KAUNO TECHNOLOGIJOS UNIVERSITETAS

INFORMATIKOS FAKULTETAS

T120B516 Objektinis programų projektavimas

Projektinio darbo ataskaita

Ernestas Seminogovas IFF-5/2

Matas Skrupskas IFF-5/3

Mindaugas Nakrošis IFF-5/3

Lukas Semaška IFF-5/3

KAUNAS 2018

Turinys

Įvadas

Darbo tikslas

Kuriant Space-Invaders (Pav. 1) tipo žaidimą išmokti taikyti projektavimo šablonus (angl. „design patterns“) ir susipažinti su jų naudojimo ypatumais. „Space Invaders“ žaidime vartotojas valdys erdvėlaivį, kuris turės įveikti skirtingo sudėtingumo priešus šaudydamas į juos. Skirtingų priešų įveikimas apdovanojamas skirtingu kiekiu taškų.

Pav. 1 Space-Invaders žaidimo pavyzdys

Žaidime naudojami 4 pagrindiniai objektai: žaidimas, žaidimo lenta, priešai ir žaidėjo valdomas erdvėlaivis.

Žaidimas – tai pagrindinis objektas. Jį sudaro žaidimo lenta ir šiuo metu žaidėjo surinkti taškai.

Žaidimo lenta – lentą sudaro priešai ir žaidėjo valdomas erdvėlaivis. Lenta priklauso žaidimui.

Priešas – priešą sudarantys atributai: gyvybės ir pozicija. Taip pat jis turi metodą patikrinti ar į jį nepataikė kulka.

Žaidėjo erdvėlaivis – tai objektas turintis gyvybių, pozicijos, iššautų kulkų ir bazinės žalos atributus.

Naudojamos technologijos:

1. .Net Core 2.0. Ją pasirinkome, nes tai naujausia .net karkaso versija leidžianti neatsilikti nuo .net naujovių ir paobulinimų. Be to patobulėjęs greitis palyginus su pirmtakėmis versijomis.
2. MVC (Model-View-Controller) architektūra
3. Entity Framework

Pirmas laboratorinis darbas

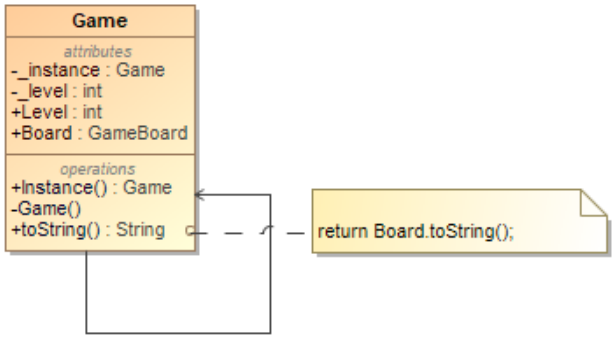
Programavimo šablonai

Singleton

Panaudojimo tikslas:

Panaudojome *Singleton* programavimo šabloną, kad užtikrintume, kad vienu metu žaidėjas žaidžia tik vieną žaidimą.

UML diagrama:



Pav. 2. Singleton programavimo šablonas

Kodas:

```
public class Game
{
    public GameBoard Board { get; set; }
    public int Level { get; set; }

    private int _level { get; set; }
    private static Game _instance;

    public static Game Instance => _instance ?? (_instance = new Game()); // Singleton

    private Game() { }

    public override string ToString()
    {
        return Board.ToString();
    }
}
```

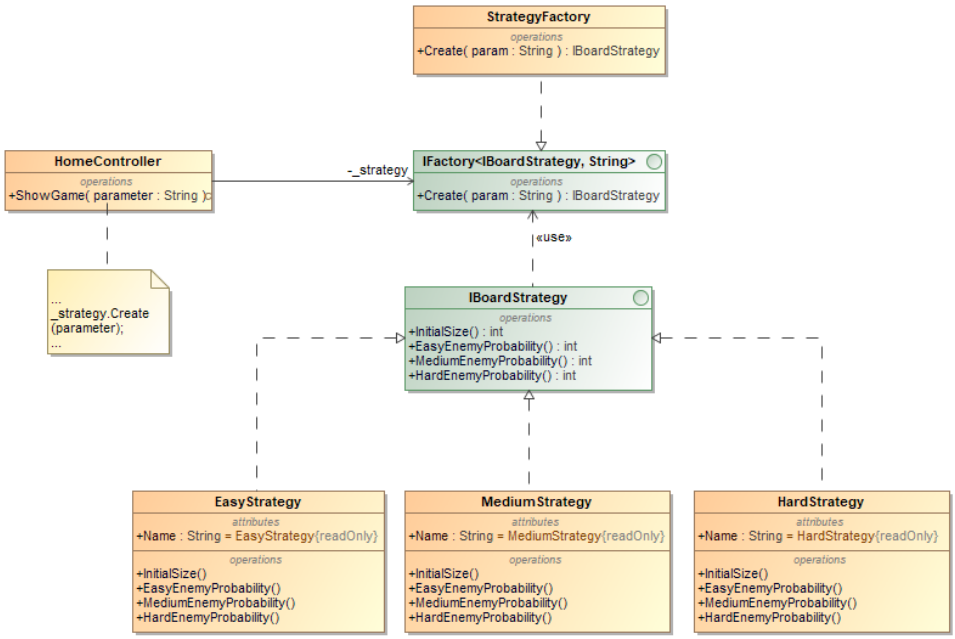
Factory

Panaudojimo tikslas:

Kadangi turime keletą skirtingų žaidimo lygių (*EasyStrategy*, *MediumStrategy*, *HardStrategy*), tai norint juos sukonstruoti, kad būtų galima naudoti žaidimo lentos kūrimui, mes panaudojame „Factory“ programavimo šabloną, kuris pagal vartotojo įvestą pasirinkimą („*Easy*“, „*Medium*“, „*Hard*“) sukurs

strategiją. Šį programavimo šabloną galime panaudoti su strategijomis, kadangi turime abstraktų strategijos tipą.

UML diagrama:



Pav. 3. Factory programavimo šablonas

Kodas:

```
public class StrategyFactory : IFactory<IBoardStrategy, string>
{
   public IBoardStrategy Create(string param)
   {
      switch (param)
      {
          case HardStrategy.Name:
              return new HardStrategy();
          case MediumStrategy.Name:
              return new MediumStrategy();
          default:
              return new EasyStrategy();
       }
   }
}
```

```
public interface IFactory<out T, in TParam> where T : class where TParam: class
{
   T Create(TParam param);
}
```

```
public class HomeController : Controller, IHomeController
{
    private readonly IFactory<IBoardStrategy, string> _strategyFactory;
```

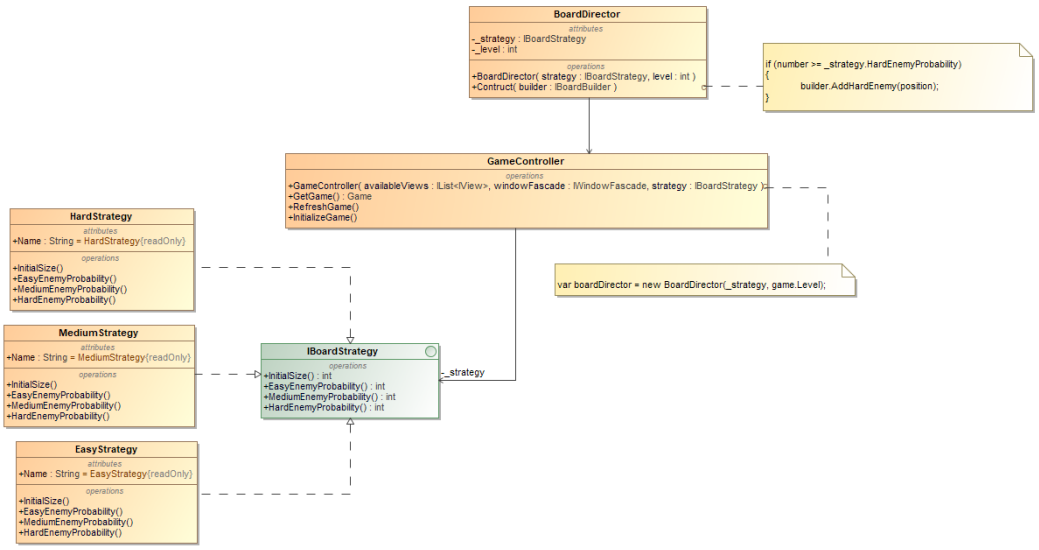
```
    public HomeController(IList<IView> views, IWindowFascade windowFascade,
IPlayerRepository playerRepository) : base(views, windowFascade)
    {
            _playerRepository = playerRepository;
            _strategyFactory = new StrategyFactory();
    }

        public void ShowGame(string parameter)
        {
            var strategy = _strategyFactory.Create(parameter);
            var controller = ChangeController(Contracts.Contracts.GameController,
strategy);
            WindowFacade.ChangeController(controller);
        }
    }
```

Strategy

Panaudojimo tikslas:

Kadangi vartotojas gali norėti pasirinkti žaidimo sudėtingumo lygį, kuris nurodys, kiek ir kokio sudėtingumo priešų bus žaidime, todėl įgyvendinome *„Strategy“* programavimo šabloną, kuris padėjo nenaudojant *„concrete class“* objektų, o vietoj to naudojantis abstrakčiu tipu *„IboardStrategy“* įgyvendinti skirtingus žaidimo sudėtingumo lygius.

UML diagrama:



Pav. 4 Strategy programavimo šablonas

Kodas:

```
public interface IBoardStrategy
{
    int InitialSize { get; }
    int EasyEnemyProbability { get; }
    int MediumEnemyProbability { get; }
```

```
    int HardEnemyProbability { get; }
}
```

```
public class EasyStrategy : IBoardStrategy
{
    public const string Name = "EasyStrategy";
    public int InitialSize => 5;
    public int EasyEnemyProbability => 0;
    public int MediumEnemyProbability => 50;
    public int HardEnemyProbability => 80;
}
```

```
public class MediumStrategy : IBoardStrategy
{
     public const string Name = "MediumStrategy";
     public int InitialSize => 7;
     public int EasyEnemyProbability => 0;
     public int MediumEnemyProbability => 30;
     public int HardEnemyProbability => 70;
}
```

```
public class HardStrategy : IBoardStrategy
{
    public const string Name = "HardStrategy";
    public int InitialSize => 10;
    public int EasyEnemyProbability => 0;
    public int MediumEnemyProbability => 30;
    public int HardEnemyProbability => 50;
}
```

```
public class BoardDirector
{
    private IBoardStrategy _strategy;
    private int _level;

    public BoardDirector(IBoardStrategy strategy, int level)
    {
       _strategy = strategy;
       _level = level;
    }

    public void Construct(IBoardBuilder builder)
    {
        var random = new Random();
        var count = 0;
        for (var column = 0; column < Contracts.GameSizeHeight; column++)
        {
            for (var row = 0; row < Contracts.GameSizeWidth; row=row+10)
            {
               var number = random.Next(0, 100);

                var position = new Block() {
                    From = new Position(row, column),
                    To = new Position(row + 8, column)
                };

                if (number >= _strategy.HardEnemyProbability)
```

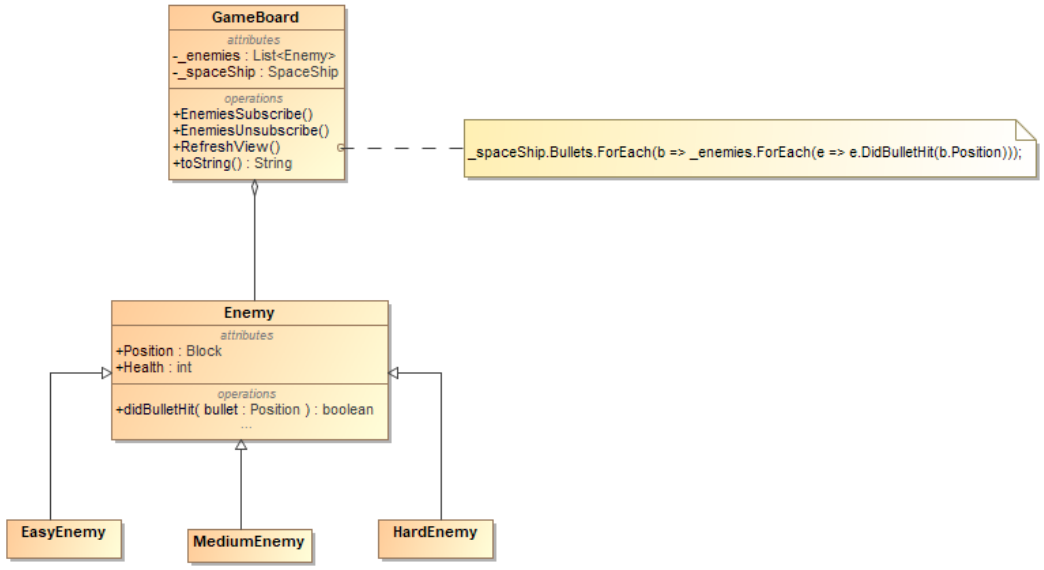
```
                {
                    builder.AddHardEnemy(position);
                }
                else if (number >= _strategy.MediumEnemyProbability)
                {
                    builder.AddMediumEnemy(position);
                }
                else
                {
                    builder.AddEasyEnemy(position);
                }

                count++;

                if (count > _strategy.InitialSize + _level)
                {
                    break;
                }
            }
            if (count > _strategy.InitialSize + _level)
            {
                break;
            }
        }
        builder.AddSpaceShip(4);
    }
}
```

Observer

Panaudojimo tikslas:

Kadangi žaidimo lentoje gali būti daug priešų, mums reikia sekti kada žaidėjo iššauta kulka kliudo priešų erdvėlaivį, todėl vietoj to, kad patys šalintume priešus, mes pranešam jiems, kad išauta kulka ir jie galėtų apdoroti šią informaciją.

UML diagrama:



Pav. 5 Observer programavimo šablonas

Kodas:

```
public class GameBoard // Subject
{
    private SpaceShip _spaceShip;

    private readonly List<Enemy> _enemies; // enemies subscribers

    public int EnemiesCount => _enemies.Count;

    public GameBoard(SpaceShip spaceShip)
    {
        _spaceShip = spaceShip;

        _enemies = new List<Enemy>();
    }

     // Adding enemies to the board
     public void EnemiesSubscribe(Enemy enemy)
     {
         _enemies.Add(enemy);
     }

     // Removing dead enemies from the board
     public void EnemieUnsubscribe(Enemy enemy)
     {
         _enemies.Remove(enemy);
     }

     public void RefreshView()
     {
         var enemiesToDelete = new List<Enemy>();

// Notify all subscribed enemies that shot was fired to update their health if they are
// hit
```

```
          _spaceShip.Bullets.ForEach(b =>
             _enemies.ForEach(e =>
             {
                 e.DidBulletHit(b.Position);
                 if (e.Health == 0)
                 {
                     enemiesToDelete.Add(e);
                 }
             }));

          enemiesToDelete.ForEach(EnemieUnsubscribe);
      }
```

```
public abstract class Enemy
{
    public Block Position { get; set; }
    public int Health { get; set; }

    public bool DidBulletHit(Position bullet) // Update
    {
        if (bullet.X < Position.From.X || bullet.X > Position.To.X) return false;
        if (bullet.Y < Position.From.Y || bullet.Y > Position.To.Y) return false;
        Health--;
        return true;
    }

    public abstract Enemy Clone();
}
```

```
public class EasyEnemy : Enemy
{
    public EasyEnemy()
    {
       Health = 1;
    }

    public override Enemy Clone()
    {
        return this.MemberwiseClone() as Enemy;
    }
}
```

```
public class MediumEnemy : Enemy
{
   public MediumEnemy()
   {
       Health = 3;
   }

   public override Enemy Clone()
   {
       return this.MemberwiseClone() as Enemy;
   }
}
```

```
public class HardEnemy : Enemy
{
```

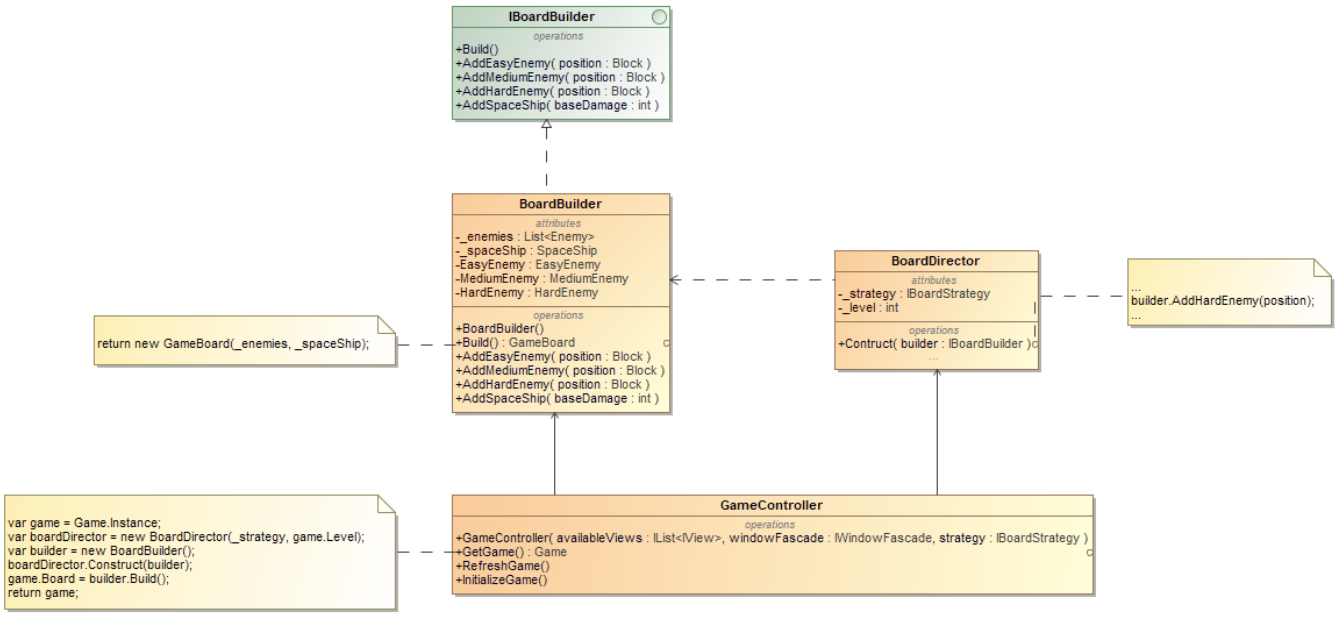
```
    public HardEnemy()
    {
        Health = 2;
    }

    public override Enemy Clone()
    {
        return this.MemberwiseClone() as Enemy;
    }
}
```

Builder

Panaudojimo tikslas:

Kurdami žaidimo lentą pastebėjome, kad norint ją sukurti reikia panaudoti daug skirtingų objektų: *SpaceShip*, *EasyEnemy*, *MediumEnemy*, *HardEnemy*, todėl nusprendėme, kad būtų gerai panaudoti builder programavimo šabloną, kad perkeltume šią logiką į vieną vietą.

UML diagrama:



Pav. 6 Builder programavimo šablonas

Kodas:

```
public interface IBoardBuilder
{
   GameBoard Build();
   void AddEasyEnemy(Block position);
   void AddMediumEnemy(Block position);
   void AddHardEnemy(Block position);
   void AddSpaceShip(int baseDamage);
```

```
}
```

```
public class BoardBuilder : IBoardBuilder
{
    private List<Enemy> _enemies = new List<Enemy>();
    private SpaceShip _spaceShip = new SpaceShip();

    private EasyEnemy EasyEnemy { get; }
    private MediumEnemy MediumEnemy { get; }
    private HardEnemy HardEnemy { get; }

    public BoardBuilder()
    {
        EasyEnemy = new EasyEnemy();
        MediumEnemy = new MediumEnemy();
        HardEnemy = new HardEnemy();
    }

    public GameBoard Build()
    {
        var board = new GameBoard(_spaceShip);

        foreach (var enemy in _enemies)
        {
            board.EnemiesSubscribe(enemy);
        }

        return board;
    }

    public void AddEasyEnemy(Block position)
    {
        var easyEnemy = EasyEnemy.Clone();
        easyEnemy.Position = position;
        _enemies.Add(easyEnemy);
    }

    public void AddMediumEnemy(Block position)
    {
        var mediumEnemy = MediumEnemy.Clone();
        mediumEnemy.Position = position;
        _enemies.Add(mediumEnemy);
    }

    public void AddHardEnemy(Block position)
    {
        var hardEnemy = HardEnemy.Clone();
        hardEnemy.Position = position;
        _enemies.Add(hardEnemy);
    }

    public void AddSpaceShip(int baseDamage)
    {
        _spaceShip = new SpaceShip()
        {
            Position = new Block()
            {
```

```
                From = new Position(40, Contracts.GameSizeHeight),
                To = new Position(60, Contracts.GameSizeHeight)
            }
        };
    }
}
```

```
public class BoardDirector
{
    private IBoardStrategy _strategy;
    private int _level;

    public BoardDirector(IBoardStrategy strategy, int level)
    {
       _strategy = strategy;
       _level = level;
    }

    public void Construct(IBoardBuilder builder)
    {
        var random = new Random();
        var count = 0;
        for (var column = 0; column < Contracts.GameSizeHeight; column++)
        {
            for (var row = 0; row < Contracts.GameSizeWidth; row=row+10)
            {
                    var number = random.Next(0, 100);

                    var position = new Block() {
                        From = new Position(row, column),
                        To = new Position(row + 8, column)
                    };

                    if (number >= _strategy.HardEnemyProbability)
                    {
                        builder.AddHardEnemy(position);
                    }
                    else if (number >= _strategy.MediumEnemyProbability)
                    {
                        builder.AddMediumEnemy(position);
                    }
                    else
                    {
                        builder.AddEasyEnemy(position);
                    }
                    count++;

                    if (count > _strategy.InitialSize + _level)
                    {
                        break;
                    }
                }
                if (count > _strategy.InitialSize + _level)
                {
                    break;
                }
            }
```

```
        builder.AddSpaceShip(4);
    }
}
```

```
public class GameController : Controller, IGameController
{
    private readonly IBoardStrategy _strategy;

    public GameController(IList<IView> availableViews, IWindowFascade windowFascade,
IBoardStrategy strategy) : base(availableViews, windowFascade)
    {
        _strategy = strategy;
    }

    private Game GetGame()
    {
        var game = Game.Instance;
        var boardDirector = new BoardDirector(_strategy, game.Level);
        var builder = new BoardBuilder();
        boardDirector.Construct(builder);
        game.Board = builder.Build();
        return game;
    }

    public void RefreshGame()
    {
        WindowFacade.View.InsertData(GetGame());
    }

    public void InitializeGame()
    {
        ChangeView(Contracts.Contracts.GameView, GetGame());
        WindowFacade.View.AddController(this);
    }
}
```