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 3
1.1 Introdução ao problema 3
1.2 Introdução à linguagem 3
1.3 Introdução ao *framework* Monogame 3
2 VISÃO GERAL DA LINGUAGEM 4
2.1 Apresentação da linguagem escolhida 4
3 ANÁLISE CRíTICA 5
3.1 Critérios e Propriedades 5
3.1.1 Tabela de Critérios e Propriedades 5
3.1.2 Simplicidade 5
3.1.3 Ortogonalidade 6
3.1.4 Expressividade 7
3.1.5 Adequabilidade e variedade de estruturas de controle 8
3.1.6 Mecanismos de definição de tipos 8
3.1.7 Suporte a abstração de dados e de processos 8
3.1.8 Modelo de tipos 10
3.1.9 Portabilidade 10
3.1.10 Reusabilidade 11
3.1.11 Suporte e Documentação 11
3.1.12 Tamanho de Código 11
3.1.13 Generalidade 11
3.1.14 Eficiência e Custo 13
4 REQUISITOS 14
4.1 Paradigma OO 14
4.1.1 Classes 14
4.1.2 Encapsulamento e Proteção 16
4.1.3 Destrutores 17
4.1.4 Espaços de nomes 18
4.1.5 Herança 19
4.1.6 Polimorfismo por Inclusão 20
4.1.7 Polimorfismo Paramétrico 23
4.1.8 Polimorfismo por Sobrecarga 24
4.1.9 *Delegates* 25
4.2 Paradigma funcional 26
4.2.1 Elementos Imutáveis e Funções Puras 26
4.2.2 Funções *Lambda* 27
4.2.3 *Pattern Matching* 28
4.2.4 Funções de Ordem Superior 28
4.2.5 Funções de Ordem Superior Prontas 28
4.2.6 Funções Como Elementos de Primeira Ordem 29
4.2.7 Recursão 29
4.2.8 *Currying* 29
5 CONCLUSÃO 31
5.1 Analisando os paradigmas 31
5.2 Benefícios e Limitações 31
6 REFERÊNCIAS 33

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);

        }
```

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

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

Listing 3.4 – 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 3.5 – 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 *getters* e textitsetters, 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 *getters* e textitsetters.

```
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;
                                ...

                                /* 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. Além disso, também apresenta classes genéricas já disponíveis, como a classe *List*. No trecho de código a seguir, é apresentada a implementação de uma pilha 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.

4 REQUISITOS

O código encontra-se disponível no GitHub, no repositório: TowerDefenseGame. Na *branch develop*, encontra-se a implementação no paradigma OO e, na *branch functional*, encontra-se a implementação no paradigma funcional. A *branch master* está alinhada à *branch functional*.

Na primeira etapa, foi desenvolvida a implementação no paradigma OO. E, na segunda etapa, a implementação no paradigma funcional.

Nas seções seguintes, estão exemplificados, com trechos de código, os requisitos implementados na implementação utilizando cada um dos paradigmas.

4.1 Paradigma OO

4.1.1 Classes

Para o devido funcionamento do jogo, foram utilizadas 11 classes distintas, sendo uma delas exclusivamente para fácil acesso e modificação de constantes necessárias, e outra caracterizada como a classe principal, que instancia todas as demais e contém o loop principal do jogo. As outras classes se apresentam em níveis de hierarquia distintos, e encapsulam devidamente os atributos, assim como os usam em métodos da lógica do jogo. Seguem dois trechos de código demonstrando a classe *ArrowTower*, e a classe de constantes anteriormente referida.

```
public class ArrowTower : Tower
        {
                ...

                /* Constructor */
        public ArrowTower(Texture2D texture, Texture2D
           bulletTexture, Vector2 position)
            : base(texture, bulletTexture, position)
        {
            this.cost = Constants.ARROW_TOWER_COST;
            this.damage = Constants.ARROW_TOWER_DAMAGE;
```

```
            this.radius = Constants.ARROW_TOWER_RADIUS;
                                                this.cost =
                                                   Constants.ARROW_TOWER_COST;
                                                this.bulletSpeed
                                                   =
                                                   Constants.ARROW_TOWER_BULLE

                                                ...
        }
        }
```

Listing 4.1 – Trecho de código C# retirado da implementação deste trabalho

```
class Constants
        {
                        ...

                        /* ABOUT THE GAME */
                        static public int MAX_ENEMIES = 7;
                        static public int ENEMY_SPAWN_TIME = 5;

                        /* ABOUT THE ENEMIES */
                        static public int ENEMY_START_HEALTH =
                           100;
                        static public float ENEMY_SPEED = 2f;

                        /* ABOUT THE MAP */
                        static public int MAP_TILE_SIZE = 64;

                        /* ABOUT THE PLAYER */
                        static public int PLAYER_START_GOLD =
                           50;
                        static public int PLAYER_START_LIFES =
                           3;

                        /* ABOUT THE ARROW TOWER TOWER */
```

```
                        static public int ARROW_TOWER_DAMAGE =
                            50;
                        static public int ARROW_TOWER_RADIUS =
                            150;
                        static public int ARROW_TOWER_COST = 0;
                        static public int
                            ARROW_TOWER_BULLET_SPEED = 5;

                        ...

        }
```

Listing 4.2 – Trecho de código C# retirado da implementação deste trabalho

4.1.2 Encapsulamento e Proteção

Todas as classes apresentam atributos privados ou protegidos (de acordo com a necessidade de uso através das heranças) e tais atributos apenas podem ser acessados através de métodos públicos.

```
 public class Bullet : Sprite
{
                /* Attributes */
    protected int damage;
    protected int age;
    protected int speed;

                /* Getters */
                public int GetDamage()
                {
                        return damage;
                }

                public int GetAge()
                {
                        return age;
```

```
        }

        public int GetSpeed()
        {
                return speed;
        }

        /* Setters */
        public void SetDamage(int damage)
        {
                this.damage = damage;
        }

        public void SetAge(int age)
        {
                this.age = age;
        }

        public void SetSpeed(int speed)
        {
                this.speed = speed;
        }

        ...
}
```

Listing 4.3 – Trecho de código C# retirado da implementação deste trabalho

4.1.3 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.

4.1.4 Espaços de nomes

No *Tower Defense* desenvolvido estão contidos dois espaços de nomes distintos, sendo um deles para as classes que implementam a lógica e organização das estruturas do jogo como um todo, chamada MonoGame2D, e outra que envolve duas classes usadas para configurações do jogo, *Button* e *Constants*, sob o *namespace* de *Configurations*. Abaixo um exemplo de classe sob o *namespace MonoGame2D* e outro sob o *Configurations*. Ao acessar tais classes de um espaço de nomes diferente, é preciso especificar o correspondente.

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

}
```

Listing 4.4 – Trecho de código C# retirado da implementação deste trabalho

```
namespace Configurations
{
        class Constants
        {
                /* ABOUT THE GAME */
                static public int MAX_ENEMIES = 7;
                static public int ENEMY_SPAWN_TIME = 5;
                ...
        }
```

```
}
```

Listing 4.5 – Trecho de código C# retirado da implementação deste trabalho

4.1.5 Herança

Sobre a relação de herança das classes presentes, existe uma classe chamada *Sprite* que é a mais abstrata, utilizada por todos os elementos que apresentam algum tipo de movimento no jogo, e precisam dos atributos de Posição, Velocidade e Rotação, além do atributo padrão Textura, que aplica a imagem correspondente ao elemento. Essa classe tem dois níveis de herança abaixo de si, sendo eles a classe *Tower*, que controla genericamente as torres do jogo, e como filha desta a classe *ArrowTower*, que é uma especialização de torre com características próprias. Dessa forma, é possível futuramente adaptar o jogo para a criação de diversos tipos de torres mais facilmente, herdando os atributos de *Tower*. Segue abaixo um diagrama de classes ilustrando as hierarquias citadas, e trechos de código indicando os três níveis de herança.

```
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;

        }
```

Listing 4.6 – Trecho de código C# retirado da implementação deste trabalho

```
public class ArrowTower : Tower
        {
                /* Constructor */
        public ArrowTower(Texture2D texture, Texture2D
           bulletTexture, Vector2 position)
```

```
            : base(texture, bulletTexture, position)
        {
            this.cost = Constants.ARROW_TOWER_COST;
            this.damage = Constants.ARROW_TOWER_DAMAGE;
            this.radius = Constants.ARROW_TOWER_RADIUS;
                        this.cost = Constants.ARROW_TOWER_COST;
                        this.bulletSpeed =
                            Constants.ARROW_TOWER_BULLET_SPEED;
        }
                }
```

Listing 4.7 – Trecho de código C# retirado da implementação deste trabalho

4.1.6 Polimorfismo por Inclusão

Esse tipo de polimorfismo é usado exaustivamente, pois está muito presente no próprio *framework* utilizado para desenvolvimento. Grande parte das classes implementadas para o trabalho são herdeiras da classe *Sprite*, que contém métodos como *Update()* e *Draw()*. Contudo, tais classes herdeiras também implementam suas próprias funções *Update()* e *Draw()* como apresentado nos trechos a seguir.

```
namespace MonoGame2D
{
    public class Sprite
    {
                                ...

                public virtual void Update(GameTime gameTime)
        {
            this.center = new Vector2(position.X +
               texture.Width / 2,
                position.Y + texture.Height / 2);
        }

                                ...
```

```
        public virtual void Draw(SpriteBatch spriteBatch, Color
            color)
        {
            spriteBatch.Draw(texture, center, null, color,
                rotation, origin, 1.0f,
                                SpriteEffects.None, 0);
        }
    }
}
```

Listing 4.8 – Trecho de código C# retirado da implementação deste trabalho

```
namespace MonoGame2D
{
    public class Enemy : Sprite
    {

                            ...

                                public override void
                                    Update(GameTime gameTime)
        {
            base.Update(gameTime);

            Vector2 position = GetPosition();
            if (waypoints.Count > 0)
                        {
                                if (DistanceToDestination < 1f)
                                {
                                        position =
                                            waypoints.Peek();
                                        waypoints.Dequeue();
                    SetPosition(position);
                                }
```

```
                    else
                    {
                        Vector2 direction =
                            waypoints.Peek() -
                            position;
                        direction.Normalize();

                        velocity =
                            Vector2.Multiply(direction,
                            speed);

                        position += velocity;
                SetPosition(position);
                    }
                }
                else
                {
                    outOfScreen = true;
                    alive = false;
                }

                if (currentHealth <= 0)
                {
                    alive = false;
                }

        }

        public override void Draw(SpriteBatch spriteBatch)
        {
            if (alive)
            {
                float healthPercentage = (float)currentHealth /
                    (float)startHealth;

                base.Draw(spriteBatch, Color.White);
```

```
            }
        }
        }
}
```

Listing 4.9 – Trecho de código C# retirado da implementação deste trabalho

4.1.7 Polimorfismo Paramétrico

A linguagem disponibiliza uma biblioteca de classes genéricas já implementadas, como a classe *List*, a qual utilizamos de forma exaustiva na implementação OO. Além disso, através dessa biblioteca, também é possível de definir novos tipos genéricos, em que foi definido o tipo GenericList.

```
public class GenericList<T>
{
        T[] obj = new T[Constants.ENEMIES_TEXTURES];
        int count = 0;

        // Adding items mechanism into generic type
        public void Add(T item)
        {
                // Checking length
                if (count + 1 <
                   Constants.ENEMIES_TEXTURES)
                {
                        obj[count] = item;
                }
                count++;
        }

        public int getSize()
        {
                return count;
        }
```

```
        public T getOnIndex(int index)
        {
                return obj[index];
        }

        public void setOnIndex(int index, T item)
        {
                obj[index] = item;
        }

    }
```

Listing 4.10 – Trecho de código C# retirado da implementação deste trabalho

```
    public class Game1 : Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Random random = new Random();
    Map map = new Map();

    List<Enemy> enemies = new List<Enemy>();

                        GenericList<Texture2D>
                           enemyTextures = new
                           GenericList<Texture2D>();
                        ...
        }
```

Listing 4.11 – Trecho de código C# retirado da implementação deste trabalho

4.1.8 Polimorfismo por Sobrecarga

Utilizaram-se construtores alternativos durante a codificação do programa, onde um dos construtores recebe parâmetros e seta atributos. Além de polimorfismo por inclu-

são, citado anteriormente, o método *Draw()* da classe *Sprite* também implementa sofre polimorfismo por sobrecarga, como apresentado nos exemplos abaixo. Dessa forma, é possível escolher qual dos métodos *Draw()* será chamado através dos parâmetros passados.

```
namespace MonoGame2D
{
    public class Sprite
    {
                        ...

                public virtual void Draw(SpriteBatch
                    spriteBatch)
        {
            spriteBatch.Draw(texture, center, null, Color.White,
                rotation, origin, 1.0f, SpriteEffects.None, 0);
        }

        public virtual void Draw(SpriteBatch spriteBatch, Color
            color)
        {
            spriteBatch.Draw(texture, center, null, color,
                rotation, origin, 1.0f, SpriteEffects.None, 0);
        }
    }
}
```

Listing 4.12 – Trecho de código C# retirado da implementação deste trabalho

4.1.9 *Delegates*

Seu uso foi combinado com a solução encontrada para implementação de expressões *Lambda*, e o trecho de código correspondente se encontra na seção dos requisitos funcionais *Lambda*.

4.2 Paradigma funcional

4.2.1 Elementos Imutáveis e Funções Puras

As listas principais do jogo são elementos imutáveis. Isto é, lista de torres criadas pelo jogadores, lista de inimigos vivos, lista de texturas de inimigos e listas de balas em curso lançadas por cada uma das torres. Para isso, foi utilizada a classe *ImmutableList* oferecida por uma das bibliotecas da linguagem. Dessa forma, sempre que uma lista é modificada, deve-se criar uma nova lista a partir da anterior.

```
ImmutableList<Enemy> enemies = ImmutableList.Create<Enemy>();

ImmutableList<Texture2D> enemyTextures =
   ImmutableList.Create<Texture2D>();
```

Listing 4.13 – Trecho de código C# retirado da implementação deste trabalho

```
public void ClearTowers()
{
                towers = towers.Clear();
}
```

Listing 4.14 – Trecho de código C# retirado da implementação deste trabalho

Em relação às funções puras, definiu-se a função *multiplyingByTileSize*, responsável por multiplicar um vetor por uma constante. Essa função é independente de I/O, estado do programa ou qualquer efeito colateral, ela sempre realizará o mesmo cálculo de acordo com sua entrada.

```
public Vector2 multiplyingByTileSize(Vector2 vector)
{
                Vector2 multiplied = vector *
                   Constants.MAP_TILE_SIZE;

                return multiplied;
}
```

Listing 4.15 – Trecho de código C# retirado da implementação deste trabalho

4.2.2 Funções *Lambda*

Foram usadas funções *Lambda* como um método de facilitar mensagens passadas, e para isso foram combinados *delegates* com seus devidos parâmetros compatíveis. A seguir estão os trechos de código contendo a declaração do *delegate* em questão, e seu uso combinado com as funções *lambda* de instrução, que recebem as mensagens diretamente da classe *Constants*.

```
public class Toolbar
    {
                                    ...

        // A delegate to be used with the Lambda Functions
        delegate void OnPassValue(string s);

                                    ...
        OnPassValue passValueLives = (x) => { textLives =
            string.Format(x, player.GetLives()); };
        OnPassValue passValueGold = (x) => { textGold =
            string.Format(x, player.GetGold()); };

        passValueLives(Constants.toolBarMessageLives); //Lambda
            expression with delegate
        passValueGold(Constants.toolBarMessageGold);   //Lambda
            expression with delegate
        spriteBatch.DrawString(font, textLives,
            textPositionLives, Color.SaddleBrown);
        spriteBatch.DrawString(font, textGold,
            textPositionGold, Color.SaddleBrown);
    }
```

Listing 4.16 – Trecho de código C# retirado da implementação deste trabalho

4.2.3 *Pattern Matching*

A correspondência de tipos foi usada justamente para realizar a checagem de um tipo de dados passado como parâmetro para uma função, através do operando is presente na linguagem C#. É realizada a comparação do objeto passado, e se ele passa pela verificação, é feito um cast para o objeto desejado, para então utilizar um método daquele objeto de classe. Caso a verificação não tenha um valor de sucesso verdadeiro, o retorno da função é nulo. Segue o exemplo do uso de*Pattern Matching*.

```
public static Enemy setPath(object functionValue, Map map)
        {
            Enemy enemy = null;
            if (functionValue is Enemy)
            {
                enemy = (Enemy)functionValue;
                enemy.SetWaypoints(map.GetWaypoints());
            }
            return enemy;
        }
```

Listing 4.17 – Trecho de código C# retirado da implementação deste trabalho

4.2.4 Funções de Ordem Superior

Não implementado.

4.2.5 Funções de Ordem Superior Prontas

Utilizou-se a função já implementada *Where*, que recebe uma função como parâmetro e retorna um objeto contendo os elementos da lista de entrada que satisfazem a condição de tal função. Transforma-se essa objeto em uma lista novamente utilizando o método *toList()*.

```
List<Tower> filteredList = towers.Where(tower =>
```

```
    tower.GetPosition() == new Vector2(tileX, tileY)).ToList();
```

Listing 4.18 – Trecho de código C# retirado da implementação deste trabalho

4.2.6 Funções Como Elementos de Primeira Ordem

Uma função com elementos de primeira ordem simplesmente representa a possibilidade de associar uma função à uma variável, e passar essa última como parâmetro, invocando-a. Nesse caso, ela está presente no código tanto no uso das funções *Lambda* da classe *Toolbar*, quanto no uso de *currying*, que utiliza funções anônimas, presente na classe principal *Game1*. Para evitar redundâncias, os códigos exemplificando o requisito de funções com elementos de primeira ordem está presente na seção Funções *Lambda* e Funções que utilizam *currying*.

4.2.7 Recursão

Não implementado.

4.2.8 *Currying*

Foi usado currying para realizar operações matemáticas de subtração consecutivas dentro do código, onde o parâmetro passado representa o valor que receberá a modificação, e o retorno é o valor inicial com as devidas operações já realizadas sobre ele.

```
internal static int DecrementCurrying(int i)
        {
            // int -> (int -> int)
            Func<int, Func<int, int>> higherOrderSubtract = a
               => new Func<int, int>(b => b - a);
            Func<int, int> dec = higherOrderSubtract(1); //
               Equivalent to: b => b - a.
            int curriedResult = dec(i);

            return curriedResult;
```

```
        }
```

Listing 4.19 – Trecho de código C# retirado da implementação deste trabalho

5 CONCLUSÃO

5.1 Analisando os paradigmas

Durante o desenvolvimento do trabalho utilizando OO, a inserção dos elementos relacionados a esse paradigma ocorreu naturalmente. Isto é, utilizamos os recursos da linguagem de forma intuitiva e não foram necessários muitos ajustes para adequar o programa aos requisitos solicitados.

Já no desenvolvimento funcional, muitos dos requisitos solicitados para esse paradigma foram forçados. Ou seja, houve a necessidade de explorarmos recursos menos utilizados da linguagem a fim de adequar o programa aos requisitos. O raciocínio funcional não era intuitivo para o desenvolvimento.

Nesse sentido, acreditamos que a facilidade em OO e dificuldade em funcional ocorreram, principalmente, porque C# é naturalmente uma linguagem Orientada a Objetos, mas também pois o paradigma funcional não é o paradigma ao qual as desenvolvedoras costumam programar. Logo, o raciocínio durante o desenvolvimento dificilmente nos levava à utilização dos recursos do paradigma funcional.

Apesar disso, concluímos que os dois paradigmas oferecem recursos úteis. A organização do código em classes, herança e polimorfismo, por exemplo, trouxeram facilidade no desenvolvimento do trabalho. Já a possibilidade de usar funções de ordem superior, como filter, map e outras, pode trazer flexibilidade ao código.

5.2 Benefícios e Limitações

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 familizariadas às respectivas sintaxes.

Além disso, C# é uma linguagem com paradigma OO com diversos artifícios úteis já disponíveis, como tipos genéricos. Portanto, durante o desenvolvimento utilizando o paradigma OO, a estruturação do código utilizando os recursos desse paradigma aconteceu naturalmente à medida em que o código era incrementado. Em contrapartida, o desenvolvimento funcional não foi intuitivo, e os diversos recursos do paradigma foram inseridos forçadamente no código.

Por fim, acreditamos que nossos maiores desafios consistiram-se no desenvolvimento funcional e 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

Stackoverflow - Any Difference between first class function and high-order function

Weblogs - Understanding Csharp Features

Sitepoint - Functional programming pure functions