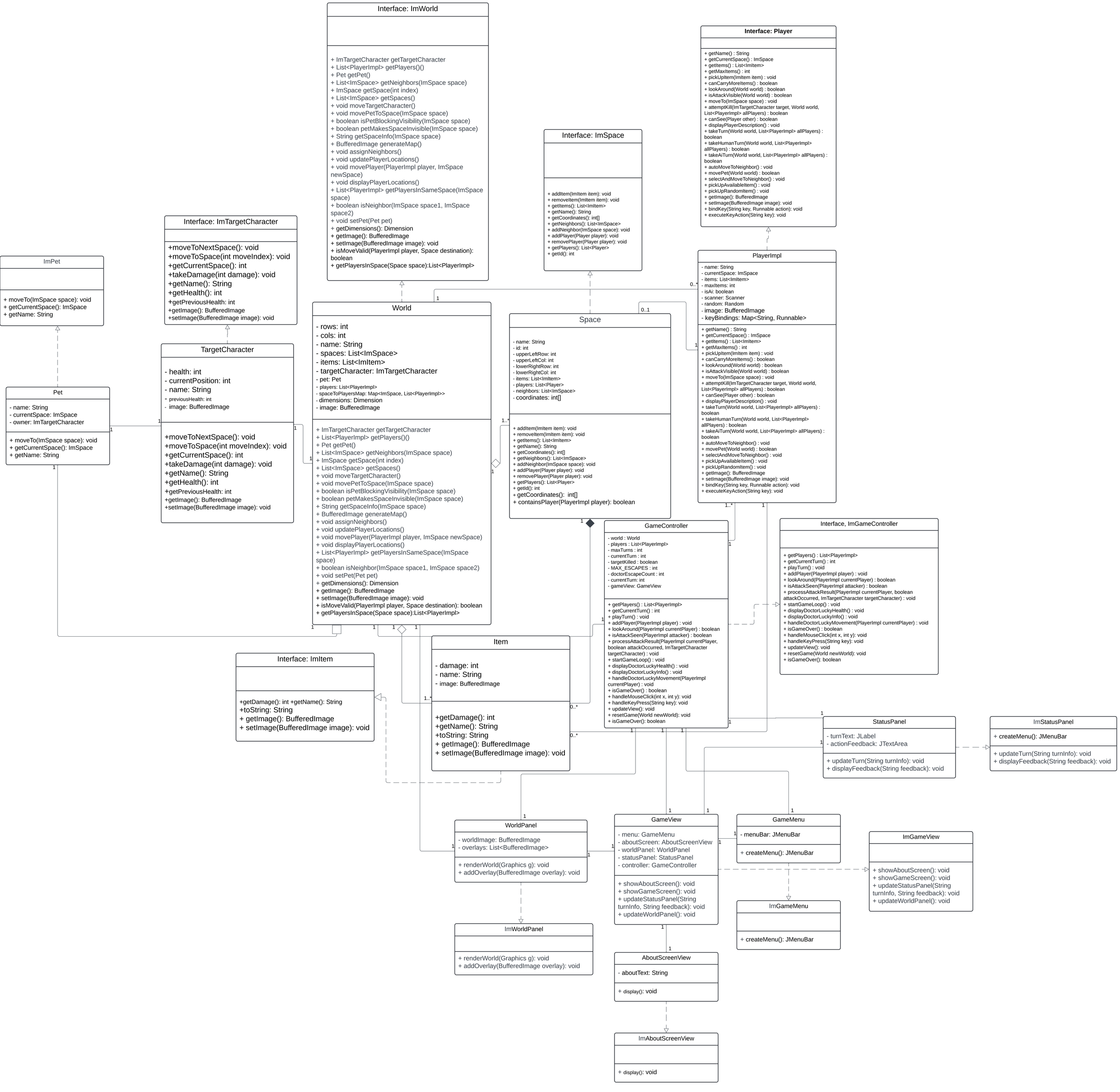
Interface: ImWorld
+ ImTargetCharacter getTargetCharacter
+ List<PlayerImpl> getPlayers()()
+ Pet getPet()
+ List<ImSpace> getNeighbors(ImSpace space)
+ ImSpace getSpace(int index)
+ List<ImSpace> getSpaces()
+ void moveTargetCharacter()
+ void movePetToSpace(ImSpace space)
+ boolean isPetBlockingVisibility(ImSpace space)
+ boolean petMakesSpaceInvisible(ImSpace space)
+ String getSpaceInfo(ImSpace space)
+ BufferedImage generateMap()
+ void assignNeighbors()
+ void updatePlayerLocations()
+ void movePlayer(PlayerImpl player, ImSpace newSpace)
+ void displayPlayerLocations()
+ List<PlayerImpl> getPlayersInSameSpace(ImSpace space)
+ boolean isNeighbor(ImSpace space1, ImSpace space2)
+ void setPet(Pet pet)
+ getDimensions(): Dimension
+ getImage(): BufferedImage
+ setImage(BufferedImage image): void
+ isMoveValid(PlayerImpl player, Space destination): boolean
+ getPlayersInSpace(Space space):List<PlayerImpl>
Interface: ImSpace
+ addItem(ImItem item): void
+ removeItem(ImItem item): void
+ getItems(): List<ImItem>
+ getName(): String
+ getCoordinates(): int[]
+ getNeighbors(): List<ImSpace>
+ addNeighbor(ImSpace space): void
+ addPlayer(Player player): void
+ removePlayer(Player player): void
+ getPlayers(): List<Player>
+ getId(): int
Interface: Player
+ getName() : String
+ getCurrentSpace() : ImSpace
+ getItems() : List<ImItem>
+ getMaxItems() : int
+ pickUpItem(ImItem item) : void
+ canCarryMoreItems() : boolean
+ lookAround(World world) : boolean
+ isAttackVisible(World world) : boolean
+ moveTo(ImSpace space) : void
+ attemptKill(ImTargetCharacter target, World world, List<PlayerImpl> allPlayers) : boolean
+ canSee(Player other) : boolean
+ displayPlayerDescription() : void
+ takeTurn(World world, List<PlayerImpl> allPlayers) : boolean
+ takeHumanTurn(World world, List<PlayerImpl> allPlayers) : boolean
+ takeAiTurn(World world, List<PlayerImpl> allPlayers) : boolean
+ autoMoveToNeighbor() : void
+ movePet(World world) : boolean
+ selectAndMoveToNeighbor() : void
+ pickUpAvailableItem() : void
+ pickUpRandomItem() : void
+ getImage(): BufferedImage
+ setImage(BufferedImage image): void
+ bindKey(String key, Runnable action): void
+ executeKeyAction(String key): void
Interface: ImTargetCharacter
+moveToNextSpace(): void
+moveToSpace(int moveIndex): void
+getCurrentSpace(): int
+takeDamage(int damage): void
+getName(): String
+getHealth(): int
+getPreviousHealth: int
+getImage(): BufferedImage
+setImage(BufferedImage image): void
ImPet
+ moveTo(ImSpace space): void
+ getCurrentSpace(): ImSpace
+ getName: String
World
- rows: int
- cols: int
- name: String
- spaces: List<ImSpace>
- items: List<ImItem>
- targetCharacter: ImTargetCharacter
- pet: Pet
- players: List<PlayerImpl>
- spaceToPlayersMap: Map<ImSpace, List<PlayerImpl>>
- dimensions: Dimension
- image: BufferedImage
+ ImTargetCharacter getTargetCharacter
+ List<PlayerImpl> getPlayers()()
+ Pet getPet()
+ List<ImSpace> getNeighbors(ImSpace space)
+ ImSpace getSpace(int index)
+ List<ImSpace> getSpaces()
+ void moveTargetCharacter()
+ void movePetToSpace(ImSpace space)
+ boolean isPetBlockingVisibility(ImSpace space)
+ boolean petMakesSpaceInvisible(ImSpace space)
+ String getSpaceInfo(ImSpace space)
+ BufferedImage generateMap()
+ void assignNeighbors()
+ void updatePlayerLocations()
+ void movePlayer(PlayerImpl player, ImSpace newSpace)
+ void displayPlayerLocations()
+ List<PlayerImpl> getPlayersInSameSpace(ImSpace space)
+ boolean isNeighbor(ImSpace space1, ImSpace space2)
+ void setPet(Pet pet)
+ getDimensions(): Dimension
+ getImage(): BufferedImage
+ setImage(BufferedImage image): void
+ isMoveValid(PlayerImpl player, Space destination): boolean
+ getPlayersInSpace(Space space):List<PlayerImpl>
Space
- name: String
- id: int
- upperLeftRow: int
- upperLeftCol: int
- lowerRightRow: int
- lowerRightCol: int
- items: List<ImItem>
- players: List<Player>
- neighbors: List<ImSpace>
- coordinates: int[]
+ addItem(ImItem item): void
+ removeItem(ImItem item): void
+ getItems(): List<ImItem>
+ getName(): String
+ getCoordinates(): int[]
+ getNeighbors(): List<ImSpace>
+ addNeighbor(ImSpace space): void
+ addPlayer(Player player): void
+ removePlayer(Player player): void
+ getPlayers(): List<Player>
+ getId(): int
+ getCoordinates(): int[]
+ containsPlayer(PlayerImpl player): boolean
PlayerImpl
- name: String
- currentSpace: ImSpace
- items: List<ImItem>
- maxItems: int
- isAi: boolean
- scanner: Scanner
- random: Random
- image: BufferedImage
- keyBindings: Map<String, Runnable>
+ getName() : String
+ getCurrentSpace() : ImSpace
+ getItems() : List<ImItem>
+ getMaxItems() : int
+ pickUpItem(ImItem item) : void
+ canCarryMoreItems() : boolean
+ lookAround(World world) : boolean
+ isAttackVisible(World world) : boolean
+ moveTo(ImSpace space) : void
+ attemptKill(ImTargetCharacter target, World world, List<PlayerImpl> allPlayers) : boolean
+ canSee(Player other) : boolean
+ displayPlayerDescription() : void
+ takeTurn(World world, List<PlayerImpl> allPlayers) : boolean
+ takeHumanTurn(World world, List<PlayerImpl> allPlayers) : boolean
+ takeAiTurn(World world, List<PlayerImpl> allPlayers) : boolean
+ autoMoveToNeighbor() : void
+ movePet(World world) : boolean
+ selectAndMoveToNeighbor() : void
+ pickUpAvailableItem() : void
+ pickUpRandomItem() : void
+ getImage(): BufferedImage
+ setImage(BufferedImage image): void
+ bindKey(String key, Runnable action): void
+ executeKeyAction(String key): void
TargetCharacter
- health: int
- currentPosition: int
- name: String
- previousHealth: int
- image: BufferedImage
+moveToNextSpace(): void
+moveToSpace(int moveIndex): void
+getCurrentSpace(): int
+takeDamage(int damage): void
+getName(): String
+getHealth(): int
+getPreviousHealth: int
+getImage(): BufferedImage
+setImage(BufferedImage image): void
Pet
- name: String
- currentSpace: ImSpace
- owner: ImTargetCharacter
+ moveTo(ImSpace space): void
+ getCurrentSpace(): ImSpace
+ getName: String
GameController
- world : World
- players : List<PlayerImpl>
- maxTurns : int
- currentTurn : int
- targetKilled : boolean
- MAX_ESCAPES : int
- doctorEscapeCount : int
- currentTurn: int
- gameView: GameView
+ getPlayers() : List<PlayerImpl>
+ getCurrentTurn() : int
+ playTurn() : void
+ addPlayer(PlayerImpl player) : void
+ lookAround(PlayerImpl currentPlayer) : boolean
+ isAttackSeen(PlayerImpl attacker) : boolean
+ processAttackResult(PlayerImpl currentPlayer, boolean attackOccurred, ImTargetCharacter targetCharacter) : void
+ startGameLoop() : void
+ displayDoctorLuckyHealth() : void
+ displayDoctorLuckyInfo() : void
+ handleDoctorLuckyMovement(PlayerImpl currentPlayer) : void
+ isGameOver() : boolean
+ handleMouseClick(int x, int y): void
+ handleKeyPress(String key): void
+ updateView(): void
+ resetGame(World newWorld): void
+ isGameOver(): boolean
Interface, ImGameController
+ getPlayers() : List<PlayerImpl>
+ getCurrentTurn() : int
+ playTurn() : void
+ addPlayer(PlayerImpl player) : void
+ lookAround(PlayerImpl currentPlayer) : boolean
+ isAttackSeen(PlayerImpl attacker) : boolean
+ processAttackResult(PlayerImpl currentPlayer, boolean attackOccurred, ImTargetCharacter targetCharacter) : void
+ startGameLoop() : void
+ displayDoctorLuckyHealth() : void
+ displayDoctorLuckyInfo() : void
+ handleDoctorLuckyMovement(PlayerImpl currentPlayer) : void
+ isGameOver() : boolean
+ handleMouseClick(int x, int y): void
+ handleKeyPress(String key): void
+ updateView(): void
+ resetGame(World newWorld): void
+ isGameOver(): boolean
Item
- damage: int
- name: String
- image: BufferedImage
+getDamage(): int
+getName(): String
+toString: String
+ getImage(): BufferedImage
+ setImage(BufferedImage image): void
Interface: ImItem
+getDamage(): int +getName(): String
+toString: String
+ getImage(): BufferedImage
+ setImage(BufferedImage image): void
StatusPanel
- turnText: JLabel
- actionFeedback: JTextArea
+ updateTurn(String turnInfo): void
+ displayFeedback(String feedback): void
ImStatusPanel
+ createMenu(): JMenuBar
+ updateTurn(String turnInfo): void
+ displayFeedback(String feedback): void
WorldPanel
- worldImage: BufferedImage
- overlays: List<BufferedImage>
+ renderWorld(Graphics g): void
+ addOverlay(BufferedImage overlay): void
ImWorldPanel
+ renderWorld(Graphics g): void
+ addOverlay(BufferedImage overlay): void
GameView
- menu: GameMenu
- aboutScreen: AboutScreenView
- worldPanel: WorldPanel
- statusPanel: StatusPanel
- controller: GameController
+ showAboutScreen(): void
+ showGameScreen(): void
+ updateStatusPanel(String turnInfo, String feedback): void
+ updateWorldPanel(): void
GameMenu
- menuBar: JMenuBar
+ createMenu(): JMenuBar
ImGameView
+ showAboutScreen(): void
+ showGameScreen(): void
+ updateStatusPanel(String turnInfo, String feedback): void
+ updateWorldPanel(): void
ImGameMenu
+ createMenu(): JMenuBar
AboutScreenView
- aboutText: String
+ display(): void
ImAboutScreenView
+ display(): void

Testing Plan for GameView Class

Responsibilities:

Manages all GUI components for the game, including GameMenu, AboutScreenView, WorldPanel, and StatusPanel.

Methods:

1. **showAboutScreen()**
    - **Test Case 1**: Displays the About Screen
        - **Condition**: Verifies that the about screen is displayed when called.
        - **Example Data**: gameView.showAboutScreen().
        - **Expected Outcome**: The About Screen is displayed, and other components are hidden.
2. **showGameScreen()**
    - **Test Case 1**: Switches to Game Screen
        - **Condition**: Verifies that the main game screen is displayed after the game starts.
        - **Example Data**: gameView.showGameScreen().
        - **Expected Outcome**: The WorldPanel and StatusPanel are visible.
3. **updateStatusPanel(String turnInfo, String feedback)**
    - **Test Case 1**: Update Turn Info
        - **Condition**: Verifies that the turn information is displayed correctly.
        - **Example Data**: turnInfo = "Player 1's turn", feedback = "Moved to Kitchen".
        - **Expected Outcome**: The StatusPanel displays "Player 1's turn" and "Moved to Kitchen".
    - **Test Case 2**: Update with No Feedback
        - **Condition**: Verifies behavior when feedback is empty.
        - **Example Data**: turnInfo = "Player 2's turn", feedback = "".
        - **Expected Outcome**: Only the turn info is displayed.
4. **updateWorldPanel()**
    - **Test Case 1**: Refresh World Map
        - **Condition**: Verifies that the map is refreshed to reflect the current state.
        - **Example Data**: Player moves, and gameView.updateWorldPanel() is called.
        - **Expected Outcome**: The new positions of players are displayed on the map.

Testing Plan for GameMenu Class

Responsibilities:

Provides game menu options for starting a new game, loading an existing game, or quitting.

Methods:

1. **createMenu()**
    - **Test Case 1**: Menu Initialization
        - **Condition**: Verifies that the menu is created with the correct

options.
        - **Example Data**: Call gameMenu.createMenu().
        - **Expected Outcome**: Menu contains options: "New Game", "Load Game", and "Quit".
2. **Action Handling**
    - **Test Case 1**: Start New Game
        - **Condition**: Verifies behavior when "New Game" is selected.
        - **Example Data**: Click "New Game".
        - **Expected Outcome**: The game initializes with a new world specification.
    - **Test Case 2**: Quit Game
        - **Condition**: Verifies behavior when "Quit" is selected.
        - **Example Data**: Click "Quit".
        - **Expected Outcome**: The application exits.

Testing Plan for AboutScreenView Class

Responsibilities:

Displays the welcome screen with game information and credits.

Methods:

1. **display()**
    - **Test Case 1**: Welcome Message Display
        - **Condition**: Verifies that the welcome message is displayed.
        - **Example Data**: aboutScreenView.display().
        - **Expected Outcome**: The welcome message and credits are visible.
    - **Test Case 2**: Hides Other Components
        - **Condition**: Ensures other UI components are hidden when the about screen is displayed.
        - **Example Data**: Call aboutScreenView.display().
        - **Expected Outcome**: Only the About Screen is visible.

Testing Plan for WorldPanel Class

Responsibilities:

Displays the graphical map of the world and overlays for players and the target character.

Methods:

1. **renderWorld(Graphics g)**
    - **Test Case 1**: Render Initial World Map
        - **Condition**: Verifies that the world map is rendered correctly.
        - **Example Data**: A 5x5 grid world with no players.
        - **Expected Outcome**: The grid is displayed without overlays.
    - **Test Case 2**: Render with Players and Target Character
        - **Condition**: Verifies that players and the target character are overlaid on the map.
        - **Example Data**: Player at (1,1), target character at (2,2).
        - **Expected Outcome**: Players and target character are shown at the

correct positions.

2. **addOverlay(BufferedImage overlay)**
    - **Test Case 1**: Add Single Overlay
        - **Condition**: Verifies that an overlay is added.
        - **Example Data**: Add an overlay for Player 1.
        - **Expected Outcome**: Overlay appears on the map.
    - **Test Case 2**: Add Multiple Overlays
        - **Condition**: Verifies that multiple overlays can be added.
        - **Example Data**: Add overlays for two players.
        - **Expected Outcome**: Both overlays are visible on the map.

Testing Plan for StatusPanel Class

Responsibilities:

Displays the current player's turn and action feedback.

Methods:

1. **updateTurn(String turnInfo)**
    - **Test Case 1**: Display Turn Info
        - **Condition**: Verifies that the correct turn information is displayed.
        - **Example Data**: turnInfo = "Player 1's Turn".
        - **Expected Outcome**: Displays "Player 1's Turn".
2. **displayFeedback(String feedback)**
    - **Test Case 1**: Display Action Feedback
        - **Condition**: Verifies that action feedback is displayed.
        - **Example Data**: feedback = "Picked up a Sword".
        - **Expected Outcome**: Displays "Picked up a Sword".
    - **Test Case 2**: Handle Empty Feedback
        - **Condition**: Ensures no error occurs when feedback is empty.
        - **Example Data**: feedback = "".
        - **Expected Outcome**: Displays no feedback but remains operational.

Testing Plan for Pet Class

Responsibilities:

Represents the pet in the game that moves through spaces and is owned by the target character.

Attributes:

1. name: String – The name of the pet.
2. currentSpace: ImSpace – The space where the pet is currently located.
3. owner: ImTargetCharacter – The target character that owns the pet.

Methods:

---

1. moveTo(ImSpace space): void

Description: Moves the pet to a specified space.

- **Test Case 1**: Move to a valid space
    - **Condition**: Move the pet to a valid, existing space.

    - **Example Data**: Current space = Living Room; Destination space = Kitchen.
    - **Expected Outcome**: The pet is moved to the Kitchen.
- **Test Case 2**: Move to a null space
    - **Condition**: Attempt to move the pet to a null space.
    - **Example Data**: Destination space = null.
    - **Expected Outcome**: Throws a NullPointerException.
- **Test Case 3**: Move to the same space
    - **Condition**: Attempt to move the pet to its current space.
    - **Example Data**: Current space = Living Room; Destination space = Living Room.
    - **Expected Outcome**: The method does nothing, and the pet remains in the Living Room.

---

2. getCurrentSpace(): ImSpace

Description: Returns the current space where the pet is located.

- **Test Case 1**: Pet in a valid space
    - **Condition**: The pet is in a valid space.
    - **Example Data**: Pet is in Kitchen.
    - **Expected Outcome**: Returns the Kitchen space object.
- **Test Case 2**: Pet in no space
    - **Condition**: The pet has not been placed in any space.
    - **Example Data**: currentSpace = null.
    - **Expected Outcome**: Returns null.

---

3. getName(): String

Description: Returns the name of the pet.

- **Test Case 1**: Pet with a valid name
    - **Condition**: The pet has been assigned a valid name.
    - **Example Data**: Pet name = "Lucky".
    - **Expected Outcome**: Returns "Lucky".
- **Test Case 2**: Pet with an empty name
    - **Condition**: The pet's name is an empty string.
    - **Example Data**: Pet name = "".
    - **Expected Outcome**: Returns an empty string "".
- **Test Case 3**: Pet with a null name
    - **Condition**: The pet's name is null.
    - **Example Data**: Pet name = null.
    - **Expected Outcome**: Throws a NullPointerException or returns a default name.

Testing Plan for TargetCharacter Class

Responsibilities:

Represents the target character in the game, including its movement, health management, and graphical representation.

Attributes:

1. health: int – The current health of the target character.
2. currentPosition: int – The index of the space where the target character is currently located.
3. name: String – The name of the target character.
4. previousHealth: int – The character's health before the most recent damage.
5. image: BufferedImage – The graphical representation of the target character.

Methods:

---

1. moveToNextSpace(): void

Description: Moves the target character to the next space in the sequence.

- **Test Case 1**: Move to the next valid space
  - **Condition**: The target character moves sequentially to the next space.
  - **Example Data**: Current position = 2; Number of spaces = 5.
  - **Expected Outcome**: The target character moves to position 3.
- **Test Case 2**: Wrap around to the first space
  - **Condition**: The target character moves from the last space to the first space.
  - **Example Data**: Current position = 4; Number of spaces = 5.
  - **Expected Outcome**: The target character moves to position 0.

---

2. moveToSpace(int moveIndex): void

Description: Moves the target character to a specified space.

- **Test Case 1**: Move to a valid space
  - **Condition**: Move the character to a specific, valid space index.
  - **Example Data**: moveIndex = 3; Number of spaces = 5.
  - **Expected Outcome**: The target character moves to position 3.
- **Test Case 2**: Invalid index (negative)
  - **Condition**: Attempt to move to a negative index.
  - **Example Data**: moveIndex = -1.
  - **Expected Outcome**: Throws an IndexOutOfBoundsException.
- **Test Case 3**: Invalid index (out of range)
  - **Condition**: Attempt to move to a space index greater than the total number of spaces.
  - **Example Data**: moveIndex = 10; Number of spaces = 5.
  - **Expected Outcome**: Throws an IndexOutOfBoundsException.

---

3. getCurrentSpace(): int

Description: Returns the index of the space where the target character is currently located.

- **Test Case 1**: Target in a valid space
  - **Condition**: The target is in a valid space.
  - **Example Data**: Current position = 2.
  - **Expected Outcome**: Returns 2.
- **Test Case 2**: Target not yet placed
  - **Condition**: The target has not been assigned a space.

- **Example Data**: currentPosition = -1.
- **Expected Outcome**: Returns -1.

---

4. takeDamage(int damage): void

Description: Reduces the health of the target character by the specified damage amount.

- **Test Case 1**: Reduce health with valid damage
  - **Condition**: Health is reduced by a positive damage value.
  - **Example Data**: health = 50; damage = 10.
  - **Expected Outcome**: Health becomes 40.
- **Test Case 2**: Damage greater than current health
  - **Condition**: The damage exceeds the target's current health.
  - **Example Data**: health = 10; damage = 15.
  - **Expected Outcome**: Health becomes 0 (no negative values).
- **Test Case 3**: Zero damage
  - **Condition**: No change occurs when damage is 0.
  - **Example Data**: health = 50; damage = 0.
  - **Expected Outcome**: Health remains 50.

---

5. getName(): String

Description: Returns the name of the target character.

- **Test Case 1**: Valid name
  - **Condition**: The target character has a valid name.
  - **Example Data**: name = "Doctor Lucky".
  - **Expected Outcome**: Returns "Doctor Lucky".
- **Test Case 2**: Empty name
  - **Condition**: The target character's name is an empty string.
  - **Example Data**: name = "".
  - **Expected Outcome**: Returns "".

---

6. getHealth(): int

Description: Returns the current health of the target character.

- **Test Case 1**: Retrieve valid health
  - **Condition**: The target character has a positive health value.
  - **Example Data**: health = 50.
  - **Expected Outcome**: Returns 50.
- **Test Case 2**: Health after taking damage
  - **Condition**: Retrieve health after applying damage.
  - **Example Data**: health = 50; damage = 10.
  - **Expected Outcome**: Returns 40.

---

7. getPreviousHealth(): int

Description: Returns the previous health of the target character before the most recent damage.

- **Test Case 1**: Health changes after taking damage

    - **Condition**: Verify that previous health reflects the value before damage was applied.
    - **Example Data**: previousHealth = 50; currentHealth = 40.
    - **Expected Outcome**: Returns 50.
- **Test Case 2**: No health changes
    - **Condition**: Verify that previous health equals current health if no damage was applied.
    - **Example Data**: previousHealth = 50; currentHealth = 50.
    - **Expected Outcome**: Returns 50.

---

8. getImage(): BufferedImage

Description: Returns the graphical representation of the target character.

- **Test Case 1**: Valid image
    - **Condition**: An image has been assigned to the target character.
    - **Example Data**: image = BufferedImage (valid instance).
    - **Expected Outcome**: Returns the image instance.
- **Test Case 2**: No image assigned
    - **Condition**: No image has been set for the target character.
    - **Example Data**: image = null.
    - **Expected Outcome**: Returns null.

---

9. setImage(BufferedImage image): void

Description: Assigns a graphical image to the target character.

- **Test Case 1**: Assign a valid image
    - **Condition**: Set a valid image to the target character.
    - **Example Data**: Assign a BufferedImage instance.
    - **Expected Outcome**: Image is successfully assigned.
- **Test Case 2**: Assign a null image
    - **Condition**: Attempt to set a null image.
    - **Example Data**: image = null.
    - **Expected Outcome**: Throws a NullPointerException.

Testing Plan for World Class

Responsibilities:

Manages the game world, including spaces, players, items, the pet, and the target character. Handles interactions, movements, visibility, and graphical representation.

Attributes:

1. rows: int – Number of rows in the world grid.
2. cols: int – Number of columns in the world grid.
3. name: String – The name of the world.
4. spaces: List<ImSpace> – The list of spaces in the world.
5. items: List<ImItem> – The list of items in the world.
6. targetCharacter: ImTargetCharacter – The target character in the game.

7. pet: Pet – The pet in the game.
8. players: List<PlayerImpl> – The list of players in the game.
9. spaceToPlayersMap: Map<ImSpace, List<PlayerImpl>> – Mapping of spaces to players in those spaces.
10. dimensions: Dimension – Dimensions of the world for rendering.
11. image: BufferedImage – Graphical representation of the world.

Methods:

---

1. getTargetCharacter(): ImTargetCharacter

Description: Returns the target character in the game.

- **Test Case 1**: Retrieve valid target character
  - **Condition**: The target character is initialized.
  - **Example Data**: targetCharacter = Doctor Lucky.
  - **Expected Outcome**: Returns Doctor Lucky.
- **Test Case 2**: No target character
  - **Condition**: The target character is not set.
  - **Example Data**: targetCharacter = null.
  - **Expected Outcome**: Returns null.

---

2. getPlayers(): List<PlayerImpl>

Description: Returns the list of players in the game.

- **Test Case 1**: Retrieve multiple players
  - **Condition**: The game has multiple players.
  - **Example Data**: Players = Alice, Bob, Charlie.
  - **Expected Outcome**: Returns a list containing Alice, Bob, and Charlie.
- **Test Case 2**: No players
  - **Condition**: No players are added to the game.
  - **Example Data**: players = [].
  - **Expected Outcome**: Returns an empty list.

---

3. getPet(): Pet

Description: Returns the pet in the game.

- **Test Case 1**: Retrieve valid pet
  - **Condition**: The pet is initialized.
  - **Example Data**: pet = Lucky.
  - **Expected Outcome**: Returns Lucky.
- **Test Case 2**: No pet
  - **Condition**: The pet is not set.
  - **Example Data**: pet = null.
  - **Expected Outcome**: Returns null.

---

4. getNeighbors(ImSpace space): List<ImSpace>

Description: Returns the list of neighboring spaces for a given space.

- **Test Case 1**: Valid neighbors

    - **Condition**: The space has valid neighbors.
    - **Example Data**: Space A neighbors = Space B, Space C.
    - **Expected Outcome**: Returns Space B and Space C.
- **Test Case 2**: No neighbors
    - **Condition**: The space is isolated.
    - **Example Data**: Space Z neighbors = [].
    - **Expected Outcome**: Returns an empty list.

---

5. getSpace(int index): ImSpace

Description: Returns the space at the specified index.

- **Test Case 1**: Valid index
    - **Condition**: The index is within range.
    - **Example Data**: index = 3; spaces = [A, B, C, D].
    - **Expected Outcome**: Returns Space D.
- **Test Case 2**: Invalid index (negative)
    - **Condition**: The index is negative.
    - **Example Data**: index = -1.
    - **Expected Outcome**: Throws an IndexOutOfBoundsException.
- **Test Case 3**: Invalid index (out of range)
    - **Condition**: The index exceeds the list size.
    - **Example Data**: index = 5; spaces.size() = 4.
    - **Expected Outcome**: Throws an IndexOutOfBoundsException.

---

6. moveTargetCharacter(): void

Description: Moves the target character to the next space.

- **Test Case 1**: Valid move
    - **Condition**: Moves the target character to the next space.
    - **Example Data**: Current space = Space 2; Total spaces = 5.
    - **Expected Outcome**: Target character moves to Space 3.
- **Test Case 2**: Wrap around
    - **Condition**: Moves the target character from the last space to the first.
    - **Example Data**: Current space = Space 4; Total spaces = 5.
    - **Expected Outcome**: Target character moves to Space 0.

---

7. movePetToSpace(ImSpace space): void

Description: Moves the pet to a specified space.

- **Test Case 1**: Valid move
    - **Condition**: The pet is moved to a valid space.
    - **Example Data**: Pet moves to Kitchen.
    - **Expected Outcome**: Pet is now in the Kitchen.
- **Test Case 2**: Null space
    - **Condition**: Attempt to move the pet to a null space.
    - **Example Data**: space = null.
    - **Expected Outcome**: Throws a NullPointerException.

---

8. isPetBlockingVisibility(ImSpace space): boolean

Description: Checks if the pet blocks visibility for a specific space.

- **Test Case 1**: Pet blocks visibility
  - **Condition**: The pet is in the specified space and blocks visibility.
  - **Example Data**: Pet is in Space A; check visibility for Space A.
  - **Expected Outcome**: Returns true.
- **Test Case 2**: Pet does not block visibility
  - **Condition**: The pet is in a different space.
  - **Example Data**: Pet is in Space B; check visibility for Space A.
  - **Expected Outcome**: Returns false.

---

9. generateMap(): BufferedImage

Description: Generates a graphical map of the world.

- **Test Case 1**: Valid map generation
  - **Condition**: The world has valid dimensions and spaces.
  - **Example Data**: 5x5 grid.
  - **Expected Outcome**: Returns a BufferedImage representing the map.
- **Test Case 2**: Empty world
  - **Condition**: The world has no spaces.
  - **Example Data**: rows = 0, cols = 0.
  - **Expected Outcome**: Returns an empty BufferedImage.

Testing Plan for Space Class

Responsibilities:

Represents a specific area or room in the game. Tracks items, players, neighbors, and coordinates.

Attributes:

1. name: String – The name of the space.
2. id: int – A unique identifier for the space.
3. upperLeftRow, upperLeftCol, lowerRightRow, lowerRightCol: int – Defines the boundaries of the space.
4. items: List<ImItem> – A list of items present in the space.
5. players: List<Player> – A list of players currently in the space.
6. neighbors: List<ImSpace> – Adjacent spaces connected to this space.
7. coordinates: int[] – Graphical coordinates for rendering.

Methods:

---

1. addItem(ImItem item): void

Description: Adds an item to the space.

- **Test Case 1**: Add a single item
  - **Condition**: Verify that the item is added to the space.
  - **Example Data**: Add a Sword to the Living Room.
  - **Expected Outcome**: The Sword is in the list of items for the Living Room.
- **Test Case 2**: Add a null item

    - **Condition**: Attempt to add a null item.
    - **Example Data**: item = null.
    - **Expected Outcome**: Throws a NullPointerException.
- **Test Case 3**: Add duplicate items
    - **Condition**: Add the same item multiple times.
    - **Example Data**: Add Sword twice to the Kitchen.
    - **Expected Outcome**: Sword appears twice in the list of items.

---

2. removeItem(ImItem item): void

Description: Removes an item from the space.

- **Test Case 1**: Remove an existing item
    - **Condition**: Remove an item that is present in the space.
    - **Example Data**: Space contains a Shield; remove the Shield.
    - **Expected Outcome**: The Shield is removed from the list of items.
- **Test Case 2**: Remove a non-existent item
    - **Condition**: Attempt to remove an item not in the space.
    - **Example Data**: Remove a Torch that is not in the Living Room.
    - **Expected Outcome**: No change to the list of items.
- **Test Case 3**: Remove a null item
    - **Condition**: Attempt to remove a null item.
    - **Example Data**: item = null.
    - **Expected Outcome**: Throws a NullPointerException.

---

3. getItems(): List<ImItem>

Description: Returns the list of items in the space.

- **Test Case 1**: Space with items
    - **Condition**: The space contains items.
    - **Example Data**: Items = Key, Map.
    - **Expected Outcome**: Returns a list containing Key and Map.
- **Test Case 2**: Empty space
    - **Condition**: The space has no items.
    - **Example Data**: Items = [].
    - **Expected Outcome**: Returns an empty list.

---

4. getName(): String

Description: Returns the name of the space.

- **Test Case 1**: Valid name
    - **Condition**: The space has a valid name.
    - **Example Data**: name = "Kitchen".
    - **Expected Outcome**: Returns "Kitchen".
- **Test Case 2**: Empty name
    - **Condition**: The space name is an empty string.
    - **Example Data**: name = "".
    - **Expected Outcome**: Returns an empty string.

---

5. getCoordinates(): int[]

Description: Returns the graphical coordinates of the space.

- **Test Case 1**: Valid coordinates
  - **Condition**: The space has valid coordinates.
  - **Example Data**: Coordinates = [0, 0, 5, 5].
  - **Expected Outcome**: Returns [0, 0, 5, 5].
- **Test Case 2**: No coordinates set
  - **Condition**: The coordinates are not initialized.
  - **Example Data**: coordinates = null.
  - **Expected Outcome**: Returns null.

---

6. getNeighbors(): List<ImSpace>

Description: Returns the list of neighboring spaces.

- **Test Case 1**: Space with neighbors
  - **Condition**: The space has valid neighbors.
  - **Example Data**: Neighbors = Space A, Space B.
  - **Expected Outcome**: Returns a list containing Space A and Space B.
- **Test Case 2**: No neighbors
  - **Condition**: The space has no neighbors.
  - **Example Data**: Neighbors = [].
  - **Expected Outcome**: Returns an empty list.

---

7. addNeighbor(ImSpace space): void

Description: Adds a neighboring space.

- **Test Case 1**: Add valid neighbor
  - **Condition**: Add a valid space as a neighbor.
  - **Example Data**: Add Kitchen as a neighbor to Living Room.
  - **Expected Outcome**: Kitchen is added to the neighbors of the Living Room.
- **Test Case 2**: Add null neighbor
  - **Condition**: Attempt to add a null space as a neighbor.
  - **Example Data**: space = null.
  - **Expected Outcome**: Throws a NullPointerException.

---

8. addPlayer(Player player): void

Description: Adds a player to the space.

- **Test Case 1**: Add valid player
  - **Condition**: Add a player to the space.
  - **Example Data**: Add Alice to Kitchen.
  - **Expected Outcome**: Alice is added to the list of players in Kitchen.
- **Test Case 2**: Add null player
  - **Condition**: Attempt to add a null player.
  - **Example Data**: player = null.
  - **Expected Outcome**: Throws a NullPointerException.

---

9. removePlayer(Player player): void

Description: Removes a player from the space.

- **Test Case 1**: Remove existing player
  - **Condition**: Remove a player who is in the space.
  - **Example Data**: Remove Alice from Living Room.
  - **Expected Outcome**: Alice is removed from the list of players.
- **Test Case 2**: Remove non-existent player
  - **Condition**: Attempt to remove a player not in the space.
  - **Example Data**: Remove Bob from Kitchen where Bob is not present.
  - **Expected Outcome**: No change to the list of players.

---

10. getPlayers(): List<Player>

Description: Returns the list of players in the space.

- **Test Case 1**: Space with players
  - **Condition**: The space has multiple players.
  - **Example Data**: Players = Alice, Bob.
  - **Expected Outcome**: Returns a list containing Alice and Bob.
- **Test Case 2**: Empty space
  - **Condition**: The space has no players.
  - **Example Data**: Players = [].
  - **Expected Outcome**: Returns an empty list.

---

11. containsPlayer(PlayerImpl player): boolean

Description: Checks if a specific player is in the space.

- **Test Case 1**: Player present
  - **Condition**: The specified player is in the space.
  - **Example Data**: Alice is in Kitchen.
  - **Expected Outcome**: Returns true.
- **Test Case 2**: Player not present
  - **Condition**: The specified player is not in the space.
  - **Example Data**: Bob is not in Kitchen.
  - **Expected Outcome**: Returns false.

Testing Plan for PlayerImpl Class

Responsibilities:

Represents a player (human or AI) in the game, responsible for movement, item management, interactions, and taking turns.

Attributes:

1. name: String – Name of the player.
2. currentSpace: ImSpace – The space where the player is currently located.
3. items: List<ImItem> – Items the player has collected.
4. maxItems: int – Maximum number of items the player can carry.
5. isAi: boolean – Whether the player is AI-controlled.
6. scanner: Scanner – For user input (human players).

7. random: Random – For decision-making (AI players).
8. image: BufferedImage – Graphical representation of the player.
9. keyBindings: Map<String, Runnable> – Key-action bindings for user input.

---

1. getName(): String

Description: Returns the player's name.

- **Test Case 1**: Valid name
  - **Condition**: Player has a valid name.
  - **Example Data**: name = "Alice".
  - **Expected Outcome**: Returns "Alice".
- **Test Case 2**: Empty name
  - **Condition**: Player's name is an empty string.
  - **Example Data**: name = "".
  - **Expected Outcome**: Returns an empty string.

---

2. getCurrentSpace(): ImSpace

Description: Returns the space where the player is currently located.

- **Test Case 1**: Valid space
  - **Condition**: Player is in a valid space.
  - **Example Data**: Current space = Kitchen.
  - **Expected Outcome**: Returns the Kitchen space object.
- **Test Case 2**: No space assigned
  - **Condition**: Player has not been placed in any space.
  - **Example Data**: currentSpace = null.
  - **Expected Outcome**: Returns null.

---

3. getItems(): List<ImItem>

Description: Returns the list of items the player has collected.

- **Test Case 1**: Player with items
  - **Condition**: Player has collected items.
  - **Example Data**: Items = Sword, Shield.
  - **Expected Outcome**: Returns a list containing Sword and Shield.
- **Test Case 2**: Empty inventory
  - **Condition**: Player has not collected any items.
  - **Example Data**: Items = [].
  - **Expected Outcome**: Returns an empty list.

---

4. pickUpItem(ImItem item): void

Description: Allows the player to pick up an item.

- **Test Case 1**: Successfully pick up an item
  - **Condition**: Player has space in their inventory.
  - **Example Data**: Item = Key; maxItems = 5; currentItems = 3.
  - **Expected Outcome**: Key is added to the player's inventory.
- **Test Case 2**: Inventory full

    - **Condition**: Player's inventory is at capacity.
    - **Example Data**: Item = Key; maxItems = 5; currentItems = 5.
    - **Expected Outcome**: Throws an InventoryFullException.
- **Test Case 3**: Pick up a null item
    - **Condition**: Attempt to pick up a null item.
    - **Example Data**: item = null.
    - **Expected Outcome**: Throws a NullPointerException.

---

5. canCarryMoreItems(): boolean

Description: Checks if the player can carry more items.

- **Test Case 1**: Space available
    - **Condition**: Player has room for more items.
    - **Example Data**: maxItems = 5; currentItems = 3.
    - **Expected Outcome**: Returns true.
- **Test Case 2**: Inventory full
    - **Condition**: Player's inventory is at capacity.
    - **Example Data**: maxItems = 5; currentItems = 5.
    - **Expected Outcome**: Returns false.

---

6. lookAround(World world): boolean

Description: Allows the player to look around their current space for neighbors.

- **Test Case 1**: Neighbors visible
    - **Condition**: The player is in a space with visible neighbors.
    - **Example Data**: Current space = Living Room; neighbors = Kitchen, Bathroom.
    - **Expected Outcome**: Returns true and displays Kitchen, Bathroom.
- **Test Case 2**: No neighbors
    - **Condition**: The player is in a space with no neighbors.
    - **Example Data**: Current space = Attic; neighbors = [].
    - **Expected Outcome**: Returns false and displays "No neighbors found".

---

7. moveTo(ImSpace space): void

Description: Moves the player to a specified space.

- **Test Case 1**: Valid move
    - **Condition**: Player moves to a valid neighboring space.
    - **Example Data**: Current space = Kitchen; Destination = Living Room.
    - **Expected Outcome**: Player is now in the Living Room.
- **Test Case 2**: Move to a non-neighboring space
    - **Condition**: Player attempts to move to a non-neighboring space.
    - **Example Data**: Current space = Kitchen; Destination = Attic.
    - **Expected Outcome**: Throws an InvalidMoveException.
- **Test Case 3**: Move to a null space
    - **Condition**: Player attempts to move to a null space.
    - **Example Data**: space = null.

- **Expected Outcome**: Throws a NullPointerException.

---

8. attemptKill(ImTargetCharacter target, World world, List<PlayerImpl> allPlayers): boolean

Description: Attempts to attack the target character.

- **Test Case 1**: Successful attack
  - **Condition**: Player attacks the target with sufficient visibility and resources.
  - **Example Data**: Target health = 10; Player visibility = true.
  - **Expected Outcome**: Returns true, and target health decreases.
- **Test Case 2**: Failed attack due to visibility
  - **Condition**: Player lacks visibility of the target.
  - **Example Data**: Player visibility = false.
  - **Expected Outcome**: Returns false.

---

9. takeTurn(World world, List<PlayerImpl> allPlayers): boolean

Description: Executes the player's turn.

- **Test Case 1**: Human player takes a valid action
  - **Condition**: Player performs an action during their turn.
  - **Example Data**: Player moves to Kitchen.
  - **Expected Outcome**: Returns true.
- **Test Case 2**: AI player automatically takes a turn
  - **Condition**: AI selects and performs an action.
  - **Example Data**: AI moves to a neighbor.
  - **Expected Outcome**: Returns true.

---

10. bindKey(String key, Runnable action): void

Description: Binds an action to a specific key.

- **Test Case 1**: Bind valid key
  - **Condition**: Bind an action to a valid key.
  - **Example Data**: key = "W"; action = moveUp.
  - **Expected Outcome**: Key "W" triggers the moveUp action.
- **Test Case 2**: Bind null key
  - **Condition**: Attempt to bind a null key.
  - **Example Data**: key = null.
  - **Expected Outcome**: Throws a NullPointerException.

Testing Plan for GameController Class

Responsibilities:

Manages game interaction between the model (World, PlayerImpl, ImTargetCharacter) and the user interface (GameView). Responsible for handling game logic, player turns, and updating the view.

Attributes:

1. world: World – The current game world.
2. players: List<PlayerImpl> – The list of players in the game.

3. maxTurns: int – Maximum allowed turns in the game.
4. currentTurn: int – Tracks the current turn number.
5. targetKilled: boolean – Indicates if the target character has been killed.
6. MAX_ESCAPES: int – Maximum allowed escapes for the target character.
7. doctorEscapeCount: int – Number of times the target character has escaped.
8. gameView: GameView – The view component of the game.

---

1. getPlayers(): List<PlayerImpl>

Description: Returns the list of players in the game.

- **Test Case 1**: Retrieve multiple players
  - **Condition**: The game has multiple players.
  - **Example Data**: Players = Alice, Bob.
  - **Expected Outcome**: Returns a list containing Alice and Bob.
- **Test Case 2**: No players
  - **Condition**: The game has no players added.
  - **Example Data**: Players = [].
  - **Expected Outcome**: Returns an empty list.

---

2. getCurrentTurn(): int

Description: Returns the current turn number.

- **Test Case 1**: Retrieve valid turn number
  - **Condition**: Game is in progress.
  - **Example Data**: currentTurn = 5.
  - **Expected Outcome**: Returns 5.
- **Test Case 2**: Game has not started
  - **Condition**: No turns have been taken.
  - **Example Data**: currentTurn = 0.
  - **Expected Outcome**: Returns 0.

---

3. playTurn(): void

Description: Processes the current player's turn and advances to the next turn.

- **Test Case 1**: Player takes a valid turn
  - **Condition**: The player performs an action during their turn.
  - **Example Data**: Player moves to Kitchen.
  - **Expected Outcome**: Turn advances, and currentTurn increments by 1.
- **Test Case 2**: Game is over
  - **Condition**: Attempt to play a turn when the game is over.
  - **Example Data**: targetKilled = true.
  - **Expected Outcome**: Method does nothing, and currentTurn remains unchanged.

---

4. addPlayer(PlayerImpl player): void

Description: Adds a player to the game.

- **Test Case 1**: Add valid player

    - **Condition**: Add a new player to the game.
    - **Example Data**: Add Alice.
    - **Expected Outcome**: Alice is added to the players list.
- **Test Case 2**: Add null player
    - **Condition**: Attempt to add a null player.
    - **Example Data**: player = null.
    - **Expected Outcome**: Throws a NullPointerException.

---

5. lookAround(PlayerImpl currentPlayer): boolean

Description: Allows the current player to look around their current space for neighbors.

- **Test Case 1**: Neighbors visible
    - **Condition**: Player is in a space with neighbors.
    - **Example Data**: Current space = Kitchen; neighbors = Living Room, Hallway.
    - **Expected Outcome**: Returns true.
- **Test Case 2**: No neighbors
    - **Condition**: Player is in an isolated space.
    - **Example Data**: Current space = Attic; neighbors = [].
    - **Expected Outcome**: Returns false.

---

6. isAttackSeen(PlayerImpl attacker): boolean

Description: Checks if an attack attempt is visible to other players.

- **Test Case 1**: Attack is visible
    - **Condition**: Other players are in neighboring spaces.
    - **Example Data**: Attacker = Alice; Neighbor = Bob.
    - **Expected Outcome**: Returns true.
- **Test Case 2**: Attack is not visible
    - **Condition**: No players are in spaces with visibility to the attack.
    - **Example Data**: Attacker = Alice; No neighbors.
    - **Expected Outcome**: Returns false.

---

7. processAttackResult(PlayerImpl currentPlayer, boolean attackOccurred, ImTargetCharacter targetCharacter): void

Description: Handles the result of an attack attempt on the target character.

- **Test Case 1**: Successful attack
    - **Condition**: The attack was successful.
    - **Example Data**: attackOccurred = true; target.health = 0.
    - **Expected Outcome**: targetKilled is set to true, and the game ends.
- **Test Case 2**: Failed attack
    - **Condition**: The attack failed.
    - **Example Data**: attackOccurred = false.
    - **Expected Outcome**: No changes to target.health or targetKilled.

---

8. isGameOver(): boolean

Description: Checks if the game is over.

- **Test Case 1**: Game over due to target killed
    - **Condition**: The target character has been killed.
    - **Example Data**: targetKilled = true.
    - **Expected Outcome**: Returns true.
- **Test Case 2**: Game over due to escape count
    - **Condition**: The target has escaped the maximum allowed times.
    - **Example Data**: doctorEscapeCount = MAX_ESCAPES.
    - **Expected Outcome**: Returns true.
- **Test Case 3**: Game still ongoing
    - **Condition**: The game is not yet over.
    - **Example Data**: targetKilled = false; doctorEscapeCount < MAX_ESCAPES.
    - **Expected Outcome**: Returns false.

---

9. resetGame(World newWorld): void

Description: Resets the game with a new world configuration.

- **Test Case 1**: Reset with valid world
    - **Condition**: Provide a valid new world.
    - **Example Data**: newWorld has valid spaces, players, and configurations.
    - **Expected Outcome**: The game is reset, world is updated, and currentTurn is set to 0.
- **Test Case 2**: Reset with null world
    - **Condition**: Attempt to reset with a null world.
    - **Example Data**: newWorld = null.
    - **Expected Outcome**: Throws a NullPointerException.

---

10. handleKeyPress(String key): void

Description: Handles key presses for player actions.

- **Test Case 1**: Valid key binding
    - **Condition**: The key is bound to a valid action.
    - **Example Data**: key = "W"; action = moveUp.
    - **Expected Outcome**: Executes the bound action.
- **Test Case 2**: Unbound key
    - **Condition**: The key has no binding.
    - **Example Data**: key = "X".
    - **Expected Outcome**: No action is executed.

---

11. updateView(): void

Description: Updates the graphical view to reflect the current game state.

- **Test Case 1**: View reflects changes
    - **Condition**: Player moves to a new space.
    - **Example Data**: Player moves from Kitchen to Living Room.
    - **Expected Outcome**: View updates to show the player's new position.
- **Test Case 2**: No changes to reflect
    - **Condition**: The game state remains unchanged.

- **Example Data**: No player actions.
- **Expected Outcome**: View remains the same.

Testing Plan for Item Class

Responsibilities:

Represents an item in the game that players can interact with. Items can have damage values and graphical representations.

Attributes:

1. damage: int – The damage value of the item.
2. name: String – The name of the item.
3. image: BufferedImage – The graphical representation of the item.

Methods:

---

1. getDamage(): int

Description: Returns the damage value of the item.

- **Test Case 1**: Valid damage value
  - **Condition**: Retrieve the damage value for an item with a positive damage value.
  - **Example Data**: Item = Sword; damage = 10.
  - **Expected Outcome**: Returns 10.
- **Test Case 2**: Zero damage value
  - **Condition**: Retrieve the damage value for an item with no damage.
  - **Example Data**: Item = Key; damage = 0.
  - **Expected Outcome**: Returns 0.
- **Test Case 3**: Negative damage value (if allowed)
  - **Condition**: Verify behavior for an item with a negative damage value.
  - **Example Data**: Item = Poison; damage = -5.
  - **Expected Outcome**: Returns -5 (if allowed by the game logic).

---

2. getName(): String

Description: Returns the name of the item.

- **Test Case 1**: Valid name
  - **Condition**: Retrieve the name of an item with a valid name.
  - **Example Data**: Item name = "Sword".
  - **Expected Outcome**: Returns "Sword".
- **Test Case 2**: Empty name
  - **Condition**: Retrieve the name of an item with an empty string.
  - **Example Data**: Item name = "".
  - **Expected Outcome**: Returns an empty string "".
- **Test Case 3**: Null name (if allowed)
  - **Condition**: Verify behavior for an item with a null name.
  - **Example Data**: Item name = null.
  - **Expected Outcome**: Returns null or throws a NullPointerException depending on implementation.

---

3. toString(): String

Description: Returns a string representation of the item, typically combining its name and damage.

- **Test Case 1**: Standard item
  - **Condition**: Generate a string representation for an item with a name and damage.
  - **Example Data**: Item = Sword, damage = 10.
  - **Expected Outcome**: Returns "Item: Sword, Damage: 10".
- **Test Case 2**: Item with no damage
  - **Condition**: Generate a string representation for an item with zero damage.
  - **Example Data**: Item = Key, damage = 0.
  - **Expected Outcome**: Returns "Item: Key, Damage: 0".
- **Test Case 3**: Item with an empty name
  - **Condition**: Generate a string representation for an item with no name.
  - **Example Data**: Item name = ""; damage = 5.
  - **Expected Outcome**: Returns "Item: , Damage: 5".

---

4. getImage(): BufferedImage

Description: Returns the graphical representation of the item.

- **Test Case 1**: Valid image
  - **Condition**: Retrieve the image of an item with a valid image assigned.
  - **Example Data**: Item = Sword; Image = BufferedImage instance.
  - **Expected Outcome**: Returns the assigned BufferedImage instance.
- **Test Case 2**: No image assigned
  - **Condition**: Retrieve the image of an item with no image set.
  - **Example Data**: Image = null.
  - **Expected Outcome**: Returns null.

---

5. setImage(BufferedImage image): void

Description: Assigns a graphical representation to the item.

- **Test Case 1**: Assign a valid image
  - **Condition**: Set a valid BufferedImage for an item.
  - **Example Data**: Item = Shield; Image = BufferedImage instance.
  - **Expected Outcome**: The image is successfully assigned to the item.
- **Test Case 2**: Assign a null image
  - **Condition**: Attempt to set a null image for an item.
  - **Example Data**: Image = null.
  - **Expected Outcome**: Throws a NullPointerException or sets the image to null based on implementation.