Justification Document

48926782

Limitations

Many limitations were found which were improved upon to increase game clarity and functionality, and these are outlined below.

Limitation 1: When power ups apply effects, they print into the terminal unnecessarily.

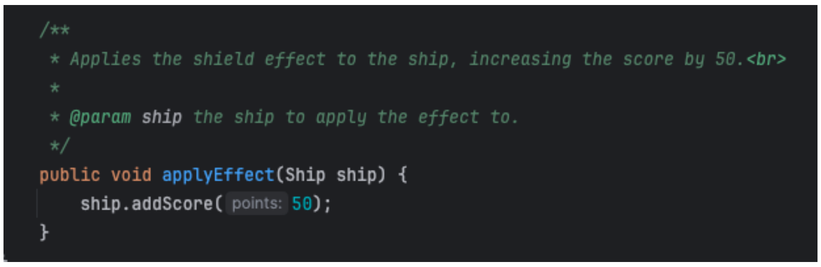
Game → Core → ShieldPowerUp
Game → Core → HealthPowerUp

Issue:
In the code for A1, the PowerUps log messsgaes to System.out. For example, the following messages were logged shield and health power up respectively.

System.out.println("Shield activated! Score increased by 50.");
System.out.println("Health restored by 20!");

This was unnecessary, as the terminal output is not used in gameplay, and violates the single responsibility principle, by placing the logging logic within the PowerUp classes, which should be focussed only on their role in the gameplay.

Fix:
To fix this issue, it was required to remove System.out.println("{relevant message}") from the Game.Core.ShieldPowerUp.applyEffect() and Game.Core.HealthPowerUp.applyEffect(). Below is an example of the new Game.Core.ShieldPowerUp.applyEffect() applyEffect method, which simply applies the effect without logging.



```
/**
 * Applies the shield effect to the ship, increasing the score by 50.<br>
 *
 * @param ship the ship to apply the effect to.
 */
public void applyEffect(Ship ship) {
    ship.addScore( points: 50);
}
```

Relevant Tests:
Two test classes were created to ensure this new logic had correctly been implemented.

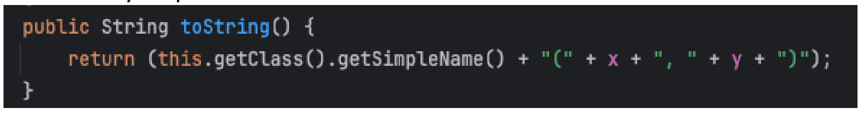
Test.Game.Core.HealthPowerUpTest:
- public void testHealingAmount() : tests that the health power up correctly performs the healing.
- public void testNoPrinting() : tests that the health power up does not print any message to terminal. To implement this, System.out was redirected into a new PrintStream so any output could be recorded.

Test.Game.Core.ShieldPowerUpTest:
- public void testScoreAmount() : tests that the score was correctly applied by the ShieldPowerUp.
- public void testNoPrinting() : mirrors logic in the HealthPowerUpTest.testNoPrinting() to ensure no messages were printed.

Limitation 2: Subclasses of ObjectWithPosition have no toString() method.

Game → Core → ObjectWithPosition

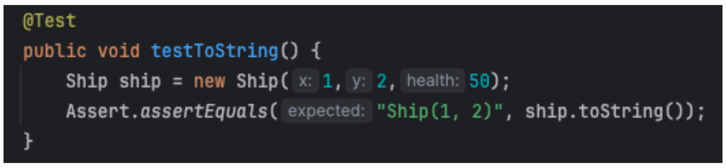
All subclasses which implement the ObjectWithPosition class originally had no toString() method, which is important for getting a string representation of each object.
Fix: addition of a toString() method in ObjectWithPosition which is inherited by all subclasses.
The following method was added to ObjectWithPosition, which represents each object by its name followed by its position.



```java
public String toString() {
    return (this.getClass().getSimpleName() + "(" + x + ", " + y + ")");
}
```

Relevant Test:
Test.Game.Core.ObjectWithPositionTest checks that the toString() method has been correctly implemented.

- public void testToString() instantiates a ship, and tests the toString method to ensure that it has been correctly implemented, taking the simple name of the class and giving the representation with its current coordinates.



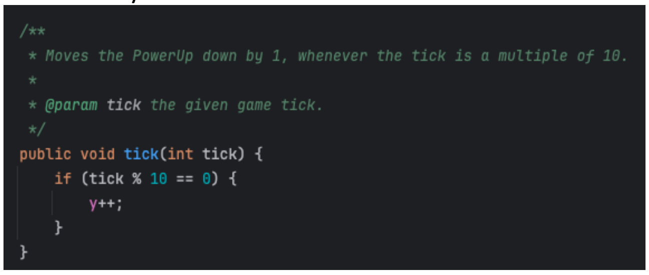
```java
@Test
public void testToString() {
    Ship ship = new Ship( x: 1, y: 2, health: 50);
    Assert.assertEquals( expected: "Ship(1, 2)", ship.toString());
}
```

Limitation 3: PowerUps remain at the top of the screen.

Game → Core → PowerUp

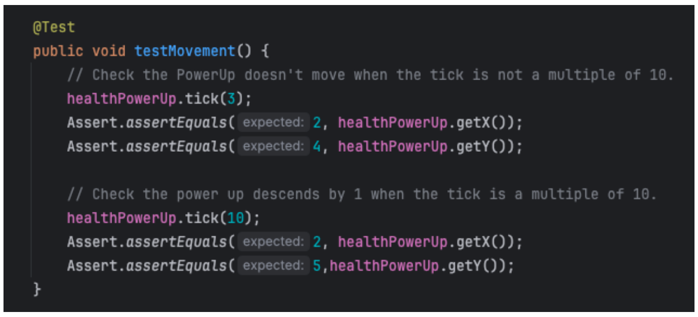
In A1, PowerUps had no tick dependent behaviour. This meant that they spawned at the top of the screen and would remain there for the duration of the game or until collected. This was not effective for gameplay as the inconvenience made them difficult to collect. Ideally, the PowerUps should spawn at the top and continue moving down, similarly to the enemies and asteroids.

Fix:
addition of tick dependent behaviour to Game.Core.PowerUp, which is inherited by all subclasses (Game.Core.HealthPowerUp and Game.Core.ShieldPowerUp).
Since all power ups have the same tick dependent behaviour (moving down the screen by one y value every time the tick is a multiple of 10), the following method was created for PowerUp, which can be inherited by all subclasses:



```java
/**
 * Moves the PowerUp down by 1, whenever the tick is a multiple of 10.
 *
 * @param tick the given game tick.
 */
public void tick(int tick) {
    if (tick % 10 == 0) {
        y++;
    }
}
```

Relevant tests:
Test.Game.Core.HealthPowerUpTest: Since the same tick-dependent behaviour is inherited by both subclasses, only one test needs to be made. Therefore, a test was added to this existing class, which ensures the PowerUp's y value changes correctly depending on the tick, and the x value does not change.



```
@Test
public void testMovement() {
    // Check the PowerUp doesn't move when the tick is not a multiple of 10.
    healthPowerUp.tick(3);
    Assert.assertEquals( expected: 2, healthPowerUp.getX());
    Assert.assertEquals( expected: 4, healthPowerUp.getY());

    // Check the power up descends by 1 when the tick is a multiple of 10.
    healthPowerUp.tick(10);
    Assert.assertEquals( expected: 2, healthPowerUp.getX());
    Assert.assertEquals( expected: 5,healthPowerUp.getY());
}
```

Limitation 4: When the game ends, there is no detection or handling of this process.

Game → GameModel
Game → GameController

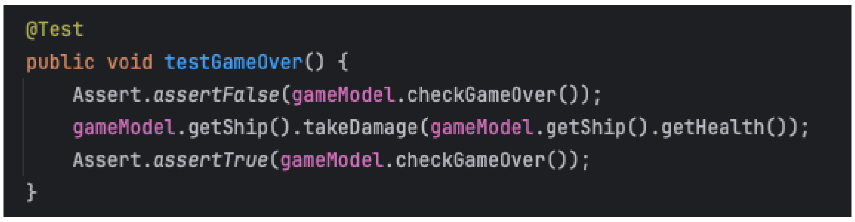
Previously, when the game would end (i.e. the ship would lose all its health) the game would keep running, as there was no method to check when this occurred or perform relevant actions when it did. This was extremely ineffective, as the player often may not notice when they died.

Fix:

- A Game.GameModel.checkGameOver() method was added. This method returns true when the boats health is less than or equal to 0.
- Furthermore, a Game.GameController.showGameOverWindow() which handles the UI, displaying a new screen which informs the player the game has ended and displays relevant statistics
- Game.GameController.onTick() was updated to implement these two methods. It now checks whether the game is over every tick, and if so updates the game over window.

Relevant Tests:

- Tests.Game.GameModelTests.testCheckGameOver(): this tests checks that the check game over method is functioning correctly, and is seen below. This first checks that the newly initialised game is not over and then once the ship has taken full damage (its health) the game should be over as the health reaches 0.



```
@Test
public void testGameOver() {
    Assert.assertFalse(gameModel.checkGameOver());
    gameModel.getShip().takeDamage(gameModel.getShip().getHealth());
    Assert.assertTrue(gameModel.checkGameOver());
}
```

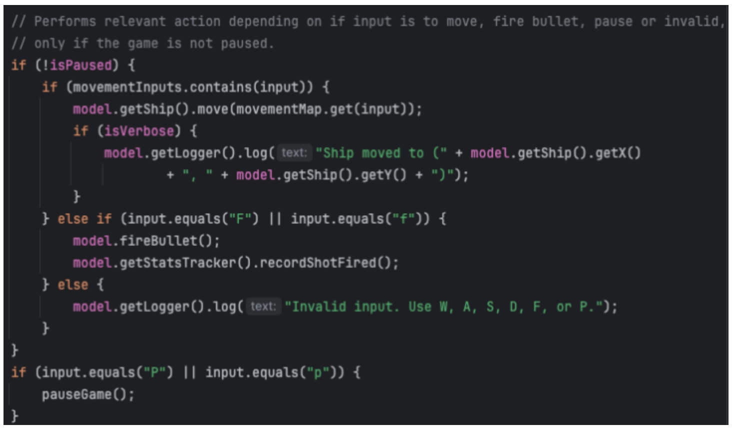
- The showGameOverWindow() is not tested as the UI handling cannot be reasonably tested within the scope of this assignment.

Limitation 5: Character input is still allowed when the game is paused.

Game → GameController

When the game is paused, players were still able to move around the screen like usual, as the Game.GameController.handlePlayerInput(String input) function did not check for pausing before performing movement actions. This is ineffective, meaning the game state can be modified during a paused time, which clearly leads to gameplay discontinuities.

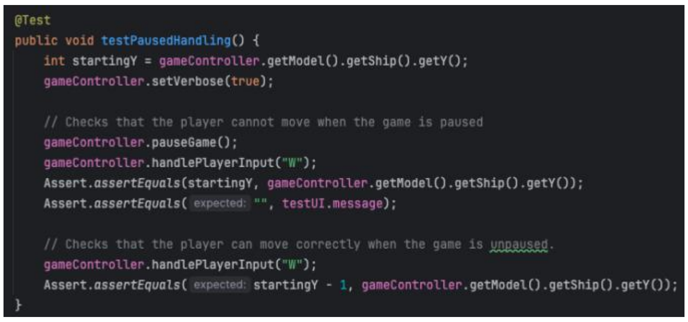
Fix: The GameController uses a field isPaused to monitor whether the game is paused or not. In Game.GameController.handlePlayerInput(String input), all of the actions which should not be possible if the game is paused are surrounded by an if statement. Outside of that, if the game is paused, the player is only allowed to unpause the game, no other action or printing is allowed.



```java
// Performs relevant action depending on if input is to move, fire bullet, pause or invalid,
// only if the game is not paused.
if (!isPaused) {
    if (movementInputs.contains(input)) {
        model.getShip().move(movementMap.get(input));
        if (isVerbose) {
            model.getLogger().log( text: "Ship moved to (" + model.getShip().getX()
                    + ", " + model.getShip().getY() + ")");
        }
    } else if (input.equals("F") || input.equals("f")) {
        model.fireBullet();
        model.getStatsTracker().recordShotFired();
    } else {
        model.getLogger().log( text: "Invalid input. Use W, A, S, D, F, or P.");
    }
}
if (input.equals("P") || input.equals("p")) {
    pauseGame();
}
```

Relevant tests:

- Tests.Game.ControllerTests.testPausedHandling(): this method first pauses the game and checks that the player is not able to move, and nothing is logged, even is verbose is true. It then unpauses the game, and ensures the player can now move correctly.



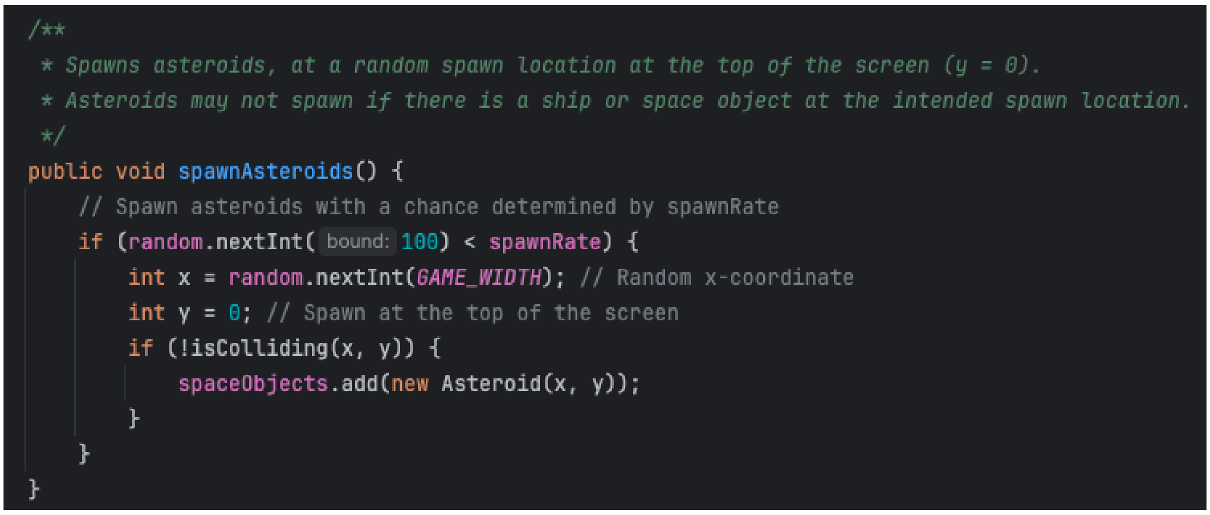
```java
@Test
public void testPausedHandling() {
    int startingY = gameController.getModel().getShip().getY();
    gameController.setVerbose(true);

    // Checks that the player cannot move when the game is paused
    gameController.pauseGame();
    gameController.handlePlayerInput("W");
    Assert.assertEquals(startingY, gameController.getModel().getShip().getY());
    Assert.assertEquals( expected: "", testUI.message);

    // Checks that the player can move correctly when the game is unpaused.
    gameController.handlePlayerInput("W");
    Assert.assertEquals( expected: startingY - 1, gameController.getModel().getShip().getY());
}
```

Limitation 6: There was no collision detection for PowerUps and Enemies when spawning new objects.
Game → GameModel

When new PowerUps and enemies were spawning, there was no detection of whether their spawn location would collide with an existing space object. There was detection for whether it would collide with the ship, but not other enemies and powerups. This meant that at higher levels when lots of enemies and powerups were spawning, the game would become extremely messy and difficult to play, and some locations would have multiple enemies, which were difficult to kill. Furthermore, many powerups would be collected at once, which was jarring for gameplay as you could jump multiple levels by going into one of the overlapping powerup positions.

Fix: In the helper methods created for Game.GameModel.spawnObjects() (which were made to adhere to the open-closed principle), specifically Game.GameModel.spawnAsteroids(), Game.GameModel.spawnEnemies() and Game.GameModel.spawnPowerUps(), there is logic to check that the intended spawn location is not clashing with an existing spaceObject or the ship.

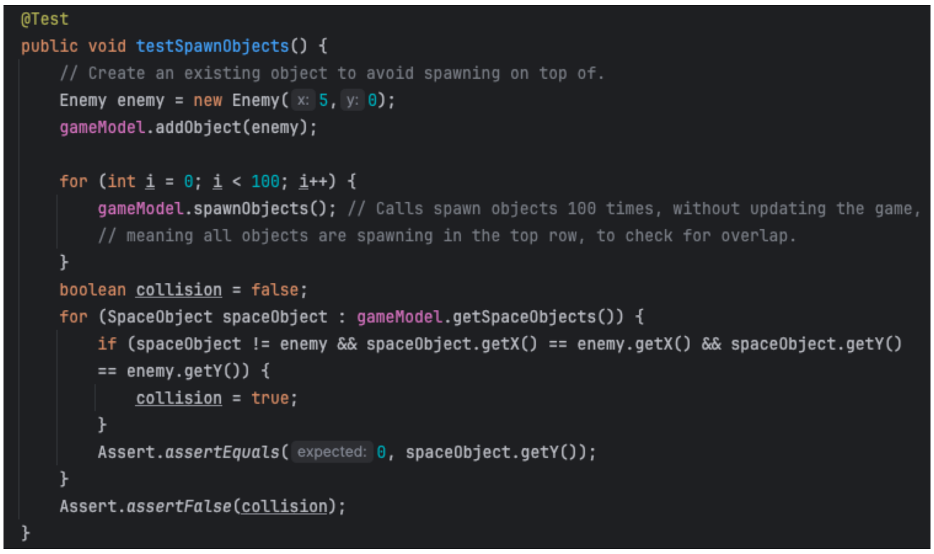
Specifically, the intended spawn x position is first decided through the call to random, and it is then checked if this new location (x, 0) at the top of the screen would collide with another space object. To adhere to the single responsibility principle and reduce complexity of the method, this was done using a private helper method, isColliding(x, y), which checks if the (x, y) coordinate provided is colliding with any space object or the ship. An example is provided for one of the private methods, spawnAsteroids().



```java
/**
 * Spawns asteroids, at a random spawn location at the top of the screen (y = 0).
 * Asteroids may not spawn if there is a ship or space object at the intended spawn location.
 */
public void spawnAsteroids() {
    // Spawn asteroids with a chance determined by spawnRate
    if (random.nextInt( bound: 100) < spawnRate) {
        int x = random.nextInt(GAME_WIDTH); // Random x-coordinate
        int y = 0; // Spawn at the top of the screen
        if (!isColliding(x, y)) {
            spaceObjects.add(new Asteroid(x, y));
        }
    }
}
```

Relevant tests:

- Test.Game.ModelTests.testSpawnObjects() was created to test that there was no overlap in objects when they spawn. Specifically, it first creates an enemy at the top of the screen, so it can be checked that no objects spawn on top of it. It then calls Game.GameModel.spawnObjects() 100 times, without updating the game tick. This means that all of the spaceobjects would be trying to spawn at the top of the screen without moving down. Hence, there is a very high likelihood that the intended spawn location of one of the SpaceObjects is the same as that of the existing enemy. It then iterates through all of the SpaceObjects, checks that the y value of their spawn point is 0, and then asserts that there has been no collision between any of the objects and the enemy. This is demonstrated below.



```java
@Test
public void testSpawnObjects() {
    // Create an existing object to avoid spawning on top of.
    Enemy enemy = new Enemy( x: 5, y: 0);
    gameModel.addObject(enemy);

    for (int i = 0; i < 100; i++) {
        gameModel.spawnObjects(); // Calls spawn objects 100 times, without updating the game,
        // meaning all objects are spawning in the top row, to check for overlap.
    }
    boolean collision = false;
    for (SpaceObject spaceObject : gameModel.getSpaceObjects()) {
        if (spaceObject != enemy && spaceObject.getX() == enemy.getX() && spaceObject.getY()
        == enemy.getY()) {
            collision = true;
        }
        Assert.assertEquals( expected: 0, spaceObject.getY());
    }
    Assert.assertFalse(collision);
}
```

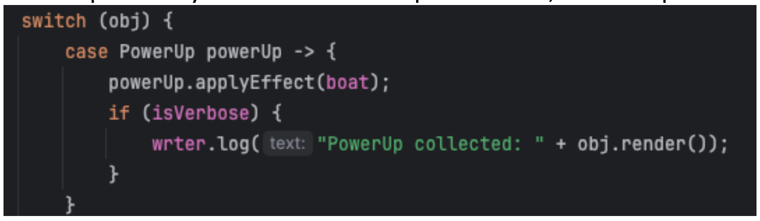
Limitation 7: Power ups were logging incorrect messages.

Game → GameModel

Previously, in Game.GameModel.checkCollisions(), when a power up was collected, the "Power-up collected: " + obj.render().

Fix: In Game.GameModel.checkCollisions(), when power ups are collected they now log the correct message, which is “PowerUp collected: " + obj.render(). However, this is only called when the game model is in a verbose state. The render method on each of the power ups returns its emoji representation. This is a more accurate representation for the PowerUp class.

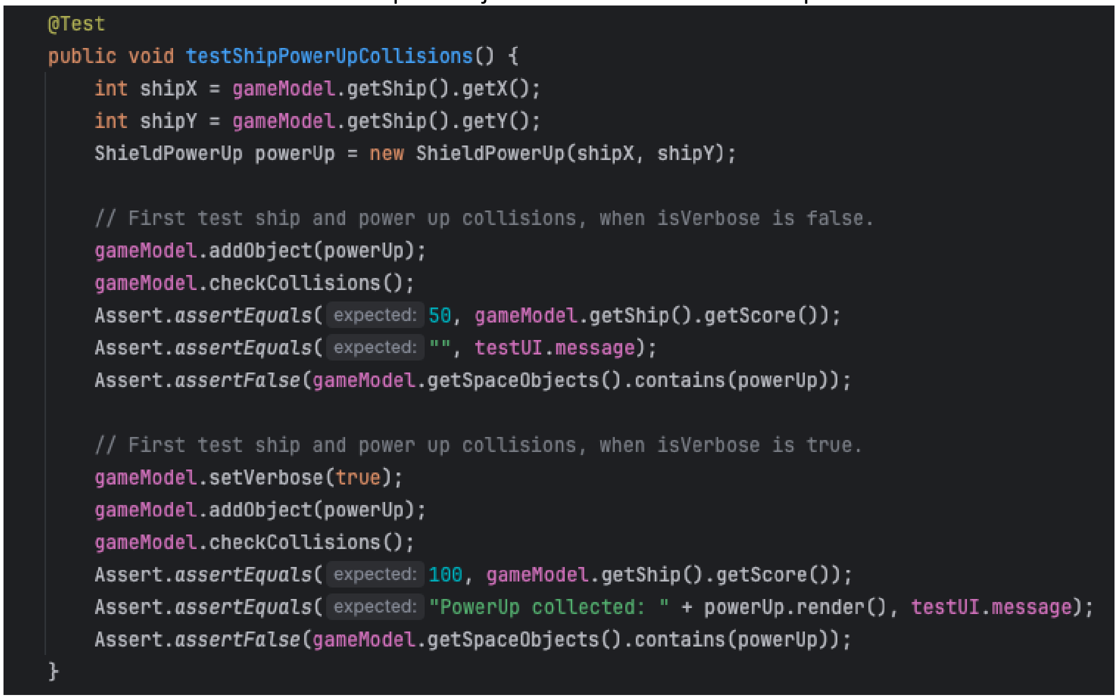
This is specifically checked in the helper method, checkShipCollisions(), as seen below.



```
switch (obj) {
    case PowerUp powerUp -> {
        powerUp.applyEffect(boat);
        if (isVerbose) {
            wrter.log( text: "PowerUp collected: " + obj.render());
        }
    }
```

Relevant tests:

- Tests.Game.GameModelTests.testShipPowerUpCollisions() : this method checks if when colliding, the PowerUps are correctly applied and the correct message is logged, if the model is in a verbose state. Specifically, it tests the collision between a ship and shield power up when the model is not verbose, and ensures that no message is logged but the effect is applied. It then sets the model state to be verbose, and performs the same action, checking that the effect is applied and the updated message is logged correctly. It also checks that the power up was correctly removed from the list of space objects after the collision took place.



```
@Test
public void testShipPowerUpCollisions() {
    int shipX = gameModel.getShip().getX();
    int shipY = gameModel.getShip().getY();
    ShieldPowerUp powerUp = new ShieldPowerUp(shipX, shipY);

    // First test ship and power up collisions, when isVerbose is false.
    gameModel.addObject(powerUp);
    gameModel.checkCollisions();
    Assert.assertEquals( expected: 50, gameModel.getShip().getScore());
    Assert.assertEquals( expected: "", testUI.message);
    Assert.assertFalse(gameModel.getSpaceObjects().contains(powerUp));

    // First test ship and power up collisions, when isVerbose is true.
    gameModel.setVerbose(true);
    gameModel.addObject(powerUp);
    gameModel.checkCollisions();
    Assert.assertEquals( expected: 100, gameModel.getShip().getScore());
    Assert.assertEquals( expected: "PowerUp collected: " + powerUp.render(), testUI.message);
    Assert.assertFalse(gameModel.getSpaceObjects().contains(powerUp));
}
```

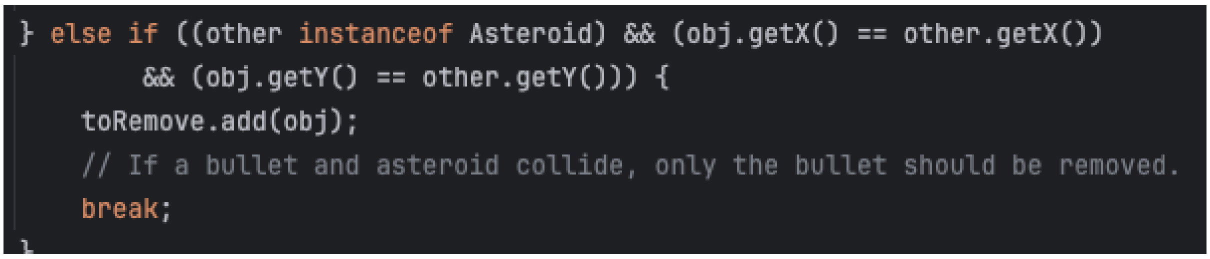
Limitation 8: Collisions of asteroids with bullets and the ship were incorrectly handled.

Game → GameModel

When a bullet and asteroid collided originally, the bullet would pass through the asteroid without having an effect. This was clearly incorrect, as the objects should not be able to pass through each other, and it meant that the bullets were killing more enemies as they should be, as they could kill enemies hiding behind asteroids.

Furthermore, when an asteroid collided with the ship, …

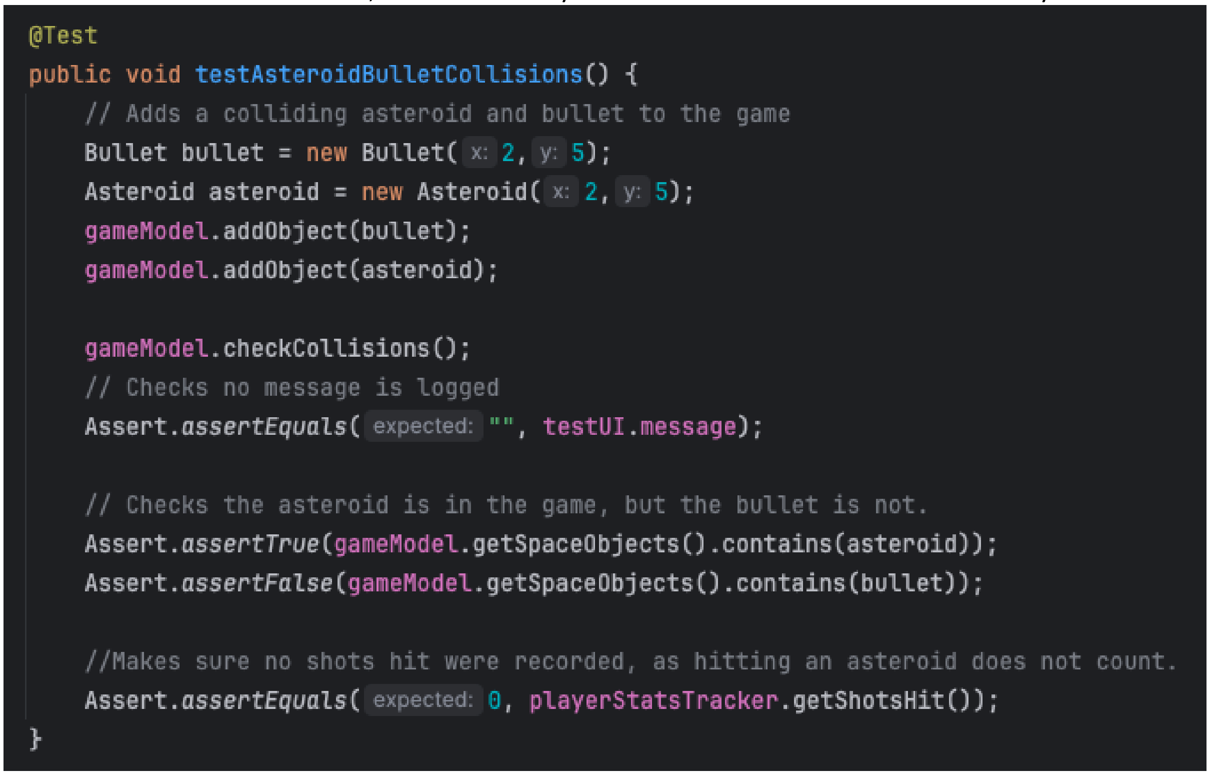
Fix: in the updated method, Game.GameModel.checkCollisions(), the helper method, Game.GameModel.checkBulletCollisions() was implemented. This method removes the bullet when it collides with an asteroid, as seen below.



```
} else if ((other instanceof Asteroid) && (obj.getX() == other.getX())
        && (obj.getY() == other.getY())) {
    toRemove.add(obj);
    // If a bullet and asteroid collide, only the bullet should be removed.
    break;
}
```

Relevant tests:

- Test.Game.GameModelTests.testAsteroidBulletCollisions() tests to make sure this is handled correctly. It first initialises a bullet and asteroid which are colliding and adds them into the game. It then calls GameModel.checkCollisions(), and ensures that afterwards, the bullet is removed from the game, but the asteroid is still in the game. It also makes sure nothing is logged, and no shots hit were recorded, as shots are only recorded as hit if the bullet hits an enemy.



```
@Test
public void testAsteroidBulletCollisions() {
    // Adds a colliding asteroid and bullet to the game
    Bullet bullet = new Bullet( x: 2, y: 5);
    Asteroid asteroid = new Asteroid( x: 2, y: 5);
    gameModel.addObject(bullet);
    gameModel.addObject(asteroid);

    gameModel.checkCollisions();
    // Checks no message is logged
    Assert.assertEquals( expected: "", testUI.message);

    // Checks the asteroid is in the game, but the bullet is not.
    Assert.assertTrue(gameModel.getSpaceObjects().contains(asteroid));
    Assert.assertFalse(gameModel.getSpaceObjects().contains(bullet));

    //Makes sure no shots hit were recorded, as hitting an asteroid does not count.
    Assert.assertEquals( expected: 0, playerStatsTracker.getShotsHit());
}
```

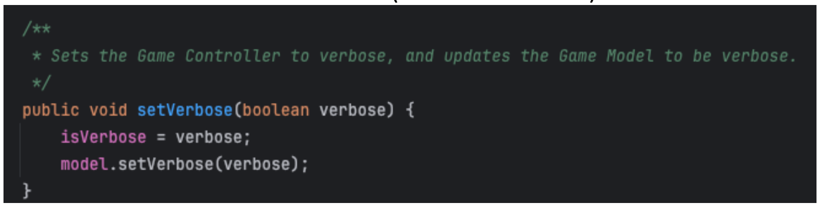
Limitation 9: The game lacks any way to control the verbosity.

Game → GameModel
Game → GameController

Previously, any messages that were to be logged had to be logged every single time they occurred, as there was no way to decide whether the game should be in a verbose state or not. This meant that messages may be logged unnecessarily which could be distracting to the player. The log was constantly being updated with new messages telling them their every action, which was not helpful at all as they could already see their actions on the screen.

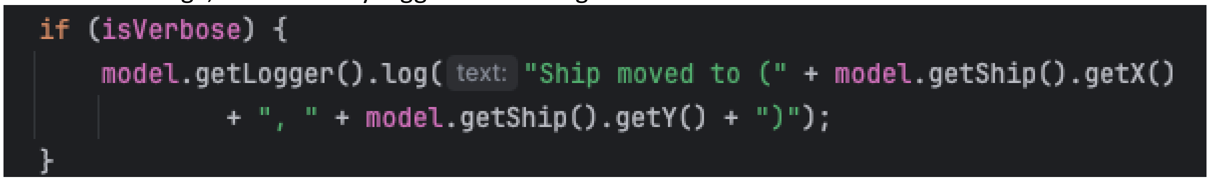
Fix: GameModel and GameController both get their own field which is a Boolean isVerbose. This can keep track of whether they are verbose, and can decide whether to log messages accordingly. Some messages must be logged regardless of the verbose state, but less important ones should only be logged when the state is verbose. Game.GameModel.setVerbose(boolean verbose) and Game.GameController.setVerbose(boolean verbose) methods were both added, to set the respective verbose states. The setVerbose method of the game controller also updated the game model, which is essential to make sure the game does not have any inconsistencies across different classes.

Game.GameController.setVerbose(boolean verbose) can be seen below.



```java
/**
 * Sets the Game Controller to verbose, and updates the Game Model to be verbose.
 */
public void setVerbose(boolean verbose) {
    isVerbose = verbose;
    model.setVerbose(verbose);
}
```

An example of checking the verbose state is seen below, in Game.GameController.handlePlayerInput(String input). Logging what movement has occurred is not an essential message, so this is only logged when the game is in a verbose state.



```java
if (isVerbose) {
    model.getLogger().log( text: "Ship moved to (" + model.getShip().getX()
            + ", " + model.getShip().getY() + ")");
}
```

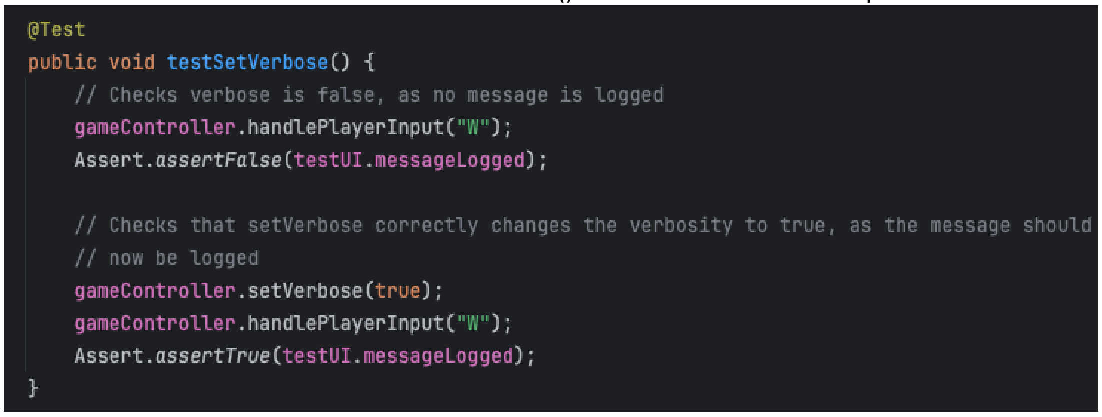
Many methods modified to implement this new verbosity control in both game controller and game model, including Game.GameController.handlePlayerInput(String input), Game.GameModel.checkCollisions(), Game.GameModel.levelUp() and the new method Game.GameController.refreshAchievements().

Relevant tests:

Since isVerbose is a private variable in both the game controller and game model, and no public tests can be made to access it, it was difficult to test whether the setVerbose functions were working correctly. Therefore, to test this, the following tests were created. These instead performed actions which set the verbose state then executed methods which log messages only when the verbose state is true. This relied on the other methods working correctly, but these were tested as well to ensure no issues arose.

- Tests.Game.GameControllerTests.testSetVerbose(): this tests that the setVerbose(boolean verbose) method in the game controller is functioning correctly, by calling handlePlayerInput("W") and checking that the logging only occurred after setVerbose(true) had been called.
- Tests.Game.GameModelTests.testSetVerbose(): this test works similarly but calls the level up method in game model twice. It checks that the first time, the verbose level should be false, meaning no message is logged, but after setVerbose(true) is called, it changes to the true state, and logs the next level up method.

Tests.Game.GameControllerTests.testSetVerbose() is seen below as an example of this.



```java
@Test
public void testSetVerbose() {
    // Checks verbose is false, as no message is logged
    gameController.handlePlayerInput("W");
    Assert.assertFalse(testUI.messageLogged);

    // Checks that setVerbose correctly changes the verbosity to true, as the message should
    // now be logged
    gameController.setVerbose(true);
    gameController.handlePlayerInput("W");
    Assert.assertTrue(testUI.messageLogged);
}
```

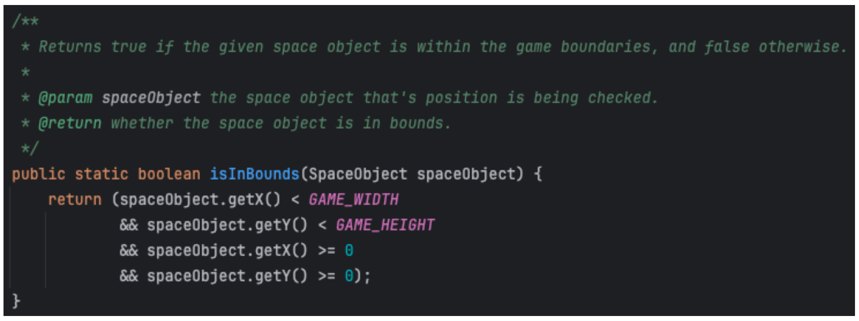
Other tests created to ensure verbosity functionality include:
- Tests.Game.GameControllerTests.testVerboseHandling(): checks that movement is logged correctly when verbose.
- Tests.Game.GameControllerTests.testNotVerboseHandling(): checks movement is not logged when it is not verbose.
- Tests.Game.GameModelTests.testLevelUp(): checks that the level up is only logged when the game state is verbose
- Tests.Game.GameModelTests.testShipPowerUpCollisions(): checks that the power up effect is only logged when the game state is verbose
- Tests.Game.GameModelTests.testShipEnemyCollisions(): checks that the enemy collision is only logged when the game state is verbose
- Tests.Game.GameModelTests.testShipAsteroidCollisions(): checks that the asteroid collision is only logged when the game state is verbose

Limitation 10: no method for checking if a space object is in the game boundaries.

Game → GameModel

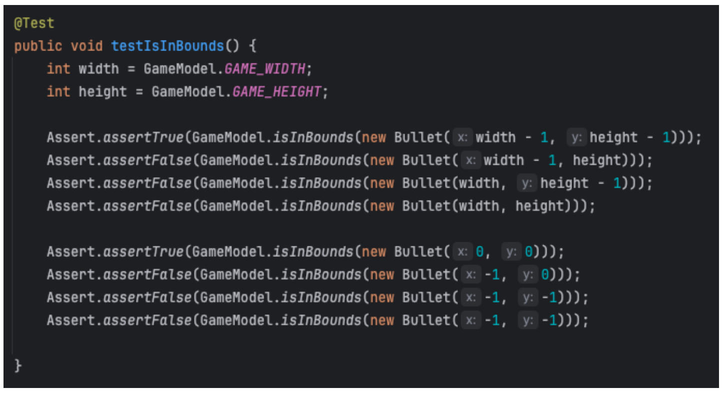
Initially, every time it needed to be checked that a space object was in the game boundaries, it had to be done so manually which was ineffective. This should be its own method to adhere to the single responsibility principle at a method level and reduce the complexity of some longer methods.

Fix: the Game.GameModel.isInBounds(SpaceObject spaceObject) method was created, which checks that the provided SpaceObject is within the game boundaries, based on the game width and game height. This is implemented in Game.GameModel.updateGame() to remove off-screen objects. The method is shown below:



```java
/**
 * Returns true if the given space object is within the game boundaries, and false otherwise.
 *
 * @param spaceObject the space object that's position is being checked.
 * @return whether the space object is in bounds.
 */
public static boolean isInBounds(SpaceObject spaceObject) {
    return (spaceObject.getX() < GAME_WIDTH
            && spaceObject.getY() < GAME_HEIGHT
            && spaceObject.getX() >= 0
            && spaceObject.getY() >= 0);
}
```

Relevant tests:
- Tests.Game.GameModelTests.testIsInBounds(): this test checks a bullet at a variety of positions to check that the test correctly identifies when something is or is not in bounds. To make sure this method was thorough, all of the game boundaries were tested, first at the maximum dimensions and then at the minimum dimensions of the game screen. This can be seen below.



```java
@Test
public void testIsInBounds() {
    int width = GameModel.GAME_WIDTH;
    int height = GameModel.GAME_HEIGHT;

    Assert.assertTrue(GameModel.isInBounds(new Bullet( x: width - 1,  y: height - 1)));
    Assert.assertFalse(GameModel.isInBounds(new Bullet( x: width - 1, height)));
    Assert.assertFalse(GameModel.isInBounds(new Bullet(width,  y: height - 1)));
    Assert.assertFalse(GameModel.isInBounds(new Bullet(width, height)));

    Assert.assertTrue(GameModel.isInBounds(new Bullet( x: 0,  y: 0)));
    Assert.assertFalse(GameModel.isInBounds(new Bullet( x: -1,  y: 0)));
    Assert.assertFalse(GameModel.isInBounds(new Bullet( x: -1,  y: -1)));
    Assert.assertFalse(GameModel.isInBounds(new Bullet( x: -1,  y: -1)));

}
```

Design Choices

Many design choices were made to make sure the codebase had maximum readability, method decomposition, a clear structure and descriptive naming.

One of the first changes made was to go through and change variable names to make sure they were easily understandable, as several variables had unclear purposes. For example:

- endTime → startTime: this variable represents the time the game started and should be named accordingly (Game.GameController)
- aManager→ achievementManager: represents the achievement manager, so aManager is indescriptive and unclear (Game.GameModel)
- lvl → level: this int variable represents the game level, and was unclear so was renamed (Game.GameModel)
- wrter → logger: this Logger variable represents the logger and therefore should be named clearly to show this (Game.GameModel)
- boat → ship: this is a Ship instance, therefore boat is unnecessarily confusing (Game.GameModel)
-

Several strategies were used to reduce coupling between methods. Specifically, whenever possible interfaces and abstract types were used rather than concrete classes, and only interfaces were passed in as method arguments. It is vital to avoid passing in whole class instances as arguments, as this leads to high coupling, where one class can manipulate the other classes. This also violates the dependency inversion principle, which is further elaborated on below. Any time values were required to be passed between methods, as much information was hidden as possible, meaning only specific values of the class were passed in rather than the whole class object. In addition, dependency injection was utilised, meaning new instances were always initialised outside of a method and then passed in whenever possible, not instantiated within the method itself.

Single Responsibility Principle:

The single responsibility principle states that a class or method should have only one reason to change, meaning it is only responsible for one aspect of the codebase.

Due to the nature of the assignment, there were limitations in implementing the single responsibility principle at a class level, as no new public classes can be made. There were many classes which performed multiple roles, which is not ideal as these complexities mean it can be difficult to understand where changes may arise. For example, both the GameModel and GameController classes were quite complex. The GameController had many responsibilities, including managing the initialisation of the game, pausing and ending the game, handling inputs within the game and updating the achievements. Had there been less limitations on this, the class could have been split into multiple classes to focus on a single responsibility. For example, it could be divided into InputHandler (to handle player inputs), AchievementHandler (to communicate with the achievement manager, handle refreshing and log to the PlayerStatsTracker and ui), GameLoopManager (to update the game using the onTick function, assist with pausing), and any other required classes to perform only one responsibility.

Despite this, the single responsibility principle could be implemented at a smaller scale, specifically pertaining to the methods. Some examples of the implementation of this principle are listed below:

- GameModel.checkCollisions(). This method was very complex, as it was handling multiple responsibilities, first checking collisions between the ship and other SpaceObjects, then checking collisions between any bullets and other SpaceObjects. To better adhere to the single responsibility principle, this method was split up into two private methods GameModel.checkShipCollisions() and GameModel.checkBulletCollisions().

- Game.Achievements.FileHandler.read() : the single responsibility principle was adhered to when implementing the new achievements section of the assignment. For example, the method Game.Achievements.FileHandler.read() was particularly long, as it first had to read all of the data from the file location, and then had to convert it into a list of strings to be returned in the correct format required. To avoid violating the principle, this method was split up to instead use a private helper method Game.Achievements.FileHandler.readData() which first read the data as a string while handling all the possible exceptions which could arise, before performing the read() method.

- Game.GameController.renderGame() calls the method getElapsedSeconds() from player stats tracker, as it should not have to check the elapsed seconds since the game start manually.

- Game.GameModel.updateGame(): the new modification updated in the limiations section allows this method to call the helper method Game.GameModel.isInBounds(). UpdateGame should not be checking boundaries, it should only be updating the game state. This means it is now focussing on only one responsibility.

- Game.GameModel.spawnObjects() calls isColliding(): spawn objcets should only be focussing on spawning the objects. By implementing this helper method, duplicate code can be removed, and it can focus on its purpose. isColliding() also calls isCollidingWithShip(), which is another method to remove duplicate code from several methods and adhere to this principle.

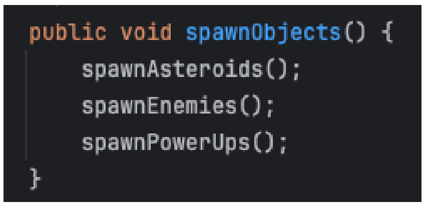
- Game.ui.gui.* and Game.ui.utility.* already adhere well to this principle, as each method has only one singular purpose and one reason to change.

Open-Closed Principle:

This principles state that classes and methods should be open to extension but closed for modification. Similar to the single responsibility principle, this had some limitations to being implemented on a class level. There were several times when class modification could have helped to implement this.

For example, each achievement type could ideally be made into a subclass of Game.Achievements.GameAchievement, each with their own refresh() method to be called in Game.GameController.refreshAchievements(). However, this was not possible. To handle this, these were instead performed within the class, on a method level. Some newly implemented and existing examples of this principle are listed below:

- Game.GameController.refreshAchievements(): since this could not be implemented on a class level, the refresh methods were defined individually for each achievement type. This means that each achievement can have its own method, such as Game.GameController.refreshSharpShooter(). By defining as consumers and placing them into Map<String, Consumer<Achievement>> refreshMethods, this means that the refreshAchievements() method is closed for modification. It already calls the adequate refresh method on each achievement. If more achievements needed to be added, this would be open to modification as new methods could be created and simply added into the refreshMethods map.
- Game.GameModel.spawnObjects(): By splitting this method into several helper methods, as seen below, this adheres far better to the open-closed principle. If a new object was added which needed to be spawned, this would be easily open for extension as a new method could be created to handle its specific spawning scenario. While the spawnObjects() method is not fully “closed” for modification, the modification would be extremely minor if this was the case, so no issues should arise in implementing this new method.



```
public void spawnObjects() {
    spawnAsteroids();
    spawnEnemies();
    spawnPowerUps();
}
```

- Game.Core.SpaceObject.*: the SpaceObject interface and all the subclasses adhere well to this principle, as it is easily open to being extended if new space objects need to be created. The interface can simply be extended, and the methods implemented. There should be no need to modify any of the existing classes, thus meaning it is closed for modification.
- Game.ui.* and Game.utility.* also adhere well to the principle already. In Game.ui.gui.*, the gui extends from the ui interface, so if required new UI extensions could be implemented by simply extending the interface.
- Game.achievements.*: When implementing the new achievement section, it was important to adhere to this principle. For this reason, the achievement and achievement file interfaces were used. With FileHander() implementing the AchievementFile interface, this makes it open for extension, should a new type of file handling be required in addition to the existing method, it could just extend the same interface without having to induce any modifications. Similarly, GameAchievement extends the Achievement interface, so should any other types of achievement be needed which require different behaviour, this could be extended without modifications.

Liskov Substitution Principle:

This principle states that a subclass should always be able to replace an instance of its superclass. Even when methods are overridden, the contract should still be adhered to. This means the preconditions must always be the same or weaker than that of the parent class, and the postconditions must always be the same or stronger. No unexpected exceptions should be thrown by subclasses, as they would not be able to be handled. To check this, we can ensure that all implementing classes and subclasses give expected behaviour, and do not strengthen preconditions or weaken postconditions. Some examples of how this has been checked are discussed below:

- Game.Core.*: In core, all objects inherit from SpaceObject. It can be observed that the functionality between the inheriting classes implement all methods from space object without changing any of the behaviour or violating the contract, as they all have simply return their x and y coordinates, and allow themselves to be rendered with their representation. Furthermore, many classes inherit from ObjectWithPosition. None of the methods in object with position have conditions to be adhered to, and these are not changed when inherited by the subclasses. To adhere to this principle, all of the subclasses closely mirror the behaviour of the parent classes without changing functionality. Similarly, the power ups extend PowerUp which implements PowerUpEffect. In all stages of this implementation, only minor changes occur, and all subclasses have a valid implementation of the superclass methods, meaning they could be substituted at any time. The Junit4 tests assist in ensuring that when these superclass methods are called on instances of the subclass, no issues arise and no exceptions are thrown.

- Game.Achievements.*: In this package, this principle is implemented. We can consider how FileHandler implements the AchievementFile interface. None of the methods have preconditions or postconditions so no contract violations occur and all methods from the interface are used in the implementing class, thoroughly checking for any exceptions that could be thrown to ensure no unpredictable behaviour.
  We can also consider GameAchievement implementing the Achievement interface. In this case, there are several preconditions and postconditions. For example, in the interface, getProgress() ensures 0.0 <= getProgress() <= 1.0, and the GameAchievmenet has the same condition, meaning

it has not been weakened. Similarly, the preconditions and postconditions for the setProgress() method are kept the same, to ensure no contract violations. When GameAchievement implements these methods, it performs checks to ensure there are no violations. Throughout this package, all the methods accurately implement any methods from their interface.

- Game.GameController, Game.GameModel, Game.utility.* and Game.Main do not implement any interfaces or superclasses meaning this principle is not relevant to them.

- Game.ui.* and Game.ui.gui.* adhere to this principle, with GUI implementing the UI interface. In this implementation, it is evident that all the methods of the interface are implemented, meaning it could be a valid substitute any time UI is used. There are no precondition or postcondition change violations and the class still adheres to the contract.

Interface Segregation Principle:

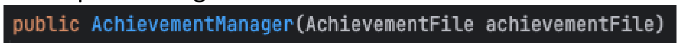
The interface segregation principle states that classes should not be forced to depend on interfaces they do not use, and should not need to implement methods they don't use. To check this, we just need to ensure that there are no methods left empty by classes implementing interfaces.

The following checks performed are outlined below:
- Game.Core.*: no methods were left empty. All classes only inherit from interfaces whose methods they rely on. For example, all subclasses of PowerUp implement all methods from the PowerUpEffect interface that they implement. Similarly all implementing classes of SpaceObject implement all methods.
- Game.Achievements.*: all classes were checked for empty methods, but none were found meaning it adheres to the principle. All implementing classes of the Achievement and AchievmentFile interfaces implemented all the methods.
- Similarly, in Game.utility.*, Game,ui.*, Game.ui.gui.*, any implementations of interfaces did not result in any empty methods, only relevant interfaces were used.
- Game.Main, Game.Model and Game.Controller do not implement any interfaces or superclasses meaning this principle is not an issue, but it was double checked that no empty methods were present in any of these classes.

Dependency Inversion Principle:

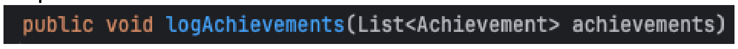
The dependency inversion principle states that higher level classes should not rely on, or be affected by changes in lower level classes. Instead, they should both depend on interfaces. Additionally, details (classes and their specific methods) should depend on abstractions, and the abstractions should not rely on the details. To check this, we can go through and ensure that dependencies are injected via abstracted interfaces rather than concrete implementation, and concrete classes are hidden behind interfaces (concrete classes are defined with the type being the interface, not the more reductionist class). This should mean it is easy to swap dependencies. Some of the checks performed are outlined below:
- Game.core.*: generally in the core package, the methods are simple and use primitive and built types as their arguments, rather than specific classes, which adheres to this principle. The only exception to this is the applyEffect(Ship ship) method, which is required since the ship class has specific implementation which must be acted on, and it can therefore not be substituted with a higher interface such as controllable. Specifically, the power ups need to act on the ship's score and health.
- Game.utility.*: does not take any class of interface arguments only primitive and built in, thus adhering to the principle.
- Game.achievements.*: When implementing achievement functionality, this principle was carefully adhered to. In achievement manager, the interfaces Achievment and AchievementFile are always passed as method arguments rather than concrete classes. In the other three classes of achievements, only primitive and built in arguments are used and no classes are passed in,

therefore also satisfying requirements. For example, an AchievementFile is passed in rather than the implementing FileHandler.



```
public AchievementManager(AchievementFile achievementFile)
```

- Game.ui.* and Game.ui.gui.*: Similarly, it was checked that in all of the methods, no classes are passed in. Each method takes the interface argument allowing for maximum flexibility and no unnecessary dependencies on concrete classes which could cause issues. For example, Game.UI.GUI.logAchievements takes a list of Achievements rather than the concrete implementation GameAchievement.



```
public void logAchievements(List<Achievement> achievements)
```