**Comprehensive Testing Plan for the Game World Design**

**Overview**

This testing plan provides detailed test cases for all methods in the classes of my game world design. For each test case, the following details are provided:

* **Condition being tested**
* **Example data used for testing (specific inputs)**
* **Expected outputs or behaviors**

This thorough approach ensures that my code is rigorously tested under various scenarios, including normal cases, edge cases, and error conditions.

**Classes and Methods to Test**

1. **World**
   * World(String filePath)
   * getNeighbors(Space space): List<Space>
   * getSpaceInfo(Space space): String
   * moveTargetCharacter(): void
   * generateWorldMap(): BufferedImage
2. **Space**
   * Space(int id, String name, Coordinate upperLeft, Coordinate lowerRight)
   * getNeighbors(): List<Space>
   * addItem(Item item): void
   * getItems(): List<Item>
   * getVisibleSpaces(): List<Space>
3. **Weapon**
   * Weapon(String name, int damage, Space location)
   * getName(): String
   * getDamage(): int
   * getLocation(): Space
4. **TargetCharacter**
   * TargetCharacter(String name, int health, List<Space> movementPath)
   * move(): void
   * getCurrentSpace(): Space
5. **Coordinate**
   * Coordinate(int row, int column)
   * getRow(): int
   * getColumn(): int

**1. World Class Testing**

**1.1. Testing World(String filePath) Constructor**

**Test Case 1: Valid World File**

* **Condition:** Loading a valid world file.
* **Example Data:**
  + filePath = "worlds/valid\_world.txt"
  + Contents of valid\_world.txt:

**World Name: Fantasy Land**

**Rows: 5**

**Columns: 5**

**Spaces:**

**- id: 1, name: "Town Square", upperLeft: (0,0), lowerRight: (1,1)**

**- id: 2, name: "Forest", upperLeft: (0,2), lowerRight: (1,3)**

**Items:**

**- name: "Sword", damage: 10, locationId: 1**

**TargetCharacter:**

**- name: "Goblin", health: 50, movementPathIds: [1,2]**

* **Expected Outcome:**
  + A World object is created with:
    - name = "Fantasy Land"
    - rows = 5
    - columns = 5
    - spaces list contains two Space objects with correct properties.
    - items list contains one Weapon object.
    - targetCharacter is initialized with specified properties.

**Test Case 2: Invalid File Path**

* **Condition:** Loading a world file from an invalid path.
* **Example Data:**
  + filePath = "worlds/nonexistent.txt"
* **Expected Outcome:**
  + An IOException is thrown with the message: "File not found: worlds/nonexistent.txt"

**Test Case 3: Malformed World File**

* **Condition:** Loading a world file with incorrect formatting.
* **Example Data:**
  + filePath = "worlds/malformed\_world.txt"
  + Contents of malformed\_world.txt:

**World Name: Mystery Land**

**Rows: Five**

**Columns: 5**

* **Expected Outcome:**
  + A ParseException or custom exception is thrown with the message: "Invalid number format for rows in worlds/malformed\_world.txt"

**Test Case 4: Empty World File**

* **Condition:** Loading an empty world file.
* **Example Data:**
  + filePath = "worlds/empty\_world.txt"
  + Contents of empty\_world.txt: *(empty file)*
* **Expected Outcome:**
  + An exception is thrown with the message: "World file worlds/empty\_world.txt is empty or invalid"

**1.2. Testing getNeighbors(Space space): List<Space>**

**Test Case 1: Space with Neighbors**

* **Condition:** Retrieving neighbors for a space surrounded by other spaces.
* **Example Data:**
  + World with a 3x3 grid of spaces.
  + Space at position (1,1) (center of the grid).
* **Expected Outcome:**
  + Returns a list containing the four adjacent Space objects at positions:
    - North: (0,1)
    - South: (2,1)
    - East: (1,2)
    - West: (1,0)

**Test Case 2: Edge Space**

* **Condition:** Space at the edge of the world.
* **Example Data:**
  + Space at position (0,1) (top edge of the grid).
* **Expected Outcome:**
  + Returns a list containing three adjacent Space objects:
    - South: (1,1)
    - East: (0,2)
    - West: (0,0)
    - *(No northern neighbor since it's at the top edge)*

**Test Case 3: Corner Space**

* **Condition:** Space at a corner of the world.
* **Example Data:**
  + Space at position (0,0) (top-left corner).
* **Expected Outcome:**
  + Returns a list containing two adjacent Space objects:
    - South: (1,0)
    - East: (0,1)

**Test Case 4: Null Space Input**

* **Condition:** Passing null as the space parameter.
* **Example Data:**
  + space = null
* **Expected Outcome:**
  + Throws an IllegalArgumentException with message: "Space cannot be null"

**1.3. Testing getSpaceInfo(Space space): String**

**Test Case 1: Space with Items and TargetCharacter**

* **Condition:** Retrieving information for a space containing items and the target character.
* **Example Data:**
  + Space with:
    - id = 1
    - name = "Dungeon"
    - Contains items: ["Sword", "Shield"]
    - Contains targetCharacter
* **Expected Outcome:**
  + Returns the string:

**"Space ID: 1**

**Name: Dungeon**

**Items: Sword, Shield**

**Target Character is here."**

**Test Case 2: Space with No Items**

* **Condition:** Space without any items and without the target character.
* **Example Data:**
  + Space with:
    - id = 2
    - name = "Empty Room"
    - Empty items list
    - Does not contain targetCharacter
* **Expected Outcome:**
  + Returns the string:

**"Space ID: 2**

**Name: Empty Room**

**No items present.**

**Target Character is not here."**

**Test Case 3: Null Space Input**

* **Condition:** Passing null as the space parameter.
* **Example Data:**
  + space = null
* **Expected Outcome:**
  + Throws an IllegalArgumentException with message: "Space cannot be null"

**1.4. Testing moveTargetCharacter(): void**

**Test Case 1: Normal Movement**

* **Condition:** Moving the target character along its movement path.
* **Example Data:**
  + targetCharacter with movementPath = [space1, space2, space3]
  + currentSpace = space1
* **Execution:**
  + Call moveTargetCharacter()
* **Expected Outcome:**
  + targetCharacter.currentSpace is updated to space2

**Test Case 2: End of Movement Path (Loop Back)**

* **Condition:** Target character at the last space; movement path loops.
* **Example Data:**
  + movementPath = [space1, space2, space3]
  + currentSpace = space3
* **Execution:**
  + Call moveTargetCharacter()
* **Expected Outcome:**
  + targetCharacter.currentSpace is updated to space1

**Test Case 3: Empty Movement Path**

* **Condition:** targetCharacter has an empty movement path.
* **Example Data:**
  + movementPath = []
* **Execution:**
  + Call moveTargetCharacter()
* **Expected Outcome:**
  + Throws an IllegalStateException with message: "Movement path is empty"

**Test Case 4: targetCharacter is null**

* **Condition:** World's targetCharacter is not initialized.
* **Example Data:**
  + targetCharacter = null
* **Execution:**
  + Call moveTargetCharacter()
* **Expected Outcome:**
  + Throws an IllegalStateException with message: "TargetCharacter is not initialized"

**1.5. Testing generateWorldMap(): BufferedImage**

**Test Case 1: Normal Map Generation**

* **Condition:** Generating a map for a populated world.
* **Example Data:**
  + World with:
    - rows = 5
    - columns = 5
    - Several Space objects with items and target character.
* **Execution:**
  + Call generateWorldMap()
* **Expected Outcome:**
  + Returns a BufferedImage object.
  + The image visually represents the world grid, with spaces, items, and target character correctly depicted.

**Test Case 2: Empty World**

* **Condition:** Generating a map when the world has no spaces or items.
* **Example Data:**
  + World with:
    - Empty spaces list
* **Execution:**
  + Call generateWorldMap()
* **Expected Outcome:**
  + Returns a BufferedImage object.
  + The image is blank or shows a default background indicating an empty world.

**Test Case 3: Large World**

* **Condition:** Generating a map for a large world.
* **Example Data:**
  + World with:
    - rows = 1000
    - columns = 1000
    - Adequate Space objects to fill the grid.
* **Execution:**
  + Call generateWorldMap()
* **Expected Outcome:**
  + Returns a BufferedImage object without throwing memory or performance-related exceptions.
  + The map correctly represents the large world.

**2. Space Class Testing**

**2.1. Testing Space(int id, String name, Coordinate upperLeft, Coordinate lowerRight) Constructor**

**Test Case 1: Valid Input**

* **Condition:** Creating a Space with valid parameters.
* **Example Data:**
  + id = 1
  + name = "Castle"
  + upperLeft = new Coordinate(0, 0)
  + lowerRight = new Coordinate(5, 5)
* **Expected Outcome:**
  + Space object is created with the specified properties.

**Test Case 2: Negative Coordinates**

* **Condition:** Using negative values for coordinates.
* **Example Data:**
  + upperLeft = new Coordinate(-1, 0)
  + lowerRight = new Coordinate(5, 5)
* **Expected Outcome:**
  + Throws an IllegalArgumentException with message: "Coordinates cannot be negative"

**Test Case 3: Invalid Coordinate Relationship**

* **Condition:** upperLeft coordinates are greater than lowerRight coordinates.
* **Example Data:**
  + upperLeft = new Coordinate(6, 6)
  + lowerRight = new Coordinate(5, 5)
* **Expected Outcome:**
  + Throws an IllegalArgumentException with message: "upperLeft coordinates must be less than or equal to lowerRight coordinates"

**Test Case 4: Null Parameters**

* **Condition:** Passing null for parameters.
* **Example Data:**
  + name = null
  + upperLeft = null
  + lowerRight = new Coordinate(5, 5)
* **Expected Outcome:**
  + Throws an IllegalArgumentException with message: "Name and coordinates cannot be null"

**2.2. Testing getNeighbors(): List<Space>**

**Test Case 1: Space with Neighbors**

* **Condition:** Space surrounded by other spaces.
* **Example Data:**
  + Space at position (2,2) in a World with adjacent spaces at (1,2), (3,2), (2,1), (2,3)
* **Execution:**
  + Call space.getNeighbors()
* **Expected Outcome:**
  + Returns a list containing the four adjacent Space objects.

**Test Case 2: Edge Space**

* **Condition:** Space at the edge of the grid.
* **Example Data:**
  + Space at position (0,2) with no northern neighbor.
* **Execution:**
  + Call space.getNeighbors()
* **Expected Outcome:**
  + Returns a list containing adjacent spaces at (1,2), (0,1), (0,3)

**Test Case 3: Isolated Space**

* **Condition:** Space with no neighbors.
* **Example Data:**
  + Space surrounded by non-space cells or boundaries.
* **Execution:**
  + Call space.getNeighbors()
* **Expected Outcome:**
  + Returns an empty list.

**2.3. Testing addItem(Item item): void**

**Test Case 1: Adding a Valid Item**

* **Condition:** Adding a valid Item to the space.
* **Example Data:**
  + Item is a Weapon with name = "Axe", damage = 15
* **Execution:**
  + Call space.addItem(axeItem)
* **Expected Outcome:**
  + axeItem is added to space.items list.

**Test Case 2: Adding null Item**

* **Condition:** Adding null as the item.
* **Example Data:**
  + item = null
* **Execution:**
  + Call space.addItem(null)
* **Expected Outcome:**
  + Throws an IllegalArgumentException with message: "Item cannot be null"

**Test Case 3: Adding Duplicate Items**

* **Condition:** Adding the same item multiple times.
* **Example Data:**
  + Item is a Weapon named "Bow"
* **Execution:**
  + Call space.addItem(bowItem) twice
* **Expected Outcome:**
  + Depending on implementation:
    - **Option 1:** Both instances are added; space.items.size() == 2
    - **Option 2:** Second addition is ignored; space.items.size() == 1

**2.4. Testing getItems(): List<Item>**

**Test Case 1: Space with Items**

* **Condition:** Retrieving items from a space with items.
* **Example Data:**
  + space.items contains ["Sword", "Shield"]
* **Execution:**
  + Call space.getItems()
* **Expected Outcome:**
  + Returns a list containing the Item objects for "Sword" and "Shield"

**Test Case 2: Space with No Items**

* **Condition:** Space has an empty items list.
* **Execution:**
  + Call space.getItems()
* **Expected Outcome:**
  + Returns an empty list.

**2.5. Testing getVisibleSpaces(): List<Space>**

**Test Case 1: Space with Visible Spaces**

* **Condition:** Spaces visible from current space.
* **Example Data:**
  + Visibility is determined by direct line of sight within a radius of 2 units.
  + Space at (2,2) in a grid; no obstructions.
* **Execution:**
  + Call space.getVisibleSpaces()
* **Expected Outcome:**
  + Returns a list of Space objects within a 2-unit radius from (2,2)

**Test Case 2: Space with Obstructions**

* **Condition:** Obstructions blocking visibility.
* **Example Data:**
  + Walls or blocked spaces around space.
* **Execution:**
  + Call space.getVisibleSpaces()
* **Expected Outcome:**
  + Returns a list of visible spaces, excluding those blocked by obstructions.

**Test Case 3: No Visible Spaces**

* **Condition:** Space completely surrounded by obstructions.
* **Execution:**
  + Call space.getVisibleSpaces()
* **Expected Outcome:**
  + Returns an empty list.

**3. Weapon Class Testing**

**3.1. Testing Weapon(String name, int damage, Space location) Constructor**

**Test Case 1: Valid Input**

* **Condition:** Creating a Weapon with valid parameters.
* **Example Data:**
  + name = "Hammer"
  + damage = 20
  + location = space1
* **Expected Outcome:**
  + Weapon object is created with specified properties.

**Test Case 2: Negative Damage Value**

* **Condition:** Setting damage to a negative number.
* **Example Data:**
  + damage = -5
* **Expected Outcome:**
  + Throws an IllegalArgumentException with message: "Damage value cannot be negative"

**Test Case 3: Null Parameters**

* **Condition:** Passing null for name or location.
* **Example Data:**
  + name = null, damage = 10, location = space1
* **Expected Outcome:**
  + Throws an IllegalArgumentException with message: "Name and location cannot be null"

**3.2. Testing getName(): String**

**Test Case 1: Valid Name**

* **Condition:** Retrieving the name of the weapon.
* **Example Data:**
  + Weapon with name = "Crossbow"
* **Execution:**
  + Call weapon.getName()
* **Expected Outcome:**
  + Returns "Crossbow"

**3.3. Testing getDamage(): int**

**Test Case 1: Valid Damage Value**

* **Condition:** Retrieving the damage value.
* **Example Data:**
  + Weapon with damage = 25
* **Execution:**
  + Call weapon.getDamage()
* **Expected Outcome:**
  + Returns 25

**3.4. Testing getLocation(): Space**

**Test Case 1: Valid Location**

* **Condition:** Retrieving the location of the weapon.
* **Example Data:**
  + Weapon located in space2
* **Execution:**
  + Call weapon.getLocation()
* **Expected Outcome:**
  + Returns space2

**4. TargetCharacter Class Testing**

**4.1. Testing TargetCharacter(String name, int health, List<Space> movementPath) Constructor**

**Test Case 1: Valid Input**

* **Condition:** Creating a TargetCharacter with valid parameters.
* **Example Data:**
  + name = "Dragon"
  + health = 200
  + movementPath = [spaceA, spaceB, spaceC]
* **Expected Outcome:**
  + TargetCharacter object is created with specified properties.
  + currentSpace is set to spaceA

**Test Case 2: Negative Health**

* **Condition:** Setting health to a negative value.
* **Example Data:**
  + health = -50
* **Expected Outcome:**
  + Throws an IllegalArgumentException with message: "Health cannot be negative"

**Test Case 3: Empty Movement Path**

* **Condition:** Providing an empty movementPath.
* **Example Data:**
  + movementPath = []
* **Expected Outcome:**
  + Throws an IllegalArgumentException with message: "Movement path cannot be empty"

**Test Case 4: Null Parameters**

* **Condition:** Passing null for name or movementPath.
* **Example Data:**
  + name = null, health = 100, movementPath = [spaceA, spaceB]
* **Expected Outcome:**
  + Throws an IllegalArgumentException with message: "Name and movement path cannot be null"

**4.2. Testing move(): void**

**Test Case 1: Normal Movement**

* **Condition:** Moving along the movement path.
* **Example Data:**
  + movementPath = [spaceA, spaceB, spaceC]
  + currentSpace = spaceA
* **Execution:**
  + Call targetCharacter.move()
* **Expected Outcome:**
  + currentSpace updates to spaceB

**Test Case 2: At End of Movement Path (Loop Back)**

* **Condition:** At the last space; movement path loops.
* **Example Data:**
  + currentSpace = spaceC
* **Execution:**
  + Call targetCharacter.move()
* **Expected Outcome:**
  + currentSpace updates to spaceA

**Test Case 3: Modified Movement Path During Movement**

* **Condition:** movementPath changes during movement.
* **Example Data:**
  + movementPath initially [spaceA, spaceB]
  + After one move, update movementPath to [spaceA, spaceB, spaceD]
* **Execution:**
  + Call targetCharacter.move() twice
* **Expected Outcome:**
  + First move: currentSpace is spaceB
  + Update movementPath
  + Second move: currentSpace is spaceD

**Test Case 4: Empty Movement Path During Movement**

* **Condition:** movementPath becomes empty during movement.
* **Execution:**
  + Remove all elements from movementPath
  + Call targetCharacter.move()
* **Expected Outcome:**
  + Throws an IllegalStateException with message: "Movement path cannot be empty"

**4.3. Testing getCurrentSpace(): Space**

**Test Case 1: After Initialization**

* **Condition:** Immediately after creation.
* **Execution:**
  + Call targetCharacter.getCurrentSpace()
* **Expected Outcome:**
  + Returns the first space in movementPath

**Test Case 2: After Movement**

* **Condition:** After moving the character.
* **Execution:**
  + Call targetCharacter.move()
  + Call targetCharacter.getCurrentSpace()
* **Expected Outcome:**
  + Returns the updated currentSpace

**5. Coordinate Class Testing**

**5.1. Testing Coordinate(int row, int column) Constructor**

**Test Case 1: Valid Coordinates**

* **Condition:** Creating with positive integers.
* **Example Data:**
  + row = 5, column = 10
* **Expected Outcome:**
  + Coordinate object is created with row = 5, column = 10

**Test Case 2: Zero Coordinates**

* **Condition:** Using zero for row and column.
* **Example Data:**
  + row = 0, column = 0
* **Expected Outcome:**
  + Coordinate object is created at origin.

**Test Case 3: Negative Coordinates**

* **Condition:** Negative values for row or column.
* **Example Data:**
  + row = -3, column = 5
* **Expected Outcome:**
  + Throws an IllegalArgumentException with message: "Row and column must be non-negative integers"

**Test Case 4: Large Coordinates**

* **Condition:** Using large integer values.
* **Example Data:**
  + row = Integer.MAX\_VALUE, column = Integer.MAX\_VALUE
* **Expected Outcome:**
  + Coordinate object is created with large values.

**5.2. Testing getRow(): int**

**Test Case 1: Valid Coordinate**

* **Condition:** Retrieving row value.
* **Example Data:**
  + Coordinate with row = 7, column = 8
* **Execution:**
  + Call coordinate.getRow()
* **Expected Outcome:**
  + Returns 7

**5.3. Testing getColumn(): int**

**Test Case 1: Valid Coordinate**

* **Condition:** Retrieving column value.
* **Example Data:**
  + Coordinate with row = 7, column = 8
* **Execution:**
  + Call coordinate.getColumn()
* **Expected Outcome:**
  + Returns 8

**6. Additional Considerations**

**6.1. Testing equals() and hashCode() (If Implemented)**

**Test Case 1: equals() with Identical Coordinates**

* **Condition:** Comparing two Coordinate objects with the same values.
* **Example Data:**
  + coordinate1 = new Coordinate(5, 5)
  + coordinate2 = new Coordinate(5, 5)
* **Execution:**
  + coordinate1.equals(coordinate2)
* **Expected Outcome:**
  + Returns true

**Test Case 2: hashCode() Consistency**

* **Condition:** Two equal objects have the same hash code.
* **Execution:**
  + coordinate1.hashCode() == coordinate2.hashCode()
* **Expected Outcome:**
  + Returns true

**Test Case 3: equals() with Different Coordinates**

* **Condition:** Comparing Coordinate objects with different values.
* **Example Data:**
  + coordinate1 = new Coordinate(5, 5)
  + coordinate2 = new Coordinate(5, 6)
* **Execution:**
  + coordinate1.equals(coordinate2)
* **Expected Outcome:**
  + Returns false

**6.2. Testing toString() Methods (If Implemented)**

**Test Case 1: Coordinate toString()**

* **Condition:** Checking string representation.
* **Example Data:**
  + Coordinate with row = 3, column = 4
* **Execution:**
  + coordinate.toString()
* **Expected Outcome:**
  + Returns "Coordinate(row=3, column=4)"

**Test Case 2: Space toString()**

* **Condition:** Checking string representation.
* **Example Data:**
  + Space with id = 1, name = "Garden"
* **Execution:**
  + space.toString()
* **Expected Outcome:**
  + Returns "Space(id=1, name='Garden')"