Kauno technologijos universitetas

Informatikos fakultetas

Projektinis darbas nr. 1

T120B516 Objektinis programų projektavimas

Martynas Kuliešius IFF-1/9
Jonas Kučinskas IFF-1/7
Simonas Tijušas IFF-1/8
Arminas Misevičius IFF-1/0
Projekto autoriai

Doc. prak. Evaldas Guogis
Doc. prak. Kęstutis Valinčius
Vadovai

Kaunas, 2024

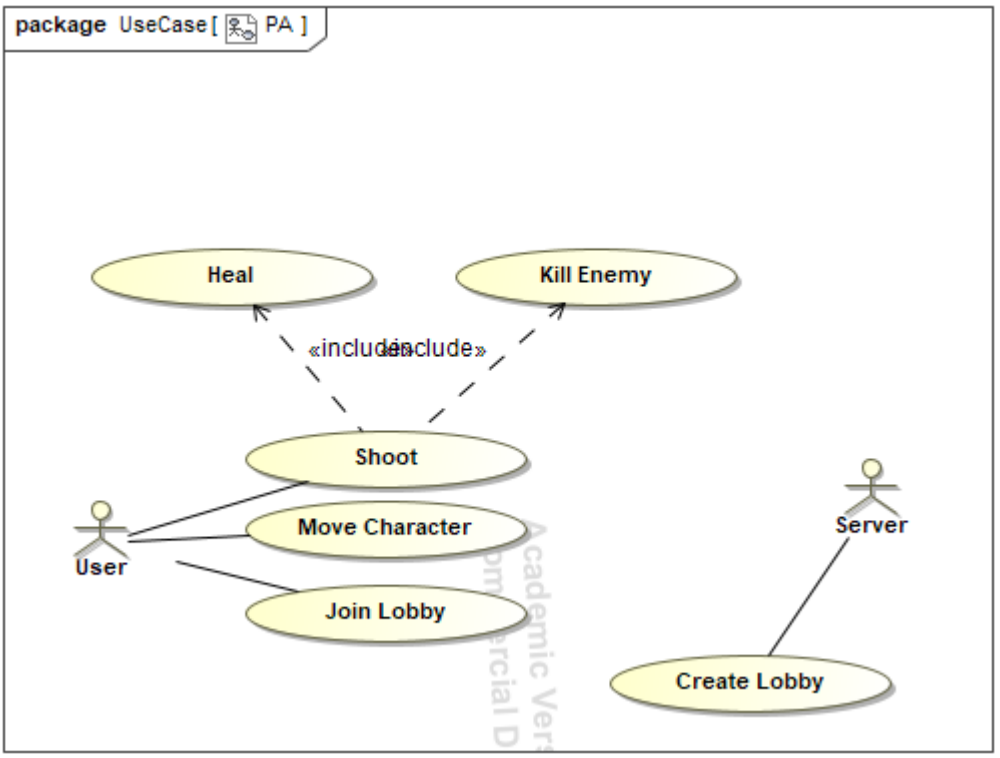
Turinys

Įvadas

Šis dokumentas aprašo programavimo šablonų pritaikymą 2D daugelio žaidėjų žaidimui „killThemAll”. Šio žaidimo pagrindinis tikslas - šaudant priešininkus ir šaudant į burbuliukus išgyventi ir įveikti savo priešą. Žaidimas naudoja gydymo burbuliukus, kurie atsiranda atsitiktinėse vietose ir gražina šiek tiek gyvybių juos nušovus tam, kad žaidėjai galėtų ilgiau kovoti.

1. „UML Use Case” diagrama

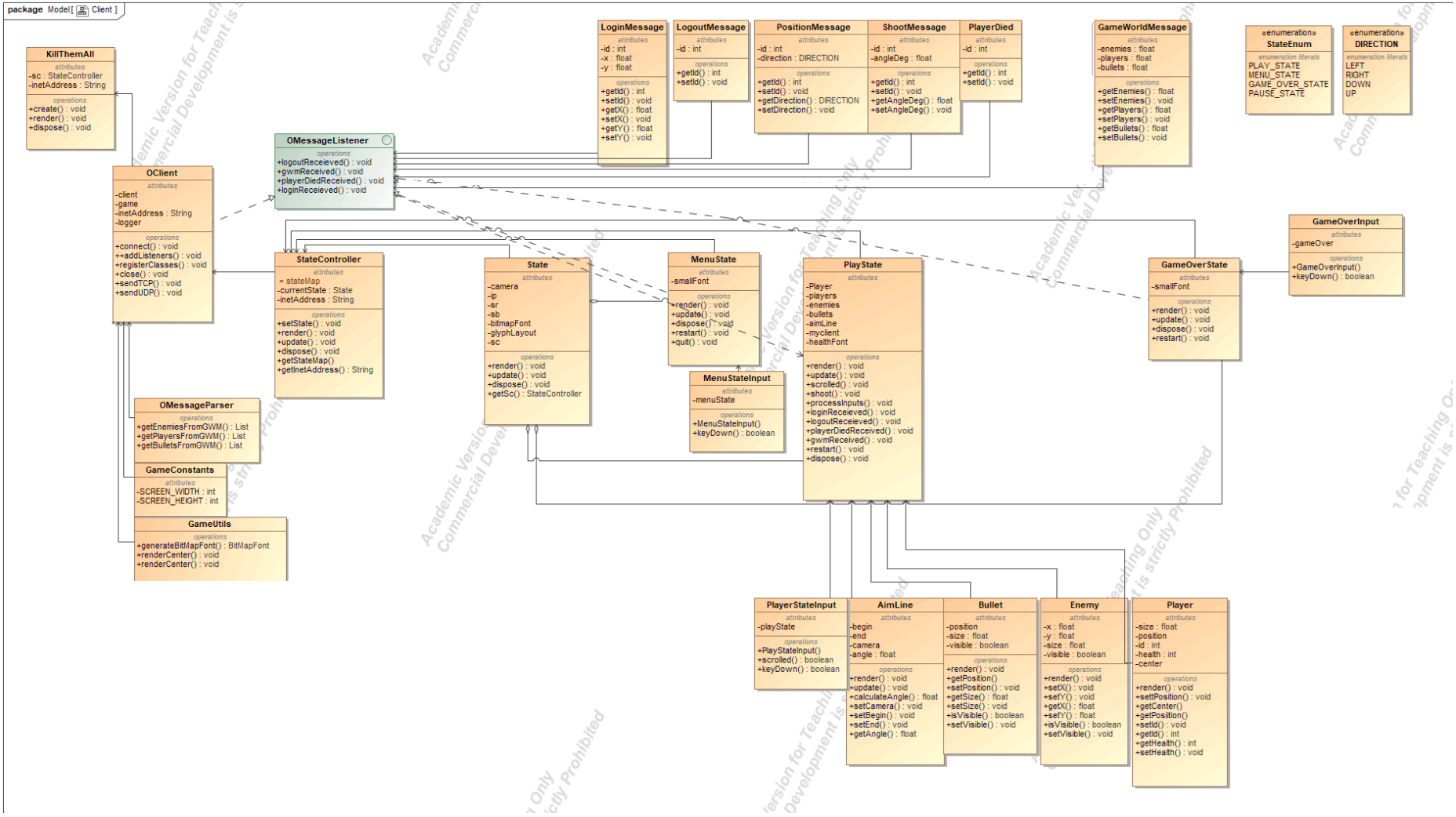
Žaidimo funkcionalumą apima keletas skirtingų panaudojimo atvejų, kurie išdėstyti žaidimo panaudojimo atvejų diagramoje (1 pav.):



1 pav. „killThemAll” panaudojimo atvejų diagrama

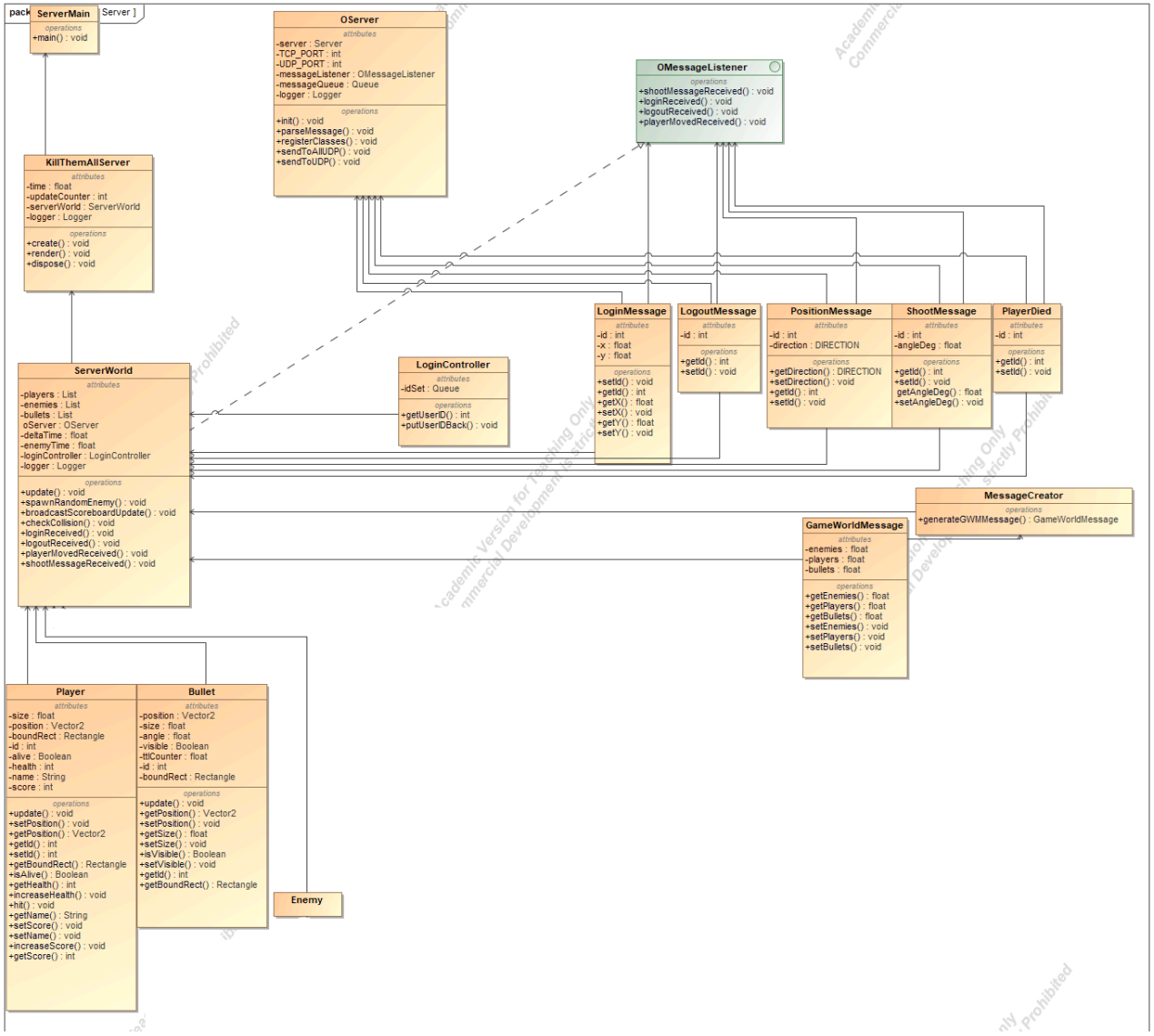
2. UML Klasių diagrama

Pradinę kliento pusės klasių diagramą (2 pav.) sudaro žaidimo komponentai, atsakingi už žaidimo arenos gyvavimą, žaidėjus, kulkas, priešus, būsenas, komunikaciją su serveriu naudojant KryoNet:



2 pav. Kliento pusės Klasių diagrama

Pradinę serverio pusės klasių diagramą (3 pav.) taip pat sudaro žaidimo komponentai, atsakingi už žaidimo arenos gyvavimą, žaidėjus, kulkas, priešus, būsenas, komunikaciją su klientu naudojant KryoNet:



3 pav. Serverio pusės Klasių diagrama

3. Šablonai

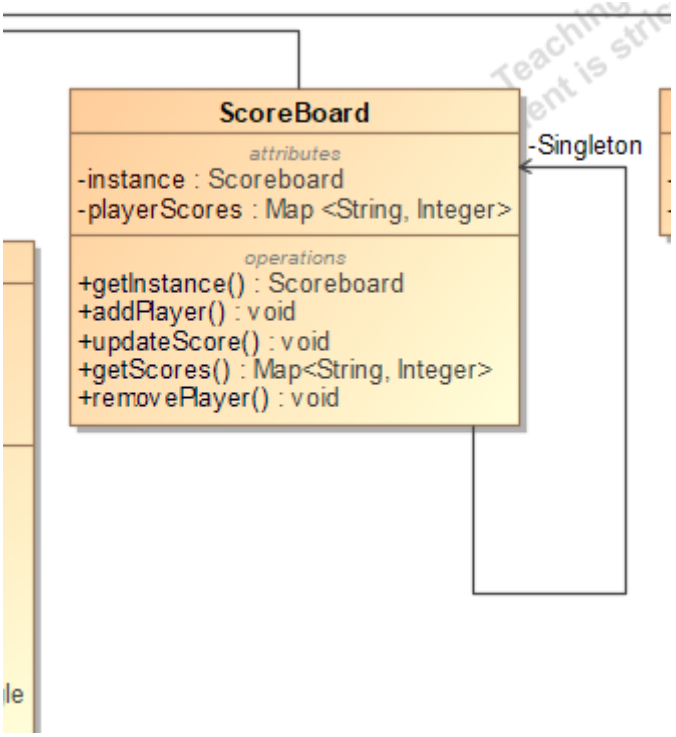
Žaidimo pradinė versija buvo pildoma šablonais naujam funkcionalumui implementuoti. Kiekvieną šabloną pridedant buvo sprendžiamos problemos su suderinamumu tarp skirtingų šablonų bei pradinio kodo. Ši klasė turi metodą „getInstance" užtikrinantį, kad būtų sukuriamas vienas ir tas pats instance, kai reikia naudoti metodą. Visoje programoje naudojamas metodas norint pasinaudoti kitais klasės metodais.

3.1. Singleton

Aprašymas

Singleton yra dizaino šablonas užtikrinantis, kad programoje visoje programoje būtų tik vienas egzempliorius tam tikros klasės. Šiuo atveju, Singleton šabloną taikau kurdamas žaidimui rezultatų lentelę („Scoreboard").

UML klasių diagramos



Pav. 4. Singleton UML šablonas

Kodas

```
package com.javakaian.shooter.shapes;

import com.badlogic.gdx.graphics.g2d.BitmapFont;
import com.badlogic.gdx.graphics.g2d.SpriteBatch;
import java.util.HashMap;
```

```java
import java.util.List;
import java.util.Map;

// DEPRECATED CLASS

/// Singleton Klase Scoreboard kuri
/// privatus konstruktorius, isorines klases negali sukurti naujo scoreboardo
/// statinis instance kuris bus vienintelis zaidime
/// globalus access pointas gauti scoreboardui
/// tasku pridejimo metodas.
///
public class Scoreboard {
    private static volatile Scoreboard instance;
    private Map<String, Integer> playerScores;

    private BitmapFont font;
    private SpriteBatch batch;

    private Scoreboard() {
        playerScores = new HashMap<>();
        font = new BitmapFont(); // Default font
        batch = new SpriteBatch(); // Used for drawing
    }

    public static Scoreboard getInstance() {
        Scoreboard result = instance;
        if (result == null) {
            synchronized (Scoreboard.class) {
                result = instance;
                if (result == null) {
                    result = instance = new Scoreboard();
                }
            }
        }
        return instance;
    }

    public void updateScore(String playerName, int score) {
        playerScores.put(playerName, playerScores.getOrDefault(playerName, 0) +
score);
    }
    public void updateScores(Map<String, Integer> newScores) {
        this.playerScores = newScores != null ? newScores : new HashMap<>(); //
Fallback to empty map if null
    }
```

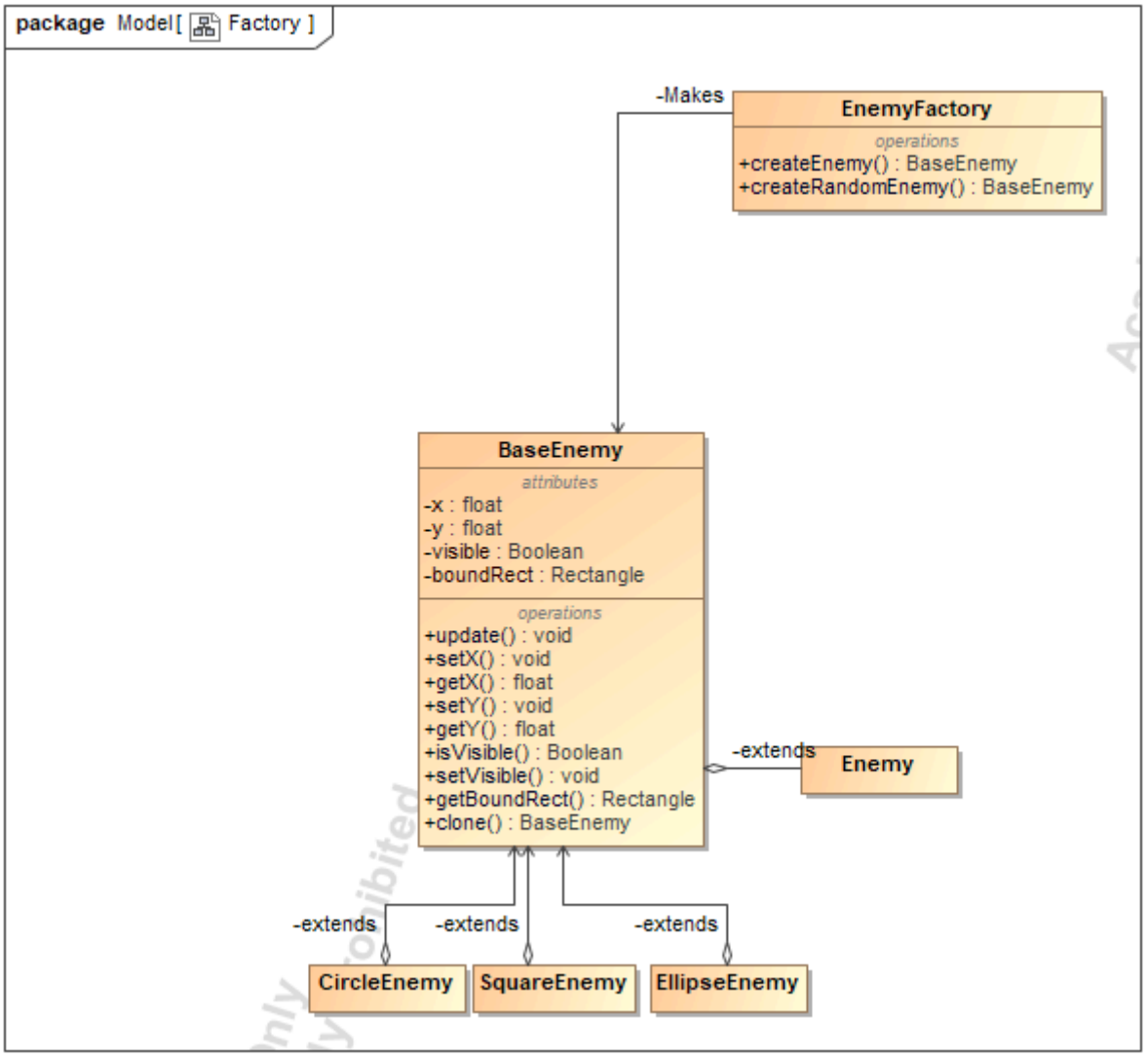
```
    public int getScore(String playerName) {
        return playerScores.getOrDefault(playerName, 0);
    }

    public Map<String, Integer> getPlayerScores() {
        return playerScores;
    }

    // nenaudojamas
    public void render() {
        if (playerScores == null) return; // Safety check, but ideally it should
never be null here

        batch.begin();
        font.draw(batch, "-----Scoreboard-----", 850, 980);
        int yOffset = 960;


        for (Map.Entry<String, Integer> entry : playerScores.entrySet()) {
            font.draw(batch, entry.getKey() + ": " + entry.getValue(), 850,
yOffset);
            yOffset -= 20;
        }

        batch.end();
    }

    // Call this method when disposing of resources, e.g., at the end of the game
    public void dispose() {
        font.dispose();
        batch.dispose();
    }

    public void removePlayer(int playerId) {
        String playerName = "Player_" + playerId;
        playerScores.remove(playerName);
        render();
    }
}
```

3.2. Factory

Aprašymas

EnemyFactory yra „Factory” šablono klasė, kuri sukuria skirtingus žaidimo priešo objektus pagal nurodytą tipą. Priklausomai nuo „Square”/„Ellipse”/„Circle”, EnemyFactory sukuria atitinkamą priešą, paveldintį bazinės „BaseEnemy” klasės savybes. Šis metodas leidžia lengvai pridėti naujų priešo tipų į žaidimą.

UML klasių diagramos



Kodas

EnemyFactory.java:

```
package com.javakaian.util;

import com.javakaian.shooter.shapes.BaseEnemy;
import com.javakaian.shooter.shapes.*;
```

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class EnemyFactory {

    private static final List<String> ENEMY_TYPES = new ArrayList<>();

    static {
        ENEMY_TYPES.add("Ellipse");
        ENEMY_TYPES.add("Circle");
        ENEMY_TYPES.add("Square");
    }

    public static BaseEnemy createEnemy(String type, float x, float y) {
        switch(type){
            case "Ellipse":
                return new EllipseEnemy(x,y);
            case "Circle":
                return new CircleEnemy(x,y);
            case "Square":
                return new SquareEnemy(x,y);
            default:
                //return new SquareEnemy(x,y);
                throw new IllegalArgumentException("Unknown enemy type: " +
type);
        }
    }

    // Method to create a random enemy
    public static BaseEnemy createRandomEnemy(float x, float y) {
        String randomType = ENEMY_TYPES.get(new
Random().nextInt(ENEMY_TYPES.size()));
        return createEnemy(randomType, x, y);
    }
}
```

BaseEnemy.java:

```java
package com.javakaian.shooter.shapes;

import com.badlogic.gdx.math.Rectangle;

public abstract class BaseEnemy implements Cloneable {

    protected float x;
    protected float y;
```

```
   protected int health;

   protected boolean visible = true;
   protected String shape;
   protected Rectangle boundRect;

   public BaseEnemy(){}
   public BaseEnemy(float x, float y, float size) {
      this.x = x;
      this.y = y;
      //this.boundRect = new Rectangle(x, y, size, size);
   }
   public BaseEnemy(float x, float y,  String shape, int health, int sizeX, int
sizeY) {
      this.x = x;
      this.y = y;
      this.shape = shape;
      this.health = health;
      this.boundRect = new Rectangle(x, y, sizeX, sizeY);
   }

   public Rectangle getBoundRect() {
      return boundRect;
   }

   public void update(float deltaTime) {

      this.boundRect.x = x;
      this.boundRect.y = y;
   }

   public void setX(float x) {
      this.x = x;
   }

   public void setY(float y) {
      this.y = y;
   }

   public float getX() {
      return x;
   }

   public float getY() {
      return y;
   }
```

```
    public int getHealth() {
        return health;
    }

    public void setHealth(int health) {
        this.health = health;
    }

    public boolean isVisible() {
        return visible;
    }

    public String getShape() {
        return shape;
    }
    public void setVisible(boolean visible) {
        this.visible = visible;
    }

    //public Rectangle getBoundRect() {
    // return boundRect;
    //}

    @Override
    public BaseEnemy clone() {
        try {
            return (BaseEnemy) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }

    public abstract void takeDamage(int damage);
    public abstract void applyEffect(Player player);
}
```

CircleEnemy.java:

```
package com.javakaian.shooter.shapes;

import com.badlogic.gdx.math.Circle;

public class CircleEnemy extends BaseEnemy {
    protected Circle circle;
```

```
    public CircleEnemy(){}
    public CircleEnemy(float x, float y) {
        super(x,y,"Circle", 1, 20, 20);
        this.circle = new Circle(x,y,10);
    }

    @Override
    public void takeDamage(int damage) {
        this.health -= damage;
    }

    @Override
    public void applyEffect(Player player) {
        player.increaseScore(player.getScore() + 2*health);

    }
}
```

SquareEnemy.java:

```
package com.javakaian.shooter.shapes;

import com.badlogic.gdx.math.Rectangle;

public class SquareEnemy extends BaseEnemy{

    protected Rectangle rectangle;

    public SquareEnemy(){}
    public SquareEnemy(float x, float y) {
        super(x,y, "Square", 1, 20, 20);
        this.rectangle = new Rectangle(x,y,10,10);
    }
    @Override
    public void takeDamage(int damage) {
        this.health -= damage;
    }

    @Override
    public void applyEffect(Player player) {
        player.increaseHealth(20);
    }
}
```

EllipseEnemy.java:

```
package com.javakaian.shooter.shapes;
```

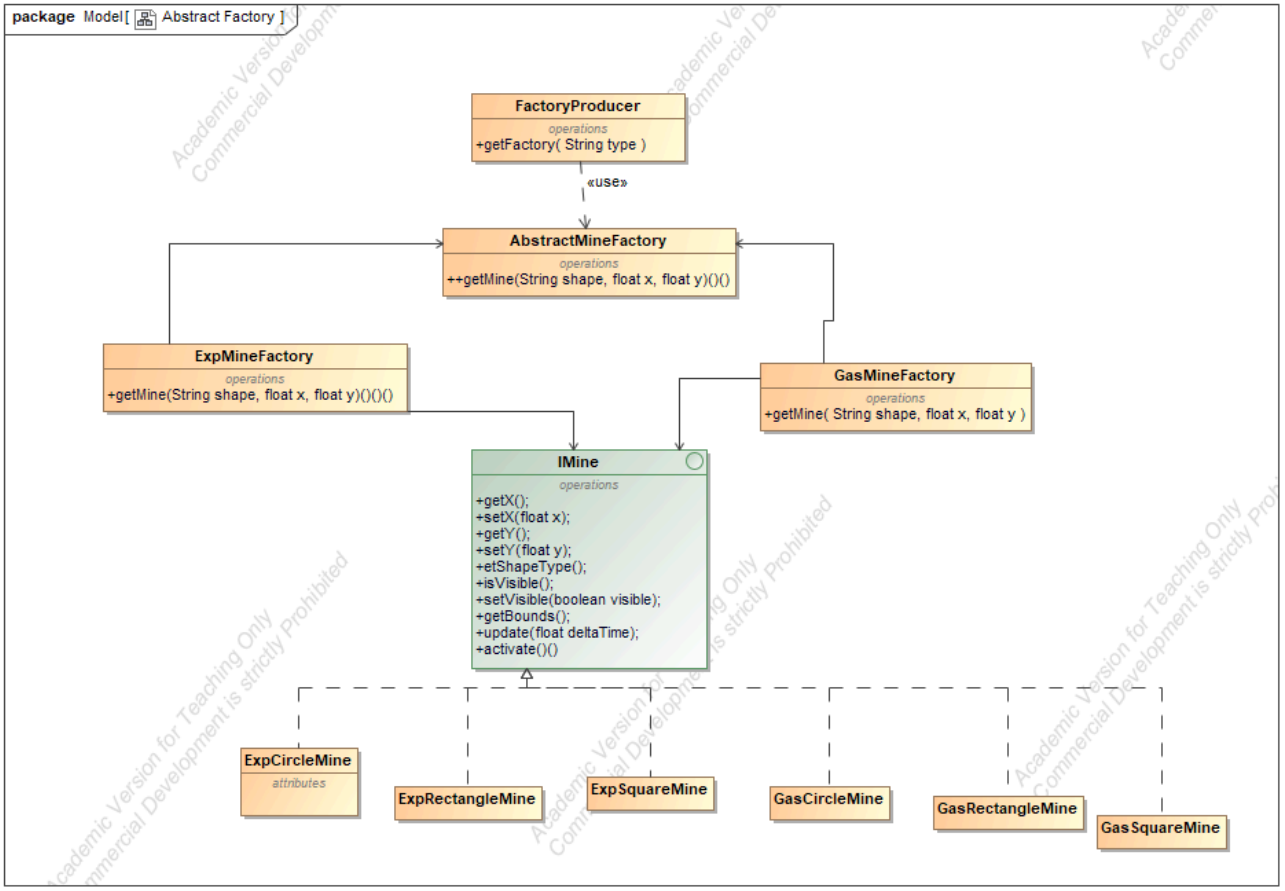
```java
import com.badlogic.gdx.math.Ellipse;

public class EllipseEnemy extends BaseEnemy{

    protected Ellipse ellipse;

    public EllipseEnemy(){}

    public EllipseEnemy(float x, float y){
        super(x,y,"Ellipse", 2, 30,30);
        this.ellipse = new Ellipse(x,y, 30,20);
    }


    @Override
    public void takeDamage(int damage) {
        this.health -= damage*2;
    }

    @Override
    public void applyEffect(Player player){
        //sitas enemy tsg padaro
    }
}
```

3.3. Abstract Factory

Aprašymas

Šis dizainas leidžia lengvai pridėti naujus minų tipus be pagrindinės gamyklos struktūros modifikavimo, laikantis Atvirumo/Uždarymo principo. Be to, jis inkapsuliuoja skirtingų minų tipų kūrimo logiką, atitinka Abstrakčiosios gamyklos šablono tikslą – pateikti sąsają, kuri sukuria susijusius arba priklausomus objektus, nenurodant konkrečių jų klasių.

UML klasių diagramos



Kodas

```java
package com.javakaian.shooter.shapes.mine;

public abstract class AbstractMineFactory {
    public abstract IMine getMine(String shape, float x, float y);
}
```

```java
public class GasCircleMine implements IMine {
    private float x, y;
    private boolean visible;
    private String shapeType = "GasCircle";
    private Rectangle bounds;

    public GasCircleMine(float x, float y) {
        this.x = x;
        this.y = y;
        this.visible = true;
        this.bounds = new Rectangle(x, y, 20, 20); // Adjust size as needed
    }

    public float getX() { return x; }
    public void setX(float x) { this.x = x; }

    public float getY() { return y; }
    public void setY(float y) { this.y = y; }

    public String getShapeType() { return shapeType; }

    public boolean isVisible() { return visible; }
    public void setVisible(boolean visible) { this.visible = visible; }

    public Rectangle getBounds() { return bounds; }

    public void update(float deltaTime) {
```

```java
public class GasCircleMine implements IMine {
    private float x, y;
    private boolean visible;
    private String shapeType = "GasCircle";
    private Rectangle bounds;

    public GasCircleMine(float x, float y) {
        this.x = x;
        this.y = y;
        this.visible = true;
        this.bounds = new Rectangle(x, y, 20, 20); // Adjust size as needed
    }

    public float getX() { return x; }
    public void setX(float x) { this.x = x; }

    public float getY() { return y; }
    public void setY(float y) { this.y = y; }

    public String getShapeType() { return shapeType; }

    public boolean isVisible() { return visible; }
    public void setVisible(boolean visible) { this.visible = visible; }

    public Rectangle getBounds() { return bounds; }

    public void update(float deltaTime) {
```

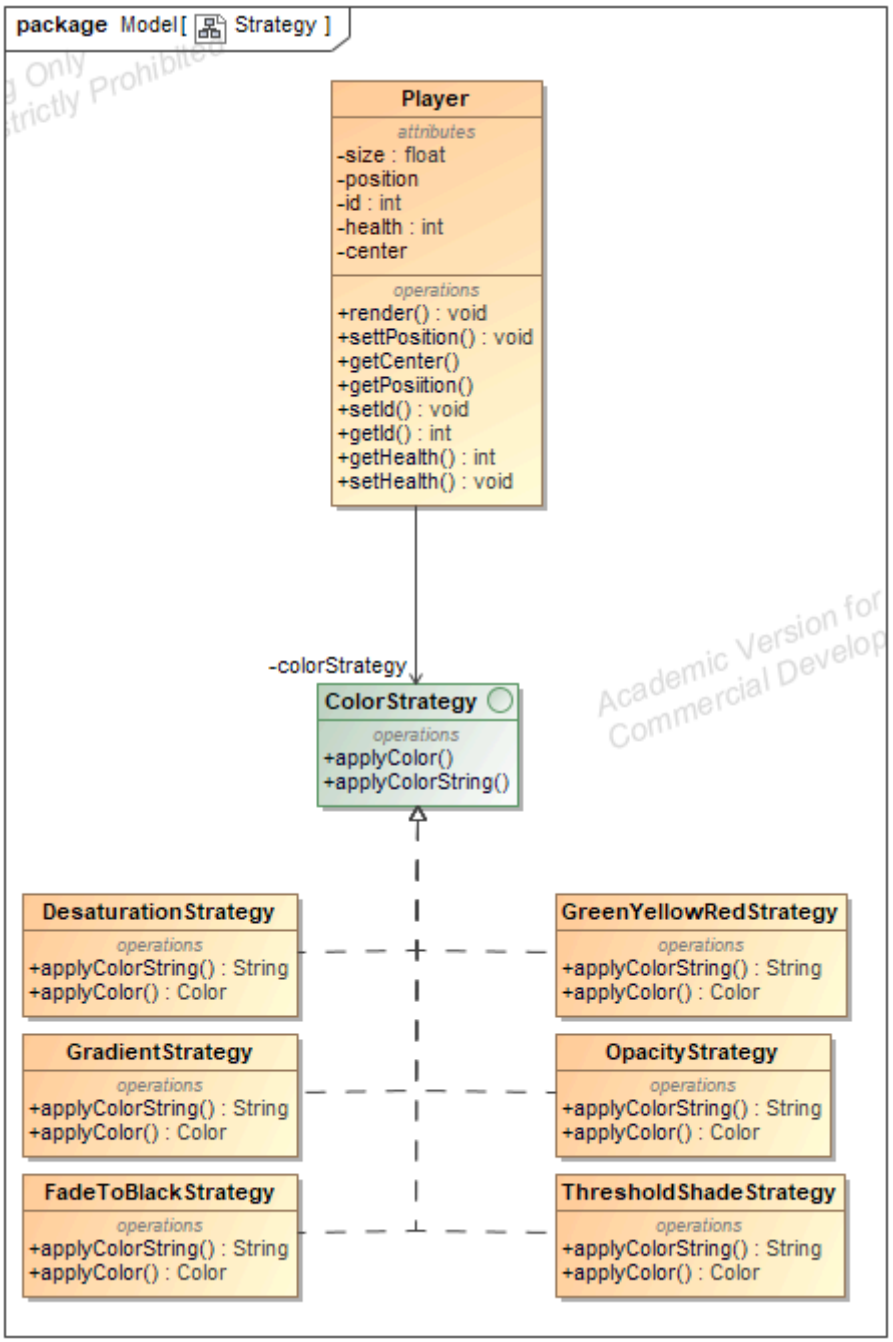
```java
package com.javakaian.shooter.shapes.mine;

public class ExpMineFactory extends AbstractMineFactory {

    @Override
    public IMine getMine(String shape, float x, float y) {
        switch (shape.toLowerCase()) {
            case "circle":
                return new ExpCircleMine(x, y);
            case "rectangle":
                return new ExpRectangleMine(x, y);
            case "square":
                return new ExpSquareMine(x, y);
            default:
                throw new IllegalArgumentException("Unknown mine shape: " + shape);
        }
    }
}
```

3.4. Strategy

Aprašymas

Šį šabloną taikome norint pakeisti žaidėjo atvaizdavimą žaidime. Pagal tai, kiek gyvybių taškų „health” turi žaidėjas, skirtingas strategy pakeičia žaidėjo valdomo kvadrato spalvas.

UML klasių diagrama



Kodas

ColorStrategy. java:

```
package com.javakaian.shooter.Strategy;
import com.badlogic.gdx.graphics.Color;

/// Galima pakeisti strategy PlayState failo Login metode.
/// new GradienStrategy()
/// new BlinkingRedStrategy()
/// new GreenYellowRedStrategy()
/// new OpacityStrategy()
/// new ThresholdShadeStrategy()
/// new FadeToBlackStrategy()
public interface ColorStrategy {
   Color applyColor(int Health);
   String applyColorString(int health); // grazina spalva pagal hp
}
```

DesaturationStrategy.java:

```
package com.javakaian.shooter.Strategy;
import com.badlogic.gdx.graphics.Color;

public class DesaturationStrategy implements ColorStrategy {
   @Override
   public String applyColorString(int health) {
       return "RED";
   }

   @Override
   public Color applyColor(int health) {
       float saturation = health / 100.0f;
       return new Color(1 * saturation, 0 * saturation, 0 * saturation + (1 -
saturation), 1);
   }
}
```

GradientStrategy.java:

```
package com.javakaian.shooter.Strategy;
import com.badlogic.gdx.graphics.Color;
public class GradientStrategy implements ColorStrategy {
   @Override
   public String applyColorString(int health) {
       int red = (int) (255 * (1 - (health / 100.0)));
       int green = (int) (255 * (health / 100.0));
       return String.format("#%02X%02X00", red, green); // Returns a hex color
code
   }

   @Override
   public Color applyColor(int health) {
       float greenValue = health / 100.0f;
       float redValue = 1.0f - greenValue;
       return new Color(redValue, greenValue, 0, 1); // Creates a gradient from
green to red
   }
}
```

GreenYellowRedStrategy.java:

```
package com.javakaian.shooter.Strategy;
import com.badlogic.gdx.graphics.Color;
public class GreenYellowRedStrategy implements ColorStrategy {
    @Override
    public String applyColorString(int health){
        if (health > 70) {return "Green";}
        else if (health > 30) { return "Yellow";}
        else { return "Red";}
    }

    @Override
    public Color applyColor(int health) {
        if(health>100){
            return Color.PURPLE;
        }
        else if (health > 70 && health <= 100) {
            return Color.GREEN;
        } else if (health > 30 && health <= 70) {
            return Color.YELLOW;
        } else {
            return Color.RED;
        }
    }
}
```

OpacityStrategy.java:

```
package com.javakaian.shooter.Strategy;
import com.badlogic.gdx.graphics.Color;
public class OpacityStrategy implements ColorStrategy{
    @Override
    public String applyColorString(int health) {
        int alpha = (int) (255 * (health / 100.0)); // Calculate alpha based on
health
        return String.format("rgba(255, 0, 0, %d)", alpha); // Red with varying
opacity
    }
    @Override
    public Color applyColor(int health) {
        float alpha = health / 100.0f; // Alpha decreases as health decreases
        return new Color(1, 0, 0, alpha); // Red with varying opacity
    }
}
```

ThresholdShadeStrategy.java:

```
package com.javakaian.shooter.Strategy;
import com.badlogic.gdx.graphics.Color;

public class ThresholdShadeStrategy implements ColorStrategy {
    @Override
    public String applyColorString(int health){
        return "RED";
    }
    @Override
    public Color applyColor(int health) {
        if (health > 70) {
            return new Color(1, 0.6f, 0.6f, 1); // Light red
        } else if (health > 40) {
            return new Color(1, 0.3f, 0.3f, 1); // Medium red
```

```
        } else {
            return Color.RED; // Deep red for critical health
        }
    }
}
```

FadeToBlackStrategy.java:

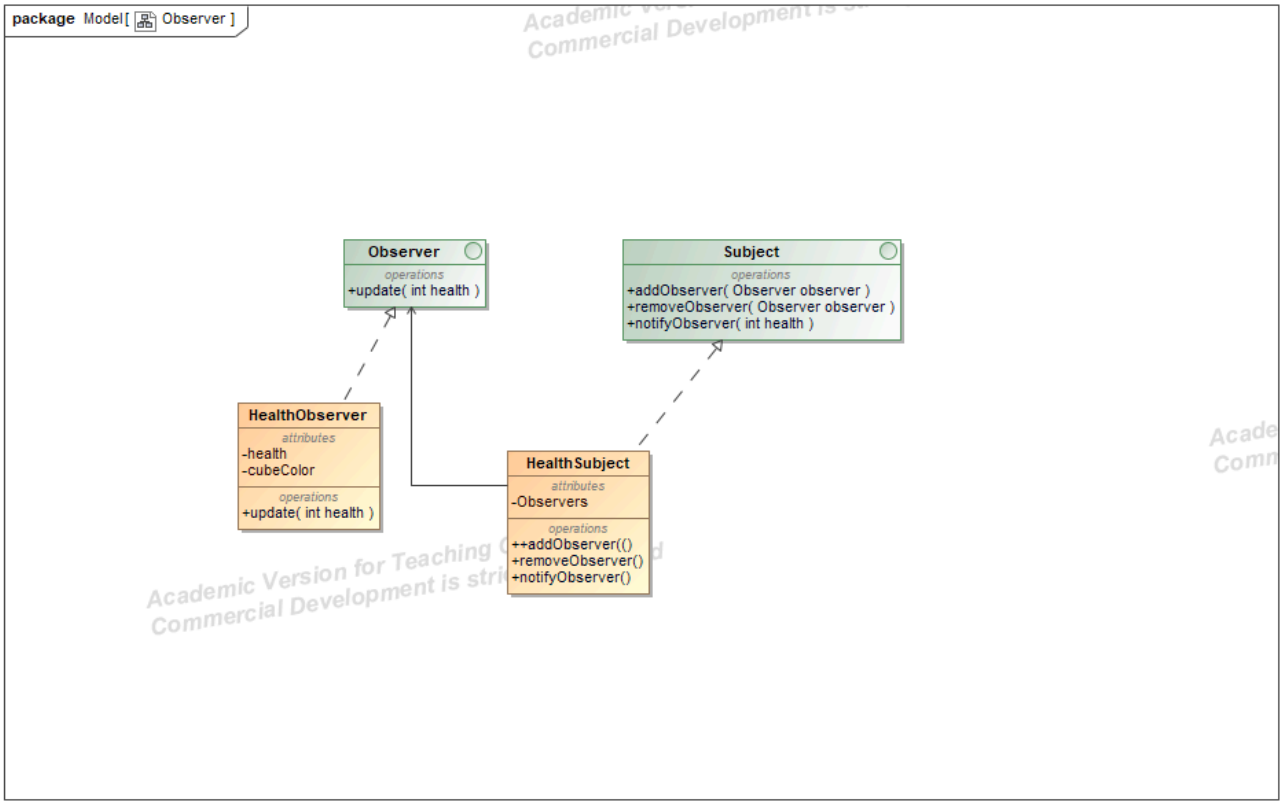
```
package com.javakaian.shooter.Strategy;
import com.badlogic.gdx.graphics.Color;
public class FadeToBlackStrategy implements ColorStrategy {
    @Override
    public String applyColorString(int health){
        if(health>50){
            return "RED";
        }
        else{
            return "BLACK";
        }
    }
    @Override
    public Color applyColor(int health) {
        float healthRatio = health / 100.0f;
        Color color = new Color(Color.RED); // Start with red color
        color.lerp(0, 0, 0, 1, 1 - healthRatio); // Lerp to black (0, 0, 0, 1)
        return color;
    }
}
```

3.5. Observer

Aprašymas

Šis dizaino šablonas leidžia HealthSubject objekto sveikatos būklei pasikeitus informuoti visus prijungtus stebėtojus HealthObserver, kad jie galėtų atitinkamai reaguoti. Tai užtikrina gerą programos struktūros palaikymą ir leidžia lengvai pridėti naujus stebėtojus be reikšmingų pagrindinio kodo pakeitimų.

UML klasių diagramos



Kodas

```java
public interface Observer {
    void update(int health);
}
```

```java
public interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(int health);
}
```

```java
public class HealthSubject implements Subject {
    private List<Observer> observers = new ArrayList<>();

    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers(int health) {
        for (Observer observer : observers) {
            observer.update(health);
        }
    }
}
```

```java
public class HealthObserver implements Observer {
    private int health;
    private Color cubeColor;

    public HealthObserver() {
        this.cubeColor = Color.WHITE; // Default color
    }

    @Override
    public void update(int health) {
        this.health = health;
        updateCubeColor();
    }

    private void updateCubeColor() {
        if (health > 75) {
            cubeColor = Color.GREEN;
```

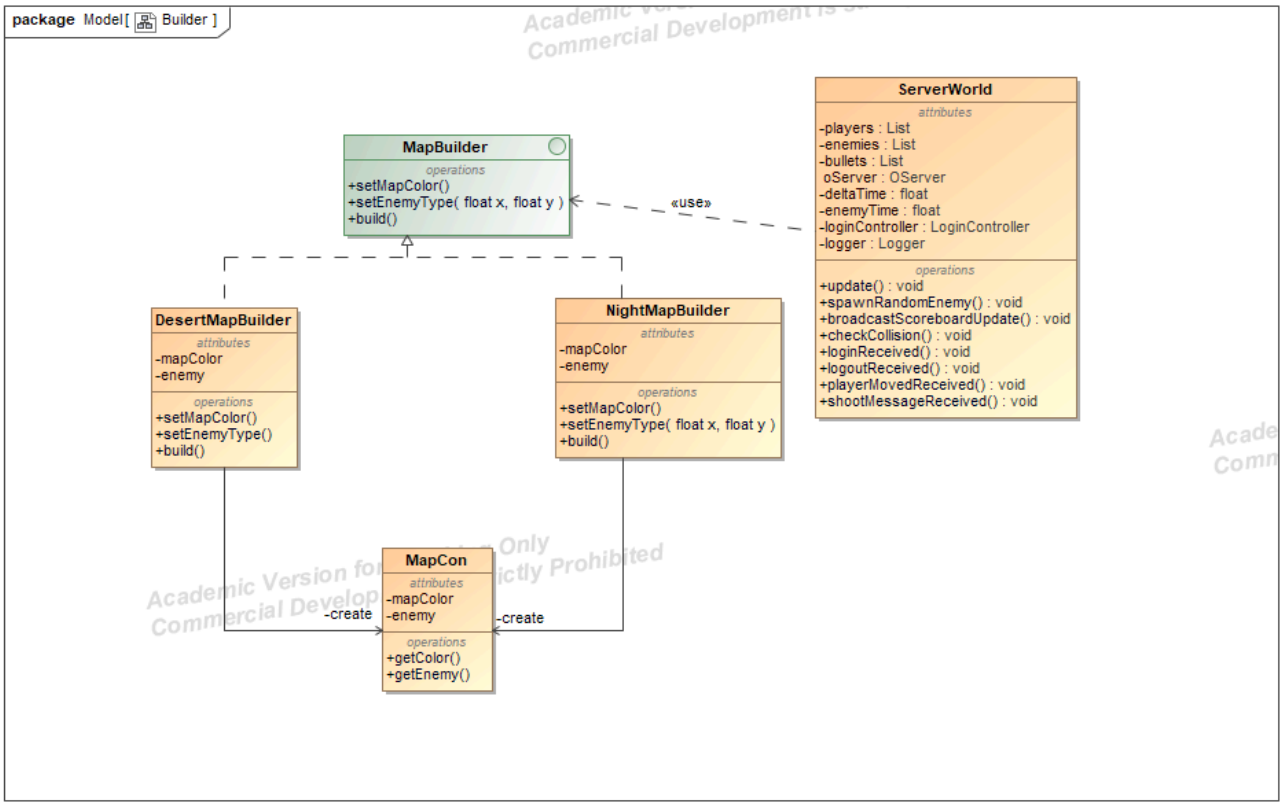
```
    } else if (health > 50) {
        cubeColor = Color.YELLOW;
    } else if (health > 25) {
        cubeColor = Color.ORANGE;
    } else {
        cubeColor = Color.RED;
    }
}

public Color getCubeColor() {
    return cubeColor;
}
```

3.6. Builder

Aprašymas

Šis šablonas leidžia lanksčiai ir aiškiai kurti sudėtingus objektus, tokius kaip žemėlapiai su skirtingais aplinkos parametrais, nepriklausomai nuo ServerWorld klasės vidinės struktūros. Konstruktoriaus šablonas padeda išvengti sudėtingų konstruktorių ir leidžia pritaikyti įvairias variacijas

UML klasių diagramos



Kodas

```
public interface MapBuilder {
    MapBuilder setMapColor();
    MapBuilder setEnemyType(float x, float y);;
    MapCon build();
}
```

```java
public class DesertMapBuilder implements MapBuilder {
    private int mapColor;
    private BaseEnemy enemy;

    @Override
    public MapBuilder setMapColor() {
        this.mapColor = 1;
        return this;
    }

    @Override
    public MapBuilder setEnemyType(float x, float y) {
        this.enemy = EnemyFactory.createRandomEnemy("Circle",x,y);
        return this;
    }
    @Override
    public MapCon build() {
        return new MapCon(mapColor, enemy);
    }
}
```

```java
package com.javakaian.util;

import com.javakaian.shooter.shapes.BaseEnemy;
import com.javakaian.shooter.shapes.*;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class MapCon {
    private final int mapColor;
    private final BaseEnemy enemy;

    public MapCon(int mapColor, BaseEnemy enemy) {
        this.mapColor = mapColor;
        this.enemy = enemy;
    }

    public int getMapColor() { return mapColor; }
    public BaseEnemy getEnemy() { return enemy; }
}
```

```java
public class NightMapBuilder implements MapBuilder {
    private int mapColor;
    private BaseEnemy enemy;

    @Override
    public MapBuilder setMapColor() {
        this.mapColor = 2;
        return this;
    }

    @Override
    public MapBuilder setEnemyType(float x, float y) {
        this.enemy = EnemyFactory.createRandomEnemy("Square",x,y);
        return this;
    }
    @Override
    public MapCon build() {
        return new MapCon(mapColor, enemy);
    }
}
```

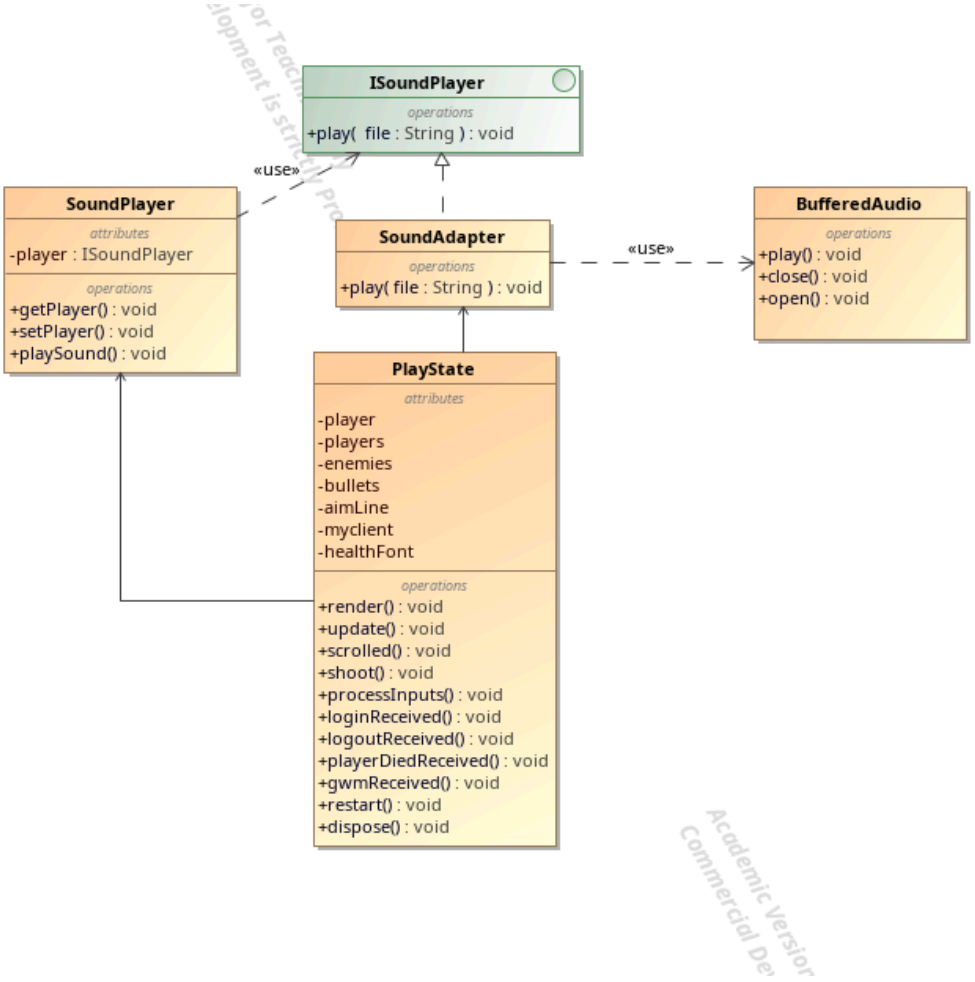
```java
private void spawnRandomEnemy() {

    if (enemyTime >= 0.4 && enemies.size() <= 30) {
        enemyTime = 0;
        if (enemies.size() % 5 == 0)
            logger.debug("Number of enemies : " + enemies.size());

        float x = new SecureRandom().nextInt(1000);
        float y = new SecureRandom().nextInt(1000);
        //BaseEnemy e = new Enemy(new SecureRandom().nextInt(1000), new SecureRandom().nextInt(1000), 10);
        config = map.setMapColor().setEnemyType(x,y).build();
        BaseEnemy randomEnemy = config.getEnemy();
        MapColor = config.getMapColor();
```

3.7. Adapter

Aprašymas

Šioje implementacijoje “SoundAdapter” klasė pritaiko išorinės bibliotekos klasę “BufferedAudio” naudojimui “PlayState”, vykstant žaidimui. Taip pat naudojama pagalbinė “SoundPlayer” klasė, per kurią daromi kvietimai į metodą playSound(). Ši klasė pasitarnaus, jei nuspręsime naudoti kelias garso bibliotekas, kurioms reikės skirtingų Adapter implementacijų.

UML klasių diagrama



Kodas

SoundPlayer.java:

```
public class SoundPlayer {

   private ISoundPlayer player;

   public ISoundPlayer getPlayer(){
       return player;
   }

   public void setPlayer(ISoundPlayer player){
       this.player = player;
   }

   public void playSound(String file){

       if (player == null){
           return;
       }

       player.play(file);
   }
}
```

SoundAdapter.java:

```
public class SoundAdapter implements ISoundPlayer{

   public void play(String file) {
       new Thread(new Runnable() {
           @Override
           public void run() {
               try {
                   BufferedAudio audio = new BufferedAudio(file);
                   audio.open();
                   audio.play();

                   Thread.sleep(audio.getLength());

                   audio.close();
                   audio = null;
               } catch (AudioException | InterruptedException e) {
                   e.printStackTrace();
               }
           }
       }).start();
```

```
    }
}
```

ISoundPlayer.java:

```
public interface ISoundPlayer {
    void play(String file);
}
```

PlayState.java:

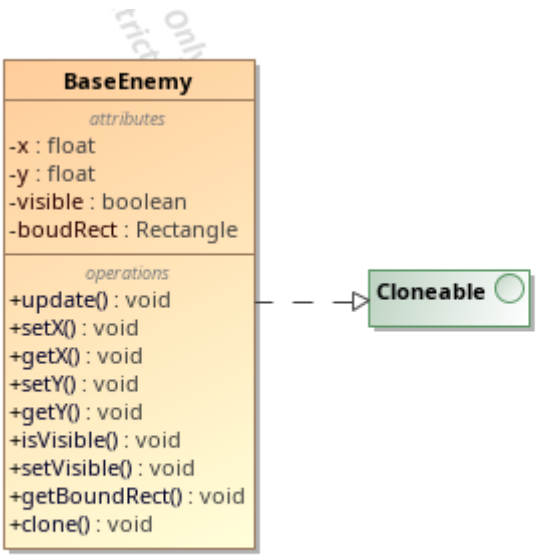
```
private void init() {

        ISoundPlayer soundAdapter = new SoundAdapter();
        soundPlayer = new SoundPlayer();
        soundPlayer.setPlayer(soundAdapter);
}
public void shoot() {

        ShootMessage m = new ShootMessage();
        m.setId(player.getId());
        m.setAngleDeg(aimLine.getAngle());
        myclient.sendUDP(m);
        soundPlayer.playSound("client/assets/sounds/bulletShoot.wav");
   }
```

3.8. Prototype

Aprašymas

Prototype šablonas naudojamas priešų “deep” kopijavimui BaseEnemy klasėje.
Objektų atminties adresai naudojant `System.identityHashCode(enemy)`:
Originalaus objekto: 535483854
Kopijuoto objekto: 992536224

UML klasių diagramos



Kodas

```
public abstract class BaseEnemy implements Cloneable {

    protected float x;
    protected float y;
    protected int health;

    protected boolean visible = true;
    protected String shape;
    protected Rectangle boundRect;

    public BaseEnemy(){}
    public BaseEnemy(float x, float y, float size) {
        this.x = x;
        this.y = y;
        //this.boundRect = new Rectangle(x, y, size, size);
    }
    public BaseEnemy(float x, float y,  String shape, int health, int
sizeX, int sizeY) {
        this.x = x;
        this.y = y;
        this.shape = shape;
        this.health = health;
        this.boundRect = new Rectangle(x, y, sizeX, sizeY);
    }
```

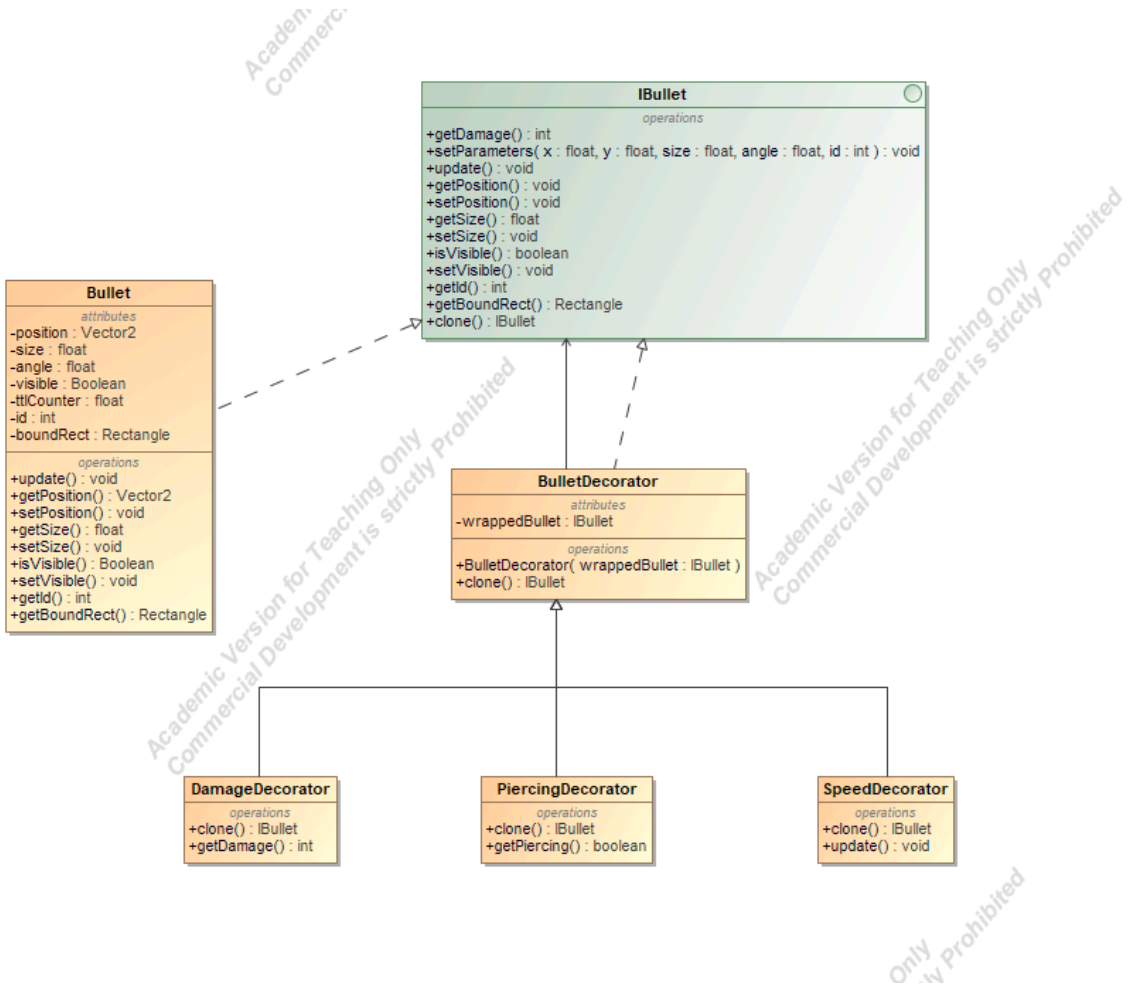
```
    @Override
    public BaseEnemy clone() {
        try {
            BaseEnemy clonedEnemy = (BaseEnemy) super.clone();

            if (this.shape != null) {
                clonedEnemy.shape = new String(this.shape);
            }

            if (this.boundRect != null) {
                clonedEnemy.boundRect = new Rectangle(this.boundRect);
            }

            return clonedEnemy;

        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

3.9. Decorator

Aprašymas

Šioje implementacijoje ,,BulletDecorator” klasė, kuri įgyvendina bendrą sąsają su ,,IBullet” turi nuorodą į ,,wrappedBullet” objektą. Yra trys dekoratoriaus lygiai: ,,DamageDecorator”, ,,PiercingDecorator” ir ,,SpeedDecorator”. ,,DamageDecorator” prideda papildomą žalą kulkai, ,,SpeedDecorator” keičia kulkos greitį, o ,,PiercingDecorator" keičia kulkos pramušimo savybę.

UML klasių diagramos



Kodas

`BulletDecorator.java:`

```java
public abstract class BulletDecorator implements IBullet {

    15 usages
    protected IBullet wrappedBullet;

    yousimas
    public BulletDecorator(IBullet wrappedBullet) { this.wrappedBullet = wrappedBullet; }

    yousimas
    @Override
    public int getDamage() { return wrappedBullet.getDamage(); }

    yousimas
    @Override
    public void setParameters(float x, float y, float size, float angle, int id) {
        wrappedBullet.setParameters(x, y, size, angle, id);
    }
    yousimas
    @Override
    public void update(float deltaTime) { wrappedBullet.update(deltaTime); }

    yousimas
    @Override
    public Vector2 getPosition() { return wrappedBullet.getPosition(); }

    yousimas
    @Override
    public void setPosition(Vector2 position) { wrappedBullet.setPosition(position); }

    yousimas
    @Override
    public float getSize() { return wrappedBullet.getSize(); }

    yousimas
    @Override
    public void setSize(float size) { wrappedBullet.setSize(size); }
```

```
    yousimas
    @Override
    public boolean isVisible() { return wrappedBullet.isVisible(); }

    yousimas
    @Override
    public void setVisible(boolean visible) { wrappedBullet.setVisible(visible); }

    yousimas
    @Override
    public void setPiercing(boolean piercing){wrappedBullet.setPiercing(piercing);}

    yousimas
    @Override
    public boolean getPiercing(){return wrappedBullet.getPiercing();}

    yousimas
    @Override
    public int getId() { return wrappedBullet.getId(); }

    yousimas
    @Override
    public Rectangle getBoundRect() { return wrappedBullet.getBoundRect(); }
    yousimas
    @Override
    public float getAngle() { return wrappedBullet.getAngle(); }
    yousimas
    @Override
    public abstract IBullet clone();
}
```

DamageDecorator.java:

```
public class DamageDecorator extends BulletDecorator {

    3 usages
    private int additionalDamage;

    1 usage  yousimas
    public DamageDecorator(IBullet wrappedBullet, int additionalDamage) {
        super(wrappedBullet);
        this.additionalDamage = additionalDamage;
    }

    yousimas
    @Override
    public int getDamage() { return  additionalDamage; // Increase damage }

    yousimas
    @Override
    public IBullet clone() { return new DamageDecorator(wrappedBullet.clone(), additionalDamage); }
}
```

PiercingDecorator.java:

```java
public class PiercingDecorator extends BulletDecorator {

    3 usages
    private boolean piercing;

    1 usage  yousimas
    public PiercingDecorator(IBullet wrappedBullet, boolean piercing) {
        super(wrappedBullet);
        this.piercing = piercing;
    }

    yousimas
    @Override
    public boolean getPiercing() { return  this.piercing; }

    yousimas
    @Override
    public IBullet clone() { return new PiercingDecorator(wrappedBullet.clone(), piercing); }
}
```

SpeedDecorator.java:

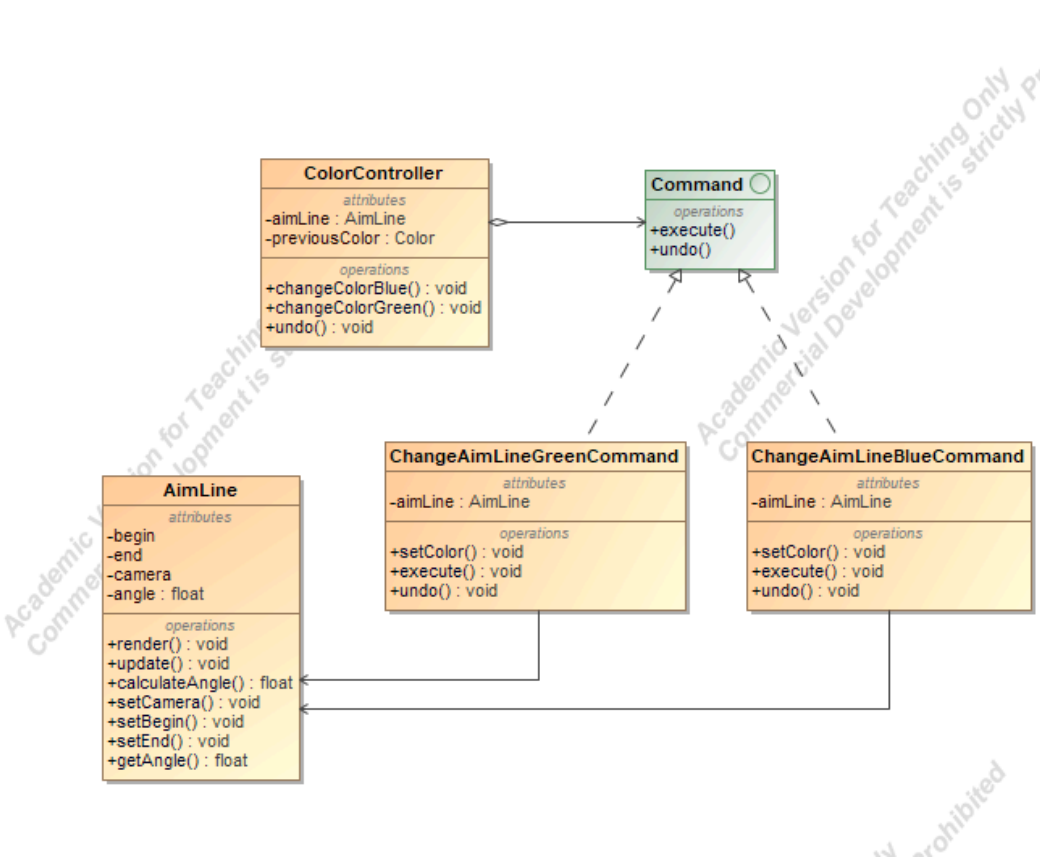
```java
public class SpeedDecorator extends BulletDecorator {

    3 usages
    private float speedMultiplier;

    1 usage  yousimas
    public SpeedDecorator(IBullet wrappedBullet, float speedMultiplier) {
        super(wrappedBullet);
        this.speedMultiplier = speedMultiplier;
    }

    yousimas *
    @Override
    public void update(float deltaTime) {

        super.update(deltaTime);

        Vector2 position = wrappedBullet.getPosition();

        float angle = wrappedBullet.getAngle();

        double originalSpeed = deltaTime * 800;  // Base speed (can be adjusted)
        double adjustedSpeed = originalSpeed * speedMultiplier;

        position.x += Math.cos(angle) * adjustedSpeed;
        position.y -= Math.sin(angle) * adjustedSpeed;

        wrappedBullet.setPosition(position);
    }
    yousimas
    @Override
    public IBullet clone() { return new SpeedDecorator(wrappedBullet.clone(), speedMultiplier); }
}
```

3.10. Command

Aprašymas

Šioje implementacijoje „ColorController” klasė iškviečia komandų metodus, kurie keičia šaudymo linijos spalvas.

UML klasių diagramos



Kodas

ColorController.java:

```java
public class ColorController {

    4 usages
    private AimLine aimLine;
    1 usage
    private Color previousColor;

    yousimas
    public ColorController(AimLine aimLine) {
        this.aimLine = aimLine;
        this.previousColor = aimLine.getColor();
    }

    yousimas
    public void changeColorBlue() {
        Command changeColorCommand = new ChangeAimLineBlueCommand(aimLine);
        changeColorCommand.execute();
    }

    yousimas
    public void changeColorGreen() {
        Command changeColorCommand = new ChangeAimLineGreenCommand(aimLine);
        changeColorCommand.execute();
    }

    yousimas
    public void undo() {
        Command changeColorCommand = new ChangeAimLineBlueCommand(aimLine);
        changeColorCommand.undo();
    }
}
```

Command.java:

```java
public interface Command {
    yousimas
    void execute();
    yousimas
    void undo();
}
```

ChangeAimLineBlueCommand.java:

```java
public class ChangeAimLineBlueCommand implements Command {

    3 usages
    private AimLine aimLine;

    yousimas
    public ChangeAimLineBlueCommand(AimLine aimLine) { this.aimLine = aimLine; }

    yousimas
    public void setColor(Color color) { aimLine.setColor(color); }

    yousimas
    @Override
    public void execute() { setColor(Color.BLUE); }

    yousimas
    public void undo() { aimLine.setColor(Color.RED); }
}
```

ChangeAimLineGreenCommand.java:

```java
public class ChangeAimLineGreenCommand implements Command {

    3 usages
    private AimLine aimLine;

    yousimas
    public ChangeAimLineGreenCommand(AimLine aimLine) { this.aimLine = aimLine; }

    yousimas
    public void setColor(Color color) { aimLine.setColor(color); }

    yousimas
    @Override
    public void execute() { setColor(Color.GREEN); }

    yousimas
    public void undo() { aimLine.setColor(Color.RED); }
}
```

3.11. Facade

Aprašymas

UML klasių diagramos

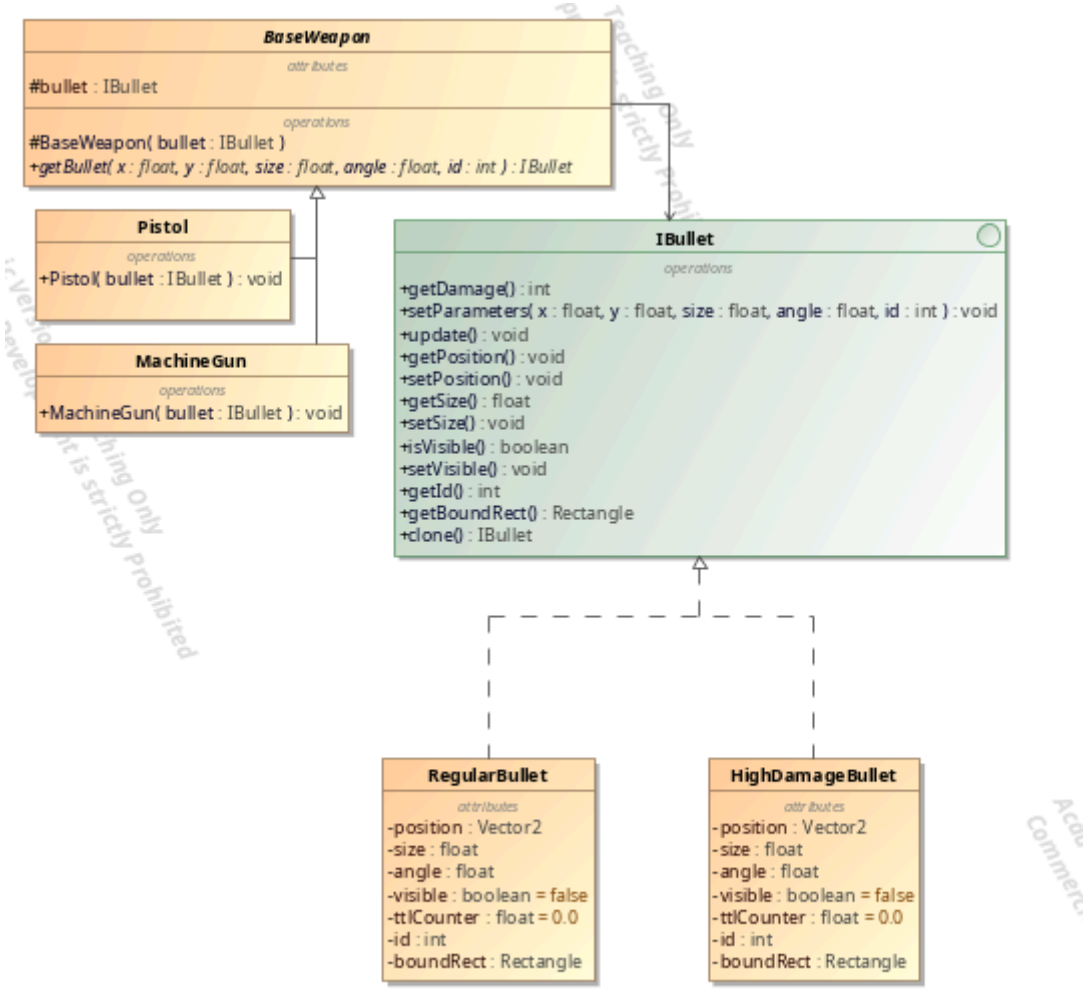
Kodas

3.12. Bridge

Aprašymas

Bridge yra šablonas, leidžiantis suskaldyti didelę klasę į kelias klases, kurias galima individualiai keisti. Pasinaudojant abstrakciją ir implementacija, išvengiama daug kombinacijų kaip atskirų klasių. Šiuo atveju **"**BaseWeapon**"** klasė yra abstrakti, kuri turi sąsają **"**IBullet**"**, kuri turi skirtingas implementacijas kaip RegularBullet ir "HighDamageBullet". Kurių reikšmes galime apsirašyti atskirai, nekuriant naujų ir atskirų "BaseWeapon" klasės variacijų.

UML klasių diagramos

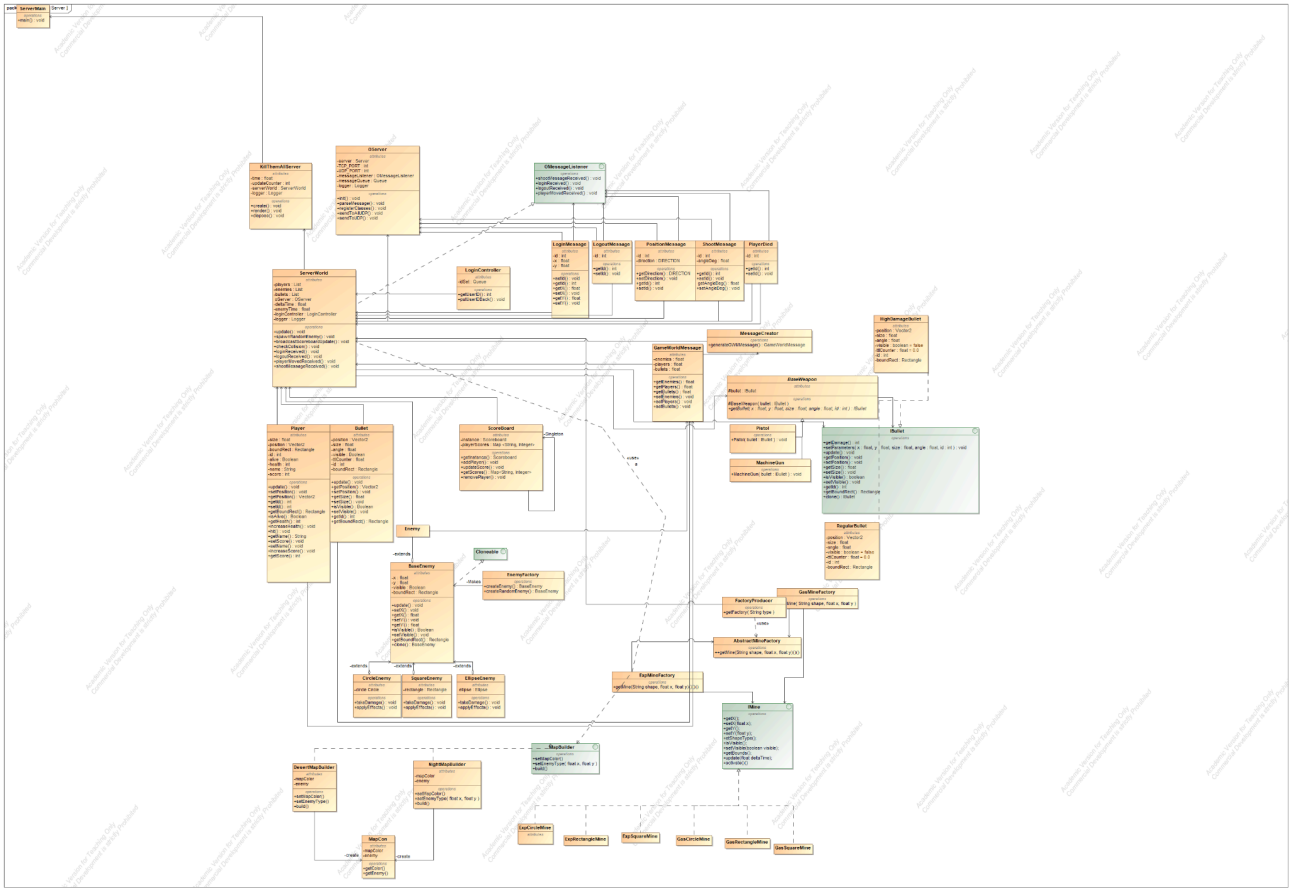


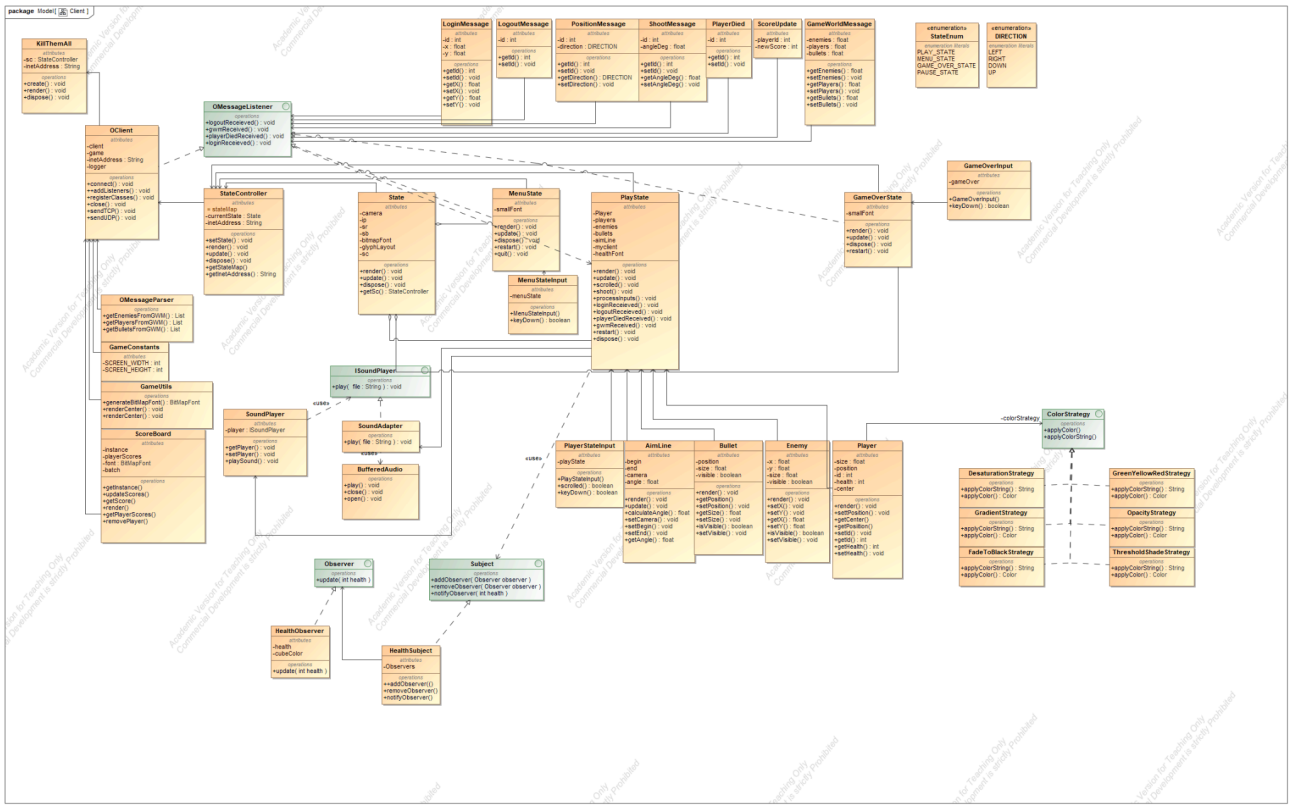
Kodas

```
public class Pistol extends BaseWeapon{

    public Pistol(IBullet bullet){
        super(bullet);
    }

    @Override
    public IBullet getBullet(float x, float y, float size, float angle, int
id) {
        bullet.setParameters(x, y, size, angle, id);
        return bullet;
    }

}
public class MachineGun extends BaseWeapon{

    public MachineGun(IBullet bullet){
        super(bullet);
    }

    @Override
    public IBullet getBullet(float x, float y, float size, float angle, int
id) {
        bullet.setParameters(x, y, size, angle, id);
        return bullet;
    }
}
public class RegularBullet implements IBullet {

    private Vector2 position;
    private float size;
    private float angle;
    private boolean visible = true;

    private float ttlCounter = 0;
    private int id;
    private Rectangle boundRect;
}
public class HighDamageBullet implements IBullet{

    private Vector2 position;
    private float size;
    private float angle;
    private boolean visible = true;
```

```
    private float ttlCounter = 0;
    private int id;
    private Rectangle boundRect;

    @Override
    public int getDamage() {
        return 25;
    }
}
public interface IBullet {
    abstract public int getDamage();
    abstract public void setParameters(float x, float y, float size, float
angle, int id);
    abstract public void update(float deltaTime);
    abstract public Vector2 getPosition();
    abstract public void setPosition(Vector2 position) ;
    abstract public float getSize();
    abstract public void setSize(float size);
    abstract public boolean isVisible();
    abstract public void setVisible(boolean visible);
    abstract public int getId();
    abstract public Rectangle getBoundRect();
    abstract public IBullet clone();
}

public void loginReceived(Connection con, LoginMessage m) {

        int id = loginController.getUserID();

        BaseWeapon weapon = new Pistol(new HighDamageBullet());



        Player newPlayer = new Player(m.getX(), m.getY(), 50, id,
"Player_"+id, weapon);
        players.add(newPlayer);
        logger.debug("Login Message received from : "+
newPlayer.getName());

        m.setId(id);
        oServer.sendToUDP(con.getID(), m);
    }
```

4. Galutinės UML diagramos





5. Literatūros sąrašas

- Originalus projektas: https://github.com/ogzkrt/killthemall
- Modifikuotas projektas: https://github.com/MartynasKul/OPP