



O Grande Jogo do DEISI

Enunciado do projecto prático - Segunda Parte

Objectivo

Este projecto tem como objectivo o desenvolvimento de uma aplicação (programa) usando as **linguagens Java** (versão Java 16) e **Kotlin** (versão 1.5) aplicando os conceitos de modelação e Programação **Orientada a Objetos** (encapsulamento, herança, polimorfismo, etc.) e os conceitos de programação Funcional (*mapping, filtering, streams*).

O projecto está dividido em 3 partes:

- Primeira Parte - incidiu na modelação, e implementação do modelo e de um conjunto de funcionalidades iniciais;
- **Segunda Parte** - apresentada neste mesmo enunciado, onde se apresentam novos requisitos, que poderão (ou não) requerer a alteração do modelo submetido na Primeira Parte e implementação de novas funcionalidades.
- Terceira Parte - novamente com novos requisitos, aos quais os alunos deverão dar resposta. Alguns dos novos requisitos terão de ser implementados usando conceitos e técnicas de Programação Funcional.

As 3 partes são de entrega obrigatória e terão prazos de entrega distintos. O não cumprimento dos prazos de entrega de qualquer uma das partes do projecto levará automaticamente à reprovação dos alunos na avaliação de primeira época da componente prática.

Objectivos - Segunda Parte

Nesta **segunda parte** do projecto vamos alterar o modelo e o código desenvolvido na primeira parte para dar resposta a novos requisitos. **Devem portanto continuar a trabalhar no repositório que criaram para a parte 1** (i.e., não devem criar um novo repositório para a parte 2).

Passam a existir vários tipos de casas, que irão ter efeitos diferentes sempre que um programador lá caia. Nomeadamente haverão **abismos** (casas com efeitos negativos sobre o avanço dos programadores) e **ferramentas** (que podem ser usadas para reduzir ou eliminar os efeitos negativos dos abismos).

O trabalho dos alunos será:

- Criar um **diagrama de classes** em **UML** que seja suficiente para modelar a realidade apresentada nesta segunda parte, e responder aos requisitos funcionais apresentados ao longo do enunciado da mesma.
- Implementar o modelo proposto, em linguagem **Java 16**.
- Implementar algumas funcionalidades, também em **Java 16**.
- Criar testes automáticos unitários para uma parte do modelo implementado, usando **JUnit 4**.

Nos capítulos seguintes serão apresentados em pormenor os novos requisitos.

Restrições Técnicas - Segunda Parte

Nesta segunda parte do projecto não existem quaisquer restrições técnicas.

Domínio do Problema

Neste capítulo vamos descrever os conceitos envolvidos neste problema que foram introduzidos ou alterados relativamente à parte 1. A compreensão destes conceitos é essencial para a criação do modelo de classes do projecto.

Os programadores

Mantêm-se os requisitos da primeira parte relativamente aos programadores.

No entanto, poderão ser necessários atributos adicionais que suportem a interação destes com os abismos e com as ferramentas. Fica ao critério de cada grupo definir que atributos serão necessários adicionar.

Tabuleiro do Jogo

O jogo decorre num mapa dividido em N casas. Cada casa está numerada de 1 até N . O tamanho do mapa (N) poderá variar de jogo para jogo.

No seguinte exemplo mostra-se um mapa de tamanho $N = 10$:

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

A casa da posição 1 é a “casa partida”.

Num mapa de tamanho N , a casa da posição N é a “meta”.

Nesta **segunda** parte do projecto, as casas do mapa poderão:

- Representar um Espaço normal, ou;
- Representar um Abismo

Para além disso, cada casa pode:

- Estar vazia;

- Conter uma ferramenta, ou;
- Estar ocupada por um ou mais programadores.

Os Abismos

Os Abismos são casas que prejudicam o avançar dos programadores no jogo.

Os Abismos são caracterizados por:

- O ID do seu tipo - um número inteiro
- O seu título

Os tipos de Abismo conhecidos nesta parte do projecto são os seguintes:

ID de Tipo	Título	Descrição
0	Erro de sintaxe	O programador recua 1 casa.
1	Erro de lógica	O programador recua N casas, sendo N metade do valor que tiver saído no dado, arredondado para baixo. Por exemplo, se o dado deu 6, o programador recua 3 casas. Se o dado deu 3, o programador recua 1 casa.
2	Exception	O programador recua 2 casas.
3	File Not Found Exception	O programador recua 3 casas.
4	Crash (aka Rebentação)	O programador volta à primeira casa do jogo.
5	Duplicated Code	O programador recua até à casa onde estava antes de chegar a esta casa.
6	Efeitos secundários	O programador recua para a posição onde estava há 2 movimentos atrás.

Universidade Lusófona de Humanidades e Tecnologias
Linguagens de Programação II
LEI / LIG / LEIRT
2021/22 – 1º Semestre
Segunda Parte
v1.0.0

Bruno Cipriano, Lúcio Studer, Pedro Alves

7	Blue Screen of Death	O programador perde imediatamente o jogo.
8	Ciclo infinito	<p>O programador fica preso na casa onde está até que lá apareça outro programador para o ajudar.</p> <p>O programador que aparecer para ajudar, fica ele próprio preso (mas liberta o que já lá estava).</p> <p>Caso o programador que aparece tenha uma ferramenta que permita livrar-se do abismo, ele não fica preso mas também já não liberta o programador que lá estava.</p>
9	Segmentation Fault	<p>Este Abismo apenas é activado caso existam dois ou mais programadores na mesma casa.</p> <p>Todos os jogadores nessa casa recuam 3 casas.</p> <p>Caso apenas esteja um programador neste Abismo, então não existe nenhum efeito a aplicar.</p>

Nota1: caso a queda num Abismo resulte em recuar casas, o jogador nunca pode recuar para trás da primeira casa do tabuleiro. Nesses casos, fica na primeira casa.

Nota2: caso um programador caia num Abismo que o faça mover-se para outro Abismo, os efeitos do segundo Abismo não devem ser considerados. Por outras palavras, o programador que caia num Abismo, num turno, fica imune aos efeitos de um eventual segundo Abismo em que caía no mesmo turno.

Exemplo:

Considerando este mapa no início do turno do Jogador1:

square=1	square=2	square=3	square=4	square=5	square=6
Jogador1	Blue Screen Of Death			File Not Found Exception	

- O Jogador1 lança o dado e recebe o valor 4;
- O Jogador1 avança para a casa que tem File Not Found Exception;
- O Abismo “File Not Found Exception” tem como consequência o recuo de 3 casas por parte do Jogador1;
- Ao recuar 3 casas, o Jogador1 cai na casa do “Blue Screen Of Death” (BSOD);
- Como o “BSOD” é o segundo Abismo deste jogador, neste turno, não lhe acontece mais nada.

As Ferramentas

A Ferramentas são coisas que os programadores podem encontrar ao longo do jogo e que os podem ajudar a evitar ou minimizar os efeitos dos abismos.

Quando um jogador calha numa casa com uma ferramenta, apanha automaticamente essa ferramenta. Mas “nasce” uma nova ferramenta para os jogadores seguintes. Ou seja, as casas com ferramentas vão sempre fornecer essa ferramenta ao jogador que lá calhar.

Um jogador só pode ter em sua posse apenas uma ferramenta de cada tipo. Caso calhe numa casa com ferramenta e já tenha essa ferramenta, não acontece nada.

As Ferramentas são caracterizadas por:

- ID do Tipo
- Título

Um programador pode coleccionar várias ferramentas. No entanto, cada ferramenta só pode ser usada uma vez. Isto porque após usar a ferramenta, o jogador deixa de a ter em sua posse.

ID do Tipo	Título	Efeito
0	Herança	Evita os efeitos de: - ???
1	Programação Funcional	Evita os efeitos de: - ??? - ???
2	Testes unitários	Evita os efeitos de: - ???

3	Tratamento de Excepções	Evita os efeitos de: <ul style="list-style-type: none">• ???• ???
4	IDE	Evita os efeitos de: <ul style="list-style-type: none">• ???
5	Ajuda Do Professor	Evita os efeitos de: <ul style="list-style-type: none">• Erro de Sintaxe• Erro de Lógica• Exception• File Not Found Exception

Nota: como podem verificar, esta tabela não está completa. Devem completá-la tendo em conta os vossos conhecimentos de programação, tal como o feedback dos testes do DP, e implementar o programa de acordo com isso.

Devem incluir no README.md do vosso projeto a tabela completa.

Regras do Jogo

(As regras são as mesmas da parte 1, com alterações indicadas a amarelo)

- Todos os jogadores começam o jogo posicionados na “Casa partida”.
- O jogo está dividido em turnos.
- Em cada turno, joga um dos programadores.
- As jogadas são feitas por ordem do ID:
 - o jogador que tiver o ID mais pequeno joga primeiro.
- Jogar implica lançar um dado de 6 lados, para determinar qual o número de casas que o programador irá avançar neste turno.
- Após lançar o dado, o programador vai-se mover até à casa $A + M$, onde A representa o número da sua casa actual e M representa o número que saiu no dado.
- Ao chegar à casa $A + M$, irá ocorrer uma de três hipóteses:
 - Caso a casa seja um Espaço Vazio , o programador fica nessa casa até à sua próxima jogada;

- Caso a casa seja um Abismo, devem ser aplicadas as regras indicadas na tabela dos Abismos, considerando as Ferramentas que o Programador tem em sua posse;
- Caso a casa contenha uma Ferramenta, o jogador deverá “apanhar” a mesma.

Após ser tratada a opção correcta, o movimento do jogador termina e o turno passa para o jogador seguinte.

- Caso um programador ultrapasse a casa final do jogo, então deve recuar o número de casas em excesso.

- Exemplo:
 - O Mundo tem 100 casas;
 - O programador do turno actual está na casa 99;
 - O dado é lançado e sai o número 3;
 - Nesta situação, há 2 casas em excesso ($100 - 99 - 3 = -2$);
 - O programador recua então para a posição $100 - 2 = 98$.

- O jogo termina quando for atingida a seguinte condição:

- Um dos programadores chegar à casa final do jogo;
- Exista apenas um programador em jogo.

Nota: Os testes do Drop Project garantem que não existem jogos que não terminem com uma das condições listadas

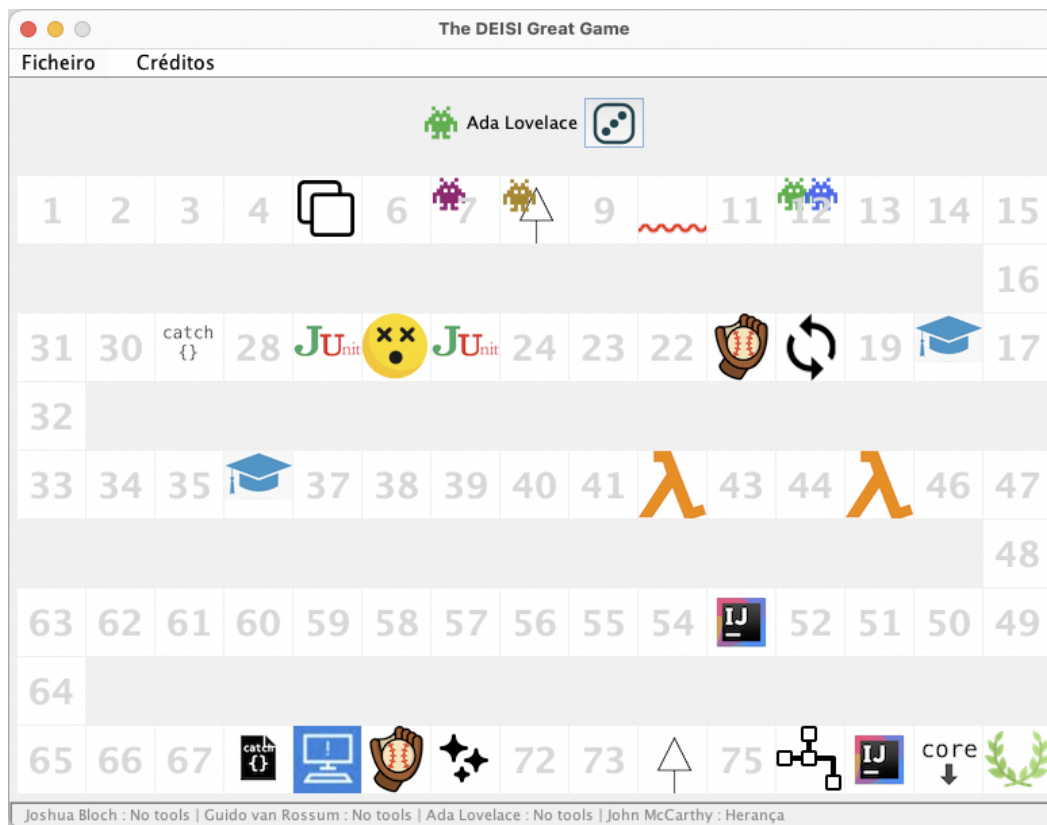
Visualizador Gráfico

O visualizador foi atualizado para:

- Mostrar abismos e ferramentas, com base no novo parâmetro que é passado ao `createInitialBoard(...)`
- Mostrar uma tooltip quando se passa o rato por cima de um abismo ou ferramenta, com o nome da mesma. Essa tooltip é obtida a partir do retorno da função `getTitle(...)`
- Reagir à entrada de jogadores em casas que contenham abismos ou ferramentas. Nesses casos mostrará uma mensagem explicativa (baseada no retorno da função `reactToAbyssOrTool(...)`)
- Mostrar uma status bar com o estado atual dos jogadores, relativamente às ferramentas que têm em sua posse (baseado no retorno da função `getProgrammersInfo()`)

Nota: Estas funções estão descritas no anexo.

A imagem seguinte apresenta o novo aspecto geral do visualizador:



Podemos ver na status bar (em baixo), que o John McCarthy tem em sua posse a ferramenta “Herança”. Os restantes jogadores não têm ferramentas.

Existe uma forma de “configurar” a imagem que o visualizador apresenta para cada abismo/ferramenta, de forma a que o jogo fique ao gosto de cada um. Para isso devem implementar a função `getImagePng(...)`.

A lista completa de classes e métodos cuja implementação é obrigatória está no **Anexo I - Informações e Restrições sobre o Visualizador**.

O uso do visualizador fornecido é obrigatório!

Durante o semestre, poderão ser publicadas novas versões do visualizador (p.e. com correcções de bugs e/ou com as alterações necessárias para suportar as várias partes do projecto). Se isto acontecer, será publicada uma mensagem informativa em moodle.

Usabilidade / Customização do Visualizador / Criatividade

Os alunos têm à sua disposição três formas de customização da usabilidade desta aplicação:

- Os ícones que aparecem em cada posição do tabuleiro;
- As mensagens que são apresentadas aos utilizadores;
- A janela de créditos.

Os alunos podem (e devem) aproveitar essas formas para tornar a sua aplicação mais interessante e apelativa. Ao fazerem isso, candidatam-se a um bónus de até 1 valor na nota desta segunda parte do projecto.

Regras para receber bónus:

Apenas serão considerados para o bónus os projectos que cumpram as regras que se seguem:

- Esta componente será avaliada pela visualização do vídeo que os alunos irão entregar (ver secção “O que entregar”).
- Durante o vídeo, os alunos devem mostrar as partes que consideram mais relevantes.
 - a. (incluindo as mensagens e o painel de Créditos)
- A decisão da atribuição do bónus irá caber aos Professores, que irão escolher as interfaces mais criativas e atribuir o bónus aos respectivos grupos.

Resultados da execução do sistema

No final do jogo, o Visualizador irá pedir ao código dos alunos a informação descrita abaixo. Isso será feito através de uma chamada ao método `getGameResults()` que terá de ser implementado pelos alunos de forma a devolver uma lista de `Strings` com o formato indicado.

~~O formato dessa lista é exatamente igual ao que foi pedido na parte 1.~~

O formato da lista a retornar é semelhante ao que foi pedido na Parte 1 do projecto, existindo apenas uma diferença relativamente à sub-secção onde se apresentam os “RESTANTES” jogadores:

- Caso dois ou mais jogadores estejam empatados na mesma posição, o código dos alunos deve-os desempatar por ordem alfabética (crescente).

Testes Unitários Automáticos

Nesta componente, os alunos devem implementar **casos de teste unitários automáticos** para garantir a qualidade do seu projecto. Por “caso de teste” entende-se a definição de um método que verifique se, para certo “*input*” é obtido o “*output*” / *resultado esperado*.

Nesta segunda parte, não existe um número mínimo de testes a implementar.

Esta componente será avaliada pela “percentagem de cobertura” que os testes garantam do código de cada grupo. A “percentagem de cobertura” define-se como a percentagem de código do projecto que é executada quando se corre uma bateria de testes.

Desta forma, recomenda-se que os casos de teste sejam interessantes e testem cenários diferentes (casos normais, casos especiais, casos limite, etc.).

Os casos de teste devem ser implementados usando **JUnit 4**, tal como demonstrado nas aulas.

Cada caso de teste deve ser implementado num método anotado com `@Test`.

Os testes devem ser implementados em uma ou mais classes com o nome `TestXYZ` (em que `XYZ` é o nome da classe que está a ser testada). Por exemplo, uma classe chamada `ContaBancaria` teria os seus testes numa classe chamada `TestContaBancaria`. As classes de teste devem estar no mesmo package que as classes que estão a ser testadas.

Caso precisem de ler ficheiros nos vossos testes, estes devem ser colocados na pasta `test-files` (ver secção “Estrutura do projeto” mais à frente). Devem aceder aos ficheiros usando caminhos relativos. Por exemplo:

```
File ficheiro = new File("test-files/somefile1.txt")
```

Entrega

O que entregar

Nesta segunda parte do projecto, os alunos têm de entregar:

- Um diagrama de classes em **UML** (em formato **png**), assim como eventuais comentários que os alunos decidam fazer no sentido de justificarem as suas escolhas de modelação;
- Ficheiros Java onde seja feita a implementação das diversas classes que façam parte do modelo;
- Ficheiros Java que implementem os testes unitários automáticos (JUnit).
- Um **vídeo** que demonstre o funcionamento do jogo no visualizador.
 - Este vídeo vai funcionar como teste de integração, de forma a mostrar que o código dos alunos interage bem com o visualizador fornecido.
 - O vídeo deve apresentar um jogo completo.
 - Caso os alunos não implementem a 100% a função `gameIsOver()`, então o vídeo deve apresentar pelo menos 7 turnos do jogo, que incluam a queda em abismos e a utilização de ferramentas para ultrapassar os abismos.
 - O vídeo não pode exceder 4 minutos de duração.
 - O vídeo deve ser carregado para o youtube e configurado como “**unlisted**” de forma a não aparecer nos resultados de pesquisa. O título do vídeo deve incluir os números dos alunos (ex: “deisi-game-21701234-21704567”).
 - O URL do vídeo deve estar no ficheiro `README.md` do projecto.
 - Vídeos com som serão valorizados (ex: narração do que está a acontecer) mas não é obrigatório ter som. Também será valorizada a narração de ambos os elementos do grupo.
 - Caso tenham implementado as indicações da secção **Usabilidade / Customização do Visualizador / Criatividade**, devem demonstrá-las no vídeo.

Nota: Projectos que não tenham o ficheiro `README` ou cujo ficheiro `README` não inclua o URL do vídeo terão uma penalização de 7 valores na nota final.

Nota importante: O programa submetido pelos alunos tem de compilar e executar no contexto do visualizador e no contexto do Drop Project.

No visualizador:

- Carregar nos botões **não pode** resultar em erros de *runtime*. Projectos que não cumpram esta regra serão considerados como não sendo entregues. Isto significa que todos os métodos obrigatórios terão de estar implementados e a devolver valores que cumpram as restrições indicadas.

No Drop Project:

- Projectos que não passem as fases de **estrutura** e de **compilação** serão considerados como não entregues.
- Projectos que não passem a fase de **CheckStyle** serão penalizados em 3 valores.

Ficheiro README.md

O ficheiro README.md do repositório github deve contar os seguintes artefactos:

- Apresentação da imagem do diagrama UML;
- Texto com comentários que os alunos decidam fazer no sentido de justificarem as suas escolhas de modelação.
 - Estes comentários não devem ser mais do que 3 parágrafos.
 - Estes comentários irão influenciar a nota do item “Modelo de classes”.
- URL do vídeo.
- Tabela com mapeamento entre abismos e ferramentas.

Para incluírem a imagem do vosso diagrama no README.md devem usar o código seguinte:

```

```

Nota: Projectos que não tenham o ficheiro README.md ou cujo ficheiro README.md não inclua o diagrama UML terão uma penalização de 7 valores na nota final.

Estrutura do projecto

O projecto deve estar organizado na seguinte estrutura de pastas:

```
AUTHORS.txt (contém linhas NUMERO_ALUNO;NOME_ALUNO, uma por aluno do grupo)
diagrama.pdf (ou.png)
+ src
|---+ pt
|-----+ ulusofona
|-----+ lp2
|-----+ deisiGreatGame
|-----+ GameManager.java
|-----+ Programmer.java
|-----+ ... (outros ficheiros java do projecto)
|-----+ TestXXX.java
|-----+ ... (outras classes de testes)
+ test-files
|--- somefile1.txt
|--- ... (outros ficheiros de input para os testes unitários)
```

Como entregar - Repósitorio git (github)

A entrega deste projecto terá que ser feita **usando o mesmo repositório git que foi usado para entrega da primeira parte do projecto**. Não serão aceites outras formas de entrega do projecto.

Todos os ficheiros a entregar (seja o código, seja o UML) devem ser colocados no repositório git.

Relembra-se que o user id / username de cada aluno no *github* tem de incluir o respectivo número de aluno.

Os alunos terão também acesso a uma página no **Drop Project [DP]** a partir da qual poderão pedir para que o estado actual do seu repositório (ou seja, o *commit* mais recente) seja testado. Nesta segunda parte do projecto, a página do DP a usar é a seguinte:

<https://deisi.ulusofona.pt/drop-project/upload/lp2-2122-projecto-1a-epoca-p2>

O trabalho será avaliado tendo em conta as submissões feitas no DP.

Filosofia do uso do DP

O objectivo do DP não é servir de guião àquilo que os alunos têm que implementar. Os alunos devem implementar o projecto de forma autónoma tendo apenas em conta o enunciado e usar o DP apenas para validar que estão no bom caminho. Nas empresas nas quais um dia irão trabalhar não vão ter o DP para vos ajudar. Nesse sentido, decidimos limitar as submissões ao DP: passam a poder fazer uma submissão a cada meia hora (isto é, têm que esperar meia hora até que possam fazer nova submissão).

Regra dos Commits - Parte 2

Nesta segunda parte do projecto, deverão ser feitos (pelo menos) **quatro (4) commits não triviais** (que tenham impacto na funcionalidade do programa), por cada aluno do grupo. **Os commits têm que demonstrar a contribuição do aluno para o projecto.** Os alunos que não cumpram esta regra **serão penalizados em 3 valores** na nota final desta parte do projecto.

Prazo de entrega

A entrega deverá ser feita através de um *commit* no repositório git previamente criado e partilhado com o Professor. Recomenda-se que o repositório git seja criado (e partilhado) o mais rápido possível, de forma a evitar que surjam problemas de configuração no envio.

Para efeitos de avaliação do projecto, será considerado o último *commit* feito no repositório.

A data limite para fazer o último *commit* é o dia **19 de Dezembro de 2021 (Domingo), pelas 23h59m** (hora de Lisboa, Portugal). Recomenda-se que os alunos verifiquem que o *commit* foi enviado (*pushed*), usando a interface *web* do github. Não serão considerados *commits* feitos após essa data e hora.

Avaliação

A avaliação do projecto será dividida pelas 3 partes:

- Nas três partes existirão baterias de testes automáticos implementadas no sistema **Drop Project** ([DP]) que irão avaliar o projecto do ponto de vista funcional.
- Existirá uma nota em cada parte do projecto.
- Nesta segunda parte aplica-se a nota mínima de **oito (8)** valores
 - Quem não alcançar essa nota mínima ficará excluído da avaliação prática de primeira época.
- Após a entrega final, será atribuída ao projecto uma nota final quantitativa, que será calculada considerando a seguinte fórmula:
 - $0.2 * \text{NotaParte1} + 0.5 * \text{NotaParte2} + 0.3 * \text{NotaParte3}$

NOTA: Foram alterados os pesos relativos de cada parte, relativamente ao que estava no enunciado da parte 1.

- A nota final mínima (antes da defesa) é de 9,5 valores.
 - Projectos que não tenham esta nota não poderão ser defendidos em primeira época.
- Para garantir que os projectos têm um conjunto de funcionalidade mínima para poderem ser defendidos, alguns dos testes desta parte vão ser obrigatórios.
 - Apresentam-se mais detalhes sobre isto a seguir à tabela de cotações.

Segue-se uma tabela de resumo dos itens de avaliação para a **segunda parte** do projecto:

Tema	Descrição	Cotação (valores)
Modelo de Classes	Análise do Diagrama UML por parte do Professor. O diagrama deve seguir as regras UML apresentadas nas aulas. Deve também estar em conformidade com o código	2

	apresentado.	
Testes automáticos	A nota será calculada em função da percentagem de cobertura de testes feitos pelos alunos (quanto maior a cobertura, maior a nota).	2
Avaliação funcional (DP)	O DP irá testar o Simulador dos alunos considerando situações de abertura de ficheiro, situações de jogada (quer válidas, quer inválidas), situações de detecção da condição de paragem (vitória, empate, etc.).	12
Avaliação funcional (Professores)		4

Defesa do projecto:

Para garantir que os alunos que vão à defesa do projecto (só após a entrega da parte 3) têm um conjunto mínimo de funcionalidade implementada para que o projecto possa ser defendido com sucesso, alguns dos testes vão ser **obrigatórios**.

- Estes testes estarão identificados com o sufixo “**OBG**”.
- Os alunos que entreguem projectos que não passem **todos os testes obrigatórios** serão reprovados em primeira época.

Avaliação - outras informações

Relativamente à análise do diagrama UML e do código do projecto:

- Serão valorizadas soluções que:
 - usem os mecanismos da programação orientada por objectos (encapsulamento, *method overloading*, etc.) nas situações apropriadas.
- Serão penalizadas soluções que:
 - não façam uso de mecanismos OO
 - (p.e. todo o projecto implementado em uma única classe, etc.).

Bruno Cipriano, Lúcio Studer, Pedro Alves

- cujo código Java que não corresponda ao modelo apresentado em UML.
- tenham situações de acesso directo a atributos
 - (p.e. alteração directa de atributos de uma classe por parte de métodos de outra classe)
- implementem métodos que não modificam nem consultam qualquer atributo/variável da classe respectiva.
- façam uso não justificado de funções ou variáveis “static”.
- façam uso não justificado do `instanceof`

Cópias

Trabalhos que sejam identificados como cópias serão anulados e os alunos que os submetam terão nota zero (0).

Uma cópia numa das entregas intermédias afastará os alunos (pelo menos) da restante avaliação de primeira época, podendo inclusivamente ter efeitos nas restantes épocas.

Notem que, em caso de cópia, serão penalizados quer os alunos que copiarem, quer os alunos que deixarem copiar.

Nesse sentido, recomendamos que não partilhem o código do vosso projecto (total ou parcialmente). Se querem ajudar colegas em dificuldade, expliquem-lhes o que têm que fazer, não lhes forneçam o vosso código pois assim eles não estão a aprender!

A decisão sobre se um trabalho é uma cópia cabe exclusivamente aos docentes da unidade curricular.

Outras informações relevantes

(mantêm-se todas as informações constantes da parte 1 mas realça-se:

– Os grupos de alunos definidos para a primeira parte do projecto devem ser mantidos para a segunda e terceira partes do projecto. Não serão permitidas entradas de novos alunos nos grupos entre as duas partes do projecto.

Anexo I - Instruções e Restrições sobre o Visualizador

Como foi referido, o projecto irá correr em cima de um visualizador gráfico, distribuído em forma de .jar.

Para que o visualizador funcione correctamente, mantêm-se todas as restrições indicadas na parte 1, com as seguintes alterações (indicadas a amarelo):

- 1) A classe `GameManager` tem de conter (pelo menos) os métodos seguintes:

Assinatura	Comportamento
<pre>boolean createInitialBoard(String[][] playerInfo, int worldSize, String[][] abyssesAndTools)</pre>	<p>Cada elemento do array playerInfo vai ter a informação de um programador:</p> <ul style="list-style-type: none">- [0] => O ID do Jogador- [1] => O Nome do Jogador- [2] => A sua lista de linguagens (as linguagens serão separadas por “;”)- [3] => A cor do boneco que identifica o jogador. Os valores possíveis são: “Purple”, “Green”, “Brown” e “Blue”. <p>Nota: os IDs podem não vir ordenados.</p> <p>O <code>int worldSize</code> vai ter o tamanho do mundo (N).</p> <p>Cada elemento do array abyssesAndTools vai ter a informação de um abismo ou ferramenta:</p> <ul style="list-style-type: none">- [0] => 0 ou 1, consoante se trata de um abismo (0) ou de uma ferramenta (1)- [1] => O ID do tipo do abismo/ferramenta (conforme aparece nas tabelas respectivas, ver secções “Os Abismos” e “As Ferramentas”)- [2] => Posição do tabuleiro onde está esse abismo/ferramenta

Universidade Lusófona de Humanidades e Tecnologias
Linguagens de Programação II
LEI / LIG / LEIRT
2021/22 – 1º Semestre
Segunda Parte
v1.0.0
Bruno Cipriano, Lúcio Studer, Pedro Alves

	<p>A função deve validar os dados recebidos. Caso algum dos dados seja inválido, a função deverá retornar false.</p> <p>Dados a validar:</p> <ul style="list-style-type: none"> Os IDs dos programadores. Não podem haver dois jogadores com o mesmo ID. Para além disso, o ID tem de ser um valor que pertença à gama esperada. Os nomes dos programadores. Não podem ser null nem estar vazios. A cor de cada jogador deve ser uma das quatro possíveis. Não podem haver 2 jogadores com a mesma cor. O número de jogadores. O tabuleiro tem de ter, pelo menos, duas posições por cada jogador que esteja em jogo. Os valores passados no array <code>abyssesAndTools</code> têm que ser válidos.
<pre>boolean createInitialBoard(String[][] playerInfo, int worldSize)</pre>	<p>Tem o mesmo comportamento que a função anterior mas não gera abismos nem ferramentas (igual à parte 1)</p> <p>Nota: Evitar duplicação de código com a função anterior. Ver slides sobre <i>method overloading</i>.</p>
<pre>String getImagePng(int position)</pre>	<p>Deve devolver o nome do ficheiro de imagem (formato PNG) que representa o “quadrado” do mapa.</p> <p>Caso se trate um “quadrado” vazio (isto é, sem abismo nem ferramenta), deve retornar <code>null</code>. Caso contrário, deve retornar o nome do ficheiro com a imagem.</p> <p>De forma a tornar o jogo mais amigável para o utilizador, recomendamos que tenham uma imagem para cada abismo, ferramenta e para</p>

Universidade Lusófona de Humanidades e Tecnologias
Linguagens de Programação II
LEI / LIG / LEIRT
2021/22 – 1º Semestre
Segunda Parte
v1.0.0
Bruno Cipriano, Lúcio Studer, Pedro Alves

	<p>a última posição (meta). O jogo inclui imagens para todos os abismos e ferramentas, assim como para a meta ("glory.png) mas os alunos estão convidados a incluir as suas próprias imagens. A lista das imagens incluídas está no anexo II.</p> <p>(As imagens a usar devem ser colocadas na pasta src/imagens e devem ter tamanho 50x50).</p> <p>Caso o <code>position</code> seja inválido (p.e. Maior do que o tamanho do tabuleiro), a função deve retornar null.</p>
<code>String getTitle(int position)</code>	<p>Deve devolver o título do abismo/ferramenta associado à posição passada como parâmetro. Caso essa posição esteja fora do tabuleiro ou não tenha nenhum abismo/ferramenta associados, deve retornar null. O título deve corresponder exatamente ao que está nas tabelas de abismos e ferramentas apresentadas anteriormente.</p>
<code>List<Programmer></code> <code>getProgrammers(boolean includeDefeated)</code> <p>Nota: na parte 1 este método devolvia <code>ArrayList</code> e não recebia parâmetros.</p>	<p>Devolve uma lista com todos os objectos <code>Programmer</code> que existem ou já existiram em jogo. Caso o parâmetro <code>includeDefeated</code> seja true, deve incluir os jogadores derrotados.</p>
<code>List<Programmer></code> <code>getProgrammers(int position)</code> <p>Nota: na parte 1 este método devolvia <code>ArrayList</code>.</p>	<p>Devolve uma lista com os objectos <code>Programmer</code> que se encontrem numa determinada posição do mundo.</p> <p>Caso o <code>position</code> seja inválido ou caso não existam programadores na posição indicada, a função deve devolver null.</p>
<code>String getProgrammersInfo()</code>	<p>Devolve uma <code>String</code> com um resumo da situação dos jogadores relativamente às ferramentas que possuem.</p> <p>A <code>String</code> deve ter o seguinte formato:</p> <pre><INFO_JOGADOR_1> <INFO_JOGADOR_2> ...</pre> <p>em que cada <code>INFO_JOGADOR</code> é:</p> <pre><NOME_JOGADOR> :</pre>

Universidade Lusófona de Humanidades e Tecnologias
Linguagens de Programação II
LEI / LIG / LEIRT
2021/22 – 1º Semestre
Segunda Parte
v1.0.0
Bruno Cipriano, Lúcio Studer, Pedro Alves

	<p><DESCR_FERRAMENTA_1>;<DESCR_FERRAMENTA_2> ;...</p> <p>Caso o jogador não tenha ferramentas, deve ficar</p> <p><NOME_JOGADOR> : No tools</p> <p>Os jogadores devem estar ordenados por ID</p> <p>Exemplo completo:</p> <p>Joshua Bloch : No tools Ada Lovelace : Herança;Testes unitários</p>
int getCurrentPlayerID()	Devolve o ID do programador que se encontra activo no turno actual.
boolean moveCurrentPlayer(int nrSpaces)	<p>Movimenta o programador do turno actual tantas casas quantas as indicadas no argumento nrSpaces.</p> <p>A jogada tem de ser validada, considerando as seguintes regras:</p> <ul style="list-style-type: none"> • O argumento nrSpaces não pode ser menor do que 1 ou maior do que 6, porque o dado tem 6 lados. • O jogador actual está impossibilitado de se mover (por ex., por ter caído num ciclo infinito) <p>Caso alguma destas regras não seja cumprida, então a função deve devolver false. Em caso contrário, a função deve devolver true.</p> <p>NOTA: Na parte 2, esta função deixa de avançar com o turno. Ver reactToAbyssOrTool().</p>
String reactToAbyssOrTool()	<p>"Ativa" uma possível reacção do jogador actual à casa onde está actualmente. Este método é sempre chamado após o moveCurrentPlayer().</p> <p>Retorna uma mensagem explicativa se a casa é um abismo ou ferramenta. Caso contrário retorna null. A mensagem explicativa não será validada pelos testes,</p>

Universidade Lusófona de Humanidades e Tecnologias
Linguagens de Programação II
LEI / LIG / LEIRT
2021/22 – 1º Semestre
Segunda Parte
v1.0.0

Bruno Cipriano, Lúcio Studer, Pedro Alves


























	<p>mas será mostrada pelo visualizador (ex: "Caiu num loop infinito! Irá ficar aqui retido até aparecer outro programador...")</p> <p>O efeito da reação poderá ser sobre o próprio jogador e/ou sobre a sua posição no tabuleiro</p> <p>NOTA: Como esta função é sempre chamada após o <code>move()</code> e pode originar outra deslocação do jogador, passa a ser aqui que deve ser efetuada a mudança de turno e não no <code>moveCurrentPlayer()</code>, independentemente de esta casa ter ou não um abismo/ferramenta.</p>
<code>boolean gameIsOver()</code>	Deve devolver <code>true</code> caso já tenha sido alcançada uma das condições de paragem do jogo e <code>false</code> em caso contrário.
<p><code>public List<String></code> <code>getGameResults()</code></p> <p>Nota: na parte 1 este método devolvia <code>ArrayList</code>.</p>	<p>Devolve uma lista de <code>Strings</code> que representam os resultados do jogo, conforme descrito na secção dos "Resultados da execução ...".</p> <p>Este método não pode devolver <code>null</code>. Caso não calculem a informação respectiva, devem devolver uma lista vazia.</p>
<code>public JPanel getAuthorsPanel()</code>	<p>Devolve um <code>JPanel</code> que pode ter o conteúdo gráfico que os alunos queiram que seja apresentado ao carregar na janela de "Créditos".</p> <p>O <code>JPanel</code> devolvido pelos utilizadores será apresentado numa janela 300x300.</p> <p>Caso os alunos não pretendam implementar o <code>JPanel</code>, devem devolver <code>null</code>.</p>

- 2) A classe **Programmer** mantém os métodos da parte 1. No entanto, o `toString` sofre alterações:

Assinatura	Comportamento
------------	---------------

<code>int getId()</code>	Deve devolver o ID do programador.
<code>String getName()</code>	Deve devolver o nome do programador.
<code>ProgrammerColor getColor()</code>	Deve devolver a cor do boneco que representa o programador no tabuleiro.
<code>String toString()</code>	<p>Retorna uma <code>String</code> com a informação sobre o programador. Esta <code>String</code> será apresentada pelo visualizador como tooltip do objecto gráfico que representa cada jogador.</p> <p>Sintaxe:</p> <pre>"<ID> <Nome> <Pos> <Ferramentas> <Linguagens favoritas> <Estado>"</pre> <p>Onde:</p> <ul style="list-style-type: none"> • <code><Pos></code> é um número inteiro que identifica a posição actual do programador no tabuleiro. • <code><Estado></code> deve ter o valor “Em Jogo” (caso o jogador ainda esteja em jogo) ou “Derrotado” (caso o jogador tenha saído do jogo). • <code><Ferramentas></code> é uma <code>String</code> com o título das ferramentas na posse deste programador, separadas por “;” (ponto e vírgula). A ordem é irrelevante. Caso não tenha ferramentas, deve aparecer “No tools”. • <code><Linguagens favoritas></code> é uma <code>String</code> com os nomes das linguagens favoritas, separadas por “; ” (ponto e vírgula seguido de espaço). A lista deve ser ordenada alfabeticamente pelo nome da linguagem. <p>Nota: Para programadores que saiam do jogo, <code><Pos></code> deve ter a posição onde estavam quando perderam o jogo.</p>

Anexo II - Ficheiros de imagens incluídos no Visualizador

 ajuda-professor.png
 blank.png
 bsod.png
 catch.png
 core-dumped.png
 crash.png
 dice.png
 duplicated-code.png
 exception.png
 file-not-found-exception.png
 functional.png
 glory.png
 IDE.png
 infinite-loop.png
 inheritance.png
 logic.png
 playerBlack.png
 playerBlue.png
 playerBrown.png
 playerGreen.png
 playerPurple.png
 secondary-effects.png
 syntax.png
 unit-tests.png
 unknownPiece.png

FIM