Departamento de Engenharia Informática e de Sistemas
Licenciatura em Engenharia Informática
Licenciatura em Engenharia Informática - PL

Trabalho Prático

Final

Programação Orientada a Objetos

2020/2021

Diogo Pires, 2018017890, P6

Rui Canas, 2018020744, P6

Índice

Introdução

O presente trabalho prático realiza-se no âmbito da cadeira de Programação Orientada a Objectos, do 2º ano da Licenciatura em Engenharia Informática, e tem como objetivo aprofundar os conhecimentos teórico-práticos adquiridos em aula.

Depois de implementadas todas as funcionalidades pedidas para o projeto, é necessário responder a uma serie de perguntas neste relatório, relativas à aplicação desenvolvida.

Classes consideradas na primeira versão da aplicação

Para a primeira meta do trabalho prático que foi proposto foram utilizadas as seguintes classes:

- **GameData** – classe que encapsula todas as classes envolvidas na lógica e vai receber a informação da interface. Vai ser responsável pela execução de ações que foram pedidas pelo utilizador.
- **World** – classe que vai conter os territórios disponíveis para conquista durante o jogo.
- **Territory** – classe que contém a informação detalhada de um território.
- **Empire** – contém toda a informação relativa a territórios conquistados. Também é composta pela força militar, armazém e cofre.
- **Army** – classe que contém toda a informação da força militar do império.
- **SafeBox** – classe que contém informação sobre o ouro que o império possui.
- **Storage** – classe que contém toda a informação sobre os produtos que estão sob a posse do império.
- **FileReader** – classe responsável por tratar da leitura de ficheiros bem como a informação que os mesmo possuem. Esta classe também já faz a filtragem de comandos inválidos que possam estar presentes nos ficheiros de texto.
- **Interface** – classe responsável pela parte gráfica. Através dela, o utilizador consegue interagir com o programa.

Leitura do enunciado

Ao ler o enunciado, identificamos de imediato classes que seriam cruciais para o funcionamento do programa – **World** e **Empire.**

Também foi identificada uma classe que seria responsável pela **leitura do ficheiro**. O conceito de **conquistar territórios** também nos levou a pensar numa classe do tipo **território.**

Mais adiante, foi necessário desenvolver outras classes (cada uma com a sua importância para um bom funcionamento do programa) destacando-se as classes:

- **Interface –** responsável por fazer a interação com o utilizador. Vai fazer a leitura dos comandos inseridos pelo utilizador e assim, fazer com que sejam desencadeadas as ações na parte da lógica do programa – **GameData**.

Duas principais classes da aplicação

As duas principais classes para o bom funcionamento do programa seriam as seguintes:

GameData

No início do programa, é imperativo que seja sempre criado um objeto do tipo **GameData**.

```
GameData* gd = new GameData();
```

Figura 1 - Construção da classe GameData

Este objeto vai tratar de toda a lógica do jogo sendo assim armazenado na classe Interface para que seja possível a execução de comandos.

```
Interface::~Interface()
{
    delete(gD);
}
```

Assim que for feito o pedido para o términus do programa, está prevista a sua devida destruição.

Figura 2 - Destruição da classe GameData

A destruição do **GameData** deverá ocorrer no destrutor da **Interface** uma vez que esta é a classe que vai estar a encapsular a classe referida inicialmente.

Interface

Classe que também é construída no início do programa. Necessita de receber como parâmetro um ponteiro do tipo **GameData.**

Figura 3 - Construção da classe interface

É através do parâmetro que recebe que vai conseguir fazer com que seja possível efetuar a execução dos comandos introduzidos pelo utilizador.

Caso esta classe não existisse, não seria possível separar a lógica da interface e o código tornar-se-ia confuso.

O destrutor desta classe também está a ser chamado, como se pode verificar na figura dois.

Exemplo de uma responsabilidade de encapsulamento

A responsabilidade de "guardar territórios" está atribuído à classe **World**, que vai ter um vetor de ponteiro para **Territory**.

Em todas as classes criadas, é tido em atenção o *private* no ínicio que, apesar de ser feito por *default*, torna o código mais claro no que diz respeito ao encapsulamento de classes.

```
class Territory
{
private:
    Utils converter;
    static int count;
    std::string name;
    int resistance;
    int prodCreation;
    int goldCreation;
    int winPoints;
    bool conquered;
```

Figura 4 - Classe utilizada como exemplo

Duas classes com objetivo focado, coeso e sem dispersão

O melhor exemplo para classes de objetivo focado será a classe **Interface** e a classe **FileReader**.

A classe **Interface** é única e exclusiva que age como uma interface de texto, ou seja, apenas recebe e mostra informação (*stdin/stdout*), pede a execução de certos comandos introduzidos pelo utilizador e o seu único propósito é somente esse. Assim, teremos código que se encontra organizado e irá tornar o programa mais fácil de manipular para futuras alterações.

A classe **FileReader** tem como único objetivo ler ficheiros de texto e devolver o resultado da sua leitura à classe que o está a utilizar. Para além do que foi descrito, esta classe também irá realizar uma filtragem de conteúdo do ficheiro pelo que só irá retornar conteúdo válido após a leitura de um ficheiro de texto.

Desta forma, temos duas classes diferentes, porém com objetivos específicos e que não dispersão em ocasião alguma.

Interface vs. Lógica

A classe **Interface** é a classe que vai ser responsável pela interface, como já foi referido neste documento.

A classe **GameData** irá possuir toda a lógica do jogo.

```
int main() {
    GameData* gd = new GameData();
    Interface i(gd);
    i.run();
}
```

Figura 5 - Classe main

Primeiro objeto para além da camada de interação com o utilizador que recebe e coordena uma funcionalidade de natureza lógica

As ordens vindas da camada de interação com o utilizador são recebidas e processadas através de um objeto da classe **GameData.**

Figura 6 - Exemplo de chamada da classe GameData a partir da classe Interface

Figura 7 - Metódo do GameData que é chamado na classe Interface

Envolvente de toda a lógica

A classe **GameData** é responsável por delegar funções às restantes classes existentes no programa. Portanto pode-se considerar que esta é quem possui a envolvente de toda a lógica.

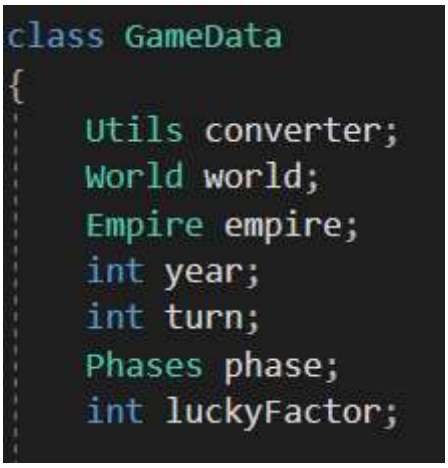

Figura 8 - Classe mencionada que vai delegar funções

Exemplo de uma funcionalidade que varia conforme o tipo do objeto que a invoca

Um dos exemplos do uso do polimorfismo neste trabalho é a classe **Technology**. Desta classe derivam algumas classes que dependendo do seu tipo, ao chamar o método *'applyTech()'* tem um resultado diferente.

```
class Technology {
protected:
    bool active;
    int price;
public:
    Technology(int price);
    virtual void applyTech() = 0;
```

Figura 9 - Método 'applyTech()' declarado como virtual puro.

Por exemplo se for um objeto do tipo **Drone** a chamar este método é o limite máximo de força militar é aumentado.

```
void Drone::applyTech() {
    setActiveTrue();
    army->setMaxMiliForce(MAX_MILIFORCE_WITH_DRONE);
}
```

Figura 10 - Efeito do método 'applyTech()' na classe Drone

Caso seja um objeto do tipo **CentralBank** a chamar este método serão os limites máximos do cofre e armazém do império a serem aumentados.

```
void CentralBank::applyTech() {
    setActiveTrue();
    empireStorage->setMaxProducts(MAX_STORAGE_WITH_CENTRALBANK);
    empireSafeBox->setMaxGold(MAX_SAFEBOX_WITH_CENTRALBANK);
}
```

Figura 11 - Efeito do método 'applyTech()' na classe CentralBank

Principais classes da aplicação

Classe: **Interface**

Responsabilidades:

- Interagir com o utilizador;
- Receber comandos;
- Chamar funções consoante o comando inserido.

Colaborações: *GameData.*

Classe: **GameData**

Responsabilidades:

- Conter toda a lógica do jogo pedido;
- Armazena os métodos necessários ao bom funcionamento de uma qualquer interface.

Colaborações: *Empire, World.*

Classe: **World**

Responsabilidades:

- Armazenar ponteiros para todos os territórios criados no jogo;
- Apagar o vetor de territórios da memória dinâmica.

Colaborações: *Territory*.

Classe: **Territory**

Responsabilidades:

- Armazenar a informação de um território;
- Tem a capacidade para retornar informação de um território;
- Delega tarefas nas classes derivadas.

Colaborações: *Nenhuma.*

Classe: **Empire**

Responsabilidades:

- Armazenar ponteiros para todos os territórios que pertencem ao império;
- Armazenar o cofre e o armazém do império;
- Armazenar um Força Militar do império;

- Armazenar tecnologias como banco central e bolsa de valores.

Colaborações: *Army, SafeBox, Storage, Utils, Territory.*

Classe: **Army**

Responsabilidades:

- Armazenar os valores relativos à Força Militar;
- Armazenar as tecnologias relativas ao exército, disponíveis pelo império.

Colaborações: Nenhuma.

Classe: **SafeBox**

Responsabilidades:

- Armazenar o ouro do império.

Colaborações: Nenhuma.

Classe: **Storage**

Responsabilidades:

- Armazenar os produtos do império.

Colaborações: Nenhuma.

Classe: **FileReader**

Responsabilidades:

- Efetua a leitura de um ficheiro de texto que lhe é indicado;
- Faz a filtragem de texto inválido dentro do ficheiro de texto.

Colaborações: *Nenhuma.*

Classe: **Technology**

Responsabilidades:

- Gere a existência das tecnologias no imperio.
- Delega tarefas nas classes derivadas.

Colaborações: *Nenhuma.*

Classe: **Event**

Responsabilidades:

- Delega as várias funcionalidades de eventos nas classes derivadas.

Colaborações: *GameData.*

Funcionalidades implementadas

Componente do trabalho	**Realizado**	**Parcialmente realizado**	**Não Realizado**
Meta 1	X		
Comandos guarda, ativa e apaga.	X		
Comandos de Debug pedidos no enunciado	X		
Diferentes fases do jogo	X		
Encerramento correto do jogo	X		
Robustez das classes	X		