

Deliverable 2 – Kayla Cheung

1) Review My Initial Proposal

Question Section:

- I have clearly stated the game option (space shooter) and its justification.
- The initial class diagram (with a base Enemy class and specialized subclasses like Alien and Asteroid) is still a strong foundation.
- However, new ideas such as adding boss phases, dynamic projectile mechanics, and multiple enemy wave spawns require refinement.

Reflection Section:

- In my initial design, each class has only ONE specific responsibility (Game for overall flow, Spaceship for player actions etc), which helps maintenance.
- The use of polymorphism and inheritance was also found.
- The initial design, however, needs more detail regarding the boss phase. Specifically, how many bosses appear per level, their unique health bar, and attack behaviours (e.g., blood projectiles)
- The logic for regenerating enemy waves (and how new rows are spawned regardless of existing enemies) required clarification and refinement as well.

2) Detailed Pseudocode

- **Main Game Loop (run() method):**

```
1. FUNCTION run():  
#main game loop  
2.   INITIALSE clock  
3.  
4.   WHILE game is running:  
5.     CALL handle_events()  
6.     CALL update_game_state()  
7.     CALL draw()  
8.     LIMIT frame rate to FPS  
9.   END WHILE  
10. END FUNCTION
```

- **Encapsulation:** I have ensured that the run() method does not modify game objects directly but calls encapsulated methods (handle_events, update_game_state, draw) to keep logic separate and modular
- **Abstraction:** I have hidden unnecessary implementation details from the main game loop by using method calls (instead of just placing all my codes in the method)
- **Error Handling:** I have implemented exception handling in sub-methods (not in this main game loop), such as handle_events(), ensuring that errors do not crash the entire game.
- I can use **unit testing** for handle_events(), update_game_state(), and draw() separately
- **Frame rate limiting** ensures smooth rendering and consistent game behaviour

- **Collision Detection (check_collisions() method):**

```
1. FUNCTION check_collisions():  
2. #Check for collisions between projectiles, enemies, and the player  
3.   FOR each projectile IN projectiles:  
4. # Check if projectiles hit enemies  
5.     FIND enemies_hit WHERE projectile collides with enemies  
6.     FOR each enemy IN enemies_hit:  
7.       DESTROY projectile  
8.       REDUCE enemy health  
9.       IF enemy health <= 0:  
10.        REMOVE enemy  
11.        IF enemy is boss:  
12.          REMOVE from boss list  
13.          STOP boss attack timer  
14.          INCREASE score #Increase score when enemy is killed  
15.        END IF  
16.     END FOR  
17.   IF player collides with boss blood:  
18.     DECREASE player lives BY 1  
19.   END IF  
20.  
21.   IF player collides with enemies:  
22.     DECREASE player lives BY 1  
23.   END IF  
24. END FUNCTION
```

- **Encapsulation:** I have designed this method to only check for collisions, while the logic for enemy movement, shooting, and health management is encapsulated elsewhere (one logical task per function → good programming practice)
- **Polymorphism:** Different types of game objects (projectiles, enemies, bosses) respond to collisions in different ways

- **Error Handling:** I have included error handling by verifying object existence before performing collision checks, such as ensuring a projectile exists before attempting to access its properties, to prevent crashes
- **Unit Testing:** I can test collision outcomes using mock objects
- **Boundary Testing:** I must ensure that collisions are detected correctly at screen edges and object boundaries

- **Input Handling (handle_events() method):**

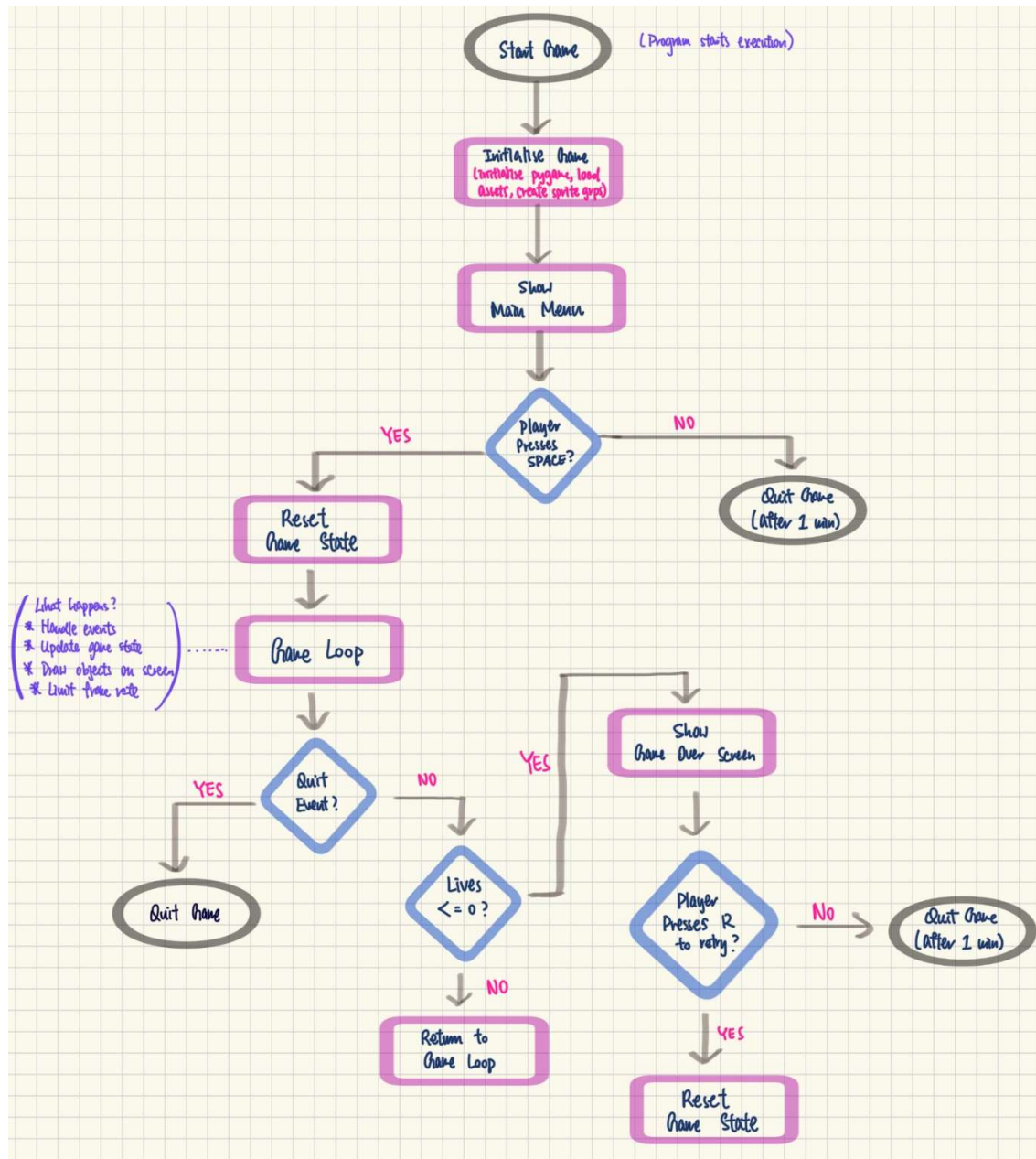
```

1. FUNCTION handle_events()
2.   FOR each event in pygame.event.get()
3.     TRY:
4.       IF event.type == pygame.QUIT THEN
5.         CALL quit_game()
6.
7.       ELSE IF event.type == SPAWN_EVENT THEN
8.         CALL spawn_enemy_wave()
9.
10.      ELSE IF event.type == pygame.KEYDOWN THEN
11.        CALL handle_keydown(event)
12.
13.      ELSE IF event.type == BOSS_BLOOD_EVENT THEN
14.        CALL handle_boss_blood()
15.      END IF
16.
17.    CATCH Exception AS e:
18.      PRINT "Error handling event: ", e
19. END FUNCTION

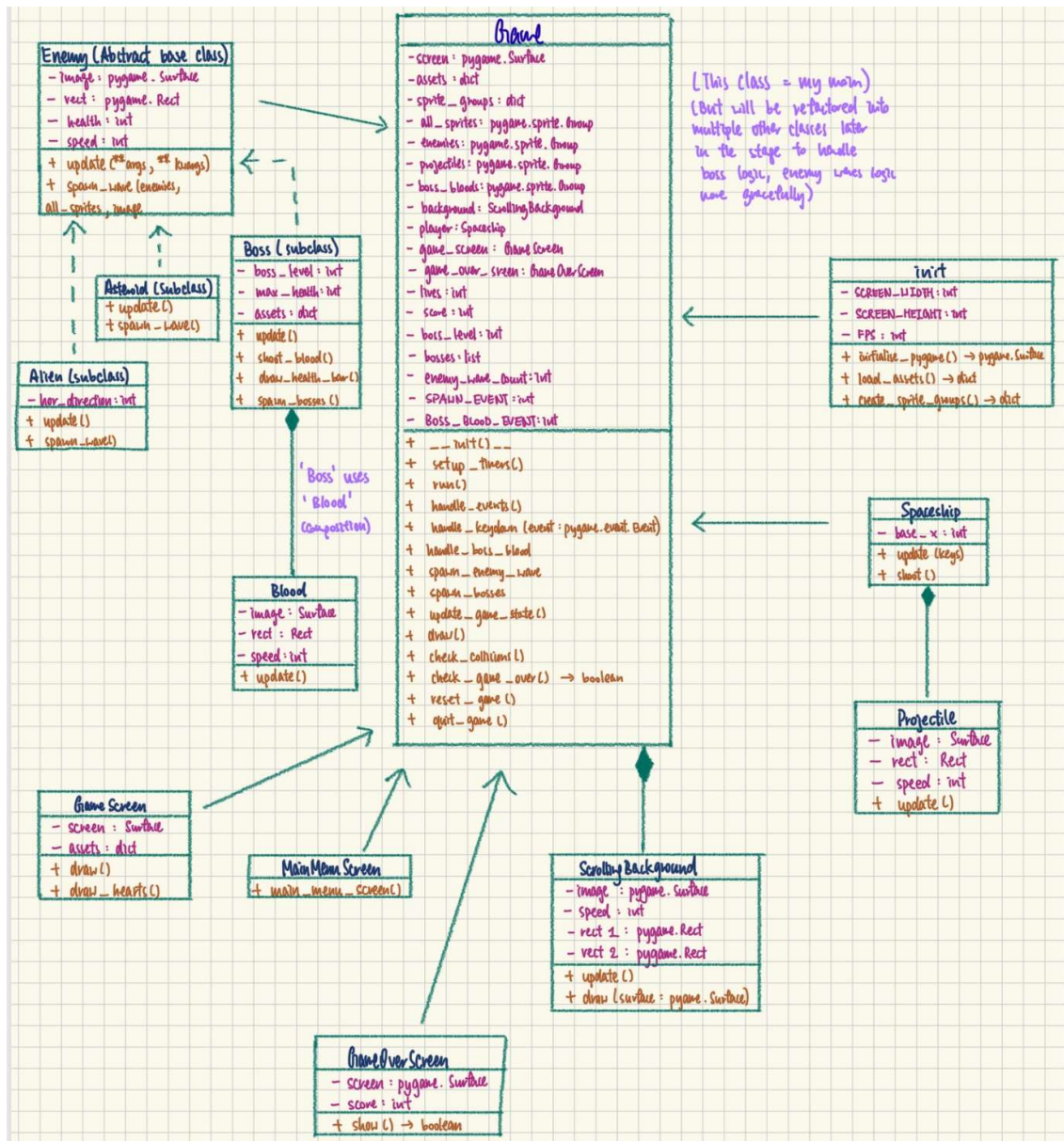
```

- **Encapsulation:** I have structured the handle_events() method to process user inputs and game events without directly modifying game objects. Instead, I delegate specific tasks to well-defined methods like quit_game(), spawn_enemy_wave(), and handle_keydown(), keeping the event-handling logic modular and maintainable
- **Abstraction:** I have hidden event-handling details by, again, using method calls. This makes the code easier to read and modify, as I don't need to worry about how enemies spawn or how key presses are processed within this method.
- **Polymorphism:** The method accommodates different types of events (QUIT, SPAWN_EVENT, KEYDOWN, BOSS_BLOOD_EVENT) by calling appropriate methods. Each event type triggers a different behaviour, demonstrating how objects (such as enemies or the player) respond differently to user inputs and game events
- **Error Handling:** I have included a try-except block to catch and print errors that might occur while processing events. This prevents unexpected crashes and makes debugging easier by logging errors instead of stopping the game abruptly
- **Unit Testing:** I can test this method by simulating different event inputs and verifying that the correct methods are called
- **Mocking Events:** By using mock objects for pygame.event.get(), I can ensure that various game events are handled as expected without requiring real user input
- **Exception Testing:** I can intentionally introduce faulty events to check whether errors are caught and logged properly

3) Flowchart



4) Refined Class Diagram



5) System Architecture and Module Interaction

Space Shooter Game Project/

Assets/

Images/

Alien.png
Asteroid.png
Background.png
Blood.png
Boss.png
Heart.png
Projectile.png
Spaceship.png

Sounds/

(sound files here but I haven't added sound to my game)

Source Code/

Screens/

Game_over_screen.py
Game_screen.py
Main_menu_screen.py
Scrolling_background.py

Init.py (Contains configuration settings, initialisation functions, etc.)

Alien.py (Enemy subclass for aliens)

Asteroid.py (Enemy subclass for asteroids)

Blood.py (Contains the Blood class (boss attack projectile))

Boss.py (Contains the Boss class with health, blood attack methods, etc)

Enemy.py (Abstract enemy class used for inheritance)

Game.py (Main game engine file (the game loop, state management, collision detection, etc.)

Projectile.py (Contains the Projectile class)

Spaceship.py (Contains the Spaceship class)

6) Design Decisions: Error Handling and Testing Strategies

- **Error Handling**

- I use try-except blocks to catch runtime exceptions (e.g., division by zero, invalid operations)
- If a critical error occurs, I ensure the program exits gracefully with an error message

- **Invalid User Input**

- I validate user inputs (e.g., checking if the key press is valid) before processing
- If input is invalid, I use default values or prompt the player to retry

- **File I/O Errors**

- Before reading/writing files, I check if the file exists using `os.path.exists`.
- If an asset file (e.g., image) is missing, I use a fallback (e.g., a solid colour or simple shape)

- **Testing Strategies – Unit Tests (Testing individual component)**

- I use unittest for writing and running individual unit tests
- For example, I test if `Spaceship.shoot()` creates a projectile

```
1. import unittest
2. import pygame
3. from spaceship import Spaceship
4. from projectile import Projectile
5.
6. class TestSpaceshipShoot(unittest.TestCase):
7.     def setUp(self):
8.         pygame.init()
9.         # Create a mock spaceship image
10.        self.mock_spaceship_image = pygame.Surface((50, 50))
11.        # Create a mock projectile image
12.        self.mock_projectile_image = pygame.Surface((10, 10))
13.        # Create a Spaceship instance
14.        self.spaceship = Spaceship(x=100, y=200, speed=5, image=self.mock_spaceship_image)
15.        # Create sprite groups
16.        self.all_sprites = pygame.sprite.Group()
17.        self.projectiles = pygame.sprite.Group()
18.
19.    def test_shoot_creates_projectile(self):
20.        # Call the shoot method
21.        self.spaceship.shoot(self.all_sprites, self.projectiles, self.mock_projectile_image)
22.
23.        # Check if a projectile was added to all_sprites group
24.        self.assertEqual(len(self.all_sprites), 1) # Spaceship + Projectile
25.        # Check if a projectile was added to the projectiles group
26.        self.assertEqual(len(self.projectiles), 1)
27.
28.        # Verify the projectile is an instance of the Projectile class
29.        projectile = self.projectiles.sprites()[0]
30.        self.assertIsInstance(projectile, Projectile)
31.
32.        # Verify the projectile's position matches the spaceship's position
33.        self.assertEqual(projectile.rect.centerx, self.spaceship.rect.centerx)
```

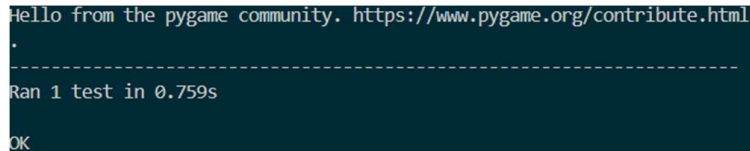


```

34.         # Account for the projectile's height when checking rect.top
35.         self.assertEqual(projectile.rect.top, self.spaceship.rect.top -
(projectile.rect.height // 2))
36.
37.     def tearDown(self):
38.         pygame.quit()
39.
40. if __name__ == "__main__":
41.     unittest.main()
42.

```

Result:



```

Hello from the pygame community. https://www.pygame.org/contribute.html
.
-----
Ran 1 test in 0.759s
OK

```

What this test achieves:

- Verifies that the Spaceship.shoot() method correctly creates a projectile
- Verifies the projectile's position matches the spaceship's position
- Tests the shoot method in isolation, without relying on other parts of the game (e.g., game loop, rendering)
- Confirms that the **shoot** method behaves as expected:
- Creates a projectile
- Adds it to the correct groups
- Positions it correctly relative to the spaceship
- Finds Bugs Early:
- Catches issues in the shoot method before they affect the rest of the game
- For instance, if the projectile is not created or added to the correct groups, the test will fail immediately

- **Testing Strategies – Integration Tests**

- Verify that the ScrollingBackground class works correctly within the Pygame system
- This allows me to confirm the background scrolling functionality, detect integration issues, and it provides visual feedback on behaviour

```

1. import pygame
2. from init import FPS, SCREEN_HEIGHT, SCREEN_WIDTH, load_assets
3. from screens.scrolling_background import ScrollingBackground
4.
5. def test_background():
6.     pygame.init()
7.     screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
8.     clock = pygame.time.Clock()
9.     assets = load_assets()
10.    background = ScrollingBackground(assets["background"], speed=3)

```



```
11.  
12.     running = True  
13.     while running:  
14.         for event in pygame.event.get():  
15.             if event.type == pygame.QUIT:  
16.                 running = False  
17.  
18.                 background.update()  
19.                 background.draw(screen)  
20.                 pygame.display.flip()  
21.                 clock.tick(FPS)  
22.  
23.     pygame.quit()  
24.  
25. test_background()  
26.
```

What it helps determine:

- Whether the background scrolls correctly
- Whether the background resets correctly
- Whether the background is rendered correctly
- Ensures the ScrollingBackground class integrates correctly with Pygame's display and rendering system
- Detects issues that might not appear in unit tests (e.g., rendering errors, incorrect scrolling behaviour)

7) Additional Design Considerations

Ethical and Legal Considerations:

I ensure all game assets are self-created, avoiding any copyright infringements/plagiarism. Fair gameplay is maintained by designing balanced mechanics that do not exploit or mislead the player. I also upheld code of conduct throughout the development process.

Reflection on Design Choices:

I chose a modular design with clear separation of concerns, using OOP principles such as inheritance, encapsulation, and polymorphism to keep each class focused on a single responsibility. This approach facilitates future improvements and debugging by isolating functionality in discrete modules, allowing me to extend or modify features without impacting the core game mechanics.