Repositorio 2

Link: https://github.com/hungthuanmk/HCMIU_PlantsAndZombies

Violación del principio OCP(2):

Podemos observar que en la clase “Peashooter3” en el método attack(ArrayList<Bullet> bulletArrayList) se están implementando varios condicionales que definen comportamientos de ataque distintos en función de otro método llamado getFramePassed(). Esto puede ocasionar que la clase “Peashooter3” deba ser modificada cuando se quiera agregar o modificar alguna estrategia de ataque.

```
@Override
public void attack(ArrayList<Bullet> bulletArrayList) {
        if (getFramePassed() == getAttackInterval()) {
                bulletArrayList.add(new pz.bullet.BPeashooter((getPos().x + getWidth() * 0.8f) ,
                                                                (getPos().y + getHeight() * 0.15f),
                                                                getDamage()));

        }
        else
                if (getFramePassed() == getAttackInterval()+10) {
                        bulletArrayList.add(new pz.bullet.BPeashooter((getPos().x + getWidth() * 0.8f) ,
                                                                        (getPos().y + getHeight() * 0.15f),
                                                                        getDamage()) {

                                @Override
                                public void setSpeed(float speed) {
                                        // TODO Auto-generated method stub
                                        super.setSpeed(-speed);
                                }
                        });

                }
                else
                        if (getFramePassed() == getAttackInterval()+20) {
                                bulletArrayList.add(new pz.bullet.BPeashooter((getPos().x + getWidth() * 0.8f) ,
                                                                                (getPos().y + getHeight() * 0.15f),
                                                                                getDamage()));

                                //setFramePassed(0);
                        }
                        else
                                if (getFramePassed() == getAttackInterval()+30) {
                                        bulletArrayList.add(new pz.bullet.BPeashooter((getPos().x + getWidth() * 0.8f) ,
                                                                                        (getPos().y + getHeight() * 0.15f),
                                                                                        getDamage()) {

                                                @Override
                                                public void setSpeed(float speed) {
                                                        // TODO Auto-generated method stub
                                                        super.setSpeed(-speed);
                                                }
                                        });
                                        setFramePassed(0);
                                }

        setFramePassed(getFramePassed()+1);
}
```

Solución:

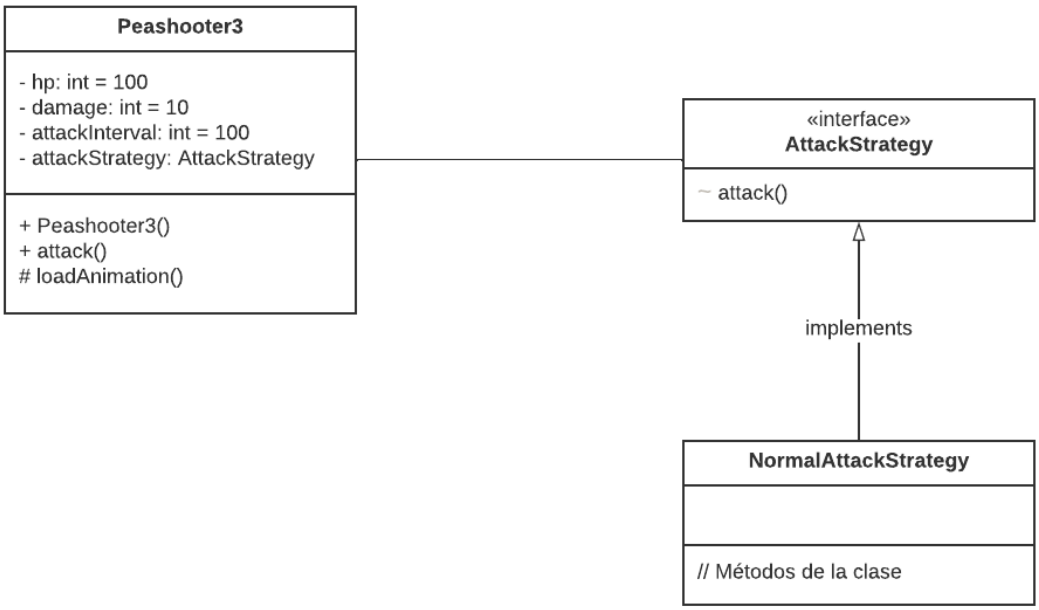
Para evitar tener que usar los condicionales que se encuentran ahí, podríamos definir una interfaz aparte llamada AttackStrategy de tal manera que sea un atributo de la clase “Peashooter3” y esté presente también en el constructor de la clase. Luego se podría definir diferentes clases que representen las distintas estrategias de ataque y a su vez estas mismas implementen la interfaz AttackStrategy.

```java
public class Peashooter3 extends pz.Plant {

    private static int _hp = 100;
    private static int _damage = 10;
    private static int _attackInterval = 100;
    private AttackStrategy attackStrategy;

    public Peashooter3(Position pos, AttackStrategy attackStrategy) {
        super("Peashooter3", _hp, _damage, _attackInterval, pos);
        this.attackStrategy = attackStrategy;
    }

    @Override
    protected void loadAnimation() {
        setAnimation(AnimationLoader.getAnimationFromFolder("res/Plants/PeaShooter/Idle", 30));
    }

    @Override
    public void attack(ArrayList<Bullet> bulletArrayList) {
        attackStrategy.attack(bulletArrayList, peashooter: this); // Delegar el ataque a la estrategia
    }

    @Override
    public void move() {

    }
}
```

```java
package com.mycompany.mavenproject3;

import java.util.ArrayList;

public interface AttackStrategy {
    void attack(ArrayList<Bullet> bulletArrayList, Peashooter3 peashooter);
}
```

```java
//Ejemplo con un ataque normal
public class NormalAttackStrategy implements AttackStrategy {

    @Override
    public void attack(ArrayList<Bullet> bulletArrayList, Peashooter3 peashooter) {
        // Lógica del programa
    }
}
```

UML:


Peashooter3
- hp: int = 100
- damage: int = 10
- attackInterval: int = 100
- attackStrategy: AttackStrategy
+ Peashooter3()
+ attack()
# loadAnimation()
«interface»
AttackStrategy
~ attack()
implements
NormalAttackStrategy
// Métodos de la clase

Violación del principio LSP(1):

Tenemos una clase padre llamada “Character” de la cual las clases “Plant” y “Zombies” heredan y a su vez clases más específicas de estos tipos heredan de estas respectivamente. Se puede observar que en la clase “Bloomerang” tenemos un método llamado move() que al mismo tiempo está en la clase “Character” pero sabemos que de por si las plantas no se mueven e incluso el método move() en “Bloomerang” está vacío. Es entonces cuando se está incumpliendo el principio de LSP, ya que si se intenta utilizar un objeto Bloomerang donde se espera un objeto Character, el comportamiento del programa podría ser diferente.

```java
public class Bloomerang extends pz.Plant {

	private static int hp = 100;
	private static int damage = 5;
	private static int attackInterval = 100;

	public Bloomerang(Position pos) {
		super("Bloomerang", hp, damage, attackInterval, pos);
	}

	@Override
	public void attack(ArrayList<Bullet> bulletList) {
		if (getAnimation().getFrame() == 14 && getFramePassed() > getAttackInterval()) {
			bulletList.add(new BBloomerang(getPos().x + getWidth()/2f, getPos().y + getHeight()*0.33f, getDamage()));
			setFramePassed(0);
		}
		setFramePassed(getFramePassed()+1);
	}

	@Override
	public void move() {

	}

	@Override
	protected void loadAnimation() {
		setAnimation(AnimationLoader.getAnimationFromFolder("res/Plants/Bloomerang/Idle", 150));
	}

}
```

Solución:

La solución que se plantea sería definir una interfaz aparte Movable() la cual la clase Bloomerang implementaría y así eliminamos el método move() de la clase padre “Character”, de esta forma se evita un comportamiento extraño en el programa y de esta forma todas las clases derivadas de “Character” sean reemplazables en cualquier lugar donde se espera un objeto “Character”.

```java
public abstract class Character {
    // Lógica de la clase
    // No incluye el método move()
}

public abstract class Plant extends Character {
    // Lógica de la clase
    // No incluye el método move()
}

public class Bloomerang extends Plant implements Movable {
    // Lógica de la clase
    // Implementa el método move() específico para las plantas móviles
}
```

```java
package com.mycompany.mavenproject3;

public interface Movable {
    void move();
}
```

Violación del principio LSP(2):

La situación se repite en la clase “Peashooter2” con el método move() vacío.

```java
package pz.plant;

import java.util.ArrayList;

import com.Position;

import gui.AnimationLoader;

import pz.Bullet;

public class Peashooter2 extends pz.Plant {

	private static int _hp = 100;
	private static int _damage = 10;
	private static int _attackInterval = 100;
	private static boolean _attackCoolDown = false;

	public Peashooter2(Position pos) {
		super("Peashooter2", _hp, _damage, _attackInterval, pos);
	}

	@Override
	protected void loadAnimation() {
		setAnimation(AnimationLoader.getAnimationFromFolder("res/Plants/PeaShooter2/Idle", 30));
	}

	@Override
	public void attack(ArrayList<Bullet> bulletArrayList) {
		if ((getAnimation().getFrame() == 16 || getAnimation().getFrame() == 27) && _attackCoolDown == false) {
			bulletArrayList.add(new pz.bullet.BPeashooter((getPos().x + getWidth() * 0.8f) ,
                                                          (getPos().y + getHeight() * 0.15f),
                                                          getDamage()));

			_attackCoolDown = true;
		}
		if ((getAnimation().getFrame() >= 17 && getAnimation().getFrame() <=26) ||
				(getAnimation().getFrame() >= 28 && getAnimation().getFrame() <= 40) ||
				(getAnimation().getFrame() >= 0 && getAnimation().getFrame() <= 15)) {
			_attackCoolDown = false;
		}

	}

	@Override
	public void move() {

	}
}
```

Solución:

La solución es similar a la que planteamos anteriormente implementado ahora la interfaz en la clase “Peashooter2”.

```java
public abstract class Character {
    // Lógica de la clase
    // No incluye el método move()
}

public abstract class Plant extends Character {
    // Lógica de la clase
    // No incluye el método move()
}

public class Peashooter2 extends Plant implements Movable {
    // Lógica de la clase
    // Implementa el método move() específico para las plantas móviles
}
```

Violación del principio LSP(3):

La situación se repite en la clase “Peashooter2” con el método move() vacío.

```java
public class Peashooter3 extends pz.Plant {

	private static int _hp = 100;
	private static int _damage = 10;
	private static int _attackInterval = 100;

	public Peashooter3(Position pos) {
		super("Peashooter3", _hp, _damage, _attackInterval, pos);
	}

	@Override
	protected void loadAnimation() {
		setAnimation(AnimationLoader.getAnimationFromFolder("res/Plants/PeaShooter/Idle", 30));
	}
```

```java
public class Peashooter3 extends pz.Plant {
	public void attack(ArrayList<Bullet> bulletArrayList) {
		if (getFramePassed() == getAttackInterval()) {
			bulletArrayList.add(new pz.bullet.BPeashooter((getPos().x + getWidth() * 0.8f) ,
                                                          (getPos().y + getHeight() * 0.15f),
                                                          getDamage()));
		}
		else
			if (getFramePassed() == getAttackInterval()+10) {
				bulletArrayList.add(new pz.bullet.BPeashooter((getPos().x + getWidth() * 0.8f) ,
                                                              (getPos().y + getHeight() * 0.15f),
                                                              getDamage()) {

					@Override
					public void setSpeed(float speed) {
						// TODO Auto-generated method stub
						super.setSpeed(-speed);
					}
				});

			}
			else
				if (getFramePassed() == getAttackInterval()+20) {
					bulletArrayList.add(new pz.bullet.BPeashooter((getPos().x + getWidth() * 0.8f) ,
                                                                  (getPos().y + getHeight() * 0.15f),
                                                                  getDamage()));

					//setFramePassed(0);
				}
				else
					if (getFramePassed() == getAttackInterval()+30) {
						bulletArrayList.add(new pz.bullet.BPeashooter((getPos().x + getWidth() * 0.8f) ,
                                                                      (getPos().y + getHeight() * 0.15f),
                                                                      getDamage()) {

							@Override
							public void setSpeed(float speed) {
								// TODO Auto-generated method stub
								super.setSpeed(-speed);
							}
						});
						setFramePassed(0);
					}

		setFramePassed(getFramePassed()+1);
	}

	@Override
	public void move() {

	}
}
```

Solución:

La solución es similar a la que planteamos anteriormente implementado ahora la interfaz en la clase “Peashooter3”.

```
public abstract class Character {
    // Lógica de la clase
    // No incluye el método move()
}

public abstract class Plant extends Character {
    // Lógica de la clase
    // No incluye el método move()
}

public class Peashooter3 extends Plant implements Movable {
    // Lógica de la clase
    // Implementa el método move() específico para las plantas móviles
}
```

UML para los tres casos de Liskov: