UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO E ENGENHARIA DE COMPUTAÇÃO

PIETRA FREITAS
THAYNÁ MINUZZO

Relatório Parcial - Trabalho Final de MLP
Tower Defense em C# - Fullbar

Relatório apresentado como requisito parcial para a obtenção de conceito na Disciplina de Modelos de Linguagens de Programação

Prof. Dr. Lucas Mello Schnorr
Orientador

Porto Alegre
2017

SUMÁRIO

1 INTRODUÇÃO

O objetivo deste trabalho consiste na exploração dos paradigmas orientado a objetos e funcional dado um problema e uma linguagem, escolhidos dentre os sugeridos pelo professor, a fim de comparar as características e funcionalidades de tal linguagem de programação em tais paradigmas.

1.1 Introdução ao problema

Tem-se como problema base a ser abordado, a implementação de um jogo do tipo Tower Defense através dos conhecimentos sobre paradigmas obtidos ao longo do semestre. A ideia básica é executar a defesa de algum elemento presente na tela, fazendo uso de recursos contidos no jogo, que auxiliem nesse trabalho.

1.2 Introdução à linguagem

Para a abordagem do problema, foi escolhida a linguagem de programação C#, que possui múltiplos paradigmas de programação. Serão utilizados os paradigmas de orientação a objetos e funcional, presentes na referida linguagem, juntamente com *frameworks* suportados por ela para auxílio da implementação do problema.

1.3 Introdução ao *framework* Monogame

Para o desenvolvimento do trabalho, utilizamos o *framework*, para desenvolvimento de jogos, Monogame - uma implementação *open-source* do *framework* XNA. Com ele, é possível desenvolver jogos em C# com maior facilidade.

2 VISÃO GERAL DA LINGUAGEM

2.1 Apresentação da linguagem escolhida

A linguagem escolhida, C#, é uma linguagem influenciada por diversas outras linguagens, como C++ e JAVA. Contudo, durante sua criação, o objetivo consistia em unir os benefícios oferecidos por outras linguagens. É possível observar isso analisando, por exemplo, sua sintaxe, que é muito similar à sintaxe da linguagem C++ e, portanto, uma sintaxe simples e de fácil aprendizagem.

Além disso, é uma linguagem fortemente tipada e apresenta recursos significativos como tipos de valor nulo, enumerações, delegações, expressões lambdas, acesso direto à memória e suporte a métodos e tipos genéricos. Por ser uma linguagem orientada a objetos, é possível utilizar os recursos associados a esse paradigma, como encapsulamento, herança e polimorfismo.

Nesse sentido, uma de suas aplicações está no *framework* .NET, em que foi utilizada na maioria das classes. Também possui grande aplicação na engine de games Unity. O trabalho desenvolvido no paradigma OO utiliza o *framework Monogame*, portanto, mais uma aplicação de C#.

3 ANÁLISE CRÍTICA

3.1 Critérios e Propriedades

3.1.1 Tabela de Critérios e Propriedades

Critério ou Propriedade	Nota
Simplicidade	9
Ortogonalidade	6
Expressividade	9
Adequabilidade e Variedade de Estruturas de Controle	9
Mecanismos de Definição de Tipos	8
Suporte a Abstração de Dados e de Processos	9
Modelo de Tipos	7
Portabilidade	8
Reusabilidade	8
Suporte à Documentação	10
Tamanho do Código	8
Generalidade	8
Eficiência e Custo	7

3.1.2 Simplicidade

A linguagem adotada tem um bom potencial quando o quesito avaliado é a simplicidade de código, pois é possível fazer uso de representações precisas, sem ambiguidades. Isso está relacionado também com o fator de ortogonalidade, que é pouco presente, mas que quando se encontra com frequência em uma determinada linguagem pode gerar tais ambiguidades. De fato, C# se caracteriza como uma linguagem que possui poucas exceções às regras de construção sintática. Semanticamente, foi implementada uma classe de constantes, o que torna o código simples de entender independentemente do contexto em que ele se encontra.

```
class Constants
{
```

```
/* ABOUT THE GAME */
static public int MAX_ENEMIES = 3;


/* ABOUT THE ENEMIES */
static public int ENEMY_SPAWN_TIME = 4;
static public int ENEMY_START_HEALTH = 100;
static public float ENEMY_SPEED = 1f;
```

Listing 3.1 – Trecho de código C# retirado da implementação deste trabalho

```
protected void LoadEnemies()
{
if (spawn >= Constants.ENEMY_SPAWN_TIME) // Respawns an enemy
   each ENEMY_SPAWN_TIME
{
spawn = 0;
if (enemies.Count <= Constants.MAX_ENEMIES) // Limits the
   respawn
{
Enemy enemy = new Enemy(enemyTextures[random.Next(0,
   enemyTextures.Count)], Vector2.Zero);
enemy.SetWaypoints(map.GetWaypoints());

enemies.Add(enemy);
```

Listing 3.2 – Trecho de código C# retirado da implementação deste trabalho

3.1.3 Ortogonalidade

Existem divergências acerca da questão ortogonalidade para a linguagem escolhida. Por um lado, ela se torna ortogonal por não apresentar exceções no escopo que define seu sistema de tipos. Por outro, ortogonalidade significa apresentar atribuições por intermédio de diferentes tipos de combinações possíveis e ainda gerar resultados coerentes. Seguindo esse segundo princípio, a ortogonalidade de C é fraca, principalmente pelo fato de ser fortemente tipada.

3.1.4 Expressividade

A linguagem possui diversas instruções que tornam mais expressivas a escrita dos comandos, através da definição de operadores que abrangem uma grande quantidade de computação.Temos como exemplo os operadores ++ ou - - , que substituem o uso de x = x + 1, por exemplo, garantindo a expressividade. O uso de estruturas de controle que permitem uma fácil expressão do que se quer representar, como o foreach, também são considerados pontos positivos em relação à expressividade do código. Seguem abaixo exemplos de ambas implementações.

```
 protected void UpdateEnemies(GameTime gameTime)
        {
            foreach (Enemy enemy in enemies)
                enemy.Update(gameTime);

        }

for (int i = 0; i < enemies.Count; i++)
            {
                if (enemies[i].GetOutOfScreen()) // Removes
                   enemies out of screen
                {
                    enemies.RemoveAt(i);
                    i--;
             ...
```

Listing 3.3 – Trecho de código C# retirado da implementação deste trabalho

3.1.5 Adequabilidade e variedade de estruturas de controle

A linguagem escolhida se mostra bastante completa quando o quesito é estruturas de controle. Durante a codificação, exploramos isso através de sub-rotinas, estruturas sequenciais, estruturas de seleção e estruturas de repetição, conforme a necessidade e demanda do domínio do problema. Segue um exemplo do uso de sub-rotinas.

```
protected override void Update(GameTime gameTime)
```

```
        {
                spawn +=
                    (float)gameTime.ElapsedGameTime.TotalSeconds;
                KeyboardHandler();

                UpdateEnemies(gameTime);

                LoadEnemies();

                player.Update(gameTime, enemies);

               base.Update(gameTime);
        }
```

Listing 3.4 – Trecho de código C# retirado da implementação deste trabalho

3.1.6 Mecanismos de definição de tipos

Por utilizar o paradigma OO, C# tem a possibilidade da criação de classes. Além disso, é possível criar structs. Portanto, é uma linguagem que conta com mecanismos de definição de tipo.

3.1.7 Suporte a abstração de dados e de processos

Por se tratar de uma linguagem orientada a objetos, ela oferece mecanismos básicos para tal, como herança, encapsulamento, e polimorfismo. A respeito da abstração de processos, observa-se que seu uso se torna algo quase essencial através de subprogramas, para modularização e legibilidade do código como um todo. Ambos os tipos de abstração estão bastante presentes no software desenvolvido, que apresenta onze classes distintas com níveis de herança entre si, todas elas encapsuladas (têm em sua estrutura métodos setters e getters, e seus atributos são protegidos). Os subprogramas existem tanto como métodos de classes, quanto como funções que auxiliam na classe principal. Segue um exemplo que demonstra uma das classes com seu devido encapsulamento, tendo seus atributos protegidos, sendo acessíveis apenas por métodos setters e getters.

```
namespace MonoGame2D
{
    abstract public class Tower : Sprite
        {
                /* Attributes */
        protected float bulletTimer; // How long ago was a
           bullet fire
                protected List<Bullet> bulletList = new
                   List<Bullet>();
        protected Enemy target;

        protected int cost; // How much will the tower cost to
           make
        protected int damage; // The damage done to enemy's
        protected float radius; // How far the tower can shoot
                protected int bulletSpeed;
```

Listing 3.5 – Trecho de código C# retirado da implementação deste trabalho

```
/* Getters */
        public int GetCost()
                {
            return cost;
        }

        public int GetDamage()
        {
            return damage;
                }

        public float GetRadius()
        {
                        return radius;
                }
```

```
            /* Setters */
            public void SetCost(int cost)
            {
        this.cost = cost;
    }
```

Listing 3.6 – Trecho de código C# retirado da implementação deste trabalho

3.1.8 Modelo de tipos

A linguagem escolhida é fortemente tipada, portanto há checagem de tipo. Contudo, realiza coerção, o que diminui a confiabilidade do código, visto que podem haver coerções não desejadas e não detectadas pelo desenvolvedor à primeira vista.

3.1.9 Portabilidade

Tanto a linguagem quanto o *framework* utilizado, Monogame, podem ser considerados altamente portáveis através de ferramentas específicas. Para portar C# de forma genérica, usa-se uma implementação do *framework* .NET chamado Mono, que permite desenvolver até mesmo para dispositivos móveis. A respeito do Monogame, a documentação presente o descreve como *plataform-agnostic*. Nossa implementação foi desenvolvida especificamente para o sistema operacional Windows. Mas, através de alguns ajustes, poderia ser utilizada em outras plataformas.

3.1.10 Reusabilidade

É possível implementar códigos tão reusáveis quanto possíveis com a linguagem adotada, pois, além de ser orientada a objetos - o que permite reaproveitamento de código -, ela apresenta uma série de ferramentas para garantir modularidade, como por exemplo, delegates, templates, parametrização. O TowerDefense Game utiliza algumas dessas ferramentas, como, por exemplo, delegates. O reuso é implementado também em escopos menores, com os subprogramas para evitar repetições de código.

3.1.11 Suporte e Documentação

É possível encontrar a documentação da linguagem no site da Microsoft. Portanto, tem documentação de fácil acesso.

3.1.12 Tamanho de Código

Por ser bastante reutilizável, é possível implementar lógicas com certa complexidade, em C#, com poucas linhas de código.

3.1.13 Generalidade

C# apresenta mecanismos para a definição de estruturas de dados genéricas, portanto é uma linguagem genérica.

```
public class Stack<T>
{
   readonly int m_Size;
   int m_StackPointer = 0;
   T[] m_Items;
   public Stack():this(100)
   {}
   public Stack(int size)
   {
      m_Size = size;
      m_Items = new T[m_Size];
   }
   public void Push(T item)
   {
      if(m_StackPointer >= m_Size)
         throw new StackOverflowException();
      m_Items[m_StackPointer] = item;
      m_StackPointer++;
   }
```

```
    public T Pop()
    {
        m_StackPointer--;
        if(m_StackPointer >= 0)
        {
            return m_Items[m_StackPointer];
        }
        else
        {
            m_StackPointer = 0;
            throw new InvalidOperationException("Cannot pop an
                empty stack");
        }
    }
}
```

Listing 3.7 – Trecho de código C# retirado de documentação online da linguagem

3.1.14 Eficiência e Custo

A eficiência da linguagem está fortemente relacionada com a implementação da linguagem, que no nosso caso, se apresenta como híbrida, pois apesar de criar um arquivo com extensão executável, existe a necessidade de uma Maquina Virtual .NET para sua interpretação. Ganha-se assim em portabilidade também, pois o código pode ser exportado para plataformas que possuam a VM instalada. Porém, perde-se em performance para a execução. A linguagem possui um alto nível de abstração, logo o custo associado a treinamentos e desenvolvimento torna-se bastante baixo.

3.2 Analisando os paradigmas

–

4 SOBRE O JOGO

4.1 Acesso

O código encontra-se disponível no GitHub, no repositório: TowerDefenseGame.

4.2 Uso de Destrutores

Para a implementação do código, não foram usados destrutores de classe. Isso se deve parcialmente ao fato da existência de um método próprio do framework Monogame chamada UnloadContent(), que tem como objetivo descarregar todo o conteúdo previamente carregado quando ele deixa de ser usado, ou quando existe a invocação do método em questão.

5 CONCLUSÃO

Durante o desenvolvimento do trabalho, uma das grandes vantagens da linguagem escolhida foi sua semelhança, sintaticamente, com a linguagem C++ e, portanto, com a linguagem C, linguagens com as quais as integrantes já estavam familiarizadas.

Além disso, C# é uma linguagem com paradigma OO. Portanto, a estruturação do código utilizando os recursos desse paradigma aconteceu naturalmente à medida em que o código era incrementado.

Em relação à linguagem, não houve grandes dificuldades. O maior desafio consistiu na utilização do *framework* Monogame. Entretanto, por efeito da utilização desse *framework*, os laços e atualizações necessárias para o fluxo do jogo tornaram-se significativamente mais fáceis de serem desenvolvidos.

6 REFERÊNCIAS

Devmedia - Introdução à linguagem C#

PT Wikipedia - Microsoft XNA

Caelum - O que é C# e .NET

Oyyou - Monogame Tutorials

Monogame - Documentation

Rikidot - Monogame Tutorials

Microsoft - C# / Programming Guide

Microsoft - C# / Documentation

Sophia Javeriana - Object Oriented Software Construction - Meyer