Computer Science Space Shooter Project Documentation – Kayla Cheung

### Deliverable 1 – Project Proposal

1. **My Challenge and Game Option**

* My challenge is to develop a fast-paced space shooter game where the player pilots a spaceship through waves of enemies
* The game challenges the player with increasing enemy difficulty, dynamic obstacles, and engaging controls
* I chose this game because I used to be obsessed with it when I was year 3

1. **Justification for Choosing “Space Shooter”**

* **Engaging Game:**
  + It is a fast-paced action and exciting game that a lot of teenagers nowadays love
  + Encourages **real-time input handling** and **collision detection**
* **Challenges and Complexity:**
  + Start with basic mechanics (movement, shooting)
  + Incrementally add features like **enemy and power-ups** to increase difficulty of levels
* **Allows rich OOP Integration:** 
  + Allows me to utilize object-oriented programming concepts: inheritance, encapsulation, polymorphism, abstraction
  + I can create a base Enemy class with specialized subclasses (e.g., Alien, Asteroid) to illustrate OOP principles
* **Educational** 
  + The game allows me, as a Year 12 computer science student, to have a better in-depth understanding of different complex OOP concepts

1. **Problem Definition**

* To design and implement a modular space shooter game using Python
* Must ensure smooth controls, varied enemy behaviours, collision detection, scoring, and health mechanics
* Must complete the project over an 11‑week period with clear incremental milestones

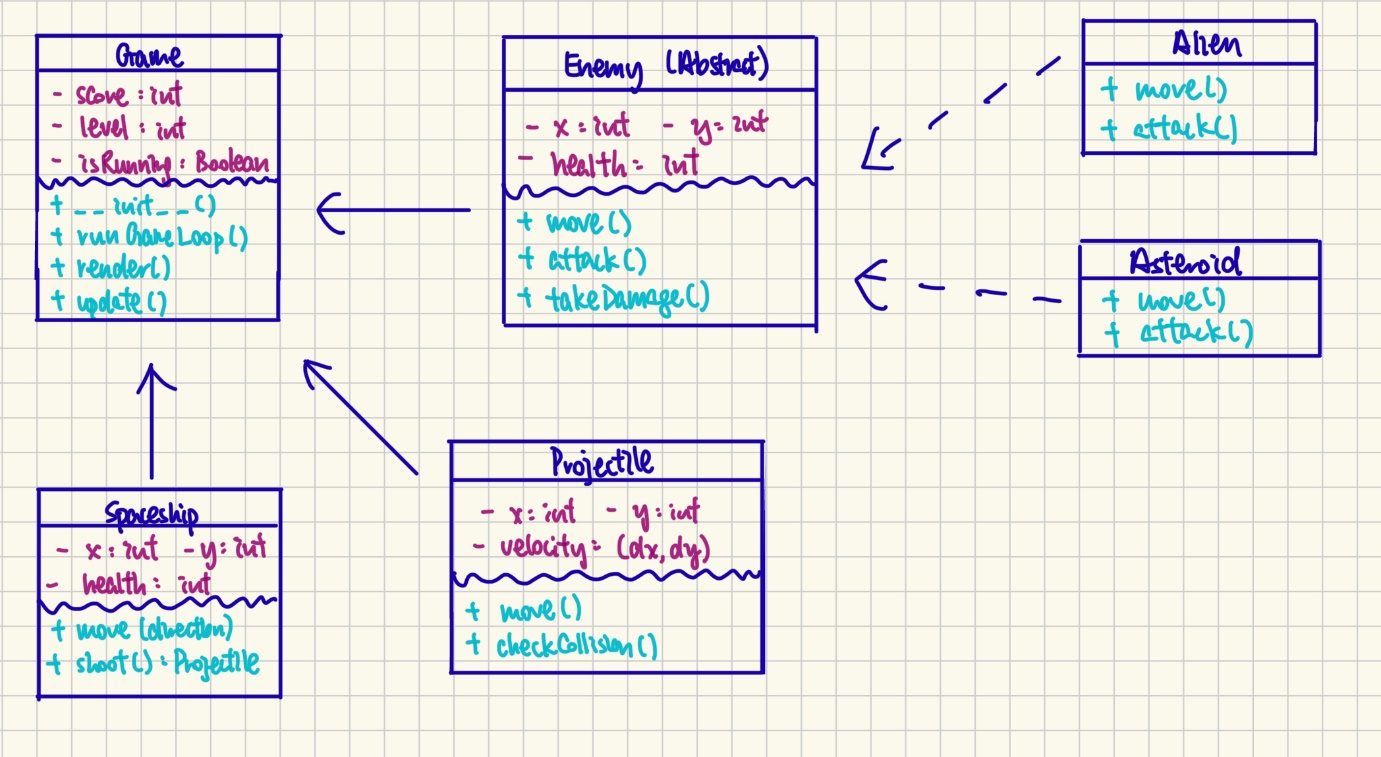
1. **Target Audience**

* **Gamers:** Casual gamers interested in shooting games
* **Computer Science Teachers/ Students:** this game allows students to understand OOP concepts in great detailed 🡪 a great educational material
* **Investors:** Companies that are interested in purchasing this game

1. **Objectives**

* To develop engaging gameplay mechanics (controls, enemy patterns, level progression)
* Must apply and demonstrate OOP concepts
  + **Inheritance and Polymorphism:** utilize classes, subclasses, methods and attributes
  + **Encapsulation**: hide internal states (e.g., health, position) and expose behaviours via methods
  + **Abstraction:** define abstract base classes to show only important/ necessary details

1. **Rough Class Diagram**

****

1. **Encapsulation Example**

* **Game Class:** 
  + Holds global variable such as score, level, and game state (isRunning)
  + Provides methods to initialize, update, and render the game **without exposing internal state details** (hide internal details 🡪 encapsulation)
* **Spaceship Class:** 
  + Manages its own position and health internally
  + Use methods like move() and shoot() so other parts of the game can interact with it **without knowing its internal structure**
* **Projectile Class:** 
  + Encapsulates its position and velocity/ speed
  + Handles its own movement and collision checking, **keeping internal logic hidden**
* **Enemy Class (&Subclasses):** 
  + Maintain their own properties (position, health) and provide methods (move, attack, takeDamage)
  + The internal implementation is **hidden**; the game engine only calls the public methods

1. **Inheritance Example**

* **Abstract Enemy Class:** 
  + A blueprint for all enemy types
  + Contains common attributes (e.g., position, health) and methods (e.g., move(), attack(), takeDamage())
* **Alien Class:** 
  + Inherits from the abstract Enemy class
  + Automatically gains common attributes and behaviours from Enemy
  + Overrides move() and attack() to implement alien-specific behaviors (e.g., zigzag movement or diving attacks)
* **Asteroid Class:**
  + Also inherits from the abstract Enemy class
  + Reuses shared attributes and methods defined in Enemy
  + Provides its own implementations of move() and attack() to reflect its unique movement (e.g., drifting or rotating) and impact behaviour

1. **Polymorphism Examples**

* **Enemy (Abstract) Class**
  + Defines common attributes (position, health) and abstract methods (move() and attack())
  + Forces subclasses (Alien and Asteroid) to implement these methods
  + A list or some methods named enemies will hold objects of both Alien and Asteroid
  + In the game loop, each enemy (Alien, Asteroid and potentially Boss (haven’t added)) is treated as an instance of Enemy, so calling enemy.move() and enemy.attack() **executes the specific implementation in each subclass**

1. **Improvements Made to Game and WHY**

* Initially, the Spaceship class I designed was responsible for movement, shooting, health management, and score tracking
* I realised this class holds too many methods and that some of them are irrelevant to each other
* Therefore, **improvements were made**
  + Moved score tracking from **the Spaceship class** to **the Game class**, so the Spaceship focuses solely on movement and shooting
  + Introduced **an abstract Enemy class**, and derived specialized enemy classes (Alien and Asteroid) from it
  + This allowed shared enemy attributes (like position and health) to be defined once, reducing code duplication and making future enhancements/ debugging easier
* This new design uses clear separation of responsibilities, ensuring each class handles its specific role (e.g., **Game for overall control, Spaceship for player actions**) 🡪 Good Programming Practice!
* Simplifies maintenance and reduces bugs
* Inheritance was used to centralize common enemy behaviour, making it straightforward to add new enemy types while keeping the code organized and scalable (enemy class and subclasses

1. **Game Layout Sketch**

A grid with yellow writing on it

AI-generated content may be incorrect.

NO REPEATING FUNCTION PLS