UML Class Diagram Design for a "Plants vs. Zombies" Inspired Game

I. Introduction to the Game Design Challenge

The project involves the creation of a Java desktop game application, drawing inspiration from the popular "Plants vs. Zombies" title. The fundamental objective for the player is to strategically deploy a limited supply of plants and seeds to establish defenses, thereby preventing a relentless mob of zombies from invading their virtual home. The game concludes either when a zombie successfully breaches the player's defenses or when a three-minute game timer expires, determining victory for either the plants or the zombies.[1]

The primary purpose of this report is to lay out a comprehensive architectural blueprint for the game's core systems through a detailed UML Class Diagram. This diagram is not merely a visual representation but a foundational document that defines the classes, their attributes, methods, and the intricate relationships between them. Such a blueprint is essential for ensuring adherence to robust object-oriented design principles, promoting system maintainability, and facilitating future extensibility, particularly given the phased development approach outlined for the project.[1]

The game environment is structured as a grid-based board, comprising multiple lanes, each further divided into a series of tiles. This grid serves as the operational space where all game entities, including plants and zombies, are positioned and interact.[1] The specification indicates that the board can have "5 or 6 lanes" and each lane can have "9 to 10 tiles".[1] This variability in board dimensions, coupled with the requirement to implement "at least three different levels, i.e., three different maps" [1], necessitates a flexible design for the

GameBoard class. Rather than hardcoding specific dimensions, the GameBoard must be initialized with parameters for the number of lanes and tiles per lane. This

approach ensures that the same GameBoard class can seamlessly adapt to various map configurations without requiring structural modifications, directly supporting the multi-level game requirement and enhancing overall system adaptability.

II. Core Game Entities: Base Classes and Hierarchies

To foster code reusability and enable polymorphic interactions, a high-level abstract class, GameEntity, is established as the foundational base for all active elements that will reside on the game board. This class is designed to encapsulate common properties universally applicable to both plants and zombies, such as their spatial coordinates (row, column) and their capacity to sustain damage (health).

Both Zombie and Plant entities share fundamental characteristics: they occupy specific locations on the game board, possess a health attribute, and engage in interactions within the game environment. Despite their distinct behaviors, their shared existence on a grid and their inherent capacity for state management (like health) make a single abstract base class a logical and efficient design choice. This approach aligns with the principles of object-oriented design by promoting inheritance and significantly reducing redundant code, thereby contributing to a more streamlined and maintainable codebase. A GameEntity base class simplifies the management of objects on the GameBoard (e.g., a Tile can generically hold a GameEntity), facilitates common operations like collision detection, and supports polymorphic method calls across diverse entity types.

Designing the Zombie Class Hierarchy

The Zombie class is designed as an abstract base class, defining the common attributes and behaviors shared by all zombie types. These core attributes include speed, damage dealt to plants, and health, representing the damage a zombie can withstand.[1] Common methods for all zombies would include

move(), attack(Plant target), takeDamage(int amount), and isAlive().

Several specialized Zombie subtypes inherit from this base class, each implementing

unique characteristics or modifying base behaviors:

- **NormalZombie**: This serves as the baseline, with specified attributes of speed = 4, damage = 10, and health = 70.[1]
- **FlagZombie**: Described as moving "slightly faster" and signaling "a huge wave incoming".[1] The specification also notes an "armor effect" that "slows down the zombie by 1".[1] This suggests that while its base movement might be inherently faster, the flag it carries imposes a slight speed penalty, or its base speed is adjusted to account for this. Its primary unique function is to trigger game events related to zombie waves.
- **ConeheadZombie**: This zombie "uses a traffic cone to protect itself".[1] Its inherent armor decreases its speed by 2 and its damage by 2, signifying a trade-off between defense and offensive capabilities.[1]
- **PoleVaultingZombie**: Characterized by a "single jump" that allows it to "jump over the first plant it encounters with a pole".[1] This represents a unique movement behavior that bypasses direct plant interaction initially.
- **BucketheadZombie**: This zombie "has a bucket that is extremely resistant to damage".[1] Its armor significantly impacts its attributes, decreasing speed by 3 and damage by 5, while implying a substantial increase in its effective health or damage tolerance.[1]

The description of zombies "wearing armor" might initially suggest a design where armor is a separate, attachable component, potentially using a Decorator pattern. However, the subsequent descriptions consistently list "Flag Zombie," "Conehead Zombie," "Pole Vaulting Zombie," and "Buckethead Zombie" as distinct, specialized *subtypes* of zombies, each with predefined "armor effects" or unique behaviors directly tied to their type.[1] For the initial design phase (MCO1), treating these "armor effects" as inherent properties of these specific

Zombie subtypes simplifies the inheritance hierarchy. This means a FlagZombie *is* a zombie that inherently possesses flag-related properties, rather than a NormalZombie that *wears* a flag. This approach leverages inheritance for specialization, where the unique attributes or behavior modifications are integrated into the constructor or overridden methods of the specific subtype. While a Decorator pattern could offer greater runtime flexibility for combining different armor types, it introduces a level of complexity not explicitly mandated for the current scope and can be considered a future refinement.

Designing the Plant Class Hierarchy

Similar to zombies, the Plant class is an abstract base class that defines common attributes and behaviors for all plant types. These attributes include sunCost (the amount of sun needed for deployment), regenerateRate (cooldown time), damage dealt to zombies, health (damage sustained), range (how far its attack reaches), directDamage (damage at closer range), and speed (attack frequency).[1] Common methods would include

attack(Zombie target), takeDamage(int amount), isAlive(), and potentially generateResource() for sun-producing plants or activateAbility() for plants with special effects.

Specific Plant subtypes inherit from this base class, each implementing its unique function:

- **Sunflower**: Its primary function is to "give you additional sun".[1] This plant would specialize the
  generateResource() method.
- **Peashooter**: This plant "shoots peas at attacking zombies".[1] Its
  attack() method would be specialized for projectile-based damage.
- **CherryBomb**: This plant "blows up all zombies in an area".[1] This indicates an area-of-effect (AoE) attack, likely a single-use ability.
- **WallNut**: Primarily a defensive unit, it "blocks all zombies and protects your other plants".[1] It would typically have high health and no offensive capabilities.
- **PotatoMine**: This plant "explodes on contact but takes time to arm itself".[1] Its implementation requires managing an internal state (armed/unarmed) and a delay mechanism before activation.
- **SnowPea**: This plant "shoots frozen peas that damage and slow the enemy".[1] It combines direct damage with a debuff effect on the target.

An interesting consideration arises with the Shovel. Although listed under "Plants" in the specifications [1], its described function is to "dig up a plant to make room for other plant".[1] This description clearly indicates that the

Shovel is an *action* or a *tool* used by the player, rather than a static entity placed on the game board that interacts with zombies. It does not possess typical plant attributes such as damage or health. Including the Shovel as a Plant subtype would violate the Single Responsibility Principle and create an illogical hierarchy within the

Plant family. Instead, it is more appropriately modeled as a method within a Player or GameController class (e.g., player.useShovel(Tile targetTile)). This design choice ensures a cleaner, more logical object-oriented structure, representing the Shovel as an item or action trigger within the player's interface rather than a deployable game entity.

III. Game Environment and Board Structure

The game's operational space is defined by two key classes: GameBoard and Tile.

GameBoard Class

The GameBoard class represents the entire grid-based playing field. It is responsible for managing the collection of individual tiles that constitute the game area.

- **Attributes**: It will contain a data structure, such as a 2D array or a list of Tile objects, to hold the game's tiles. It will also store numLanes (rows) and numTilesPerLane (columns) to define its dimensions.
- **Methods**: Essential methods include placeEntity(GameEntity entity, int row, int col) to position an entity on a specific tile, removeEntity(int row, int col) to clear a tile, getEntity(int row, int col) to retrieve an entity at a given position, getLane(int row) to access all tiles in a specific lane, and isValidPosition(int row, int col) to validate coordinates.[1]

Tile Class

The Tile class represents a single cell within the GameBoard. Each tile acts as a container for game entities.

- **Attributes**: It will store its own row and column coordinates and hold a reference to the occupyingEntity, which will be a GameEntity object if the tile is occupied, or null if it is empty.

- **Methods**: Key methods include isOccupied() to check its occupancy status, setOccupyingEntity(GameEntity entity) to place an entity on it, and getOccupyingEntity() to retrieve the entity currently occupying it.

Both plants and zombies occupy specific tiles on the game board.[1] For instance, plants are placed "on the tiles," and zombie movement is tracked by their tile positions, as demonstrated by console outputs like "Zombie previously in Row 1 Column 8 now moved to Row 1 Column 7".[1] This necessitates that each

Tile maintains a reference to the GameEntity currently occupying it. This design enables efficient spatial queries, such as determining "what plant is at this tile?" or "are there zombies in this lane?". This structure establishes a clear separation of concerns: the GameBoard manages the overall grid, the Tile manages the state of an individual cell, and the GameEntity represents the interactive object. This hierarchical organization is fundamental for implementing core game logic, including entity movement, attack interactions, and placement rules.

IV. Game Mechanics and System Interactions

The central orchestrator of the entire game flow, state management, and interactions between entities is the GameEngine (or GameController) class. This class effectively serves as the "Model" in an MVC (Model-View-Controller) architectural pattern.[1]

GameEngine Class Responsibilities

- **Attributes**: The GameEngine maintains references to the gameBoard, tracks the playerSun currency, manages the gameTimeRemaining, and keeps lists of activeZombies and activePlants. It also holds the gameStatus (e.g., RUNNING, PLANTS_WIN, ZOMBIES_WIN).
- **Methods**:
  - startGame(): Initializes the game state.
  - updateGame(double deltaTime): This is the core game loop method, responsible for advancing game time, updating the states and positions of all active entities, and checking for win/loss conditions.

- generateSun(): Manages the constant rate at which sun drops.[1]
- collectSun(): Handles player interaction for collecting sun.
- spawnZombie(): Manages the generation of new zombies based on predefined time intervals and lane probabilities.[1]
- placePlant(PlantType type, int row, int col): Processes player requests to place plants, verifying sun cost and placement rules.[1]
- processAttacks(): Orchestrates interactions where plants attack zombies and vice-versa.
- checkWinConditions(): Evaluates the criteria for game termination.
- handlePlayerInput(InputEvent event): Receives and processes input from the user interface.

The game's progression is heavily reliant on time-based events. The game has a fixed duration of three minutes, sun drops at a "constant rate," and zombies are generated according to specific "time intervals".[1] This necessitates a precise time-keeping mechanism, managed centrally by the

GameEngine or a dedicated GameTimer component. This component is responsible for triggering all time-dependent events, such as the periodic generation of sun, the spawning of new zombie waves, and monitoring the overall game duration. This design ensures that all game events occur consistently and adhere strictly to the specified timings, which is crucial for maintaining game balance and progression.

Furthermore, the GameEngine must continuously monitor the game state to determine win or loss conditions. The game concludes if "one zombie reaches the player's home" or "When game time ends".[1] Therefore, the

GameEngine must constantly track the positions of all active zombies to detect if any breach the "home" boundary and regularly check the elapsed game time. These checks are integral to the main updateGame loop. This highlights the necessity for clear state variables within the GameEngine and efficient methods to query entity positions and time, ensuring accurate and timely game termination.

Resource Management (Sun)

The GameEngine is responsible for managing the player's sun currency. Sun is generated at a constant rate throughout the game and can be collected by the player.

Plants have a specific sunCost that must be met before they can be deployed.[1]

Entity Generation Logic

- **Zombie Generation**: Zombies are generated based on specific time intervals, as detailed in the table below.[1] There is an equal probability for a zombie to appear in any given lane.[1] The
  GameEngine will utilize a ZombieFactory or a similar mechanism to create instances of specific zombie types according to this schedule.

Period (seconds)	Interval (seconds)
30 to 80	Every 10 seconds
81 to 140	Every 5 seconds
141 to 170	Every 3 seconds
171 to 180	Wave of zombies

- **Plant Generation (Placement)**: Plants become available for use based on their regenerateRate (cooldown) and can only be placed if the player has collected sufficient sunCost.[1] Plant placement is initiated by player input, as indicated by requirements for user interaction for placing plants and collecting sun.[1]

Interactions: Attack and Defense

- **Plant Attacks**: Plants engage zombies within their specified range.[1] Some plants, like the
  Peashooter, shoot projectiles, while others, such as the CherryBomb, perform area-of-effect attacks. The SnowPea uniquely combines damage with a slowing effect on enemies. A notable aspect of plant attacks is the distinction between Damage and Direct damage attributes.[1] The
  Plant class includes both a general Damage attribute and a Direct damage attribute for "closer range" engagements. This implies a conditional attack

mechanism: when a plant attacks, it must assess the distance to its target zombie. If the zombie is within a predefined "closer range" threshold, the directDamage value is applied; otherwise, the general damage value is applied (provided the zombie is within the plant's overall range). This requires the plant's attack logic to dynamically access the positions of both the plant and its target, adding a layer of strategic depth to plant placement and requiring careful implementation of the attack method within Plant subclasses.

- **Zombie Movement and Attacks**: Zombies continuously move towards the player's home. Upon encountering a plant, they engage in attacks. The PoleVaultingZombie exhibits a unique movement pattern, capable of jumping over the first plant it encounters.[1]

V. Detailed Attribute Specifications and Data Types

This section provides a structured overview of the attributes for both Zombie and Plant entities. This comprehensive table is critical for accurate implementation and balancing of game mechanics. It serves as a centralized reference, directly translating textual specifications into a format suitable for defining class members and their initial values. It highlights how base attributes are specialized or modified by different subtypes, which is essential for understanding the inheritance hierarchy and implementing constructors.

Table 1: Game Entity Attributes (Base & Subtype Specifics)

Entity Type	Attribute	Value/Description
Zombie (Abstract)	speed	How fast the zombie moves
	damage	How much damage it deals to the plant it is attacking

	health	How much damage it can sustain
NormalZombie	speed	4
	damage	10
	health	70
FlagZombie	speed	(Faster base, but armor slows by 1)
	damage	(Base)
	health	(Base)
	Unique Behavior	Moves slightly faster, signals huge wave incoming.[1] Armor slows by 1.[1]
ConeheadZombie	speed	(Base, decreased by 2 due to armor)
	damage	(Base, decreased by 2 due to armor)
	health	(Base)
	Unique Behavior	Uses a traffic cone to protect itself.[1] Armor decreases speed by 2, decreases damage by 2.[1]
PoleVaultingZombie	speed	(Base)
	damage	(Base)
	health	(Base)

	Unique Behavior	Single jump, jumps over the first plant it encounters with a pole.[1]
BucketheadZombie	speed	(Base, decreased by 3 due to armor)
	damage	(Base, decreased by 5 due to armor)
	health	(Base, extremely resistant to damage)
	Unique Behavior	Has a bucket that is extremely resistant to damage.[1] Armor decreases speed by 3, decreases damage by 5.[1]
Plant (Abstract)	sunCost	How much sun needed to use this
	regenerateRate	How much time it needs to regenerate
	damage	How much damage it deals the zombie
	health	How much damage it can sustain
	range	How far its attack reaches
	directDamage	How much damage it costs when zombie is at a closer range
	speed	How fast the next attack will be

Sunflower	sunCost	(TBD)
	regenerateRate	(TBD)
	damage	-
	health	(TBD)
	range	-
	directDamage	-
	speed	-
	Function	Gives additional sun.[1]
Peashooter	sunCost	(TBD)
	regenerateRate	(TBD)
	damage	(TBD)
	health	(TBD)
	range	(TBD)
	directDamage	-
	speed	(TBD)
	Function	Shoots peas at attacking zombies.[1]
CherryBomb	sunCost	(TBD)
	regenerateRate	(TBD)
	damage	(TBD)

	health	(TBD)
	range	Area
	directDamage	-
	speed	-
	Function	Blows up all zombies in an area.[1]
WallNut	sunCost	(TBD)
	regenerateRate	(TBD)
	damage	-
	health	(TBD)
	range	-
	directDamage	-
	speed	-
	Function	Blocks all zombies and protects your other plants.[1]
PotatoMine	sunCost	(TBD)
	regenerateRate	(TBD)
	damage	(TBD)
	health	(TBD)
	range	Contact
	directDamage	(TBD)

	speed	-	
	Function	Explodes on contact but takes time to arm itself.[1]	
SnowPea	sunCost	(TBD)	
	regenerateRate	(TBD)	
	damage	(TBD)	
	health	(TBD)	
	range	(TBD)	
	directDamage	-	
	speed	(TBD)	
	Function	Shoots frozen peas that damage and slow the enemy.[1]	
Note: "(TBD)" indicates values not explicitly defined in the provided specifications, which will require referring to external references or using scaled values as permitted.[1]			

Data Types Consideration

For numerical attributes such as speed, damage, health, sunCost, and regenerateRate, int or double data types will be appropriate depending on whether fractional values are required. range can be represented as an int indicating the

number of tiles. Positional attributes like row and column will be int. gameTimeRemaining could be an int for seconds or a long for milliseconds to ensure sufficient precision. The use of enumerations (e.g., for ZombieType, PlantType, GameStatus) is recommended to enhance code clarity and type safety, preventing invalid states.

VI. Applying Object-Oriented Design Principles

The design of this game system rigorously adheres to fundamental object-oriented programming (OOP) principles to ensure robustness, maintainability, and extensibility.

Encapsulation and Information Hiding

A cornerstone of the design is the principle of encapsulation. All class attributes are declared as private, restricting direct external access. Access to these attributes is exclusively provided through public getter methods, and, where appropriate, setter methods. This approach ensures that the internal state of an object can only be modified or retrieved in a controlled manner, preventing unintended external interference. Furthermore, complex internal logic and state changes are managed entirely within the class itself, exposing only the necessary interfaces to other components of the system. This directly conforms to the project's requirement to "Exhibit proper object-based / object-oriented concepts, like encapsulation and information-hiding, etc.".[1]

Inheritance and Polymorphism

The system extensively leverages inheritance and polymorphism through the GameEntity -> Zombie/Plant hierarchy. This structure allows for the management of common behaviors and attributes at a higher level of abstraction. Polymorphism enables the GameBoard and GameEngine to interact with GameEntity objects generically, regardless of their specific subtype. For example, the GameEngine can

invoke entity.takeDamage() on any GameEntity without needing to determine if the object is a Zombie or a Plant, simplifying game logic and promoting code flexibility.

Abstract Classes and Interfaces

GameEntity, Zombie, and Plant are designed as abstract classes. This means they define common interfaces and may provide partial implementations, but they cannot be instantiated directly. Instead, they compel their concrete subclasses to provide specific implementations for abstract methods, ensuring that all entities adhere to a defined contract. Additionally, the design considers the use of interfaces for specific capabilities, such as Attackable for entities that can be targeted by attacks, ResourceGenerator for plants like the Sunflower, or AreaEffect for abilities like the Cherry Bomb. This further enhances modularity and allows for flexible composition of behaviors.

Design Patterns (Initial Considerations)

Several design patterns are considered to enhance the system's architecture:

- **Factory Pattern**: A GameEntityFactory or similar factory mechanism would be beneficial for creating instances of Zombie and Plant types. This pattern abstracts the object creation process, allowing for flexible generation of entities with varying attributes and supporting the game's generation logic.
- **Strategy Pattern**: This pattern could be applied to different attack behaviors (e.g., PeashooterAttackStrategy, CherryBombAttackStrategy) or movement patterns (e.g., for the PoleVaultingZombie). It allows algorithms to be selected at runtime, providing flexibility in how entities perform actions.
- **Observer Pattern**: While the Model-View-Controller (MVC) pattern broadly handles communication between the game model and its view, the Observer pattern could be considered for more granular notifications, such as the GameEngine notifying the View of specific state changes or events.

VII. UML Class Diagram Best Practices and Structure

The UML Class Diagram must prioritize clarity, readability, and adherence to standard notation to effectively serve as the system's architectural blueprint.

Clarity and Readability

- **Naming Conventions**: All class names, attribute names, and method signatures will be clear, concise, and follow standard Java naming conventions.
- **Logical Organization**: Classes will be organized logically within distinct packages (e.g., com.game.entities, com.game.board, com.game.engine) to improve modularity and navigability of the diagram.
- **Consistent Notation**: Standard UML notation will be consistently applied for associations, generalizations (inheritance), and dependencies to ensure unambiguous interpretation.

Initial Focus for MCO1 Deliverables

For the MCO1 deliverable, the UML Class Diagram will specifically highlight the Zombie class (initially focusing on NormalZombie as a concrete representation, or Zombie as an abstract base), Sunflower, and Peashooter classes.[1] This implies clearly showing the

Plant abstract base class and the inheritance relationships for Sunflower and Peashooter. Basic associations with the GameBoard and Tile classes will also be included to illustrate how these entities are placed and managed within the game environment.

Diagram Elements to Include

The UML Class Diagram will incorporate the following standard elements:

- **Classes**: Represented by rectangles divided into three compartments: the class name, its attributes, and its methods.
- **Abstract Classes**: Indicated by italicized names to denote their abstract nature.
- **Attributes**: Each attribute will specify its visibility (- for private, + for public, # for protected), its name, and its data type.
- **Methods**: Each method will specify its visibility, name, parameters, and return type.
- **Relationships**:
    - **Generalization (Inheritance)**: Depicted by a solid line with a hollow arrowhead pointing from the subclass to its superclass.
    - **Association**: Represented by a solid line connecting related classes, often accompanied by multiplicity indicators (e.g., 1 for one, * for many, 0..1 for zero or one) and optional role names to clarify the nature of the relationship.
    - **Composition/Aggregation**: A specialized form of association indicated by a diamond shape at the "whole" end of the relationship, with a filled diamond for composition (strong ownership) and an unfilled diamond for aggregation (weak ownership).
    - **Dependency**: Shown as a dashed line with an open arrowhead, indicating that one class depends on another (e.g., a method in one class uses an object of another class as a parameter).

VIII. Conclusion and Future Design Considerations

The proposed design for the "Plants vs. Zombies"-inspired game system rigorously adheres to core object-oriented principles, including encapsulation, inheritance, and polymorphism. This foundational approach provides a robust and scalable architecture, essential for managing the complexity inherent in game development. By abstracting common behaviors and attributes into base classes and specializing them through inheritance, the design minimizes redundancy and promotes code reusability. Encapsulation ensures that internal states are protected, leading to more predictable and maintainable code.

The current design inherently facilitates extensibility. The clear separation of concerns between game entities, the board structure, and the central game engine means that adding new Zombie types, Plant types, or even entirely new game mechanics can be

achieved with minimal impact on existing code. New entities can simply extend the appropriate abstract base classes and implement their unique behaviors, without requiring extensive refactoring of the core game loop or board management logic.

Looking ahead to MCO2 and beyond, several enhancements are planned or can be considered. The immediate next step involves the full implementation of all specified zombies and plants, building upon the established class hierarchy. A critical component for MCO2 is the integration of a robust Graphical User Interface (GUI) with mouse-controlled inputs, which will necessitate the complete implementation of the Model-View-Controller (MVC) architectural pattern.[1] This will involve developing distinct View and Controller layers that interact seamlessly with the existing Model (

GameEngine and entity classes).

Further enhancements could include advanced features such as saving and loading the current game status to files, which would require serialization mechanisms. The design also allows for the easy addition of new specialized zombies or plants, which are identified as potential bonus points.[1] Should more complex combinations of armor or power-ups be desired in the future, the current approach of modeling armor as an inherent type property could be refined to a more flexible Decorator pattern, allowing for dynamic modification of entity attributes at runtime. This systematic design approach ensures that the project can evolve and expand while maintaining a solid and manageable codebase.

Works cited

1. MP Specs (Term 3, AY 20124-25)-Nats.pdf