**Project Title: Asteroids Game**

**Use Case Description**

The **Asteroids Game** is an arcade-style space shooter where the player controls a spaceship (represented by the Player class) navigating through an asteroid field. The player must shoot and destroy asteroids (represented by the Asteroids class) to survive. As asteroids break apart, smaller fragments are generated, creating an ongoing challenge. The game will keep track of the player's health and survival time, providing a game-over screen when health reaches zero. The user can interact with the game through keyboard controls and visuals rendered on the screen.

**Class List**

**1. Player**

* **Purpose**: Represents the player-controlled spaceship with attributes such as position, health, speed, and shooting mechanics.
* **Responsibilities**:
  + Movement and rotation based on user input.
  + Shooting projectiles.
  + Managing health and collisions with asteroids.
  + Rendering the player’s ship on the screen.

**2. PlayerProj**

* **Purpose**: Represents the projectiles shot by the player to destroy asteroids.
* **Responsibilities**:
  + Manage projectile speed, direction, and collision detection with asteroids.
  + Render projectiles.
  + Handle projectile lifetime (i.e., disappear when off-screen or after hitting an asteroid).

**3. Projectiles**

* **Purpose**: Abstract base class for all projectiles in the game (e.g., PlayerProj).
* **Responsibilities**:
  + Defines common attributes like position, speed, and damage.
  + Declares virtual methods for movement, rendering, and collision detection, to be implemented by subclasses.

**4. Enemy**

* **Purpose**: Abstract base class for all enemies (e.g., asteroids).
* **Responsibilities**:
  + Defines common attributes such as position, speed, and health.
  + Declares virtual methods for movement, rendering, and collision detection with the player.
  + Acts as the parent class for Asteroids.

**5. Asteroids**

* **Purpose**: Represents an asteroid in the game, with different sizes and behaviors (splitting into smaller asteroids).
* **Responsibilities**:
  + Movement and collision with the player and projectiles.
  + Splitting into smaller asteroids when destroyed.
  + Rendering asteroids on the screen.

**Data and Function Members**

**Player**

* **Attributes**:
  + pl\_pos: The player's position on the screen (Vector2).
  + pl\_health: The player's remaining health (int).
  + pl\_speed: The player's movement speed (Vector2).
  + projectiles: A vector to store active PlayerProj objects.
* **Functions**:
  + movement(): Controls player movement based on user input.
  + shoot(): Fires projectiles from the player’s position.
  + takeDamage(): Decreases player health upon collision.
  + draw(): Renders the player’s spaceship.

**PlayerProj**

* **Attributes**:
  + proj\_pos: The projectile’s position (Vector2).
  + proj\_speed: The projectile’s speed (Vector2).
  + active: Boolean to check if the projectile is still active.
* **Functions**:
  + update(): Updates the position of the projectile.
  + EnemyCollisionCheck(): Checks if the projectile collides with an enemy.
  + draw(): Renders the projectile on the screen.

**Projectiles (Abstract Class)**

* **Attributes**:
  + proj\_pos: Position of the projectile (Vector2).
  + proj\_speed: Speed of the projectile (Vector2).
  + proj\_damage: Damage dealt by the projectile (int).
* **Functions**:
  + setProjPos(), setProjSpeed(), setProjDamage(): Setters for the attributes.
  + update(): Virtual method for updating the position of the projectile.
  + draw(): Virtual method for rendering the projectile.

**Enemy (Abstract Class)**

* **Attributes**:
  + enemy\_pos: Position of the enemy (Vector2).
  + enemy\_speed: Movement speed of the enemy (Vector2).
  + enemy\_health: Health of the enemy (int).
* **Functions**:
  + setEnemyPos(), setEnemySpeed(), setEnemyHealth(): Setters for the attributes.
  + update(): Virtual method for updating the position and state of the enemy.
  + draw(): Virtual method for rendering the enemy.

**Asteroids**

* **Attributes**:
  + asterScale: Size scaling factor for the asteroid (float).
  + asterDmg: Damage dealt by the asteroid (int).
  + Aster\_Bounding: Bounding box size for collision detection (Vector2).
* **Functions**:
  + movement(): Updates the asteroid’s position.
  + breakApart(): Splits the asteroid into smaller ones when destroyed.
  + checkColPlayer(): Checks for collision with the player.
  + draw(): Renders the asteroid.

**Relationships between Classes**

* **PlayerProj** inherits from **Projectiles**, allowing projectiles fired by the player to share common behavior with any future projectile types.
* **Asteroids** inherits from **Enemy**, which provides basic movement, health, and collision logic, allowing future enemies to share common behavior with asteroids.
* **Player** has a **composition relationship** with **PlayerProj** by holding a vector of projectiles, enabling the player to fire multiple projectiles during the game.
* **Asteroids** and **Player** interact through collision detection, with **Asteroids** having functions to handle collisions with both **Player** and **PlayerProj**.

**Project Task List and Timeline**

1. **Design and Plan (2 Days)**:
   * Class structure design (Player, Projectiles, Enemy, Asteroids).
   * Plan relationships and interaction between game objects.
2. **Implement Base Classes (3 Days)**:
   * Write Projectiles and Enemy as abstract base classes.
   * Implement basic Player and PlayerProj functionality.
3. **Asteroids Class Development (3 Days)**:
   * Implement movement, collision detection, and break-apart mechanics.
4. **Game Loop and Interaction (2 Days)**:
   * Implement game loop with player controls and interactions (shooting, moving, asteroid spawning).
5. **Testing and Debugging (2 Days)**:
   * Write unit tests for individual classes and run gameplay tests for collisions, movement, and game state.
6. **Final Debugging and Refinement (2 Days)**:
   * Polish user interaction, game-over screen, and UI.

**User Interaction Description**

The user interacts with the game using keyboard controls:

* **W, A, S, D**: Control player movement.
* **SPACE**: Fire projectiles.
* **R**: Restart the game after a game-over.

The game displays the player's health, the number of active projectiles and asteroids, and the time survived. When the player's health reaches zero, a "Game Over" screen is shown with the option to restart.

**Unit Testing and Debugging Plan**

* **Unit Tests**:
  + Test individual methods like shoot(), movement(), takeDamage() for the Player class.
  + Test projectile collision handling in the PlayerProj class.
  + Ensure correct asteroid splitting and collision detection in the Asteroids class.
* **Gameplay Tests**:
  + Run full gameplay tests to ensure that the game loop functions correctly, with proper collision handling, health updates, and projectile management.
* **Debugging**:
  + Use error logging to capture unexpected behavior.
  + Test with various input cases (e.g., multiple simultaneous projectiles, rapid player movement).