Limitation, Fixes, and Code Design Decision: **1**

Limitation, Fixes, and Code Design Decision:

1. applyEffect() - game.core.HealthPowerUp

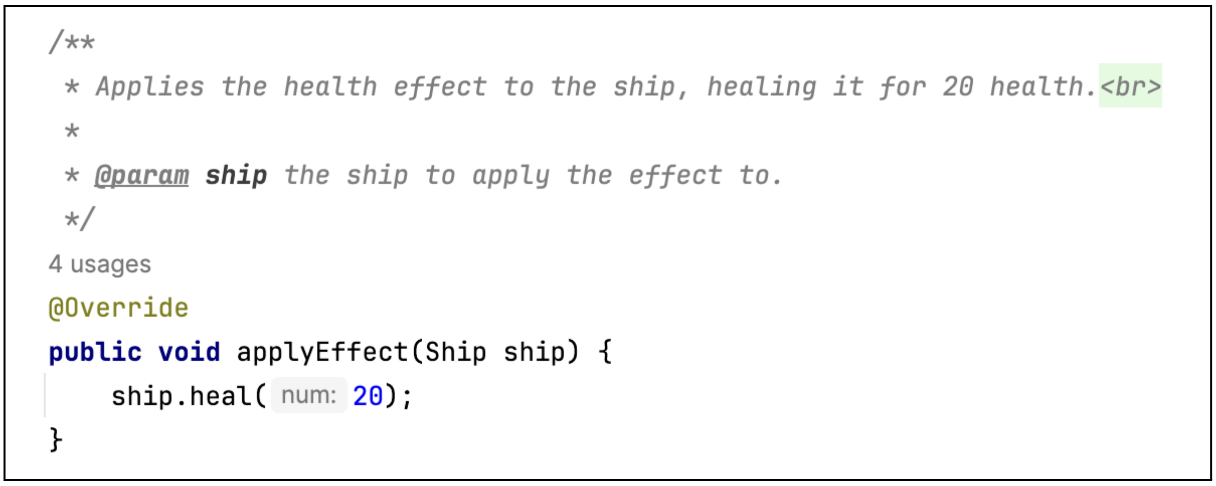
The difference lies in the applyEffect() method, where the A1 Javadocs specifies that it should produce a standard output using System.out.println(), while in A2, this requirement has been removed.

Fixes:

After removing the System.out.println() call from the applyEffect() method in the HealthPowerUp class, the method now solely applies the healing effect to the ship without producing any console output.

Code Design:

- Single Responsibility Principle (SRP): The System.out.println() was removed from HealthPowerUp.applyEffect() to ensure the method focuses solely on game logic and makes the class cleaner, easier to test, and consistent with A2's Design standards.



```
/**
 * Applies the health effect to the ship, healing it for 20 health.<br>
 *
 * @param ship the ship to apply the effect to.
 */
4 usages
@Override
public void applyEffect(Ship ship) {
    ship.heal( num: 20);
}
```

2. applyEffect() - game.core.ShieldPowerUp

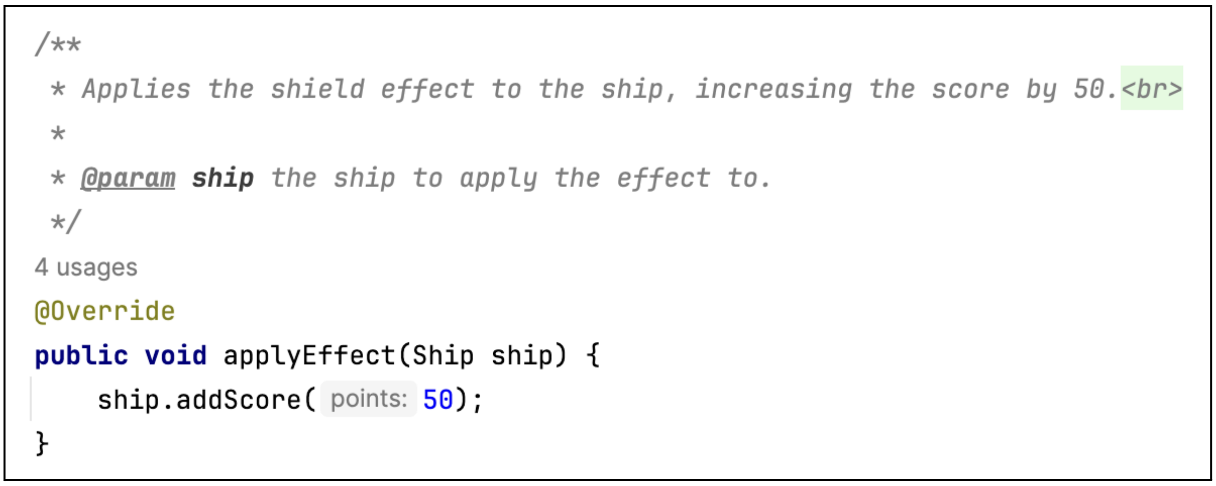
Similar to HealthPowerUp, the key difference lies in the applyEffect() method. In A1, it includes output to standard output via System.out.println(), whereas in A2, this requirement has been removed.

Fixes:

After removing the System.out.println() call from the applyEffect() method in the ShieldPowerUp class, the method now increases the ship's score by 50 without displaying any output to the console.

Code Design:

- Single Responsibility Principle (SRP): The System.out.println() was removed from ShieldPowerUp.applyEffect() to follow the SRP by separating game logic from output concerns. This improves testability, avoids unnecessary console output during gameplay.



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

3. tick() - game.core.PowerUp

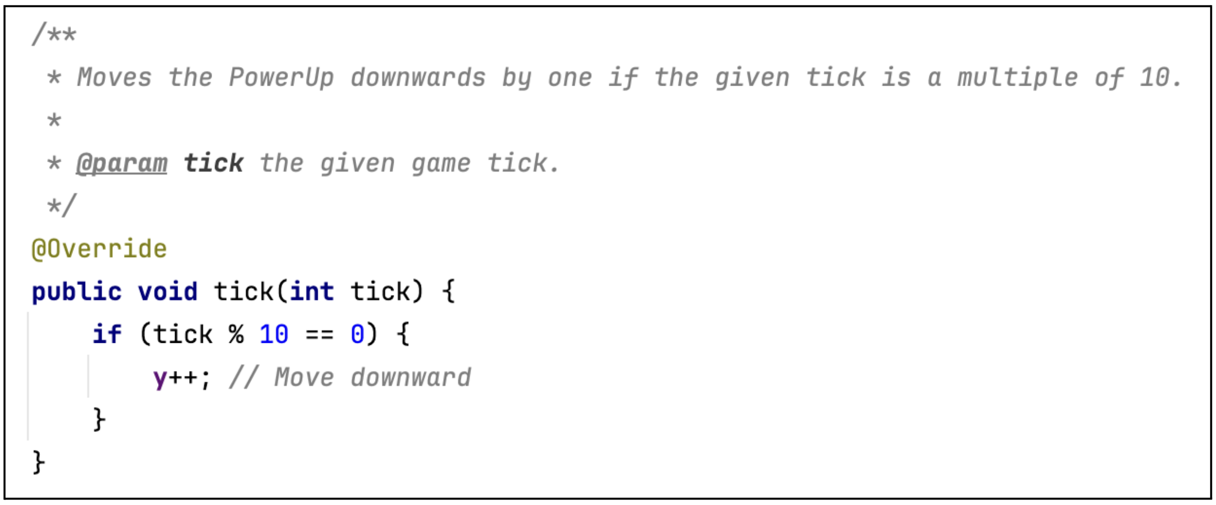
In A1, the tick() method in PowerUp is exclusively defined to perform no action, as PowerUps were not tick-dependent. However, in A2, this method is redefined to move the PowerUp downward every 10 game ticks, introducing tick-based behaviour.

Fixes:

The tick() method in PowerUp was updated to align with A2 specifications by adding behaviour that moves the PowerUp downward every 10 game ticks. Previously, PowerUps would remain static at the top of the screen after spawning. With this fix, PowerUps now fall over time, like other space objects,

Code Design:

- Single Responsibility Principle (SRP): This change supports the SRP by encapsulating tick-based movement within the tick() method, rather than relying on external logic to move PowerUps. It also demonstrates cohesion to
- Open/Closed Principle (OCP): It also demonstrates cohesion to OCP as the class is extended with new behaviour without modifying unrelated parts of the system.



```java
/**
 * Moves the PowerUp downwards by one if the given tick is a multiple of 10.
 *
 * @param tick the given game tick.
 */
@Override
public void tick(int tick) {
    if (tick % 10 == 0) {
        y++; // Move downward
    }
}
```

4. toString() - game.core.ObjectWithPosition

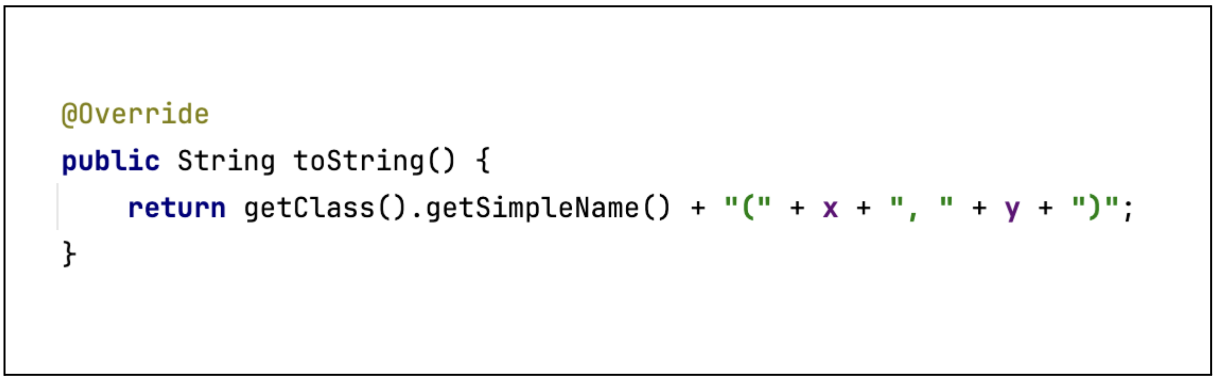
The toString() method was not present in the A1 Javadocs and is newly introduced in A2. Its purpose is to provide a string representation of an object's type and coordinates, e.g., "Bullet(2, 4)".

Fixes:

By adding this method, it returns the simple class name along with the object's current (x, y) position, making it easier to identify objects when debugging or printing logs.

Code Design:

- Single Responsibility Principle (SRP): The addition of toString() adheres to the SRP by giving each object responsibility over how it presents itself as text. For instance, instead of requiring external classes (like Logger, UI, or test code) to manually retrieve getX(), getY(), or class names to display an object's identity, the object itself now knows how to describe its position clearly (example output: Bullet(2, 4)).



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

5. checkCollisions() - game.GameModel

In A1, all collisions log messages unconditionally. In A2, logging only happens if verbose is true, and the recordShotHit() method was added to Enemy bullet collisions for tracking stats.

Fixes:

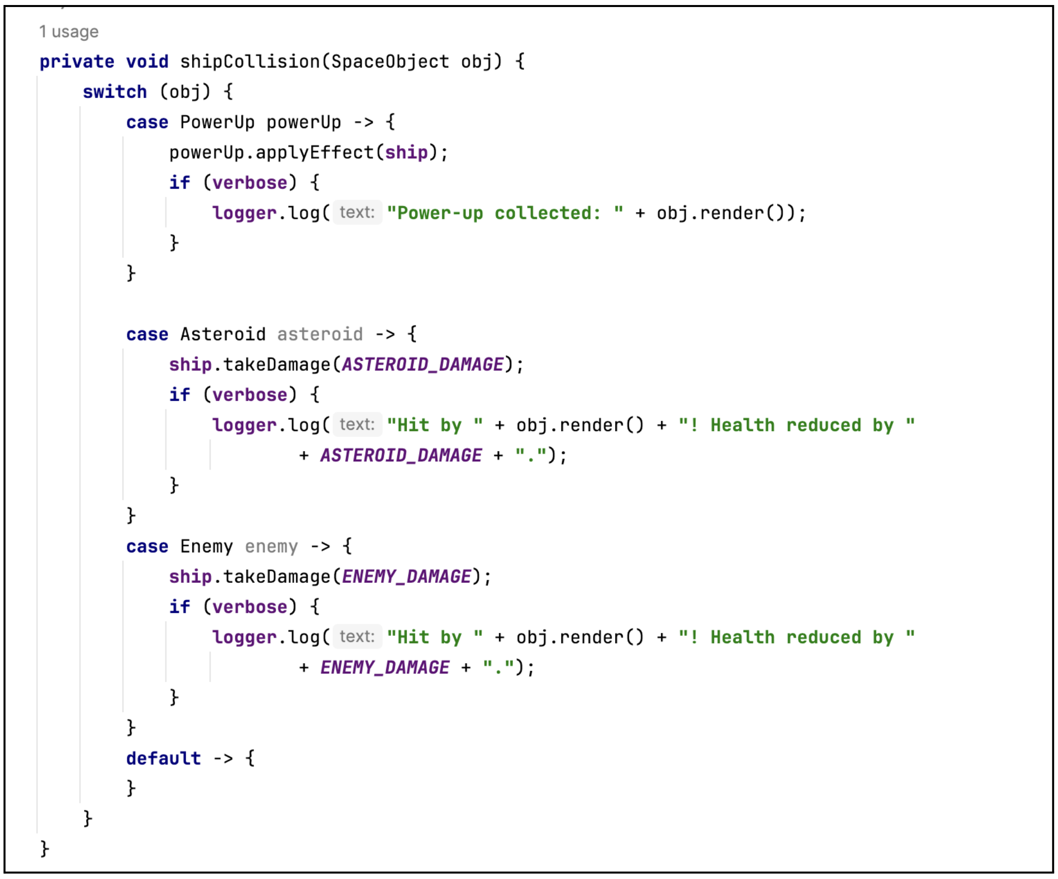
- Added conditional logging using the verbose flag to avoid unnecessary console clutter during normal gameplay or testing.
- Added recordShotHit() to bullet collisions with Enemy to enable achievement tracking like accuracy or kill count.
- Refactored logic into shipCollision() and bulletCollision() helper methods to simplify checkCollisions() and improve maintainability.

Ship collisions now log only when verbose mode is enabled, and bullets increment hit count only when hitting enemies (ignoring asteroids) making the game more challenging while also supporting achievement tracking. The refactored logic using helper methods simplifies checkCollisions() and improves maintainability and future extensibility.

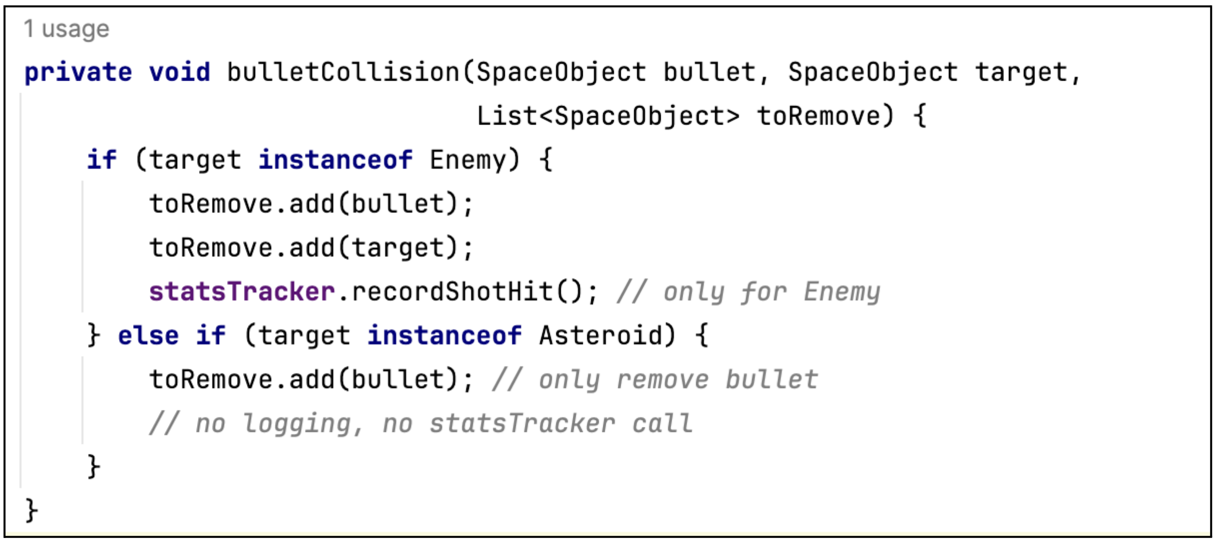
Code Design:

- Single Responsibility Principle (SRP): Each method has one clear purpose checkCollisions() manages the flow, shipCollision() handles Ship-related logic, and bulletCollision() handles Bullet interactions. This keeps responsibilities separated and manageable.
- Open/Closed Principle (OCP): The Design is open for extension (e.g., adding new SpaceObject types or collision behaviours) but closed for modification future changes will not require editing the core loop.
- Liskov Substitution Principle (LSP): All SpaceObject subclasses like PowerUp, Asteroid, and Enemy can be handled generically. Behaviour remains correct regardless of which subclass is used.
- Dependency Inversion Principle (DIP): The logger and statsTracker are passed into the model, rather than being hard-coded. This allows better modularity and easier testing.
- Encapsulation: Collision logic and game mechanics are kept inside the model, while logging is guarded by a verbose flag. This ensures clean separation between game logic and UI/debug output.

```java
public void checkCollisions() {
    List<SpaceObject> toRemove = new ArrayList<>();
    // Check collisions with the Ship
    for (SpaceObject obj : spaceObjects) {
        // Skip checking Ships (No ships should be in this list)
        if (obj instanceof Ship) {
            continue;
        }
        // Check Ship collision (except Bullets)
        if (isCollidingWithShip(obj.getX(), obj.getY()) && !(obj instanceof Bullet)) {
            shipCollision(obj); // Delegate to helper method
            toRemove.add(obj); // Remove object after collision
        }
    }
    //Check collisions with Bullets
    for (SpaceObject obj : spaceObjects) {
        // Check only Bullets
        if (!(obj instanceof Bullet)) {
            continue;
        }
        // Check Bullet collision with Enemy or Asteroid
        for (SpaceObject other : spaceObjects) {
            if (!(other instanceof Enemy || other instanceof Asteroid)) {
                continue;
            }
            if ((obj.getX() == other.getX()) && (obj.getY() == other.getY())) {
                bulletCollision(obj, other, toRemove); // Delegate to helper method
                break;
            }
        }
    }

    spaceObjects.removeAll(toRemove); // Remove all collided objects
}
```



```
1 usage
private void shipCollision(SpaceObject obj) {
    switch (obj) {
        case PowerUp powerUp -> {
            powerUp.applyEffect(ship);
            if (verbose) {
                logger.log( text: "Power-up collected: " + obj.render());
            }
        }

        case Asteroid asteroid -> {
            ship.takeDamage(ASTEROID_DAMAGE);
            if (verbose) {
                logger.log( text: "Hit by " + obj.render() + "! Health reduced by "
                        + ASTEROID_DAMAGE + ".");
            }
        }
        case Enemy enemy -> {
            ship.takeDamage(ENEMY_DAMAGE);
            if (verbose) {
                logger.log( text: "Hit by " + obj.render() + "! Health reduced by "
                        + ENEMY_DAMAGE + ".");
            }
        }
        default -> {
        }
    }
}
```



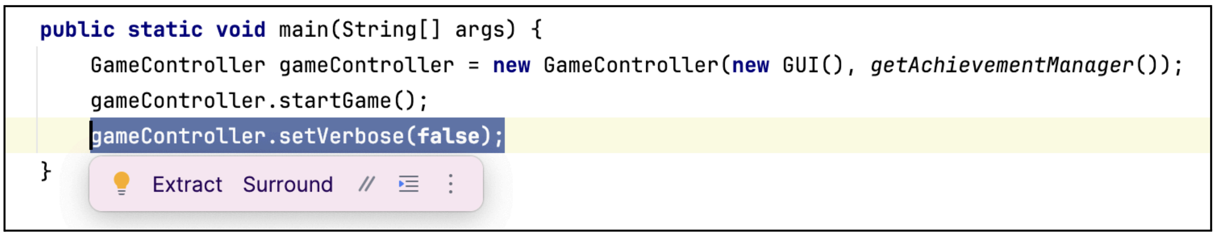
```
1 usage
private void bulletCollision(SpaceObject bullet, SpaceObject target,
                             List<SpaceObject> toRemove) {
    if (target instanceof Enemy) {
        toRemove.add(bullet);
        toRemove.add(target);
        statsTracker.recordShotHit(); // only for Enemy
    } else if (target instanceof Asteroid) {
        toRemove.add(bullet); // only remove bullet
        // no logging, no statsTracker call
    }
}
```

6. levelUp() - game.GameModel

In A1, the level-up message is always logged when the level increases. In A2, logging is now conditional on the verbose flag. This means the game will still level up when the score threshold is met, but the log message only appears when verbose == true.

Fixes:

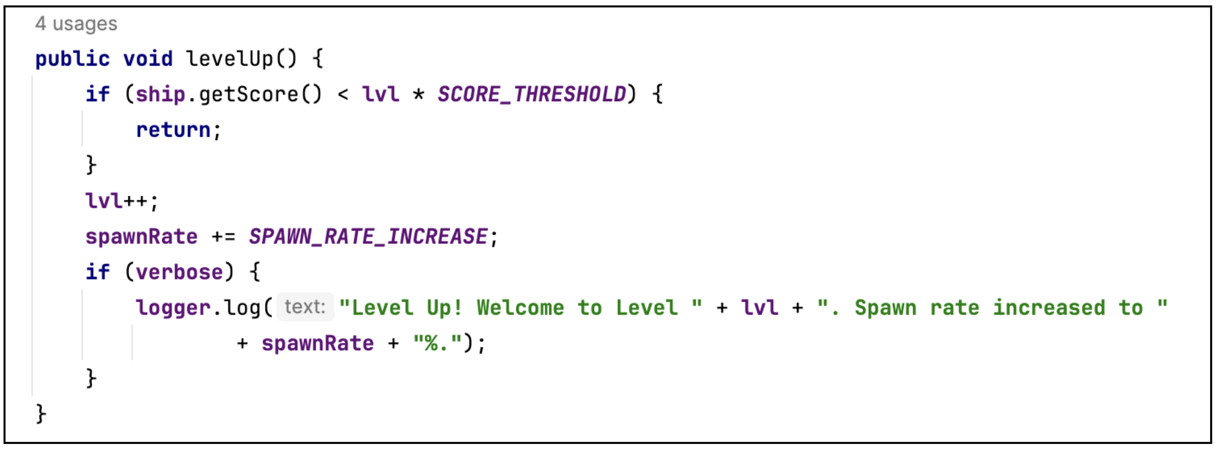
Wrapped the logging inside if (verbose) { logger.log(...);}. This ensures the level-up message is optional, controlled via the verbosity setting.



```
public static void main(String[] args) {
    GameController gameController = new GameController(new GUI(), getAchievementManager());
    gameController.startGame();
    gameController.setVerbose(false);
}
```

Code Design:

- SRP (Single Responsibility): levelUp() now focuses purely on logic; logging is treated as optional output, maintaining separation.
- OCP (Open/Closed): By checking verbose, we can extend or modify how logs are handled without altering the core functionality.
- Encapsulation: verbose acts as a flag controlling external behaviour without affecting the core game progression.



```
4 usages
public void levelUp() {
    if (ship.getScore() < lvl * SCORE_THRESHOLD) {
        return;
    }
    lvl++;
    spawnRate += SPAWN_RATE_INCREASE;
    if (verbose) {
        logger.log( text: "Level Up! Welcome to Level " + lvl + ". Spawn rate increased to "
                + spawnRate + "%.");
    }
}
```

7. spawnObjects() - game.GameMode

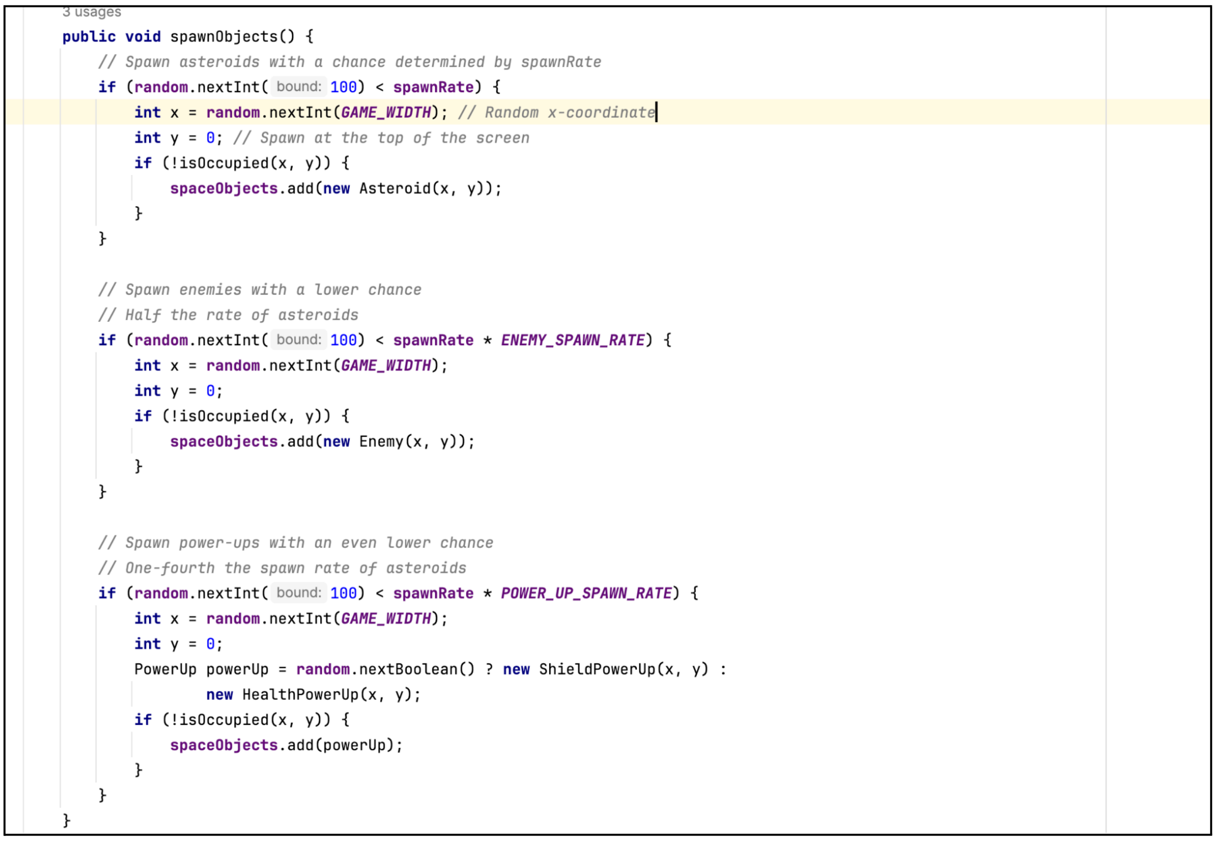
In A1, new objects (Asteroids, Enemies, PowerUps) could only be blocked from spawning if a ship was already at that position. In A2, the check was expanded to also prevent spawning on top of any existing space object. This change ensures objects don't overlap visually or logically during gameplay.

Fixes:

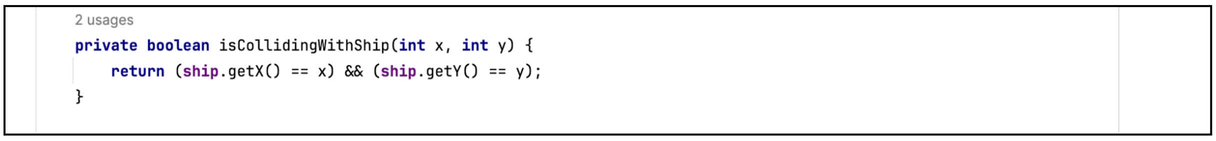
Added a private helper method isOccupied(x, y) which checks if a spawn location is already occupied either by the ship (isCollidingWithShip) or any object in spaceObjects. This fix prevents overlapping spawns and objects now only spawn on clear tiles, improving fairness and visual clarity in gameplay.

Code Design:

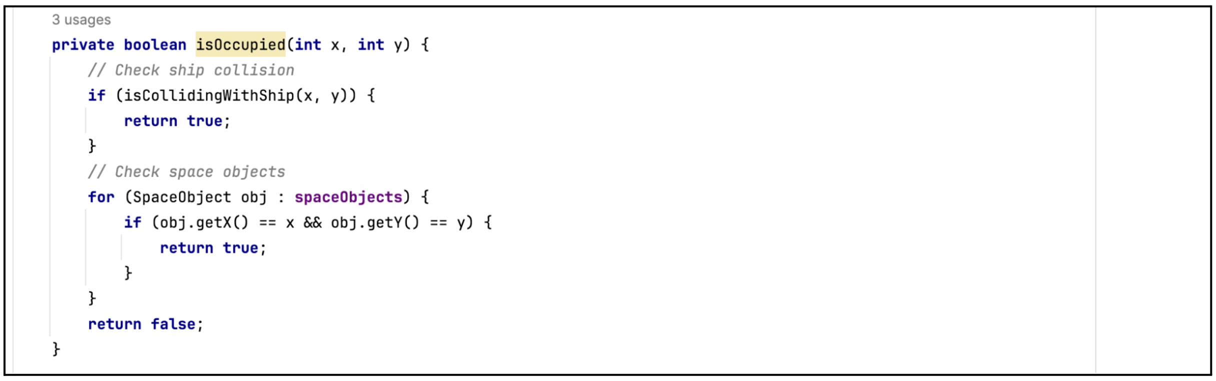
- Single Responsibility Principle (SRP): isOccupied() cleanly separates position-checking logic from spawning logic, improving readability.
- Open/Closed Principle (OCP): Spawning logic can be extended (e.g., new object types) without modifying the core collision logic.



```
3 usages
public void spawnObjects() {
    // 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 (!isOccupied(x, y)) {
            spaceObjects.add(new Asteroid(x, y));
        }
    }

    // Spawn enemies with a lower chance
    // Half the rate of asteroids
    if (random.nextInt( bound: 100) < spawnRate * ENEMY_SPAWN_RATE) {
        int x = random.nextInt(GAME_WIDTH);
        int y = 0;
        if (!isOccupied(x, y)) {
            spaceObjects.add(new Enemy(x, y));
        }
    }

    // Spawn power-ups with an even lower chance
    // One-fourth the spawn rate of asteroids
    if (random.nextInt( bound: 100) < spawnRate * POWER_UP_SPAWN_RATE) {
        int x = random.nextInt(GAME_WIDTH);
        int y = 0;
        PowerUp powerUp = random.nextBoolean() ? new ShieldPowerUp(x, y) :
                new HealthPowerUp(x, y);
        if (!isOccupied(x, y)) {
            spaceObjects.add(powerUp);
        }
    }
}
```



```
2 usages
private boolean isCollidingWithShip(int x, int y) {
    return (ship.getX() == x) && (ship.getY() == y);
}
```



```
3 usages
private boolean isOccupied(int x, int y) {
    // Check ship collision
    if (isCollidingWithShip(x, y)) {
        return true;
    }
    // Check space objects
    for (SpaceObject obj : spaceObjects) {
        if (obj.getX() == x && obj.getY() == y) {
            return true;
        }
    }
    return false;
}
```

8. updateGame() - game.GameModel

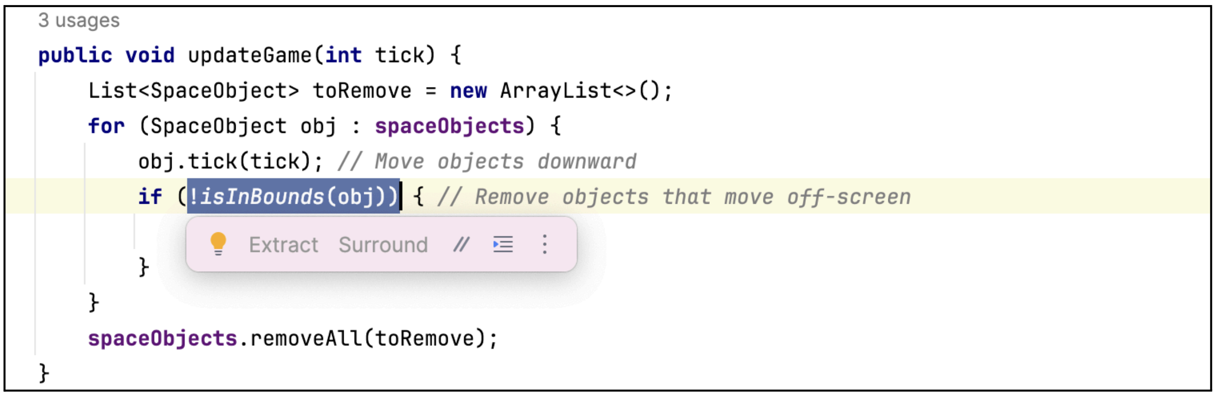
In A1, off-screen detection was hardcoded to check if y > GAME_HEIGHT. Meanwhile in A2, this logic is abstracted into a reusable static method isInBounds(SpaceObject), which checks both x and y boundaries.

Fixes:

Replaced the manual check (y > GAME_HEIGHT) with a call to isInBounds(). This improves consistency across the codebase, centralises boundary logic, and avoids code duplication. All space object removals now use the same rule for boundary checks. This prevents future bugs and makes boundary logic easier to maintain.

Code Design:

- Single Responsibility Principle (SRP): isInBounds() encapsulates boundary-checking logic in one place.
- DRY (Don't Repeat Yourself): Reuses boundary logic instead of rewriting it in each method.



```
3 usages
public void updateGame(int tick) {
    List<SpaceObject> toRemove = new ArrayList<>();
    for (SpaceObject obj : spaceObjects) {
        obj.tick(tick); // Move objects downward
        if (!isInBounds(obj)) { // Remove objects that move off-screen

        }
    }
    spaceObjects.removeAll(toRemove);
}
```

9. handlePlayerInput() - game.GameController

In A1, all inputs are processed and movement is always logged regardless of verbosity, while in A2, inputs are ignored when the game is paused (except "P"), movement logging only occurs if verbose is true, and firing also records a shot via recordShotFired().

Fixes:

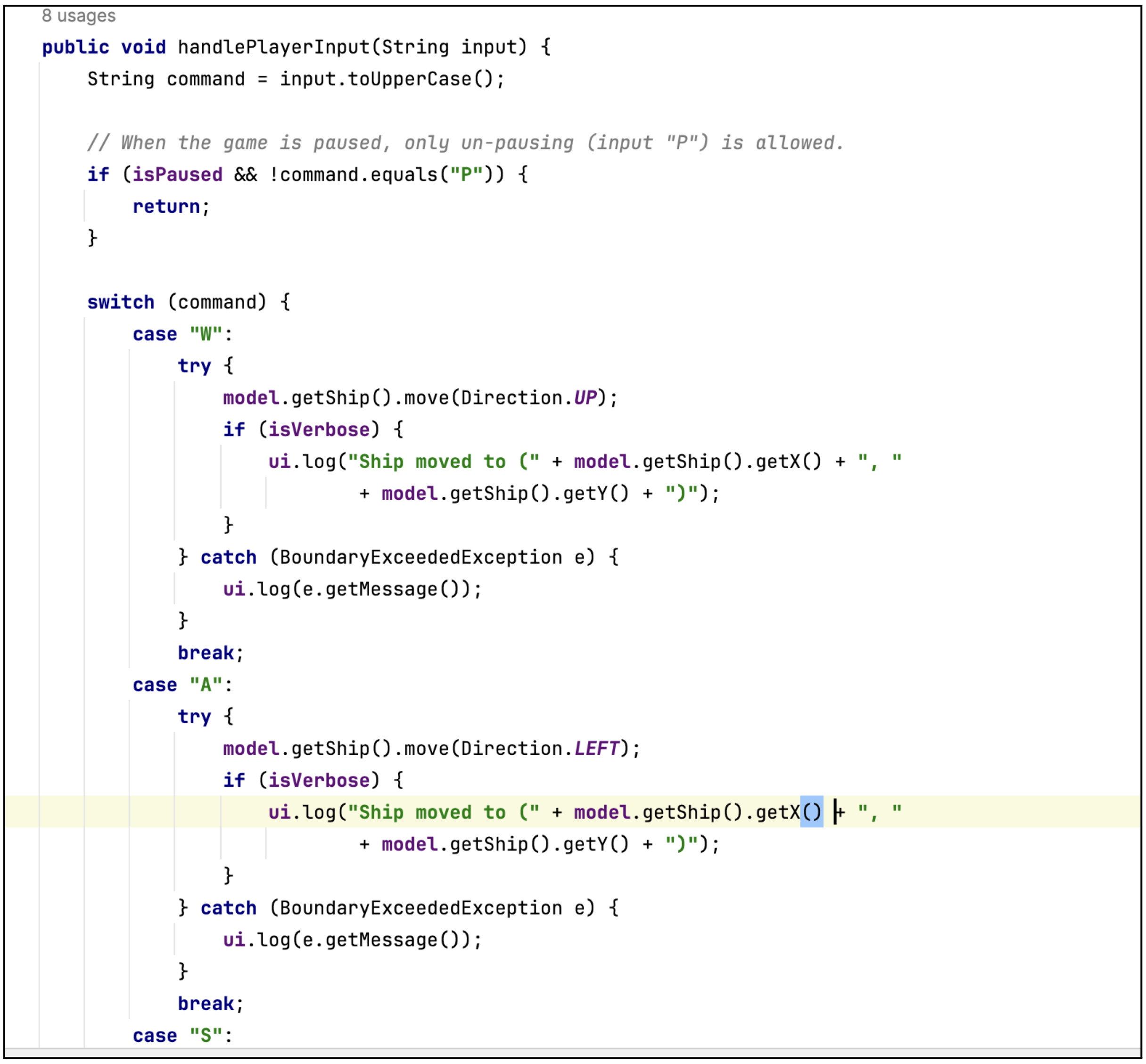
- Checks if the game is paused and ignores inputs except "P".
- Wraps movement logs in if (isVerbose) to reduce noise.
- Tracks bullet shots via statsTracker.recordShotFired() (fixing stat tracking).

This stops unnecessary logs during gameplay and ensures consistent game state behaviour when paused.

Code Design:

- SRP (Single Responsibility): Keeps input handling logic within controller.
- Open/Closed Principle: Extensible input structure via switch-case.

- Encapsulation: Uses isVerbose and isPaused flags to control flow, instead of exposing internal behaviour.

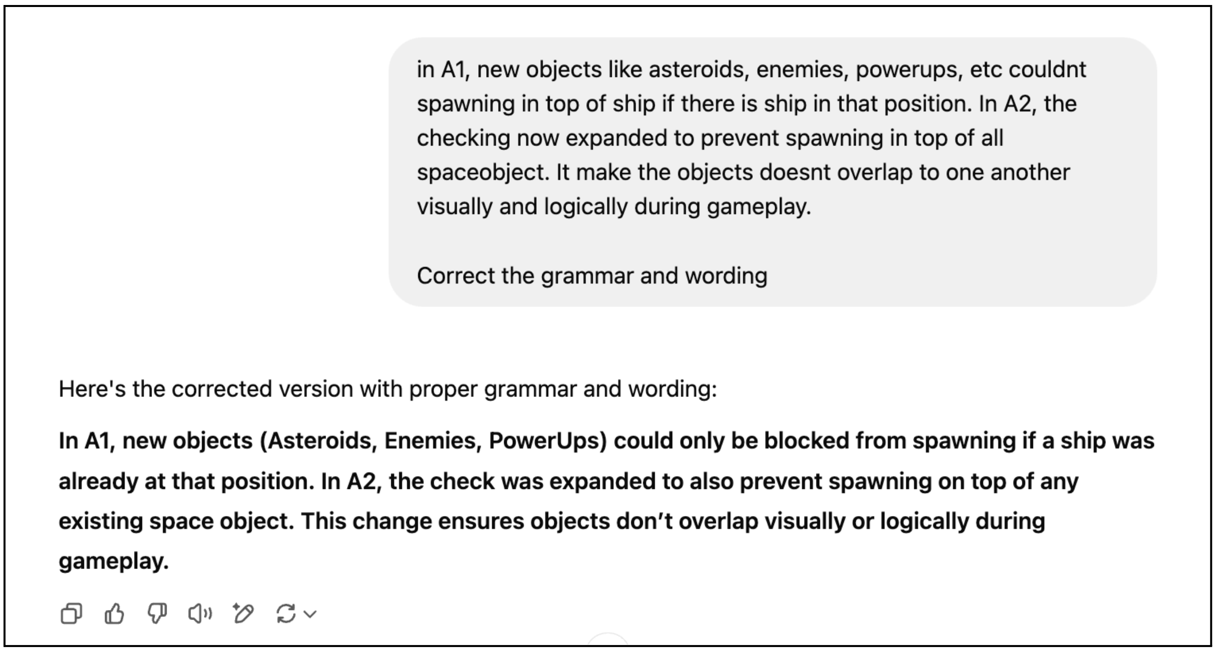


```
8 usages
public void handlePlayerInput(String input) {
    String command = input.toUpperCase();

    // When the game is paused, only un-pausing (input "P") is allowed.
    if (isPaused && !command.equals("P")) {
        return;
    }

    switch (command) {
        case "W":
            try {
                model.getShip().move(Direction.UP);
                if (isVerbose) {
                    ui.log("Ship moved to (" + model.getShip().getX() + ", "
                            + model.getShip().getY() + ")");
                }
            } catch (BoundaryExceededException e) {
                ui.log(e.getMessage());
            }
            break;
        case "A":
            try {
                model.getShip().move(Direction.LEFT);
                if (isVerbose) {
                    ui.log("Ship moved to (" + model.getShip().getX() + ", "
                            + model.getShip().getY() + ")");
                }
            } catch (BoundaryExceededException e) {
                ui.log(e.getMessage());
            }
            break;
        case "S":
```

APPENDIX & AI ACKNOWLEDGMENT

Source:
https://www.geeksforgeeks.org/oop-design-principles-java-programmers/
https://www.jrebel.com/blog/solid-principles-in-java

I acknowledge the use of Generative AI (ChatGPT) throughout the development of this document. AI assistance was used to correct grammar and wording in this document. The prompt used to my writing:



in A1, new objects like asteroids, enemies, powerups, etc couldnt spawning in top of ship if there is ship in that position. In A2, the checking now expanded to prevent spawning in top of all spaceobject. It make the objects doesnt overlap to one another visually and logically during gameplay.

Correct the grammar and wording

Here's the corrected version with proper grammar and wording:

In A1, new objects (Asteroids, Enemies, PowerUps) could only be blocked from spawning if a ship was already at that position. In A2, the check was expanded to also prevent spawning on top of any existing space object. This change ensures objects don't overlap visually or logically during gameplay.