IPV – Instituto Politécnico de Viseu
ESTGV – Escola Superior de Tecnologia e Gestão de Viseu
Departamento de Informática

Relatório do Projeto Prático

Licenciatura em Engenharia Informática

Realizado em
Programação Orientada a Objetos
Por

Cristina Santos Nº 15247

Francisco Lopes Nº 18705

João Lopes Nº 17179

Viseu, 2024

IPV – Instituto Politécnico de Viseu
ESTGV – Escola Superior de Tecnologia e Gestão de Viseu
Departamento de Informática

Relatório do Projeto Prático

Licenciatura em Engenharia Informática

Realizado em
Programação Orientada a Objetos

Por

Cristina Santos Nº 15247

Francisco Lopes Nº 18705

João Lopes Nº 17179

Viseu, 2024

Índice

Índice de Figuras

1. Introdução

A atividade de jogo existe há centenas de anos, maioritariamente ilegal, mas frequentada por uma vasta gama da sociedade. Com a evolução dos tempos, algo tão lucrativo tornou-se legal e atrativo.

Na década de 40 surgem os primeiros casinos, as salas decoradas, as novas máquinas, mesas de jogo, roletas, tudo era impulso para cativar clientes.

A expansão dos casinos ao longo dos anos é notória, apesar de recentemente ter surgido os casinos online, os físicos continuam com receitas elevadas e a beneficiar da confiança da economia e do crescimento do turismo. Com esta evolução a gestão dos mesmos foi se tornando mais complexa e desafiante.

O presente relatório documenta um programa desenvolvido em linguagem C++, com o intuito de simular o funcionamento e gestão de um casino. Tendo por base que um casino é frequentado por pessoas (Jogadores) que podem jogar nas diversas máquinas de diversão.

A estrutura do programa segue a lógica de um casino real, com horário de abertura e encerramento. Os jogadores, ao ingressarem no casino, detêm a possibilidade de se associar a uma máquina específica, entrar em filas de espera, trocar dinheiro por fichas correspondentes e realizar jogadas. O casino dispõe de diversas máquinas, como *slots*, roletas, *poker* e *blackjack*, cada uma com comportamentos específicos durante a execução do programa.

A gestão do casino abrange funcionalidades como adição, edição, remoção e alteração da posição das máquinas, além do controle de utilizadores. São monitorizados e manipulados inúmeros detalhes, tais como, ajuste da temperatura da máquina, a probabilidade das máquinas mais próximas caso concedam prêmios aos jogadores e quando necessário, são implementadas funções de manutenção como avarias e reparos nas máquinas. Por último, existem relatórios detalhados sobre o estado do casino e monitorização do estado das máquinas.

2. Estrutura de Dados

2.1 Esquema

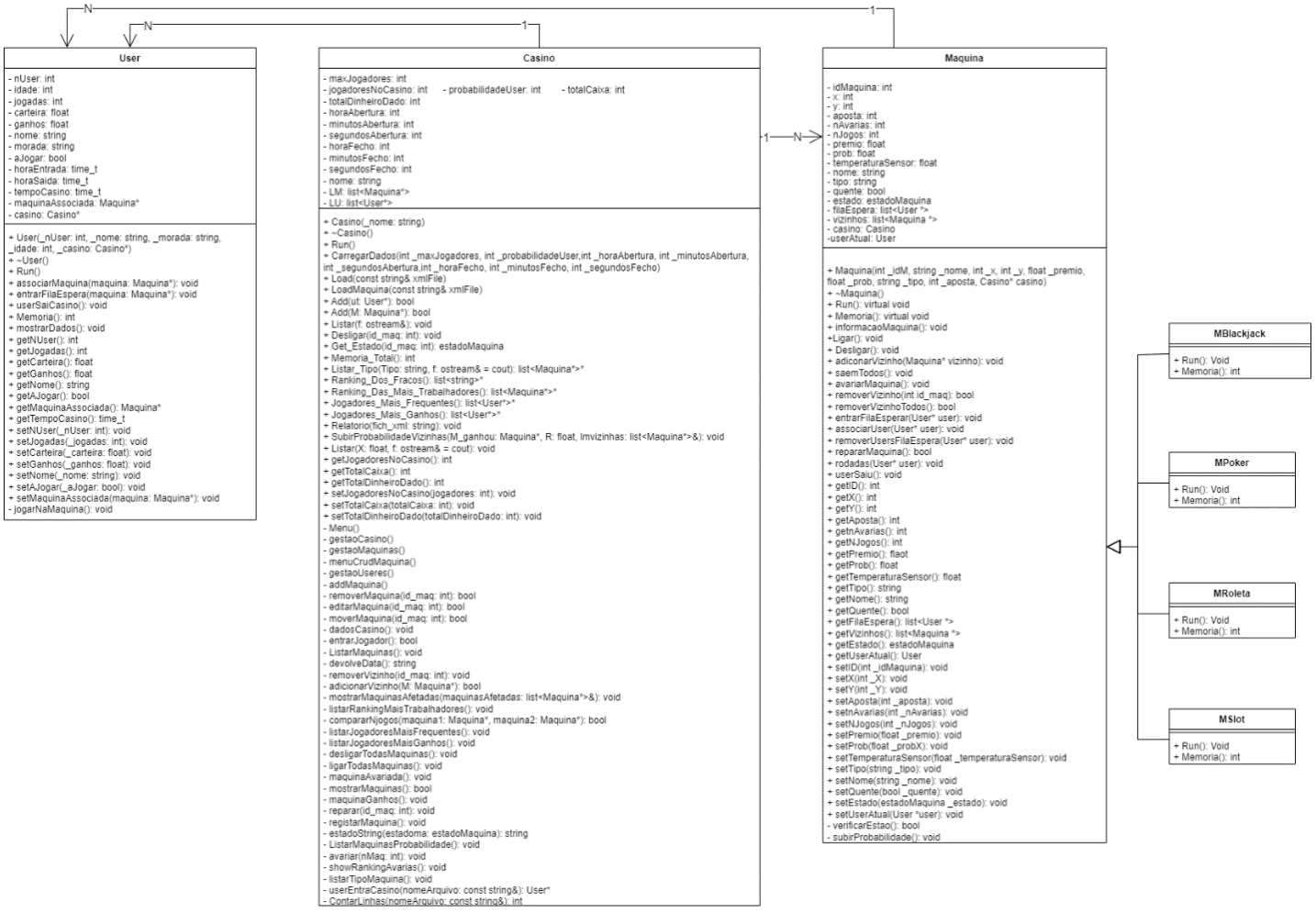

Figura 1- Modelo de Dados

3. Funções

Serve o presente capítulo para ilustrar e contextualizar as funções solicitadas relativamente à elaboração do presente projeto. Todas estão devidamente identificadas e fazendo-se acompanhar do esclarecimento, quanto à logica e onde foram aplicadas.

3.1 Load

A função Load() é responsável pelo carregamento dos dados relativamente ao casino.

3.1.1 Declaração

```
bool Casino::Load(const string& ficheiro) {
```

Figura 2- Declaração da função "Load"

3.1.2 Implementação

Inicia-se as variáveis do casino, referentes ao número máximo de jogadores, horário de abertura e fecho. De seguida as variáveis são passadas por parâmetro à classe “XMLReader” da função LoadCasinoData (). Posto isto, verifica-se se o resultado foi bem sucedido, caso contrário imprime uma mensagem de erro.

```
//Processo de carregar dados do casino atraves do XML
bool Casino::Load(const string& ficheiro) {
    int maxJogadores, horaAbertura, minutosAbertura, segundosAbertura, horaFecho, minutosFecho, segundosFecho;
    //Chama XMLReader::LoadCasinoData para obter os dados do XML
    bool result = XMLReader::LoadCasinoData("XML_Projecto.xml", maxJogadores, horaAbertura,minutosAbertura, segundosAbertura, horaFecho, minutosFecho, segundosFecho);
    if (result) { //Se a operacao de leitura do XML for bem sucedida
        //Chama CarregarDados passando por parametros os dados obtidos
        CarregarDados(maxJogadores,horaAbertura,minutosAbertura, segundosAbertura, horaFecho, minutosFecho, segundosFecho);
    } else {
        cout << "Falha ao carregar as configuracoes do Casino a partir do XML." << endl;
    }
    return true;
}
```

Figura 3- Implementação da função Load()

A função CarregarDados() atua com um segundo construtor da classe Casino. Esta recebe os dados essenciais, como o número máximo de jogadores e os horários de aberura e de encerramento, e inicia as variáveis correspondentes da classe

```
void Casino::CarregarDados(int _maxJogadores, int _horaAbertura, int _minutosAbertura, int _segundosAbertura,
                           int _horaFecho, int _minutosFecho, int _segundosFecho)
{
    maxJogadores = _maxJogadores;
    horaAbertura = _horaAbertura;
    minutosAbertura = _minutosAbertura;
    segundosAbertura = _segundosAbertura;
    horaFecho = _horaFecho;
    minutosFecho = _minutosFecho;
    segundosFecho = _segundosFecho;
    jogadoresNoCasino=0;
    totalCaixa=0;
    totalDinheiroDado=0;
}
```

Figura 4- Implementação da função CarregarDados()

3.2 Adicionar

3.2.1 Adicionar Máquina

3.2.1.1 Declaração

```
bool Add(Maquina *M);
```

Figura 5- Declaração da função Add() máquina

3.2.1.2 Implementação

A função Add, *Figura 6*, tal como o nome indica, é responsável por adicionar uma máquina ao casino. Esta recebe por parâmetro um ponteiro para o objeto "Maquina", verifica se o ponteiro é *null*, caso contrário adiciona à lista de vizinhos e à lista de máquinas do casino.

```
//Adicionar máquina
bool Casino::Add(Maquina *m){ //recebe por parametro um ponteiro para um objecto do tipo Maquina
    bool resultado = false;
    if (m == nullptr) { //Se ponteiro de Maquina for nulo
        resultado = false;
    }else{
        adicionarVizinho(m); //Função para adicionar vizinho
        LM.push_back(m); //Adicionar maquina a lisat de maquinas do casino
        resultado = true;
    }
    return resultado;
}
```

Figura 6- Implementação da função Add() máquina

3.2.2 Adicionar Utilizador

3.2.2.1 Declaração

```
bool Add(User *ut);
```

Figura 7- Declaração da função Add() utilizador

3.2.2.2 Implementação

A lógica da função, *Figura 8*, é muito semelhante à anterior, recebe como parâmetro um ponteiro para o objeto "*User*", verifica se o ponteiro tem valor *null*, caso contrário adiciona o novo utilizador à lista de utilizadores e retorna o resultado das operações descritas anteriormente.

```
//adicionar jogador/user
bool Casino::Add(User *ut){
    bool resultado = false;
    if (ut == nullptr) { //Se ponteiro do User fou nulo
        resultado = false;
    }else{
        //Adiciona o User à lista de users do casino
        LU.push_back(ut);
        resultado = true;
    }
    return resultado;
}
```

Figura 8- Implementação da função Add() utilizador

3.3 Construtor

3.3.1 Declaração

```
Casino::Casino(string _nome)
```

Figura 9- Declaração da função Casino()

3.3.2 Implementação

A declaração da classe Casino recebe um parâmetro "_nome" do tipo *string* e é responsável por inicializar a variável nome da classe com o valor fornecido como argumento. A implementação realiza essa inicialização dentro do construtor.

```
Casino::Casino(string _nome)
{//ctor
    // Inicializa as variaveis da classe com os valores passados como argumentos
    nome = _nome;
}

Casino::~Casino()
{
    //dtor
    //destui Maquinas
    for (list<Maquina *>::iterator it = LM.begin(); it != LM.end(); ++it){
        delete (*it);
    }
    LM.clear();
    //destui User
    for (list<User *>::iterator it = LU.begin(); it != LU.end(); ++it){
        delete (*it);
    }
    LU.clear();
}
```

Figura 10- Implementação da função Casino()

3.4 Listar

3.4.1 Declaração

A função Listar(),*Figura 12*, tem como objetivo exibir o estado atual do casino, apresentando informações detalhadas relativamente a cada máquina presente no casino, tais como Id, nome, tipo, probabilidade de entregar prémio, estado e temperatura. Além disso, a função imprime estatísticas globais do casino, incluindo o número total de máquinas, jogadores no casino e o valor de dinheiro em caixa.

```
void Casino::Listar(ostream &f = std::cout){
```

Figura 11- Declaração da função Listar()

3.4.2 Implementação

A função inicia imprimindo na consola uma mensagem correspondente ao cabeçalho da função. Em seguida, a função percorre a lista de máquinas do casino, mostrando os dados de cada máquina e escreve num ficheiro fornecido como parâmetro. Enquanto percorre as listas de máquinas, utiliza uma formatação especifica para descartar visualmente o estado de cada máquina, utilizando cores diferentes para cada estado da máquina. Após listar as máquinas a função mostra dados do casino, como o número total de máquinas, jogadores presentes, dinheiro em caixa e dinheiro dado.

```
void Casino::Listar(ostream &f = std::cout){

    cout << "Listar estado atual do Casino" << endl;
    cout << endl; //escreve nova linha
    cout << "| ID Maquina | Nome Maquina | Probabilidade | Estado | temperatura |" << endl; //printa campos que vao ser mostrados

    f << "Estado Casino " << nome << endl; //escreve no ficheiro
    for (list<Maquina *>::iterator it = LM.begin(); it != LM.end(); it++) { //percorre todas as maquinas do casino
        //escreve no ficheiro os campos - id, nome, probabilidade, estado, temperatura
        f << "| ID: " << (*it)->getID() << " | Nome: " << (*it)->getNome() << " | Tipo: " << (*it)->getTipo() << " | Probabilidade: " << (*it)->getProb()
        << " | Estado: " << estadoString((*it)->getEstado()) << " | Temperatura: " << (*it)->getTemperaturaSensor() << endl;
        if((*it)->getEstado()== ON){ //verifica se o estado e ON para ajustar a cor do estado quando printa
            //printa dados
            cout << "| ID: " << (*it)->getID() << " | Nome: " << (*it)->getNome() << " | Tipo: " << (*it)->getTipo() << " | Probabilidade: " << (*it)->getProb()
            << " | Estado: \033[1;32m"<<estadoString((*it)->getEstado())<<"\033[0m" << " | Temperatura: " << (*it)->getTemperaturaSensor() << endl;
        }else if ((*it)->getEstado()== OFF){ //verifica se o estado  ON para ajustar a cor do estado quando printa
            //printa dados
            cout << "| ID: " << (*it)->getID() << " | Nome: " << (*it)->getNome() << " | Tipo: " << (*it)->getTipo() << " | Probabilidade: " << (*it)->getProb()
             << " | Estado: \033[1;33m"<<estadoString((*it)->getEstado())<<"\033[0m" << " | Temperatura: " << (*it)->getTemperaturaSensor() << endl;
        }else if ((*it)->getEstado()== AVARIADA){ //verifica se o estado e ON para ajustar a cor do estado quando printa
            //printa dados
            cout << "| ID: " << (*it)->getID() << " | Nome: " << (*it)->getNome() << " | Tipo: " << (*it)->getTipo() << " | Probabilidade: " << (*it)->getProb()
            << " | Estado: \033[1;31m"<<estadoString((*it)->getEstado())<<"\033[0m" << " | Temperatura: " << (*it)->getTemperaturaSensor() << endl;
        }
    }
    cout << endl;
    cout << "Numero de Maquinas: " << LM.size() << endl; //maximo de jogadores no casino
    cout << endl; //adiciona linha
    cout << "Jogadores no casino: " << jogadoresNoCasino << endl; //printa a quantidade de jogadores no casino atualmente
    cout << "Numero total de jogadores: " << LU.size() << endl; //printa a quantidade de jogadores que j passaram pelo casino
    cout << endl; //adiciona linha
    cout << "Dinheiro em caixa: " << totalCaixa << endl; //maximo de jogadores no casino
    cout << "Dinheiro dado: " << totalDinheiroDado << endl; //hora de abertura
}
```

Figura 12- Implementação da função Listar()

3.5 Listar_Tipo

3.5.1 Declaração

A função Listar_Tipo(), tal como o nome indica, é responsável por listar as máquinas de um tipo específico. Esta recebe como parâmetro o tipo de máquina desejado e um ficheiro, onde serão exibidas e guardadas as informações das máquinas. A função cria uma lista de máquinas do tipo específico, escreve as informações no ficheiro e retorna a lista.

```
list<Maquina *> * Casino::Listar_Tipo(string Tipo, std::ostream &f) {
```

Figura 13- Declaração da função Listar_Tipo()

3.5.2 Implementação

A função Listar_Tipo() começa gerando uma lista de ponteiros para máquinas, com o intuito de guardar as máquinas do tipo desejado. Prontamente, percorre a lista de máquinas do casino, adicionando as máquinas do tipo desejado à lista criada e escreve as informações no ficheiro fornecido. Para finalizar o processo, a função retorna a lista de máquinas do tipo.

```cpp
//listar maquinas de um dado tipo
list<Maquina *> * Casino::Listar_Tipo(string Tipo, std::ostream &f) {
    //Cria lista armazenar maquinas
    list<Maquina *> * maquinasDoTipo = new list<Maquina*>;
    for (list<Maquina *>::iterator it = LM.begin(); it != LM.end(); it++) {
        if ((*it)->getTipo() == Tipo) { //Verifica se a maquina atual tem o tipo desejado
            //Adiciona a maquina a lista
            maquinasDoTipo->push_back((*it));
            //Escreve informacoes sobre a maquina no ficheiro
            f << "ID: " << (*it)->getID() << " | Nome: " << (*it)->getNome() << " | Tipo: " << (*it)->getTipo() << endl;
        }
    }
    return maquinasDoTipo;
}
```

Figura 14-Implementação da função Listar_Tipo()

A função listarTipoMaquina() fornece suporte à interface, solicitando o tipo de máquina desejado. Gera um ficheiro para armazenar as informações das máquinas desse tipo.
Em caso de erro ao abrir o ficheiro, exibe uma mensagem de erro, caso contrário, chama a função principal, Listar_Tipo() para exibir as informações das máquinas do tipo específico. Ao finalizar o processo, a função liberta a memória alocada para a lista de máquinas do tipo e fecha o arquivo.

```cpp
//Funcao complementar para listar maquinas de um dado tipo
void Casino::listarTipoMaquina(){
    string Tipo; //Varivel do tipo string para guardar o tipo de maquina desejada
    cout << "Tipo de maquina: "; //Pede o tipo de maquina pretendida
    cin >> Tipo; //Guarda tipo de maquina desejada
    ofstream F("MaquinasTipo.txt"); //Abrir fecheiro
    if (!F.is_open()) { //Se ficheiro nao foi aberto corretamente
        cout << "Erro ao abrir o arquivo MaquinasTipo.txt" << endl;
    }else{
        list<Maquina*>* maquinasDoTipo = Listar_Tipo(Tipo, F); //Invoca funcao para listar maquinas de um dado tipo
        for (list<Maquina *>::iterator it = maquinasDoTipo->begin(); it != maquinasDoTipo->end(); it++) {
            if ((*it)->getTipo() == Tipo) {
                cout << "ID: " << (*it)->getID() << " | Nome: " << (*it)->getNome() << " | Tipo: " << (*it)->getTipo() << endl;
            }
        }
        delete maquinasDoTipo; //Liberta memoria alocada pela lista
        F.close(); //Fechar ficheiro
    }
}
```

Figura 15- Implementação da função listarTipoMaquina()

3.6 Ranking_Dos_Fracos

3.6.1 Declaração

A função Ranking_Dos_Fracos() retorna um ponteiro para uma lista de *strings* que representa o *ranking* das máquinas, com base no número de avarias. A lista é inicializada com uma cópia da lista original das máquinas e ordenada pelo número de avarias de forma decrescente. De seguida, são geradas *strings* formatadas, compostas pelas informações relevantes de cada máquina e adicionadas à lista de *rankingAvariadas*.

```cpp
list<string> * Casino::Ranking_Dos_Fracos()
```

Figura 16- Declaração da função Ranking_Dos_Fracos()

3.6.2 Implementação

A função Ranking_Dos_Fracos() cria uma lista de *strings* para armazenar as informações do *ranking*. Em seguida, cria uma cópia da lista original das máquinas ordenando-a com base no número de avarias de cada máquina. Dentro do *"for"*, são geradas *strings* formatadas para cada máquina ordenada e adicionadas à lista de *ranking*. Por fim a lista é retornada.

```
list<string> * Casino::Ranking_Dos_Fracos(){
    // Lista para armazenar as informaçoes das maquinas avariadas ordenadas por numero de avarias
    list<string>* rankingAvariadas = new list<string>;
    // Criar uma copia da lista principal
    list<Maquina*> maquinasCopy(LM.begin(), LM.end());
    // Ordenar a lista de maquinas por numero de avarias (do maior para o menor)
    maquinasCopy.sort([](Maquina* a, Maquina* b) {
        return a->getnAvarias() > b->getnAvarias();
    });
    // Preencher a lista de strings com informaçoes das maquinas ordenadas
    for (list<Maquina *>::iterator it = maquinasCopy.begin(); it != maquinasCopy.end(); ++it) {
        string info =    " |ID: " + to_string((*it)->getID()) +
                         " | Tipo: " + (*it)->getTipo() +
                         " | Nome: " + (*it)->getNome() +
                         " | Avarias: " + to_string((*it)->getnAvarias());
        rankingAvariadas->push_back(info);
    }
    return rankingAvariadas;
}
```

Figura 17- Implementação da função Ranking_Dos_Fracos()

A função showRankingAvarias() obtém a lista do *ranking* criado na função Ranking_Dos_Fracos() e, se houver máquinas avariadas, exibe-as na consola. A lista de *ranking* é liberada da memória após a exibição. A função showRankingAvarias() utiliza a função Ranking_Dos_Fracos() para obter o *ranking* e, em seguida, exibe as informações formatadas das máquinas avariadas.
Caso não haja máquinas avariadas, uma mensagem de erro é apresentada. A lista de *ranking* é libertada da memória após a utilização.

```
void Casino:: showRankingAvarias(){
    // Chamar a funcao para obter o ranking das m□quinas avariadas
    list<string>* rankingAvariadas = Ranking_Dos_Fracos();
    // Verificar se a lista nao esta vazia antes de tentar acessar os elementos
    if (rankingAvariadas->size() <= 0) {
        cout << "Nenhuma maquina avariada encontrada." << endl;
    } else {
        // Percorrer e exibir as informacoes das maquinas no ranking
        for (list<string>::iterator it = rankingAvariadas->begin(); it != rankingAvariadas->end(); ++it) {
            cout << (*it) << endl;
        }
    }
    delete rankingAvariadas; //Liberta memoria alocada pela lista
}
```

Figura 18- Implementação da função showRankingAvarias()

3.7 Ranking_Das_Mais_Trabalhadores

3.7.1 Declaração

```
list<Maquina *> *Casino::Ranking_Das_Mais_Trabalhadores()
```

Figura 19- Declaração da função Ranking_Das_Mais_Trabalhadores()

A função Ranking_Das_Mais_Trabalhadores() é responsável por gerar um *ranking* das máquinas com base no número de jogos realizados. Esta cria uma cópia da lista de máquinas do casino, ordena por ordem decrescente de acordo com o número de jogos realizados por cada máquina e retorna a lista.

3.7.2 Implementação

A função Ranking_Das_Mais_Trabalhadores() produz uma cópia da lista de máquinas do casino, utilizando o construtor que aceita o início e o final da lista original. Em seguida, esta é ordenada com base nos jogos realizados. Por fim a função retorna a lista ordenada, que é utilizada pela função listarRankingMaisTrabalhadores().

```
//Ranking dos mais trabalhadores - as que sao usadas
list<Maquina *> *Casino::Ranking_Das_Mais_Trabalhadores() {
    //Crie uma copia da lista de maquinas
    list<Maquina *> * copiaMaquinas = new list<Maquina *>(LM.begin(), LM.end());
    //Ordene a lista usando a funcao de comparacao
    copiaMaquinas->sort([](Maquina* a, Maquina* b) {
        return a->getNJogos() > b->getNJogos();
    });
    return copiaMaquinas;
}
```

Figura 20- Implementação da função Ranking_Das_Mais_Trabalhadores()

A Figura 21Figura 20, listarRankingMaisTrabalhadores(), representa a função aplicada para listar as máquinas mais utilizadas no casino. Esta função cria uma cópia do *ranking* das máquinas mais trabalhadoras, percorre-a e imprime as informações, como Id, nome, tipo e número de jogos. Após isso a memória alocada a lista é libertada.

```
void Casino::listarRankingMaisTrabalhadores(){
    list<Maquina*>* maquinasMaisTrabalhadores = Ranking_Das_Mais_Trabalhadores();
    for (list<Maquina *>::iterator it = maquinasMaisTrabalhadores->begin(); it != maquinasMaisTrabalhadores->end(); ++it) {
        cout << "ID: " << (*it)->getID() << " | Nome: " << (*it)->getNome() << " | Tipo: " << (*it)->getTipo() << " | Numero de jogos: " << (*it)->getNJogos() << endl;
    }
    delete maquinasMaisTrabalhadores; //Liberta memoria alocada pela lista
}
```

Figura 21- Implementação da função listarRankingMaisTrabalhadores()

3.8 Jogadores_Mais_Frequentes

3.8.1 Declaração

A função Jogadores_Mais_Frequentes() tem como objetivo gerar um *ranking* dos jogadores mais frequentes no casino com base no tempo total que frequentaram as máquinas.

```
list<User *> * Casino::Jogadores_Mais_Frequentes ()
```

Figura 22- Declaração da função Jogadores_Mais_Frequentes()

3.8.2 Implementação

A função, *Figura 23*, produz uma cópia da lista de *Users* do casino, recorrendo ao construtor que aceita o início e o fim da lista original. De seguida ordena esta cópia com base no tempo que o *user* passou no casino a jogar, ordenando-a de forma decrescente. Por fim, retorna uma lista ordenada, utilizada pela função listarjogadoresMaisFrequentes().

```
//jogadores mais frequentes - jogadores que mais tempo passaram a jogar
list<User *> * Casino::Jogadores_Mais_Frequentes (){
    // Crie uma copia da lista de maquinas
    list<User *> *copiaUser = new list<User *>(LU.begin(), LU.end());
    copiaUser->sort([](User* a, User* b) {
        return a->getTempoCasino() > b->getTempoCasino();
    });
    return copiaUser;
}
```

Figura 23- Implementação da função Jogadores_Mais_Frequentes()

Esta função, *Figura 24* , cria uma cópia da lista retornada, percorre-a e imprime na consola dados como, Id, nome e tempo total em minutos. Por último, a memória alocada a lista é libertada.

```
void Casino::listarJogadoresMaisFrequentes(){
    list<User*>* copiaUser = Jogadores_Mais_Frequentes();
    for (list<User *>::iterator it = copiaUser->begin(); it != copiaUser->end(); ++it) {
        cout << "ID: " << (*it)->getNUser() << " | Nome: " << (*it)->getNome() << " | tempo casino: " << (*it)->getTempoCasino() << endl;
    }
    delete copiaUser; //Liberta memoria alocada pela lista
}
```

Figura 24- Implementação da função listarjogadoresMaisFrequentes()

3.9 Jogadores_Mais_Ganhos

3.9.1 Declaração

A função jogadores_Mais_Ganhos() tem como objetivo gerar um *ranking* dos jogadores que mais dinheiro ganharam no casino com base no premio das máquinas.

```
list<User *> * Casino::Jogadores_Mais_Ganhos ()
```

Figura 25- Declaração da função jogadores_Mais_Ganhos()

3.9.2 Implementação

A função, *Figura 26*, cria uma cópia da lista de *Users* do casino, utilizando o construtor que aceita o início e o fim da lista original, de seguida ordena esta cópia com base no dinheiro ganho pelo jogador na máquina onde jogou, ordenando-a de forma decrescente. Por fim, a função retorna uma lista ordenada, utilizada pela função listarJogadoresMaisGanhos().

```
//jogadores que ganharam mais dinheiro
list<User *> * Casino::Jogadores_Mais_Ganhos () {
    // Crie uma copia da lista de maquinas
    list<User *> *copiaUser = new list<User *>(LU.begin(), LU.end());
    copiaUser->sort([](User* a, User* b) {
        return a->getGanhos() > b->getGanhos();
    });
    return copiaUser;
}
```

Figura 26- Implementação da função jogadores_Mais_Ganhos()

Esta função, *Figura 27*, cria uma cópia da lista retornada, percorre-a e imprime na consola dados como, Id, nome e total de ganhos, de seguida a memória alocada a lista é libertada.

```
void Casino::listarJogadoresMaisGanhos(){
    list<User*>* jogadoresMaisGanhos = Jogadores_Mais_Ganhos();
    for (list<User *>::iterator it = jogadoresMaisGanhos->begin(); it != jogadoresMaisGanhos->end(); ++it) {
        cout << "ID: " << (*it)->getNUser() << " | Nome: " << (*it)->getNome() << " | ganhos: " << (*it)->getGanhos() << endl;
    }
    delete jogadoresMaisGanhos; //Liberta memoria alocada pela lista
}
```

Figura 27- Implementação da função listarJogadoresMaisGanhos()

3.10 Desligar

3.10.1 Declaração

A função Desligar(), tem como objetivo desligar uma máquina do casino com base no Id recebido por parâmetro. Esta percorre a lista de máquinas do casino, procurando a máquina com o Id correspondente e posteriormente desligar a máquina.

```
void Casino::Desligar(int id_maq)
```

Figura 28- Declaração da função Desligar()

3.10.2 Implementação

A função percorre a lista de máquinas "LM" e compara o Id de cada máquina com o Id fornecido como parâmetro. Se a máquina corresponder, o método Desligar() da classe "Máquina" é chamado, e a variável desligada é definida como verdadeira.
Após o *loop*, a função verifica se a máquina foi desligada com sucesso. Se não, imprime a mensagem" máquina não encontrada".

```
void Casino::Desligar(int id_maq) { //recebe o id da maquina que pretendem desligar
    bool desligada = false;
    for (list<Maquina *>::iterator it = LM.begin(); it != LM.end(); it++) { //percorre todas as maquinas do casino
        if((*it)->getID() == id_maq){ //se id maquina for igual ao id da maquina que pretendem desligar
            (*it)->Desligar(); //chama funcao para desligar maquina
            desligada = true;
            cout << "Maquina " << (*it)->getNome() << " desligada." << endl;
        }
    }
    if(desligada == false){ //Verifica se a maquina desejada foi desligada
        cout << "Maquina nao encontrada." << endl; //se nao encontrar a maquina desejada mostra mensagem de "erro"
    }
}
```

Figura 29- Implementação da função Desligar()

3.11 Estado

3.11.1 Declaração

A função Get_Estado() tem como propósito fornecer o estado atual de uma máquina com base no id fornecido como parâmetro. Esta por sua vez é chamada dentro da função estadoString(), para converter o estado da máquina para uma *string*.

```
estadoMaquina Casino::Get_Estado(int id_maq)
```

Figura 30- Declaração da função Get_Estado()

3.11.2 Implementação

A função Get_Estado(), percorre a lista de máquinas do casino, verificando se o Id fornecido corresponde a alguma máquina. Se encontrar, retorna o estado da máquina, caso contrário, apresenta uma mensagem de erro e retorna "OFF".

```
//Saber o estado de uma dada Maquina dado o seu ID
estadoMaquina Casino::Get_Estado(int id_maq) { //recebe id da maquina desejada
    bool maquina = false;
    for (list<Maquina *>::iterator it = LM.begin(); it != LM.end(); it++) { //percorre todas as maquinas do casino
        if((*it)->getID() == id_maq){ //se id maquina for igual ao id da maquina que pretendem desligar
            maquina = true;
        }
        if(maquina){
            return (*it)->getEstado(); //returna estado da maquina
        }
    }
    if(!maquina){
        cout << "Maquina nao encontrada." << endl; //se nao encontrar a maquina desejada mostra mensagem de "erro"
        return estadoMaquina::OFF; //e retorna estado OFF
    }
    return estadoMaquina::OFF; //e retorna estado OFF
}
```

Figura 31- Implementação da função Get_Estado()

A função estadoString(), *Figura 32*, utiliza o valor retornado por Get_Estado() e converte-o numa *string*, utilizando um s*witch* para associar o estado ao seu equivalente em *string.* A criação desta função auxiliar surge pela necessidade de transformar o "*enum"* retornado por Get_Estado() (que representa valores inteiros) em uma *string*, com o intuito de facilitar a leitura do estado.

```
//Traduzir estado para string
string Casino::estadoString(estadoMaquina estado){ //Recebe um valor do tipo estadoMaquina
    string estadoString; //Variavel que guarda o valor como string
    switch (estado) { //Verifica estado
        case ON:
            // Se o estado for ON, atribui a string "ON" a varivel estadoString
            estadoString = "ON";
            break;
        case OFF:
            //Se o estado for OFF, atribui a string "OFF" a varivel estadoString
            estadoString = "OFF";
            break;
        case AVARIADA:
            //Se o estado for AVARIADA, atribui a string "AVARIADA" a varivel estadoString
            estadoString = "AVARIADA";
            break;
        default:
            //Se o estado nao corresponder a nenhum dos casos anteriores, atribui a string "Unknown" a varivel estadoString
            estadoString = "Unknown";
            break;
    }
    return estadoString;
}
```

Figura 32- Implementação da função estado_String()

3.12 Memoria_Total

3.12.1 Declaração

A função tem como finalidade calcular a quantidade total de memória ocupada pelo programa. Para isso considerou-se os tamanhos dos membros estáticos e dinâmicos da classe "casino", incluindo as listas de máquinas e *users*, bem como as memórias associadas a cada máquina e user presentes nas listas.

```
int Casino::Memoria_Total()
```

Figura 33- Declaração da função Memoria_Total()

3.12.2 Implementação

A função inicia a variável “mem” com o tamanho do objeto “casino”, considerando a memória ocupada pelos membros estáticos da classe. Em seguida, adiciona o tamanho da *string* “nome”, a quantidade de máquinas na lista “LM” multiplicada pelo tamanho do ponteiro para “Maquina” individualmente através da função Memoria() de cada uma. Após calcular a memória associada às máquinas, a função considera a memória associada aos *users* presentes na lista “LU”. Isso inclui o tamanho da lista de ponteiros dos *users* e a memória associada a cada user individualmente através da função Memoria(). Por fim a função imprime a memória total calculada e retorna esse valor.

```
//memoria toral do programa
int Casino::Memoria_Total() {
    int mem = 0;
    mem = sizeof(*this);
    mem += nome.size();
    mem += LM.size() * sizeof(Maquina*);
    for (list<Maquina *>::iterator it = LM.begin(); it != LM.end(); ++it) {
        //mem += sizeof(*it);
        mem += (*it)->Memoria();
    }
    // Adicione a memoria associada a membros dinamicos, se houver
    mem += sizeof(User*) * LU.size(); // Tamanho da lista de ponteiros de usuarios
    // Itera sobre os usuarios e adiciona a memoria associada a cada um
    for (list<User*>::iterator it = LU.begin(); it != LU.end(); ++it){
        mem += (*it)->Memoria();
    }
    cout << "Memoria total Casino: " << mem << endl;
    return mem;
}
```

Figura 34- Implementação da função Memoria_Total()

3.13 Relatorio

3.13.1 Declaração

A função Relatorio() tem como propósito gerar um relatório em formato XML contendo informações sobre as máquinas presentes no casino. Recebendo como parâmetro o nome

do ficheiro XML, a função utiliza a classe “XMLWriter” para criar e escrever no documento. Durante a execução, percorre a lista de máquinas do casino, obtendo dados como Id, tipo e estado de cada máquina, e posteriormente insere-os no ficheiro XML.

```
void Casino::Relatorio(string fich_xml)
```

Figura 35- Declaração da função Relatorio()

3.13.2 Implementação

A função inicia, ao imprimir o cabeçalho com o objetivo da função, de seguida mostra o nome do arquivo recebido por parâmetro. Em seguida, é criada uma instância da classe “*XMLWritter*” com o nome “XX”. Podendo utilizar os métodos dessa classe, como “*WriteStartDocument*” e “*WriteStartElement*”, o documento XML é inicializado e o elemento raiz “RELATORIO” é aberto. De imediato percorre a lista de máquinas do casino, e para cada máquina, é aberto o elemento “Maquina”, e são inseridos os elementos “ID”, “TIPO” e “ESTADO”, com suas respetivas informações. Ao final de cada iteração, o elemento “Maquina” é fechado. Após o loop o elemento raiz “RELATORIO” é fechado, concluindo o documento XML.

```
//Relatorio casino - informacao do estado atual de cada maquina
void Casino::Relatorio(string fich_xml) {
    cout << "Gerar Relatorio" << endl; //Printa cabecalho da funcao
    cout<< "Nome ficheiro: " << fich_xml <<endl; //Printa nome do ficheiro
    XMLWriter XX; //Cria uma instancia de XMLWriter para escrever no arquivo XML
    XX.WriteStartDocument(fich_xml); //Escreve no documanto XML com o nome passado por parametro
    XX.WriteStartElement("RELATORIO"); //Abre o elemento "Relatorio"
    for (list<Maquina *>::iterator it = LM.begin(); it != LM.end(); it++) { //Percorre as maquinas do casino
        XX.WriteStartElement("Maquina"); //Abre o elemento Maquina
            XX.WriteElementString("ID",to_string((*it)->getID())); //Escreve no elemento "ID" com o id da maquina
            XX.WriteElementString("TIPO",(*it)->getTipo()); //Escreve no elemento "TIPO" com o tipo da maquina
            XX.WriteElementString("ESTADO",estadoString((*it)->getEstado())); //Escreve no elemento "ESTADO" com o estado da maquina
        XX.WriteEndElement(); //Fecha o Elemento Maquina
    }
    XX.WriteEndElement(); //Fecha o Elemento RELATORIO
}
```

Figura 36- Implementação da função Relatorio()

3.14 SubirProbabilidadeVizinhas

3.14.1 Declaração

A função SubirProbabilidadeVizinhas() recebe o ponteiro da máquina que ganhou, um valor R (raio), e a lista de máquinas vizinhas da máquina que ganhou. Ela aumenta a probabilidade de todas as máquinas vizinhas que estão dentro do raio R em relação à máquina que ganhou. A função mostrarMaquinasAfetadas() é chamada para mostrar as máquinas afetadas pela subida da probabilidade.

```
void Casino::SubirProbabilidadeVizinhas(Maquina *M_ganhou, float R, list<Maquina *> &lmvizinhas)
```

Figura 37- Declaração da função SubirProbabilidadeVizinhas()

3.14.2 Implementação

A função SubirProbabilidadeVizinhas() percorre a lista de máquinas vizinhas e verifica se a distância em relação à máquina que ganhou é menor ou igual ao raio R. Se a condição for atendida, a probabilidade da máquina vizinha é aumentada. As máquinas afetadas são armazenadas em uma lista, e a função mostrarMaquinasAfetadas() é chamada para exibir essas máquinas.

```
//subir probabilidade das maquinas vizinhas da maquina que ganhou
void Casino::SubirProbabilidadeVizinhas(Maquina *M_ganhou, float R, list<Maquina *> &lmvizinhas) {
    // Lista para armazenar maquinas afetadas
    list<Maquina *> maquinasAfetadas;
    for (list<Maquina *>::iterator it = lmvizinhas.begin(); it != lmvizinhas.end(); ++it) {
        int distanciaY = abs((*it)->getY() - M_ganhou->getY());
        if (distanciaY <= R) {
            float probAt = (*it)->getProb();
            (*it)->setProb(probAt + 1.0);
            maquinasAfetadas.push_back((*it));
        }
    }
    mostrarMaquinasAfetadas(maquinasAfetadas);
}
```

Figura 38- Implementação da função SubirProbabilidadeVizinhas()

A função mostrarMaquinasAfetadas(),*Figura 39*, exibe na consola as máquinas afetadas, mostrando o Id e a nova probabilidade de cada máquina.

```
void Casino::mostrarMaquinasAfetadas(list<Maquina *> &maquinasAfetadas) {
    cout << "Maquinas afetadas:" << endl;
    for (list<Maquina *>::iterator it = maquinasAfetadas.begin(); it != maquinasAfetadas.end(); ++it) {
        cout << "ID: " << (*it)->getID() << " | Probabilidade: " << (*it)->getProb() << endl;
    }
}
```

Figura 39- Implementação da função mostrarMaquinasAfetadas()

3.15 Listar()

3.15.1 Declaração

A função Listar() é invocada pela função auxiliar ListarMaquinasProbabilidade(), recebendo como parâmetro a probabilidade a ser pesquisada e o ficheiro onde os resultados da pesquisa serão armazenados.

```
void Casino::Listar(float X, ostream &f)
```

Figura 40- Declaração da função Listar()

3.15.2 Implementação

A função tem como objetivo listar todas as máquinas com probabilidade superior àquela passada por parâmetro. Inicialmente, o valor da probabilidade escolhida é registado no ficheiro. Em seguida, a função percorre a lista de máquinas, comparando a probabilidade de cada máquina com a probabilidade desejada. Se a afirmação for verdadeira, o Id e a probabilidade da máquina são registados no arquivo.

```
//Listar maquinas com probabilidade de ganhar maior que X
void Casino::Listar(float X, ostream &f) { //Recebe um float com a probabilidade desejada e uma ref
    f << "Maquinas com probabilidade " << X << " maior que de ganhar:" << endl; //Escreve no fichei
    for (list<Maquina *>::iterator it = LM.begin(); it != LM.end(); it++) { //Percorre todas as maq
        if ((*it)->getProb() > X) { //Se a probabilidade maquina for superior a probabilidade desej
            f << "ID: " << (*it)->getID() << " | Probabilidade: " << (*it)->getProb() << endl; //Es
            cout << "ID: " << (*it)->getID() << " | Probabilidade: " << (*it)->getProb() << endl; /
        }
    }
}
```

Figura 41- Implementação da função Listar()

3.16 Run

3.16.1 Declaração

A função Run() é a principal do programa e gere o funcionamento do casino. Esta verifica o horário de funcionamento do casino, chama a função que irá tratar do funcionamento das máquinas, permite a entrada de jogadores, chama a função que irá tratar do funcionamento dos *users* e realiza verificações relacionadas à fila de espera e à saída de jogadores. Além disso, permite a abertura do menu ao pressionar a tecla 'm' ou 'M'. A função também gera relatórios ao fechar o casino.

```
void Casino::Run(){
```

Figura 42- Declaração da função Run()

3.16.2 Implementação

A função começa por obter o horário atual e verifica se o casino está aberto ou fechado. Enquanto o casino está aberto executa várias operações, como avariar e correr as máquinas, permitindo a entrada de jogadores, associando jogadores as máquinas disponíveis, permitindo também os adicionar a fila de espera, caso a máquina esteja ocupada, por fim a função corre os jogadores, para que estes possam jogar nas máquinas associadas. Caso o casino esteja fechado, a função gera um relatório e desliga as máquinas, caso ainda estejam ligadas. Permite também pressionar a tecla 'm' ou 'M' a qualquer momento para abrir o menu. A função apresenta um *loop* infinito, garantindo que o programa continue em execução, podendo observar-se as ações dos jogadores, estado das máquinas e do casino.

```cpp
//Funçao Run Casino
void Casino::Run(){
    char key;
    bool maquinasJaLigadas = false;
    bool casinoEncerrado = false;
    while(true){
        cout<< "Menu" <<endl;
        time_t now = time(nullptr);
            tm* current_time = localtime(&now);

        int currentHour = current_time->tm_hour;//Hora atual
        int currentMinute = current_time->tm_min;//Minuto atual
        int currentSecond = current_time->tm_sec;//Segundo atual

        int openingTimeInSeconds = horaAbertura * 3600 + minutosAbertura * 60 + segundosAbertura;// Converte o horario de abertura para segundos
        int closingTimeInSeconds = horaFecho * 3600 + minutosFecho * 60 + segundosFecho;// Converte o horario de fechamento para segundos
        int currentTimeInSeconds = currentHour * 3600 + currentMinute * 60 + currentSecond;// Converte o horario atual para segundos

        if (currentTimeInSeconds >= openingTimeInSeconds && currentTimeInSeconds <= closingTimeInSeconds) {

                    // O casino esta aberto!
                    cout << "O casino esta aberto!" << endl;
                    casinoEncerrado = false;
                    if (!maquinasJaLigadas) { //Verifica se ja tem todas as maquinas ligadas
                        ligarTodasMaquinas(); //Liga todas as maquinas
                        maquinasJaLigadas = true; //Guarda estado como ja as ligou
                    }
                    // Todas as maquinas existentes passam pela funçao avariar e corre maquina
                    for (list<Maquina *>::iterator it = LM.begin(); it != LM.end(); ++it){
                        avariar((*it)->getID()); //Funçao avariar
                        (*it)->Run(); //Corre maquina
                    }

                    // Verifica se o Casino esta cheio  ou se podem entrar jogadores
                    if(jogadoresNoCasino < maxJogadores){
                            if(entrarJogador()){ //Verifica se entrou algum jogador
                                User *user = userEntraCasino("pessoas.txt"); //Guarda no apontador user o user retornado pela funçao userEntraCasino
                                if (Add(user)){ //Adiciona User
                                    cout << endl;
                                    cout << "Jogador entrou no Casino com sucesso!" << endl;
                                    jogadoresNoCasino ++; //Incrementa mais um jogador
                                    int randomIndex = rand() % LM.size(); //Vai buscar um numero random entre 0 e tamanho da lista
                                    auto it = next(LM.begin(), randomIndex); //o next faz com que se obtenha a posiçao correta da maquina
                                    //o auto it serve para ir buscar o tipo correto de dados que neste caso a maquina
                                    Maquina *maquina = *it; //guarda no apontador maquina o ponteiro it referente ao objeto amquina radnom
                                    if(maquina->getUserAtual()!=nullptr){ //Verifica se a maquina escolhida anteriormente esta a ser usada
                                        user->entrarFilaEspera(maquina); //Se estiver o user entra na fila de espera
                                    }else{
                                        user->associarMaquina(maquina); //se nao estiver o user associa-se a uma maquina
                                    }
                                    user->mostrarDados(); //Mostra dados do jogador
                                }else{
                                    cout << "Erro: Este jogador nao entrou pois esta a nulo." << endl; //mensagem de erro caso o usser seja invalido
                                }
                            }
                    }
```

```
            //Se existirem mais do que 1 user percorre a lista de useres
            if(LU.size()>=1){
                for (list<User *>::iterator it = LU.begin(); it != LU.end(); ++it){
                        cout << endl;
                        (*it)->Run(); //correr user
                        cout << endl;
                }
            }

            // Verifica se saiu algum user
            for (list<Maquina *>::iterator it = LM.begin(); it != LM.end(); ++it) {
                if ((*it)->getUserAtual()==nullptr) { //verifica se a maqina atual nao possui nenhum jogador
                    if (!(*it)->getFilaEspera().empty()) { //verifica se lista de espera da maquina nao esta vazia
                        User *useruser = (*it)->getFilaEspera().front(); //Vai buscar o user a frente na lista de espera da maquina
                        useruser->associarMaquina(*it); //user associa-se a maquina
                        (*it)->removerUsersFilaEspera(useruser); //e a retirado da lista de espera

                        cout << "User: " <<  useruser->getNome() << " saiu da fila de espera e sentou-se na maquina" << endl;

                    }else{
                        (*it)->setUserAtual(nullptr); //se n□o houver ninguem na lisat de espera o useratual da maquinafica a null
                    }
                }
            }

        } else {
            cout << "O casino esta fechado. Aguardando o horario de abertura." << endl;

            if (!casinoEncerrado){
                string nome = devolveData();
                Relatorio(nome);
                casinoEncerrado = true;
                if (maquinasJaLigadas) { //Se casino fechado ent□o verifica se as amquina ainda est□o ligadas
                    desligarTodasMaquinas(); //Desliga todas as maquina
                    maquinasJaLigadas = false; //Guarda estado como j□ desligou
                }
```

Figura 43- Implementação da função Run()

4 Conclusão

Com a conclusão do projeto prático da unidade curricular de Programação Orientada a Objetos, explorou-se os conceitos fundamentais como manipulação de ficheiros, listas, e gestão de memória A possibilidade colocar em prática o que é lecionado consente uma perspetiva diferente em relação aos tópicos, sentir as limitações, desafios e conquistas foi um incentivo para a investigação e desenvolvimento deste projeto

No decorrer do projeto o grupo sentiu certas dificuldades. A complexidade do que era pretendido e os requisitos específicos foram originando contratempos ao longo do desenvolvimento. Contudo o projeto foi concluído, cumpre os requisitos e encontra-se funcional.

Para trabalho futuro, seria interessante otimizar determinados conceitos e métodos de implementação, de forma a fornecer uma experiência mais otimizada ao utilizador, amplificando a sua interatividade com a simulação.