

Informatikos fakultetas

**T120B516 Objektinis programų projektavimas**

**2-ojo laboratorinio darbo ataskaita**

|  |  |  |
| --- | --- | --- |
|  | Studentai: | Ignas Damijonaitis,  Lukas Raila,  Marius Blažys  IFF-4/1 |
| Dėstytojas: | Andrej Ušaniov |

Kaunas 2017

Turinys

[1. State 3](#_Toc500313195)

[2. Chain of Responsibility 5](#_Toc500313196)

[3. Memento 7](#_Toc500313197)

[4. Proxy 10](#_Toc500313198)

[5. Mediator 12](#_Toc500313199)

[6. Template 14](#_Toc500313200)

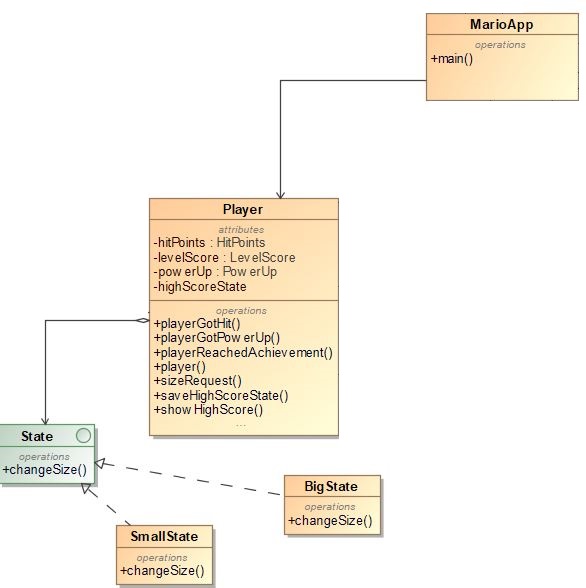
[7. Iterator 17](#_Toc500313201)

[8. Null Object 19](#_Toc500313202)

[9. Išvados 21](#_Toc500313203)

# State

UML diagrama:



Pav 1 State pattern

Kodo fragmentai:

Player.cs:

public class Player

{

private int hitPoints;

private int levelScore;

private int powerUp;

private HitPoints \_hitPoints;

private LevelScore \_levelScore;

private PowerUp \_powerUp;

GunFactory gunFactory;

Gun gun;

private IState \_state;

private int highScoreState;

public Player(int newHitPoints, int newLevelScore, int newPowerUp, IState state)

{

hitPoints = newHitPoints;

levelScore = newLevelScore;

powerUp = newPowerUp;

\_hitPoints = new HitPoints();

\_levelScore = new LevelScore();

\_powerUp = new PowerUp();

this.\_state = state;

}

public IState IState

{

get { return \_state; }

set

{

\_state = value;

Console.WriteLine("Player's size: " + \_state.GetType().Name);

}

}

public void sizeRequest()

{

\_state.changeSize(this);

}

}

IState.cs:

public interface IState

{

void changeSize(Player player);

}

SmallState.cs:

public class SmallState : IState

{

public void changeSize(Player player)

{

player.IState = new BigState();

}

}

BigState.cs:

public class BigState : IState

{

public void changeSize(Player player)

{

player.IState = new SmallState();

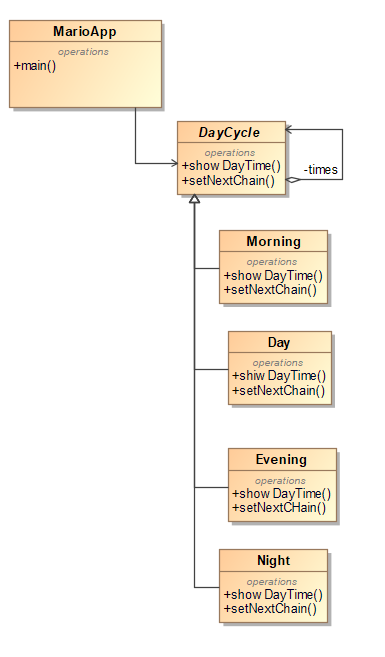
}

}

Šablono tikslas – keisit objekto elgesį, priklausomai nuo būsenos. Šiame pavyzdyje keičiasi žaidėjo dydis per interfeisą „Istate“. „Player“ klasė tiesiogiai neimplementuoja būsenos elgsenos. Tai „Player“ klasę padaro nepriklausomą nuo to, kaip konkreti elgsena yra implementuojama. „SmallState“ ir „BigState“ klasės implementuoja „Istate“ interfeisą. Taip žaidėjas eidamas per trąsą, tai padidėja, tai pamažėja, priklausomai nuo atliktų pasiekimų.

# Chain of Responsibility

UML diagrama:



Pav 2 Chain of Responsibility pattern

Kodo fragmentai:

DayCycle.cs:

abstract class DayCycle

{

protected DayCycle successor;

public void setNextChain(DayCycle successor)

{

this.successor = successor;

}

public abstract void showDayTime(int time);

}

Morning.cs:

class Morning : DayCycle

{

public override void showDayTime(int time)

{

if (time >= 6 && time < 12)

{

Console.WriteLine("It's Morning now!");

}

else if (successor != null)

{

successor.showDayTime(time);

}

}

}

Day.cs:

class Day : DayCycle

{

public override void showDayTime(int time)

{

if (time >= 12 && time < 18)

{

Console.WriteLine("It's Day now!");

}

else if (successor != null)

{

successor.showDayTime(time);

}

}

}

Evening.cs:

class Evening : DayCycle

{

public override void showDayTime(int time)

{

if (time >= 18 && time < 24)

{

Console.WriteLine("It's Evening now!");

}

else if (successor != null)

{

successor.showDayTime(time);

}

}

}

Night.cs:

class Night : DayCycle

{

public override void showDayTime(int time)

{

if (time >= 0 && time < 6)

{

Console.WriteLine("It's Night now!");

}

else if (successor != null)

{

successor.showDayTime(time);

}

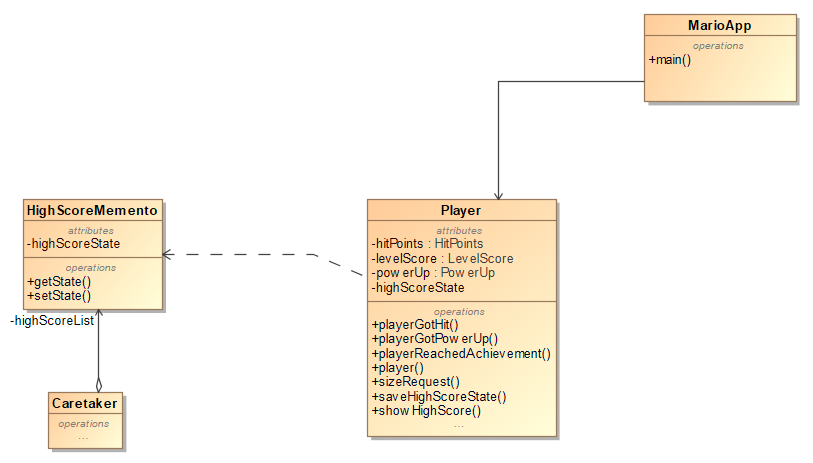
}

}

Chain of Responsibility šablonas leidžia atskirti užklausos gavėją nuo siuntėjo, kad jie nepriklausytų vienas nuo kito ir kad daugiau negu vienas objektas galėtų įvykdyti užklausą. Realizuotu atveju pagal duotą valandą, tikrinama koks dienos metas yra tuo metu žaidime. Šablonas išskaido objektų kūrimą į atskiras klases, kurios gali kisti nepriklausomai.

# Memento

UML diagrama:



Pav 3 Memento pattern

Kodo fragmentai:

Player.cs:

public class Player

{

private int hitPoints;

private int levelScore;

private int powerUp;

private HitPoints \_hitPoints;

private LevelScore \_levelScore;

private PowerUp \_powerUp;

GunFactory gunFactory;

Gun gun;

private IState \_state;

private int highScoreState;

public Player(int newHitPoints, int newLevelScore, int newPowerUp, IState state)

{

hitPoints = newHitPoints;

levelScore = newLevelScore;

powerUp = newPowerUp;

\_hitPoints = new HitPoints();

\_levelScore = new LevelScore();

\_powerUp = new PowerUp();

this.\_state = state;

}

public int HighScoreState

{

get { return highScoreState; }

set

{

highScoreState = value;

Console.WriteLine("High score: " + highScoreState);

}

}

public HighScoreMemento showHighScore()

{

return (new HighScoreMemento(highScoreState));

}

public void saveHighScoreState(HighScoreMemento memento)

{

Console.WriteLine("Restoring high score...");

HighScoreState = memento.HighScoreState;

}

}

HighScoreMemento:

public class HighScoreMemento

{

private int highScoreState;

public HighScoreMemento(int state)

{

this.highScoreState = state;

}

public int HighScoreState

{

get { return highScoreState; }

}

}

Caretaker.cs:

class Caretaker

{

private HighScoreMemento \_memento;

public HighScoreMemento HighScoreMemento

{

set { \_memento = value; }

get { return \_memento; }

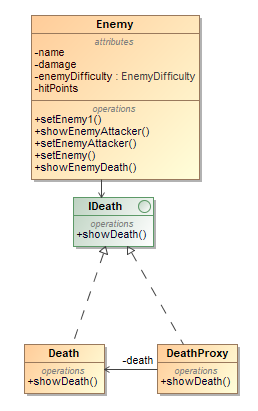
}

}

Memento šablonas nepažeidžiant enkapsuliacijos, užfiksuoja objekto būseną, kad tas objektas galėtų vėliau į tą būseną grįžti. Realizuotame pavyzdyje yra saugomas žaidėjo rekordas ir žaidimo metu yra surenkamas naujas rekordas, todėl jeigu naujasis rekordas nepralenkia senojo, rekordo būsena grįžta į pradinę. Šablonas implementuotas trimis objektais – „Player“ (Originator), „HighScoreMemento“ (Memento) ir „Caretaker“. „Player‘ – objektas turintis tą būseną. „Caretaker“ – keičia žaidėjo rekordą, bet gali ir panaikinti pakeitimą.

# Proxy

UML diagrama:



Pav 4 Proxy pattern

Programinis kodas:

Enemy.cs

public class Enemy

{

private String name;

private int damage;

private int hitPoints;

private EnemyDifficulty \_enemyDifficulty;

private IEnemyAttacker enemyAttacker = null;

public Enemy(EnemyDifficulty difficulty)

{

this.\_enemyDifficulty = difficulty;

}

public void setEnemy()

{

\_enemyDifficulty.addDifficulty();

}

public void setEnemyAttacker(IEnemyAttacker attacker)

{

enemyAttacker = attacker;

}

public void showEnemyAttacker()

{

enemyAttacker.Fire();

enemyAttacker.Move();

}

public void showEnemyDeath()

{

DeathProxy deathProxy = new DeathProxy();

deathProxy.ShowDeath();

}

}

IDeath.cs

public interface IDeath

{

void ShowDeath();

}

Death.cs

class Death : IDeath

{

public void ShowDeath()

{

Console.WriteLine("Enemy died magnificently");

}

}

DeathProxy.cs

class DeathProxy : IDeath

{

private Death realDeath;

public void ShowDeath()

{

if (realDeath == null)

{

realDeath = new Death();

}

realDeath.ShowDeath();

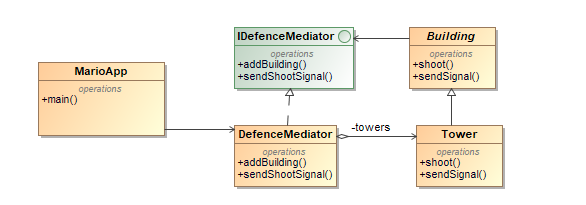
}

}

Proxy šablono tikslas yra sukurti pagrindinės klasės pakaitalą, kuris gali turi priėjimą prie pagrindinės klasės. Mūsų projekte buvo sukurta Death klasė, kuri atvaizduoja priešo mirtį, ir DeathProxy, kuri turi priėjimą prie Death klasės. Abi klasės naudoją tą patį Ideath interfeisą ir implementuoja tuos pačius metodus skirtingai.

# Mediator

UML diagrama:



Pav 5 Mediator pattern

Programinis kodas:

Building.cs

public abstract class Building

{

protected IDefenceMediator mediator;

public Building(IDefenceMediator mediator)

{

this.mediator = mediator;

}

public abstract void SendSignal(string signal);

public abstract void Shoot(string signal);

}

IDefenceMediator.cs

public interface IDefenceMediator

{

void AddBuildng(Building building);

void SendShootSignal(string signal);

}

DefenceMediator.cs

public class DefenceMediator : IDefenceMediator

{

List<Building> towers = new List<Building>();

public void AddBuildng(Building building)

{

towers.Add(building);

}

public void SendShootSignal(string signal)

{

foreach(Building building in towers){

                building.Shoot(signal);

            }

        }

}

Tower.cs

public class Tower : Building

{

// Constructor

public Tower(IDefenceMediator mediator)

: base(mediator)

{

}

public override void SendSignal(string signal)

{

mediator.SendShootSignal(signal);

}

public override void Shoot(string signal)

{

Console.WriteLine("Tower gets signal: "

+ signal + " Starting to shoot!");

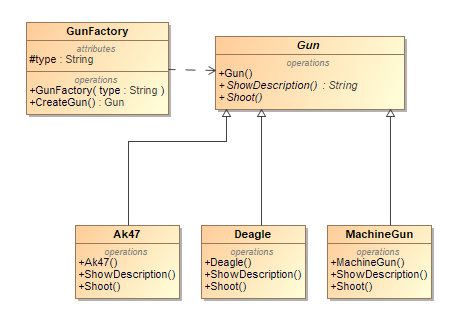
}

}

Mediator šablonas sukuria objektą, kuris nusako kaip turi elgtis rinkinys kitų objektų. Jis naudojamas kai projekte galimai yra daug nepriklausomų objektų, kiekvienas objektas žino apie kitus esančius objektus. Mūsų projekte yra DefenceMediator, kuris turi sąrašą Tower objektų. Vienas bokštas išsiuntes signalą, aktyvuoja kitus sąraše esančius bokštus.

# Template

UML diagrama:



Pav 6 Template pattern

Programinis kodas:

GunFactory.cs

public class GunFactory

{

protected string type;

public GunFactory(string t)

{

type = t;

}

public Gun createGun()

{

Console.WriteLine(type);

if (type.Equals("Ak47")){

return new Ak47();

}

else if (type.Equals("Deagle")){

return new Deagle();

}

else {

return new MachineGun();

}

}

}

Gun.cs

public abstract class Gun

{

public Gun()

{

}

public abstract string ShowDescription();

public abstract string Shoot();

}

Ak47.cs

public class Ak47 : Gun

{

public Ak47()

{

}

public override string ShowDescription()

{

return "Ak47";

}

public override string Shoot()

{

return "Ak47 shooting";

}

}

Deagle.cs

public class Deagle : Gun

{

public Deagle()

{

}

public override string ShowDescription()

{

return "Deagle";

}

public override string Shoot()

{

return "Deagle shooting";

}

}

MachineGun.cs

public class MachineGun : Gun

{

public MachineGun()

{

}

public override string ShowDescription()

{

return "MachineGun";

}

public override string Shoot()

{

return "MachineGun Tratatatatatatatatata";

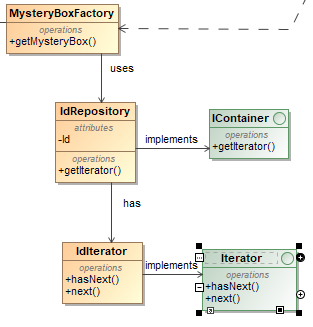
}

}

Template šablonas leidžia subklasėms pakeisti algoritmo žingsnius nekeičiant orginalios algoritmo struktūros. *Factory* metodai dažnai yra naudojami kartu su *template* šablonu, mūsų projekte šie šablonai taip pat yra implemetuoti. *GunFactory* klasė gali sukurti *Gun* tipo objektą, kuris gali atvaizduoti vieną iš trijų ginklų klasių: *Ak47, Deagle, MachineGun*. Kiekviena paveldinti ginklų klasė turi skirtingai implementuotus metodus.

# Iterator

UML diagrama:



Kodas:

|  |
| --- |
| namespace Mario2k17.Map.Iterator  {      interface Iterator      {          Boolean hasNext();          int next();      }  } |

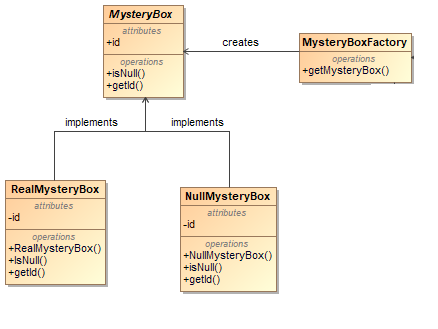
|  |
| --- |
| namespace Mario2k17.Map.Iterator  {      class IdRepository : IContainer      {          public static int[] ids = { 0, 1, 2, 3, 4 };          IdIterator idIterator = new IdIterator();          public new Iterator GetITerator()          {                return idIterator;          }            class IdIterator : Iterator          {                int index = 0;                public bool hasNext()              {                  if (index < ids.Length)                  {                      return true;                  }                  return false;          }                public int next()              {                  if(this.hasNext())                  {                      return ids[index++];                  }                  return -5;              }          }      }  } |

|  |
| --- |
| namespace Mario2k17.Map.Iterator  {      interface IContainer      {          Iterator GetITerator();      }  } |

„Iterator“ šablonas leidžia pasiimti elementus iš grupės paeiliui, nesukant galvos dėl eilės ar tvarkos, nes elementai yra imami iš eilės kaip yra nurodyta be jokio papildomo kodo. Aš šį šabloną panaudojau imdamas „MysteryBox id“ iš eilės kiekvieną kartą, kai yra kuriamas naujas objektas.

# Null Object

UML diagrama:



Kodas:

|  |
| --- |
| namespace Mario2k17.Map  {      class MysteryBoxFactory      {          public int[] ids = { 0, 1, 2, 3, 4 };            IdRepository newID = new IdRepository();            public MysteryBox GetBox()          {              int id = newID.GetITerator().next();              for (int i = 0; i < ids.Length; i++)              {                  if (ids[i] == id)                  {                      return new RealMysteryBox(id);                  }              }                return new NullMsteryBox();          }      }  } |

|  |
| --- |
| namespace Mario2k17.Map  {      abstract class MysteryBox      {          public int Id;              abstract public bool isNull();          abstract public int getId();        }  } |

|  |
| --- |
| namespace Mario2k17.Map  {      class NullMsteryBox : MysteryBox      {              public override int getId()          {              return -1;          }            public override bool isNull()          {              return true;          }      }  } |

|  |
| --- |
| namespace Mario2k17.Map  {      class RealMysteryBox : MysteryBox      {          public RealMysteryBox(int id)          {              this.Id = id;          }            public override int getId()          {              return Id;          }            public override bool isNull()          {              return false;          }      }  } |

„Null Object“ šablonas pakeičia tikrinimą – ar objektas nėra lygus „null“; pritaikius šį šabloną, objektas, kuris yra lygus „null“, nieko nedaro. Šis šablonas gali būti naudojamas apibūdinti standartinį objektą, kuris funkcionuos neturėdamas jokios informacijos.

# Išvados