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
Simonas Sankauskas 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. 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

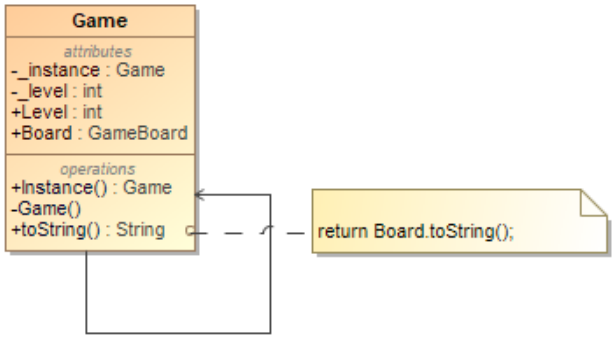
Naudojamos technologijos:

1. .Net Core 2.0
2. MVC (Model-View-Controller) architektūra
3. Entity Framework

Pirmas laboratorinis darbas (1 dalis)

Singleton

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



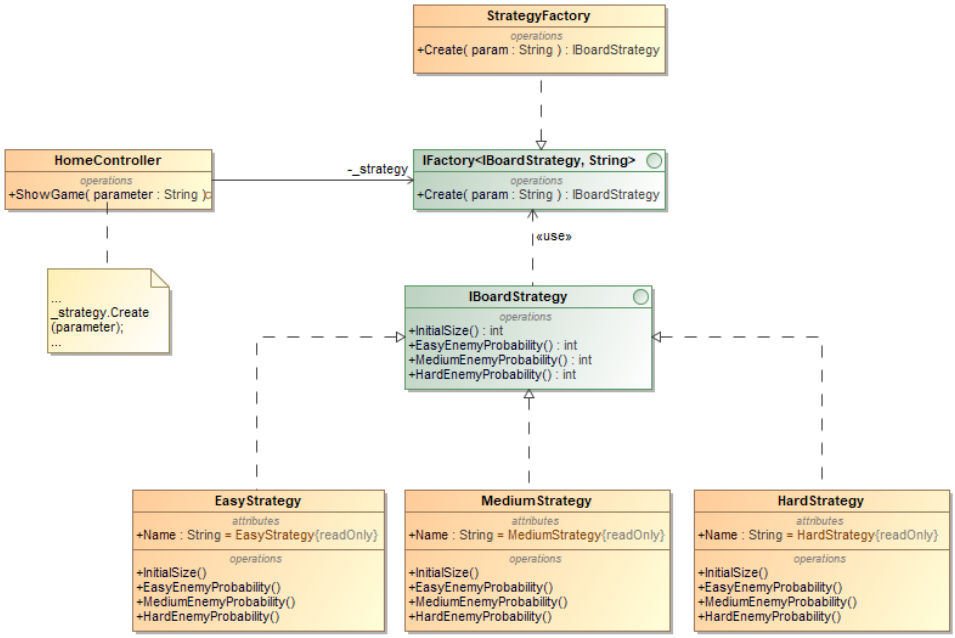
Pav. 2. Singleton šablono įgyvendinimo UML diagrama.

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

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.



Pav. 3. Factory šablono įgyvendinimo UML diagrama.

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();
       }
   }
}
```

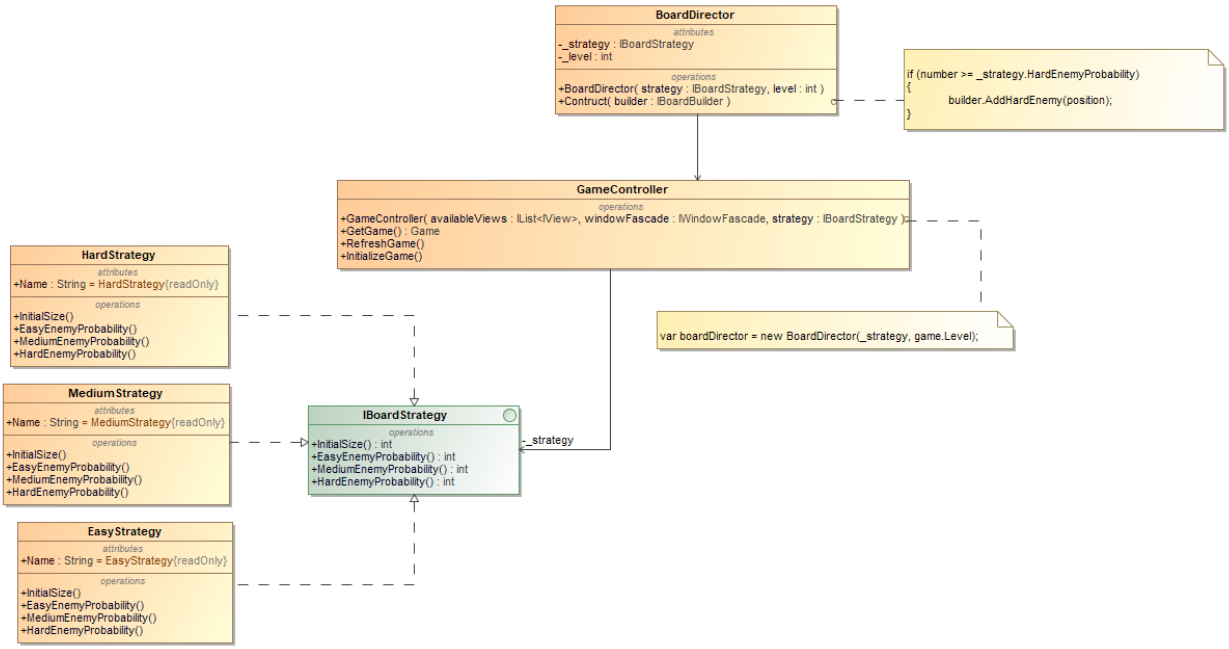
```
public interface IFactory<out T, in TParam> where T : class where TParam: class
{
   T Create(TParam param);
}
```

```
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

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.



Pav. 4 Strategy šablono įgyvendinimo UML diagrama.

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;
}
```

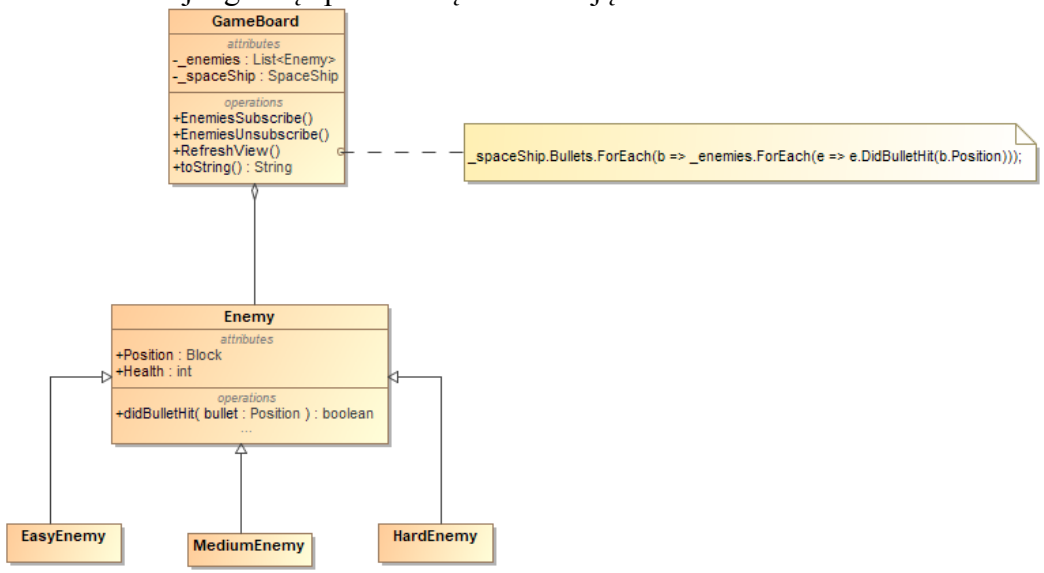
```
public class HardStrategy : IBoardStrategy
{
    public const string Name = "HardStrategy";
    public int InitialSize => 10;
    public int EasyEnemyProbability => 0;
    public int MediumEnemyProbability => 30;
    public int HardEnemyProbability => 50;
}
```

```
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

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.



Pav. 5 Observer šablono įgyvendinimo UML diagrama.

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;
    }
}
```

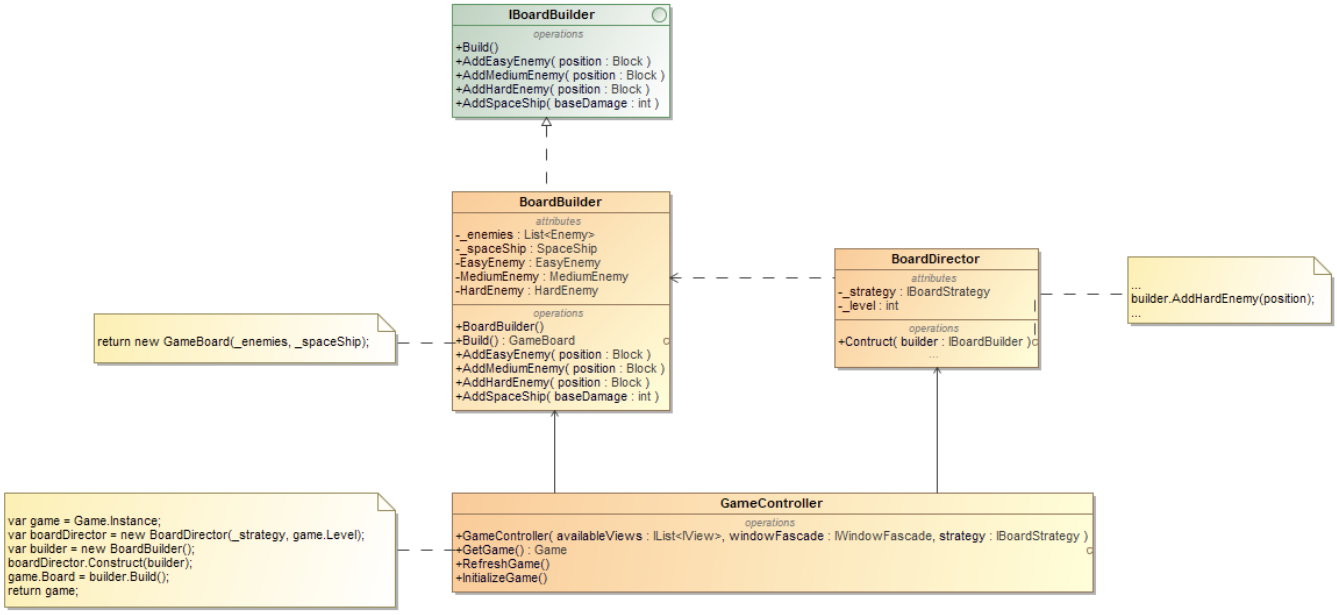
```
public class MediumEnemy : Enemy
{
   public MediumEnemy()
   {
       Health = 3;
   }

   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

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



Pav. 6 Builder šablono įgyvendinimo UML diagrama.

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)
            }
        };
```

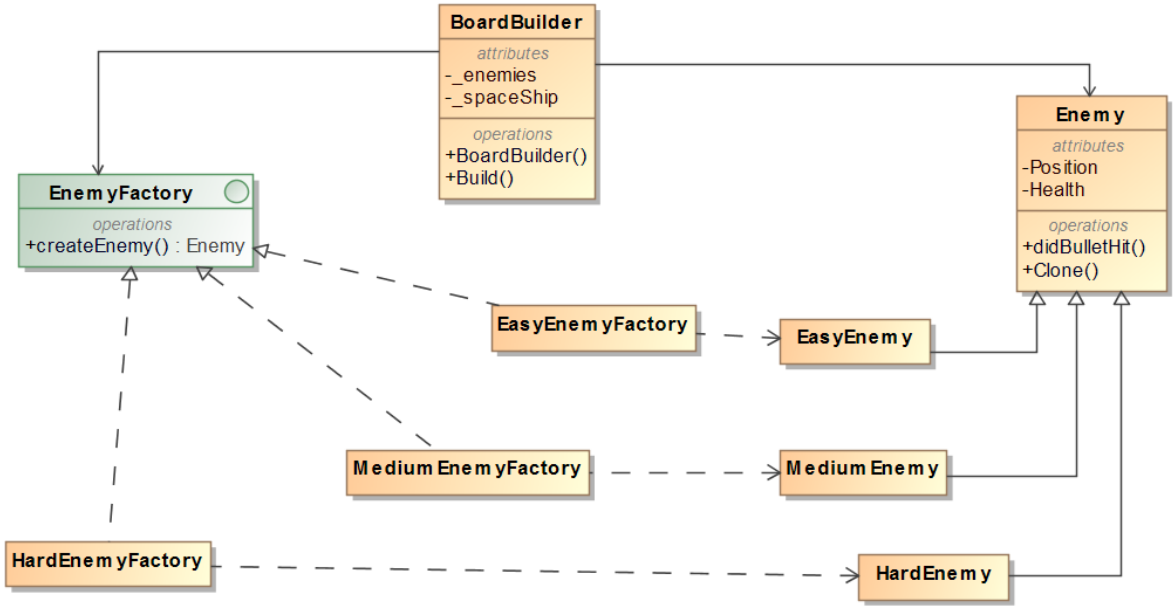
```
    }
}
```

```
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);
    }
}
```

Abstract Factory

Kurdami žadidimo priešus nusprendėme naudoti *Abstract Factory* programavimo šabloną, kad galėtume sukurti skirtingo lygio priešus(*Easy*, *Medium*, *Hard*).



Pav 7 Abstract Factory šablono įgyvendinimo UML diagrama.

```
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 abstract void Accept(EnemyVisitorBase visitor);
    }
```

```
public interface EnemyFactory
    {
        Enemy createEnemy();
    }
```

```
 public class EasyEnemyFactory : EnemyFactory
    {
        public Enemy createEnemy()
        {
```

```
            return new EasyEnemy();
        }
    }
```

```
 public class MediumEnemyFactory : EnemyFactory
    {
        public Enemy createEnemy()
        {
            return new MediumEnemy();
        }
    }
```

```
public class HardEnemyFactory : EnemyFactory
    {
        public Enemy createEnemy()
        {
            return new HardEnemy();
        }
    }
```

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

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

        public override void Accept(EnemyVisitorBase visitor)
        {
            visitor.AddScore(this);
        }
    }
```

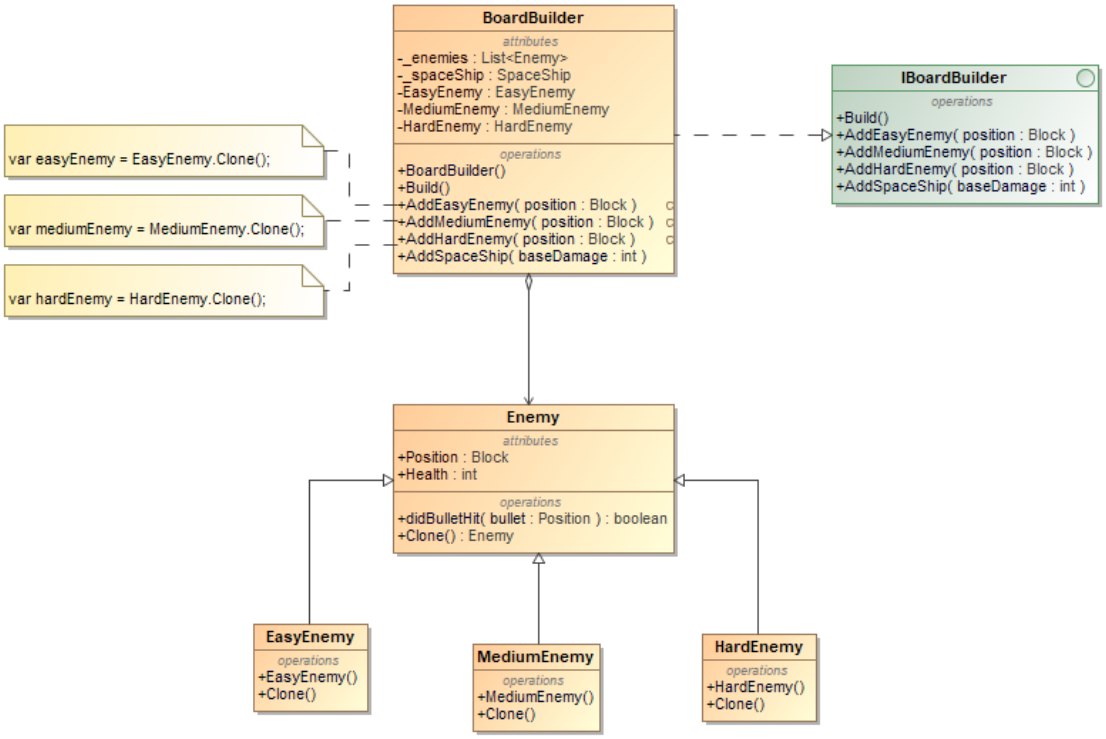
```
public class MediumEnemy : Enemy
    {
        public MediumEnemy()
        {
            Health = 2;
        }

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

        public override void Accept(EnemyVisitorBase visitor)
        {
            visitor.AddScore(this);
        }
    }
```

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

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

        public override void Accept(EnemyVisitorBase visitor)
        {
            visitor.AddScore(this);
        }
    }
```

Pirmas laboratorinis darbas (2 dalis)

Prototype

Kadangi žaidimo lenta gali turėti daug priešų erdvėlaivių ir jie gali būti skirtingų tipų, tai nusprendėme, kad bus patogiau naudoti *Prototype* programavimo šabloną negu kurti naujus skirtingų tipų objektus.



Pav. 8 Prototype šablono įgyvendinimo UML diagrama.

Kodas:

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

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

    public BoardBuilder()
    {
        EasyEnemy = new EasyEnemyFactory().createEnemy();
        MediumEnemy = new MediumEnemyFactory().createEnemy();
        HardEnemy = new HardEnemyFactory ().createEnemy();
     }

     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 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;
   }
```

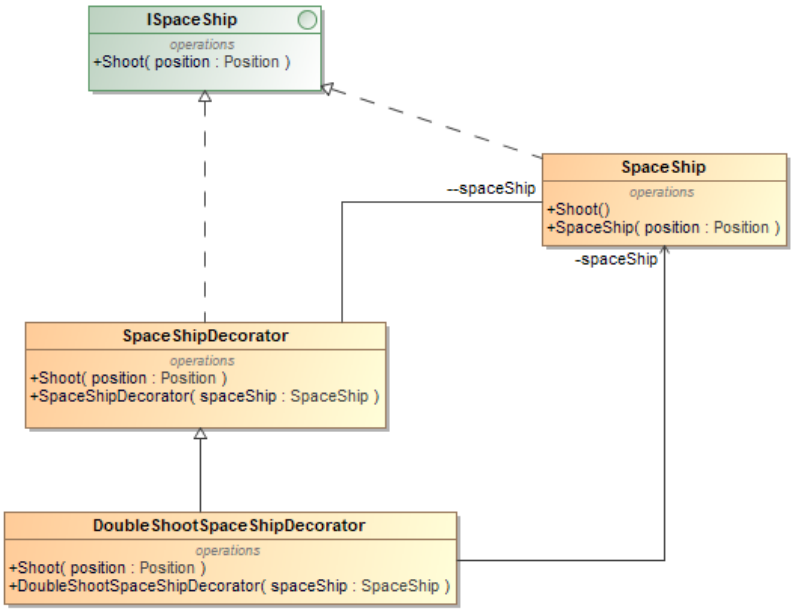
```
}
```

```
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;
    }
}
```

Decorator

Pasirinkom naudoti *Decorator* programavimo šabloną, nes žaidėjas pabaigęs lygį, gali rinktis vieną iš kelių laivo patobulinimų. Vietoj to kad naudoti *switch case* ar kažką panašaus priklausomai nuo pasirinkto patobulinimo, panaudojom *Decorator* programavimo šabloną, kad jau prie esančio laivo patobulinimai būtų pridedami be didesnių sunkumų.



Pav. 9 Decorator šablono įgyvendinimo UML diagrama..

Kodas:

```
public interface ISpaceShip
{
    List<Bullet> Bullets { get; set; }

    Block Position { get; set; }

    int BaseDamage { get; }

    void Shoot(Position position);
}
```

```
public class SpaceShip : ISpaceShip
{
   public List<Bullet> Bullets { get; set; }

    public Block Position { get; set; }

    public int BaseDamage { get; private set; }

    public SpaceShip()
    {
       Bullets = new List<Bullet>();
       BaseDamage = 1;
    }

    public void Shoot(Position position)
```

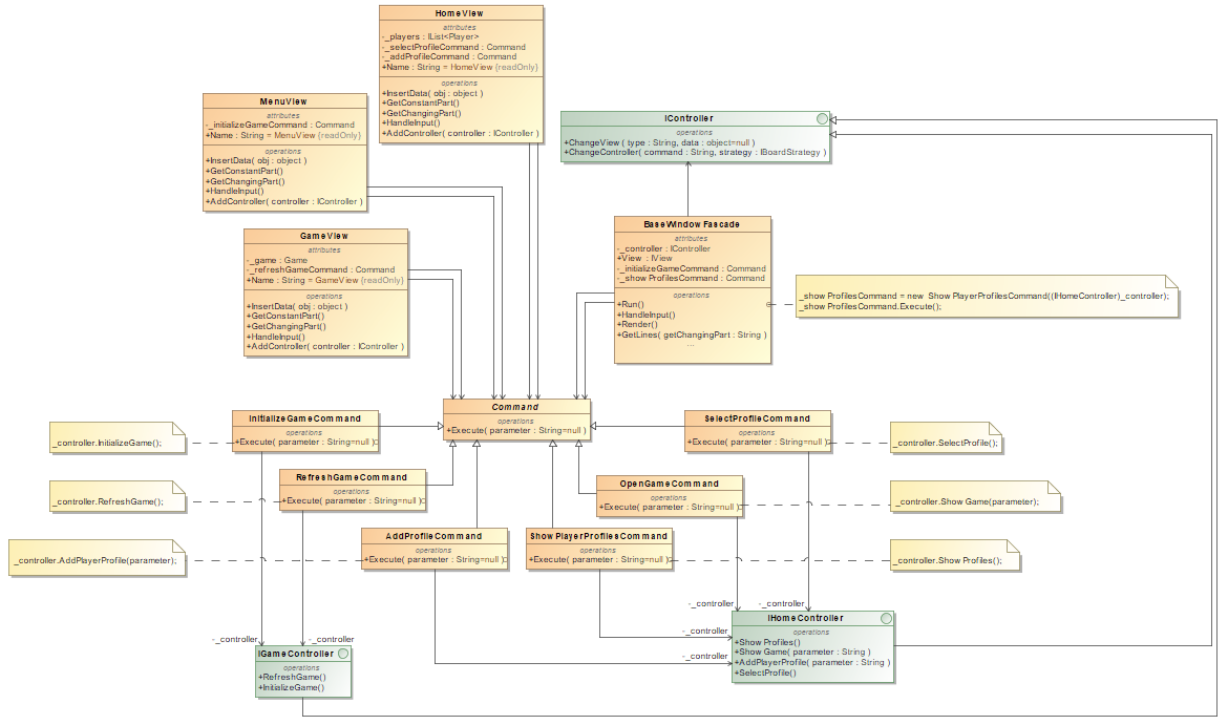
```
    {
        Bullets.Add(new Bullet(position));
    }
}
```

```
public abstract class SpaceShipDecorator : ISpaceShip
{
    private readonly SpaceShip DecoratedSpaceShip;

    public List<Bullet> Bullets { get; set; }

    public Block Position { get; set; }

    public int BaseDamage { get; }

    public SpaceShipDecorator(SpaceShip spaceShip)
    {
        Bullets = spaceShip.Bullets;
        Position = spaceShip.Position;
        DecoratedSpaceShip = spaceShip;
        BaseDamage = spaceShip.BaseDamage;
    }

    public virtual void Shoot(Position position)
    {
        DecoratedSpaceShip.Shoot(position);
    }
}
```

```
public class DoubleShotSpaceShipDecorator : SpaceShipDecorator
{
   public DoubleShotSpaceShipDecorator(SpaceShip spaceShip) : base(spaceShip)
   {
   }

   public override void Shoot(Position position)
   {
       base.Shoot(position);
       base.Bullets.Add(new Bullet(new Position(position.X+2, position.Y)));
   }
}
```

Command

Norint, kad *controller* nežinotų apie komandas ir jų apdorojimo logiką nusprendėme panaudoti *Command* programavimo šabloną. Tai mums leido nepririšti *View* prie *Controller* ir neturėti didelių *switch* struktūrų, kurios apdoroja skirtingas komandas.



Pav. 10 Command šablono įgyvendinimo UML diagrama.

Kodas:

```
public abstract class Command
{
    public abstract void Execute(string parameter = null);
}
```

```
public class InitializeGameCommand : Command
{
    private readonly IGameController _controller;

    public InitializeGameCommand(IGameController controller)
    {
        _controller = controller;
    }

    public override void Execute(string parameter = null)
    {
        _controller.InitializeGame();
    }

}
```

```
public class RefreshGameCommand : Command
{
```

```
    private readonly IGameController _controller;

    public RefreshGameCommand(IGameController controller)
    {
         _controller = controller;
    }

    public override void Execute(string parameter = null)
    {
        _controller.RefreshGame();
    }
}
```

```
public class AddProfileCommand : Command
{
    private readonly IHomeController _controller;

    public AddProfileCommand(IHomeController controller)
    {
        _controller = controller;
    }

    public override void Execute(string parameter = null)
    {
        _controller.AddPlayerProfile(parameter);
    }
}
```

```
public class OpenGameCommand : Command
{
    private readonly IHomeController _controller;

    public OpenGameCommand(IHomeController controller)
    {
        _controller = controller;
    }

    public override void Execute(string parameter = null)
    {
        _controller.ShowGame(parameter);
    }
}
```

```
public class SelectProfileCommand : Command
{
    private readonly IHomeController _controller;

    public SelectProfileCommand(IHomeController controller)
    {
        _controller = controller;
    }

    public override void Execute(string parameter = null)
    {
        _controller.SelectProfile();
    }
}
```

```
public class ShowPlayerProfilesCommand : Command
{
     private readonly IHomeController _controller;

     public ShowPlayerProfilesCommand(IHomeController controller)
     {
        _controller = controller;
     }

     public override void Execute(string parameter = null)
     {
        _controller.ShowProfiles();
     }
}
```

```
public class GameView : IGameView
{
        private Game _game;

        private Command _refreshGameCommand;

        public string Name => "GameView";

        public void InsertData(object obj)
        {
            _game = (Game) obj;
        }

        public void HandleInput()
        {
            _refreshGameCommand.Execute();
        }

        public void AddController(IController controller)
        {
            _refreshGameCommand = new RefreshGameCommand((IGameController) controller);
        }
 }
```

```
public class HomeView : IHomeView
{
        private IList<Player> _players;

        private Command _selectProfileCommand;
        private Command _addProfileCommand;

        public string Name => "HomeView";

        public void HandleInput()
        {
            var name = Console.ReadLine();
            var user = _players.FirstOrDefault(x => x.Name.Equals(name));
            if(user != null)
            {
                _selectProfileCommand.Execute(name);
            }
```

```
            else
            {
                _addProfileCommand.Execute(name);
            }
        }

        public void AddController(IController controller)
        {
            _selectProfileCommand = new SelectProfileCommand((IHomeController)
controller);
            _addProfileCommand = new AddProfileCommand((IHomeController) controller);
        }
}
```

```
public class MenuView : IMenuView
{
        public string Name => "MenuView";

        private Command _initializeGameCommand;

        public string GetConstantPart()
        {
            var builder = new StringBuilder();
            builder.AppendLine("Select game difficulty");
            builder.AppendLine("1. Easy");
            builder.AppendLine("2. Medium");
            builder.AppendLine("3. Hard");
            return builder.ToString();
        }

        public void HandleInput()
        {
            var strategy = Console.ReadLine();
            if (strategy != null)
            {
                _initializeGameCommand.Execute(strategy);
            }
        }

        public void AddController(IController controller)
        {
            _initializeGameCommand = new OpenGameCommand((IHomeController)controller);
        }
}
```

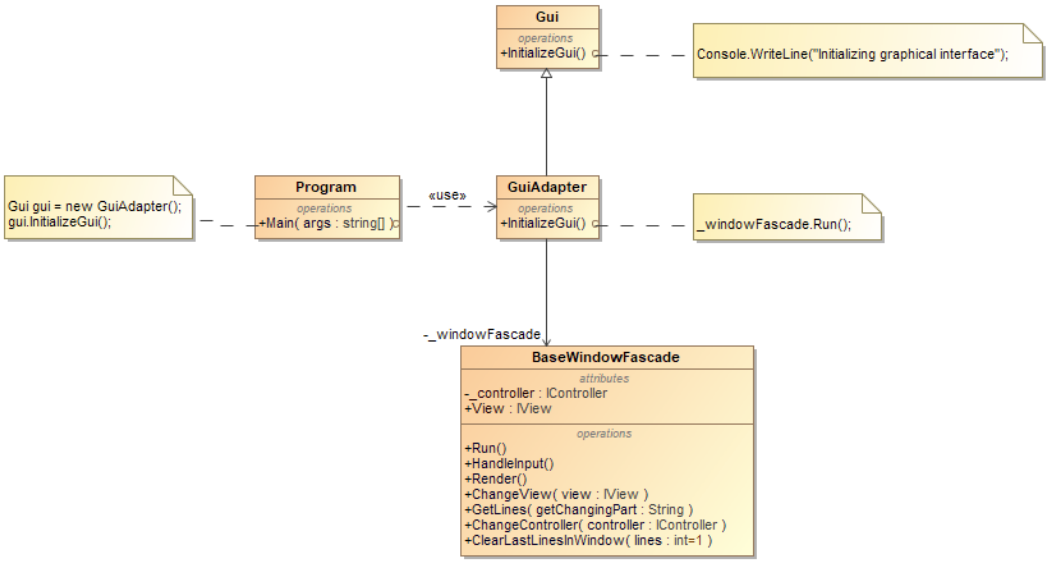
```
public interface IGameController : IController
    {
        void RefreshGame();
        void InitializeGame();
    }
```

```
public interface IHomeController : IController
    {
        void ShowProfiles();
        void ShowGame(string parameter);
        void AddPlayerProfile(string parameter);
        void SelectProfile();
```

```
    }

public class BaseWindowFascade : IWindowFascade
    {
        public IView View { get; private set; }
        private IController _controller;
        private Command _initializeGameCommand;
        private Command _showProfilesCommand;

        public BaseWindowFascade()
        {
            Console.SetWindowSize(160,40);
            Console.CursorVisible = false;
        }

        public void Run()
        {
            Console.WriteLine("Facade works");
            Console.ReadKey();
            _controller = new HomeController(ViewsFactory.Create("HomeController"),
this, new PlayerRepository());
            _showProfilesCommand = new
ShowPlayerProfilesCommand((IHomeController)_controller);
            _showProfilesCommand.Execute();

            while (true)
            {
               Render();
               HandleInput();
            }
        }

        public void ChangeController(IController controller)
        {
            _controller = controller;
            _initializeGameCommand = new
InitializeGameCommand((IGameController)controller);
            _initializeGameCommand.Execute();
        }
    }
```

Adapter

Kadangi planavome naudoti *GUI* sąsają, bet dėl laiko trūkumo, nusprendėme panaudoti komandinės eilutės sąsają, todėl panaudojome *Adapter* programavimo šabloną, kad ateityje galėtume lengvai pakeisti sąsajos tipą.



Pav. 71 Adapter šablono įgyvendinimo UML diagrama.

Kodas:

```
public class Gui
{
   public virtual void InitializeGui()
   {
      Console.WriteLine("Initializing graphical interface");
   }
}
```

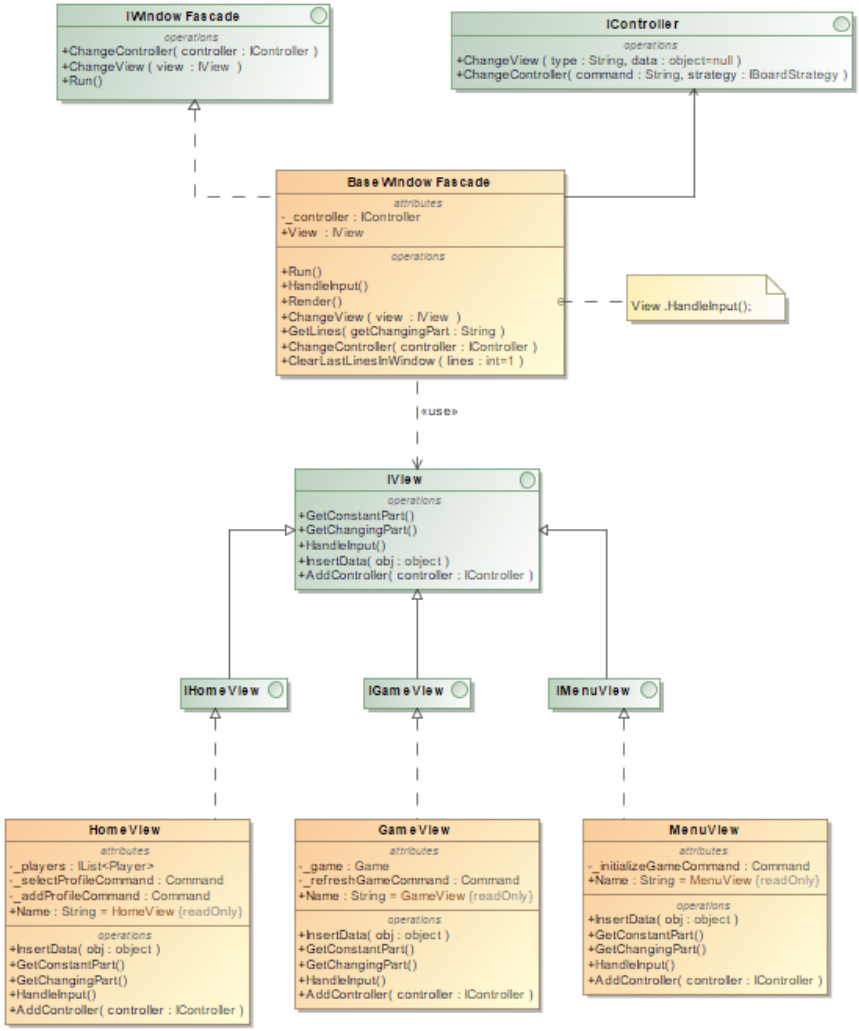
```
public class GuiAdapter : Gui
{
   private readonly BaseWindowFascade _windowFascade = new BaseWindowFascade();

   public override void InitializeGui()
   {
       Console.WriteLine("Adapter works");
       Console.ReadKey();
       _windowFascade.Run();
   }
}
```

```
public class Program
{
   public static void Main(string[] args)
   {
       var diSetup = new DependencyInjectionSetup().GetScope();

       Gui gui = new GuiAdapter();
       gui.InitializeGui();
```

```
    }
}
```

Facade

Kadangi žaidimo metu reikia ne tik atnaujinti grafinę sąsają, bet ir atlikti kitus veiksmus, kaip apdoroti vartotojo įvestį ar įvykdyti komandas, todėl nusprendėme panaudoti *Facade* programavimo šabloną, kad paslėptume šį funkcionalumą.



Pav. 8 Facade šablono įgyvendinimo UML diagrama.

Kodas:

```
public class BaseWindowFascade : IWindowFascade
    {
        public IView View { get; private set; }
        private IController _controller;
        private Command _initializeGameCommand;
        private Command _showProfilesCommand;

        public BaseWindowFascade()
        {
            Console.SetWindowSize(160,40);
```

```
            Console.CursorVisible = false;
        }

        public void Run()
        {
            Console.WriteLine("Facade works");
            Console.ReadKey();
            _controller = new HomeController(ViewsFactory.Create("HomeController"),
this, new PlayerRepository());
            _showProfilesCommand = new
ShowPlayerProfilesCommand((IHomeController)_controller);
            _showProfilesCommand.Execute();

            while (true)
            {
               Render();
               HandleInput();
            }
        }

        private void HandleInput()
        {
            View.HandleInput();
        }

        private void Render()
        {
            if (View != null)
            {
                ClearLastLinesInWindow(GetLines(View.GetChangingPart()));
                Console.Write(View.GetChangingPart());
            }
            else
            {
                Console.WriteLine("Loading...");
                ClearLastLinesInWindow();
            }
        }

        private static int GetLines(string getChangingPart)
        {
            var numLines = getChangingPart.Split('\n').Length;
            return numLines-1;
        }

        public void ChangeView(IView view)
        {
            View = view;
            Console.Clear();
            if (View != null)
            {
                Console.Write(View.GetConstantPart());
            }
        }

        public void ChangeController(IController controller)
        {
            _controller = controller;
```

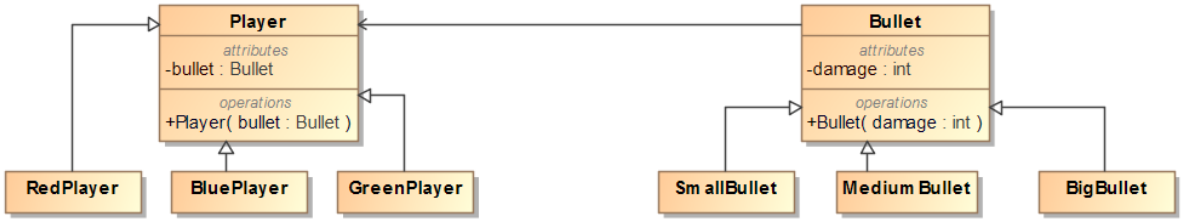
```
            _initializeGameCommand = new
InitializeGameCommand((IGameController)controller);
            _initializeGameCommand.Execute();
        }

        private static void ClearLastLinesInWindow(int lines = 1)
        {
            for (var i = 1; i <= lines; i++)
            {
                if (Console.CursorTop - 1 <= 0) continue;
                Console.SetCursorPosition(0, Console.CursorTop - 1);
                Console.Write(new string(' ', Console.WindowWidth));
                Console.SetCursorPosition(0, Console.CursorTop - 1);
            }
        }
}
```

```
public interface IView
{
    string GetConstantPart();
    string GetChangingPart();
    void HandleInput();
    void InsertData(object obj);
    void AddController(IController controller);

    string Name { get; }
}
```

```
public interface IController
{
    void ChangeView(string type, object data = null);
    IController ChangeController(string command, IBoardStrategy strategy = null);
    IWindowFascade WindowFacade { get; }
}
```

Bridge

Kadangi skirtingi žaidėjai gali tureti skirtingas kulkas. Nusprendėme naudoti „bridge“ programavimo šabloną. Šis šablonas puikiai tinka tokiems atvėjams, nes atskyrėme sąsają nuo jos įgyvendinimo.



Pav 13 Bridge šablono įgyvendinimo UML diagrama.

```
public class Player
{
    Bullet bullet;
    public Player(Bullet bullet)
    {
        this.bullet = bullet;
    }
}
```

```
public class RedPlayer : Player
{
    public RedPlayer(Bullet bullet) : base(bullet) {}
}
```

```
public class GreenPlayer : Player
{
    public GreenPlayer(Bullet bullet) : base(bullet) {}
}
```

```
public class BluePlayer : Player
{
    public BluePlayer(Bullet bullet) : base(bullet) {}
}
```

```
public class Bullet
{
    int damage = 0;
    public Bullet(int damage)
    {
        this.damage = damage;
    }
}
```

```
public class MediumBullet : Bullet
{
    public MediumBullet() : base(2) {}
}
```

```
public class BigBullet : Bullet
{
    public BigBullet() : base(3) {}
}
```