

OOP Game Report

**Members**

Lê Hồng Đức

Lý Hoàng Vũ

Nguyễn Trần Duy Thanh

Lê Hoàng Phú Thành

# Introduction

In this report, we introduce our remake version of the PlantsVersusZombies game. This is a well-known game, and it is challenging to create this game from scratch. We developed this game with the help of the JSwing framework of Java. We will give you a brief and concise overview of our game.

# UML Diagram and Design Explanation:

Firstly, we included have one basic abstract class to represent all game object: GameEntity.

This basic class will contains common methods like:

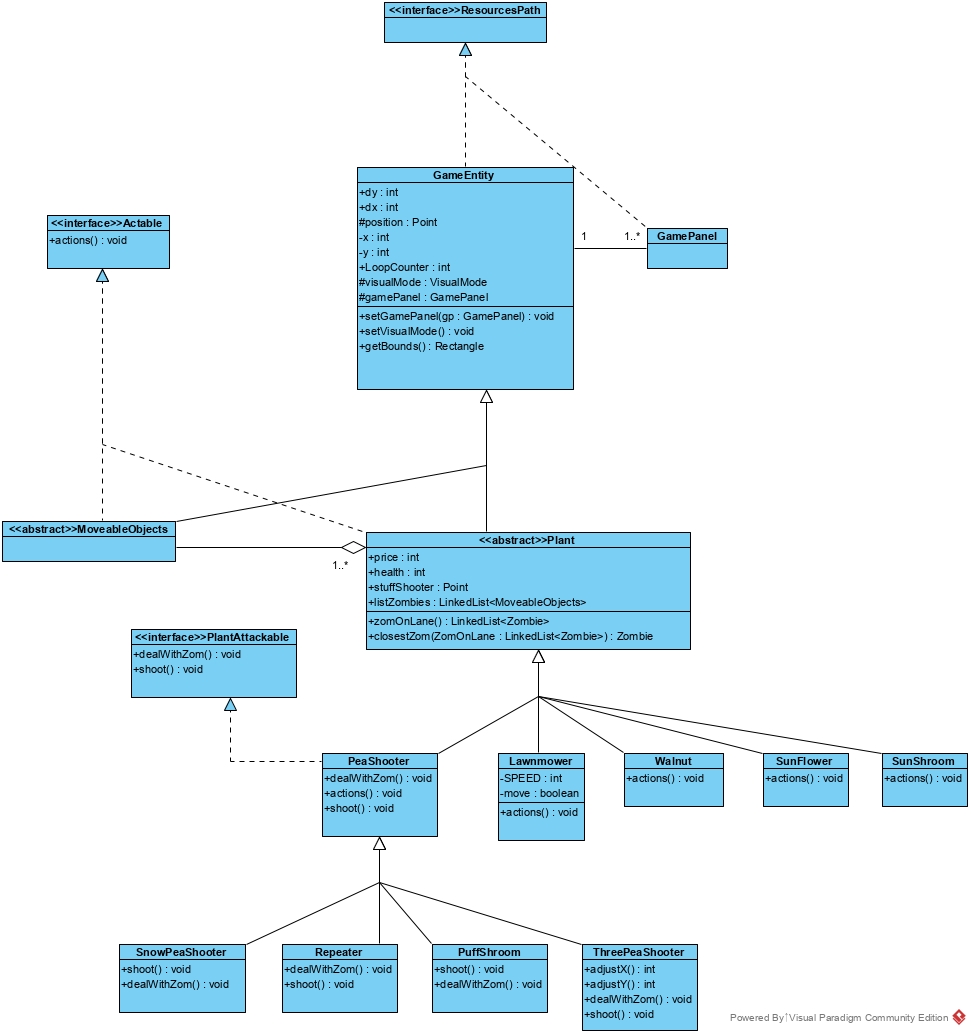
* getImage(): call in paintComponent() in every game loop
* action(): call in every game loop

Then, from this base class, we created 3 abstract subclasses to categorizes the entities further:

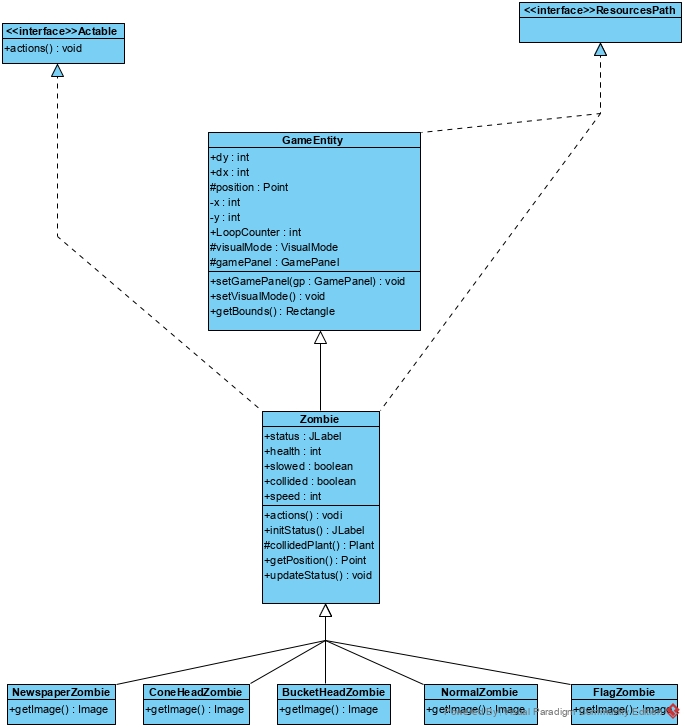
* Plants
* Zombies
* Movable Objects(Peas, Suns, Snows, …)

Then, using polymorphism, we extend each abstract class above to create various concrete classes, which each class overwrite the common methods above:

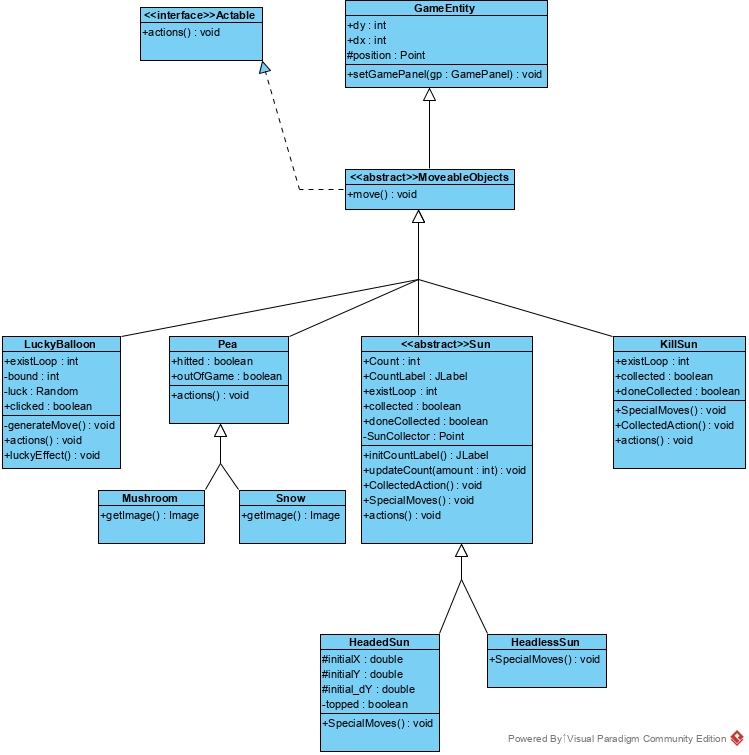
* getImage(): return the appropriate image
* action(): do the correct actions
* Constructors



**Figure 1.** Plants diagram



**Figure 2.** Zombies diagram

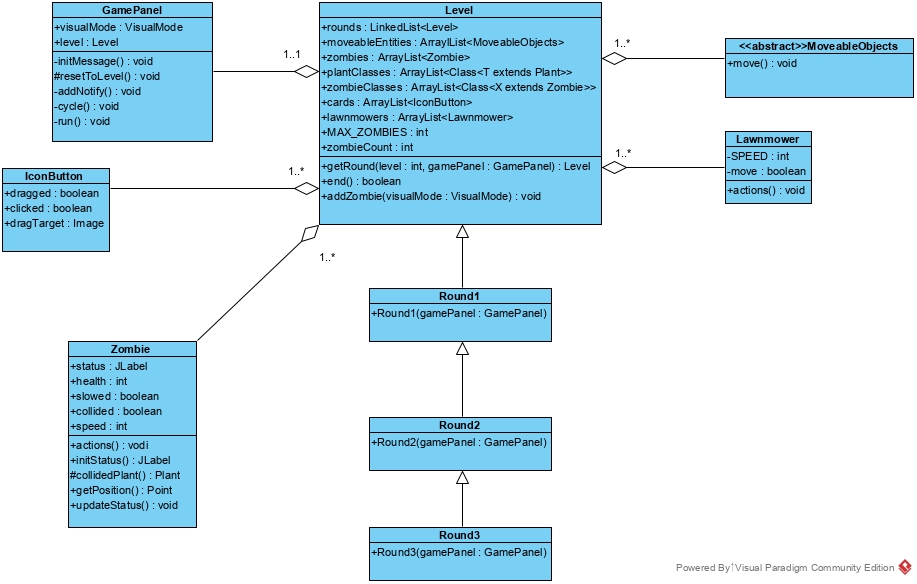


**Figure 3.** Movable objects diagram

The next diagram is the central system in our game that control the game logic, and this is also where we apply Reflection and Strategy Design Pattern to solve various problems related to switching or adding rounds and visual modes dynamically.

Level base class declared references to all variables including lists of plants classes, zombie classes, maximum zombie number, visual mode,… Each of those references will be instantiated within the constructors of the concrete subclasses: Round1, Round2, Round3, ... Any round will extend the previous round to reuse the code as we just add more in each round. In case of a whole new round, it will be extended straight from Level.

The list mechanism with support form Reflection technique and Strategy design pattern helps us to switch round dynamically at run time as the player finishes a round and move to the next round, in more details, we pass a level number and return the right Round class in the list of Levels then instantiate that class. The same applies when we change visual mode dynamically as the final round uses NightMode, we also pass a level number and return a visual mode.



**Figure 4.** Game levels diagram

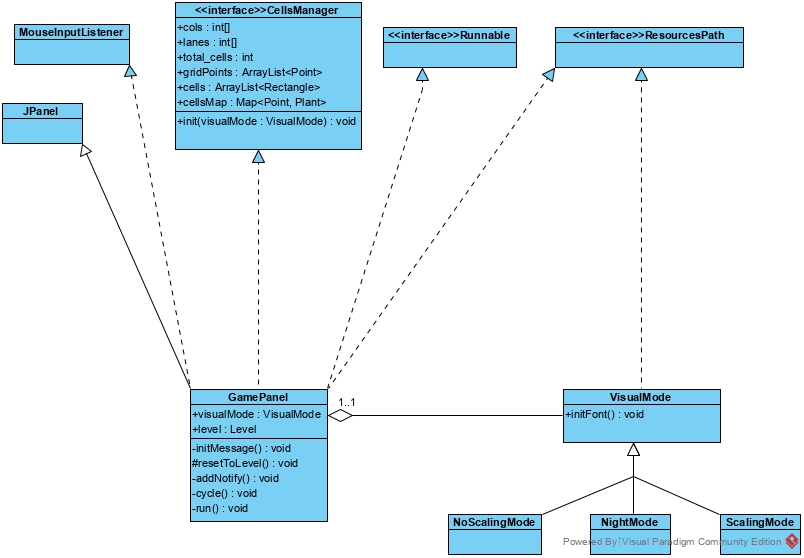
The last diagram describes the main game panel, it basically uses everything described above, combine, loop through each of them, call and display them (with Graphics g) at the right time using a loop counter to track time with a certain delay between each loop.

The important thing to notice here is the CellManager and ResourcePath interfaces.

* The ResourcePath interface do 2 important things, get the right related path (machine independent) and, from that, create and store strings representing relative paths to every resource file.
* CellsManager interface store the manually measured distance between each cell (as the distances between cells are not uniform) in the map so that we can display plants things at suitable positions on the map later.

The main advantage that led us to use interfaces for these things is that all variables declared in an interface is public static final, which matched perfectly to what we need for those variables as they are, literally, just constants.

Each of the visual modes, uses the links in ResourcePath interface and get its needed resources. E.g.: NightMode get image from dark background path while NoScalingMode get image from light background path.



**Figure 5.** GUI implementation diagram

# Design Patterns/Principles

The following design patterns and principles are applied in this project, an explanation for their strengths is in Section V.

Fully implemented patterns/principles

* KISS philosophy
* Strategy design pattern

Partially implemented patterns/principles

* Defensive design
* Discoverability principle
* Don’t Repeat yourself principle
* Uniform access principle
* Minimal Agile model

# Evaluation

## Overall evaluation

The project makes good uses of basic Object Oriented programming ideas such as inheritance, interfaces, abstract classes to express the relationship of components of the game. By dividing the project into many levels of abstraction using Strategy design pattern, it is possible to keep the minimum change needed for any upcoming feature/optimization as well as flexibility in the game mechanism, dependencies, and resources. Nonetheless, the lack of a consistent set of principles is also displayed in the code and remains a barrier as the codebase grows larger.

## Strength

* KISS
  + Physics-like model
  + Discoverability
* Uniform Access Principle
  + Access modifiers and getters and setters are strictly under-controlled
* Interface and abstract
  + Strategy design pattern
  + Polymorphism and encapsulation
* Defensive design
  + Try catch clauses

The project presents a simple natural physics-like model of the world (*dx,dy, etc*), makes learning and adapting this codebase feasible (and therefore this project is great for educational purposes). Due to this pure object-oriented thinking, it has the advantage of discoverability in which finding bugs or a visual object’s behavior classes extremely easy.

Access modifiers and getters and setters are strictly under-controlled, thus let the project has the consistency of the Uniform access principle, as a change in the implementation of an API of a class does not lead to another’s change.

Inheritance is exploited as a way to minimize the boilerplate codes, keeps it DRY. Interface and abstract classes are adopted as a way to implement Strategy design pattern for they provide an anecdote to describe polymorphism and encapsulation. Though it is far from being Single responsibility principle, the results are plausible to some extent.

Try catch clauses are heavily abused as part of the defensive design to provide a smooth experience for the players and testing utilities for developers.

By dividing a complex system of PlantsVersusZombies into small parts yet still maintain its stability in design, the developers are free to express and implement their ideas in the components under their responsibilities. This helps the game to have richer features and faster parallel deployment/testing workflow.

## Subject to improvement

* Lack of a set of rigorous principles
* Unable to accept open source community workforces
* Under refinement

As mentioned above, the lack of a set of rigorous principles and a framework to test them has resulted in human efforts. The consistency in design has to take a long period of time to reach and this is displayed through the commit history. The principles though are applied, not yet fully compulsory nor coherent.

The naive implementation of classes though are great for hacks of debugging, creates a barrier in accepting open source community workforces (if there are in the future). The code is also potentially prone to type cast errors without proper check.

The game is also in its most basic form and currently under refinement in both GUI and backend (E.g. savepoints), which has not created a decent user experience.

The codebase is also needed to be refactored in variable/method names and tree file structures.

That being said, all of these weaknesses subject to criterion (and more hidden ones) are under-plan to be fixed in the future. This plan could be summarized as specific checkpoints in Section VII.

# Plan for future improvements

The following checkpoints are mentioned in critical order.

1. **Refactoring names and structures**

This is the most important since we do not want to be under the impression of working on a forever-on-going project. As every successful game ever made, we should clean the codebase to reach the official v1.0. This will be the foundation for heavy changes in the future and creates a warm welcome for future workforces (as the most important thing in working in teams is the mental well-being of each member).

1. **Change in principles**

Though it is a well-known good reason for any OOP project to adapt SOLID principles, it is usually uncommon for a project to have all 5 of them. Based on the current properties of this project, in the future, we will try to gradually change the codebase by applying the following principles in order: I-D, S, O, (L). This plan of adapting principles are subject to discussion.

1. **Refinements of GUI and savepoints**

Though it’s hard to define and measure user experience, we think savepoints are needed to make a good game. The current game has 3 levels, the average time to clear them all is about 20 minutes. Savepoints are needed as giving players peace of mind as well as growing the user base in the popular short-span-of-freedom gaming direction.

There is no specific list of needed improvements for the GUI yet, however a proper designer is added to the list of enhancements.

## 