**Testing Plan for Player Interface**

**Responsibilities:**  
Represents a generic player in the game, with methods that are implemented in both HumanPlayer and ComputerPlayer.

**Methods:**

**1. getName()**

* **Test Case 1: Valid Player Name**
  + **Condition:** Verifying that the correct player name is returned.
  + **Example Data:** Player name = "Alice".
  + **Expected Outcome:** Returns "Alice".
* **Test Case 2: Empty Player Name**
  + **Condition:** Verifying how the system handles an empty player name.
  + **Example Data:** Player name = "".
  + **Expected Outcome:** Returns an empty string "".
* **Test Case 3: Null Player Name**
  + **Condition:** Verifying the system’s behavior when the player name is null.
  + **Example Data:** Player name = null.
  + **Expected Outcome:** Throws a NullPointerException or returns a default name.

**2. getCurrentSpace()**

* **Test Case 1: Player in Valid Space**
  + **Condition:** Checking if the player's current space is returned correctly.
  + **Example Data:** Player located in Living Room.
  + **Expected Outcome:** Returns the Living Room space object.
* **Test Case 2: Player in No Space**
  + **Condition:** Checking if a player without an assigned space returns null.
  + **Example Data:** Player created but not placed in any space.
  + **Expected Outcome:** Returns null.
* **Test Case 3: Player Moved Between Spaces**
  + **Condition:** Verifying the space after the player moves.
  + **Example Data:** Player moves from Living Room to Kitchen.
  + **Expected Outcome:** Returns Kitchen after the move.

**3. getItems()**

* **Test Case 1: Empty Inventory**
  + **Condition:** When the player has no items.
  + **Example Data:** Player just joined the game with an empty inventory.
  + **Expected Outcome:** Returns an empty list.
* **Test Case 2: Non-empty Inventory**
  + **Condition:** Verifying items when the player carries items.
  + **Example Data:** Player holds Sword and Shield.
  + **Expected Outcome:** Returns a list containing Sword and Shield.
* **Test Case 3: Picking Up Items**
  + **Condition:** Verifying that items are added correctly to the inventory.
  + **Example Data:** Player picks up a Potion.
  + **Expected Outcome:** Returns a list containing Potion.

**4. getMaxItems()**

* **Test Case 1: Maximum Capacity of Items**
  + **Condition:** Verifying the maximum number of items a player can carry.
  + **Example Data:** maxItems = 5.
  + **Expected Outcome:** Returns 5.
* **Test Case 2: Changing Max Item Capacity**
  + **Condition:** Changing the max item capacity dynamically.
  + **Example Data:** Player’s capacity changes from 5 to 3.
  + **Expected Outcome:** Returns 3.

**5. move(ImSpace destination)**

* **Test Case 1: Valid Move to Neighboring Space**
  + **Condition:** Moving a player from one space to a valid neighboring space.
  + **Example Data:** Current space = Living Room, Destination = Kitchen (a neighbor).
  + **Expected Outcome:** Player successfully moves to Kitchen.
* **Test Case 2: Invalid Move to Non-neighboring Space**
  + **Condition:** Player attempts to move to a non-neighboring space.
  + **Example Data:** Current space = Living Room, Destination = Attic (not a neighbor).
  + **Expected Outcome:** Throws an InvalidMoveException.
* **Test Case 3: Move with Null Space**
  + **Condition:** Verifying behavior when the destination space is null.
  + **Example Data:** Player tries to move to a null destination.
  + **Expected Outcome:** Throws a NullPointerException.

**6. pickUpItem(ImItem item)**

* **Test Case 1: Successful Item Pickup**
  + **Condition:** Player picks up an item when they have capacity.
  + **Example Data:** Player in Kitchen, finds Knife.
  + **Expected Outcome:** Player successfully picks up the Knife.
* **Test Case 2: Inventory Full**
  + **Condition:** Player cannot pick up an item when the inventory is full.
  + **Example Data:** Player's inventory is full (5/5 items).
  + **Expected Outcome:** Throws InventoryFullException.
* **Test Case 3: Pick Up Non-existent Item**
  + **Condition:** Player attempts to pick up an item not present in the space.
  + **Example Data:** Item Axe is not present in the space.
  + **Expected Outcome:** Throws an ItemNotFoundException.

**7. lookAround()**

* **Test Case 1: Spaces with Neighbors**
  + **Condition:** Player looks around and sees neighboring spaces.
  + **Example Data:** Player in Living Room with neighbors Kitchen, Dining Room.
  + **Expected Outcome:** Displays Kitchen and Dining Room.
* **Test Case 2: Space with No Neighbors**
  + **Condition:** Player in an isolated space with no neighbors.
  + **Example Data:** Player in Attic with no adjacent spaces.
  + **Expected Outcome:** Displays a message indicating no neighbors.
* **Test Case 3: Player Without Assigned Space**
  + **Condition:** Player looks around when they are not placed in a space.
  + **Example Data:** Player not yet placed in any space.
  + **Expected Outcome:** Throws PlayerNotInSpaceException.

**8. takeTurn()**

* **Test Case 1: Player Takes a Turn**
  + **Condition:** Verifying the flow when the player takes a turn.
  + **Example Data:** Player chooses to move to Kitchen during their turn.
  + **Expected Outcome:** The player successfully moves to Kitchen.
* **Test Case 2: Player Skips Turn**
  + **Condition:** Player decides to skip their turn.
  + **Example Data:** Player input for skip turn.
  + **Expected Outcome:** The game proceeds to the next player's turn.
* **Test Case 3: Player Chooses Invalid Action**
  + **Condition:** Player selects an invalid action (e.g., fly).
  + **Example Data:** Player tries to fly to space.
  + **Expected Outcome:** Displays an error message and prompts the player to select a valid action.

**Testing Plan for HumanPlayer Class**

**Responsibilities:**  
Implements the Player interface for human-controlled players. User input drives actions.

**Attributes:**

* String name; – The player's name.
* ImSpace currentSpace; – The space where the player is currently located.
* List<ImItem> items; – A list of items the player has collected.
* int maxItems; – The maximum number of items the player can carry.

**Methods:**

**1. getName()**

(Same as in Player interface, already tested in Player Interface.)

**2. getCurrentSpace()**

(Same as in Player interface, already tested in Player Interface.)

**3. getItems()**

(Same as in Player interface, already tested in Player Interface.)

**4. getMaxItems()**

(Same as in Player interface, already tested in Player Interface.)

**5. move(ImSpace destination)**

* **Test Case 1: Valid Move**
  + **Condition:** Player moves to a valid neighboring space.
  + **Example Data:** Player moves from Living Room to Kitchen.
  + **Expected Outcome:** Player’s current space is updated to Kitchen.
* **Test Case 2: Invalid Move**
  + **Condition:** Player tries to move to a non-neighboring space.
  + **Example Data:** Player in Living Room tries to move to Attic (not a neighbor).
  + **Expected Outcome:** Throws an InvalidMoveException and keeps player in Living Room.
* **Test Case 3: Move with Full Inventory**
  + **Condition:** Player attempts to move while their inventory is full, but they can still move.
  + **Example Data:** Player with a full inventory moves from Library to Dining Room.
  + **Expected Outcome:** Player successfully moves to Dining Room.

**6. pickUpItem(ImItem item)**

* **Test Case 1: Successful Item Pickup**
  + **Condition:** Player picks up an item when inventory space is available.
  + **Example Data:** Player picks up Sword from Armory.
  + **Expected Outcome:** Sword is added to the player’s inventory.
* **Test Case 2: Inventory Full**
  + **Condition:** Player tries to pick up an item but the inventory is full.
  + **Example Data:** Player with 5 items in inventory tries to pick up Shield.
  + **Expected Outcome:** Throws InventoryFullException.
* **Test Case 3: Player Picks Up Non-existent Item**
  + **Condition:** Player attempts to pick up an item that does not exist in the space.
  + **Example Data:** Player tries to pick up a Magic Wand from Library, but it’s not there.
  + **Expected Outcome:** Throws ItemNotFoundException.

**7. lookAround()**

* **Test Case 1: Look Around with Neighbors**
  + **Condition:** Player looks around and sees neighboring spaces.
  + **Example Data:** Player in Living Room sees Kitchen and Dining Room.
  + **Expected Outcome:** Displays information about Kitchen and Dining Room.
* **Test Case 2: Look Around with No Neighbors**
  + **Condition:** Player in an isolated space with no neighbors looks around.
  + **Example Data:** Player in Attic with no neighboring spaces.
  + **Expected Outcome:** Displays "No neighbors found".

**8. takeTurn()**

* **Test Case 1: Player Takes a Valid Action**
  + **Condition:** Player moves to a neighboring space on their turn.
  + **Example Data:** Player selects the action to move from Library to Hallway.
  + **Expected Outcome:** Player successfully moves to Hallway.
* **Test Case 2: Player Chooses to Pick Up an Item**
  + **Condition:** Player picks up an item during their turn.
  + **Example Data:** Player picks up Key from Basement.
  + **Expected Outcome:** Key is added to the player’s inventory.
* **Test Case 3: Player Chooses Invalid Action**
  + **Condition:** Player tries to perform an invalid action (e.g., fly).
  + **Example Data:** Player selects fly as an action.
  + **Expected Outcome:** Throws InvalidActionException and prompts for a valid action.

**9. dropItem(ImItem item)**

(*New method for HumanPlayer*)

* **Test Case 1: Drop Valid Item**
  + **Condition:** Player drops an item from their inventory.
  + **Example Data:** Player drops Torch in Living Room.
  + **Expected Outcome:** Torch is removed from the player’s inventory and added to the room.
* **Test Case 2: Drop Item Not in Inventory**
  + **Condition:** Player tries to drop an item they do not have.
  + **Example Data:** Player tries to drop a Helmet they don't have.
  + **Expected Outcome:** Throws ItemNotFoundException.

**10. interactWithPlayer(Player player)**

(*New method for HumanPlayer*)

* **Test Case 1: Interact with Another Player in the Same Space**
  + **Condition:** Player interacts with another player in the same room.
  + **Example Data:** Player Alice interacts with Bob in the Kitchen.
  + **Expected Outcome:** Interaction occurs, such as trading or exchanging items.
* **Test Case 2: Interact with Player in Different Space**
  + **Condition:** Player tries to interact with a player who is not in the same room.
  + **Example Data:** Alice tries to interact with Bob who is in a different space.
  + **Expected Outcome:** Throws PlayerNotInSameSpaceException.

**Testing Plan for ComputerPlayer Class**

**Responsibilities:**  
Implements the Player interface for computer-controlled players. Actions are driven by automated logic.

**Attributes:**

* String name; – The AI player's name.
* ImSpace currentSpace; – The space where the player is located.
* List<ImItem> items; – Items collected by the player.
* int maxItems; – The maximum number of items the AI player can carry.

**Methods:**

**1. getName()**

(Same as in Player interface, already tested in Player Interface.)

**2. getCurrentSpace()**

(Same as in Player interface, already tested in Player Interface.)

**3. getItems()**

(Same as in Player interface, already tested in Player Interface.)

**4. getMaxItems()**

(Same as in Player interface, already tested in Player Interface.)

**5. move(ImSpace destination)**

* **Test Case 1: Automatic Move to Neighboring Space**
  + **Condition:** AI moves automatically to a neighboring space.
  + **Example Data:** AI moves from Living Room to Kitchen.
  + **Expected Outcome:** Player’s current space is updated to Kitchen without manual input.
* **Test Case 2: Invalid Move**
  + **Condition:** AI tries to move to a space that isn’t a neighbor due to a logic error.
  + **Example Data:** AI tries to move from Living Room to Attic.
  + **Expected Outcome:** Throws InvalidMoveException, keeping the AI in Living Room.

**6. pickUpItem(ImItem item)**

* **Test Case 1: Automatic Item Pickup**
  + **Condition:** AI automatically picks up an item when in a space with available items.
  + **Example Data:** AI player picks up a Key from the Hallway.
  + **Expected Outcome:** Key is added to AI's inventory.
* **Test Case 2: Inventory Full**
  + **Condition:** AI attempts to pick up an item when their inventory is full.
  + **Example Data:** AI tries to pick up Lantern but already has a full inventory.
  + **Expected Outcome:** AI does not pick up the item, and it remains in the space.

**7. lookAround()**

* **Test Case 1: AI Looks Around with Neighbors**
  + **Condition:** AI automatically scans the nearby spaces for information.
  + **Example Data:** AI in Library looks at Kitchen and Study.
  + **Expected Outcome:** Information about Kitchen and Study is logged or processed by the AI.
* **Test Case 2: AI Looks Around with No Neighbors**
  + **Condition:** AI is in an isolated room with no neighbors.
  + **Example Data:** AI in the Attic looks around.
  + **Expected Outcome:** Logs or processes that there are no neighboring spaces.

**8. takeTurn()**

* **Test Case 1: AI Decides to Move**
  + **Condition:** AI chooses to move to a neighboring space based on decision logic.
  + **Example Data:** AI in Living Room moves to Kitchen as part of their turn.
  + **Expected Outcome:** AI successfully moves to Kitchen and the next player's turn begins.
* **Test Case 2: AI Decides to Pick Up an Item**
  + **Condition:** AI chooses to pick up an item during their turn.
  + **Example Data:** AI picks up Key from Basement automatically.
  + **Expected Outcome:** Key is added to the AI’s inventory, and the next player's turn begins.
* **Test Case 3: AI Makes a Decision with No Available Actions**
  + **Condition:** AI cannot move or pick up items (e.g., inventory full and no neighbors).
  + **Example Data:** AI in Attic with a full inventory.
  + **Expected Outcome:** AI skips its turn and logs the reason for no action.

**9. evaluateOptions()**

(*New method for ComputerPlayer*)

* **Test Case 1: Evaluate Move vs Pick Up**
  + **Condition:** AI has the choice to move or pick up an item.
  + **Example Data:** AI in Hallway has a Torch available and neighboring Library.
  + **Expected Outcome:** AI evaluates the situation and makes a decision (either moves or picks up the item).
* **Test Case 2: Evaluate No Valid Actions**
  + **Condition:** AI evaluates options when it has no valid moves or available items.
  + **Example Data:** AI is in Attic with a full inventory and no neighbors.
  + **Expected Outcome:** AI skips the turn, logs the situation.

**10. interactWithTargetCharacter()**

(*New method for ComputerPlayer*)

* **Test Case 1: Interaction with Target Character in the Same Space**
  + **Condition:** AI encounters the target character in the same space and decides how to interact.
  + **Example Data:** AI and target character meet in the Garden.
  + **Expected Outcome:** AI interacts with the target character, possibly triggering a special event or action.

**11. dropItem(ImItem item)**

(*New method for ComputerPlayer*)

* **Test Case 1: Drop Item in Space**
  + **Condition:** AI decides to drop an item from its inventory due to inventory space considerations.
  + **Example Data:** AI drops a Lantern in the Library to free space.
  + **Expected Outcome:** Lantern is removed from AI’s inventory and added to the Library.
* **Test Case 2: Drop Non-Existent Item**
  + **Condition:** AI tries to drop an item it doesn’t have.
  + **Example Data:** AI tries to drop a Shield it hasn’t picked up.
  + **Expected Outcome:** Throws ItemNotFoundException.

**Testing Plan for World Class**

**Responsibilities:**  
The main class that manages spaces, players, items, and the target character. Responsible for running the game logic and turns.

**Attributes:**

* int rows; – Number of rows in the world grid.
* int cols; – Number of columns in the world grid.
* String name; – The name of the world.
* List<ImSpace> spaces; – List of spaces in the world.
* List<ImItem> items; – List of items in the world.
* List<Player> players; – List of all players in the game.
* ImTargetCharacter targetCharacter; – The target character moving automatically.

**Methods:**

**1. getNeighbors(ImSpace space)**

* **Test Case 1: Get Neighbors for a Space with Adjacent Neighbors**
  + **Condition:** Returns all neighbors that share a wall with a given space.
  + **Example Data:** For Room A, neighbors are Room B and Room C.
  + **Expected Outcome:** The method returns Room B and Room C as neighbors.
* **Test Case 2: No Neighbors for Isolated Space**
  + **Condition:** Check if the method correctly handles an isolated space with no neighbors.
  + **Example Data:** Attic has no adjacent neighbors.
  + **Expected Outcome:** Returns an empty list, indicating no neighbors.

**2. getSpaceInfo(ImSpace space)**

* **Test Case 1: Display Information for Space with Players and Items**
  + **Condition:** Displays all relevant information about a space.
  + **Example Data:** For Living Room, two players are present, and there are two items (Key, Sword).
  + **Expected Outcome:** Displays space name, list of players, and list of items.
* **Test Case 2: Display Information for Empty Space**
  + **Condition:** A space that has no players or items.
  + **Example Data:** For Library, there are no players or items.
  + **Expected Outcome:** Displays space name and indicates no players or items.

**3. moveTargetCharacter()**

* **Test Case 1: Target Character Moves to the Next Space**
  + **Condition:** Moves the target character sequentially across spaces.
  + **Example Data:** Target character moves from Room A to Room B after a player's turn.
  + **Expected Outcome:** Target character is now located in Room B.
* **Test Case 2: Target Character Wraps Around to First Space**
  + **Condition:** Target character moves to the first space when it reaches the last space.
  + **Example Data:** Target character moves from Room Z (last space) to Room A.
  + **Expected Outcome:** Target character is now located in Room A.

**4. addPlayer(Player player)**

* **Test Case 1: Add a Human Player to the World**
  + **Condition:** A new human player is added to the world.
  + **Example Data:** Player1 enters the world and starts in Room A.
  + **Expected Outcome:** Player1 is added to the list of players, and their current space is Room A.
* **Test Case 2: Add a Computer Player to the World**
  + **Condition:** A new computer player is added to the world.
  + **Example Data:** AIPlayer1 enters the world and starts in Room B.
  + **Expected Outcome:** AIPlayer1 is added to the list of players, and their current space is Room B.

**5. takeTurn()**

* **Test Case 1: Sequential Player Turns**
  + **Condition:** Players take turns in the order they were added to the game.
  + **Example Data:** Player1 goes first, followed by Player2.
  + **Expected Outcome:** Player1 takes their turn, followed by Player2.
* **Test Case 2: Automatic Target Character Movement During Player Turns**
  + **Condition:** The target character moves after every player's turn.
  + **Example Data:** Target character starts in Room A. After Player1 and Player2 take turns, the target character moves to Room B.
  + **Expected Outcome:** Target character moves to the next space automatically after each turn.

**6. displayPlayerDetails(Player player)**

* **Test Case 1: Display Player Information with Items and Location**
  + **Condition:** Displays detailed information about a player.
  + **Example Data:** Player1 is in Room A with Sword and Key.
  + **Expected Outcome:** Displays Player1's name, current location (Room A), and their items (Sword, Key).
* **Test Case 2: Display Player Information with No Items**
  + **Condition:** A player has no items.
  + **Example Data:** Player2 is in Room B with no items.
  + **Expected Outcome:** Displays Player2's name, current location (Room B), and indicates no items.

**7. generateMap()**

* **Test Case 1: Generate a Graphical Representation of the World**
  + **Condition:** Generates a BufferedImage representing the entire world.
  + **Example Data:** The world map is generated for a 5x5 grid with all spaces represented.
  + **Expected Outcome:** A BufferedImage object is created, displaying all spaces and their contents.
* **Test Case 2: Empty World**
  + **Condition:** The world contains no spaces or items.
  + **Example Data:** The world map is generated for an empty world (0x0 grid).
  + **Expected Outcome:** An empty BufferedImage object is created, with no spaces or items.

**Testing Plan for Space Class**

**Responsibilities:**  
Represents a space/room within the world. Tracks players and items in the space.

**Attributes:**

* String name; – The name of the space.
* int[] coordinates; – Coordinates (upper-left and lower-right) of the space.
* List<ImItem> items; – Items present in the space.
* List<Player> players; – Players currently in this space.

**Methods:**

**1. addItem(ImItem item)**

* **Test Case 1: Add an Item to the Space**
  + **Condition:** Adds an item to the space.
  + **Example Data:** The Living Room space, item is Sword.
  + **Expected Outcome:** Sword is added to the list of items in the Living Room.
* **Test Case 2: Add Multiple Items to the Space**
  + **Condition:** Adds multiple items to the space.
  + **Example Data:** The Kitchen space, items are Knife and Plate.
  + **Expected Outcome:** Both Knife and Plate are added to the Kitchen space.

**2. getItems()**

* **Test Case 1: Get Items in a Space with Items**
  + **Condition:** Retrieves all items in a space that contains items.
  + **Example Data:** The Bedroom has Lamp and Book.
  + **Expected Outcome:** Returns the list of items, Lamp and Book, in the Bedroom.
* **Test Case 2: Get Items in an Empty Space**
  + **Condition:** Retrieves items in a space that has no items.
  + **Example Data:** The Bathroom contains no items.
  + **Expected Outcome:** Returns an empty list for the Bathroom.

**3. addPlayer(Player player)**

* **Test Case 1: Add a Player to a Space**
  + **Condition:** Adds a player to a space.
  + **Example Data:** Player1 enters the Dining Room.
  + **Expected Outcome:** Player1 is added to the list of players in the Dining Room.
* **Test Case 2: Add Multiple Players to a Space**
  + **Condition:** Adds multiple players to the same space.
  + **Example Data:** Player1 and Player2 enter the Garage.
  + **Expected Outcome:** Both Player1 and Player2 are added to the list of players in the Garage.

**4. removePlayer(Player player)**

* **Test Case 1: Remove a Player from a Space**
  + **Condition:** Removes a player from the space.
  + **Example Data:** Player1 leaves the Living Room.
  + **Expected Outcome:** Player1 is removed from the list of players in the Living Room.
* **Test Case 2: Remove a Player Not in the Space**
  + **Condition:** Tries to remove a player that is not in the space.
  + **Example Data:** Attempt to remove Player3 from the Library (where Player3 is not present).
  + **Expected Outcome:** No change in the list of players in the Library.

**5. getName()**

* **Test Case 1: Get the Name of the Space**
  + **Condition:** Retrieves the name of the space.
  + **Example Data:** The space is called Kitchen.
  + **Expected Outcome:** Returns the name Kitchen.

**6. getCoordinates()**

* **Test Case 1: Get the Coordinates of a Space**
  + **Condition:** Retrieves the coordinates of a space.
  + **Example Data:** Living Room has coordinates [1, 1, 2, 2].
  + **Expected Outcome:** Returns the coordinates [1, 1, 2, 2] for the Living Room.

**7. getPlayers()**

* **Test Case 1: Get Players in a Space**
  + **Condition:** Retrieves all players currently in a space.
  + **Example Data:** Player1 and Player2 are in the Garage.
  + **Expected Outcome:** Returns Player1 and Player2 as the list of players in the Garage.
* **Test Case 2: Get Players in an Empty Space**
  + **Condition:** Retrieves players from a space with no players.
  + **Example Data:** The Basement has no players.
  + **Expected Outcome:** Returns an empty list for the Basement.

**Testing Plan for Item Class**

**Responsibilities:**  
Represents an item that players can find and use in the world.

**Attributes:**

* String name; – The name of the item.
* int damage; – The damage an item can inflict if used.

**Methods:**

**1. getDamage()**

* **Test Case 1: Retrieve Damage Value of an Item**
  + **Condition:** Retrieves the damage value of an item.
  + **Example Data:** The item Sword has a damage value of 10.
  + **Expected Outcome:** Returns 10 for the Sword.
* **Test Case 2: Retrieve Damage Value for an Item with No Damage**
  + **Condition:** Retrieves the damage for an item that does no damage.
  + **Example Data:** The item Candle has a damage value of 0.
  + **Expected Outcome:** Returns 0 for the Candle.

**2. getName()**

* **Test Case 1: Retrieve Name of the Item**
  + **Condition:** Retrieves the name of an item.
  + **Example Data:** The item is called Dagger.
  + **Expected Outcome:** Returns the name Dagger.
* **Test Case 2: Retrieve Name for an Item with an Empty Name**
  + **Condition:** Retrieves the name for an item with an empty string name.
  + **Example Data:** The item has an empty name "".
  + **Expected Outcome:** Returns an empty string "".

**3. equals(Object obj)**

This is crucial for comparing items, ensuring two items with the same name and damage are considered equal.

* **Test Case 1: Compare Two Identical Items**
  + **Condition:** Compares two items with the same name and damage.
  + **Example Data:** Two Sword items, each with damage 10.
  + **Expected Outcome:** Returns true, indicating the items are equal.
* **Test Case 2: Compare Two Different Items**
  + **Condition:** Compares two items with different names and/or damage values.
  + **Example Data:** Compare Sword with damage 10 to Axe with damage 15.
  + **Expected Outcome:** Returns false, indicating the items are not equal.

**4. hashCode()**

Items should be uniquely identified by their name and damage.

* **Test Case 1: Hash Code for Identical Items**
  + **Condition:** Generates hash codes for two identical items.
  + **Example Data:** Two Sword items with damage 10.
  + **Expected Outcome:** The hash codes for both Sword items should be identical.
* **Test Case 2: Hash Code for Different Items**
  + **Condition:** Generates hash codes for two different items.
  + **Example Data:** Compare hash codes for Sword with damage 10 and Axe with damage 15.
  + **Expected Outcome:** The hash codes should be different.

**Testing Plan for TargetCharacter Class**

**Responsibilities:**  
Represents the target character that moves automatically during each turn in the game.

**Attributes:**

* String name; – The name of the target character.
* int health; – The health of the target character.
* int currentPosition; – The index of the space where the target character is located.

**Methods:**

**1. moveToNextSpace()**

* **Test Case 1: Move to Next Space**
  + **Condition:** Moves the target character to the next space in sequence.
  + **Example Data:** Target character starts at space 3 in a list of 5 spaces.
  + **Expected Outcome:** After calling moveToNextSpace(), the character should be in space 4.
* **Test Case 2: Move to Next Space when at the Last Space**
  + **Condition:** Moves the target character when they are in the last space in the world.
  + **Example Data:** Target character is in space 4 in a world of 5 spaces.
  + **Expected Outcome:** After calling moveToNextSpace(), the character should move to space 0 (wrap around).

**2. getCurrentSpace()**

* **Test Case 1: Retrieve Current Space**
  + **Condition:** Returns the current space of the target character.
  + **Example Data:** Target character is in space 2.
  + **Expected Outcome:** getCurrentSpace() should return 2.
* **Test Case 2: Retrieve Current Space After Moving**
  + **Condition:** Retrieves the current space after calling moveToNextSpace().
  + **Example Data:** Target character starts in space 1 and moves to space 2.
  + **Expected Outcome:** getCurrentSpace() should return 2 after movement.

**3. takeDamage(int damage)**

* **Test Case 1: Reduce Health with Damage**
  + **Condition:** Reduces the health of the target character when they take damage.
  + **Example Data:** Target character has 50 health and takes 10 damage.
  + **Expected Outcome:** After calling takeDamage(10), the character's health should be 40.
* **Test Case 2: Health Drops Below Zero**
  + **Condition:** Ensures that the character's health cannot drop below 0.
  + **Example Data:** Target character has 5 health and takes 10 damage.
  + **Expected Outcome:** The character's health should be set to 0 (no negative values).

**4. getHealth()**

* **Test Case 1: Retrieve Initial Health**
  + **Condition:** Retrieves the initial health of the target character.
  + **Example Data:** The target character is initialized with 100 health.
  + **Expected Outcome:** getHealth() should return 100.
* **Test Case 2: Retrieve Health After Taking Damage**
  + **Condition:** Retrieves the character's health after they take damage.
  + **Example Data:** Character has 50 health and takes 10 damage.
  + **Expected Outcome:** getHealth() should return 40 after damage.

**5. getName()**

* **Test Case 1: Retrieve Name of Target Character**
  + **Condition:** Retrieves the name of the target character.
  + **Example Data:** The target character is named Doctor Lucky.
  + **Expected Outcome:** getName() should return "Doctor Lucky".

**Testing Plan for GameController Class**

**Responsibilities:**  
Manages game interaction between the model (World, Player, Space, Item) and the user. Facilitates commands like adding players, moving players, picking up items, and running the game.

**Attributes:**

* World world; – The current game world.
* List<Player> players; – List of all players in the game (both human and AI).
* int maxTurns; – The maximum number of turns allowed in the game.

**Methods:**

**1. startGame()**

* **Test Case 1: Initialize the Game World**
  + **Condition:** Ensures that the game world is initialized properly when the game starts.
  + **Example Data:** A world with 10 spaces and 3 players.
  + **Expected Outcome:** The game world is properly set up with the correct spaces and players, and startGame() initializes these elements without errors.

**2. addHumanPlayer(String name, ImSpace startSpace)**

* **Test Case 1: Add a Human Player**
  + **Condition:** Adds a new human-controlled player to the game.
  + **Example Data:** Add a human player named "Alice" starting in Space 1.
  + **Expected Outcome:** "Alice" is added to the list of players and is placed in Space 1.
* **Test Case 2: Add Human Player to an Occupied Space**
  + **Condition:** Adds a human player to a space that already has another player.
  + **Example Data:** "Alice" is in Space 1, and "Bob" is added to the same space.
  + **Expected Outcome:** Both players are successfully placed in Space 1 without conflicts.

**3. addComputerPlayer(String name, ImSpace startSpace)**

* **Test Case 1: Add a Computer Player**
  + **Condition:** Adds a computer-controlled player to the game.
  + **Example Data:** Add a computer player named "AI\_Player" starting in Space 2.
  + **Expected Outcome:** "AI\_Player" is added to the list of players and placed in Space 2.
* **Test Case 2: Add Computer Player to a Non-Existing Space**
  + **Condition:** Tries to add a computer player to a space that does not exist.
  + **Example Data:** Attempt to add "AI\_Player" to Space 99 when the world only has 10 spaces.
  + **Expected Outcome:** The method should throw an exception or return an error indicating an invalid space.

**4. movePlayer(Player player, ImSpace destination)**

* **Test Case 1: Move Player to Neighboring Space**
  + **Condition:** Moves a player to a valid neighboring space.
  + **Example Data:** Move player "Alice" from Space 1 to its neighboring Space 2.
  + **Expected Outcome:** "Alice" successfully moves from Space 1 to Space 2.
* **Test Case 2: Move Player to a Non-Neighboring Space**
  + **Condition:** Attempts to move a player to a space that is not a neighbor.
  + **Example Data:** Try to move "Alice" from Space 1 to Space 5 when they are not neighbors.
  + **Expected Outcome:** The method should return an error or prevent the move.
* **Test Case 3: Move Non-Existing Player**
  + **Condition:** Attempts to move a player that is not in the game.
  + **Example Data:** Try to move a player named "Charlie" who does not exist.
  + **Expected Outcome:** The method should return an error indicating the player is not found.

**5. pickUpItem(Player player)**

* **Test Case 1: Player Picks Up Item from Space**
  + **Condition:** Allows a player to pick up an item from the space they are currently in.
  + **Example Data:** Player "Alice" is in Space 1 with an item "Sword" in the space.
  + **Expected Outcome:** "Alice" successfully picks up the "Sword", and it is removed from the space's item list.
* **Test Case 2: Player Attempts to Pick Up Item with Full Inventory**
  + **Condition:** Prevents the player from picking up an item if their inventory is full.
  + **Example Data:** "Alice" has a max item limit of 3 and is already carrying 3 items, with a "Sword" in the space.
  + **Expected Outcome:** The method should prevent "Alice" from picking up the "Sword" and provide feedback that the inventory is full.

**6. lookAround(Player player)**

* **Test Case 1: Player Looks Around a Space**
  + **Condition:** Allows a player to look around and display information about neighboring spaces.
  + **Example Data:** "Alice" is in Space 1, which has Space 2 and Space 3 as neighbors.
  + **Expected Outcome:** The method displays the items and players in Space 1, and information about Space 2 and Space 3.
* **Test Case 2: Player Looks Around in a Space with No Neighbors**
  + **Condition:** Handles the case when a player is in a space without any neighbors.
  + **Example Data:** "Alice" is in Space 10, which has no neighboring spaces.
  + **Expected Outcome:** The method displays that there are no neighbors and shows only the information about Space 10.

**7. displayPlayerInfo(Player player)**

* **Test Case 1: Display Information About a Player**
  + **Condition:** Shows detailed information about a player, including their current space and items.
  + **Example Data:** "Alice" is in Space 1 with items "Sword" and "Shield".
  + **Expected Outcome:** The method displays "Alice"'s location in Space 1 and the items "Sword" and "Shield".
* **Test Case 2: Display Information for a Player with No Items**
  + **Condition:** Shows the information for a player with no items in their possession.
  + **Example Data:** "Bob" is in Space 2 with no items.
  + **Expected Outcome:** The method displays "Bob"'s location in Space 2 and indicates that they have no items.

**8. endTurn()**

* **Test Case 1: End Player's Turn**
  + **Condition:** Ends the current player's turn and switches to the next player.
  + **Example Data:** It's "Alice"'s turn, and after executing a command, the turn ends.
  + **Expected Outcome:** "Alice"'s turn ends, and control moves to the next player in the sequence.
* **Test Case 2: End Turn After Maximum Turns**
  + **Condition:** Ends the game after the maximum number of turns is reached.
  + **Example Data:** The maximum turns allowed are 10, and the game reaches turn 10.
  + **Expected Outcome:** The game should end after the final turn, and no further turns should be allowed.

**9. runTurns()**

* **Test Case 1: Execute Multiple Player Turns**
  + **Condition:** Loops through all players and allows each to take their turn.
  + **Example Data:** Three players, "Alice", "Bob", and "AI\_Player", each take a turn in sequence.
  + **Expected Outcome:** The method executes a turn for "Alice", then "Bob", and finally "AI\_Player", repeating the sequence for subsequent turns.
* **Test Case 2: Run Turns with a Mix of Human and AI Players**
  + **Condition:** Handles a game where there are both human and AI players.
  + **Example Data:** Two players, "Alice" (human) and "AI\_Player" (AI), take turns in sequence.
  + **Expected Outcome:** "Alice" is prompted for input, while "AI\_Player" takes an automated action during their turn.