 **KAUNO TECHNOLOGIJOS UNIVERSITETAS**

**INFORMATIKOS FAKULTETAS**

**T120B516 Objektinis programų projektavimas**

**Projektinio darbo ataskaita**

Atliko: Paulius Juozelskis IFF-4/1gr.,

Laimonas Sutkus IFF-4/1gr.,

Tomas Jablonskis IFF-4/1gr.,

Rokas Vaitkevičius IFF-4/3gr.

**KAUNAS 2017**

Turinys

[Įvadas 4](#_Toc499573732)

[Darbo tikslas: 4](#_Toc499573733)

[Naudojamos technologijos: 4](#_Toc499573734)

[Pirmas laboratorinis darbas 5](#_Toc499573735)

[Programavimo šablonai: 5](#_Toc499573736)

[1. Singleton 5](#_Toc499573737)

[2. Factory 5](#_Toc499573738)

[3. Strategy 7](#_Toc499573739)

[4. Observer 9](#_Toc499573740)

[5. Builder 12](#_Toc499573741)

[Antras laboratorinis darbas 16](#_Toc499573742)

[Programavimo šablonai: 16](#_Toc499573743)

[1. Prototype 16](#_Toc499573744)

[2. Decorator 18](#_Toc499573745)

[3. Command 20](#_Toc499573746)

[4. Adapter 25](#_Toc499573747)

[5. Façade 27](#_Toc499573748)

[Trečias laboratorinis darbas 29](#_Toc499573749)

[Programavimo šablonai: 29](#_Toc499573750)

[1. Template method 29](#_Toc499573751)

[2. Iterator 30](#_Toc499573752)

[3. Composite 31](#_Toc499573753)

[4. Flyweight 31](#_Toc499573754)

[5. State 32](#_Toc499573755)

[Ketvirtas laboratorinis darbas 33](#_Toc499573756)

[Programavimo šablonai: 33](#_Toc499573757)

[1. Chain of responsibility 33](#_Toc499573758)

[2. Interpreter 34](#_Toc499573759)

[3. Mediator 35](#_Toc499573760)

[4. Memento 35](#_Toc499573761)

[5. Visitor 36](#_Toc499573762)

[6. Null Object 37](#_Toc499573763)

[Išvados 37](#_Toc499573764)

# 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

## Naudojamos technologijos:

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

# Pirmas laboratorinis darbas

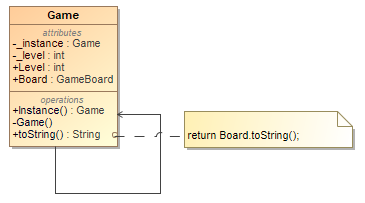
## Programavimo šablonai:

### 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:

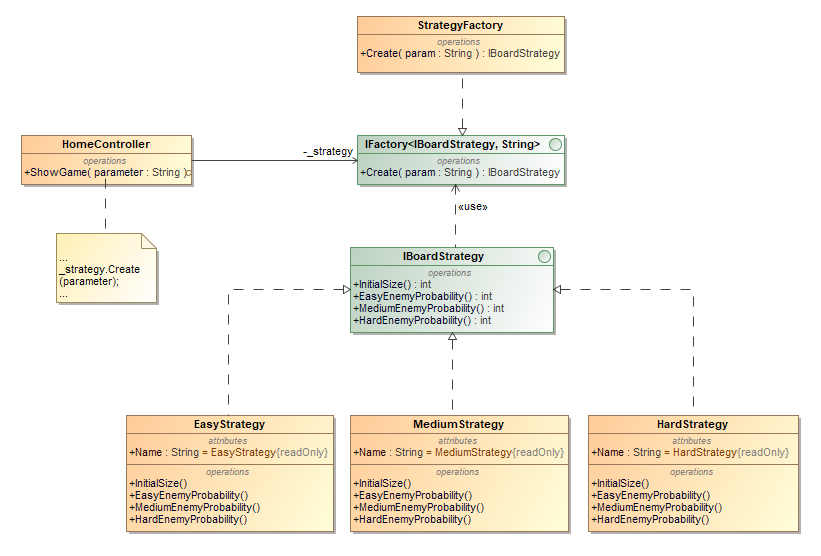
|  |
| --- |
| public class Game  {  public GameBoard Board { get; set; }  public int Level { get; set; }  private int \_level { get; set; }  private static Game \_instance;  public static Game Instance => \_instance ?? (\_instance = new Game()); // Singleton  private Game() { }  public override string 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:

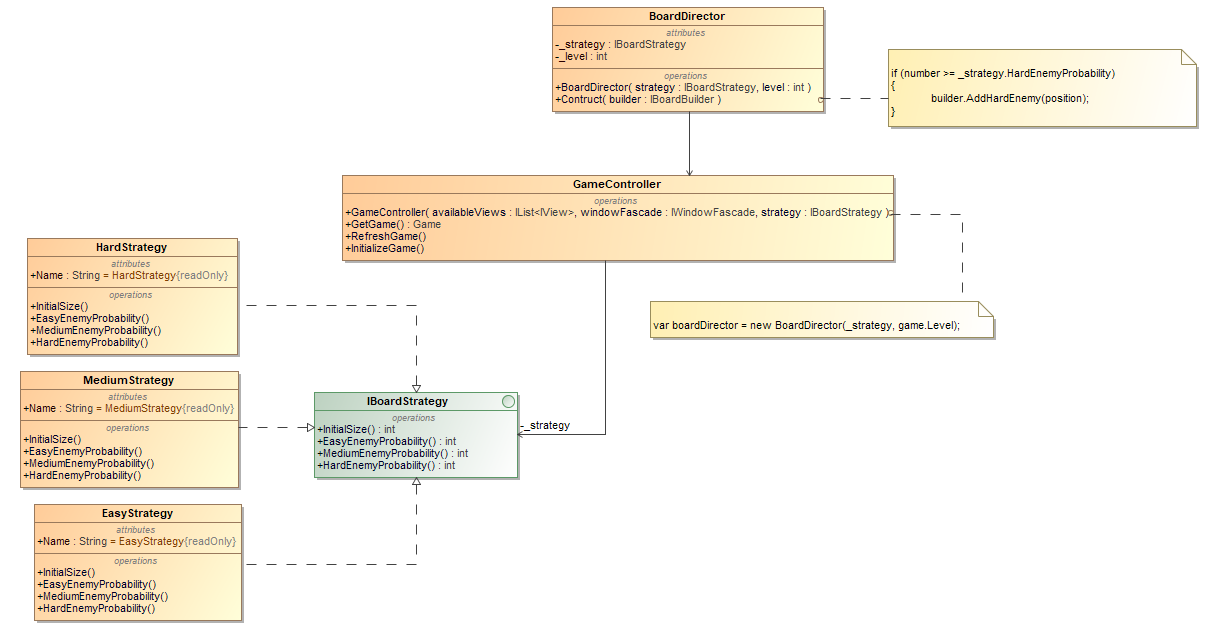
|  |
| --- |
| public class StrategyFactory : IFactory<IBoardStrategy, string>  {  public IBoardStrategy Create(string param)  {  switch (param)  {  case HardStrategy.Name:  return new HardStrategy();  case MediumStrategy.Name:  return new MediumStrategy();  default:  return new EasyStrategy();  }  }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

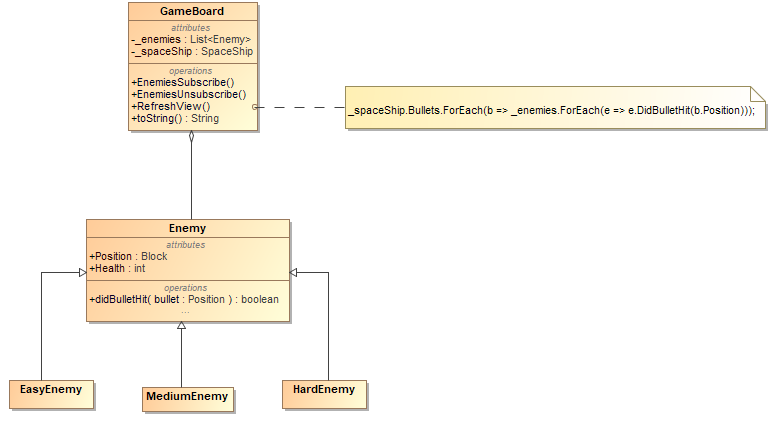
|  |
| --- |
| public interface IBoardStrategy  {  int InitialSize { get; }  int EasyEnemyProbability { get; }  int MediumEnemyProbability { get; }  int HardEnemyProbability { get; }  } |
|  |
| public class EasyStrategy : IBoardStrategy  {  public const string Name = "EasyStrategy";  public int InitialSize => 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:

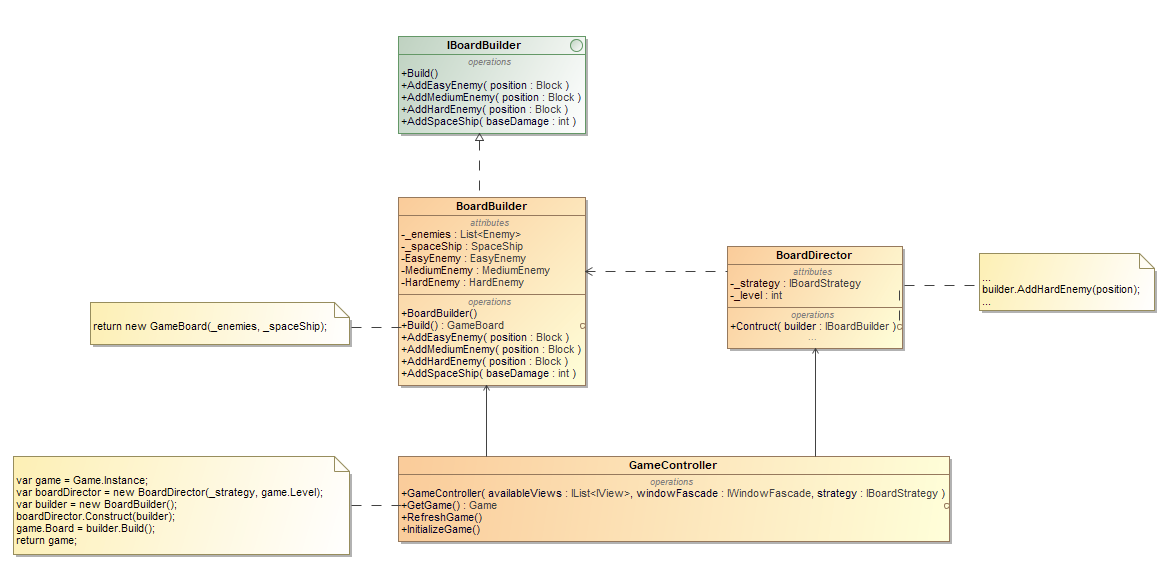
|  |
| --- |
| public class GameBoard // Subject  {  private SpaceShip \_spaceShip;  private readonly List<Enemy> \_enemies; // enemies subscribers  public int EnemiesCount => \_enemies.Count;  public GameBoard(SpaceShip spaceShip)  {  \_spaceShip = spaceShip;  \_enemies = new List<Enemy>();  }  // Adding 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, 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);  }  } |

# Antras laboratorinis darbas

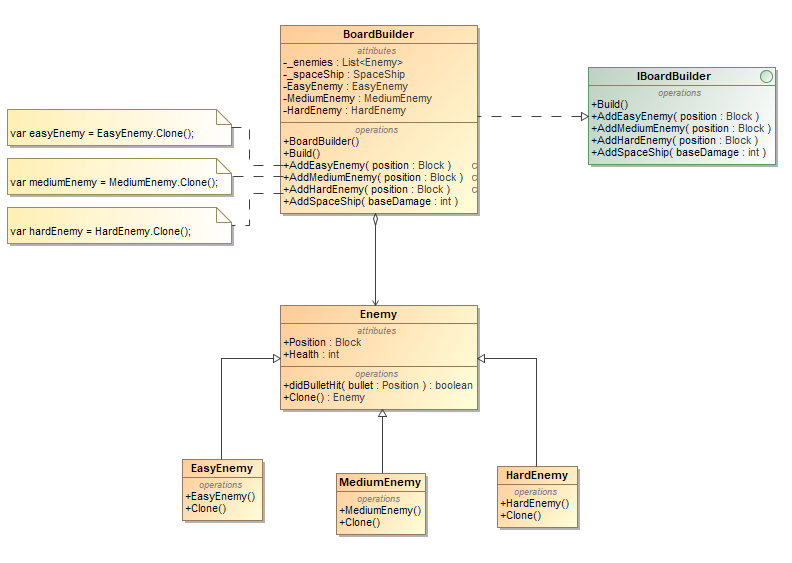
## Programavimo šablonai:

### Prototype

Panaudojimo tikslas:

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.

Uml diagrama:



Pav. 7 Prototype programavimo šablonas

Kodas:

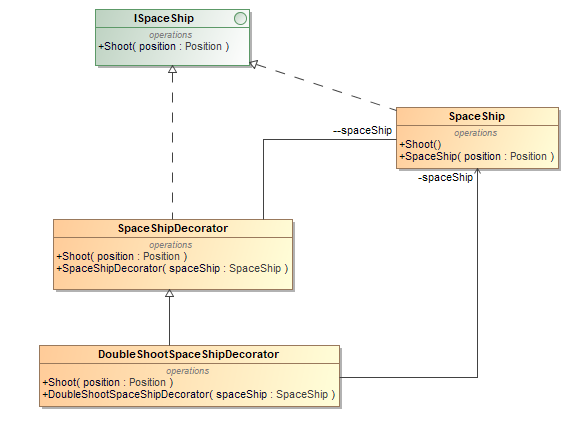
|  |
| --- |
| public class BoardBuilder : IBoardBuilder  {  private List<Enemy> \_enemies = new List<Enemy>();  private SpaceShip \_spaceShip = new SpaceShip();  private EasyEnemy EasyEnemy { get; }  private MediumEnemy MediumEnemy { get; }  private HardEnemy HardEnemy { get; }  public BoardBuilder()  {  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 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

Panaudojimo tikslas:

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

Uml diagrama:



Pav. 8 Decorator programavimo šablonas

Kodas:

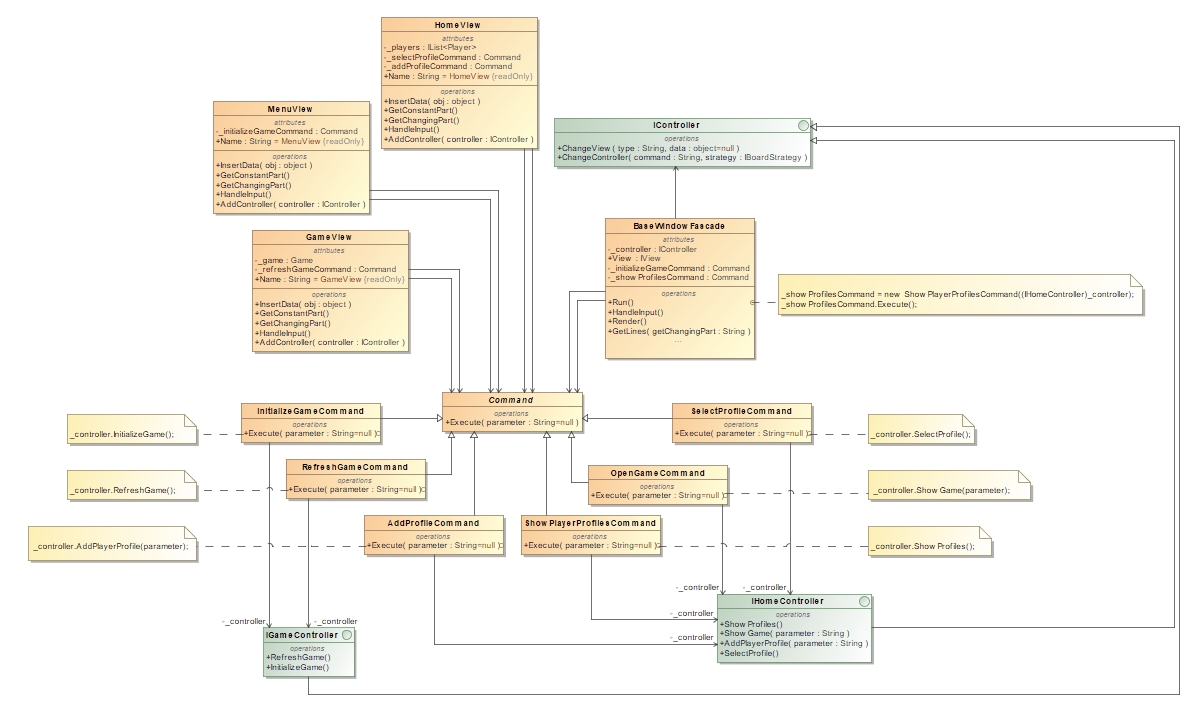
|  |
| --- |
| public interface ISpaceShip  {  List<Bullet> Bullets { get; set; }  Block Position { get; set; }  int BaseDamage { get; }  void Shoot(Position position);  } |
|  |
| public class SpaceShip : ISpaceShip  {  public List<Bullet> Bullets { get; set; }  public Block Position { get; set; }  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

Panaudojimo tikslas:

Pirmo labarotorinio darbo metu kuriamoje sistemoje buvo naudotos string tipo komandos, kurios buvo apdorojamos “controller” objektuose, tačiau išaugus komandų kiekiui ir norint, kad kontroleris nežinotų apie komandas ir jų apdorojimo logiką nusprendėme panaudoti Command programavimo šabloną. Tai mums leido nepririšti View prie Controllerio ir neturėti didelių switch struktūrų, kurios apdoroja skirtingas komandas.

Uml diagrama:



Pav. 9 Visos įgyvendintos komandos pasinaudojant Command programavimo šablonu

Kodas:

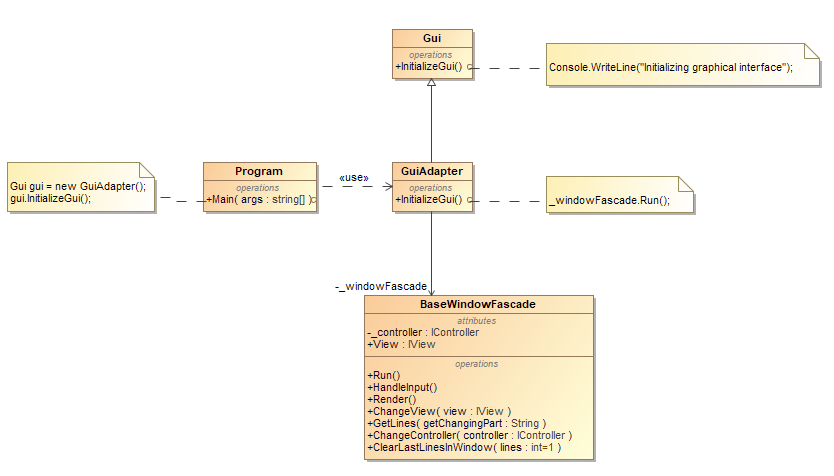
|  |
| --- |
| public abstract class Command  {  public abstract void Execute(string parameter = null);  } |
|  |
| public class InitializeGameCommand : Command  {  private readonly IGameController \_controller;  public InitializeGameCommand(IGameController controller)  {  \_controller = controller;  }  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

Panaudojimo tikslas:

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

Uml diagrama:



Pav. 10 Adapter programavimo šablonas

Kodas:

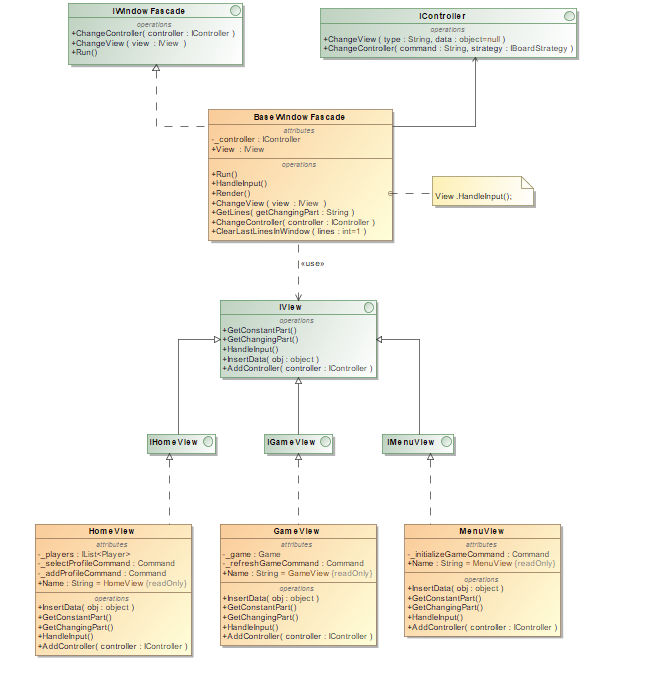
|  |
| --- |
| public class Gui  {  public virtual void InitializeGui()  {  Console.WriteLine("Initializing graphical interface");  }  } |
|  |
| public class GuiAdapter : Gui  {  private readonly BaseWindowFascade \_windowFascade = new BaseWindowFascade();  public override void InitializeGui()  {  Console.WriteLine("Adapter 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();  }  } |
|  |

### Façade

Panaudojimo tikslas:

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

Uml diagrama:



Pav. 11 Facade programavimo šablonas

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; }  } |

# Trečias laboratorinis darbas

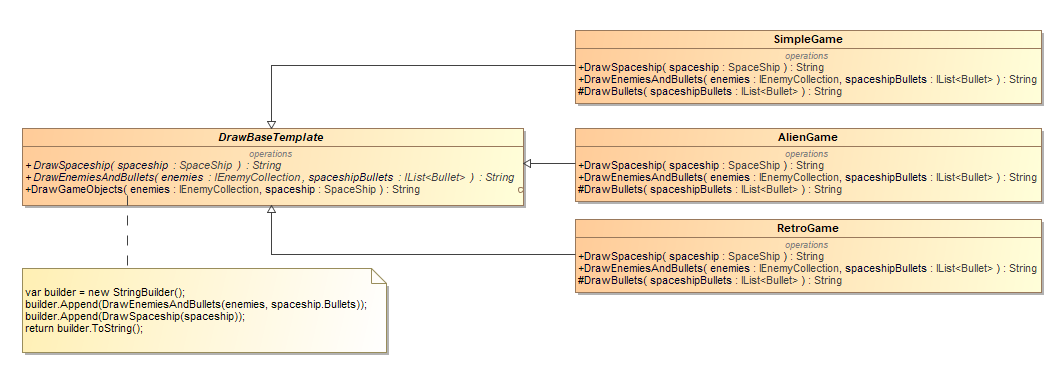
## Programavimo šablonai:

### Template method

Panaudojimo tikslas:

Šį šabloną pasirinkome skirtingai atvaizduoti žaidimo elementus, tokius kaip priešai ir laivas. Kadangi kodas ir išvaizda skiriasi šių dalykų priklausomai nuo sunkumo, o jų tikslas toks pats, tai todėl ir pasirinkome template method šabloną. Taip pat šis metodas padeda nesunkiai ateityje praplėsti ar pakeisti pagrindinių žaidimo elementų išvaizdą.

UML diagrama:



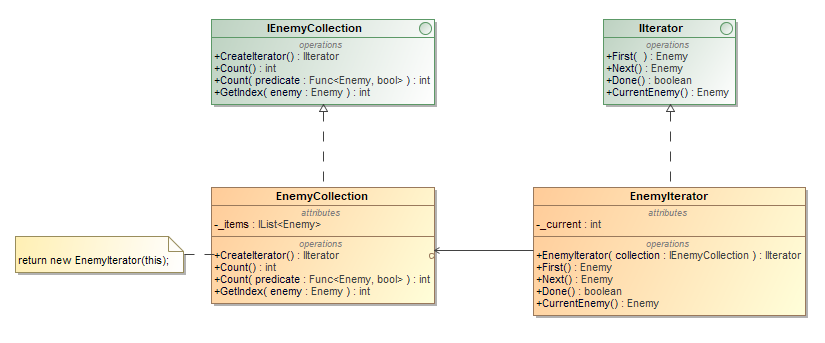
12 pav. Template method šablonas

### Iterator

Panaudojimo tikslas:

Šį programavimo šabloną pasirinkome, kadangi norėjome paslėpti, kaip yra implementuota priešų saugojimo logika. Ateityje pasinaudojus šiuo šablonu galima optimizuoti programą modifikuojant vidinę priešų saugojimo struktūrą į „binary-tree“.

UML diagrama:



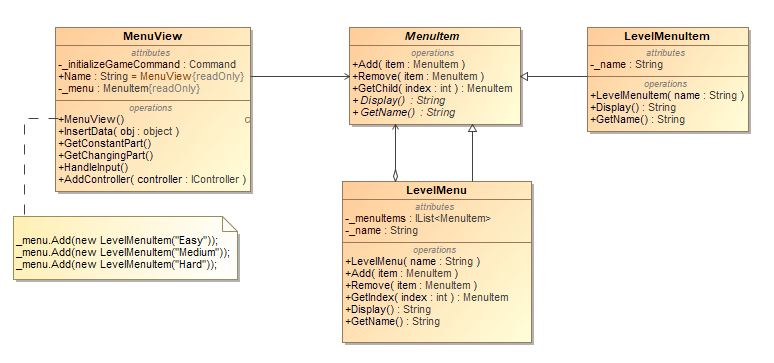
13 pav. Iterator šablonas

### Composite

Panaudojimo tikslas:

Šio programavimo šablono pasirinkimą nulėmė per daug sudėtinga logika implementuojant žaidimo lygių „meniu“, kadangi kiekvienas meniu pasirinkimas lemia skirtingą įgyvendinimą, norėjome jį išskirti į atskirus komponentus. Taipogi šis programavimo šablonas mums leido modifikuoti grafinę sąsają, kad ji būtų sudaryta iš atskirų modulių, o ne vieno monolitinio modulio.

UML diagrama:



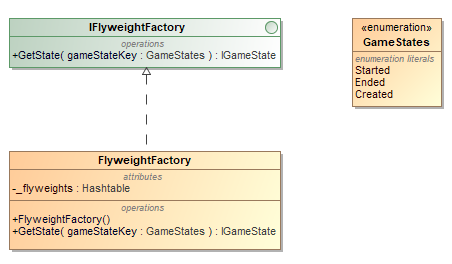
14 pav. Composite šablonas

### Flyweight

Panaudojimo tikslas:

Šablonas panaudotas saugoti visas galimas žaidimo būsenas (States). Šis šablonas naudingas tuo kad naudojamas vis tas pats objektas, todėl sutaupoma resursų ir nereikia vis kurti žaidimo būsenų objektų per naują, nes žaidimo būsena nuolatos keičiasi, todėl būtų neefektyvu vis perkurti būsenų objektus.

UML diagrama:



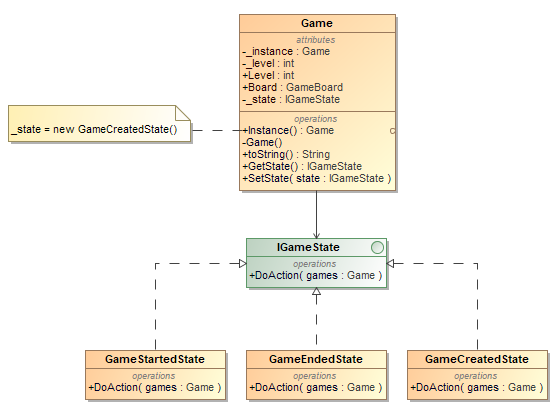
15 pav. Flyweight šablonas

### State

Panaudojimo tikslas:

Būsenos šabloną panaudojome atskirti žaidimo logikai priklausomai nuo to kokioje būsenoje yra žaidimas. Žaidimas gali būti trijose būsenose: sukurtas, pradėtas ir užbaigtas. Kiekviena būsena turi skirtinga implementaciją. Šablonas padeda ją geriau atskirti.

UML diagrama:



16 pav. State šablonas

# Ketvirtas laboratorinis darbas

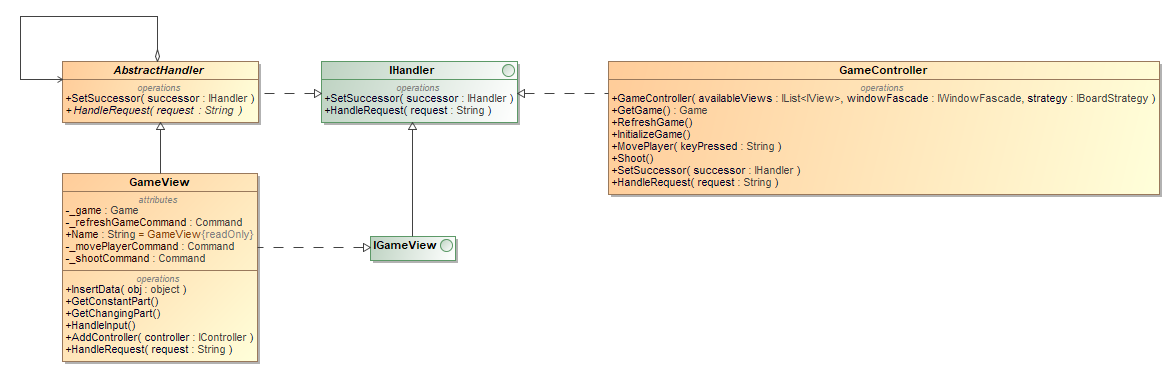
## Programavimo šablonai:

### Chain of responsibility

Panaudojimo tikslas:

Šį programavimo šabloną pasirinkimo įgyvendinti skirtingoms įvestims, kurios yra apdorojamos skirtinguose lygmeniuose. Žaidėjas žaidimo metu gali sustabdyti žaidimą aukščiausiame „IGameView“ lygmenyje, tačiau norint palikti žaidimą logika turi būti įgyvendinta controllerio lygmenyje, kadangi reikia pašalinti nenaudojamą informaciją.

UML diagrama:



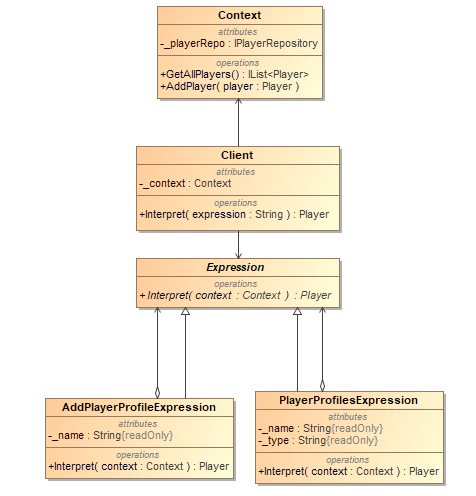
17 pav. Chain of responsibility šablonas

### Interpreter

Panaudojimo tikslas:

Interpreter programavimo šabloną pasirinkome įgyvendinti vartotojo profilio pasirinkimui/kūrimui pasitelkiant žodinę kalbą. Naudojantis šiuo šablonu galėjome nesunkiai išskaidyti vartotojo įvedamus teiginius į atskirus reiškinius (expressions), kuriuos galime įvykdyti ir gauti/pridėti vartotojo profilį į duomenų bazę.

UML diagrama:



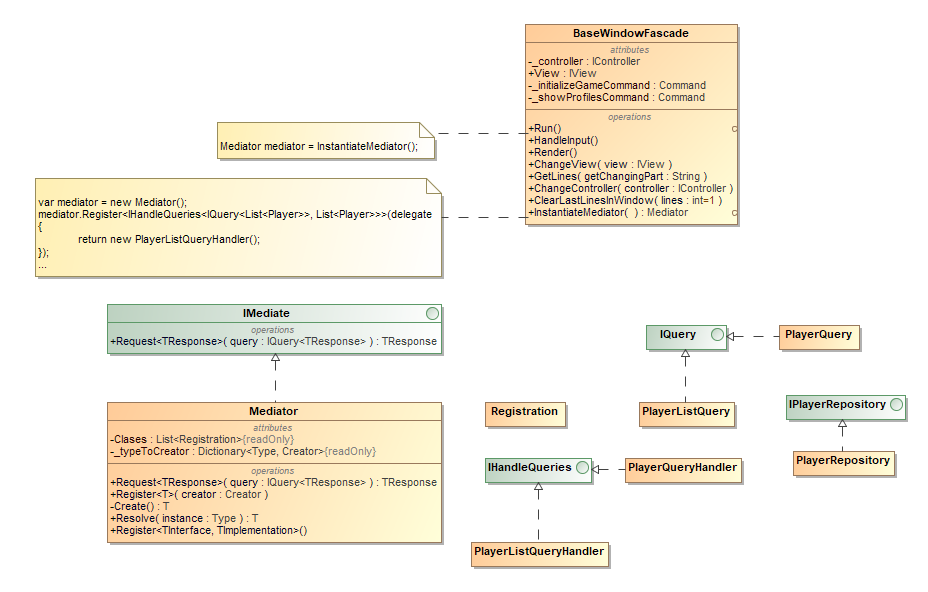
17 pav. Interpreter šablonas

### Mediator

Panaudojimo tikslas:

Mediator šabloną panaudojome rašyti užklausas į duomenų bazę. Jis padeda mums enkapsuliuoti logiką ir neužteršti repozitorijos užklausomis. Mediatorius mūsų atveju deleguos mūsų užklausos objektus atitinkamiems „handler‘iams“, kurie atliks užklausą ir grąžins rezultatą. Galima teigti, kad mediatoriuje susitinka „handler‘io“ ir užklausos objektai.

UML diagrama:



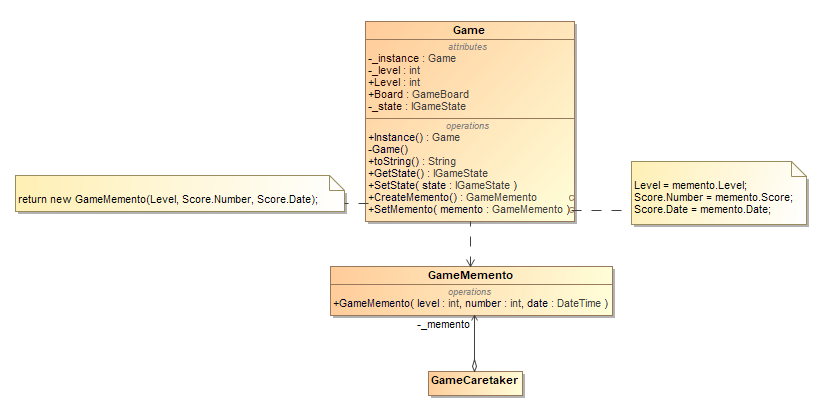
18 pav. Mediator šablonas

### Memento

Panaudojimo tikslas:

Memento programavimo šabloną pasirinkome norėdami išsaugomi žaidimo stadiją, jeigu vartotojas nori toliau tęsti žaidimą, nuo paskutinio lygio kuriame jis buvo ir išsaugoti savo rezultatą. Prieš tai žaidimo stadija buvo saugoma duomenų bazėje, tačiau šio programavimo šablono panaudojimas mums leido to atsisakyti ir duomenis saugoti atmintyje

UML diagrama:



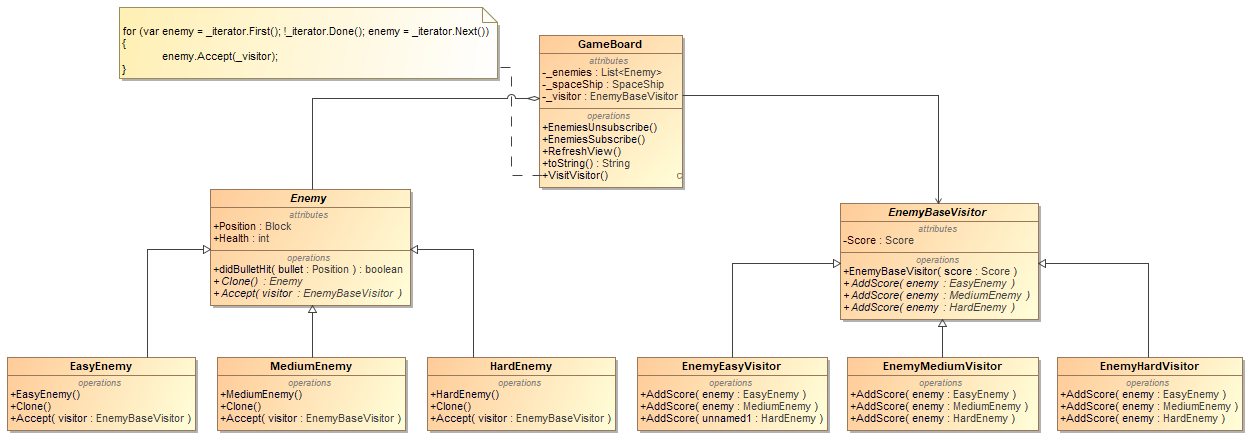
19 pav. Memento šablonas

### Visitor

Panaudojimo tikslas:

Visitor šabloną panaudojome taškų skaičiavimui. Priklausomai nuo tai koks žaidimo sunkumas ir pagal tai nuo to kokį priešą (sunkų, lengvą, vidutinį) nušovė žaidėjas, aplankomas visitor šablonas ir taškai pagal kriterijus pridedami prie sumos.

UML diagrama:



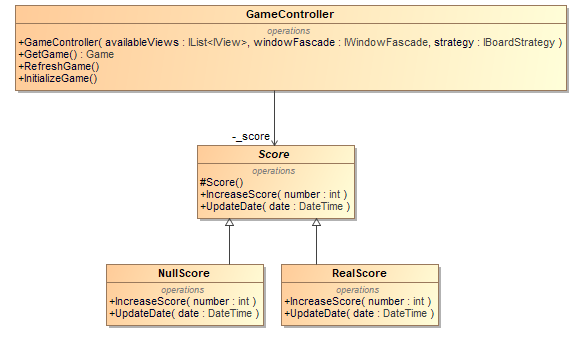
20 pav. Visitor šablonas

### Null Object

Panaudojimo tikslas:

Šį programavimo šabloną pasirinkome norėdami pašalinti neaiškumus, kaip apdoroti rezultatus, kurie nėra priskirti (null reikšmė). Naudojantis šiuo šablonu, apibrėžime objekto struktūrą, kuri neleidžia pakeisti rezultato ir yra nustatyta į nulines reikšmes.

UML diagrama:



21 pav. Null object šablonas

# Išvados

Programavimo šablonai gali palengvinti įgyvendinti ar patobulinti funkcionalumo implementaciją, tačiau darydami šį projektą pastebėjome, kad perteklinis programavimo šablonų naudojimas daugiau apsunkina programinės įrangos architektūrą nei ją palengvina. Todėl manome, kad kiekvieno programavimo šablono naudojimas turi būti reikalingas, o ne ieškoti vietų kuriose būtų galima panaudoti vieną ar kitą programavimo šabloną.