunto com o sistema de mapa do jogo, permitindo que cada cidade seja associada a uma posição específica no terreno. Ela também pode interagir com outros tipos de terreno para determinar como as cidades e outras entidades se comportam no ambiente do jogo.

Classe TerrenoTrabalhado

Responsabilidade: A classe TerrenoTrabalhado representa um terreno que foi trabalhado por trabalhadores, geralmente associado a uma cidade responsável pela sua gestão. Este tipo de terreno tem um terreno original (um tipo básico de terreno) e é manipulado por uma certa quantidade de trabalhadores, com a responsabilidade sendo atribuída a uma cidade específica. A classe permite modelar terrenos que sofreram modificações, como colheitas ou outros tipos de produção que necessitam de trabalhadores.

Atributos:
terrenoOriginal (Terrenos): O terreno básico de onde o terreno trabalhado foi derivado. Este é o terreno original que sofreu alguma transformação.
quantidadeTrabalhadores (int): A quantidade de trabalhadores que estão atualmente operando nesse terreno.
cidadeResponsavel (String): O nome da cidade responsável por esse terreno trabalhado.
x (int): A coordenada X onde o terreno trabalhado está localizado no mapa.
y (int): A coordenada Y onde o terreno trabalhado está localizado no mapa.

Métodos:
TerrenoTrabalhado(Terrenos terrenoOriginal, String cidadeResponsavel, int quantidadeTrabalhadores, int x, int y): Construtor que inicializa o terreno trabalhado com o terreno original, cidade responsável, número de trabalhadores e as coordenadas X e Y.

getTerrenoOriginal(): Retorna o terreno original (antes de ser trabalhado).
getCidadeResponsavel(): Retorna o nome da cidade responsável pelo trabalho nesse terreno.
getQuantidadeTrabalhadores(): Retorna a quantidade de trabalhadores alocados no terreno.
toString(): Retorna uma representação textual do terreno trabalhado, incluindo o terreno original, o número de trabalhadores e a cidade responsável.
incrementarTrabalhador(): Incrementa a quantidade de trabalhadores no terreno trabalhado.
getX(): Retorna a coordenada X do terreno trabalhado.
getY(): Retorna a coordenada Y do terreno trabalhado.

Relações com Outras Classes:

Herança de Terrenos:A classe TerrenoTrabalhado herda da classe Terrenos, significando que ela é um tipo especializado de terreno. Isso permite que ela tenha todas as características gerais de um terreno, mas com a adição de trabalhadores e cidade responsável.

Associação com Terrenos: A classe tem uma associação com a classe Terrenos, já que ela possui um objeto do tipo Terrenos como seu atributo terrenoOriginal. Isso permite que TerrenoTrabalhado seja uma versão modificada de um terreno básico.

Associação com Cidades: A classe TerrenoTrabalhado está associada a uma cidade (indiretamente) por meio do atributo cidadeResponsavel. Isso implica que o terreno trabalhado é controlado por uma cidade, e todas as mudanças no terreno dependem da cidade responsável por ele.

Uso no Jogo: Essa classe pode ser usada no contexto de jogos de estratégia onde diferentes tipos de terrenos (como terras cultivadas ou minas) podem ser trabalhados por trabalhadores de uma cidade para gerar recursos. Ela pode interagir com outras classes de Terrenos, Cidades e Trabalhadores para simular o progresso e a utilização de recursos em diferentes partes do mapa do jogo.

Classe TerrenoUnidade

Responsabilidade: A classe TerrenoUnidade representa um terreno no jogo que possui uma unidade (como um militar, construtor ou colono) alocada nele. Esse tipo de terreno permite associar unidades a posições específicas no mapa, facilitando a movimentação e as interações entre unidades e os recursos ou territórios que elas controlam.

Atributos:
unidade (Unidade): A unidade associada a este terreno. A unidade pode ser um tipo específico, como um militar, um construtor ou um colono.
x (int): A coordenada X onde a unidade está localizada no mapa.
y (int): A coordenada Y onde a unidade está localizada no mapa.

Métodos:
TerrenoUnidade(Unidade unidade, int x, int y): Construtor que inicializa o terreno com uma unidade específica e suas coordenadas no mapa.
getUnidade(): Retorna a unidade associada a este terreno.
setUnidade(Unidade unidade): Permite definir ou atualizar a unidade associada a este terreno.
toString(): Retorna uma representação textual do terreno com base no tipo da unidade. Dependendo do tipo de unidade, o método retorna:
"M" para unidades do tipo Militares.
"B" para unidades do tipo Construtores.
"H" para unidades do tipo Colonistas.
"?" para unidades de tipos desconhecidos ou não especificados.
getX(): Retorna a coordenada X do terreno onde a unidade está localizada.
getY(): Retorna a coordenada Y do terreno onde a unidade está localizada.

Relações com Outras Classes:

Herança de Terrenos:A classe TerrenoUnidade herda de Terrenos, permitindo que ela seja tratada como um tipo específico de terreno. Isso significa que ela herda todas as características gerais dos terrenos, mas com a adição de uma unidade associada a esse terreno.

Associação com Unidade: A classe possui um atributo unidade, que representa a unidade alocada nesse terreno. A unidade pode ser de qualquer tipo específico, como Militares, Construtores ou Colonistas, o que permite que esse terreno tenha uma unidade operando nele.
Associação com Tipos de Unidade (Militares, Construtores, Colonos):
A classe verifica, no método toString(), qual o tipo de unidade associada ao terreno e retorna um caractere representativo ("M", "B", "H") de acordo com o tipo da unidade. Isso implica uma associação com as subclasses de Unidade, como Militares, Construtores e Colonistas.

Uso no Jogo: A classe TerrenoUnidade pode ser usada para mapear a posição e os tipos de unidades no jogo. Pode ser essencial para controlar a movimentação de unidades no mapa, assim como para gerenciar as ações que essas unidades podem realizar no terreno (por exemplo, um militar se movendo, um construtor construindo algo, ou um colono colonizando um novo território).

Classe TerrenoVazio

Responsabilidade: A classe TerrenoVazio representa um terreno vazio no mapa, sem unidades ou construções associadas. Este tipo de terreno pode ser utilizado para representar áreas não desenvolvidas, onde nada está presente, mas pode ser explorado, conquistado ou alterado no futuro. A classe é parte da hierarquia de Terrenos, podendo ser útil para mapear as áreas disponíveis no jogo.

Atributos:
x (int): A coordenada X no mapa onde o terreno vazio está localizado.
y (int): A coordenada Y no mapa onde o terreno vazio está localizado.

Métodos:
TerrenoVazio(int x, int y): Construtor que inicializa as coordenadas x e y do terreno vazio, com um custo fixo associado de 1, herdado da classe Terrenos.
getX(): Retorna a coordenada X do terreno vazio.
getY(): Retorna a coordenada Y do terreno vazio.
toString(): Retorna uma representação textual do terreno vazio. Neste caso, retorna o caractere "X", que representa visualmente o terreno vazio no mapa.

Relações com Outras Classes:

Herança de Terrenos: A classe TerrenoVazio herda da classe Terrenos, o que significa que ela é tratada como um tipo específico de terreno no jogo. Ela compartilha as propriedades e comportamentos comuns de todos os terrenos, mas com a diferença de que este é um terreno sem unidades ou construções.

Associação com o Mapa: O TerrenoVazio representa um espaço em branco ou um local livre no mapa. Ele é utilizado no contexto de um sistema de mapa para mostrar onde não há nada presente. Ele pode ser convertido em um TerrenoTrabalhado, TerrenoUnidade, Agua, Floresta, Pedreira ou TerrenoCidade dependendo das ações ou do desenvolvimento do jogo

Uso no Jogo: Este tipo de terreno é utilizado para determinar onde o jogador pode construir novas cidades, movimentar unidades, ou executar outras ações de jogo, como exploração ou colonização de novos territórios. O TerrenoVazio também serve para mapear as áreas que ainda não foram alteradas ou ocupadas, ou seja, ele pode ser o ponto de partida para o crescimento do jogo.

Classe Terrenos

Responsabilidade: A classe Terrenos é uma classe abstrata que serve como a base para todos os tipos de terrenos no jogo. Ela define características comuns a todos os terrenos, como o custo de movimento e a possibilidade de ter uma unidade associada a ele. Terrenos podem representar diferentes tipos de áreas no mapa, como cidades, pedreiras, áreas vazias, etc. Além disso, ela define alguns métodos fundamentais que devem ser implementados por classes filhas, como a obtenção das coordenadas e a representação visual do terreno.

Atributos:
custoMovimento (int): O custo de movimento associado a este terreno. Este atributo é importante para determinar quanto custa para uma unidade mover-se sobre este terreno.
unidade (Unidade): A unidade que, opcionalmente, pode estar presente neste terreno. O valor de unidade pode ser null se o terreno estiver vazio ou não ocupado por nenhuma unidade.

Métodos:
Terrenos(int custoMovimento): Construtor que inicializa o custo de movimento do terreno. Este custo é fixo para o terreno, mas pode ser alterado em subclasses.
getCustoMovimento(): Retorna o custo de movimento do terreno.
getX() (método abstrato): Deve ser implementado pelas subclasses para retornar a coordenada X do terreno no mapa.
getY() (método abstrato): Deve ser implementado pelas subclasses para retornar a coordenada Y do terreno no mapa.
toString() (método abstrato): Método que deve ser implementado pelas subclasses para fornecer uma representação textual do terreno (por exemplo, "P" para pedreiras, "C" para cidades, "X" para terrenos vazios, etc.).
getUnidade(): Retorna a unidade associada ao terreno (se houver).
temUnidade(): Verifica se o terreno contém uma unidade. Isso é feito verificando se o terreno é uma instância de TerrenoUnidade, o que indica que há uma unidade sobre o terreno.

Relações com Outras Classes:

Herança e Polimorfismo: A classe Terrenos é uma classe abstrata que será estendida por outras classes concretas, como TerrenoVazio, TerrenoCidade, TerrenoUnidade, TerrenoTrabalhado, entre outras. Essas subclasses implementam os métodos abstratos de Terrenos, permitindo que o comportamento de cada tipo de terreno seja personalizado.

Associação com Unidade: A classe Terrenos tem um atributo do tipo Unidade que indica se uma unidade está ou não presente no terreno. O método temUnidade() verifica se o terreno possui uma unidade associada. Essa relação é importante para movimentação de unidades, combate, e outras ações que envolvem unidades no mapa.
Associação com Subclasses: As subclasses de Terrenos são responsáveis por fornecer comportamentos específicos para diferentes tipos de terrenos. Por exemplo, um TerrenoUnidade pode ter uma unidade associada a ele, enquanto um TerrenoVazio não tem nenhuma unidade. Essas subclasses implementam os métodos getX(), getY(), e toString(), para refletir a lógica e a aparência de cada tipo de terreno no jogo.

Uso no Jogo: A classe Terrenos serve como base para modelar os diferentes tipos de terreno no jogo, e é essencial para o sistema de movimentação, exploração e combate das unidades. O custo de movimento, por exemplo, influencia as decisões estratégicas do jogador, enquanto o método temUnidade() determina se uma unidade pode se mover para um determinado terreno.

Classe Unidade

Responsabilidade: A classe Unidade é uma classe abstrata que define a estrutura básica e o comportamento de unidades no jogo, como militares, colonos e construtores. Ela controla atributos e ações relacionados à movimentação, vida, ataque, e produção de unidades, além de fornecer métodos comuns para subclasses especializadas (como Militares, Colonos, e Construtores). A Unidade pode atacar, se mover e ser atacada, com funcionalidades adicionais para gestão de pontos de movimento, vida, e custo de produção.

Atributos:

nome (String): Nome da unidade.
posicaoX (int): Posição da unidade no eixo X do mapa.
posicaoY (int): Posição da unidade no eixo Y do mapa.
pontosMovimentos (int): Pontos de movimento disponíveis para a unidade, usados para mover-se no mapa.
maxpontosmovimento (int): O número máximo de pontos de movimento que a unidade pode ter.
custo (int): O custo para criar a unidade.
vida (int): A quantidade de vida da unidade.
ataque (int): O valor de ataque da unidade.

Métodos:

Construtor:Unidade(String nome, int maxpontosmovimento, int custo, int custoProducao): Inicializa a unidade com um nome, pontos de movimento, custo de criação e custo de produção.
getCustoProducao(): Retorna o custo de produção da unidade.
criarUnidade(String tipo): Método para criar diferentes tipos de unidades como Militar, Colono, e Construtor com base no tipo fornecido.
mover(int novaPosicaoX, int novaPosicaoY, Terrenos terrenoDestino): Move a unidade para uma nova posição no mapa, considerando o custo de movimento do terreno.
setVida(int vida): Define a vida da unidade.
resetMovementPoints(): Reseta os pontos de movimento para o valor máximo.
move(int novaPosicaoX, int novaPosicaoY): Move a unidade para a posição especificada no mapa, alterando a coordenada.
move(int custoterra): Utiliza os pontos de movimento com base no custo do terreno
movePara(int novaPosicaoX, int novaPosicaoY, int custoMovimento): Move a unidade considerando o custo de movimento especificado.
attack(Unidade inimigo): Ataca outra unidade, causando dano baseado no valor de ataque da unidade atual e da unidade inimiga.
attackCity(Cidades cidade): Ataca uma cidade, causando dano fixo à cidade.
takeDamage(int dano): Reduz a vida da unidade quando ela sofre dano. Se a vida atingir zero, a unidade é destruída.
toString(): Retorna uma representação textual da unidade, mostrando seu nome e posição no mapa.

reduzirPontosMovimento(int custo): Reduz os pontos de movimento da unidade de acordo com o custo.
resetarPontosMovimento(): Reseta os pontos de movimento para o valor máximo.

Relações com Outras Classes:

Associação com Terrenos: A unidade interage com o mapa e os terrenos através da movimentação. Os terrenos influenciam o custo de movimento, e as unidades precisam conhecer o tipo de terreno para saber como se movimentar.

Associação com Cidades: As unidades podem atacar cidades, e isso influencia o progresso do jogo. As unidades também podem ser criadas a partir de cidades ou outras fontes no jogo.

Associação com Subclasses: Militares, Colonos e Construtores são subclasses da classe Unidade, cada uma com características e comportamentos específicos. Essas subclasses utilizam e estendem as funcionalidades definidas pela classe base Unidade.

Associação com o Sistema de Produção: O custo de produção de uma unidade está diretamente relacionado à economia do jogo, e a classe Unidade controla este aspecto fundamental para criar unidades novas no jogo.

3.3.Aplicação dos Princípios de POO e SOLID

Encapsulamento

No contexto do código apresentado, o encapsulamento é aplicado adequadamente ao utilizar modificadores de acesso, como private e public, para proteger os dados internos das classes e expor apenas os métodos necessários. Por exemplo, a classe Unidade (Fig.2.1 e 2.2) pode ter seus atributos como a posição e os pontos de movimento definidos como privados (private), sendo acessados e modificados apenas por métodos públicos específicos, como getPosicaoX(), getPosicaoY() e setPosicaoX(), setPosicaoY(). Isso assegura que o acesso e a modificação de dados sensíveis sejam controlados e realizados de maneira segura.

```java
public String getNome() { return nome; }
public int getPosicaoX() { return posicaoX; }
public int getPosicaoY() { return posicaoY; }
public int getCusto() { return custo; }
public int getPontosMovimentos() { return pontosMovimentos; }

public void setPosicao(int x, int y) {
    this.posicaoX = x;
    this.posicaoY = y;
}
```

Fig.2.1 -Exemplo de encapsulamento, métodos gets e sets

```java
private String nome;    Field nome can be final
private int posicaoX;
private int posicaoY;
private int pontosMovimentos;
private int maxpontosmovimento;     Field maxpontosmovimento can be fir
private int custo;     Field custo can be final
private int vida;
private int ataque;
private int custoProducao;    Field custoProducao can be final
```

Fig.2.2. - Exemplo de Encapsulamento, atributos

Além disso, o código segue o Princípio da Responsabilidade Única (SRP), do SOLID, pois as classes são responsáveis por tarefas específicas, como a classe Unidade gerenciar as unidades do jogo e a classe Menus tratar apenas da lógica de interação do jogador com os menus. Essa organização contribui para um design mais modular e facilita a manutenção, pois alterações em uma classe não afetam diretamente outras partes do sistema.

Abstração

No código apresentado, a abstração é utilizada para ocultar os detalhes específicos e focar nos aspectos essenciais dos objetos. Por exemplo, a classe Terrenos é abstrata, com subclasses como Floresta, Pedreira, e outros tipos de terrenos, cada um implementando suas próprias funcionalidades específicas, enquanto a classe Terrenos (Fig.3) define um contrato geral. Essa abordagem permite que diferentes tipos de terrenos sejam manipulados sem que o código que utiliza o conceito de terreno precise se preocupar com as implementações específicas.

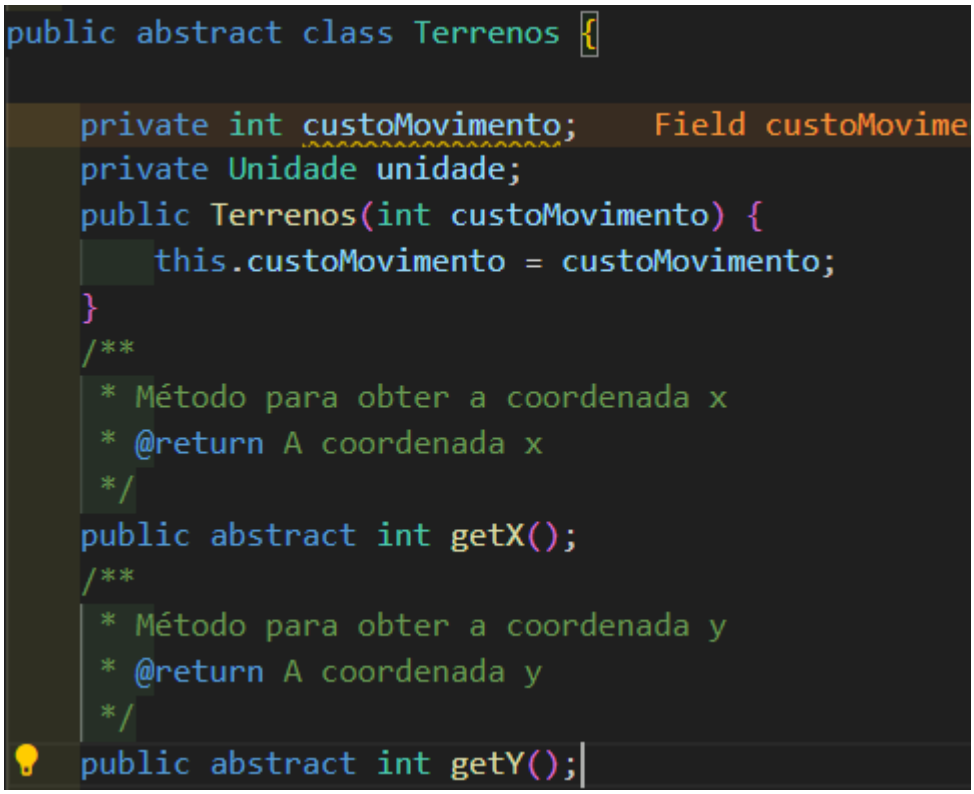


Fig.3 - Exemplo de classe abstrata, com métodos getX() e getY(), também eles abstratos.

A classe abstrata que representa unidades poderia ser outra boa aplicação de abstração. Unidade pode ser uma classe abstrata, com subclasses específicas como Militares, Colono, etc. Isso permite que cada tipo de unidade tenha comportamentos específicos, como atacar, mover, ou construir, sem que o código de manipulação da unidade precise saber os detalhes de cada tipo.

Em relação ao Princípio Aberto/Fechado (OCP), o código segue este princípio, pois é possível adicionar novas funcionalidades (como novos tipos de unidades, terrenos ou interações) sem modificar o código existente. Ao estender classes abstratas ou interfaces, é possível adicionar novas classes sem alterar o funcionamento das classes já implementadas, promovendo a extensibilidade sem modificação do código existente. Isso facilita a manutenção e evolução do sistema de forma controlada e sem riscos de afetar a funcionalidade existente.

Herança

No código, a herança é usada para promover a reutilização de código e facilitar a organização hierárquica, seguindo o conceito de "é um". Por exemplo, temos uma classe base Unidade, que define comportamentos gerais para diferentes tipos de unidades, subclasses como Militares, Colonos, etc., podem herdar esses comportamentos e adicionarem funcionalidades específicas. Isso possibilita uma estrutura hierárquica que promove a manutenção e a extensão do sistema sem duplicação de código.

Quanto ao Princípio da Substituição de Liskov (LSP), o código assegura que os objetos das subclasses podem ser utilizados no lugar de objetos da superclasse sem afetar o comportamento do sistema. Ou seja, qualquer instância de Militares pode ser usada onde uma

Unidade é esperada, e o sistema continuará funcionando corretamente, sem quebra de comportamento.

```java
/**
 * Método para atacar uma cidade
 * @param cidade A cidade a ser atacada.
 */
public void attackCity(Cidades cidade) {
    if (cidade != null) {
        System.out.println(this.nome + " está atacando a cidade " + cidade.getNome());
        int dano = 20;
        cidade.takeDamage(dano);
    } else {
        System.out.println(x:"A cidade não é válida.");
    }
}
```

Fig. 4.1- Método attackCity, método que faz um cidade levar dano

```java
/**
 * Método para atacar uma cidade
 * @param cidade A cidade a ser atacada.
 */
@Override
public void attackCity(Cidades cidade) {
    System.out.println(getNome() + " está a atacar a cidade " + cidade.getNome());
    int dano = danocombate;
    cidade.takeDamage(dano);
}
```

Fig. 4.2-Método attackCity com override, método que diz qual das cidades o militar está a atacar

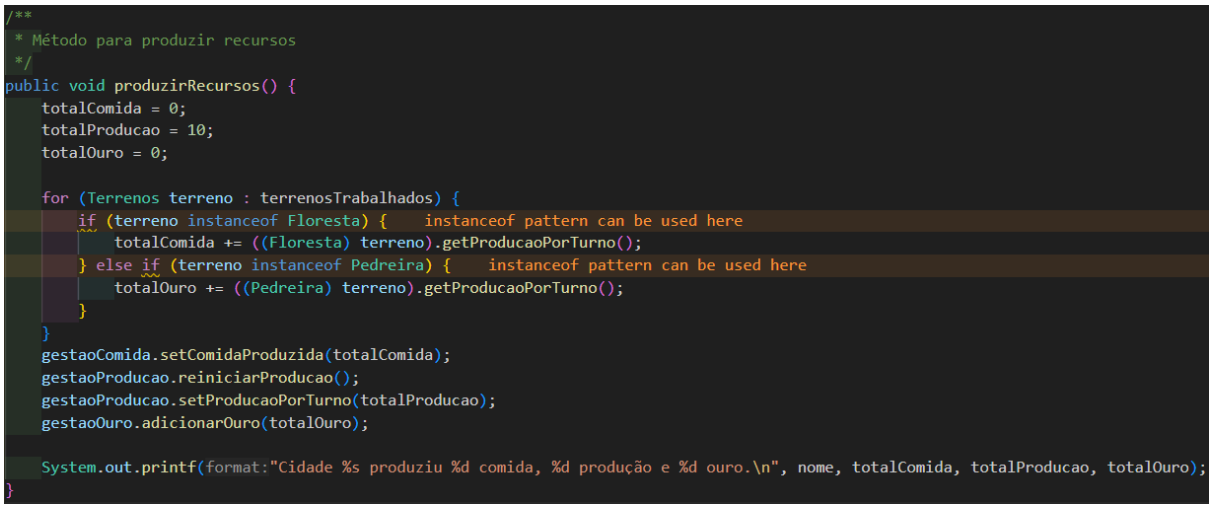
Por exemplo, se a classe Unidade tiver um método attackCity() e as subclasses Militares ou Colonos implementarem esse método de maneira específica, elas ainda mantêm o comportamento consistente, como esperado pela classe Unidade. Isso garante que a substituição de um objeto de uma subclasse por um objeto da superclasse ou vice-versa não cause falhas no funcionamento do sistema, atendendo assim ao LSP.

Em termos de uso apropriado da herança, o código deve evitar usar herança em situações onde o comportamento de uma subclasse não se alinha com o da superclasse, pois isso pode violar o LSP. Portanto, é importante garantir que as subclasses apenas estendam as funcionalidades da superclasse de forma coerente e consistente, sem alterar ou quebrar as expectativas de comportamento.

Polimorfismo

O polimorfismo no método produzirRecursos é exemplificado ao iterar sobre os terrenos trabalhados, que são tratados de forma genérica como objetos da classe base Terrenos, mas que, em tempo de execução, revelam seu comportamento específico dependendo do tipo concreto, como Floresta ou Pedreira. Esse comportamento permite que, para cada terreno, o método identifique dinamicamente o tipo apropriado e chame o método getProducaoPorTurno

definido especificamente em cada classe derivada. Por exemplo, quando o terreno é uma instância de Floresta, a produção de comida é calculada e adicionada ao total de comida, enquanto, no caso de uma Pedreira, o método calcula e adiciona ao total de ouro produzido. Essa abordagem polimórfica promove extensibilidade, permitindo a inclusão de novos tipos de terrenos com comportamentos distintos, sem a necessidade de modificar o código existente, mantendo o sistema flexível e aderente aos princípios do design orientado a objetos.



```java
/**
 * Método para produzir recursos
 */
public void produzirRecursos() {
    totalComida = 0;
    totalProducao = 10;
    totalOuro = 0;

    for (Terrenos terreno : terrenosTrabalhados) {
        if (terreno instanceof Floresta) {    instanceof pattern can be used here
            totalComida += ((Floresta) terreno).getProducaoPorTurno();
        } else if (terreno instanceof Pedreira) {    instanceof pattern can be used here
            totalOuro += ((Pedreira) terreno).getProducaoPorTurno();
        }
    }
    gestaoComida.setComidaProduzida(totalComida);
    gestaoProducao.reiniciarProducao();
    gestaoProducao.setProducaoPorTurno(totalProducao);
    gestaoOuro.adicionarOuro(totalOuro);

    System.out.printf(format:"Cidade %s produziu %d comida, %d produção e %d ouro.\n", nome, totalComida, totalProducao, totalOuro);
}
```

Fig.5 - Exemplo do uso de polimorfismo

O Princípio da Segregação de Interfaces (ISP) também está presente no método produzirRecursos devido ao uso de interfaces ou contratos específicos que cada tipo de terreno implementa. Embora o código fornecido não mostre explicitamente a definição de interfaces, o conceito pode ser deduzido da forma como os métodos, como getProducaoPorTurno, são invocados diretamente nos objetos de classes específicas, como Floresta e Pedreira. Esse princípio afirma que é melhor ter várias interfaces pequenas e específicas do que uma única interface grande e genérica que force classes a implementar métodos que não fazem sentido para elas. No contexto de Terrenos, cada classe concreta (como Floresta ou Pedreira) foca em um comportamento específico relevante para sua função.

Coesão

A coesão refere-se à centralização das responsabilidades de uma classe. Quanto mais coesa uma classe, mais clara e específica será sua responsabilidade, evitando sobrecarga e multifuncionalidade. Segundo o Princípio da Responsabilidade Única (SRP), uma classe deve ter uma única razão para mudar, ou seja, deve estar focada em cumprir uma única tarefa ou finalidade.

Por exemplo, a classe Cidades no código apresentado pode ser avaliada quanto à sua coesão analisando se suas responsabilidades são centralizadas na gestão de uma cidade. Os métodos relacionados à produção de recursos, crescimento da população e gerenciamento de unidades devem estar diretamente conectados ao propósito de modelar uma cidade em um sistema de jogo. Caso métodos ou atributos desconectados dessa responsabilidade sejam incluídos, a coesão seria comprometida.

Acoplamento

O acoplamento representa o grau de dependência entre classes em um sistema, sendo desejável manter essas dependências mínimas para garantir maior modularidade e facilitar a manutenção do código. O Princípio da Inversão de Dependência (DIP), parte dos conceitos SOLID, recomenda que tanto módulos de alto nível quanto de baixo nível dependam de abstrações, e não de implementações concretas. Isso é visível na classe Cidades, que utiliza a classe base ou interface Terrenos em vez de depender diretamente de subclasses como Floresta e Pedreira.

Ao depender apenas de abstrações, Cidades pode interagir com qualquer tipo de terreno que implemente os métodos esperados, como getProducaoPorTurno(),Fig.5, sem precisar conhecer seus detalhes. Essa abordagem reduz a rigidez do código, facilitando a adição de novos tipos de terreno sem necessidade de alterar a lógica existente. Assim, o sistema ganha em flexibilidade e extensibilidade, permitindo alterações ou expansões com impacto mínimo em outras partes do software. A aplicação do DIP neste caso garante um design que promove baixo acoplamento e alta modularidade.

Reutilização de Código e Manutenibilidade

A reutilização de código e a manutenibilidade são aspectos fundamentais na engenharia de software, garantindo que o sistema seja extensível e sustentável a longo prazo. O Princípio Aberto/Fechado (OCP), integrante do SOLID, enfatiza que classes e módulos devem estar abertos para extensão, mas fechados para modificação, o que permite adicionar novas funcionalidades sem comprometer o código existente. Esse princípio é evidente em abordagens como a utilização de polimorfismo, onde o comportamento pode ser estendido através de subclasses ou interfaces sem alterar a classe principal.

Já o Princípio da Inversão de Dependência (DIP) contribui para a modularidade ao promover o uso de abstrações em vez de dependências diretas de implementações concretas. Isso facilita a reutilização do código, permitindo que diferentes componentes trabalhem de forma independente e que novas funcionalidades sejam integradas sem grandes ajustes nas dependências existentes.

No contexto do código apresentado, essas práticas são refletidas na estrutura modular das classes, como Terrenos, que serve como base para diversas implementações específicas (Floresta, Pedreira, etc.), permitindo que novos tipos de terrenos sejam facilmente adicionados. Além disso, a nomenclatura clara dos métodos, como getPlayer(), takeDamage() e adicionarUnidade(Unidade unidade), e os comentários explicativos auxiliam na compreensão e na manutenção do sistema. A combinação de uma estrutura lógica clara, modularidade, e documentação contribui para um código mais reutilizável, extensível e fácil de manter.

```java
/**
 * Método para atacar uma cidade
 * @param dano O dano a ser causado
 */
public void takeDamage(int dano) {
    this.vida -= dano;
    if (this.vida <= 0) {
        System.out.println("A cidade " + nome + " foi destruída.");
    } else {
        System.out.println("A cidade " + nome + " sofreu dano. Vida restante: " + this.vida);
    }
}
// Getters
public Player getPlayer() {
    return player;
}
/**
 * Método para adicionar uma unidade
 * @param unidade A unidade a ser adicionada
 */
public void adicionarUnidade(Unidade unidade) {
    unidades.add(unidade);
}
```

Fig.6 - Exemplo de modularidade e facilitar a reutilização através da explicação dos métodos

Tratamento de Exceções

O tratamento adequado de exceções é um aspecto crucial para a robustez de qualquer sistema, pois garante que situações anômalas sejam gerenciadas de forma controlada e sem comprometer a estabilidade geral da aplicação. Ao adotar o Princípio da Responsabilidade Única (SRP), é possível isolar o tratamento de erros da lógica de negócio principal, o que melhora a clareza e a manutenibilidade do código, além de permitir que cada parte do sistema tenha uma responsabilidade bem definida. No exemplo da criação de uma cidade(Fig.6), ao utilizar blocos try-catch, conseguimos capturar erros específicos, como quando a posição da cidade não é adequada ou quando há um conflito com a proximidade de outra cidade, evitando que o sistema quebre inesperadamente. O uso de exceções personalizadas, como a IllegalArgumentException, torna o código mais legível e facilita a identificação de problemas específicos, permitindo que o desenvolvedor entenda rapidamente a origem do erro e forneça uma solução mais eficaz. Essa abordagem também está alinhada com a separação de responsabilidades, uma vez que o código de manipulação de exceções se concentra unicamente em tratar os erros, enquanto a lógica de negócio continua a ser responsável pela funcionalidade principal do sistema. Essa distinção entre responsabilidades não só melhora a legibilidade do código, mas também facilita a evolução e a manutenção do software ao longo do tempo, permitindo que novos desenvolvedores compreendam rapidamente como os erros são tratados sem interferir diretamente na lógica de negócio.

```java
/**
 * Cria uma nova cidade no mapa, se a posição for válida.
 *
 * @param x      coordenada X da cidade.
 * @param y      coordenada Y da cidade.
 * @param player jogador que possui a cidade.
 * @return true se a cidade foi criada com sucesso, false caso contrário.
 */
public boolean criarCidade(int x, int y, Player player) {
    try {
        if (!(mapa[y][x] instanceof TerrenoVazio)) {
            throw new IllegalArgumentException(s:"O terreno não é adequado para uma cidade.");
        }
        if (!posicaoLivre(x, y)) {
            throw new IllegalArgumentException(s:"A posição está muito próxima de outra cidade.");
        }

        Cidades novaCidade = new Cidades(populacao:10, ouroInicial:0, producaoPorTurno:10, x, y, this, player);
        posicoesCidades.add(new int[]{x, y});
        listcidades.add(novaCidade);
        player.adicionarCidade(novaCidade);

        mapa[y][x] = new TerrenoCidade(novaCidade, x, y);

        System.out.println("Cidade " + novaCidade.getNome() + " criada em (" + x + ", " + y + ")");
        return true;

    } catch (IllegalArgumentException e) {
        System.out.println("Erro ao criar cidade: " + e.getMessage());
        return false;
    } catch (Exception e) {
        System.out.println("Ocorreu um erro inesperado: " + e.getMessage());
        return false;
    }
}
```

4. Decisões de Projeto e Justificativas

4.1. Decisões Importantes Tomadas

A primeira decisão relevante no desenvolvimento do jogo foi a definição da estrutura do mapa, que foi estabelecida utilizando um duplo array (matriz bidimensional). A escolha por esse modelo surgiu pela clareza e modularidade que ele proporciona. O duplo array foi ideal para representar o mapa do jogo, já que facilita a localização e modificação das células do mapa, onde cada célula contém informações sobre os diferentes tipos de terreno (como pedreiras, florestas, etc.). Isso não só torna o acesso rápido e direto às células (acessíveis via mapa[X][Y]), como também permite a expansão do mapa no futuro, com a adição de novos tipos de terreno ou elementos interativos. Além disso, essa estrutura simplifica a manipulação de interações específicas que cada terreno pode ter, como bloqueios para unidades ou aumentos de produção de recursos.

A escolha de armazenar os players em um ArrayList se baseou na necessidade de flexibilidade e simplicidade na gestão dinâmica dos jogadores durante o jogo. O uso de ArrayList permite adicionar, remover ou alterar a posição de jogadores de forma eficiente e direta, sem que seja necessário reestruturar os dados, o que é especialmente importante quando o número de jogadores pode mudar ao longo da partida. A alternância entre turnos de jogadores se torna mais simples, bastando apenas ajustar o índice do jogador atual no ArrayList. Isso

também facilita a expansão do número de jogadores, permitindo que o sistema seja escalável, sem a necessidade de refatorações substanciais no futuro.

Da mesma forma, as cidades e unidades foram armazenadas em ArrayLists, o que garante uma gestão eficiente e dinâmica dessas entidades no decorrer do jogo. A utilização dessa estrutura de dados permite que as cidades e unidades sejam manipuladas com facilidade, como a adição ou remoção delas ao longo da partida, sem a complexidade de sistemas de dados rígidos. Isso também facilita a iteração sobre essas entidades, permitindo que ações sejam realizadas sobre elas rapidamente, como no caso de alterações durante os turnos.

Uma das primeiras funcionalidades implementadas foi a criação da cidade inicial. Decidimos que, ao iniciar o jogo, o jogador seria solicitado a escolher uma posição vazia no mapa para fundar sua primeira cidade. Essa decisão envolveu a criação de um menu interativo que exibia o mapa ao jogador, permitindo-lhe escolher estrategicamente onde posicionar a cidade. A interação com o mapa e a escolha de sua localização permite ao jogador se envolver ativamente no jogo desde o início, gerando um impacto imediato na estratégia da partida, pois o local escolhido pode afetar diretamente o desenvolvimento futuro da cidade e da civilização.

4.2. Justificativas das Decisões

Cada uma dessas decisões foi tomada com base na busca pela eficiência, flexibilidade e facilidade de manutenção do sistema ao longo do desenvolvimento e da evolução do jogo. O uso de estruturas como o duplo array e os ArrayLists permite uma gestão simplificada dos dados e uma manipulação eficiente das entidades do jogo, além de proporcionar escalabilidade ao sistema.

A escolha do duplo array para o mapa foi motivada pela simplicidade e pela possibilidade de expansão do mundo do jogo sem grandes reconfigurações. Isso é especialmente importante para um jogo que pode se expandir no futuro com novos tipos de terreno ou novos elementos interativos, garantindo que a estrutura de dados inicial seja robusta o suficiente para suportar mudanças sem impacto significativo no desempenho ou na lógica do código.

Ao armazenar os players, cidades e unidades em ArrayLists, conseguimos garantir uma estrutura flexível e adaptável. Isso permite que o jogo seja facilmente modificado e ajustado, seja para aumentar o número de jogadores ou para ajustar a quantidade de cidades e unidades em tempo real. O uso dessas coleções torna a implementação de funções adicionais, como remoção de entidades ou movimentação das mesmas, mais ágil e direta.

Por fim, a implementação da criação da cidade inicial foi decisiva para envolver o jogador logo no início da partida. Dar ao jogador o controle sobre a escolha do local da cidade oferece um fator estratégico importante, fazendo com que cada partida tenha um começo único, o que aumenta a rejogabilidade do jogo e a imersão do jogador. Essa abordagem também contribui para uma experiência mais dinâmica e envolvente desde o primeiro momento, assegurando que o jogador se sinta parte ativa do processo de construção de sua civilização.

4.3. Desafios Enfrentados

Para resolver esse problema, optamos por usar um tipo genérico de classe, ou seja, um objeto do tipo Object, em vez de um tipo fixo, para armazenar as instâncias na matriz do mapa. Isso nos permitiu substituir dinamicamente os objetos presentes em uma célula específica do mapa, sem precisar restringir o tipo de classe que poderia ser armazenado ali.

Além disso, criamos métodos auxiliares para manipulação desses objetos. Por exemplo, ao criar uma cidade ou unidade em uma posição do mapa, utilizávamos um método específico para verificar se a célula estava ocupada ou não, e então substituíamos a instância de acordo com a necessidade. Para garantir que os objetos fossem gerenciados corretamente, implementamos um controle rigoroso de alocação e desalocação de objetos, garantindo que cada célula fosse liberada corretamente quando um novo objeto fosse colocado em seu lugar. Essa abordagem garantiu flexibilidade ao atualizar e substituir objetos dinamicamente, sem prejudicar o desempenho ou a estrutura do mapa.

Outro grande desafio foi a criação de unidades e cidades e a colocação delas no mapa, assim como a gestão simultânea das listas que armazenam essas entidades no ArrayList. Precisávamos garantir que, ao criar uma unidade ou cidade, ela fosse adicionada ao mapa (na posição escolhida pelo jogador) e também fosse inserida no ArrayList correspondente (como uma lista de unidades ou cidades). A dificuldade surgiu pela necessidade de manter as listas sincronizadas e garantir que os objetos no mapa e nas listas estivessem sempre atualizados e refletissem o estado correto do jogo.

Para resolver esse desafio, criamos uma abordagem modular em que os métodos responsáveis pela criação de cidades e unidades não apenas instanciavam o objeto, mas também realizavam a inserção tanto no mapa quanto na lista de maneira coordenada. Utilizamos métodos auxiliares para gerenciar esse processo de forma organizada, onde um método seria responsável por adicionar uma unidade ao mapa, e outro método seria responsável por adicioná-la ao ArrayList de unidades.

Além disso, implementamos um controle de verificações para garantir que a posição escolhida para a cidade ou unidade fosse válida e que não houvesse sobreposição de objetos. Após a colocação do objeto no mapa, o mesmo seria adicionado ao ArrayList correspondente, o que permitia fácil manipulação e acesso posterior.

Dessa forma, garantimos que as listas de unidades e cidades estivessem sempre consistentes com o estado do mapa, permitindo a movimentação e a modificação do estado do jogo sem inconsistências.

5. Guia para Expansão e Modificação

5.1. Pontos de Extensibilidade

O jogo foi desenvolvido com uma arquitetura modular e flexível, permitindo que novas funcionalidades e elementos sejam adicionados facilmente. A seguir, apresentamos alguns dos principais pontos de extensibilidade do projeto, que permitem a adição de novos terrenos, unidades e outras características sem a necessidade de reescrever o código existente.

Expansão dos Terrenos

O sistema de terrenos no jogo foi projetado de forma extensível, permitindo a criação de novos tipos de terrenos sem grandes modificações no código principal. Cada tipo de terreno é representado por uma classe que herda de uma classe base Terreno. Para adicionar novos tipos de terrenos, basta criar novas subclasses de Terreno, onde as propriedades específicas de cada tipo de terreno, como custo de movimento, produção de recursos e efeitos no jogo, podem ser definidas. Isso garante que o sistema de terrenos seja altamente configurável e fácil de expandir.

Expansão das Unidades

De maneira semelhante aos terrenos, o sistema de unidades também foi desenvolvido com flexibilidade, permitindo que novas unidades sejam adicionadas facilmente. Cada tipo de unidade (como militares, colonos e construtores) é representado por uma classe que herda de uma classe base Unidade. Para adicionar novos tipos de unidades, basta criar uma nova subclasse de Unidade e definir suas propriedades específicas, como custo de manutenção, ataque, defesa, entre outros. Além disso, o sistema de menus já foi configurado para permitir a seleção de novas unidades conforme elas são adicionadas ao código.

Mudança de Valores das cidades

Para alterar a produção inicial, a produção inicial, comida inicial e a vida basta apenas mudar os atributos abaixo na Fig.7 por outros valores, estes atributos estão na classe Cidade.

```
private static final int POPULACAO_INICIAL = 5;
private static final int PRODUCAO_INICIAL = 10;
private static final int COMIDA_INICIAL = 50;
private static final int VIDA = 100;
```

Fig.7- Valores iniciais de uma cidade

Assim como nos valores anteriores a distância em que uma cidade tem de estar de outra, também faz parte da classe Cidade e é só mudar este parâmetro da fig.8.

```
private final int distanciaMin = 2;
```

Fig 8. Atributo distância mínima entre cidades

Para mudar o máximo pontos de movimento, é apenas necessário mudar o valor do segundo argumento do super.

```
public Militares(String nome) {
    super(nome,maxpontosmovimento:15);
    this.ataque = 10;
    this.vida = 5;
}
```

Fig. 9- Mudar máximo de pontos de movimento

5.2. Boas Práticas para Continuação

Para garantir a continuidade do desenvolvimento do projeto de forma eficiente, mantendo a qualidade e a consistência do código, aqui estão algumas recomendações importantes para desenvolvedores futuros. Essas sugestões abrangem tanto as boas práticas de codificação quanto a aplicação de padrões de design, permitindo que o código se mantenha escalável, legível e fácil de modificar.

Adotar um padrão claro para nomeação de classes, métodos, variáveis e constantes.

Continuar a comentar o código de maneira objetiva, explicando partes complexas do código, decisões de design e comportamento esperado. Isso ajudará desenvolvedores futuros a entender rapidamente a lógica por trás do código e a modificar ou expandir o sistema sem cometer erros.

Documentar funções e classes, explicando o que elas fazem e o que recebem como entrada e saída.

Evitar comentários excessivos em partes simples ou autoexplicativas do código. Mantenha a modularização do código, ou seja, divida o código em métodos e classes menores, com uma única responsabilidade. Isso facilita a leitura, a manutenção e a reutilização do código.

É importante manter a abordagem de cada casa do mapa sendo representada por uma letra (como "A" para área, "M" para montanha, etc.), pois isso oferece simplicidade e eficiência no armazenamento e manipulação do mapa. No entanto, à medida que o jogo cresce, novos recursos podem ser adicionados, como a exibição de informações adicionais sobre o terreno ou unidades. Para manter o design escalável e eficiente, considere as seguintes práticas.

5.3. Potenciais Melhorias

Uma possível melhoria seria a implementação e utilização mais abrangente do método toString() nas classes, especialmente para exibir informações detalhadas sobre unidades, cidades e terrenos, de forma que o jogador possa ter uma visão mais clara e acessível dos elementos do jogo durante as interações no menu, o que poderia aprimorar a experiência do usuário.

Uma das melhorias seria a incrementação de um Sistema de Inteligência Artificial (IA) para Jogadores Não-humanos. Em que era adicionar um sistema de inteligência artificial (IA) para controlar jogadores não-humanos (bots) seria uma melhoria considerável. Isso permitiria que o jogo tivesse uma dinâmica mais interessante, permitindo que os jogadores competissem contra adversários controlados por IA, sem depender exclusivamente de jogadores humanos.Isto tornaria o jogo mais dinâmico e desafiador, permitindo jogá-lo sem a necessidade de outros jogadores.

Implementação de diferentes níveis de dificuldade da IA, com comportamentos adaptativos.

Outra melhoria mais complexa seria adicionar um sistema de diplomacia. Este permitiria que os jogadores interagissem entre si de maneiras mais complexas do que apenas por meio de combate. Jogadores poderiam negociar, formar alianças, realizar trocas de recursos, estabelecer tratados de paz ou até mesmo declarar guerra a outros jogadores ou IA.

Uma melhoria mais estética/friendly user seria permitir que os jogadores personalizem

as unidades e cidades seria uma melhoria interessante. Isso poderia incluir opções para mudar o nome das unidades, escolher diferentes tipos de unidades (com base em subclasses ou habilidades especiais) ou até mesmo a personalização de cidades (como o design, a especialização em determinados recursos ou a aparência visual).

Por fim, achamos que poderia ser adicionado um sistema de progressão tecnológica que permitiria que os jogadores desbloqueassem novas unidades, edifícios ou habilidades à medida que progridem no jogo. Este sistema poderia ser baseado em um árvore de tecnologia, onde os jogadores escolhem quais pesquisas conduzir para melhorar sua civilização.

6.Testes e Validação

6.1.Estratégia de Testes

A estratégia de testes adotada para o desenvolvimento do jogo foi baseada em uma abordagem incremental e contínua, onde as funcionalidades foram testadas à medida que eram implementadas. Assim que cada parte do projeto era desenvolvida, seja uma classe ou um método, ela era imediatamente submetida a testes para garantir que o comportamento esperado estava sendo alcançado.

A cada novo componente do sistema, como a criação do mapa, gerenciamento dos jogadores, ou a implementação de funcionalidades específicas das cidades, um conjunto de testes unitários foi desenvolvido para verificar o funcionamento adequado de cada unidade. Esses testes incluíam verificar condições básicas, como a validação da criação de cidades e unidades, movimentação de personagens, e interação entre elementos do jogo.

Além disso, o sistema foi validado por meio de testes de integração, onde as diferentes partes do código, ao serem conectadas, foram avaliadas quanto ao seu desempenho conjunto. A interação entre as diversas funcionalidades foi cuidadosamente monitorada, buscando identificar problemas de integração que poderiam surgir quando as classes ou métodos fossem utilizados em conjunto, como problemas de consistência de dados entre o mapa, as cidades e os jogadores.

Os testes de carga também foram realizados para garantir que o sistema fosse capaz de lidar com grandes quantidades de dados, como o aumento do número de jogadores ou a adição de várias cidades e unidades. Por fim, testes exploratórios foram feitos para cobrir cenários que não haviam sido antecipados, permitindo descobrir possíveis falhas em situações inesperadas ou não planejadas.

6.2.Correção de Erros

Durante os testes, diversos bugs e falhas foram identificados e corrigidos. Esses erros surgiram em várias etapas do desenvolvimento e, em muitos casos, foram corrigidos imediatamente após a detecção.

Erro de Validação na Criação de Cidades: Um dos primeiros problemas encontrados ocorreu quando o jogador tentava construir uma cidade em uma posição aparentemente vazia do mapa, mas que na verdade já estava ocupada por um terreno. Esse bug foi corrigido

ajustando a lógica de verificação de posições livres, garantindo que a cidade só fosse criada em terrenos realmente desocupados.

Erro de Inicialização de Recursos: Durante os testes de produção de recursos das cidades, foi identificado que a quantidade de recursos não estava sendo calculada corretamente. Isso foi causado por uma falha na atualização dos totais de recursos após a fundação de uma cidade. A correção envolveu a revisão da lógica de produção de recursos, ajustando o método que atualizava os valores de produção de ouro, comida e produção em cada turno.

Erro de Sincronização entre Jogadores: Em certos momentos, durante a alternância de turnos, o sistema não estava reconhecendo corretamente o próximo jogador na sequência, resultando em turnos repetidos ou saltados. Esse problema foi corrigido ajustando a lógica de rotação entre os jogadores e garantindo que o índice do jogador fosse atualizado corretamente após cada turno.

Além de resolver esses erros específicos, também houve ajustes de desempenho para garantir que o sistema fosse eficiente, principalmente quando o número de jogadores ou cidades aumentava. Esses problemas de performance foram resolvidos otimizando as consultas e atualizações de dados, especialmente no que diz respeito ao gerenciamento do mapa e da movimentação das unidades.

O processo de testes e correções foi contínuo ao longo do desenvolvimento do projeto, garantindo que cada nova funcionalidade fosse implementada de forma robusta e sem introduzir novos problemas. O uso de testes unitários, de integração e a correção imediata de bugs permitiram uma evolução gradual e controlada do sistema, resultando em um jogo mais estável e funcional no final do ciclo de desenvolvimento.

7.Conclusão

O desenvolvimento deste projeto proporcionou uma compreensão mais profunda sobre o design de software e a aplicação dos princípios da Programação Orientada a Objetos (POO). Alguns dos principais aprendizados neste projeto foi a importância da modularidade e flexibilidade, através escolha de usar estruturas de dados como ArrayList e duplo array para o mapa do jogo permitiu uma gestão eficiente e flexível dos elementos do jogo. Isso facilitou a adição ou remoção de elementos, como jogadores, cidades e unidades, sem a necessidade de reestruturar o sistema, refletindo a aplicação dos princípios de modularidade e extensibilidade.

A aplicação de princípios como o Princípio da Responsabilidade Única (SRP) e o Princípio Aberto/Fechado (OCP) foi essencial para manter o código organizado e escalável. O SRP ajudou a garantir que cada classe tivesse uma responsabilidade bem definida, enquanto o OCP permitiu que novas funcionalidades fossem incorporadas sem modificar o código existente. O Princípio da Inversão de Dependência (DIP) também foi crucial para garantir que o código fosse flexível e fácil de testar, ao utilizar abstrações em vez de dependências diretas.

Gerenciamento Eficiente de Exceções: O tratamento adequado de exceções foi uma lição importante, especialmente no que diz respeito a capturar e lidar com erros de maneira

controlada. A implementação de blocos try-catch e a criação de exceções personalizadas permitiram que problemas fossem isolados e resolvidos sem comprometer a estabilidade do sistema.

Iteração e Testes Contínuos: Testar cada método e classe à medida que eram implementados foi uma abordagem fundamental para detectar erros precocemente e garantir que o sistema funcionasse conforme o esperado. A reflexão constante sobre os resultados dos testes ajudou a aprimorar a qualidade do código e a corrigir falhas rapidamente, contribuindo para a robustez do sistema.

Engajamento do Jogador e Experiência de Usuário: A criação de uma interface interativa, permitindo ao jogador escolher o local para a fundação das cidades, foi um destaque do projeto. Esse recurso aumentou o engajamento do jogador e fez com que o jogo fosse mais dinâmico e interessante desde o início.

A experiência de aplicar os princípios de Programação Orientada a Objetos ao longo deste projeto foi extremamente enriquecedora. A POO possibilitou a criação de um código mais organizado, reutilizável e de fácil manutenção. A ênfase na criação de classes bem definidas com responsabilidades específicas ajudou a evitar a complexidade excessiva, e a aplicação de herança e polimorfismo tornou o código mais flexível, permitindo a adição de novos recursos de forma natural.

Além disso, a prática com os princípios SOLID durante o desenvolvimento contribuiu para a criação de um sistema que não apenas atende aos requisitos do projeto, mas também está preparado para ser expandido no futuro. A experiência com a gestão de exceções e a realização de testes contínuos também foram aspectos cruciais que garantiram a estabilidade do sistema.

Em resumo, o projeto foi uma excelente oportunidade para aplicar conceitos de POO de maneira prática, reforçando a importância da organização e da estruturação do código. A experiência adquirida neste projeto será fundamental para o desenvolvimento de futuros sistemas, proporcionando uma base sólida para a construção de aplicações robustas, escaláveis e fáceis de manter.

8.Anexos

8.1.Link do diagrama UML:

https://app.diagrams.net/#G1cfCC2s_wgBR0gmCYbEFrGQQ3VCS2_Xa0#%7B"pageId"%3A"yfxD58SMssEPQ-j69YMJ"%7D

8.2.Código-fonte

Classe Agua

```
/**
 * Representa um terreno do tipo Água no jogo.
 * A água é um tipo especial de terreno que não pode ser explorado ou trabalhado,
 * mas é representado no mapa.
 */
```

```
public class Agua extends Terrenos {
    private int x;
    private int y;

    /**
     * Construtor para inicializar um terreno do tipo Água com coordenadas específicas.
     *
     * @param x a coordenada X do terreno.
     * @param y a coordenada Y do terreno.
     */
    public Agua(int x, int y) {
        super(9999); // Define um custo arbitrário muito alto para representar que este terreno é inacessível.
        this.x = x;
        this.y = y;
    }

    /**
     * Obtém a coordenada X deste terreno.
     *
     * @return a coordenada X.
     */
    @Override
    public int getX() {
        return this.x;
    }

    /**
     * Obtém a coordenada Y deste terreno.
     *
     * @return a coordenada Y.
     */
    @Override
    public int getY() {
        return this.y;
    }

    /**
     * Retorna uma representação textual do terreno.
     * Neste caso, a letra "A" representa a água no mapa.
     *
     * @return uma string contendo "A".
     */
    @Override
```

```
    public String toString() {
        return "A";
    }
}
```

Classe Cidades

```
import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author mgng2
 */
public class Cidades {
    private final int distanciaMin = 2;
    private ArrayList<int[]> posicoesCidades;
    private String nome;
    private int populacao;
    private int producao;
    private Producao gestaoProducao;
    private List<Unidade> unidades;
    private int vida;
    private int comida;
    private int posX;
    private int posY;
    private Comida gestaoComida;
    private Ouro gestaoOuro;
    private List<Terrenos> terrenosTrabalhados;
    private int comidainicial;
    private int totalComida = 10;
    private int totalProducao = 10;
    private int totalOuro = 10;
    private Player player;
    private int raioInfluencia = 3;
    private boolean raioAumentado = false;
    private static int contadorCidades = 1;
    private static final int POPULACAO_INICIAL = 5;
    private static final int PRODUCAO_INICIAL = 10;
    private static final int COMIDA_INICIAL = 50;
    private static final int VIDA = 100;
    private int contadorMilitares = 0;
    private int contadorColonos = 0;
```

```
    private int contadorConstrutores = 0;

    public Cidades(int populacao, int ouroInicial, int producaoPorTurno, int posX, int posY,
Mapa mapaa, Player player) {
        this.gestaoOuro = new Ouro(ouroInicial,0);
        this.gestaoProducao = new Producao(producaoPorTurno);
        this.posX = posX;
        this.posY = posY;
        this.comida = comidainicial;
        this.populacao = populacao;
        this.player = null;
        this.nome = "cidade" + contadorCidades++;
        this.populacao = POPULACAO_INICIAL;
        this.unidades = new ArrayList<>();
        this.terrenosTrabalhados = new ArrayList<>();
        this.gestaoProducao = new Producao(PRODUCAO_INICIAL);
        this.gestaoComida = new Comida(COMIDA_INICIAL);
        this.gestaoOuro = new Ouro(0, 0);
        this.vida = VIDA;
        this.comida = COMIDA_INICIAL;

    }

    /**
     * Método para obter a posição x
     * @param player O jogador
     */
    public void setPlayer(Player player) {
        this.player = player;
    }

    /**
     * Método para adicionar uma cidade
     * @param x coordenada de x a adicionar cidade
     * @param y coordenada de y a adicionar cidade
     * @param nomeCidade nome da cidade
     * @return true se a cidade foi adicionada, false caso contrário
     */
    public boolean adicionarCidade(int x, int y, String nomeCidade) {
        if (posicaoLivre(x, y)) {
            posicoesCidades.add(new int[]{x, y});
            System.out.println("Cidade " + nomeCidade + " foi criada na posição (" + x + ", " + y
+ ")");
            return true;
```

```
    } else {
        System.out.println("Posição ocupada ou muito próxima a outra cidade.");
        return false;
    }
}

/**
 * Método para adicionar uma cidade
 * @param cidades lista de cidades
 * @param cidade cidade a ser adicionada
 */
public void adicionarCidade(List<Cidades> cidades, Cidades cidade) {
    cidades.add(cidade);
}

/**
 * Método para verificar se a posição está livre
 * @param x coordenada de x
 * @param y coordenada de y
 * @return true se a posição está livre, false caso contrário
 */
private boolean posicaoLivre(int x, int y) {
    for (int[] cidade : posicoesCidades) {
        int cidadeX = cidade[0];
        int cidadeY = cidade[1];
        int MinX = cidadeX - distanciaMin;
        int MaxX = cidadeX + distanciaMin;
        int MinY = cidadeY - distanciaMin;
        int MaxY = cidadeY + distanciaMin;

        if (x >= MinX && x <= MaxX && y >= MinY && y <= MaxY) {
            return false;
        }
    }
    return true;
}

// Getters
public ArrayList<int[]> getPosicoesCidades(){return posicoesCidades;}
public String getNome() {return nome;}
public int getPopulacao() {return populacao;}
public int getProducao() {return producao;}
public List<Unidade> getUnidades() {return unidades;}
public int getVida() {return vida;}
```

```
public int getPosX() {
    return posX;
}
public int getPosY() {
    return posY;
}
public void setNome(String nome) {this.nome = nome;}
/**
 * Método para adicionar um terreno trabalhado
 * @param terreno O terreno a ser adicionado
 */
public void adicionarTerrenoTrabalhado(TerrenoTrabalhado terreno) {
    if (terreno != null) {
        terrenosTrabalhados.add(terreno);
    }
}


/**
 * Método para aumentar a população
 * @param quantidade A quantidade a ser aumentada
 */
public void aumentarPopulacao(int quantidade) {
    this.populacao += quantidade;
}

/**
 * Método para reduzir a população
 * @param quantidade A quantidade a ser reduzida
 */
public void reduzirPopulacao(int quantidade) {
    this.populacao -= quantidade;
    if (this.populacao < 0) {
        this.populacao = 0;
    }
}
/**
 * Método para obter a quantidade de ouro
 * @return A quantidade de ouro
 */
public int getOuro() {
    return gestaoOuro.obterQuantidade();
}
```

```
/**
 * Método para adicionar ouro
 * @param quantidade A quantidade de ouro a ser adicionada
 */
public void adicionarOuro(int quantidade) {
    gestaoOuro.adicionarOuro(quantidade);
}

/**
 * Método para remover ouro
 * @param quantidade A quantidade de ouro a ser removida
 */
public void removerOuro(int quantidade) {
    gestaoOuro.removerOuro(quantidade);
}

/**
 * Método para produzir ouro
 * @param terreno O terreno a ser trabalhado
 * @param mapa O mapa
 */
public void alocarCidadao(Terrenos terreno, Mapa mapa) {
    if (terrenosTrabalhados.size() >= populacao) {
        System.out.println("Todos os cidadãos estão alocados.");
        return;
    }
    if (!(terreno instanceof Floresta || terreno instanceof Pedreira)) {
        System.out.println("Este terreno não pode ser trabalhado.");
        return;
    }
    int distanciaX = Math.abs(terreno.getX() - this.getPosX());
    int distanciaY = Math.abs(terreno.getY() - this.getPosY());
    if (distanciaX > raioInfluencia || distanciaY > raioInfluencia) {
        System.out.println("O terreno está muito distante para ser alocado.");
        return;
    }
    if (terrenosTrabalhados.contains(terreno)) {
        System.out.println("Este terreno já foi alocado.");
        return;
    }
    Terrenos terrenoMapa = mapa.getMapa()[terreno.getX()][terreno.getY()];
    if (terrenoMapa instanceof Floresta || terrenoMapa instanceof Pedreira) {
```

```
            mapa.getMapa()[terreno.getX()][terreno.getY()] = new TerrenoTrabalhado(terreno, this.getNome(), 1, terreno.getX(), terreno.getY());
            terrenosTrabalhados.add(terreno);
            mapa.adicionarTerrenoTrabalhado(terreno);
            System.out.println("Cidadão alocado ao terreno " + terreno);
        } else {
            System.out.println("O terreno não pode ser trabalhado porque não é Floresta ou Pedreira.");
        }
    }

    /**
     * Método para desalocar um cidadão
     * @param terreno O terreno a ser desalocado
     */
    public void desalocarCidadão(Terrenos terreno) {
        if (terrenosTrabalhados.remove(terreno)) {
            System.out.println("Cidadão desalocado do terreno: " + terreno.toString());
        } else {
            System.out.println("Esse terreno não está sendo trabalhado.");
        }
    }
    /**
     * Método para produzir recursos
     */
    public void produzirRecursos() {
        totalComida = 0;
        totalProducao = 0;
        totalOuro = 0;

        for (Terrenos terreno : terrenosTrabalhados) {
            if (terreno instanceof Floresta) {
                totalComida += ((Floresta) terreno).getProducaoPorTurno();
            } else if (terreno instanceof Pedreira) {
                totalOuro += ((Pedreira) terreno).getProducaoPorTurno();
            }
        }
        gestaoComida.produzirComida(totalComida);
        gestaoProducao.setProducaoPorTurno(totalProducao);
        gestaoOuro.adicionarOuro(totalOuro);

        System.out.printf("Cidade %s produziu %d comida, %d produção e %d ouro.\n", nome, totalComida, totalProducao, totalOuro);
    }
```

```
/**
 * Método para obter a quantidade de comida
 * @return A quantidade de comida
 */
public void proximoTurno() {
    produzirRecursos();
    consumirComidaPorPopulacao();
    crescerPopulacao();
    gestaoComida.proximoTurno();
    gestaoProducao.reiniciarProducao();
}
/**
 * Método para criar uma unidade
 * @param player O jogador
 * @param tipo O tipo de unidade
 * @param nomeCidade O nome da cidade
 * @param mapa O mapa
 */
public void criarUnidade(Player player, String tipo, String nomeCidade, Mapa mapa) {
    Unidade novaUnidade = null;
    int custoproducao;
    switch (tipo.toLowerCase()) {
        case "militar":
            novaUnidade = new Militares("Militar");
            custoproducao = 5;
            break;
        case "colono":
            novaUnidade = new Colonos("Colono");
            custoproducao = 3;
            break;
        case "construtor":
            novaUnidade = new Construtores("Construtor");
            custoproducao = 3;
            break;
        default:
            System.out.println("Tipo de unidade desconhecido.");
            return;
    }
    if (gestaoProducao.usarProducao(custoproducao)) {
        if (mapa.adicionarUnidade(player, novaUnidade, nomeCidade)) {
            player.addUnidade(novaUnidade);
            incrementarContadorUnidade(novaUnidade);
```

```
            System.out.println(tipo + " criado próximo à cidade " + nomeCidade + " usando " +
custoproducao + " de produção.");

        } else {
            System.out.println("Não foi possível adicionar unidade ao mapa.");
          }
        } else {
          System.out.println("Produção insuficiente para criar " + tipo + " perto da cidade " +
nomeCidade);
        }
    }

    /**
     * Método para atacar uma cidade
     * @param dano O dano a ser causado
     */
    public void takeDamage(int dano) {
        this.vida -= dano;
        if (this.vida <= 0) {
            System.out.println("A cidade " + nome + " foi destruída.");
        } else {
            System.out.println("A cidade " + nome + " sofreu dano. Vida restante: " + this.vida);
        }
    }
    // Getters
    public Player getPlayer() {
        return player;
    }
    /**
     * Método para adicionar uma unidade
     * @param unidade A unidade a ser adicionada
     */
    public void adicionarUnidade(Unidade unidade) {
        unidades.add(unidade);
    }
    /**
     * Método para retornar a cidade
     */
    @Override
    public String toString() {
        StringBuilder info = new StringBuilder();
        info.append("Cidade: ").append(nome).append("\n");
        info.append("População: ").append(populacao).append("\n");
        info.append("Produção: ").append(gestaoProducao).append("\n");
```

```
        info.append("Comida: ").append(gestaoComida.getReservaDeComida()).append("\n");
        info.append("Vida: ").append(vida).append("\n");
        info.append("Ouro: ").append(getOuro()).append("\n");
        info.append("Unidades:\n");
        info.append("- Colonos: ").append(contadorColonos).append("\n");
        info.append("- Militares: ").append(contadorMilitares).append("\n");
        info.append("- Construtores: ").append(contadorConstrutores).append("\n");
        return info.toString();
    }
    /**
     * Método para crescer a população da cidade
     */
    public void crescerPopulacao() {
        if (gestaoComida.getReservaDeComida() >= 100) {
            gestaoComida.produzirComida(-20);
            populacao++;
            System.out.printf("Cidade %s cresceu para a população de %d.\n", nome, populacao);
            if (populacao == 10) {
                aumentarRaioInfluencia();
            }
        } else {
            System.out.printf("Cidade %s não tem comida suficiente para crescer.\n", nome);
        }
    }
    /**
     * Método para aumentar o raio de influência da cidade
     */
    private void aumentarRaioInfluencia() {
        raioInfluencia = 4; // Aumenta para 4 espaços (9x9)
        raioAumentado = true; // Marca que o raio foi aumentado
        System.out.printf("Cidade %s evoluiu e seu raio de influência cresceu para %dx%d.\n",
nome, raioInfluencia * 2 + 1, raioInfluencia * 2 + 1);
    }
    /**
     * Método para obter o raio de influência
     */
    public boolean isRaioAumentado() {
        return raioAumentado;
    }
    /**
     * Método para obter a produção que resta
     * @return A produção resta
     */
    public int getProducaoRestante() {
```

```
        return gestaoProducao.getProducaoRestante();
    }
    /**
     * Método para consumir produção
     * @param quantidade A quantidade a ser consumida
     * @return true se a produção foi consumida, false caso contrário
     */
    public boolean consumirProducao(int quantidade) {
        return gestaoProducao.usarProducao(quantidade);
    }
    /**
     * Método para consumir comida pela população
     */
    public void consumirComidaPorPopulacao() {
        int comidaConsumida = populacao * 10;
        int comidaDisponivel = gestaoComida.getReservaDeComida();

        if (comidaDisponivel < comidaConsumida) {
            int pessoasNaoAlimentadas = (comidaConsumida - comidaDisponivel) / 10;
            reduzirPopulacao(pessoasNaoAlimentadas);
            gestaoComida.produzirComida(-comidaDisponivel);
            System.out.printf("Cidade %s não tinha recursos suficientes para alimentar toda a
população. %d pessoas morreram.\n", nome, pessoasNaoAlimentadas);
        } else {
            gestaoComida.produzirComida(-comidaConsumida);
            System.out.printf("Cidade %s consumiu %d de comida. Comida restante: %d.\n",
nome, comidaConsumida, gestaoComida.getReservaDeComida());
        }
    }

    public void incrementarContadorUnidade(Unidade unidade) {
        if (unidade instanceof Colonos) {
            contadorColonos++;
        } else if (unidade instanceof Militares) {
            contadorMilitares++;
        } else if (unidade instanceof Construtores) {
            contadorConstrutores++;
        }
    }
}
```

Classe Colonos

```
import java.util.List;

/**
 *
 * @author mgng2
 */
public class Colonos extends Unidade{
    private int POPULACAO_INICIAL;
    private int OURO_INICIAL;
    private int PRODUCAO_INICIAL;
    private List<Cidades> listcidades;


    /**
     * Construtor da classe Colonos.
     * @param nome O nome da unidade.
     */
    public Colonos(String nome) {
        super(nome,  10);
        this.listcidades=listcidades;
    }

    public boolean fundarCidade(Player player, String nomeCidade, Mapa mapa, int x, int y) {
        // Verifica se a posição é válida
        if (!mapa.posicaoLivre(x, y)) {
            return false;
        }

        // Verifica se já existe uma cidade próxima
        if (mapa.existeCidadeProxima(x, y)) {
            System.out.println("Já existe uma cidade próxima.");
            return false;
        }

        // Remove colono da lista de unidades do jogador
        player.removerUnidade(this);

        // Cria a cidade
        if (mapa.criarCidade2(x, y, player)) {
            System.out.println(getNome() + " fundou uma nova cidade!");
            return true;
        }

        return false;
```

```
    }
    /*
     * Método que executa a ação da unidade
     */
    @Override
    public void performAction() {
        System.out.println(getNome() + " está a fundar uma nova cidade.");

    }
}
```

Classe Comida

```
/**
 *
 * @author nelso
 */
public class Comida{
    private int comidaConsumidaPorPopulacao;
    private int reservaDeComida;
    private int limiteParaCrescimento;
    private int populacao;

    /**
     * Construtor padrão que inicializa os valores com configurações padrão.
     * A comida produzida e a reserva inicial são 20, e a população começa com 1.
     */
    public Comida() {
        this.comidaConsumidaPorPopulacao = 1;
        this.reservaDeComida = 20;
        this.limiteParaCrescimento = 20;
        this.populacao = 1;
    }

    /**
     * Construtor que inicializa a comida produzida com um valor inicial especificado.
     *
     * @param comidaInicial a quantidade inicial de comida produzida por turno.
     */
    public Comida(int comidaInicial) {
        this.reservaDeComida = comidaInicial;
    }
```

```
    /**
     * Produz comida e adiciona à reserva de comida armazenada.
     */
    public void produzirComida(int quantidade) {
        this.reservaDeComida += quantidade;
    }

    /**
     * Consome comida com base na população atual.
     * Caso a reserva de comida seja insuficiente, a população é reduzida em 1
     * e a reserva é zerada.
     */
    public void consumirComida() {
        int comidaNecessaria = populacao * comidaConsumidaPorPopulacao;
        this.reservaDeComida -= comidaNecessaria;

        if (this.reservaDeComida < 0) {
            populacao -= 1; // Reduz a população em caso de fome.
            this.reservaDeComida = 0;
        }
    }

    /**
     * Verifica se a população pode crescer com base na reserva de comida.
     * Se houver comida suficiente para o crescimento, a população aumenta
     * e a comida necessária para o crescimento é removida da reserva.
     */
    public void verificarCrescimento() {
        if (this.reservaDeComida > limiteParaCrescimento) {
            populacao += 1; // Incrementa a população.
            this.reservaDeComida -= limiteParaCrescimento; // Consome comida para o
crescimento.
        }
    }

    /**
     * Obtém a quantidade de comida armazenada na reserva.
     *
     * @return a quantidade de comida armazenada.
```

```
     */
    public int getReservaDeComida() {
        return reservaDeComida;
    }

    /**
     * Avança para o próximo turno, realizando as ações de produzir comida,
     * consumir comida e verificar crescimento populacional.
     */
    public void proximoTurno() {
        consumirComida();
        verificarCrescimento();
    }

    /**
     * Retorna uma representação textual do estado atual da comida e da população.
     *
     * @return uma string no formato "População: X, Reserva de Comida: Y".
     */
    @Override
    public String toString() {
        return "População: " + populacao + ", Reserva de Comida: " + reservaDeComida;
    }



    /**
     * Define a quantidade de comida consumida por cada unidade de população por turno.
     *
     * @param comidaConsumidaPorPopulacao a nova quantidade de comida consumida por unidade de população.
     */
    public void setComidaConsumidaPorPopulacao(int comidaConsumidaPorPopulacao) {
        this.comidaConsumidaPorPopulacao = comidaConsumidaPorPopulacao;
    }

    /**
     * Define o limite de comida necessário para que a população cresça.
```

```
     *
     * @param limiteParaCrescimento a nova quantidade de comida necessária para crescimento populacional.
     */
    public void setLimiteParaCrescimento(int limiteParaCrescimento) {
        this.limiteParaCrescimento = limiteParaCrescimento;
    }
}
```

Classe Construtores

```
/**
 *
 * @author mgng2
 */
public class Construtores extends Unidade{
    public Construtores(String nome) {
        super(nome, 10);
    }


    @Override
    public void performAction() {
        System.out.println(getNome() + " está a construir uma melhoria no terreno.");
        // Lógica para construção de estradas, fazendas, minas, etc.
    }

    public void repairInfrastructure() {
        System.out.println(getNome() + " está a reparar infraestruturas danificadas.");
        // Lógica para reparo

    }
}
```

Classe Floresta

```
/**
 * Representa um terreno do tipo Floresta no jogo.
 * A floresta pode fornecer recursos como comida e produção por turno.
 */
public class Floresta extends Terrenos implements Recursos {
    private int comida;
    private int producaoPorTurno = 10;
    private int x;
    private int y;
```

```
/**
 * Construtor que inicializa a Floresta com coordenadas específicas.
 *
 * @param x a coordenada X da floresta.
 * @param y a coordenada Y da floresta.
 */
public Floresta(int x, int y) {
    super(4); // Define um custo fixo para movimentação neste tipo de terreno.
    this.x = x;
    this.y = y;
}

/**
 * Obtém a quantidade de comida coletada da floresta.
 *
 * @return a quantidade de comida disponível.
 */
@Override
public int getComida() {
    return comida;
}

/**
 * Obtém a quantidade de produção gerada pela floresta.
 * Neste caso, a floresta não fornece produção acumulada diretamente.
 *
 * @return 0, pois a floresta não acumula produção diretamente.
 */
@Override
public int getProducao() {
    return 0;
}

/**
 * Obtém a quantidade de ouro gerada pela floresta.
 * Florestas não produzem ouro.
 *
 * @return 0, pois a floresta não gera ouro.
 */
@Override
public int getOuro() {
    return 0;
}
```

```
/**
 * Obtém a quantidade de produção gerada pela floresta por turno.
 *
 * @return a quantidade de produção por turno, que é 10.
 */
@Override
public int getProducaoPorTurno() {
    return producaoPorTurno;
}

/**
 * Retorna uma representação textual do terreno.
 * Neste caso, a letra "F" representa a floresta no mapa.
 *
 * @return uma string contendo "F".
 */
@Override
public String toString() {
    return "F";
}

/**
 * Coleta comida gerada pela floresta, adicionando a produção por turno
 * à quantidade acumulada de comida.
 */
public void coletarComida() {
    comida += producaoPorTurno;
}

/**
 * Obtém a coordenada X da floresta.
 *
 * @return a coordenada X.
 */
@Override
public int getX() {
    return this.x;
}

/**
 * Obtém a coordenada Y da floresta.
 *
 * @return a coordenada Y.
```

```
     */
    @Override
    public int getY() {
        return this.y;
    }
}
```

Classe Mapa

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;


/**
 * Representa o mapa do jogo, contendo terrenos, cidades e unidades.
 * Permite manipular e interagir com os elementos do mapa, como criar cidades,
 * trabalhar terrenos e mover unidades.
 */
public class Mapa {
    private Terrenos[][] mapa;
    private Random random;
    private int numero;
    private List<Cidades> listcidades;
    private final int distanciaMin = 2;
    private List<Terrenos> terrenosTrabalhados;
    private List<int[]> posicoesCidades;
    private Player player;

    /**
     * Construtor que inicializa o mapa com dimensões especificadas.
     *
     * @param largura    largura do mapa.
     * @param comprimento comprimento do mapa.
     */
    public Mapa(int largura, int comprimento) {
        this.mapa = new Terrenos[comprimento][largura];
        this.random = new Random();
        this.listcidades = new ArrayList<>();
        this.posicoesCidades = new ArrayList<>();
        this.terrenosTrabalhados = new ArrayList<>();
        preencherMapa();
    }

    /**
     * Cria uma nova cidade no mapa, se a posição for válida.
```

```
     * @param x      coordenada X da cidade.
     * @param y      coordenada Y da cidade.
     * @param player jogador que possui a cidade.
     * @return true se a cidade foi criada com sucesso, false caso contrário.
     */
    public boolean criarCidade(int x, int y, Player player) {
        try {
            if (!(mapa[y][x] instanceof TerrenoVazio)) {
                throw new IllegalArgumentException("O terreno não é adequado para uma cidade.");
            }
            if (!posicaoLivre(x, y)) {
                throw new IllegalArgumentException("A posição está muito próxima de outra cidade.");
            }

            Cidades novaCidade = new Cidades(10, 0, 10, x, y, this, player);
            posicoesCidades.add(new int[]{x, y});
            listcidades.add(novaCidade);
            player.adicionarCidade(novaCidade);

            mapa[y][x] = new TerrenoCidade(novaCidade, x, y);

            System.out.println("Cidade " + novaCidade.getNome() + " criada em (" + x + ", " + y + ")");
            return true;

        } catch (IllegalArgumentException e) {
            System.out.println("Erro ao criar cidade: " + e.getMessage());
            return false;
        } catch (Exception e) {
            System.out.println("Ocorreu um erro inesperado: " + e.getMessage());
            return false;
        }
    }

    /**
     * Cria um terreno trabalhado em uma posição específica, se permitido.
     *
     * @param x      coordenada X do terreno.
     * @param y      coordenada Y do terreno.
     * @param cidades cidade que trabalha o terreno.
     * @return true se o terreno foi trabalhado com sucesso, false caso contrário.
     */
```

```
    public boolean criarTerrenoTrabalhado(int x, int y, Cidades cidades) {
        try {
            if (!(mapa[y][x] instanceof Floresta || mapa[y][x] instanceof Pedreira)) {
                System.out.println("Terreno encontrado: " + mapa[y][x].getClass().getName());
                throw new IllegalArgumentException("O terreno não é adequado para trabalho. Esperado 'Floresta' ou 'Pedreira'!");
            }
            if (mapa[y][x] instanceof TerrenoTrabalhado) {
                throw new IllegalArgumentException("Este terreno já foi trabalhado.");
            }
            if (!posicaoLivre(x, y)) {
                throw new IllegalArgumentException("A posição está ocupada por outra unidade ou cidade.");
            }

            Terrenos terrenoOriginal = mapa[y][x];
            int quantidadeTrabalhadores = 1;
            TerrenoTrabalhado terrenoTrabalhado = new TerrenoTrabalhado(terrenoOriginal, cidades.getNome(), quantidadeTrabalhadores, x, y);
            mapa[y][x] = terrenoTrabalhado;
            cidades.adicionarTerrenoTrabalhado(terrenoTrabalhado);
            System.out.println("Terreno trabalhado criado em (" + x + ", " + y + ") pela cidade " + cidades.getNome());
            return true;

        } catch (IllegalArgumentException e) {
            System.out.println("Erro ao criar terreno trabalhado: " + e.getMessage());
            return false;
        } catch (Exception e) {
            System.out.println("Ocorreu um erro inesperado: " + e.getMessage());
            return false;
        }
    }

    /**
     * Preenche o mapa com terrenos aleatórios.
     */
    public void preencherMapa() {
        for (int i = 0; i < this.mapa.length; i++) {
            for (int j = 0; j < this.mapa[i].length; j++) {
                numero = random.nextInt(101);
                if (numero < 5) {
                    this.mapa[i][j] = new Floresta(i, j);
                } else if (numero > 95) {
```

```
                this.mapa[i][j] = new Pedreira(i, j);
            } else if (numero < 85 && numero > 80) {
                this.mapa[i][j] = new Agua(i, j);
            } else {
                this.mapa[i][j] = new TerrenoVazio(i, j);
            }
        }
    }
}

/**
 * Verifica se uma posição está livre para ser ocupada.
 *
 * @param x coordenada X da posição.
 * @param y coordenada Y da posição.
 * @return true se a posição está livre, false caso contrário.
 */
public boolean posicaoLivre(int x, int y) {
    for (int[] cidade : posicoesCidades) {
        int cidadeX = cidade[0];
        int cidadeY = cidade[1];
        int minX = cidadeX - distanciaMin;
        int maxX = cidadeX + distanciaMin;
        int minY = cidadeY - distanciaMin;
        int maxY = cidadeY + distanciaMin;

        if (x >= minX && x <= maxX && y >= minY && y <= maxY) {
            return false;
        }
    }
    return true;
}

/**
 * Exibe o mapa na consola, representando cada tipo de terreno com um caractere específico.
 */
public void exibirMapa() {
    int larguraMaxima = String.valueOf(mapa[0].length - 1).length();

    System.out.print("     ");
    for (int x = 0; x < mapa[0].length; x++) {
        System.out.print(String.format("x%-" + larguraMaxima + "d ", x));
    }
    System.out.println();
```

```
    for (int y = 0; y < mapa.length; y++) {
        System.out.print(String.format("y%-" + (larguraMaxima + 1) + "d", y));

        for (int x = 0; x < mapa[y].length; x++) {
            Terrenos terreno = mapa[y][x];
            String campo;

            if (terreno instanceof TerrenoTrabalhado) {
                campo = "T";
            } else if (terreno instanceof Floresta) {
                campo = "F";
            } else if (terreno instanceof Pedreira) {
                campo = "P";
            } else if (terreno instanceof TerrenoCidade) {
                campo = "C";
            } else if (terreno instanceof TerrenoUnidade) {
                Unidade unidade = ((TerrenoUnidade) terreno).getUnidade();
                if (unidade instanceof Militares) {
                    campo = "M";
                } else if (unidade instanceof Construtores) {
                    campo = "B";
                } else if (unidade instanceof Colonos) {
                    campo = "H";
                } else {
                    campo = "?";
                }
            } else {
                campo = terreno.toString();
            }

            System.out.print("  " + campo + " ");
        }
        System.out.println();
    }
}

/**
 * Obtém o terreno em uma posição específica do mapa.
 * @param unidade unidade que vai ser movida
 * @param novaPosicaoX nova coordenada X da posição
 * @param novaPosicaoY nova coordenada Y da posição
 */
public void moverUnidade(Unidade unidade, int novaPosicaoX, int novaPosicaoY) {
```

```
        try {
            int posXAtual = unidade.getPosicaoX();
            int posYAtual = unidade.getPosicaoY();

            Terrenos terrenoDestino = mapa[novaPosicaoY][novaPosicaoX];
            int custoMovimento = terrenoDestino.getCustoMovimento();

            if (unidade.getPontosMovimentos() < custoMovimento) {
                throw new IllegalArgumentException("Pontos insuficientes para movimento");
            }

            System.out.println("Posição Inicial: (" + posXAtual + "," + posYAtual + ")");
            mapa[posYAtual][posXAtual] = new TerrenoVazio(posYAtual,posXAtual);

            mapa[novaPosicaoY][novaPosicaoX] = new
TerrenoUnidade(unidade,novaPosicaoY,novaPosicaoX);
            unidade.setPosicao(novaPosicaoX, novaPosicaoY);

            unidade.reduzirPontosMovimento(custoMovimento);

            System.out.println("Pontos de movimento restantes: " +
unidade.getPontosMovimentos());
            System.out.println("Após movimento:");
            exibirMapa();

        } catch (Exception e) {
            System.out.println("Erro: " + e.getMessage());
            e.printStackTrace();
        }
    }
    /**
     * Obtém o terreno em uma posição específica do mapa.
     * @return o terreno na posição especificada.
     */
    public List<Terrenos> getTerrenosTrabalhados() {
        return terrenosTrabalhados;
    }
    /**
     * Adiciona um terreno trabalhado à lista de terrenos trabalhados.
     * @param terreno o terreno trabalhado a ser adicionado.
     */
    public void adicionarTerrenoTrabalhado(Terrenos terreno) {
        terrenosTrabalhados.add(terreno);
    }
```

```
/**
 * Busca uma cidade pelo nome.
 * @param nome o nome da cidade a ser buscada.
 * @return a cidade encontrada, ou null se não encontrada.
 */
public Cidades buscarCidadePorNome(String nome) {
    for (Cidades cidade : listcidades) {
        if (cidade.getNome().equals(nome)) {
            return cidade;
        }
    }
    return null;
}

/**
 * Busca a posição da cidade pelo nome.
 * @param nome o nome da cidade a ser buscada.
 * @return a cidade encontrada, ou null se não encontrada.
 */
public int[] buscarPosicaoCidade(String nome) {
    for (int i = 0; i < posicoesCidades.size(); i++) {
        if (listcidades.get(i).getNome().equals(nome)) {
            return posicoesCidades.get(i);
        }
    }
    return null;
}

/**
 * Adiciona uma unidade ao mapa.
 * @param player o jogador que possui a unidade.
 * @param unidade a unidade a ser adicionada.
 * @param nomeCidade o nome da cidade onde a unidade será adicionada.
 * @return true se a unidade foi adicionada com sucesso, false caso contrário.
 */
public boolean adicionarUnidade(Player player, Unidade unidade, String nomeCidade) {
    try {
        for (Cidades cidade : player.getCidades()) {
            if (cidade.getNome().equals(nomeCidade)) {
                    int cidadeX = cidade.getPosX();
                int cidadeY = cidade.getPosY();

                for (int dx = -1; dx <= 1; dx++) {
                    for (int dy = -1; dy <= 1; dy++) {
```

```
                    if (dx == 0 && dy == 0) continue;

                    int novaX = normalizarCoordenada(cidadeX + dx, mapa[0].length);
                    int novaY = normalizarCoordenada(cidadeY + dy, mapa.length);

                    if (mapa[novaY][novaX] instanceof TerrenoVazio) {

                        unidade.setPosicao(novaX, novaY);
                        mapa[novaY][novaX] = new TerrenoUnidade(unidade, novaX, novaY);
                        System.out.println("Unidade " + unidade.getNome() + " criada ao lado da
cidade " + nomeCidade + " em (" + novaX + ", " + novaY + ").");
                        return true;
                    }
                }
            }
            throw new IllegalArgumentException("Não há espaço disponível ao redor da
cidade '" + nomeCidade + "' para adicionar a unidade.");
        }
    }
    throw new IllegalArgumentException("Cidade com o nome '" + nomeCidade + "' não
encontrada.");

  } catch (IllegalArgumentException e) {
    System.out.println("Erro ao adicionar unidade: " + e.getMessage());
    return false;
  } catch (Exception e) {
    System.out.println("Erro inesperado: " + e.getMessage());
    return false;
  }
}
/**
 * Verifica se uma coordenada é válida no mapa.
 * @param x a coordenada X.
 * @param y a coordenada Y.
 * @return true se a coordenada é válida, false caso contrário.
 */
public boolean coordenadaValida(int x, int y) {
  return x >= 0 && x < mapa[0].length && y >= 0 && y < mapa.length;
}
/**
 * Obtém o terreno em uma posição específica do mapa.
 * @param x a coordenada X da posição.
 * @param y a coordenada Y da posição.
```

```
     * @param raio o raio de busca.
     * @return a lista de terrenos no raio especificado.
     */
    public List<Terrenos> obterTerrenosNoRaio(int x, int y, int raio) {
        List<Terrenos> terrenosNoRaio = new ArrayList<>();

        for (int dx = -raio; dx <= raio; dx++) {
            for (int dy = -raio; dy <= raio; dy++) {
                int posX = normalizarCoordenada(x + dx, mapa[0].length);
                int posY = normalizarCoordenada(y + dy, mapa.length);

                if (coordenadaValida(posX, posY)) {
                    terrenosNoRaio.add(mapa[posY][posX]);
                }
            }
        }

        return terrenosNoRaio;
    }

    /**
     * Define as coordenadas de um terreno no mapa.
     * @param x a coordenada X.
     * @param y a coordenada Y.
     * @param terreno o terreno a ser definido.
     */
    public void definirCoordenadas(int x, int y, Terrenos terreno) {
        if (coordenadaValida(x, y)) {
            mapa[x][y] = terreno;
            System.out.println("Coordenadas (" + x + ", " + y + ") definidas como: " +
terreno.getClass().getSimpleName());
        } else {
            System.out.println("Erro: Coordenadas (" + x + ", " + y + ") inválidas!");
        }
    }

    /**
     * Obtém o terreno em uma posição específica do mapa.
     * @param unidade unidade que vai ser movida
     */
    public void removerUnidadeDoMapa(Unidade unidade) {
        if (unidade == null) return;
        int x = unidade.getPosicaoX();
        int y = unidade.getPosicaoY();
```

```
        if (x < 0 || x >= mapa.length || y < 0 || y >= mapa[0].length) {
            System.out.println("Posição inválida no mapa.");
            return;
        }
        if (mapa[x][y] instanceof Terrenos) {
            Terrenos terrenoAtual = mapa[x][y];
            if (terrenoAtual.temUnidade() && terrenoAtual.getUnidade() == unidade) {
                mapa[x][y] = new TerrenoVazio(x, y);
                System.out.println("Unidade removida do mapa na posição (" + x + ", " + y + ").");
            } else {
                System.out.println("A posição (" + x + ", " + y + ") não contém a unidade
especificada.");
            }
        } else {
            System.out.println("A posição (" + x + ", " + y + ") não é válida para remoção de
unidade.");
        }
    }

    /**
     * Adiciona uma unidade ao mapa.
     * @param unidade a unidade a ser adicionada.
     * @param nomeCidade o nome da cidade onde a unidade será adicionada.
     * @return true se a unidade foi adicionada com sucesso, false caso contrário.
     */
    public boolean adicionarUnidade(Unidade unidade, String nomeCidade) {
        try {
            for (int i = 0; i < posicoesCidades.size(); i++) {
                Cidades cidade = listcidades.get(i);
                if (cidade.getNome().equals(nomeCidade)) {
                    int cidadeX = posicoesCidades.get(i)[0];
                    int cidadeY = posicoesCidades.get(i)[1];

                    for (int dx = -1; dx <= 1; dx++) {
                        for (int dy = -1; dy <= 1; dy++) {
                            if (dx == 0 && dy == 0) continue;

                            int novaX = normalizarCoordenada(cidadeX + dx, mapa[0].length);
                            int novaY = normalizarCoordenada(cidadeY + dy, mapa.length);

                            if (mapa[novaY][novaX] instanceof TerrenoVazio) {
                                unidade.setPosicao(novaX, novaY);
                                mapa[novaY][novaX] = new TerrenoUnidade(unidade,novaY,novaX);
```

```
                        System.out.println("Unidade " + unidade.getNome() + " criada ao lado da cidade " + nomeCidade + " em (" + novaX + ", " + novaY + ").");
                        return true;
                    }
                }
            }

            throw new IllegalArgumentException("Não há espaço disponível ao redor da cidade '" + nomeCidade + "' para adicionar a unidade.");
        }
    }

    throw new IllegalArgumentException("Cidade com o nome '" + nomeCidade + "' não encontrada.");

  } catch (IllegalArgumentException e) {
    System.out.println("Erro ao adicionar unidade: " + e.getMessage());
    return false;
  } catch (Exception e) {
    System.out.println("Erro inesperado: " + e.getMessage());
    return false;
  }
}
/**
 * Normaliza uma coordenada para o tamanho do mapa.
 * @param coordenada a coordenada a ser normalizada.
 * @param limite o tamanho do mapa.
 * @return a coordenada normalizada.
 */
private int normalizarCoordenada(int coordenada, int limite) {
  if (coordenada < 0) {
    return (coordenada % limite + limite) % limite;
  }
  return coordenada % limite;
}
/**
 * Obtém o terreno em uma posição específica do mapa.
 * @param x a coordenada X da posição.
 * @param y a coordenada Y da posição.
 * @return o terreno na posição especificada.
 */
public Terrenos getCoordenadas(int x, int y) {
  x = normalizarCoordenada(x, this.mapa[0].length);
  y = normalizarCoordenada(y, this.mapa.length);
```

```
        return this.mapa[y][x];
    }
    /**
     * Obtém o terreno em uma posição específica do mapa.
     * @return o terreno na posição especificada.
     */
    public Terrenos[][] getMapa() {
        return this.mapa;
    }
    public boolean existeCidadeProxima(int x, int y) {
        int distanciaMinima = 2; // Distância minima entre cidades

        for (Cidades cidade : listcidades) {
            int distanciaX = Math.abs(cidade.getPosX() - x);
            int distanciaY = Math.abs(cidade.getPosY() - y);

            if (distanciaX <= distanciaMinima && distanciaY <= distanciaMinima) {
                return true;
            }
        }
        return false;
    }
    public boolean criarCidade2(int x, int y, Player player) {
        try {
            // Verifica se a posição é um colono
            Terrenos terrenoAtual = mapa[y][x];
            if (terrenoAtual instanceof TerrenoUnidade && ((TerrenoUnidade)
terrenoAtual).getUnidade() instanceof Colonos) {
                // Remove o colono (transforma o terreno ocupado por ele em vazio)
                mapa[y][x] = new TerrenoVazio(x, y);
            } else if (terrenoAtual instanceof Agua || !(terrenoAtual instanceof TerrenoVazio)) {
                throw new IllegalArgumentException("O terreno não é adequado para uma
cidade.");
            }

            // Verifica se já existe uma cidade na posição (x, y)
            for (int[] pos : posicoesCidades) {
                if (pos[0] == x && pos[1] == y) {
                    throw new IllegalArgumentException("Já existe uma cidade nesta posição.");
                }
            }

            // Cria a nova cidade
            Cidades novaCidade = new Cidades(0, 0, 0, x, y, this, player);
```

```
            novaCidade.setNome("Cidade" + (listcidades.size() + 1));

            // Atualiza o mapa para refletir a cidade
            mapa[y][x] = new TerrenoCidade(novaCidade, x, y);

            // Adiciona a cidade à lista de cidades do player
            player.adicionarCidade(novaCidade);

            // Registra a posição da cidade
            posicoesCidades.add(new int[]{x, y});

            System.out.println("Cidade " + novaCidade.getNome() + " criada em (" + x + ", " + y
+ ")");
            return true;

        } catch (IllegalArgumentException e) {
            System.out.println("Erro ao criar cidade: " + e.getMessage());
            return false;
        }
    }
}
```

Classe Menus

```
package com.mycompany.projeto2;
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to
change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this
template
 */


import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;


public class Menus {
    /**
     * Imprime o menu principal do jogo.
     * @return String com o menu principal.
```

```
     */
    public String ImprimirMenu() {
        StringBuilder menu = new StringBuilder();
        menu.append("\n=== MENU PRINCIPAL ===\n");
        menu.append("1. Mover uma unidade/criar cidade\n");
        menu.append("2. Atacar com uma unidade\n");
        menu.append("3. Gerir cidade\n");
        menu.append("4. Ver o mapa\n");
        menu.append("5. Encerrar turno\n");
        menu.append("Escolha uma opção: ");
        return menu.toString();
    }
    /**
     * Imprime o menu de criação de cidade.
     * @param scanner Scanner para leitura de dados do usuário.
     * @param player Jogador que está criando a cidade.
     * @param mapa Mapa do jogo.
     */
    public void menuUnidades(Scanner scanner, Player player, Mapa mapa) {
        while (true) {
            System.out.println("\nMenu de Unidades:");
            List<Unidade> unidades = player.getUnidades();
            if (unidades.isEmpty()) {
                System.out.println("Você não tem unidades para mover.");
                break;
            }

            System.out.println("Unidades disponíveis para mover:");
            for (int i = 0; i < unidades.size(); i++) {
                System.out.println((i + 1) + ". " + unidades.get(i).getNome() + " (Posição: "
                        + unidades.get(i).getPosicaoX() + ", " + unidades.get(i).getPosicaoY() +
")");
            }

            System.out.print("Escolha uma unidade para mover (1-" + unidades.size() + "):
");
            int escolha = scanner.nextInt() - 1;

            if (escolha < 0 || escolha >= unidades.size()) {
                System.out.println("Escolha inválida.");
```

```
            continue;
        }

        Unidade unidadeEscolhida = unidades.get(escolha);
        System.out.println("Você escolheu a unidade: " +
unidadeEscolhida.getNome());

        System.out.print("Digite a nova posição X (0-9): ");
        int novaPosicaoX = scanner.nextInt();

        System.out.print("Digite a nova posição Y (0-9): ");
        int novaPosicaoY = scanner.nextInt();
            try {
                mapa.moverUnidade(unidadeEscolhida, novaPosicaoX, novaPosicaoY);
                System.out.println("Movimento realizado com sucesso!");

                if (unidadeEscolhida instanceof Colonos) {
                    System.out.print("Deseja fundar uma cidade nesta posição? (sim/nao):
");
                    String resposta = scanner.next();
                    if (resposta.toLowerCase().startsWith("s")) {
                        String nomeCidade = scanner.nextLine();

                        Colonos colono = (Colonos) unidadeEscolhida;
                        boolean sucesso = colono.fundarCidade(player, nomeCidade, mapa,
novaPosicaoX, novaPosicaoY);

                        if (!sucesso) {
                            System.out.println("Não foi possível fundar a cidade.");
                        }
                        return;
                    }
                }
            } catch (IllegalArgumentException e) {
                System.out.println("Falha no movimento: " + e.getMessage());
            }
        System.out.print("Deseja mover outra unidade? (sim/nao): ");
        char resposta = scanner.next().charAt(0);
        if (resposta != 's' && resposta != 'S') {
            break;
```

```
            }
        }
    }

    /**
     * Selecionar uma unidade para atacar.
     * @param scanner Scanner para leitura de dados do usuário.
     * @param player Jogador que está criando a cidade.
     * @return Unidade selecionada para atacar.
     */
    private Unidade selecionarUnidade(Scanner scanner, Player player) {
        System.out.println("Unidades disponíveis para atacar:");

        List<Unidade> unidades = player.getUnidades();
        if (unidades.isEmpty()) {
            System.out.println("Você não tem unidades para atacar.");
            return null;
        }
        for (int i = 0; i < unidades.size(); i++) {
            Unidade unidade = unidades.get(i);
            System.out.println(i + ": " + unidade.getNome() + " - Tipo: Militar - Coordenadas: (" + unidade.getPosicaoX() + ", " + unidade.getPosicaoY() + ")");
        }

        System.out.println("Selecione uma unidade para atacar (insira o índice da unidade):");
        int indiceUnidade = scanner.nextInt();

        if (indiceUnidade < 0 || indiceUnidade >= unidades.size()) {
            System.out.println("Unidade inválida!");
            return null;
        }

        return unidades.get(indiceUnidade);
    }


    /**
     * Obter inimigos no raio de ataque da unidade.
     * @param unidade Unidade que está a atacar.
```

```
     * @param mapa Mapa do jogo.
     * @param raio Raio de ataque da unidade.
     * @param jogadorAtual Jogador que está a atacar.
     * @return Lista de inimigos no raio de ataque da unidade.
     */
    private List<Object> obterInimigosNoRaio(Unidade unidade, Mapa mapa, int raio,
Player jogadorAtual) {
        List<Object> inimigos = new ArrayList<>();
        int unidadeX = unidade.getPosicaoX();
        int unidadeY = unidade.getPosicaoY();
        for (int dx = -raio; dx <= raio; dx++) {
            for (int dy = -raio; dy <= raio; dy++) {
                int posX = unidadeX + dx;
                int posY = unidadeY + dy;
                if (mapa.coordenadaValida(posX, posY)) {
                    Terrenos terreno = mapa.getCoordenadas(posX, posY);
                        if (terreno instanceof TerrenoUnidade) {
                        Unidade inimiga = ((TerrenoUnidade) terreno).getUnidade();
                        if (inimiga != null && !jogadorAtual.getUnidades().contains(inimiga))
{
                            inimigos.add(inimiga);
                        }
                    }

                    if (terreno instanceof TerrenoCidade) {
                        Cidades cidade = ((TerrenoCidade) terreno).getCidade();
                        if (cidade != null && !jogadorAtual.getCidades().contains(cidade)) {
                            inimigos.add(cidade);
                        }
                    }
                }
            }
        }
        return inimigos;
    }


    /**
     * Exibir alvos disponíveis para ataque.
     * @param unidadesAlvo Lista de unidades inimigas disponíveis.
```

```
     * @param cidadesAlvo Lista de cidades inimigas disponíveis.
     */
    private void exibirAlvosDisponiveis(List<Unidade> unidadesAlvo, List<Cidades>
cidadesAlvo) {
        System.out.println("Alvos disponíveis para ataque:");

        if (!unidadesAlvo.isEmpty()) {
            System.out.println("Unidades inimigas:");
            for (int i = 0; i < unidadesAlvo.size(); i++) {
                System.out.println(i + ": " + unidadesAlvo.get(i).toString());
            }
        } else {
            System.out.println("Nenhuma unidade inimiga disponível.");
        }

        if (!cidadesAlvo.isEmpty()) {
            System.out.println("Cidades inimigas:");
            for (int i = 0; i < cidadesAlvo.size(); i++) {
                System.out.println((i + unidadesAlvo.size()) + ": " +
cidadesAlvo.get(i).getNome());
            }
        } else {
            System.out.println("Nenhuma cidade disponível.");
        }
    }


    /**
     * Selecionar e atacar um alvo, unidade ou cidade.
     * @param scanner Scanner para leitura de dados do usuário.
     * @param unidade Unidade que está a atacar.
     * @param unidadesAlvo Lista de unidades inimigas disponíveis.
     * @param cidadesAlvo Lista de cidades inimigas disponíveis.
     */
    private void selecionarEAtacarAlvo(Scanner scanner, Unidade unidade,
List<Unidade> unidadesAlvo, List<Cidades> cidadesAlvo) {
        System.out.println("Escolha o índice do alvo:");
        int indiceAlvo = scanner.nextInt();
        if (indiceAlvo < unidadesAlvo.size()) {
            Unidade alvoUnidade = unidadesAlvo.get(indiceAlvo);
```

```
            System.out.println("Atacando unidade inimiga: " + alvoUnidade.getNome());
            unidade.attack(alvoUnidade);
        } else if (indiceAlvo - unidadesAlvo.size() < cidadesAlvo.size()) {
            Cidades alvoCidade = cidadesAlvo.get(indiceAlvo - unidadesAlvo.size());
            System.out.println("Atacando a cidade: " + alvoCidade.getNome());
            unidade.attackCity(alvoCidade);
        } else {
            System.out.println("Alvo inválido.");
        }
    }
```

```
    /**
     * Menu para gerenciar cidades.
     * @param scanner Scanner para leitura de dados do usuário.
     * @param player Jogador que está gerenciando a cidade.
     * @param mapa Mapa do jogo.
     */
    public void menuGerirCidades(Scanner scanner, Player player, Mapa mapa) {
        System.out.println("Selecione uma cidade para gerenciar:");

        List<Cidades> cidades = player.getCidades();
        if (cidades.isEmpty()) {
            System.out.println("Você não possui cidades para gerenciar.");
            return;
        }

        for (int i = 0; i < cidades.size(); i++) {
            System.out.println((i + 1) + ". " + cidades.get(i).getNome());
        }
        System.out.print("Escolha o número da cidade: ");
        int escolhaCidade = scanner.nextInt();
        scanner.nextLine();

        if (escolhaCidade < 1 || escolhaCidade > cidades.size()) {
            System.out.println("Escolha inválida!");
            return;
        }
```

```
        Cidades cidadeSelecionada = cidades.get(escolhaCidade - 1);
        boolean gerenciando = true;

        while (gerenciando) {
            System.out.println("\n== GERENCIAR CIDADE: " +
cidadeSelecionada.getNome() + " ==");
            System.out.println("1. Visualizar informações da cidade");
            System.out.println("2. Criar Unidade Militar");
            System.out.println("3. Criar Unidade Colono");
            System.out.println("4. Criar Unidade Construtor");
            System.out.println("5. Gerenciar terrenos");
            System.out.println("6. Voltar ao menu principal");
            System.out.print("Escolha uma opção: ");

            int escolha = scanner.nextInt();
            scanner.nextLine();
            if (mapa == null) {
        return;
    }

            switch (escolha) {
                case 1:
                    System.out.println(cidadeSelecionada.toString());
                    break;
                case 2:

cidadeSelecionada.criarUnidade(player,"Militar",cidadeSelecionada.getNome(),mapa)
;
                    break;
                case 3:

cidadeSelecionada.criarUnidade(player,"Colono",cidadeSelecionada.getNome(),mapa
);
                    break;
                case 4:

cidadeSelecionada.criarUnidade(player,"Construtor",cidadeSelecionada.getNome(),m
apa);
                    break;
                case 5:
```

```
                menuGerirTerrenos(scanner, player, mapa);
                break;
            case 6:
                gerenciando = false;
                System.out.println("Voltando ao menu principal...");
                break;
            default:
                System.out.println("Opção inválida. Tente novamente.");
        }
    }
}

/**
 * Menu para gerenciar terrenos de uma cidade.
 * @param scanner Scanner para leitura de dados do usuário.
 * @param player Jogador que está gerenciando a cidade.
 * @param mapa Mapa do jogo.
 */
public void menuGerirTerrenos(Scanner scanner, Player player, Mapa mapa) {
    for (Cidades cidade : player.getCidades()) {
        System.out.println("Gerenciando terrenos para a cidade: " + cidade.getNome());
        boolean alocando = true;
        while (alocando) {
            System.out.println("1. Alocar cidadão a um terreno");
            System.out.println("2. Desalocar cidadão de um terreno");
            System.out.println("3. Voltar");
            int escolha = scanner.nextInt();
            switch (escolha) {
                case 1:
                exibirCoordenadas(cidade.getPosX(),cidade.getPosY(), mapa);
                System.out.println("Coordenadas da cidade: (" + cidade.getPosX() + ",
" + cidade.getPosY() + ")");
                System.out.println("Digite as coordenadas (x y) do terreno para alocar
cidadão:");
                int x = scanner.nextInt();
                int y = scanner.nextInt();
                Terrenos terreno = mapa.getCoordenadas(x, y);
                if (terreno == null) {
                    System.out.println("Coordenadas inválidas ou terreno não
encontrado.");
```

```
                    } else {
                        cidade.alocarCidadao(terreno, mapa);
                    }
                    break;
                case 2:
                    System.out.println("Digite as coordenadas (x y) do terreno para desalocar cidadão:");
                    x = scanner.nextInt();
                    y = scanner.nextInt();
                    terreno = mapa.getCoordenadas(x, y);
                    cidade.desalocarCidadão(terreno);
                    break;
                case 3:
                    alocando = false;
                    break;
                default:
                    System.out.println("Escolha inválida, tente novamente.");
            }
        }
    }
}
/**
 * Exibir coordenadas ao redor da cidade.
 * @param x Coordenada X da cidade.
 * @param y Coordenada Y da cidade.
 * @param mapa Mapa do jogo.
 */
public void exibirCoordenadas(int x, int y, Mapa mapa) {
    Terrenos[][] matrizMapa = mapa.getMapa();
    int tamanhoMapa = matrizMapa.length;
    int inicioX = Math.max(0, x - 3);
    int fimX = Math.min(tamanhoMapa - 1, x + 3);
    int inicioY = Math.max(0, y - 3);
    int fimY = Math.min(tamanhoMapa - 1, y + 3);
    System.out.println("Coordenadas ao redor da cidade:");
    for (int i = inicioX; i <= fimX; i++) {
        for (int j = inicioY; j <= fimY; j++) {
            Terrenos terreno = matrizMapa[i][j];
            if (terreno != null) {
                if (terreno instanceof Floresta) {
```

```
                System.out.println("Floresta encontrada em (" + j + ", " + i + ")");
            } else if (terreno instanceof Pedreira) {
                System.out.println("Pedreira encontrada em (" + j + ", " + i + ")");
            }
        } else {
            System.out.println("Espaço vazio em (" + j + ", " + i + ")");
        }
    }
  }

  /**
   * Menu para criar uma cidade.
   * @param scanner Scanner para leitura de dados do usuário.
   * @param player Jogador que está criando a cidade.
   * @return true se a cidade foi criada com sucesso, false caso contrário.
   */
  public boolean encerrarTurno(Scanner scanner, Player player) {
      System.out.println("Tem certeza que deseja encerrar seu turno? (1 - Sim, 2 - Não)");
      int confirmacao = scanner.nextInt();
      return confirmacao == 1;
  }


  /**
   * Menu para atacar com uma unidade.
   * @param scanner Scanner para leitura de dados do usuário.
   * @param player Jogador que está a atacar.
   * @param mapa Mapa do jogo.
   */
  public void menuAtacarUnidade(Scanner scanner, Player player, Mapa mapa) {
      System.out.println("=== Atacar com uma Unidade ===");
      Unidade unidade = selecionarUnidade(scanner, player);
      if (unidade == null) return;
      exibirUnidadesComCoordenadas(player);
      List<Object> inimigosAlvo = obterInimigosNoRaio(unidade, mapa, 1, player);
      List<Unidade> unidadesAlvo = new ArrayList<>();
      List<Cidades> cidadesAlvo = new ArrayList<>();
```

```
        for (Object inimigo : inimigosAlvo) {
            if (inimigo instanceof Unidade) {
                unidadesAlvo.add((Unidade) inimigo);
            } else if (inimigo instanceof Cidades) {
                cidadesAlvo.add((Cidades) inimigo);
            }
        }
        exibirAlvosDisponiveis(unidadesAlvo, cidadesAlvo);
        selecionarEAtacarAlvo(scanner, unidade, unidadesAlvo, cidadesAlvo);
    }



    /**
     * Exibir unidades e suas coordenadas.
     * @param player Jogador que tem a unidade.
     */
    private void exibirUnidadesComCoordenadas(Player player) {
        System.out.println("Unidades e suas coordenadas para o jogador: " +
player.getNome());
        for (Unidade unidade : player.getUnidades()) {
            int x = unidade.getPosicaoX();
            int y = unidade.getPosicaoY();
            System.out.println(unidade.getNome() + " - Coordenadas: (" + x + ", " + y +
")");
        }
    }

    /**
     * Menu para imprimir a legenda do mapa
     * @return String com a legenda do mapa.
     */
    public String imprimeLegenda(){
        StringBuilder legenda = new StringBuilder();
        legenda.append("\n=== LEGENDA ===\n");
        legenda.append("C - Cidade\n");
        legenda.append("A - Água\n");
        legenda.append("F - Floresta\n");
        legenda.append("P - Pedreira\n");
        legenda.append("T - Terreno Trabalhado\n");
```

```
        legenda.append("M - Unidade Militar\n");
        legenda.append("B - Construtores\n");
        legenda.append("H - Colonos\n");
        return legenda.toString();
    }
}
```

Classe Militares

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this
license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package com.mycompany.projeto2;
/**
 * Classe que representa os militares.
 */
public class Militares extends Unidade{
    private int ataque;
    private int vida;
    private int customanutencao;
    private int danocombate = 10;
    /**
     * Construtor da classe Militares.
     * @param nome O nome da unidade.
     */
    public Militares(String nome) {
        super(nome,15);
        this.ataque = 10;
        this.vida = 5;
    }

    // Getters
    public int getAtaque() {
        return ataque;
    }
    // Getters
    public int getVida() {
        return vida;
    }
    /**
     * Método para atacar uma cidade
     * @param cidade A cidade a ser atacada.
     */
```

```
@Override
public void attackCity(Cidades cidade) {
    System.out.println(getNome() + " está a atacar a cidade " + cidade.getNome());
    int dano = danocombate;
    cidade.takeDamage(dano);
}
/**
 * Método para reduzir a vida da unidade quando ela recebe dano.
 * @param dano O valor de dano que a unidade sofre.
 */
@Override
public void takeDamage(int dano) {
    vida -= dano;
    if (vida <= 0) {
        System.out.println(getNome() + " foi destruída.");
    } else {
        System.out.println(getNome() + " sofreu dano. Vida restante: " + vida);
    }
}

/**
 * Método para atacar outra unidade.
 * @param inimigo A unidade a ser atacada.
 */
@Override
public void attack(Unidade inimigo) {
    if (inimigo instanceof Militares) {
        Militares target = (Militares) inimigo;
        System.out.println(getNome() + " atacou " + target.getNome());
        int damage = this.ataque - (target.ataque / 2);
        target.takeDamage(damage);
    } else {
        System.out.println("Alvo inválido para ataque.");
    }
}

/**
 * Método para receber dano de outra unidade.
 * @param dano O valor de dano que a unidade sofre.
 */
public void receberDano(int dano) {
    vida -= dano;
    if (vida <= 0) {
        System.out.println(getNome() + " foi destruído.");
```

```
        }
    }

    /**
     * Método para executar a ação da unidade.
     */
    @Override
    public void performAction() {
        System.out.println(getNome() + " está em posição defensiva.");
    }

}
```

Classe Ouro

```
**
 * Classe que representa o ouro.
 */
public class Ouro  {
    private int quantidade;
    private int producaoPorTurno;

    public Ouro(int quantidadeInicial, int producaoPorTurno) {
        this.quantidade = quantidadeInicial;
        this.producaoPorTurno = producaoPorTurno;
    }



    /**
     * Método para adicionar ouro
     * @param quantidade A quantidade de ouro a ser adicionada ao tesouro.
     */
    public void adicionarOuro(int quantidade) {
        if (quantidade > 0) {
            this.quantidade += quantidade;
        }
    }

    /**
     * Método para remover ouro do tesouro
     * @param quantidade A quantidade de ouro a ser removida do tesouro.
     * @throws IllegalArgumentException Se a quantidade de ouro a ser removida for maior que a quantidade de ouro no tesouro.
     */
```

```
    public void removerOuro(int quantidade) throws IllegalArgumentException {
        if (quantidade > this.quantidade) {
            throw new IllegalArgumentException("Não há ouro suficiente no tesouro.");
        }
        this.quantidade -= quantidade;
    }

    /**
     * Método para obter a quantidade atual de ouro
     * @return A quantidade de ouro no tesouro.
     */
    public int obterQuantidade() {
        return this.quantidade;
    }
    /**
     * Método para produzir ouro
     */
    public void produzirOuro() {
        this.quantidade += this.producaoPorTurno;
    }
    /**
     * Método para exibir a quantidade de ouro
     */
    @Override
    public String toString() {
        return "Quantidade de ouro no tesouro: " + this.quantidade;
    }
    /**
     * Método para obter a produção de ouro por turno
     * @param incremento A quantidade de ouro a ser adicionada à produção por turno.
     */
    public void aumentarProducao(int incremento) {
        if (incremento > 0) {
            this.producaoPorTurno += incremento;
        } else {
            System.out.println("Valor de incremento inválido. O aumento deve ser maior que 0.");
        }
    }

}
```

Classe Pedreira

```
/**
```

```
 * Classe que representa uma pedreira.
 */
public class Pedreira extends Terrenos implements Recursos {
    private int ouro = 0;
    private int producaoPorTurno = 20;
    private int x;
    private int y;

    public Pedreira(int x, int y) {
        super(4);
        this.x = x;
        this.y = y;
    }
    /**
     * Obtém a quantidade de comida coletada da pedreira.
     */
    @Override
    public int getComida() {
        return 0;
    }
    /**
     * Obtém a quantidade de produção gerada pela pedreira.
     */
    @Override
    public int getProducao() {
        return 0;
    }
    /**
     * Obtém a quantidade de ouro gerada pela pedreira.
     */
    @Override
    public int getOuro() {
        return ouro;
    }
    /**
     * Obtém a quantidade de produção gerada pela pedreira por turno.
     */
    @Override
    public int getProducaoPorTurno() {
        return producaoPorTurno;
    }
    /**
     * Método para exibir a quantidade de ouro
     */
```

```
    @Override
    public String toString() {
        return "P";
    }
    /**
     * Método para coletar ouro
     */
    public void coletarOuro() {
        ouro += producaoPorTurno;
    }
    /**
     * Método para obter a coordenada x da pedreira
     */
    @Override
    public int getX() {
        return this.x;
    }
    /**
     * Método para obter a coordenada Y da pedreira
     */
    @Override
    public int getY() {
        return this.y;
    }

}
```

Classe Player

```
import java.util.ArrayList;
import java.util.List;

/**
 * Classe que representa o jogador.
 */
public class Player {
    private String nome;
    private List<Cidades> cidades;
    private List<Unidade> unidades;
    private int tesouro;

    public Player(String nome) {
        this.nome = nome;
        this.cidades = new ArrayList<>();
        this.unidades = new ArrayList<>();
        this.tesouro = 0;
```

```
    }
    /**
     * Adiciona uma cidade ao jogador.
     * @param cidade A cidade a ser adicionada.
     */
    public void adicionarCidade(Cidades cidade) {
        cidades.add(cidade);
        System.out.println("Cidade " + cidade.getNome() + " adicionada ao Player " + nome);
    }

    /**
     * Adiciona uma unidade ao jogador.
     * @param unidade A unidade a ser adicionada.
     */
    public void adicionarUnidade(Unidade unidade) {
        unidades.add(unidade);
        System.out.println("Unidade " + unidade.getNome() + " adicionada ao Player " + nome);
    }

    /**
     * Calcula o tesouro total do jogador.
     */
    public void calcularTesouro() {
        for (Cidades cidade : cidades) {
            tesouro += cidade.getOuro();
        }
        System.out.println("Tesouro total do Player " + nome + ": " + tesouro);
    }

    /**
     * Método para pagar a manutenção das unidades.
     */
    public void pagarManutencao() {
        int custoTotal = 0;
        for (Unidade unidade : unidades) {
            custoTotal += unidade.getCusto();
        }

        if (tesouro >= custoTotal) {
            tesouro -= custoTotal;
            System.out.println("Manutenção paga. Tesouro atual: " + tesouro);
        } else {
            System.out.println("Tesouro insuficiente! Unidades podem ser desmontadas ou penalidades aplicadas.");
```

```
        }
    }

    /**
     * Lista as unidades do jogador.
     */
    public void listarUnidades() {
        if (unidades.isEmpty()) {
            System.out.println("Nenhuma unidade disponível.");
            return;
        }
        System.out.println("===== Unidades =====");
        for (Unidade unidade : unidades) {
            System.out.println(unidade.getNome() + " - Posição (" + unidade.getPosicaoX() + ", "
+ unidade.getPosicaoY() + ")");
        }
    }

    /**
     * Lista as cidades do jogador.
     */
    public void listarCidades() {
        if (cidades.isEmpty()) {
            System.out.println("Nenhuma cidade disponível.");
            return;
        }
        System.out.println("===== Cidades =====");
        for (Cidades cidade : cidades) {
            System.out.println(cidade.getNome() + " - População: " + cidade.getPopulacao());
        }
    }

    /**
     * O método tem como propósito permitir que uma unidade ataque outra
     * @param indiceUnidade O índice da unidade a ser movida.
     * @param unidadeAlvo A unidade alvo para a movimentação.
     */
    public void atacarComUnidade(int indiceUnidade, Unidade unidadeAlvo) {
        if (indiceUnidade >= 0 && indiceUnidade < unidades.size()) {
            Unidade unidade = unidades.get(indiceUnidade);
            unidade.attack(unidade);
        } else {
            System.out.println("Unidade não encontrada.");
        }
```

```
}

/**
 * O método retorna uma lista de objetos do tipo Unidade.
 * @return A lista de cidades do jogador.
 */
public List<Unidade> getUnidades() {
   return unidades;
}

/**
 * O método retorna uma lista de objetos do tipo Cidades.
 * @return A lista de cidades do jogador.
 */
public List<Cidades> getCidades() {
   return cidades;
}

/**
 * O método retorna o nome do jogador.
 * @return O nome do jogador.
 */
public String getNome() {
   return nome;
}

/**
 * O método adiciona uma unidade à lista de unidades do jogador.
 * @param unidade A unidade a ser adicionada.
 */
public void addUnidade(Unidade unidade) {
   unidades.add(unidade);
}

/**
 * O método remove uma unidade da lista de unidades do jogador.
 * @param unidade A unidade a ser removida.
 * @return true se a unidade foi removida com sucesso, false caso contrário.
 */
public boolean removerUnidade(Unidade unidade) {
   if (unidade == null) {
      System.out.println("Unidade inválida.");
      return false;
   }
```

```
        if (unidades.remove(unidade)) {
            System.out.println("Unidade " + unidade.getNome() + " removida com sucesso.");
            return true;
        } else {
            System.out.println("Unidade não encontrada na cidade.");
            return false;
        }
    }
    public int calcularOuroTotal() {
        int ouroTotal = 0;
        for (Cidades cidade : cidades) {
            ouroTotal += cidade.getOuro();
        }
        return ouroTotal;
    }

    public boolean verificarVitoria() {
        int ouroTotal = calcularOuroTotal();
        if (ouroTotal >= 320) {
            System.out.println("VITÓRIA! " + nome + " alcançou 320 de ouro!");
            return true;
        }
        return false;
    }
}
```

Classe Producao

```
public class Producao {
    private static final int PRODUCAO_FIXA = 10;
    private int producaoPorTurno;
    private int producaoUtilizada;

    /**
     * Construtor para inicializar a produção por turno.
     *
     * @param producaoPorTurno a quantidade de produção gerada a cada turno.
     */
    public Producao(int producaoPorTurno) {
        this.producaoPorTurno = producaoPorTurno;
        this.producaoUtilizada=0;
    }

    /**
```

```
     * Reinicia a produção no início de um novo turno.
     */
    public void reiniciarProducao() {
        this.producaoUtilizada = 0;
    }

    /**
     * Tenta utilizar uma quantidade de produção disponível.
     *
     * @param quantidade a quantidade de produção que se deseja consumir.
     * @return true se a produção foi consumida com sucesso; false caso contrário.
     */
    public boolean usarProducao(int quantidade) {
        if (quantidade <= getProducaoRestante()) {
            producaoUtilizada += quantidade;
            System.out.printf("Produção de %d utilizada. Restante: %d.\n", quantidade,
getProducaoRestante());
            return true;
        } else {
            System.out.println("Produção insuficiente para atender à demanda.");
            return false;
        }
    }

    /**
     * Obtém a quantidade restante de produção neste turno.
     *
     * @return a produção restante.
     */
    public int getProducaoRestante() {
        return PRODUCAO_FIXA - producaoUtilizada;
    }

    /**
     * Atualiza a quantidade de produção gerada por turno.
     *
     * @param producaoPorTurno nova quantidade de produção por turno.
     */
    public void setProducaoPorTurno(int producaoPorTurno) {
        this.producaoPorTurno = producaoPorTurno;
    }

    /**
     * Retorna uma representação textual da produção disponível.
```

```
     *
     * @return texto com a produção disponível.
     */
    @Override
    public String toString() {
        return "Produção disponível: " + getProducaoRestante();
    }
}
```

Classe Projeto2(Main)

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 */

package com.mycompany.projeto2;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
/**
 *
 * @author nelso
 */
public class Projeto2 {

    private Floresta floresta;
    private Pedreira pedreira;
    private Menus menus;



    /*
     * Método para imprimir o mapa
     */
    public static void imprimirMapa(char[][] mapa) {
        int comprimento = mapa.length;
        int largura = mapa[0].length;

        System.out.print("    ");
        for (int col = 0; col < largura; col++) {
            System.out.printf("%3dx ", col);
        }
```

```
    System.out.println();

    for (int linha = 0; linha < comprimento; linha++) {
        System.out.printf("%3dy ", linha);
        for (int col = 0; col < largura; col++) {
            System.out.print(" " + mapa[linha][col] + "   ");
        }
        System.out.println();
    }

}


/*
 * Método principal
 */
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int largura = 20;
    int comprimento = 20;
    Menus menus = new Menus();
    Mapa mapa = new Mapa(largura, comprimento);
    boolean cidadeCriada = false;
    List<Player> players = new ArrayList<>();
    Terrenos[][] terrenos = mapa.getMapa();
    System.out.println("Quantos jogadores irao participar?");
    int numJogadores = scanner.nextInt();
    scanner.nextLine();

    for (int i = 0; i < numJogadores; i++) {
        System.out.println("Nome do jogador " + (i + 1) + ": ");
        String nome = scanner.nextLine();
        players.add(new Player(nome));
    }

    boolean jogoAtivo = true;
    int turno = 1;

    System.out.println("\n=== CRIAÇÃO DE CIDADES INICIAIS ===");
    for (Player player : players) {
```

```
        System.out.println("\n### " + player.getNome() + ", escolha uma posição para sua primeira cidade. ###");
        System.out.println(menus.imprimeLegenda());
        mapa.exibirMapa();
        cidadeCriada = false;
        while (!cidadeCriada) {
            System.out.println("Digite as coordenadas (x y) para criar sua cidade:");
            int x = scanner.nextInt();
            int y = scanner.nextInt();
            if (mapa.criarCidade(x, y, player)) {
                cidadeCriada = true;
            } else {
                System.out.println("Posição inválida para criar uma cidade. Tente novamente.");
            }
        }
    }
    while (jogoAtivo) {
        System.out.println("\n===== TURNO " + turno + " =====");
        if (turno > 1) {
            System.out.println("\n=== FINALIZANDO TURNO DAS CIDADES ===");
            for (Player player : players) {
                for (Cidades cidade : player.getCidades()) {
                    cidade.proximoTurno();
                    System.out.println("A cidade " + cidade.getNome() + " finalizou seu turno.");
                }
            }
        }
        for (Player player : players) {
            System.out.println("\n### Turno de " + player.getNome() + " ###");
            System.out.println(menus.imprimeLegenda());
            mapa.exibirMapa();
            boolean turnoAtivo = true;

            while (turnoAtivo) {
                System.out.println(menus.ImprimirMenu());
                int escolha = scanner.nextInt();
```

```
        switch (escolha) {
            case 1:
                menus.menuUnidades(scanner, player, mapa);
                break;

            case 2:
                menus.menuAtacarUnidade(scanner, player, mapa);
                break;

            case 3:
                menus.menuGerirCidades(scanner, player, mapa);
                break;

            case 4:
                System.out.println(menus.imprimeLegenda());
                mapa.exibirMapa();
                break;
            case 5:
                if (menus.encerrarTurno(scanner, player)) {
                    System.out.println(player.getNome() + " encerrou seu turno.");
                    turnoAtivo = false;
                } else {
                    System.out.println("Turno ainda ativo. Continue suas ações.");
                }
                break;

            default:
                System.out.println("Escolha inválida, tente novamente.");
        }
        int ouroTotal = player.calcularOuroTotal();
        System.out.println("Ouro total produzido pelas cidades: " + ouroTotal);
    }
}

System.out.println("\nFim do turno " + turno + ".");
System.out.println("Deseja continuar o jogo? (1 - Sim, 2 - Não)");
int continuar = scanner.nextInt();

if (continuar != 1) {
    jogoAtivo = false;
```

```
            }
            turno++;
        }

        System.out.println("\nJogo encerrado. Obrigado por jogar!");
        scanner.close();
    }
}
```

Classe Recursos

```
package com.mycompany.projeto2;

/**
 * Interface que representa os recursos.
 */
public interface Recursos {
    int getComida();
    int getProducao();
    int getOuro();
    int getProducaoPorTurno();
}
```

Classe TerrenoCidade

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */

 package com.mycompany.projeto2;

/**
 *
 * @author nelso
 */


public class TerrenoCidade extends Terrenos {
    private Cidades cidade;
    private int x;
```

```
private int y;

/**
 * Construtor da classe TerrenoCidade.
 * @param cidade A cidade.
 * @param x A coordenada x da cidade.
 * @param y A coordenada y da cidade.
 */
public TerrenoCidade(Cidades cidade, int x, int y) {
    super(1);
    this.cidade = cidade;
    this.x = x;
    this.y = y;
}

/*
 * Método para exibir a cidade
 */
@Override
public String toString() {
    return "C";
}
/**
 * Método para obter a coordenada x
 */
@Override
public int getX() {
    return this.x;
}
/**
 * Método para obter a coordenada y
 */
@Override
public int getY() {
    return this.y;
}
/**
 * Método para obter a cidade
 * @return A cidade
 */
```

```
    public Cidades getCidade() {
        return cidade;
    }
}
```

Classe TerrenoTrabalhado

```
package com.mycompany.projeto2;

public class TerrenoTrabalhado extends Terrenos {
    private Terrenos terrenoOriginal;
    private int quantidadeTrabalhadores;
    private String cidadeResponsavel;
    private int x;
    private int y;

    public TerrenoTrabalhado(Terrenos terrenoOriginal, String cidadeResponsavel, int quantidadeTrabalhadores, int x, int y) {
        super(1);
        this.terrenoOriginal = terrenoOriginal;
        this.cidadeResponsavel = cidadeResponsavel;
        this.quantidadeTrabalhadores = quantidadeTrabalhadores;
        this.x = x;
        this.y = y;
    }
    /**
     * Método para obter o terreno original
     * @return O terreno original
     */
    public Terrenos getTerrenoOriginal() {
        return terrenoOriginal;
    }

    /**
     * Método para obter a cidade responsável
     * @return A cidade responsável
     */
    public String getCidadeResponsavel() {
        return cidadeResponsavel;
    }

    /**
```

```
     * Método para obter a quantidade de trabalhadores
     * @return A quantidade de trabalhadores
     */
    public int getQuantidadeTrabalhadores() {
        return quantidadeTrabalhadores;
    }

    /**
     * Método para exibir o terreno trabalhado
     */
    @Override
    public String toString() {
        return terrenoOriginal.toString() + " - Trabalhadores: " + quantidadeTrabalhadores
+ " (" + cidadeResponsavel + ")";
    }

    /**
     * Método para incrementar a quantidade de trabalhadores
     */
    public void incrementarTrabalhador() {
        this.quantidadeTrabalhadores++;
    }

    /**
     * Método para obter a coordenada x
     */
    @Override
    public int getX() {
        return this.x;
    }
    /**
     * Método para obter a coordenada y
     */
    @Override
    public int getY() {
        return this.y;
    }

}
```

Classe TerrenoUnidade

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */

package com.mycompany.projeto2;

public class TerrenoUnidade extends Terrenos {
    private Unidade unidade;
    private int x;
    private int y;

    /**
     * Construtor da classe TerrenoUnidade.
     * @param unidade A unidade.
     * @param x A coordenada x da unidade.
     * @param y A coordenada y da unidade.
     */
    public TerrenoUnidade(Unidade unidade, int x, int y) {
        super(0);
        this.unidade = unidade;
        this.x = x;
        this.y = y;
    }
    /**
     * Método para obter a unidade
     */
    public Unidade getUnidade() {
        return unidade;
    }
    /**
     * Método para definir a unidade
     * @param unidade A unidade a ser definida
     */
    public void setUnidade(Unidade unidade) {
        this.unidade = unidade;
    }
```

```
    /**
     * Método para exibir a unidade
     */
    @Override
    public String toString() {
        if (unidade instanceof Militares) {
            return "M";
        } else if (unidade instanceof Construtores) {
            return "B";
        } else if (unidade instanceof Colonos) {
            return "H";
        } else {
            return "?";
        }
    }
    /**
     * Método para obter a coordenada x
     */
    @Override
    public int getX() {
        return this.x;
    }
    /**
     * Método para obter a coordenada y
     */
    @Override
    public int getY() {
        return this.y;
    }

}
```

Classe TerrenoVazio

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package com.mycompany.projeto2;
```

```
/**
 *
 * @author nelso
 */
public class TerrenoVazio extends Terrenos {
    private int x;
    private int y;


    /**
     * Construtor da classe TerrenoVazio.
     * @param x A coordenada x do terreno vazio.
     * @param y A coordenada y do terreno vazio.
     */
    public TerrenoVazio(int x, int y) {
        super(7);
        this.x = x;
        this.y = y;
    }
    /**
     * Método para obter a coordenada x
     */
    @Override
    public int getX() {
        return x;
    }
    /**
     * Método para obter a coordenada y
     */
    @Override
    public int getY() {
        return y;
    }

    /**
     * Método para exibir o terreno vazio
     */
    @Override
    public String toString() {
        return "X";
```

```
    }
}
```

Classe Terrenos

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package com.mycompany.projeto2;

/**
 *
 * @author nelso
 */
public abstract class Terrenos {

    private int custoMovimento;
    private Unidade unidade;

    /**
     * Construtor da classe Terrenos.
     * @param custoMovimento O custo de movimento.
     */
    public Terrenos(int custoMovimento) {
        this.custoMovimento = custoMovimento;
    }
    /**
     * Método para obter a coordenada x
     * @return A coordenada x
     */
    public abstract int getX();
    /**
     * Método para obter a coordenada y
     * @return A coordenada y
     */
    public abstract int getY();
    /**
     * Método para obter o custo de movimento
     * @return O custo de movimento
```

```
     */
    public int getCustoMovimento() {
        return custoMovimento;
    }

    /**
     * Método para exibir o terreno
     * @return O terreno
     */
    @Override
    public abstract String toString();
    /**
     * Método para obter a unidade
     * @return A unidade
     */
    public Unidade getUnidade() {
        return unidade;
    }
    /**
     * Método para definir a unidade
     * @return A unidade
     */
    public boolean temUnidade() {
        return this instanceof TerrenoUnidade;
    }

}
```

Classe Unidade

```
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package com.mycompany.projeto2;

/**
 *
 * @author mgng2
 */
```

```
public abstract class Unidade {
    private String nome;
    private int posicaoX;
    private int posicaoY;
    private int pontosMovimentos;
    private int maxpontosmovimento;
    private int custo;
    private int vida;
    private int ataque;

    /**
     * Construtor da classe Unidade.
     * @param nome O nome da unidade.
     * @param maxpontosmovimento O máximo de pontos de movimento.
     */
    public Unidade(String nome, int maxpontosmovimento) {
        this.nome = nome;
        this.maxpontosmovimento = maxpontosmovimento;
        this.pontosMovimentos = maxpontosmovimento;

    }

    /**
     * Método para criar uma unidade
     * @param tipo O tipo de unidade
     * @return A unidade
     */
    public Unidade criarUnidade(String tipo) {
        switch (tipo.toLowerCase()) {
            case "militar":
                return new Militares("Militar");
            case "colono":
                return new Colonos("Colono");
            case "construtor":
                return new Construtores("Construtor");
            default:
                System.out.println("Tipo de unidade desconhecido.");
                return null;
        }
    }
```

```
    /**
     * Método para verificar se a unidade pode ser movida, devido aos pontos de movimento
     * @param novaPosicaoX A nova posição x
     * @param novaPosicaoY A nova posição y
     * @param terrenoDestino O terreno de destino
     * @return Verdadeiro se a unidade foi movida, falso caso contrário
     */
    public boolean mover(int novaPosicaoX, int novaPosicaoY, Terrenos terrenoDestino) {
        int custoMovimento = terrenoDestino.getCustoMovimento();

        if (pontosMovimentos >= custoMovimento) {
            pontosMovimentos -= custoMovimento;
            return true;
        }
        return false;
    }

    /**
     * Método para obter a vida da unidade
     * @param vida A vida da unidade
     */
    public void setVida(int vida) {
        this.vida = vida;
    }

    /**
     * Método para reestar o custo de produção
     */
    public void resetMovementPoints() {
        this.pontosMovimentos = maxpontosmovimento;
    }

    /**
     * Método para mover a unidade
     * @param novaPosicaoX A nova posição x
     * @param novaPosicaoY A nova posição y
     */
    public void move(int novaPosicaoX, int novaPosicaoY) {
```

```
        this.posicaoX = novaPosicaoX;
        this.posicaoY = novaPosicaoY;
        System.out.println(nome + " foi movida para a posiçao (" + novaPosicaoX + ", "
+ novaPosicaoY + ").");
    }

    /**
     * Método para obter o custo de produção
     * @param custoterra O custo do terreno
     */
    public void move(int custoterra) {
        if (custoterra <= pontosMovimentos) {
            pontosMovimentos -= custoterra;
            System.out.println(nome + " moveu-se. Pontos de movimento restantes: " +
pontosMovimentos);
        } else {
            System.out.println(nome + " nao pode mover-se, custo de terreno muito alto.");
        }
    }

    /**
     * Método para mover a unidade
     * @param novaPosicaoX A nova posição x
     * @param novaPosicaoY A nova posição y
     * @param custoMovimento O custo de movimento
     * @return Verdadeiro se a unidade foi movida, falso caso contrário
     */
    public boolean movePara(int novaPosicaoX, int novaPosicaoY, int custoMovimento)
{
        if (custoMovimento <= pontosMovimentos) {
            this.posicaoX = novaPosicaoX;
            this.posicaoY = novaPosicaoY;
            pontosMovimentos -= custoMovimento;
            System.out.println(nome + " moveu para (" + novaPosicaoX + ", " +
novaPosicaoY + "). Pontos restantes: " + pontosMovimentos);
            return true;
        } else {
            System.out.println(nome + " nao tem pontos de movimento suficientes para se
mover para (" + novaPosicaoX + ", " + novaPosicaoY + ").");
            return false;
```

```
    }
}
// Getters e Setters
public String getNome() { return nome; }
public int getPosicaoX() { return posicaoX; }
public int getPosicaoY() { return posicaoY; }
public int getCusto() { return custo; }
public int getPontosMovimentos() { return pontosMovimentos; }

public void setPosicao(int x, int y) {
    this.posicaoX = x;
    this.posicaoY = y;
}

public abstract void performAction();
/**
 * Método para atacar uma unidade
 * @param inimigo A unidade a ser atacada.
 */
public void attack(Unidade inimigo) {
    if (inimigo != null) {
        System.out.println(this.nome + " atacou " + inimigo.getNome());
        int dano = this.ataque - (inimigo.ataque / 2);
        inimigo.takeDamage(dano);
    } else {
        System.out.println("O alvo não é uma unidade válida.");
    }
}
/**
 * Método para atacar uma cidade
 * @param cidade A cidade a ser atacada.
 */
public void attackCity(Cidades cidade) {
    if (cidade != null) {
        System.out.println(this.nome + " está atacando a cidade " + cidade.getNome());
        int dano = 20;
        cidade.takeDamage(dano);
    } else {
        System.out.println("A cidade não é válida.");
    }
```

```
    }
    /**
     * Método para reduzir a vida da unidade quando ela recebe dano.
     * @param dano O valor de dano que a unidade sofre.
     */
    public void takeDamage(int dano) {
        this.vida -= dano;
        if (this.vida <= 0) {
            System.out.println(this.nome + " foi destruído.");
        } else {
            System.out.println(this.nome + " sofreu dano. Vida restante: " + this.vida);
        }
    }
    /**
     * Método para receber dano de outra unidade.
     */
    @Override
    public String toString() {
        return nome + " (Movimento: " + pontosMovimentos + ", Posicao: " + posicaoX +
", " + posicaoY + ")";
    }
    /**
     * Método para reduzir os pontos de movimento
     * @param custo O custo de movimento
     */
    public void reduzirPontosMovimento(int custo) {
        this.pontosMovimentos = Math.max(0, this.pontosMovimentos - custo);
    }
    /**
     * Método para resetar os pontos de movimento
     */
    public void resetarPontosMovimento() {
        this.pontosMovimentos = this.maxpontosmovimento;
    }
}
```