## Game Contoller

**1. Controller**

- **Definition**: Assign responsibility to a controller class that represents the overall system or a use case scenario.
- **In Code**: GameController is clearly a **Controller** in GRASP—it coordinates the flow of the game, manages game state, and handles major game use cases like starting/stopping the game, updating game state, rendering, and managing entities (towers, enemies, projectiles).
- **Examples**:
    - startGame(), stopGame(), startNextWave(), update() — all handle system events and delegate work accordingly.
    -

**2. Creator**

- **Definition**: Assign the responsibility of creating an object to a class that aggregates, contains, or closely uses the created objects.
- **In Code**: GameController creates and manages Enemy, Tower, and Projectile instances.
- **Examples**:
    - addTestEnemy(), startNextWave() — both demonstrate creation of enemy instances.
    - The controller also manages collections like List<Enemy>, List<Tower>, List<Projectile>.

**3. Information Expert**

- **Definition**: Assign responsibility to the class that has the necessary information to fulfill it.
- **In Code**:
    - Tower is responsible for deciding when to shoot (update()), because it has access to range, cooldown, etc.
    - Projectile determines if it has hit its target, and applies AoE damage if applicable.
    - Enemy manages its own health and rendering.

- These responsibilities are **delegated appropriately** to the classes that have the required data.

**4. High Cohesion**

- **Definition**: Keep objects focused, manageable, and understandable by assigning closely related responsibilities.
- **In Code**:
    - GameController is cohesive in that it manages game flow and delegates specialized tasks to other classes (e.g., Enemy.update(), Tower.render()).
    - Rendering is split appropriately: render() method in GameController calls render() on map, towers, enemies, and projectiles.

**5. Low Coupling**

- **Definition**: Minimize dependencies between classes to improve reuse and maintainability.
- **In Code**:
    - GameController uses interfaces like WaveCompletedListener for event handling instead of hardcoded callbacks.
    - Each class (Enemy, Tower, Projectile) is responsible for its own state and behavior, so the controller doesn't manage their internals directly.

**6. Polymorphism**

- **Definition**: When behavior varies by type, assign responsibility for the behavior using polymorphic operations.
- **In Code**:
    - Enemy is a base class, and Goblin, Knight are polymorphic subclasses.
    - The controller calls enemy.update(deltaTime) or enemy.render(gc) without knowing the exact enemy type.

**7. Indirection**

- **Definition**: Assign a responsibility to an intermediate object to mediate between other components to reduce direct coupling.
- **In Code**:
    - The WaveCompletedListener interface acts as an **indirection layer** between the game loop and UI components that may listen to wave-completion events.

**8. Pure Fabrication *(potentially applicable depending on other parts of your codebase)***

- **Definition**: Assign a responsibility to a class that doesn't represent a concept in the problem domain but is created to achieve design goals like reuse or separation of concerns.
- **Possibly**: GameSettings is an example if it's used purely to centralize configuration rather than being part of the game world model.

**Game Settings**

**1. Singleton**

- **Definition**: Ensures a class has only one instance and provides a global point of access to it.
- **Where in Code**:

private static GameSettings instance;

public static synchronized GameSettings getInstance()

    - The constructor is private.
    - A static instance is lazily initialized.
    - Global access is through getInstance().

**2. Information Expert**

- **Definition**: Assign responsibility to the class that has the necessary information to fulfill it.
- **Where in Code**:

- All the getter methods for tower stats, enemy stats, and settings:

public int getArcherTowerDamage()

public int getGoblinHealth()

public int getDifficulty()

- GameSettings is the “expert” because it holds the data and provides access to it.

**3. Creator**

- **Definition**: Assign the responsibility of creating an object to a class that has the information needed to create it.
- **Where in Code**:

private static GameSettings loadSettings()

- GameSettings is responsible for creating (or loading) its own instance from a file.

**4. Controller**

- **Definition**: Assign responsibility for handling system events to a controller object representing a use-case scenario.
- **Where in Code**:
  - While not a typical UI controller, GameSettings acts like a **"controller" of configuration state**:
    - It manages persistence (saveSettings() and loadSettings()).
    - It handles events like resetting defaults and applying difficulty settings.

**5. High Cohesion**

- **Definition**: Keep objects focused, manageable, and understandable. Each class should have a single, focused purpose.
- **Where in Code**:
  - GameSettings is cohesive in that it manages **only** game-related configuration, with no unrelated logic.

## Tower

### 1. Creator

**Definition:** Assign the responsibility of creating an object to a class that aggregates, contains, or closely uses the object.

- **Where:** Tower creates Projectile objects in the update() method via createProjectile(target).
- **Explanation:** Since a tower "owns" the logic of firing projectiles, it makes sense that it is responsible for creating them.

### 2. Information Expert

**Definition:** Assign a responsibility to the class that has the necessary information to fulfill it.

- **Where:**
    - findBestTarget() determines which enemy to target based on range and path progress.
    - isInRange() computes distance using the tower and enemy's positions.
    - getSellRefund() calculates the refund based on cost.
- **Explanation:** Tower has all the information it needs (position, range, fire rate, damage, etc.) to perform these tasks.

### 3. Controller

**Definition:** Assign the responsibility of handling input system events to a controller class that represents the overall system or a use-case scenario.

- **Where:** The update() method acts like a mini-controller by handling timing and triggering projectile creation.
- **Explanation:** While Tower isn't a system-level controller, within the domain of combat logic, it acts as a controller of its own firing behavior.

### 4. Polymorphism

**Definition:** Assign responsibility for behavior based on type to the types for which the behavior varies.

- **Where:** createProjectile() is an abstract method implemented differently in each concrete Tower subclass.
- **Explanation:** Each tower type may fire a different kind of projectile, so polymorphism allows this variation.

**5. Low Coupling**

**Definition:** Design classes to have low dependencies on each other.

- **Where:**
    - Tower uses interfaces or abstract types like Enemy, Projectile, and DamageType without depending on their internal implementation.
- **Explanation:** The Tower class interacts with external classes in a loosely coupled manner, allowing flexibility and reuse.

**6. High Cohesion**

**Definition:** Keep objects focused, manageable, and understandable by assigning closely related responsibilities.

- **Where:** All methods in the Tower class relate to targeting, rendering, and managing firing behavior.
- **Explanation:** The class has a focused set of responsibilities related to combat towers.

From a general poinf of view, the identified GRASP patterns used in this Project:

**Information Experts**

- The GameController class is an expert in managing game state, handling game loop, and coordinating game entities
- The Map class is an expert in managing map data and tile information
- The Player class is an expert in managing player-specific data like gold and hit points

**Creators**

- The ScreenFactory class is responsible for creating different types of screens (MainMenuScreen, GameScreen, etc.)
- The WaveFactory class creates wave configurations and group compositions
- The GameController creates game sessions and manages game entities

**Controllers**

- The GameController acts as a controller for the game logic and flow
- The MapEditorController handles map editing operations
- The OptionsController manages game options and settings

**Low Couplings**

- The project uses interfaces like Updateable to reduce coupling between components
- The ResourceManager provides a centralized way to access resources, reducing direct dependencies
- The FileManager abstracts file operations, reducing coupling to specific file system implementations

**High Cohesions**

- The Tile class has a single responsibility of managing tile properties
- The Tower class and its subclasses (ArcherTower, ArtilleryTower, MageTower) have focused responsibilities
- The Enemy class and its subclasses handle enemy-specific behavior

**Polymorphisms**

- Different types of towers (ArcherTower, ArtilleryTower, MageTower) implement polymorphic behavior
- Different types of projectiles (Arrow, ArtilleryShell, Spell) demonstrate polymorphic behavior

- Different types of enemies (Goblin, Knight) show polymorphic characteristics