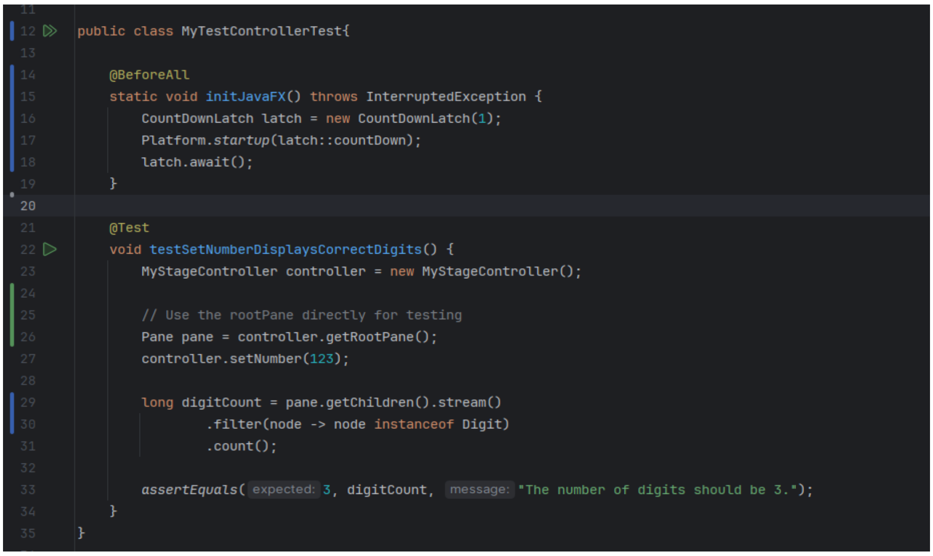
Task 2

- Sub-Task 2.1

To demonstrate test-driven refactoring for the MyStageController class, you can follow these steps:

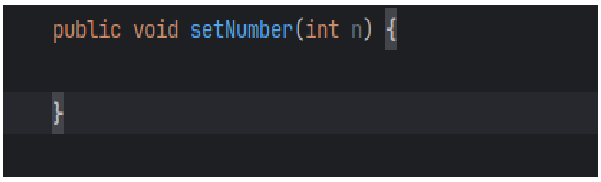
1. **Start with a failing test**: Write a test for a method that doesn't exist yet, or one that is not implemented properly.
2. **Implement the method**: Write the code to make the test pass.
3. **Refactor the code**: Once the test passes, clean up the code to improve its structure while ensuring the test still passes.

The test ensures the setNumber() method correctly adds the right number of Digit objects based on the given integer. This guarantees the game displays the correct score. The test-driven refactoring process started with a failing test, followed by implementing a stub method, and then completing the method to pass the test. This approach improves maintainability by verifying the correctness of game logic and ensuring consistent visual updates.



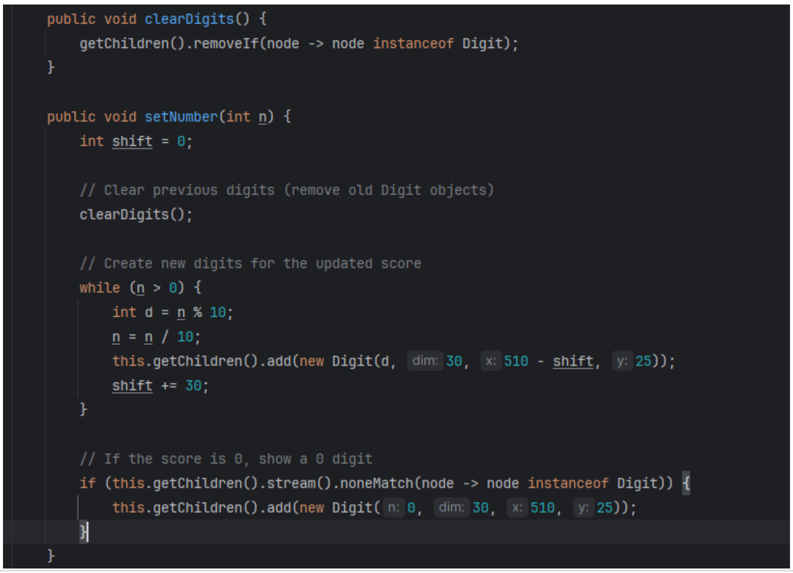
```
public class MyTestControllerTest{

    @BeforeAll
    static void initJavaFX() throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(1);
        Platform.startup(latch::countDown);
        latch.await();
    }

    @Test
    void testSetNumberDisplaysCorrectDigits() {
        MyStageController controller = new MyStageController();

        // Use the rootPane directly for testing
        Pane pane = controller.getRootPane();
        controller.setNumber(123);

        long digitCount = pane.getChildren().stream()
                .filter(node -> node instanceof Digit)
                .count();

        assertEquals( expected: 3, digitCount,  message: "The number of digits should be 3.");
    }
}
```

Figure 1 Test Case



```
public void setNumber(int n) {

}
```

Figure 2 Initial Method



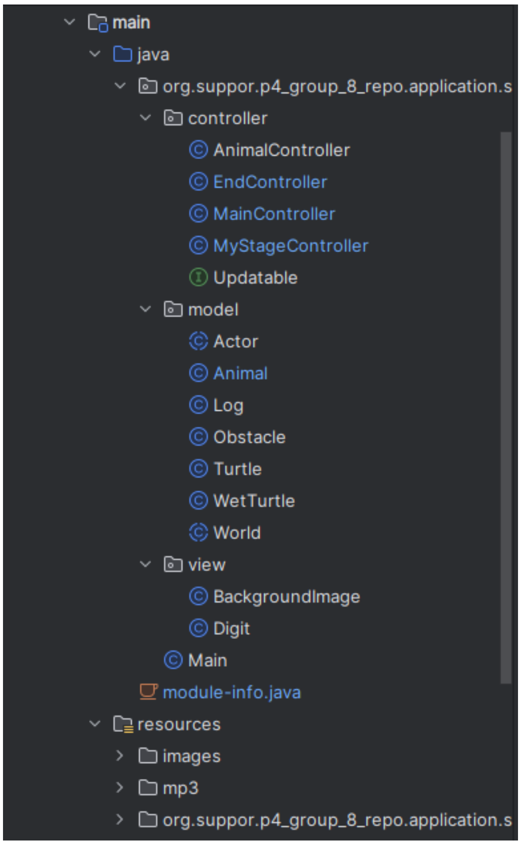
```
public void clearDigits() {
    getChildren().removeIf(node -> node instanceof Digit);
}

public void setNumber(int n) {
    int shift = 0;

    // Clear previous digits (remove old Digit objects)
    clearDigits();

    // Create new digits for the updated score
    while (n > 0) {
        int d = n % 10;
        n = n / 10;
        this.getChildren().add(new Digit(d,  dim: 30,  x: 510 - shift,  y: 25));
        shift += 30;
    }

    // If the score is 0, show a 0 digit
    if (this.getChildren().stream().noneMatch(node -> node instanceof Digit)) {
        this.getChildren().add(new Digit( n: 0,  dim: 30,  x: 510,  y: 25));
    }
}
```

Figure 3 Implemented Method

Task 3

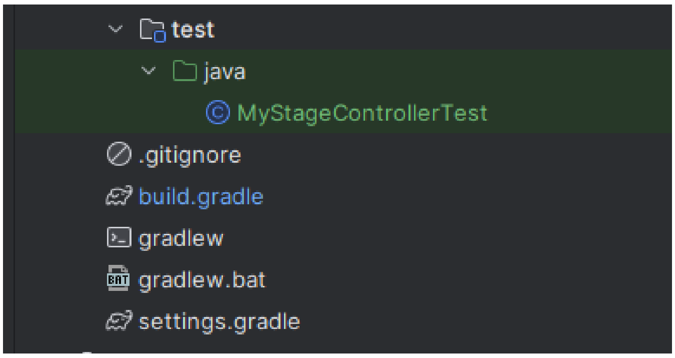
Maintain the software using refactoring techniques and principles at high level and low level

Sub-task 3.1: Refactoring at high level



```
main
  java
    org.suppor.p4_group_8_repo.application.s
      controller
        AnimalController
        EndController
        MainController
        MyStageController
        Updatable
      model
        Actor
        Animal
        Log
        Obstacle
        Turtle
        WetTurtle
        World
      view
        BackgroundImage
        Digit
      Main
    module-info.java
  resources
    images
    mp3
    org.suppor.p4_group_8_repo.application.s
```

Figure 4 Project Structure



```
test
  java
    MyStageControllerTest
.gitignore
build.gradle
gradlew
gradlew.bat
settings.gradle
```

Figure 5 Testing Package

1. Structured Code into Meaningful Packages

- After Refactoring:
  - The project is divided into three core packages:
    - controller: Manages user input, application logic, and communication between the view and model.
    - model: Represents the application's business logic, including entities like Animal, Turtle, and World.
    - view: Contains UI-related classes like BackgroundImage and Digit for graphical representation.

- Benefits:
    - Improved Organization: Developers can quickly navigate to the desired class.
    - Scalability: Adding new features or components becomes simpler with a well-structured package system.

2. MVC Architecture Implementation

- Before Refactoring:
    - The project lacked separation of concerns. UI, logic, and data were mixed in the same classes, leading to tightly coupled code.
- After Refactoring:
    - Applied MVC design pattern:
        - Model: Handles data and logic. For example:
            - Actor: Represents game entities with attributes and methods.
            - Obstacle and WetTurtle: Extends core functionality for specific entities.
        - View: Defines what the user sees. For example:
            - BackgroundImage creates graphical elements for the game.
        - Controller: Connects the view and model. For example:
            - MainController initializes the game, and AnimalController handles animal-specific actions.
- Benefits:
    - Separation of Concerns: Changes to the UI don't affect business logic and vice versa.
    - Testability: Each layer can be tested independently, improving test coverage and reliability.
    - Reusability: Models and controllers can be reused across different views or platforms.

3. Design Patterns

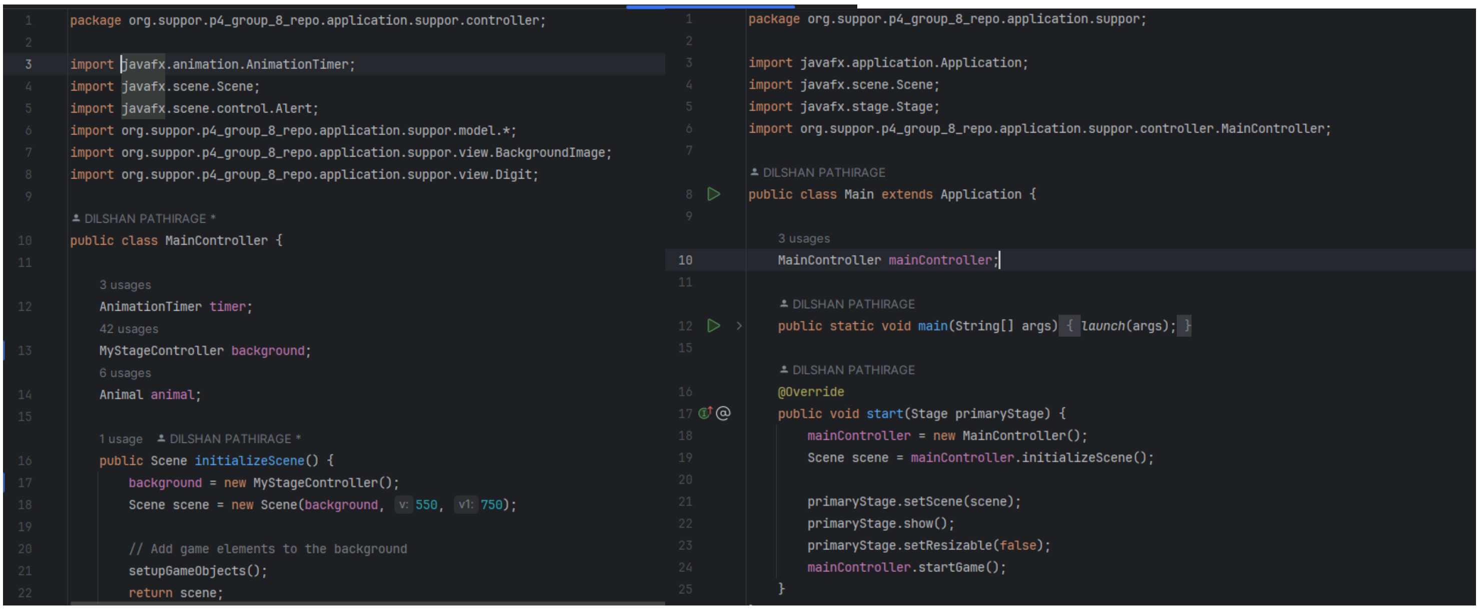
- Applied Patterns:
    - Singleton Pattern: For managing shared resources such as MyStageController.
        - Ensures only one instance of the game stage is active at any time.
        - Reduces the risk of resource conflicts.
    - Observer Pattern: (If applicable in your case) Observers can be used to notify views of model changes.

- Justification:
  - Reduced Redundancy: Centralized management of resources like the stage or event listeners.
  - Enhanced Maintainability: Patterns provide a clear structure, reducing the complexity of large-scale applications.

4. SOLID Principles

- Single Responsibility Principle:
  - Each class has a clear and single purpose.
  - Example: AnimalController handles animal-related actions, while MainController manages the game flow.
- Open/Closed Principle:
  - Classes like Animal and Obstacle are open for extension but closed for modification.
  - Example: New obstacles can be added by extending Obstacle without modifying the base class.
- Dependency Inversion Principle:
  - Controllers depend on abstractions rather than concrete implementations, improving flexibility and testability.
- Benefits:
  - Robust Codebase: SOLID principles lead to a more adaptable and future-proof design.
  - Ease of Debugging: Clear separation of responsibilities makes it easier to identify and fix bugs.

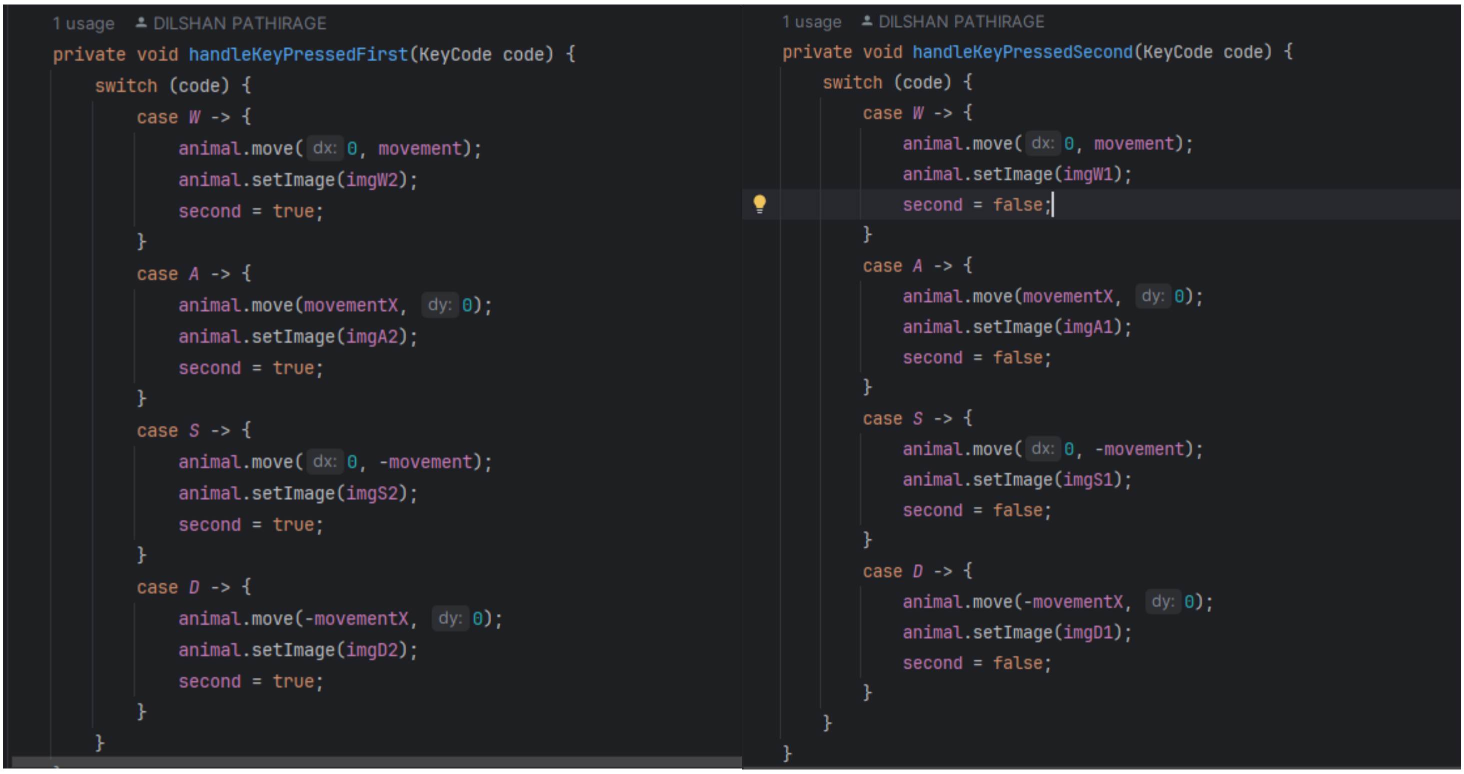
Sub-task 3.2: Refactoring activities at low level



```java
package org.suppor.p4_group_8_repo.application.suppor.controller;

import javafx.animation.AnimationTimer;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import org.suppor.p4_group_8_repo.application.suppor.model.*;
import org.suppor.p4_group_8_repo.application.suppor.view.BackgroundImage;
import org.suppor.p4_group_8_repo.application.suppor.view.Digit;

DILSHAN PATHIRAGE *
public class MainController {

    3 usages
    AnimationTimer timer;
    42 usages
    MyStageController background;
    6 usages
    Animal animal;

    1 usage   DILSHAN PATHIRAGE *
    public Scene initializeScene() {
        background = new MyStageController();
        Scene scene = new Scene(background,  v: 550,  v1: 750);

        // Add game elements to the background
        setupGameObjects();
        return scene;
```

```java
package org.suppor.p4_group_8_repo.application.suppor;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
import org.suppor.p4_group_8_repo.application.suppor.controller.MainController;

DILSHAN PATHIRAGE
public class Main extends Application {

    3 usages
    MainController mainController;

    DILSHAN PATHIRAGE
    public static void main(String[] args) { launch(args); }

    DILSHAN PATHIRAGE
    @Override
    public void start(Stage primaryStage) {
        mainController = new MainController();
        Scene scene = mainController.initializeScene();

        primaryStage.setScene(scene);
        primaryStage.show();
        primaryStage.setResizable(false);
        mainController.startGame();
    }
```

Figure 6 Main Class and MainController Class



```
1 usage  DILSHAN PATHIRAGE
private void handleKeyPressedFirst(KeyCode code) {
    switch (code) {
        case W -> {
            animal.move( dx: 0, movement);
            animal.setImage(imgW2);
            second = true;
        }
        case A -> {
            animal.move(movementX,  dy: 0);
            animal.setImage(imgA2);
            second = true;
        }
        case S -> {
            animal.move( dx: 0, -movement);
            animal.setImage(imgS2);
            second = true;
        }
        case D -> {
            animal.move(-movementX,  dy: 0);
            animal.setImage(imgD2);
            second = true;
        }
    }
```

```
1 usage  DILSHAN PATHIRAGE
private void handleKeyPressedSecond(KeyCode code) {
    switch (code) {
        case W -> {
            animal.move( dx: 0, movement);
            animal.setImage(imgW1);
            second = false;
        }
        case A -> {
            animal.move(movementX,  dy: 0);
            animal.setImage(imgA1);
            second = false;
        }
        case S -> {
            animal.move( dx: 0, -movement);
            animal.setImage(imgS1);
            second = false;
        }
        case D -> {
            animal.move(-movementX,  dy: 0);
            animal.setImage(imgD1);
            second = false;
        }
    }
}
```

Figure 7 Frogger Moving Controller Methods

1. Code Smells Eliminated

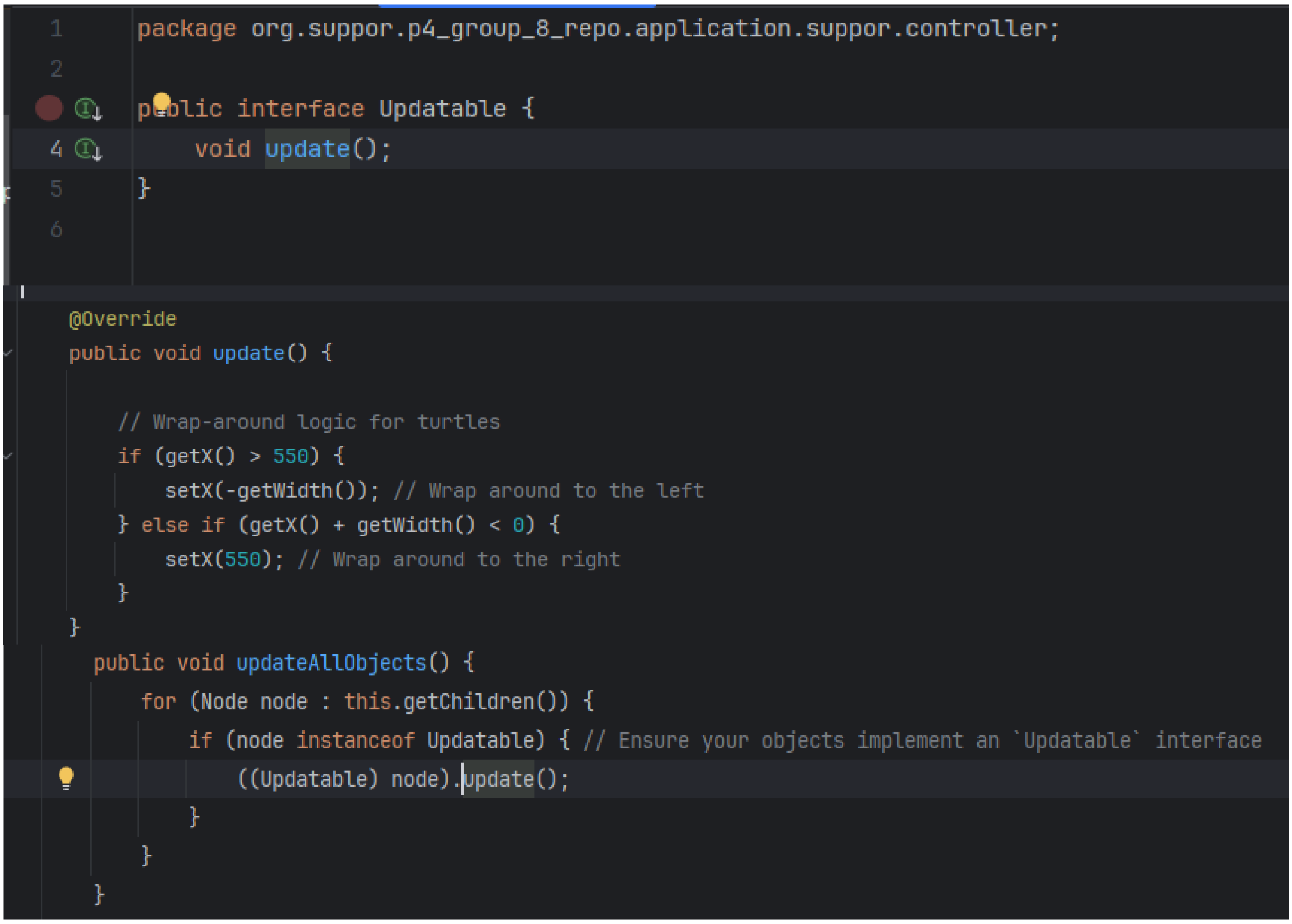
- Long Methods: Split long methods like handleKeyPressed into smaller methods (handleKeyPressedFirst, handleKeyPressedSecond).
- Duplicate Code: Removed repetitive instantiation of images by consolidating into a utility method or constructor.
- Excessive Parameters: Reduced parameter lists by introducing configuration objects or builders for game objects.

2. Coding Conventions

- Naming: Standardized class and method names (MainController instead of Main).
- Formatting: Ensured consistent indentation, spacing, and braces.

Task 4

- **Sub-task 4.1**



```java
package org.suppor.p4_group_8_repo.application.suppor.controller;

public interface Updatable {
    void update();
}
```

```java
@Override
public void update() {

    // Wrap-around logic for turtles
    if (getX() > 550) {
        setX(-getWidth()); // Wrap around to the left
    } else if (getX() + getWidth() < 0) {
        setX(550); // Wrap around to the right
    }
}
  public void updateAllObjects() {
      for (Node node : this.getChildren()) {
          if (node instanceof Updatable) { // Ensure your objects implement an `Updatable` interface
              ((Updatable) node).update();
          }
      }
  }
```

Figure 8 Reusable Method for Update All Objects

Addition: Enhanced Update Mechanism with Wrap-Around Logic

Implemented the Updatable interface to dynamically update objects and added wrap-around logic for nodes (e.g., turtles) to ensure smooth transitions when moving off-screen.

Justification:
Enhances user experience with seamless object movement and prevents objects from disappearing off-screen, ensuring consistent gameplay.

Explanation:
The Updatable interface ensures modular updates, allowing easy scalability for future components. Wrap-around logic maintains continuous motion, avoiding abrupt interruptions as objects cross screen boundaries.