Universidade Lusófona de Humanidades e Tecnologias
Linguagens de Programação II
LEI / LIG / LEIRT
2021/22 – 1º Semestre
Terceira Parte
v1.0.2
Bruno Cipriano, Lúcio Studer, Pedro Alves

O Grande Jogo do DEISI

Enunciado do projecto prático - Terceira Parte

Nota prévia

A parte 3 é a continuação da parte 2. Deverão manter o repositório no qual têm vindo a trabalhar desde a parte 1.

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 - 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** - apresentada neste mesmo enunciado, 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 - Terceira Parte

Nesta terceira parte do projecto vamos alterar o modelo e o código desenvolvido na primeira parte para dar resposta a novos requisitos:

Segue-se um resumo das novidades:

- Vai existir a capacidade de gravar um jogo e continuar o mesmo mais tarde.
- Tratamento de Erros e Excepções - o método `createInitialBoard()` vai passar a lançar uma excepção caso o input inicial seja inválido.
- Vai passar a existir uma consola que permite lançar comandos sobre o jogo. Esta consola será implementada usando o paradigma funcional e permtirá não só alterar o estado do jogo como obter estatísticas do mesmo.

O trabalho dos alunos será:

- Implementar estas funcionalidades. Algumas serão em **Java 14+,** outras em **Kotlin 1.5**.

Nos capítulos seguintes serão apresentados em pormenor os novos requisitos.

Restrições Técnicas - Terceira Parte

Na terceira parte do projecto, a implementação das estatísticas deverá ser feita através de iteração interna. Ou seja, a utilização de ciclos (while, do..while e for) está proibida.

Caso estas restrições técnicas não sejam respeitadas, os testes relativamente as mesmas serão considerados incorrectos e os alunos terão nota zero nos mesmos.

Visualizador Gráfico

O visualizador foi atualizado para:

- Permitir ler e gravar um jogo previamente gravado, através das funções `loadGame(...)` e `saveGame(...)`
- Permitir lançar uma consola para correr comandos sobre o jogo

Estas funcionalidades estão descritas em mais detalhe mais à frente.

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.

Gravar e Carregar jogo

Deve ser implementado um mecanismo de gravação e carregamento do jogo.

O mecanismo de gravação deve permitir ao jogador guardar jogos para serem concluídos mais tarde.

O mecanismo de carregamento deve conseguir carregar jogos previamente gravados e permitir continuar os mesmos.

A gravação do jogo deve ocorrer em resposta a chamadas à função: saveGame().

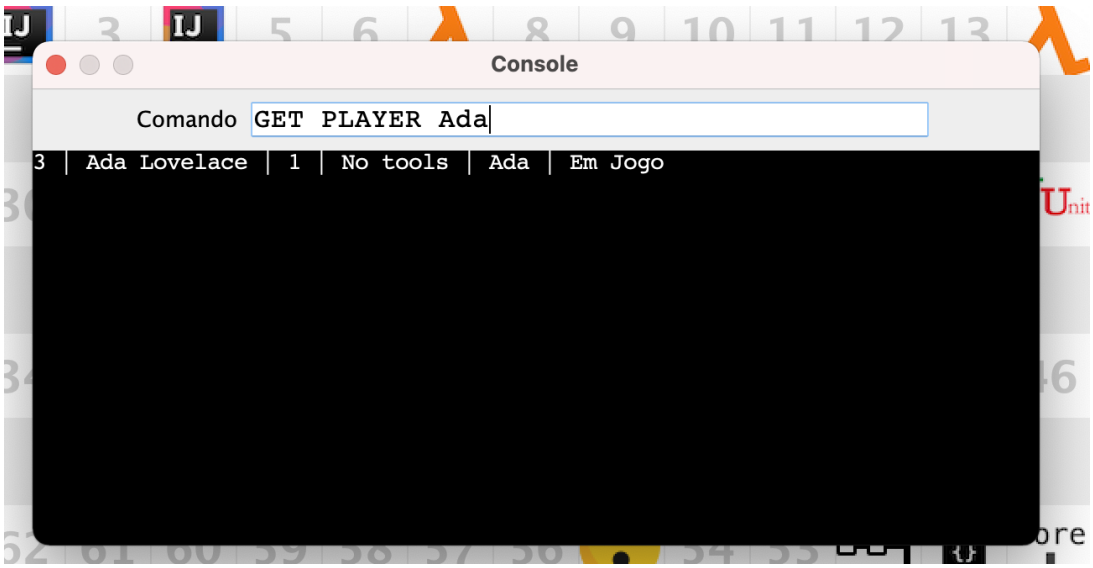
O carregamento do jogo deve ocorrer em resposta a chamadas à função: loadGame().

Mais informações na tabela do Anexo 1.

Nota: tem de ser possível carregar um jogo a meio de outro jogo.

Consola de comandos

Nesta terceira parte, surge a hipótese de interagir com o jogo através de uma consola de comandos:



Os comandos da consola obedecem ao seguinte formato:

<GET|POST> <COMMAND_NAME> [PARAMETRO_1] [PARAMETRO_2] …

Começam sempre pelas palavras “GET” (obter informação) ou “POST” (alterar o estado do jogo) seguidos obrigatoriamente pelo nome do comando e, opcionalmente, uma lista de parâmetros separados por espaço.

Nota: Os testes obedecerão sempre ao formato acima descrito.

Descrevem-se de seguida os comandos permitidos:

`GET PLAYER <playerFirstName>`	Obtém a informação do jogador cujo primeiro nome é igual ao parâmetro. A informação é a mesma que a apresentada pelo toString(), no mesmo formato

	Ex:
	`GET PLAYER Ada`
	`3 | Ada Lovelace | 1 | No tools | Ada | Em Jogo`
	
	Caso o jogador não exista, deve retornar "Inexistent player"

`GET PLAYERS_BY_LANGUAGE <language>`	Obtém a lista de jogadores associados a uma certa linguagem, separados por vírgula.
	Exemplo:
	`GET PLAYERS_BY_LANGUAGE Ruby`
	`Pedro,Bruno`
	Caso não exista nenhum jogador associado a essa linguagem, deve retornar uma String vazia
	
	Assuma que a linguagem é uma única palavra sem espaços.

`GET POLYGLOTS`	Obtém a lista com todos os programadores associados a mais do que uma linguagem de programação, ordenados por ordem crescente de número de linguagens. O resultado deve ser uma String com várias linhas em que cada linha tem o seguinte formato: `NOME_PROGRAMADOR:NUMERO_DE_LINGUAGENS` Exemplo: Joshua Bloch:3 Brunito:4 Caso hajam empates, a ordem é indiferente.
`GET MOST_USED_POSITIONS <max_results>`	Obtém as posições do tabuleiro que mais jogadores "pisaram", ordenadas da mais "pisada" para a menos "pisada". O resultado deve ser uma String com várias linhas em que cada linha tem o seguinte formato: `POSICAO:NUMERO_DE_PISADELAS` O parâmetro `<max_results>` é um inteiro indicando o número máximo de resultados

	que deve ser retornado. Para efeitos de testes, não haverão empates. NOTA: A posição 1 não entra para esta estatística. Também não entram posições que os jogadores tenham pisado como consequência de um abismo. Por exemplo, se o jogador se movimentar para a casa 5 onde está um abismo que o faz recuar 1 casa, apenas a casa 5 conta como "pisadela".
`GET MOST_USED_ABYSSES <max_results>`	Similar ao anterior, mas agora pretendem-se apenas os abismos nos quais os jogadores mais caíram, ordenados do que mais caíram para o que menos caíram. O resultado deve ser uma String com várias linhas em que cada linha tem o seguinte formato: `TITULO_TIPO_ABISMO:NUMERO_DE_QUEDAS` O `TITULO_TIPO_ABISMO` tem que coincidir com o título indicado na tabela de abismos (ver parte 2) O parâmetro `<max_results>` é um inteiro indicando o número máximo de resultados que deve ser retornado. Para efeitos de testes, não haverão empates. Se houverem várias abismos do mesmo tipo, deve ser retornado apenas aquele que tem mais "quedas". Só contam os abismos sobre o qual o jogador caiu diretamente e não como consequência de outro abismo (ex: se o jogador cai num abismo e recua para outro abismo, só conta o primeiro). Para efeitos de teste, os jogadores não terão ferramentas, por isso sofrem sempre a consequência do abismo.

`POST MOVE <numero de posições>`	Move o jogador atual tantas posições quantas o parâmetro. Caso a posição onde o jogador vai parar não tenha nenhum abismo nem ferramenta, deverá retornar “OK”
	Exemplo:
	`POST MOVE 3`
	`OK`
	(Faz avançar o jogador atual 3 posições)
	
	Caso o jogador vá parar a um abismo ou ferramenta, deverá mostrar a mensagem retornada pela função reactToAbyssOrTool
	Exemplo:
	`POST MOVE 3`
	`Caiu num ciclo infinito!`
	
	Assuma que o <número de posições> é sempre um inteiro entre 1 e 6.

`POST ABYSS <abyssTypeId> <position>`	Insere um abismo do tipo abyssTypeId na posição indicada. Caso tenha sucesso, deverá retornar “OK”. Caso a posição já esteja ocupada por um abismo ou ferramenta, deverá retornar “Position is occupied”. Apenas serão testados estes 2 casos. Podem portanto assumir que o abyssTypeId é sempre válido e que a posição está dentro do tabuleiro.

Implementação dos comandos

Deverá ser criado um ficheiro `Functions.kt`, na mesma pasta onde está o `GameManager.java`.

No topo desse ficheiro deve estar o package (exatamente igual ao do GameManager)

O ficheiro não deve conter classes, apenas funções “top-level”, tal como foi ensinado nas aulas teóricas 11 e 12.

Deve no entanto incluir um enumerado chamado `CommandType` com os possíveis valores `GET` e `POST`.

Após o utilizador inserir um comando na consola, o visualizador seguirá os seguintes passos:

- Chamará a função obrigatória `router()`, para obter uma função que recebe um `CommandType` e retorna uma função “comando”. A função “comando” deverá depender do `CommandType` que lhe foi passado.
- A função “comando” retornada deverá receber como parâmetros um objeto do tipo `GameManager` e uma lista de argumentos. A lista de argumentos são as palavras que aparecem no comando a seguir ao `GET` e ao `POST`. Por exemplo, se o comando fôr `GET PLAYER Ada`, a lista de argumentos será [“PLAYER, “Ada”]. A função executará o comando associado ao primeiro argumento com os restantes argumentos como parâmetros. Retornará uma String com o resultado ou null caso alguma coisa tenha corrido mal (ex: o comando indicado não existe).
- O visualizador escreve na consola a String retornada

Exemplo de uma função “comando”:

```
fun getPlayer(manager: GameManager, args: List<String>): String?
```

Exemplo da utilização do router:

```
val routerFn = router()
val commandGetFn = routerFn.invoke(CommandType.GET)
val result = commandGetFn.invoke(manager, listOf("PLAYER", "Joshua"))
```

Dica: Podem e devem alterar a vossa estrutura de classes de forma a suportar as estatísticas pedidas. Lembrem-se que o estado deve ser guardado em cada classe e não em variáveis globais.

Tratamento de Erros e Excepções

O método `createInitialBoard()` da segunda parte do projecto devolvia um boolean cujo valor esperado era false quando alguns dos dados fossem inválidos e true em caso contrário.

Universidade Lusófona de Humanidades e Tecnologias
Linguagens de Programação II
LEI / LIG / LEIRT
2021/22 – 1º Semestre
Terceira Parte
v1.0.2
Bruno Cipriano, Lúcio Studer, Pedro Alves

Nesta **terceira parte**, o método `createInitialBoard()` passa a ser capaz de lançar uma excepção do tipo `InvalidInitialBoardException`.

Ou seja, passa a ter a seguinte assinatura:

```
void createInitialBoard(String[][] playerInfo, int worldSize, String[][] abyssesAndTools) throws InvalidInitialBoardException
```

A excepção deve ser lançada para todos os erros que já eram validados na parte 2.

A classe `InvalidInitialBoardException` tem de:

- ser implementada pelos alunos;
- herdar da classe `Exception`;
- ser criada no package `pt.ulusofona.lp2.deisiGreatGame`.

Para verem um exemplo da criação e utilização de uma classe própria do sub-tipo `Exception`, poderão aceder ao repositório: https://github.com/ULHT-LP2-2021-22/exemplo-exceptions

Uma vez que o visualizador gera sempre um tabuleiro válido, a única forma de testarem a `exception` é através de testes unitários.

A classe `InvalidInitialBoardException` deverá ter os seguintes métodos:

- `getMessage()` que retorna uma String com uma descrição do erro.
- `isInvalidAbyss()` que retorna true se a exception foi causada por um abismo inválido
- `isInvalidTool()` que retorna true se a exception foi causada por uma ferramenta inválida
- `getTypeId()`, caso a exception tenha sido causada por um abismo ou ferramenta, retorna **uma String com** o id do tipo desse abismo/ferramenta. Nota: será garantido que este método apenas é chamado quando a causa da exception for um Abismo ou uma Ferramenta.

Testes unitários

Para testar que um método lançou uma exception, não podemos usar o assertEquals, assertTrue, etc.. Temos que rodear a chamada ao método num try/catch e chamar o fail caso não tenha lançado a Exception. Exemplo:

```
public void test() {
   try {
```

```
        metodoQueLancaException();
        fail("Deveria ter lançado uma exception");
    } catch (Exception ex) {
        assertEquals("mensagem", ex.getMessage());
    }
}
```

Testes Unitários Automáticos

Nesta terceira parte, devem continuar a usar testes unitários automáticos para testarem o vosso programa, nomeadamente as estatísticas que são complicadas de testar com o visualizador. Os testes podem ser implementados em Kotlin ou em Java

Mantém-se a componente "percentagem de cobertura", nos mesmos moldes definidos na parte 2.

Nota: Por limitações do Drop Project, não será possível incluir os ficheiros Kotlin no cálculo da cobertura.

Entrega

O que entregar

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

- Ficheiros Java onde seja feita a implementação das diversas classes que façam parte do modelo;
- Ficheiros Java que implementem os testes unitários automáticos (JUnit).

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 está 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.

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)
+ src
|---+ pt
|------+ ulusofona
|---------+ lp2
|------------+ deisiGreatGame
|--------------- GameManager.java

|--------------- Programmer.java
|--------------- ...   (outros ficheiros java do projecto)
|--------------- TestXXX.java
|--------------- ... (outras classes de testes)

|--------------- Functions.kt

+ test-files

|--- somefile1.txt

|--- ...   (outros ficheiros de input para os testes unitários)
```

Como entregar - Repósitorio git (github)

Universidade Lusófona de Humanidades e Tecnologias
Linguagens de Programação II
LEI / LIG / LEIRT
2021/22 – 1º Semestre
Terceira Parte
v1.0.2
Bruno Cipriano, Lúcio Studer, Pedro Alves

A entrega deste projecto terá que ser feita usando o mesmo repositório git que foi usado para entrega da primeira e segunda partes 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-p3

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 3

Nesta segunda parte do projecto, deverão ser feitos (pelo menos) **dois (2) *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

Universidade Lusófona de Humanidades e Tecnologias
Linguagens de Programação II
LEI / LIG / LEIRT
2021/22 – 1º Semestre
Terceira Parte
v1.0.2
Bruno Cipriano, Lúcio Studer, Pedro Alves

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 **11 de Janeiro de 2022 (Segunda-feira)**, pelas **08h00m** (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.

As defesas serão individuais e realizar-se-ão presencialmente nos dias 12 e 13 de Janeiro, em horário a definir. Todos os alunos com nota positiva no projeto (isto é, que passem os testes OBG e cuja nota final seja superior a 8 valores) estão convocados para a defesa. A não comparência na defesa implica nota zero no projeto.

Avaliação

Mantêm-se as regras indicadas na parte 2, às quais se acrescentam:

- Mantêm-se todos os testes OBG da parte 2, aos quais se acrescentam alguns da parte 3. Os alunos que entreguem projectos que não passem **todos os testes obrigatórios** (da parte 2 e parte 3) serão reprovados em primeira época.

- Após a entrega final, será atribuída ao projecto uma nota final quantitativa, que será calculada considerando a seguinte fórmula:
  - `0.2 * NotaParte1 + 0.5 * NotaParte2 + 0.3 * NotaParte3`

- A nota final quantitativa do projeto tem uma nota mínima de 8 valores.

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

Tema	Descrição	Cotação (valores)
Testes automáticos dos alunos	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.).	14
Avaliação funcional (Professores)		4

Avaliação - outras informações

- Serão penalizadas soluções que utilizem iteração externa para implementar os comandos da consola

Cópias

Mantêm-se as recomendações relativamente a cópias descritas na parte 2.

Outras informações relevantes

Mantêm-se as informações descritas na parte 2.

Anexo I - Instruções e Restrições sobre o Visualizador

Mantêm-se todas as restrições e funções obrigatórias das parte 1 e 2, às quais se acrescentam:

Assinatura	Comportamento
`void createInitialBoard(String[][] playerInfo, int worldSize,`	O comportamento é igual à parte 2, mas os erros devem originar uma

`String[][] abyssesAndTools) throws InvalidInitialBoardException`	`InvalidInitialBoardException` Mais informação na secção Tratamento de erros e excepções.
`void createInitialBoard(String[][] playerInfo, int worldSize) throws InvalidInitialBoardException`	Tem o mesmo comportamento que a função anterior mas não gera abismos nem ferramentas (igual à parte 1) Nota: Evitar duplicação de código com a função anterior. Ver slides sobre *method overloading*.
`public boolean saveGame(File file)`	Deve gravar o jogo para que o mesmo possa ser continuado mais tarde. O formato do ficheiro fica ao critério dos alunos. Caso a escrita do ficheiro decorra sem problemas, esta função deve devolver true. Caso ocorra algum erro, deve devolver false. Nunca deverá lançar exceptions.
`public boolean loadGame(File file)`	Deve ler um jogo previamente guardado e permitir continuar o mesmo. Deve assumir o formato do ficheiro que foi usado na gravação do jogo. Caso a leitura do ficheiro decorra sem problemas, esta função deve devolver true. Caso ocorra algum erro, deve devolver false. Nunca deverá lançar exceptions.
`router() : ...`	Esta função deverá ser implementada em Kotlin, no ficheiro Functions.kt. Mais informação na secção Implementação dos comandos.

FIM