



Programação Orientada a Objetos

Meta 1

Docente:

João Durães

Por:

João André Linhares Oliveira [2018012875] - LEI

João Filipe Silva de Almeida [2020144466] - LEI

Turma P3

Coimbra, 23 de Novembro de 2021

Índice

Índice	1
Estrutura / Organização	2
Estrutura de Ficheiros	2
Código	3
Frameworks / Bibliotecas	3
Ferramentas de Colaboração	3
Implementação	4
Classes	4
Island	4
Tile	4
File	4
Funções	5
Estrutura programática	6
Funcionalidades a implementar/implementadas	6
Meta 1	6

1. Estrutura / Organização

1.1. Estrutura de Ficheiros

Numa primeira fase considerámos a seguinte estrutura de ficheiros de criação:

```
|
|— CMakeLists.txt
|— main.cpp
|— mainwindow.cpp
|— mainwindow.h
|— mainwindow.ui
|— plan
|   |— plan.cpp
|   |— planpoo.svg
|— src
|   |— commands.cpp
|   |— commands.h
|   |— files.cpp
|   |— files.h
|   |— interface.cpp
|   |— interface.h
|   |— objects.cpp
|   |— objects.h
|   |— program.cpp
|   |— program.h
```

Procurámos organizar-nos separando por ficheiros as funções que mais se relacionavam com:

- criação e leitura de ficheiros (**files.cpp files.h**),
- interação com o utilizador pela consola (**interface.cpp interface.h**),
- objetos / classes (**objects.cpp objects.h**)
- tratamento e manipulação de dados (**program.cpp program.h**)

Outros Ficheiros consistem em:

- ficheiros de configuração (***.cfg**)
- ficheiros para criação da interface com a framework QT (**mainwindow.cpp mainwindow.h mainwindow.ui**)
- ficheiros para planeamento do projeto (**plan.cpp planpoo.svg**)

1.2. Código

Desde a primeira meta que tentámos organizar o trabalho visando separar os dados e a interface com o utilizador.

Esta separação é visível nas nossas funções que se especializam apenas em um dos campos.

Nesta meta usamos ainda apenas 3 classes, estas sendo: **island**, **tile** e **file** (das quais falaremos mais em detalhe na [implementação](#))

1.3. Frameworks / Bibliotecas

Decidimos fazer a interface através da framework sugerida, QT. Pelo que, desde já, organizamos o trabalho de modo a compilar utilizando as ferramentas necessárias para o QT. No entanto, na presente meta ainda apenas interagimos com o utilizador pela consola.

1.4. Ferramentas de Colaboração

Na partilha e criação de código em simultâneo, utilizamos maioritariamente as ferramentas já integradas no CLion, especificamente a ferramenta Code With Me (entre outras menos relevantes).

Para a manipulação de código em conjunto foi utilizado Git, no website Github com o cliente do Github.

Para a criação síncrona do relatório utilizámos o Google Docs.

2. Implementação

2.1. Classes

2.1.1. Island

Objeto que irá conter a informação total do mundo criado.

- Dados:
 - Dimensões da ilha.
 - Vetor de vetores do tipo 'tile'
- Responsabilidades:
 - Recebimento e validação que lhe compete de comandos para interação com o mundo
 - Transmissão dos comandos à zona adequada.

Optámos pela utilização de um vetor bidimensional para criação e divisão da ilha em zonas, isto por nos possibilitar a referência a uma zona da ilha por dois parâmetros (linha e coluna) e pela sua natureza já dinâmica.

2.1.2. Tile

Objeto auxiliar da classe "Island". Encarrega-se de criar e manter uma zona individual.

- Dados:
 - Tipo de zona
 - Se contém uma construção e qual
 - Array de trabalhadores.
- Responsabilidades:
 - Recebimento e validação de comandos para interação com o tile
 - Alteração dos dados de acordo com os comandos c«recebidos
 - Informar se o comando recebido falhou e porquê

Nos dados que competem a esta classe, optámos pela utilização de um array de inteiros para representar a quantidade de trabalhadores de cada tipo lá presentes devido à sua simplicidade uma vez que sabemos previamente a quantidade de tipos de trabalhadores que existem

2.1.3. File

Objeto usado estritamente para recuperação do estado do jogo através de ficheiros de leitura.

- Dados:
 - Histórico de comandos executados com sucesso
 - Array de dois inteiros para armazenamento das dimensões iniciais da ilha
- Responsabilidades:
 - Ser fácil o levantamento dos seus dados

Optámos por um vetor de strings para guardar os comandos simplesmente como o utilizador os ordenou. Uma vez que apenas quando o comando tem sucesso e isto não acontecerá caso haja argumentos ou um número de argumentos errôneos, este método é adequado.

2.2. Funções

```
// class file
std::vector<std::string> file::redoCommands();
void file::receiveDim(const int dims[2]);
void file::receiveCommand(const std::string& command);
int file::giveLines();
int file::giveColumns();

// class island
std::string island::showInfoIsland() const;
tile island::getTile(int l, int c) const;
std::ostream island::cont(std::vector<std::string> commandsVec);
std::ostream island::cons(std::vector<std::string> commandsVec);
bool island::isOutOfBounds(int l, int c) const;

// class tile
std::string tile::showInfoTile() const;
std::string tile::cont(const std::string& cmd);
std::string tile::getType();
std::string tile::cons(const std::string& command);

void welcome();
void newGame();
bool loadGame(const std::string& filename);
void plays(island& world, const file& savegame);
void showCredits();
std::string helpMe();
std::string treatCommand(std::string& commands, island& world, file
savegame);
std::vector<std::string> redoCommands();
bool checkFile(const std::string& filename);
file openFile(const std::string& filename);
bool saveFile(const std::string& filename, const file &filereceived);
bool saveCommands(const std::string& filename, file filereceived);
void createNewWorld(int * dim);
void createLoadedWorld(file loadedFile);
void game(island& island, const file& gamefile);
bool gameover(island& world);
void dawn(island& world);
void dusk(island& world);
int random (int low, int high);
```

2.3. Estrutura programática

As nossas funções interligam-se da seguinte forma:

A execução do programa começa na função `main()`, que irá encaminhar para a função `welcome()` que poderá chamar várias outras funções de acordo com a ordem recebida do utilizador.

A funcionalidade principal do nosso programa ocorre na função `game()`, pois esta estará num ciclo a correr as funções `dawn()`, `plays()` e `dusk()` até a função `gameover()` retornar verdadeiro.

A partir da função `plays()`, vale a pena mencionar a função `treatCommand()`, que receberá o input do utilizador e executará quaisquer funções necessárias para validar e executar os comandos lidos.

2.4. Funcionalidades a implementar/implementadas

2.4.1. Meta 1

Funcionalidade	Implementada	Parcialmente implementada	Não implementada
Leitura do ficheiro de configuração.	X		
Construção inicial da ilha.	X		
Representação simplificada para as zonas.	X		
Representação visual da ilha e conteúdo incluído.	X		
Implementação da leitura e validação de todos os comandos.	X		
Construção de edifício do tipo minaferro.	X		
Contratação de mineiros.	X		
Visualização dos dados do jogo e de zonas	X		
Projeto organizado em .h e .cpp separados	X		