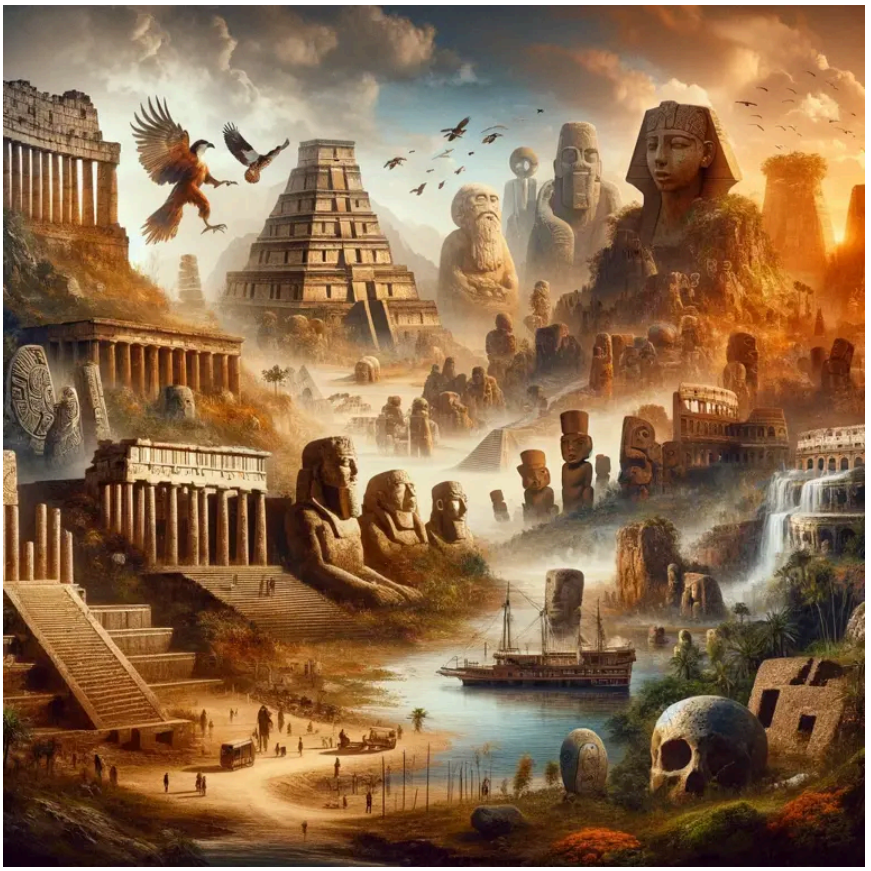
UNIVERSIDADE da MADEIRA

Programação Orientada por Objetos 2024/2025

Enunciado Projeto 2

OOP Civilizations 24/25



Data: 22 de Outubro de 2024

1 Descrição

O projeto "POO Civilizations" tem como objetivo consolidar o entendimento dos princípios fundamentais da programação orientada a objetos (POO) entre os alunos, sendo obrigatoriamente implementado em Java. Neste jogo, os estudantes irão simular um sistema de civilizações em expansão, enfrentando o desafio de implementar diversas funcionalidades relacionadas à gestão de mapas, civilizações, cidades, unidades e recursos, utilizando a linguagem de programação Java. O desenvolvimento foca na aplicação dos princípios SOLID e na promoção das boas práticas de encapsulamento, abstração, herança, polimorfismo, coesão e acoplamento.

A interface gráfica do jogo será limitada à consola do Java, com auxílio de prints personalizados para proporcionar uma melhor visualização das informações e interações durante o jogo.

As funcionalidades do jogo incluem a implementação de um mapa composto por diferentes tipos de terrenos, cada um com características próprias que influenciam a movimentação e a produção de recursos; a criação e gestão de civilizações, onde o jogador deve desenvolver cidades, gerir recursos e expandir o seu território; a produção e controlo de diferentes tipos de unidades, como militares, colonos e construtores, que desempenham papéis essenciais para o crescimento da civilização; e a definição de uma condição clara de vitória, que poderá ser alcançada através da conquista de cidades inimigas ou pela acumulação de uma quantidade significativa de ouro. A responsabilidade de escolher e implementar a condição de vitória ficará a cargo de cada grupo, permitindo-lhes definir estratégias próprias e adaptar a experiência de jogo conforme suas preferências.

2 Elementos de Avaliação do Projeto

A avaliação será composta por diferentes componentes. Os requisitos definidos são de implementação obrigatória e têm um valor total de 20 valores. O peso dos requisitos que não forem cumpridos será subtraído à nota final do trabalho. Adicionalmente, existem até 3 valores extra que os estudantes poderão obter mediante a implementação de funcionalidades opcionais, conforme especificado no final desta secção.

2.1 Requisitos

Os requisitos apresentados abaixo, que correspondem a 20 valores, são de implementação obrigatória e a respetiva cotação não pode ser recuperada através dos pontos extra. Importa referir que o foco é a implementação do requisito essencial, sendo que muitos dos detalhes descritos em cada um podem ser adaptados ou omitidos, ficando ao critério do grupo determinar a melhor abordagem e a respetiva explicação no relatório final.

2.1.1 Mapa (5 valores)

O mapa do jogo deve ser implementado de forma a representar o território de maneira clara e modular, permitindo expansões e modificações futuras. Ele incluirá diferentes tipos de terrenos, cada um com características próprias que afetam a movimentação e produção de

recursos, além de áreas inacessíveis que adicionam um elemento estratégico à jogabilidade. O mapa também será contínuo no eixo X, permitindo uma experiência de jogo mais imersiva e desafiadora. A visualização do mapa será feita na consola Java, utilizando prints personalizados para representar as diferentes características do terreno e fornecer feedback visual ao jogador.

2.1.1.1 Tipos de Terreno com Propriedades diferentes (2 Valores)

Cada célula do mapa representa um tipo específico de terreno, como floresta, montanha, planície, entre outros. Cada tipo de terreno possui propriedades distintas que afetarão o movimento das unidades e a produção de recursos. Esta diversidade de terrenos tem como objetivo enriquecer as dinâmicas do jogo, criando desafios e oportunidades diferentes para o jogador.

Deve de haver pelo menos 2 tipos de terrenos implementados e o projeto deve permitir a adição de mais terrenos no futuro, sem ser necessário modificar o código existente.

2.1.1.2 Terrenos Inacessíveis (1 Valor)

Para tornar a jogabilidade mais interessante, o mapa incluirá áreas inacessíveis, como água ou terrenos intransponíveis, que não poderão ser ocupados por unidades nem cidades.

Deve haver pelo menos 1 tipo de terreno inacessível implementado e o projeto deve permitir a adição de mais terrenos no futuro, sem ser necessário modificar o código existente.

2.1.1.3 Implementação do Mapa (1 Valor)

O mapa será constituído por uma estrutura quadrada ou retangular com dimensão definida por X por Y células. Este elemento representa o território onde decorrerão as interações do jogo, servindo como base para a localização de civilizações, cidades e unidades.

A representação do mapa deve ser clara e modular, permitindo expansões futuras.

2.1.1.4 Mapa Circular (pelo menos) no Eixo X (1 Valor)

Para aumentar a complexidade do jogo, o mapa deverá ser implementado de forma a ser circular no eixo X, criando um efeito de "contorno contínuo". Isto significa que as unidades que se movem para leste atravessaram o mapa e aparecerão no lado oposto, oeste, promovendo uma continuidade geográfica que simula a curvatura do planeta.

Por exemplo, num mapa de dimensões 5x5, se uma unidade na posição (4, 2) se mover para leste, ela reaparece na posição (0, 2).

A implementação do contorno contínuo no eixo do X é obrigatória.

2.1.2 Gestão da Civilização {Menus} (1.5 valores)

Este requisito aborda a gestão das civilizações presentes no jogo, exigindo a implementação de funcionalidades que permitam administrar unidades, cidades e recursos. As civilizações serão compostas por diversos elementos que o jogador deverá controlar para expandir o território e melhorar a prosperidade, utilizando o tesouro acumulado e otimizando a produção de recursos.

É essencial garantir que, em cada turno, o jogador não execute ações além das permitidas. Por exemplo, movimentos e ataques das unidades, construções nas cidades, geração de recursos e

outras atividades estratégicas devem respeitar os limites estabelecidos para cada turno, assegurando o equilíbrio e a integridade do jogo.

A interação do jogador com este sistema será realizada através de menus apresentados na consola, onde poderá selecionar diferentes opções para gerir a sua civilização. Por exemplo, ao ser exibido um menu, o jogador poderá escolher opções numéricas como:

1. Mover uma unidade
2. Atacar com uma unidade
3. Construir ou melhorar um edifício na cidade
4. Ver o mapa

Estes menus fornecerão uma interface simples e direta, facilitando a interação e o controlo sobre as diversas funcionalidades do jogo, enquanto garantem o cumprimento das restrições de cada turno.

A implementação dos menus deve ser modular, com recursos à classes, interfaces e métodos específicos para os menus e deve permitir adição de novos menus e elementos da interface sem ser necessário modificar o código existente.

2.1.3 Cidades (9.5 Valores)

Este requisito aborda a criação e gestão das cidades, que são elementos centrais para o desenvolvimento da civilização. As cidades devem ser estabelecidas respeitando certas regras de distância mínima entre si, e são responsáveis pela produção de recursos fundamentais como comida, produção industrial e ouro. A produção de recursos dependerá das células trabalhadas pelos cidadãos, que o jogador poderá alocar estrategicamente. A gestão dos recursos inclui a alimentação da população e a produção de unidades e edifícios. Além disso, o crescimento populacional e a administração do tesouro da civilização são componentes essenciais para expandir o território e melhorar a prosperidade da civilização. Através da gestão eficiente destes aspetos, o jogador poderá fortalecer a sua civilização e alcançar uma posição vantajosa em relação aos adversários.

2.1.3.1 Colocação e Distância entre Cidades (1 Valor)

As cidades serão estabelecidas em células do mapa, respeitando a regra de que cada cidade deve estar a pelo menos dois quadrados de distância de qualquer outra cidade. Esta restrição garante um espaçamento mínimo que evita sobreposição e promove uma gestão territorial mais estratégica.

X X X X X
X X X X X
X X C X X
X X X X X
X X X X X

Se C é uma cidade, não podem ser fundadas novas cidades nas células representadas por X.

A implementação desta verificação é obrigatória e deve ser possível modificar facilmente caso se queira aumentar ou diminuir a distância mínima entre cidades. Deve ser explicado no relatório como proceder para efetuar essa alteração.

2.1.3.2 Geração de Recursos (2 Valores)

Cada cidade é responsável pela produção de comida, produção industrial e ouro, mas isso ocorre apenas nas células que são trabalhadas pelos seus cidadãos. Por exemplo, se uma cidade tem uma população de 2, o jogador deve poder escolher quais duas células, dentro de uma distância máxima de três casas da cidade, onde irá alocar os cidadãos para trabalhar. Dessa forma, a cidade receberá os recursos produzidos por essas células. Cada cidadão de uma cidade pode trabalhar apenas uma célula, o que significa que a quantidade de recursos gerados depende do número de cidadãos e das células atribuídas a eles. Assim, a produção de uma cidade estará diretamente ligada à forma como o jogador decide distribuir os seus trabalhadores.

Para facilitar essa gestão, o jogador deve ter acesso a menus e visualizações que mostrem os terrenos próximos das cidades, as suas produções, e permitam escolher em quais terrenos colocar os cidadãos para trabalhar. Isso possibilita uma tomada de decisão estratégica sobre quais recursos priorizar em cada momento.

2.1.3.3 Gestão de Recursos da Cidade (3 Valores)

Os recursos gerados pelas cidades são utilizados de forma distinta:

Comida: Exclusiva de cada cidade, usada para alimentar a população e promover o crescimento populacional. O consumo de comida por população e a quantidade de comida necessária para crescer devem ser definidos pelos grupos. Todos os turnos, a comida produzida em excesso, ou seja, a comida gerada menos a comida consumida pela população, deve ser posta numa reserva da cidade. Se a reserva exceder um determinado limite superior, então a cidade aumenta a sua população por um e o excedente transita para a nova reserva. Se a reserva chegar a zero, porque está a ser consumida mais do que é produzida, então a cidade perde uma população no próximo turno e a sua reserva mantém-se vazia. Ver 2.1.3.4.

Produção é um recurso exclusivo de cada cidade, utilizado para construir unidades e edifícios locais. Em cada turno, a produção é consumida se for utilizada para criar algo; caso contrário, é desperdiçada. O excedente de produção não é transferido para o turno seguinte. Por exemplo, se uma cidade gera 10 pontos de produção e apenas 6 são utilizados, os 4 pontos restantes são perdidos e não se acumulam para o próximo turno. Como requisito base, a produção é apenas utilizada para produção de unidades. Ver 2.2.1 para mais informação de como utilizar a produção para outros fins.

Ouro: Enviado para o tesouro da civilização, que pertence à civilização como um todo e não a uma cidade específica. O ouro é utilizado como multiplicador de produção, permitindo acelerar a construção de unidades ou edifícios. Os valores concretos para este multiplicador, assim como o seu custo em Ouro, devem ser definidos por cada grupo, de acordo com o equilíbrio de jogo desejado. Ver 2.1.3.5.

Os recursos devem ser implementados de forma que, se for necessário acrescentar novos recursos, devem ser possível acrescentar sem modificar nada existente e reaproveitar o máximo de código possível.

2.1.3.4 Crescimento Populacional (2 Valores)

O crescimento populacional das cidades é determinado pelo excedente de comida acumulado. Cada cidade consome uma quantidade de comida por turno em função da sua população, sendo que a população determina o número de quadrados de terreno adjacentes que podem ser explorados. Este sistema de crescimento promove uma dinâmica de expansão sustentada, exigindo planejamento a longo prazo.

2.1.3.5 Tesouro da Civilização (1.5 Valores)

O ouro produzido pelas cidades será acumulado ao nível da civilização, formando um tesouro comum. Toda a produção de outro, por turno, deve ser somada ao Tesouro no turno seguinte. As unidades têm um custo de manutenção em ouro a ser pago por turno, que deve ser pago antes do excedente ser somado ao Tesouro.

2.1.4 Unidades (3.5 Valores)

Este requisito foca-se na implementação de diferentes tipos de unidades que desempenham papéis essenciais na expansão, defesa e desenvolvimento da civilização. As unidades disponíveis incluem construtores, militares e colonos, cada uma com funções específicas que permitem ao jogador explorar, construir e proteger o seu território de forma estratégica.

2.1.4.1 Tipos de Unidades (2 valores)

As unidades são um elemento fundamental do jogo, sendo divididas em construtores, militares e colonos. Cada tipo de unidade possui funções específicas, desempenhando papéis importantes na expansão e defesa da civilização:

Construtores: Responsáveis por construir melhorias no terreno, como estradas, fazendas e minas, que aumentam a produção de recursos. Os construtores também podem reparar infraestruturas danificadas. Por exemplo, um construtor pode ser utilizado para construir uma estrada que conecte duas cidades, aumentando a eficiência do movimento das unidades militares entre elas. Os construtores não têm custo de manutenção, mas são indefesos, sendo destruídos ou capturados quando atacados por militares.

Militares: Asseguram a defesa do território e têm a capacidade de atacar unidades inimigas e cidades adversárias. As unidades devem ter uma força de combate e uma quantidade de vida. Deve também ser implementado um algoritmo que calcula o resultado do confronto entre duas unidades e o confronto entre uma unidade e uma cidade (ver 2.1.4.2). Cada unidade militar deve ter um custo de manutenção, pago por turno em ouro.

Colonos: Responsáveis por fundar novas cidades. Os colonos movem-se pelo mapa até encontrar um local estratégico onde possam estabelecer uma nova cidade, contribuindo para a expansão territorial da civilização. Por exemplo, um colono pode ser enviado para uma área próxima a uma fonte de água e terrenos férteis, maximizando o potencial de crescimento da nova cidade. Os colonos não têm custo de manutenção, mas são indefesos, sendo destruídos ou capturados quando atacados por militares.

É obrigatória a implementação de pelo menos um tipo de cada unidade e o código deve permitir a adição de novos tipos de unidades sem modificar nada existente.

2.1.4.2 Mecânicas de Unidades (1 Valor)

O movimento das unidades será regido pelos custos de movimento de cada tipo de terreno, promovendo um uso estratégico do mapa. Cada unidade deve ser capaz de mover-se X casas por turno e o custo do movimento deve ser modificado conforme o tipo de território a atravessar.

As unidades militares terão a capacidade de atacar outras unidades e cidades, sendo implementada uma mecânica de combate simples para simular estas interações.

2.1.4.3 Produção de Unidades (0.5 Valor)

A produção de unidades é realizada pelas cidades, utilizando os pontos de produção gerados localmente.

2.1.5 Condição de Vitória (0.5 Valor)

Para concluir o jogo, deve ser definida uma condição clara de vitória, como conquistar todas as cidades inimigas ou atingir uma determinada quantidade de ouro. Esta condição deve ser claramente implementada e comunicada ao jogador, oferecendo um objetivo final que guie as ações durante o jogo.

2.2 Requisitos Obrigatórios e Pontos Extra

Os requisitos apresentados acima, totalizando 20 valores, são de implementação obrigatória e a sua cotação não pode ser recuperada através de pontos extras. Isto significa que, se o grupo não implementar algum dos requisitos obrigatórios, os pontos correspondentes estarão definitivamente perdidos, sem possibilidade de recuperação.

É importante salientar que a obrigatoriedade recai sobre a implementação do requisito em si. Muitos dos detalhes descritos em cada requisito podem ser adaptados ou omitidos, cabendo ao grupo decidir a melhor forma de os abordar.

Por exemplo, no caso das unidades de construtores, é imperativo que o grupo implemente pelo menos um tipo de melhoria que possa ser efetuada no terreno, como a construção de uma estrada ou de uma fazenda. A complexidade e a variedade de ações adicionais, como a edificação de minas ou a reparação de infraestruturas, ficam ao critério do grupo, permitindo-lhes decidir se desejam expandir as capacidades dos construtores além do mínimo exigido.

2.2.1 Pontos Extra (Até 4 valores)

Além dos requisitos obrigatórios, os grupos têm a oportunidade de implementar funcionalidades opcionais extra que poderão garantir até três valores adicionais. Estes pontos extra não permitirão que a nota final exceda os 20 valores, mas podem servir para compensar deficiências noutros critérios. Entre as funcionalidades opcionais encontram-se a construção de edifícios que modificam atributos das cidades, a possibilidade de trocas entre cidades e a introdução de novas unidades militares.

2.2.1.1 Edifícios (2 valores)

Pode ser adicionada uma opção para construir edifícios nas cidades. Cada edifício deve ter um custo de produção, de maneira a serem precisos vários turnos de produção até ao edifício estar concluído. Uma vez concluído, os edifícios devem pagar um custo de manutenção em ouro por turno, mas, em contrapartida, devem providenciar algum benefício. Os benefícios podem ser, por exemplo, produção militar mais rápida, maior defesa da cidade quando atacados, mais produção de ouro, etc. Para mais ideias, consultar os edifícios existentes nos jogos Civilizations para inspiração.

Para este ponto extra ser cumprido deve ser implementado pelo menos 2 edifícios diferentes e providenciar uma estrutura que permita a adição de novos edifícios sem modificar o código existente.

2.2.1.2 Trocas entre Cidades (1 Valor)

A ideia seria introduzir uma mecânica de jogo que permitisse a uma cidade enviar produção ou comida para outra cidade, para ajudar com o seu desenvolvimento.

Para este bónus ser cumprido, deve ser implementado a troca de pelo menos 1 recurso entre as cidades e deve ser possível trocar eventuais novos recursos criados sem modificar o código existente.

2.2.1.3 Unidades Adicionais (1 valor)

Introduzir novas unidades no jogo, sejam militares ou civis, que desempenham papéis específicos.

Para este bónus ser cumprido, a unidade nova implementada deve tirar partido do código já implementado para as unidades existentes, de forma a minimizar o código novo.

2.3 Critérios de Avaliação

Esta seção detalha os critérios de avaliação para o projeto, focando na aplicação dos princípios SOLID e das boas práticas de programação orientada a objetos. Cada secção aborda um conceito fundamental, descreve os princípios envolvidos e explica como será feita a avaliação.

2.3.1 Encapsulamento (20%)

Descrição

O encapsulamento consiste no uso adequado de modificadores de acesso—como private, public e protected—para proteger os dados internos das classes. Essa prática assegura que apenas métodos públicos expõem o necessário, mantendo os detalhes de implementação ocultos e permitindo alterações internas sem impactar outras partes do sistema.

Princípio SOLID Avaliado

S - Princípio da Responsabilidade Única (SRP): Assegura que cada classe tem apenas uma responsabilidade, facilitando o encapsulamento de dados e comportamentos relacionados.

Avaliação

- **Proteção de Atributos**: Verificar se as classes protegem corretamente os seus atributos, evitando acesso indevido.
- **Getters e Setters**: Analisar o uso apropriado destes métodos para controlar o acesso aos dados.
- **Responsabilidade Única**: Garantir que cada classe possui uma responsabilidade clara, evitando o acoplamento indevido de funcionalidades distintas.

2.3.2 Abstração (20%)

Descrição

A abstração foca nos aspectos essenciais de um objeto, ocultando detalhes complexos que não são relevantes para o contexto atual. Isso é alcançado através do uso de classes abstratas e interfaces para representar conceitos gerais, permitindo que detalhes específicos sejam implementados em subclasses.

Princípio SOLID Avaliado

O - Princípio Aberto/Fechado (OCP): As classes devem ser abertas para extensão, mas fechadas para modificação. A abstração permite estender funcionalidades sem alterar o código existente.

Avaliação

- **Uso de Abstrações**: Avaliar o emprego de classes abstratas e interfaces para definir contratos genéricos entre componentes do sistema.
- **Métodos Abstratos**: Verificar a implementação de métodos abstratos que são especificados em subclasses, promovendo extensibilidade.
- **Extensibilidade sem Modificação**: Projetar sistemas que permitam adicionar novas funcionalidades através de extensão, evitando modificações no código existente.

2.3.3 Herança (15%)

Descrição

A herança estabelece uma relação "é um" entre classes, permitindo a reutilização de código através de subclasses que herdam características de superclasses. Isso facilita a organização hierárquica e a manutenção do código.

Princípio SOLID Avaliado

L - Princípio da Substituição de Liskov (LSP): Objetos de uma classe devem poder ser substituídos por objetos de suas subclasses sem afetar o funcionamento do sistema.

Avaliação

- **Hierarquias Correta**: Implementação adequada de hierarquias de classes que respeitam o LSP.
- **Comportamento Consistente**: Assegurar que as subclasses mantêm o comportamento esperado das superclasses.

- **Uso Apropriado da Herança**: Utilizar herança de forma a evitar violações do LSP, garantindo a integridade e previsibilidade do sistema.

2.3.4 Polimorfismo (20%)

Descrição

O polimorfismo é a capacidade de objetos de diferentes classes serem tratados como instâncias de uma classe base comum, permitindo a substituição dinâmica de comportamentos. Isso promove flexibilidade e extensibilidade no código.

Princípio SOLID Avaliado

I - Princípio da Segregação de Interfaces (ISP): Muitas interfaces específicas são melhores do que uma interface geral. O polimorfismo facilita a implementação de interfaces mais específicas.

Avaliação

- **Métodos Sobrescritos**: Uso de anotações @Override em subclasses para implementar comportamentos específicos.
- **Interfaces Específicas**: Implementação de interfaces direcionadas para funcionalidades distintas, promovendo coesão.
- **Evitar Interfaces "Gordas"**: Prevenir a criação de interfaces extensas que forçam classes a implementar métodos desnecessários.

2.3.5 Coesão e Acoplamento (10%)

2.3.5.1 Coesão

Descrição

A coesão é um princípio fundamental na engenharia de software que se refere ao grau em que os elementos dentro de uma classe estão relacionados entre si e contribuem para uma única tarefa ou finalidade. Uma classe altamente coesa é aquela em que todos os seus métodos e atributos trabalham juntos para realizar uma única responsabilidade claramente definida.

Princípio SOLID Avaliado

S - Princípio da Responsabilidade Única (SRP): Cada classe deve ter apenas uma razão para mudar, ou seja, deve possuir uma única responsabilidade.

Avaliação

- **Responsabilidades Definidas**: Verificar se as classes têm responsabilidades bem definidas e limitadas.
- **Evitar multifuncionalidades**: Prevenir a criação de classes que acumulam funcionalidades não relacionadas, o que diminui a coesão.

2.3.5.2 Acoplamento

Descrição

O acoplamento é o grau de interdependência entre classes. Um baixo acoplamento aumenta a modularidade e facilita a manutenção e evolução do sistema. Também conhecido por código esparguete.

Princípio SOLID Avaliado

D - Princípio da Inversão de Dependência (DIP): Depende de abstrações, não de implementações concretas.

Avaliação

- **Redução de Dependências**: Uso de interfaces e abstrações para minimizar dependências diretas entre classes concretas.
- **Flexibilidade do Código**: Evitar dependências rígidas que dificultam a manutenção e a extensão do código.

2.3.6 Reutilização de Código e Manutenibilidade (10%)

Descrição

Estruturar o código para ser facilmente mantido e utilizado é fundamental. Isso inclui a organização lógica e práticas que promovam a clareza e a modularidade.

Princípios SOLID Avaliados

O - Princípio Aberto/Fechado (OCP): Facilita a extensão do código sem modificar o existente.

D - Princípio da Inversão de Dependência (DIP): Promove modularidade e facilita a reutilização.

Avaliação

- **Organização Modular**: Verificar a estruturação lógica e modular do código.
- **Nomenclatura Clara**: Uso de nomes consistentes e descritivos para classes, métodos e variáveis.
- **Documentação Adequada**: Inclusão de comentários e documentação que suportem a manutenção futura.

2.3.7 Tratamento de Exceções (5%)

Descrição

A gestão adequada de situações de erro é essencial para a robustez do sistema. O uso correto de exceções permite lidar com condições anómalas sem comprometer a estabilidade da aplicação.

Princípio SOLID Avaliado

S - Princípio da Responsabilidade Única (SRP): Manter o tratamento de erros separado da lógica de negócio principal.

Avaliação

- **Uso de Try-Catch**: Emprego apropriado de blocos try-catch para capturar e tratar exceções.
- **Exceções Personalizadas**: Criação de exceções específicas para casos particulares, melhorando a clareza e a manutenção.
- **Separação de Responsabilidades**: Garantir uma distinção clara entre a lógica de negócio e o tratamento de erros.

2.4 Relatório

A elaboração de um bom relatório técnico do software desenvolvido é fundamental para comunicar de forma eficaz as características, funcionalidades e decisões de design do projeto. Um relatório bem construído não só documenta o trabalho realizado, mas também facilita a manutenção futura, a colaboração entre equipes e a compreensão profunda do sistema por parte de terceiros. Para garantir a qualidade do relatório técnico, é essencial focar em três pilares principais: clareza e completude, apresentação profissional e profundidade técnica.

2.4.1 Clareza e Completude (50%)

A clareza e a completude são essenciais para que o relatório cumpra o seu propósito informativo. Um relatório claro e bem estruturado permite que o leitor compreenda facilmente o conteúdo sem ambiguidades ou confusões.

- **Estruturação Adequada**: O relatório deve incluir todas as seções necessárias, como introdução, objetivos, metodologia, resultados, conclusões e referências. Cada seção deve fluir logicamente para a próxima, facilitando a leitura e a compreensão do documento.
- **Descrições Compreensíveis**: As explicações devem ser detalhadas o suficiente para que o leitor entenda completamente os conceitos apresentados. Evite jargões excessivos e, quando utilizados, certifique-se de que estão devidamente explicados.
- **Coerência e Coesão**: As ideias devem ser apresentadas de forma coerente, com transições suaves entre tópicos. A coesão textual assegura que cada parte do relatório contribui para um entendimento unificado do projeto.

2.4.2 Apresentação Profissional (25%)

A apresentação profissional do relatório reflete o cuidado e o rigor aplicados no desenvolvimento do projeto. Um documento bem formatado transmite credibilidade e facilita a assimilação das informações.

- **Padrões Profissionais**: O formato do relatório deve seguir os padrões estabelecidos para documentos técnicos, incluindo margens, fontes, espaçamento e numeração de páginas. O uso consistente de estilos e formatações contribui para a uniformidade do documento.
- **Elementos Visuais**: A inclusão de figuras, tabelas e diagramas é fundamental para ilustrar conceitos complexos e apresentar dados de forma clara. Esses elementos devem ser devidamente legendados e referenciados no texto.
- **Cuidado Estético**: Atenção aos detalhes como ortografia, gramática e pontuação é crucial. Um relatório livre de erros demonstra profissionalismo e atenção ao detalhe.

2.4.3 Profundidade Técnica (25%)

A profundidade técnica do relatório evidencia o nível de entendimento e competência da equipe em relação ao projeto desenvolvido.

- **Entendimento Técnico Sólido**: O relatório deve demonstrar um conhecimento aprofundado dos conceitos técnicos envolvidos. Isso inclui explicações detalhadas sobre as tecnologias utilizadas, algoritmos implementados e desafios enfrentados.
- **Justificativas de Design**: As decisões de design tomadas durante o desenvolvimento do software devem ser bem justificadas. Explique por que determinadas abordagens foram escolhidas em detrimento de outras, considerando aspectos como eficiência, escalabilidade e manutenção.
- **Orientações para o Futuro**: Inclua recomendações e orientações úteis para a manutenção e expansão do sistema. Detalhes sobre possíveis melhorias, funcionalidades adicionais e considerações para futuras atualizações são valiosos para quem dará continuidade ao projeto.

2.5 Clarificação do Cálculo da nota

O multiplicador de POO reflete o grau de aplicação correta dos princípios SOLID e das boas práticas de programação orientada a objetos no projeto. Cada princípio é avaliado individualmente, atribuindo-se uma pontuação entre 0 e 1, de acordo com o cumprimento dos critérios estabelecidos. Os pesos atribuídos a cada princípio refletem a sua importância relativa no contexto do projeto.

Cada componente das seções 2.3 e 2.4 serão cotados da seguinte forma:

- 0 – Elemento não presente ou completamente incorreto;
- 0.25 – Elemento presente com graves falhas na sua concretização em termos conceptuais. Demonstra que o aluno não compreendeu o conceito, mas tentou implementar
- 0.50 – Elemento presente com falhas na execução. Demonstra que o aluno compreendeu o conceito parcialmente, executando corretamente a parte onde compreendeu e falhando noutras partes.
- 0.75 – Elemento presente com inconsistência na execução. Demonstra que o aluno compreendeu o conceito, mas não foi consistente na sua execução. Implementou corretamente em alguns sítios e não implementou ou implementou incorretamente noutros.
- 1 – Elemento presente e bem executado.

2.5.1 Multiplicador de POO

Multiplicador de POO=(∑(Pontuação do Princípio × Peso do Princípio)) × 0.8

Nota: O multiplicador de POO varia de 0 a 0.8, correspondendo a 80% do multiplicador total.

Exemplo de Cálculo:

- **Encapsulamento:** 0.75 * 20% = 0.15
- **Abstração:** 1 * 20% = 0.20
- **Herança:** 1 * 15% = 0.15

- **Polimorfismo:** 1 * 20% = 0.2
- **Coesão e Acoplamento:** 1 * 10% = 0.1
- **Reutilização de Código:** 1 * 10% = 0.1
- **Tratamento de Exceções:** 1 * 5% = 0.05

Multiplicador de POO = (0.15 + 0.20 + 0.15 + 0.2 + 0.1 + 0.1 + 0.05) * 0.8 = 0.95 * 0.8 = **0.76**

2.5.2 Multiplicador do Relatório

Multiplicador do Relatório=(∑(Pontuação do Critério × Peso do Critério))×0,2

Nota: O multiplicador do relatório varia de 0 a 0,2, correspondendo a 20% do multiplicador total.

Exemplo de Cálculo:

- **Clareza e Completude:** 1 × 50% = 0.5
- **Apresentação Profissional:** 0.75 × 25% = 0.1875
- **Profundidade Técnica:** 1 × 25% = 0.25

Multiplicador do Relatório = (0.5 + 0.1875 + 0.25) * 0.2 = 0.9375 * 0.2 = **0.1875**

2.5.3 Cálculo da nota final

A nota máxima possível (NMP) é calculada retirando de 20 todos os requisitos obrigatórios não implementados. O cálculo da nota final deve ser a soma de todos os valores implementados (obrigatórios e bónus), multiplicado pela soma do multiplicador de POO (MP) com o multiplicador do relatório (MR), sendo que o resultado nunca pode ser superior que NMP. Para este exemplo usaremos os valores de MP e MR obtidos acima:

NMP = 20 – Requisitos não Implementados

Nota Final = (Requisitos Implementados + Bónus) * (MP + MR)

Exemplo de cálculo da nota final:

NMP = 20 – 2 = 18

Nota Final = (18 + 4) * (0.76 + 0.1875) = 22 * 0.9475 = 20.845 = **18 valores**

Neste exemplo, a nota final é de 18 valores, pela penalização obtida pelos requisitos não implementados.

IMPORTANTE – A nota final mínima para obter aprovação no projeto são 8 valores!

2.6 Dicas para implementação do código

Seguem algumas dicas breves de como proceder para garantir uma boa cotação no MP:

- **Encapsulamento e SRP:**
    - Certifiquem-se de que cada classe tem uma única responsabilidade.

    - Protejam os atributos das classes e forneçam acesso controlado através de métodos getters e setters.
- **Abstração e OCP:**
    - Usem classes abstratas e interfaces para permitir a extensão do sistema sem modificar o código existente.
    - Planificam para que novas funcionalidades possam ser adicionadas facilmente.
- **Herança e LSP:**
    - As subclasses devem respeitar o contrato estabelecido pelas superclasses.
    - Evite alterar comportamentos de forma que quebre a substituibilidade.
- **Polimorfismo e ISP:**
    - Prefiram interfaces específicas e simples.
    - Implemente apenas os métodos necessários para a classe.
- **Coesão, Acoplamento e DIP:**
    - Mantenham as classes focadas e evite dependências fortes entre elas.
    - Usem abstrações para depender de interfaces, não de classes concretas.
- **Reutilização de Código e Manutenibilidade:**
    - Estruturam o código de forma modular.
    - Comentem e documentem para facilitar futuras manutenções.
- **Tratamento de Exceções e SRP:**
    - Separem o tratamento de erros da lógica principal.
    - Usem exceções para lidar com situações anômalas, não para controle de fluxo normal.

2.7 Dicas para elaboração do relatório

O relatório deve ser um documento profissional que descreve detalhadamente o desenvolvimento do jogo, seguindo as práticas comuns em documentação técnica empresarial. O documento deve ser claro, conciso e organizado, permitindo que qualquer pessoa compreenda o projeto e possa dar continuidade ao seu desenvolvimento.

2.7.1 Estrutura

1. **Capa**
    - Título do projeto
    - Nomes dos autores
    - Data
    - Instituição e curso
2. **Índice**
    - Lista das seções e subseções com números de página.
3. **1. Introdução**
    - **1.1. Objetivo do Documento**
        - Explique o propósito do relatório.
    - **1.2. Visão Geral do Projeto**

        - Descreva brevemente o jogo desenvolvido, seus principais objetivos e funcionalidades.

4. **2. Descrição Geral**
    - **2.1. Escopo do Projeto**
        - Detalhe o que foi implementado no projeto.
    - **2.2. Requisitos Funcionais**
        - Liste os requisitos obrigatórios implementados.
    - **2.3. Requisitos Opcionais (se aplicável)**
        - Liste as funcionalidades adicionais implementadas.
    - **2.4. Tecnologias Utilizadas**
        - Linguagens de programação, bibliotecas, ferramentas e ambientes de desenvolvimento.
5. **3. Arquitetura do Sistema**
    - **3.1. Diagrama de Classes**
        - Inclua um diagrama UML mostrando as classes, suas relações (herança, associações) e principais atributos e métodos.
    - **3.2. Descrição das Classes**
        - Para cada classe, forneça:
            - **Nome da Classe**
            - **Responsabilidade:** Explique a função da classe no sistema.
            - **Principais Atributos e Métodos:** Liste e descreva brevemente.
            - **Relações com Outras Classes:** Detalhe de heranças e associações.
    - **3.3. Aplicação dos Princípios de POO e SOLID**
        - Explique como os princípios de Programação Orientada a Objetos e SOLID foram aplicados no projeto.
        - Forneça exemplos concretos do código onde esses princípios foram implementados.
        - Discuta decisões de design relacionadas a esses princípios.
6. **4. Decisões de Projeto e Justificativas**
    - **4.1. Decisões Importantes Tomadas**
        - Liste as principais decisões de design e implementação que foram tomadas durante o desenvolvimento.
    - **4.2. Justificativas das Decisões**
        - Explique por que cada decisão foi tomada, incluindo:
            - Alternativas consideradas.
            - Benefícios e trade-offs de cada opção.
            - Como a decisão se alinha com os objetivos do projeto e princípios de POO.

- **4.3. Desafios Enfrentados**
  - Descreva quaisquer desafios técnicos ou de design e como foram superados.

7. **5. Guia para Expansão e Modificação**
   - **5.1. Pontos de Extensibilidade**
     - Indique onde e como o sistema pode ser estendido ou modificado.
     - Por exemplo:
       - Adicionar novos tipos de unidades ou terrenos.
       - Implementar funcionalidades opcionais não incluídas.
       - Melhorar a inteligência artificial.
   - **5.2. Boas Práticas para Continuação**
     - Recomendações para desenvolvedores futuros sobre como manter a consistência do código.
     - Sugestões sobre padrões de design a serem seguidos.
   - **5.3. Potenciais Melhorias**
     - Liste funcionalidades ou melhorias que não foram implementadas, mas que poderiam adicionar valor ao projeto.
8. **6. Testes e Validação**
   - **6.1. Estratégia de Testes**
     - Descreva como o sistema foi testado.
   - **6.2. Correção de Erros**
     - Documente quaisquer bugs encontrados durante os testes e como foram resolvidos.
9. **7. Conclusão**
   - Resuma os principais aprendizados e resultados do projeto.
   - Reflexão sobre a experiência de desenvolvimento e aplicação dos princípios de POO.
10. **Anexos**
    - **A. Código-Fonte**
      - Copiar e colar todo o código-fonte em anexo.
    - **B. Manual do Utilizador (opcional)**
      - Guia sobre como executar e usar o jogo.
11. **Referências**

Liste quaisquer referências bibliográficas ou recursos utilizados durante o desenvolvimento.

2.7.2 Orientações gerais

- **Clareza e Objetividade:** Escreva de forma clara, objetiva e concisa. Evite jargões técnicos sem explicação.
- **Formatação Profissional:**

    - Use fonte legível (por exemplo, Arial ou Times New Roman, tamanho 11 ou 12).
    - Mantenha consistência nos títulos e subtítulos.
    - Numere as páginas.
- **Esquemas:**
    - Inclua diagramas e figuras onde apropriado (por exemplo, diagramas de classes, fluxogramas).
    - Todas as figuras devem ter legendas e serem referenciadas no texto.
- **Referências ao Código:**
    - Quando mencionar trechos de código, formate-os adequadamente e destaque as partes relevantes.
    - Evite incluir longos blocos de código no corpo do relatório; foque em trechos que ilustram pontos importantes.
- **Revisão Ortográfica e Gramatical:**
    - Revise o documento para evitar erros de gramática e ortografia.
- **Colaboração e Originalidade:**
    - Se o projeto foi realizado em grupo, especifique as contribuições individuais.
    - Assegure-se de que todo o conteúdo é original ou devidamente referenciado.

2.7.3 Detalhes adicionais

Algumas dicas adicionais para secções específicas da estrutura apresentada acima:

3. Arquitetura do Sistema

- **Diagrama de Classes:**
    - Utilize notação UML padrão.
    - Mostre as relações de herança, agregação e associação.
- **Descrição das Classes:**
    - **Exemplo de Formato:**

```
Classe: Unidade
Responsabilidade: Representar uma unidade genérica no jogo.
Principais Atributos:
  - posição: Coordenadas no mapa.
  - pontosDeMovimento: Número de movimentos disponíveis por
turno.
Principais Métodos:
  - mover(): Move a unidade para uma posição válida.
Relações:
  - Classe base para Construtor, Militar e Colono.
```

5. Guia para Expansão e Modificação

- Seja específico sobre como alguém poderia estender o sistema.
- **Exemplos:**
    - **Adicionando um Novo Tipo de Terreno:**
        - Descreva os passos necessários para adicionar um novo terreno, como criar uma classe que herda de `Terreno` e atualizar o gerador de mapas.
    - **Implementando IA Avançada:**
        - Sugira onde implementar algoritmos de IA e como a estrutura atual suporta essa adição.

3 Instruções gerais e submissão do projeto

O projeto deverá ser realizado pelos grupos previamente definidos e deverão ser entregues um relatório e código do sistema implementado em JAVA com as funcionalidades descritas anteriormente.

Relatório: No relatório devem descrever detalhadamente a abordagem e implementação do trabalho realizado. **Deverão também incluir um diagrama de classes UML que descreve o modelo do projeto realizado**. O diagrama deve preferencialmente ser realizado recorrendo a uma ferramenta de modelação.

NOTA IMPORTANTE ANTES DE ESCREVER CÒDIGO, É **ACONSELHÁVEL** FAZER O DESENHO DO SISTEMA ANTES DE IMPLEMENTAR PARA ASSEGURAR QUE OS REQUISITOS ESTEJAM IMPLEMENTADOS

O relatório deve incluir o código do projeto em anexo como texto, NÃO como figura. Sem este requisito a entrega não será considerada válida.

Código: Deverão entregar a pasta do projeto realizado no IDE Netbeans Apache. O código deverá estar devidamente estruturado e comentado para transmitir de forma breve o objetivo de cada componente. **O código recebido pelos docentes deve ser executável sem qualquer modificação ou preparação prévia. Caso o código não execute, o grupo fica imediatamente reprovado!**

O relatório, anexo e o projeto devem ser entregues via Infoalunos. O relatório e anexo deverão também ser enviados para o Gabinete de Apoio ao Estudante.

4 Defesa do Projeto

Após a realização do projeto, os alunos deverão fazer a defesa do mesmo. A defesa será **individual** pelo que os elementos do grupo devem dominar a totalidade do trabalho realizado.

As datas de entrega referentes a cada etapa estão indicadas na tabela que se segue.

Milestones	
Data	**Entrega**
Até 20 de Dezembro de 2024	Entrega do código e relatório descrevendo o trabalho desenvolvido (via infoalunos + GAE).
Dias 3,4,5 Jan 2024	Defesa individual do projeto desenvolvido.

5. Considerações Gerais

Durante a sua implementação, o problema deve ser abordado de forma gradual:

- Desenvolvam o sistema incrementalmente e vão testando as diferentes classes à medida que vão implementando.

- Construam o esqueleto da classe *Main* que trata da entrada e saída dos dados e da interação com a aplicação.
- Vão fazendo, testando, e avançando por incrementos, à medida que as funcionalidades vão sendo desenvolvidas. Se necessário, criem pequenos programas de teste auxiliares.
- É preferível apresentarem um projeto que funciona, mas que não cumpre todas as funcionalidades do que apresentar um projeto que supostamente implementa tudo, mas não funciona.
- Os pormenores de interface com o utilizador não são considerados elementos de avaliação para o presente trabalho.

6. Código de Honestidade Académica

Espera-se que os alunos conheçam e respeitem o Código de Conduta da Universidade da Madeira (Capítulo III, 15.2) que rege esta disciplina e que pode ser consultado na página da UMa.