PRELIMINARY DESIGN

Kill The Joker – Arkham Asylum

Authors

Mohit Rajendra Mhamunkar

Subhankar Shah

Milestone 4

Choice of Model: -

There was quite a discussion on whose model was to be chosen as the implementations were totally different for a different set of features implemented in the previous requirements, but finally we decided to with Subhankar's Model for a various number of reasons stated below.

1. **Multiple Permutations of Human & Computer Players:**
   The model allowed multiple permutations of human and computer players in a single game and with no restrictions on total number of players thus far, which required the least amount of refactoring to only add a restriction of total number of players.

2. **High Accuracy DFS for Pet:**
   The DFS implemented in the following model had a high accuracy and no flaw when tested for traversing in the next space and resetting the path when explicitly moved by a particular player.

3. **Separate Interface for World Initialization:**
   Having a separate interface for world initialization gave us a advantage as the following milestone requires us to ask the user if they wished to implement a separate map from the existing one.

4. **Model Design is Scalable:**
   The model designed is very scalable to implement new features without changing any of the previous features.

Changes in Model: -

There were not a lot of changes in the model as specified above hence it was a reason of our choice, but we have implemented a read only interface which helps view get certain values from the model without making any changes to the values in it.

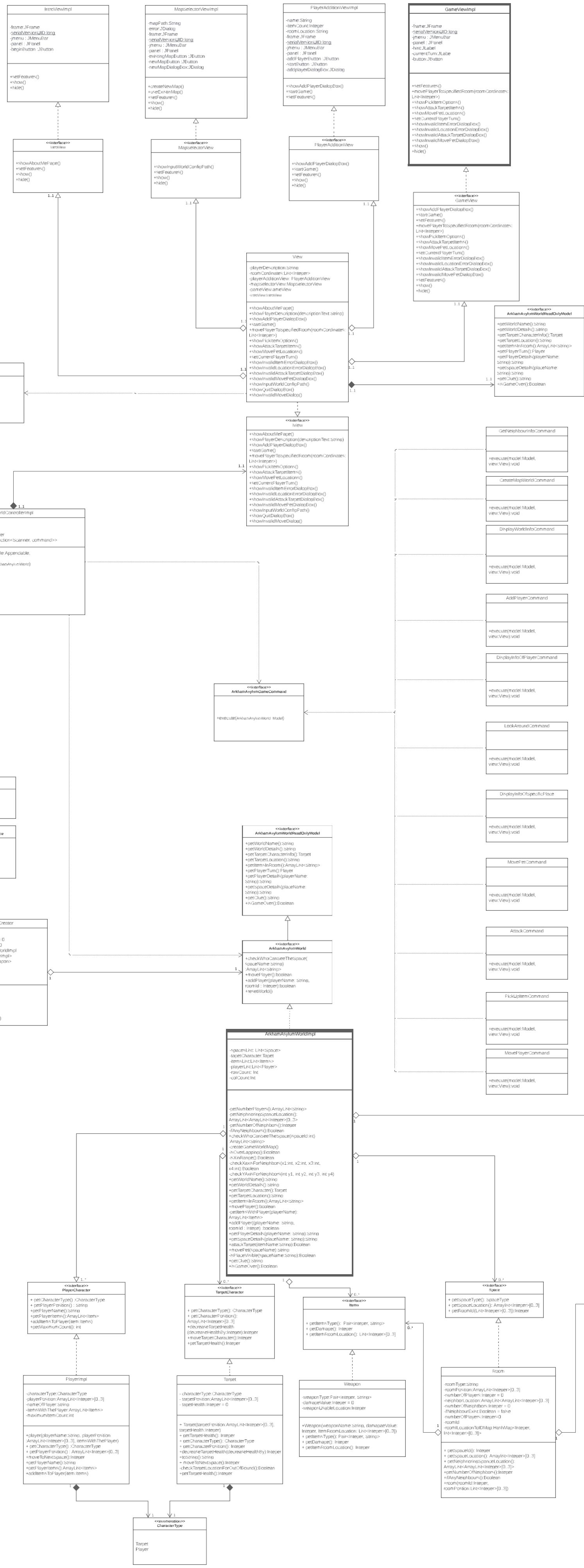
The world has an addition of a resetWorld() method which resets the values in the world whenever the option of starting a game with new or existing Map is being called while playing the game.

Plus to get the name of the space in which a player wants to make a move to a space in the world a getSpaceName() method is added which takes the mouse click coordinates as parameters and return the space name.

Arkham Asylum

Killing the joker

View

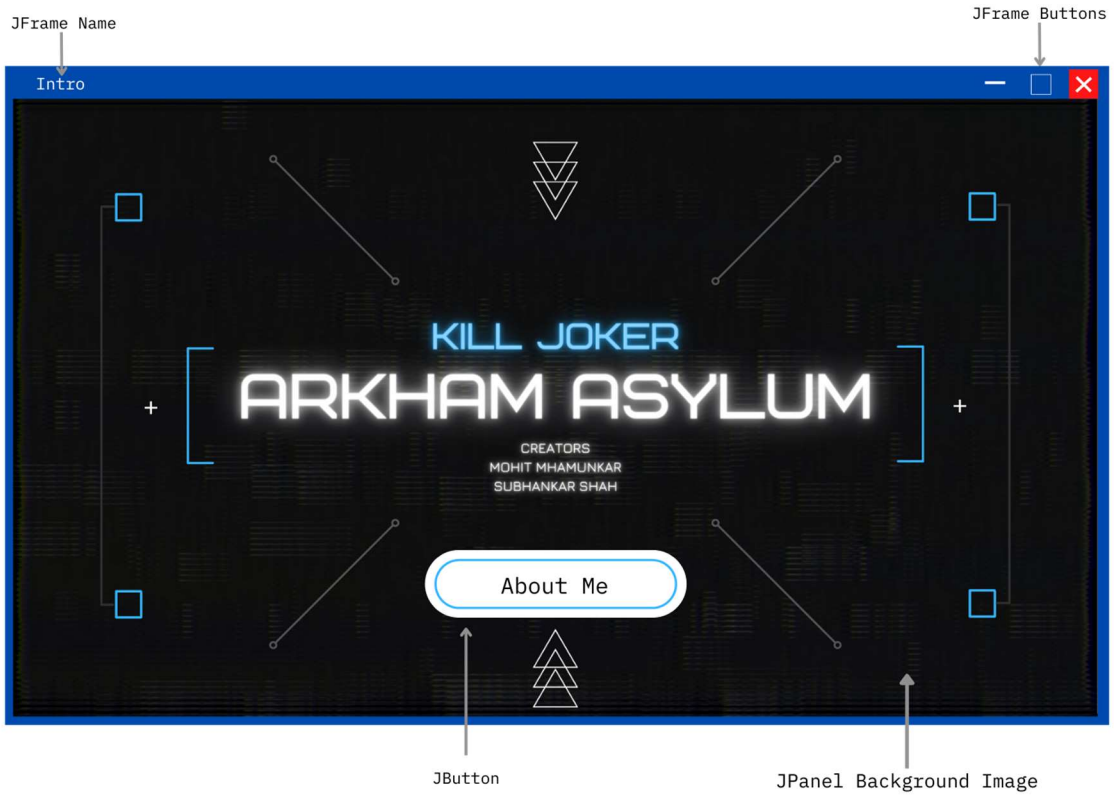


Controller

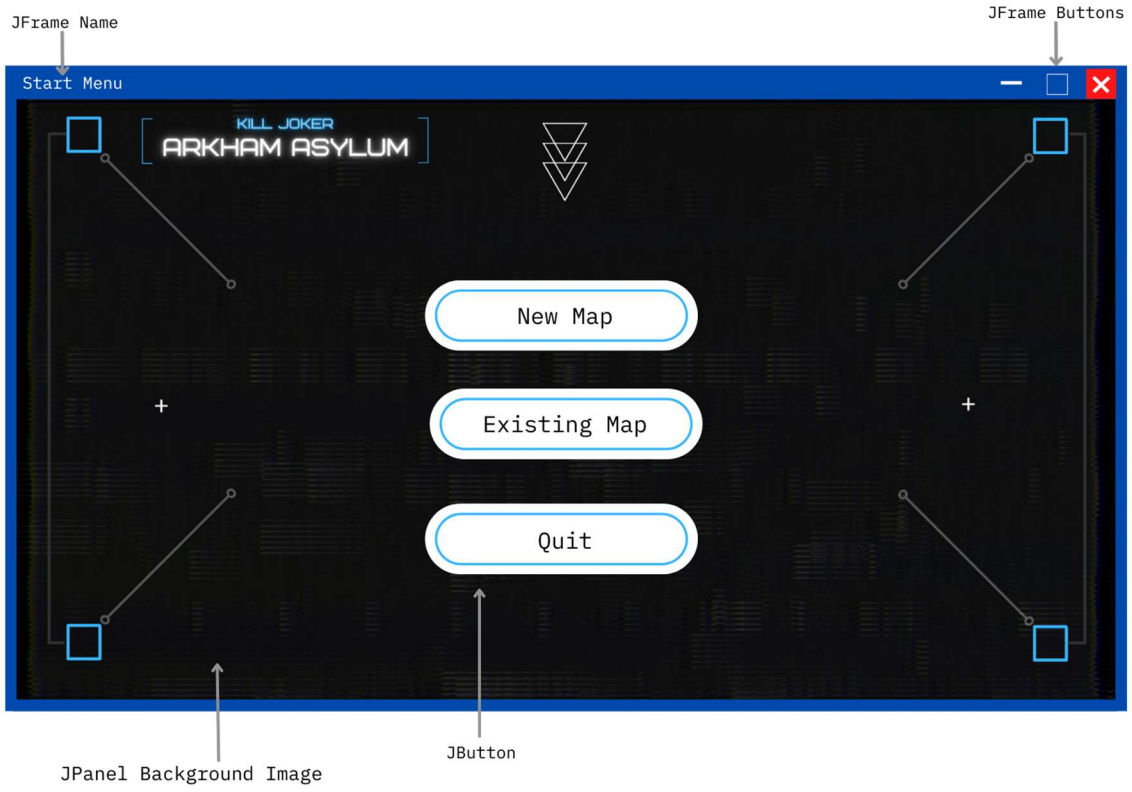
Model

UI Designs

1. Introduction Screen:

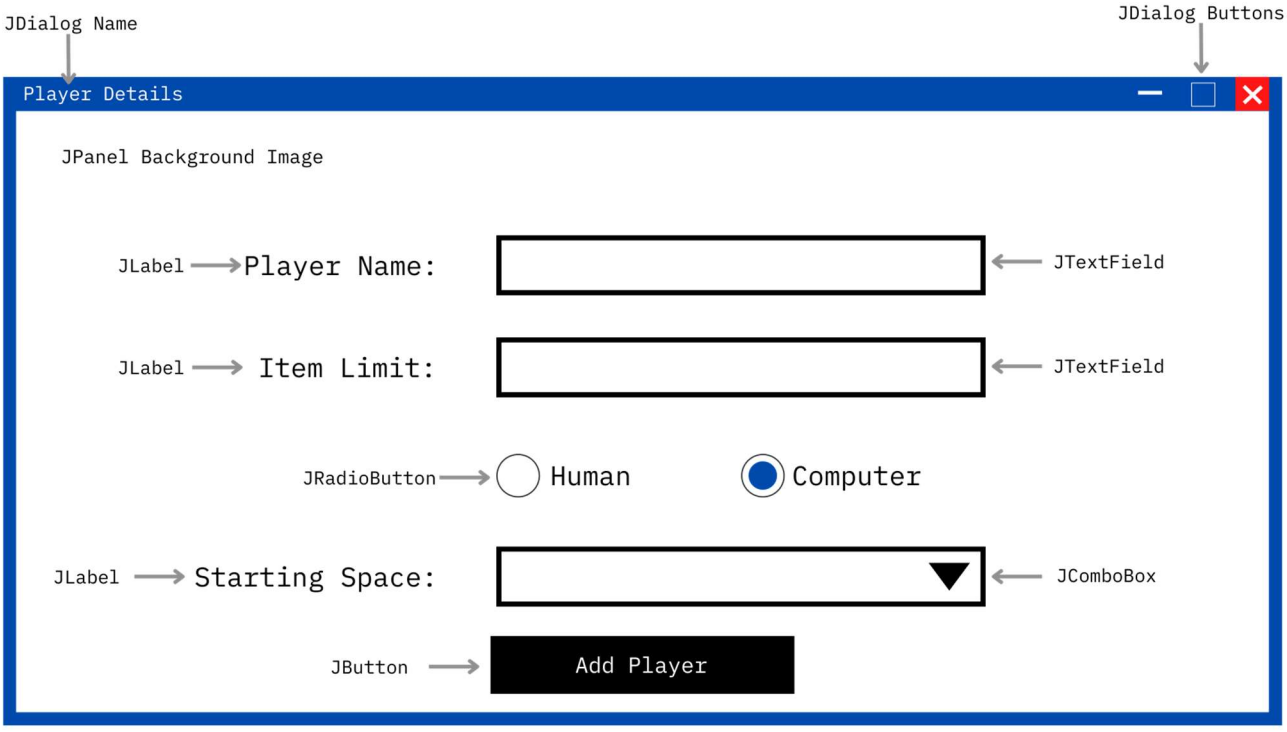


2. Start Menu:

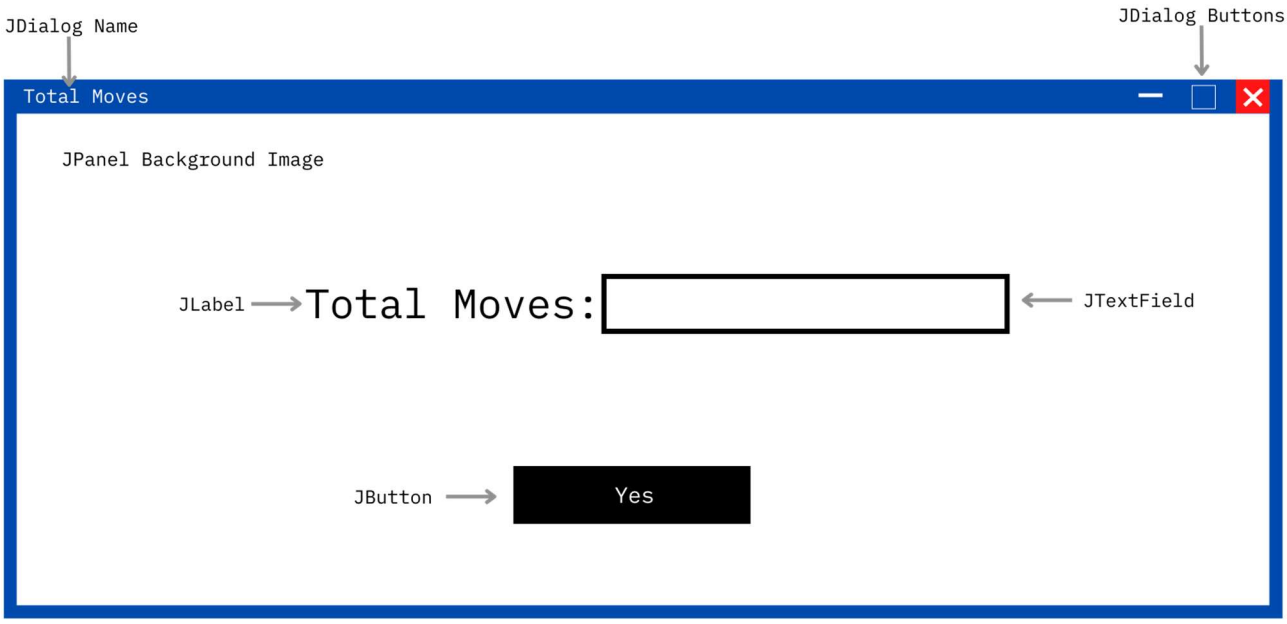


3. Add Player & Start Screen:

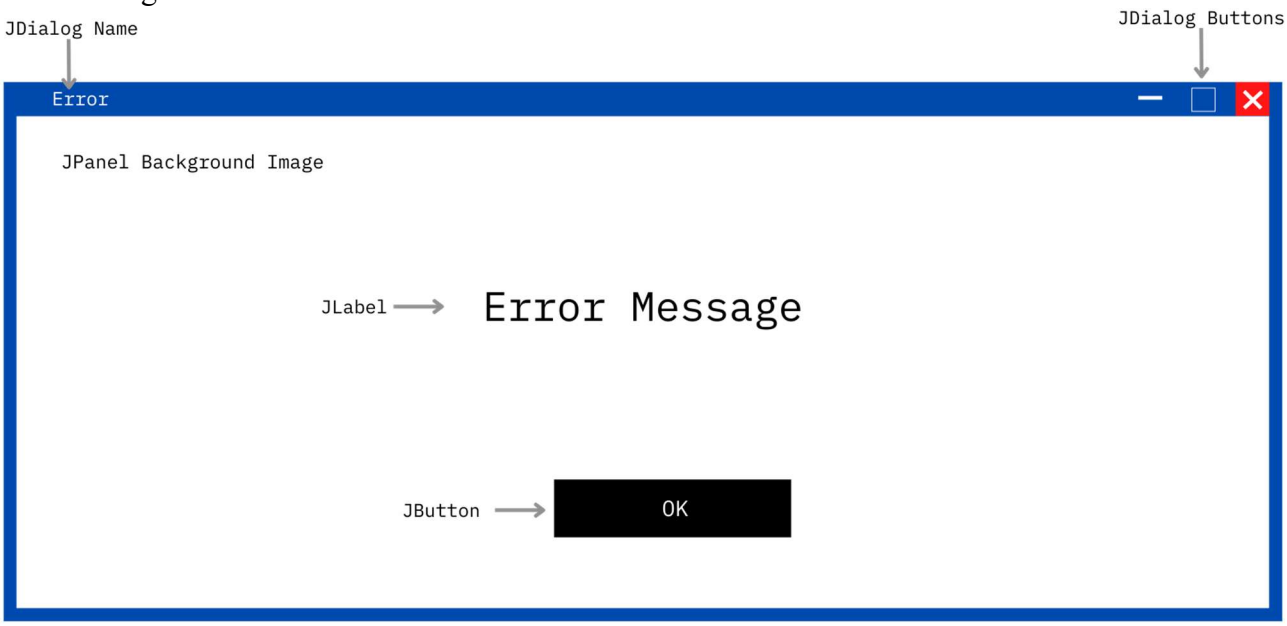
4. Add Player Details:



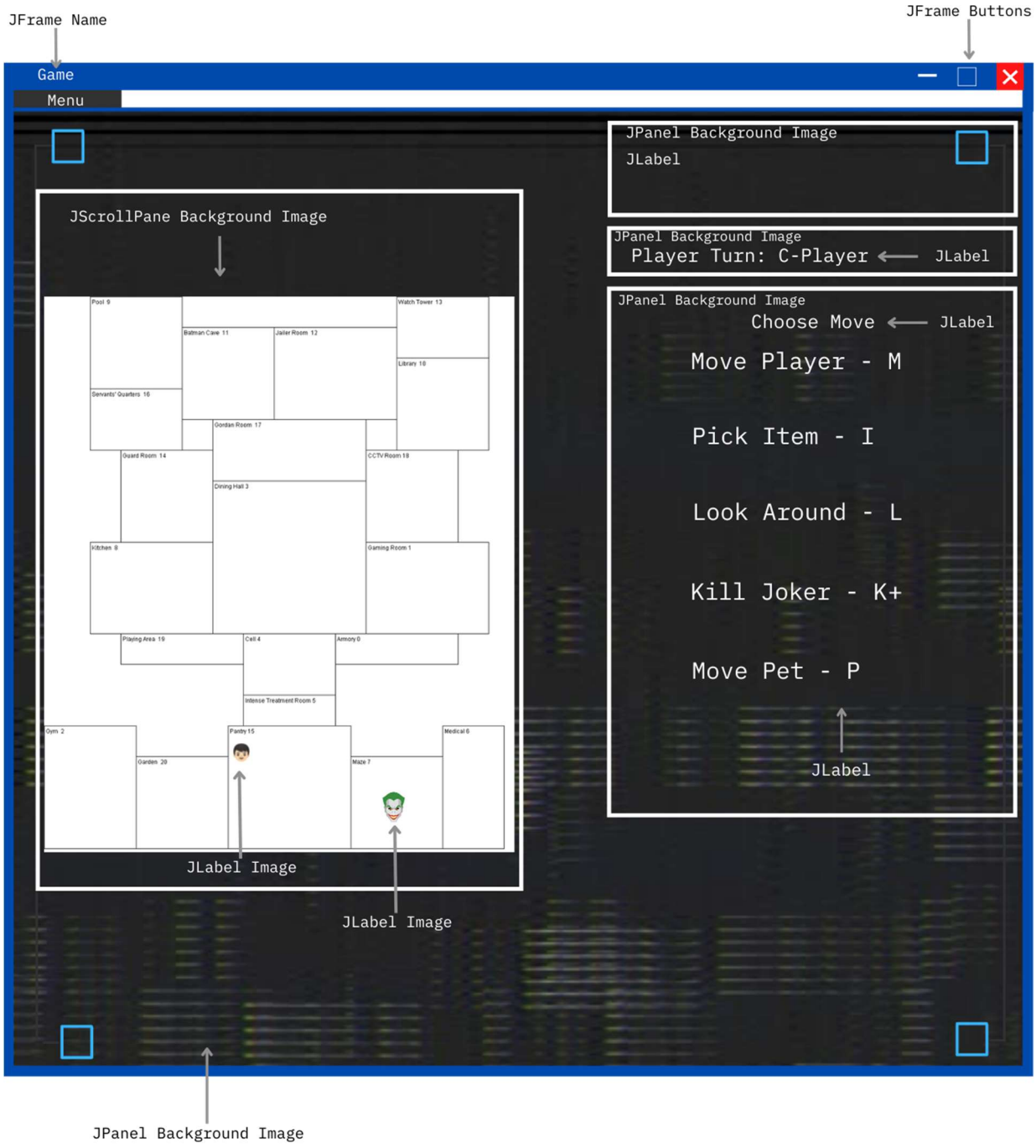
5. Total Players: Dialog:



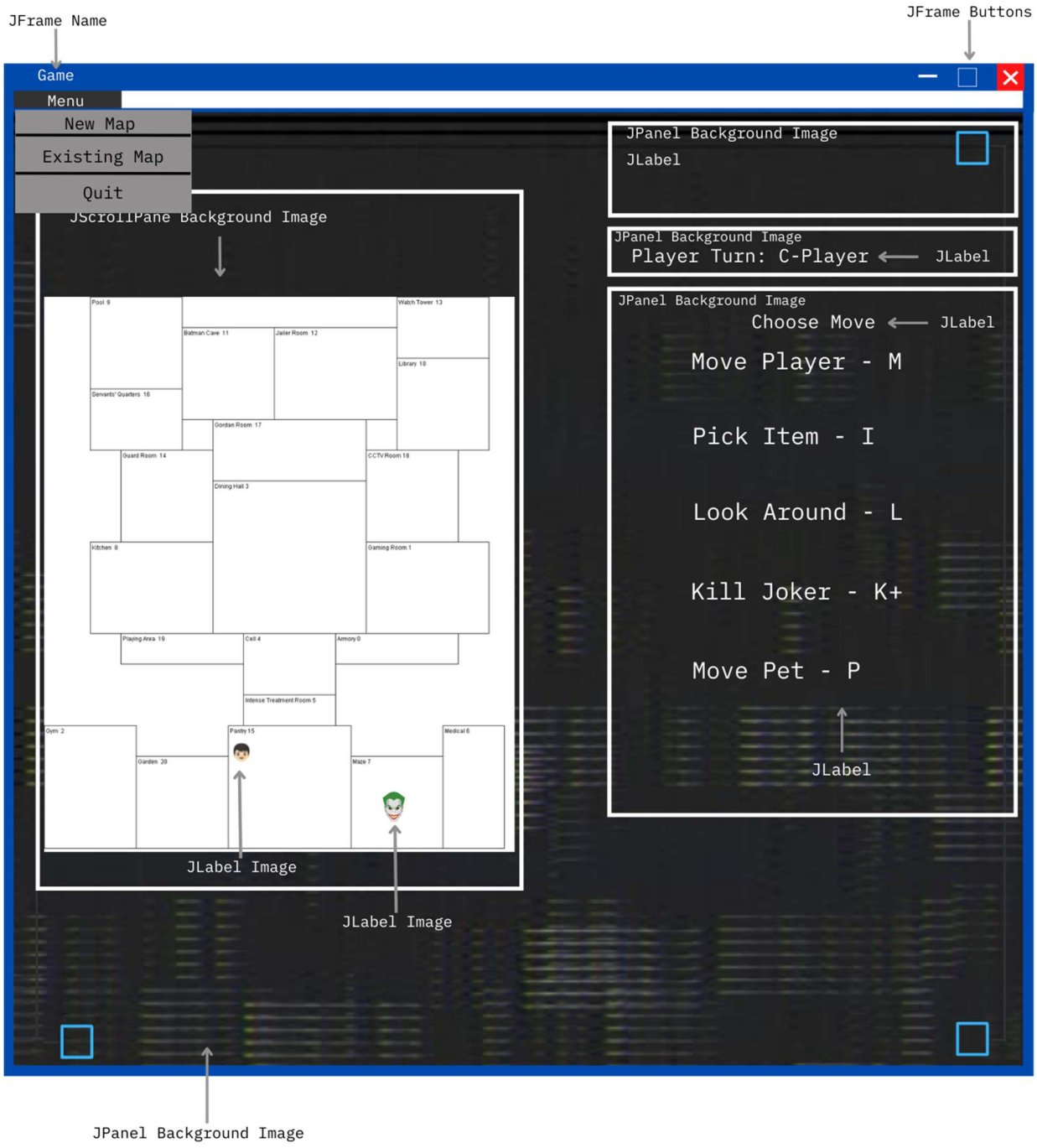
6. Error Dialog:



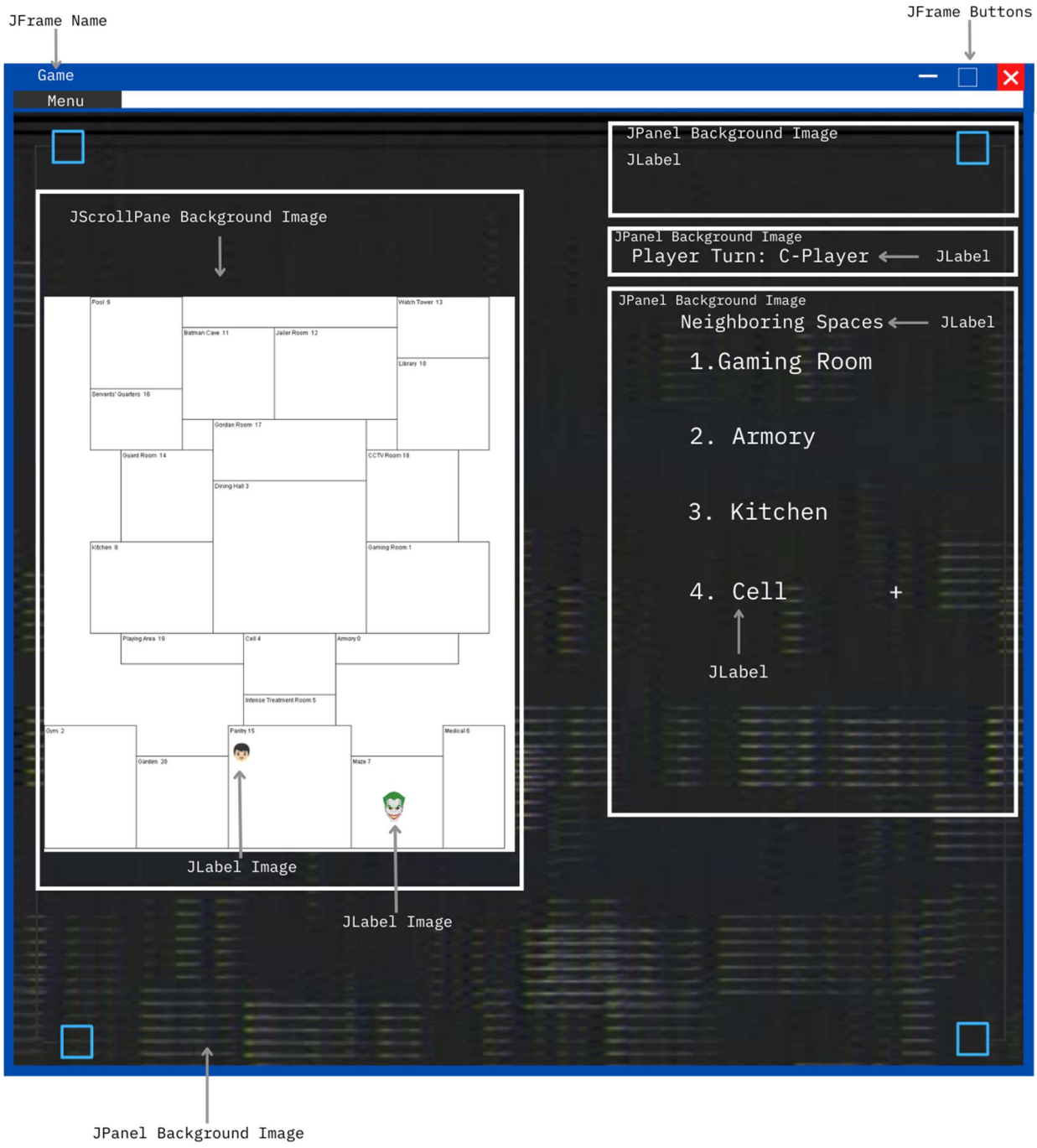
7. Choose Move:



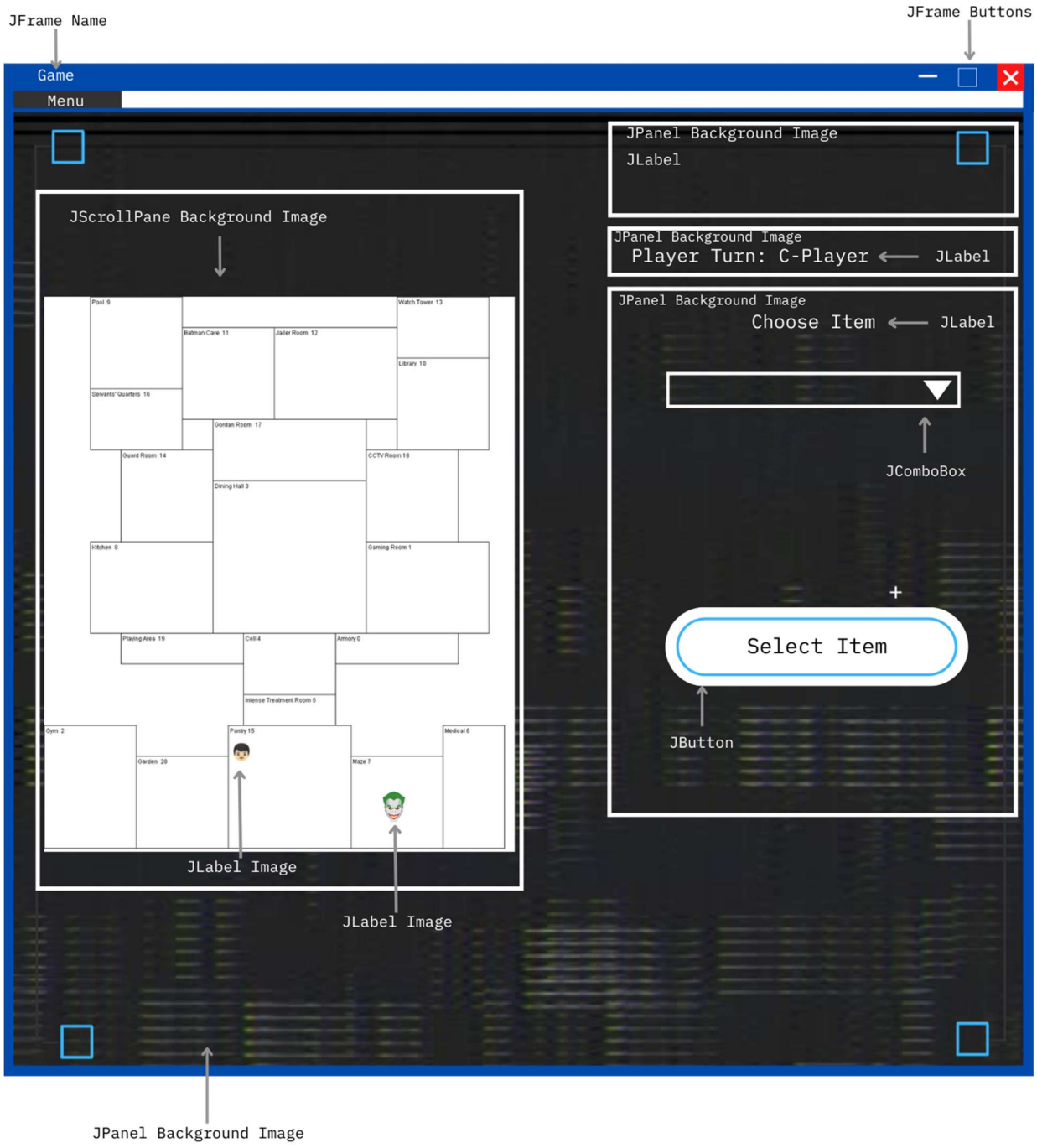
8. Menu View:


JFrame Name
JFrame Buttons
Game
Menu
New Map
Existing Map
Quit
JScrollPane Background Image
JPanel Background Image
JLabel
JPanel Background Image
Player Turn: C-Player
JLabel
JPanel Background Image
Choose Move
JLabel
Move Player - M
Pick Item - I
Look Around - L
Kill Joker - K+
Move Pet - P
JLabel
JLabel Image
JLabel Image
JPanel Background Image

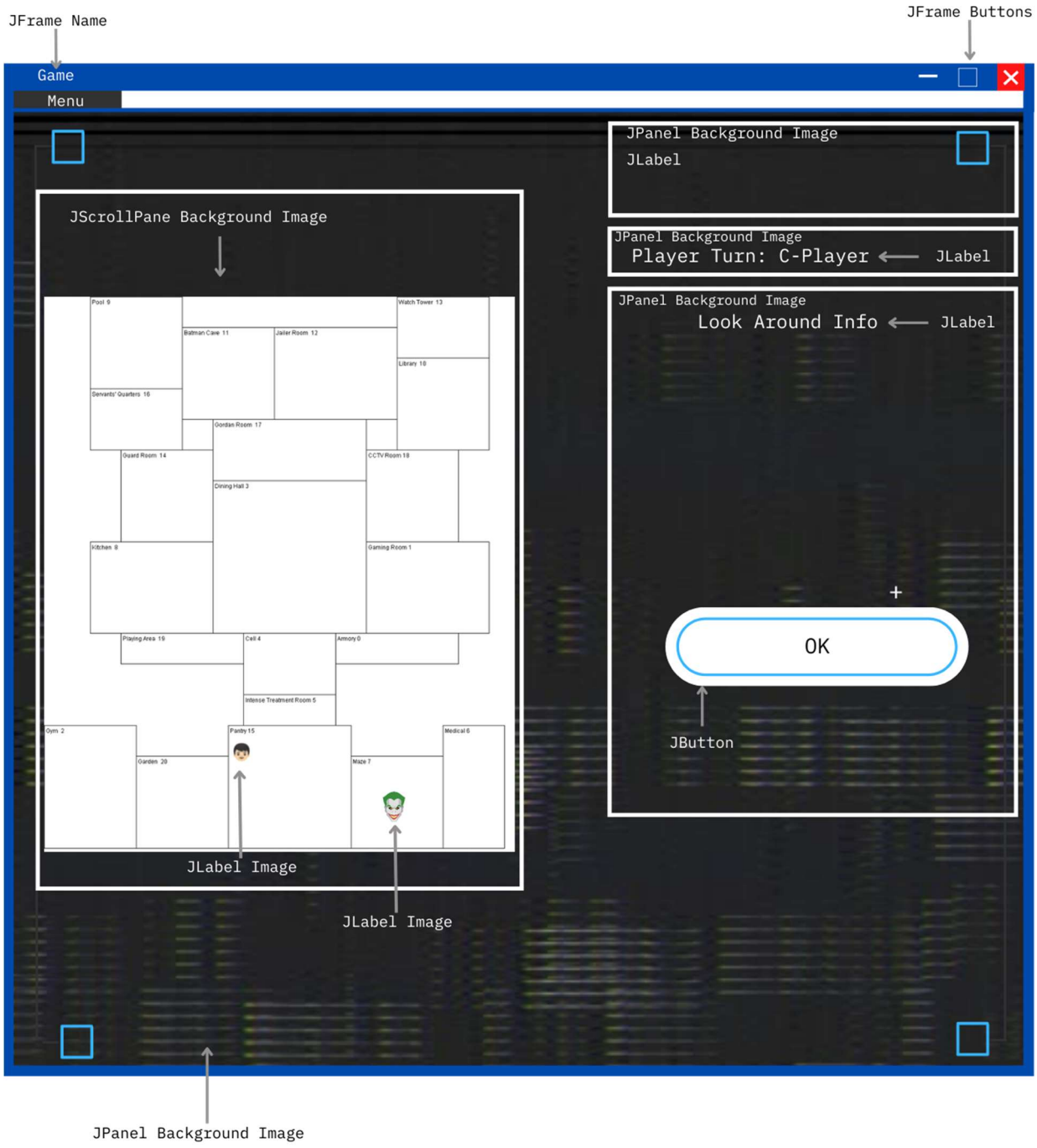
9. Move Player View:



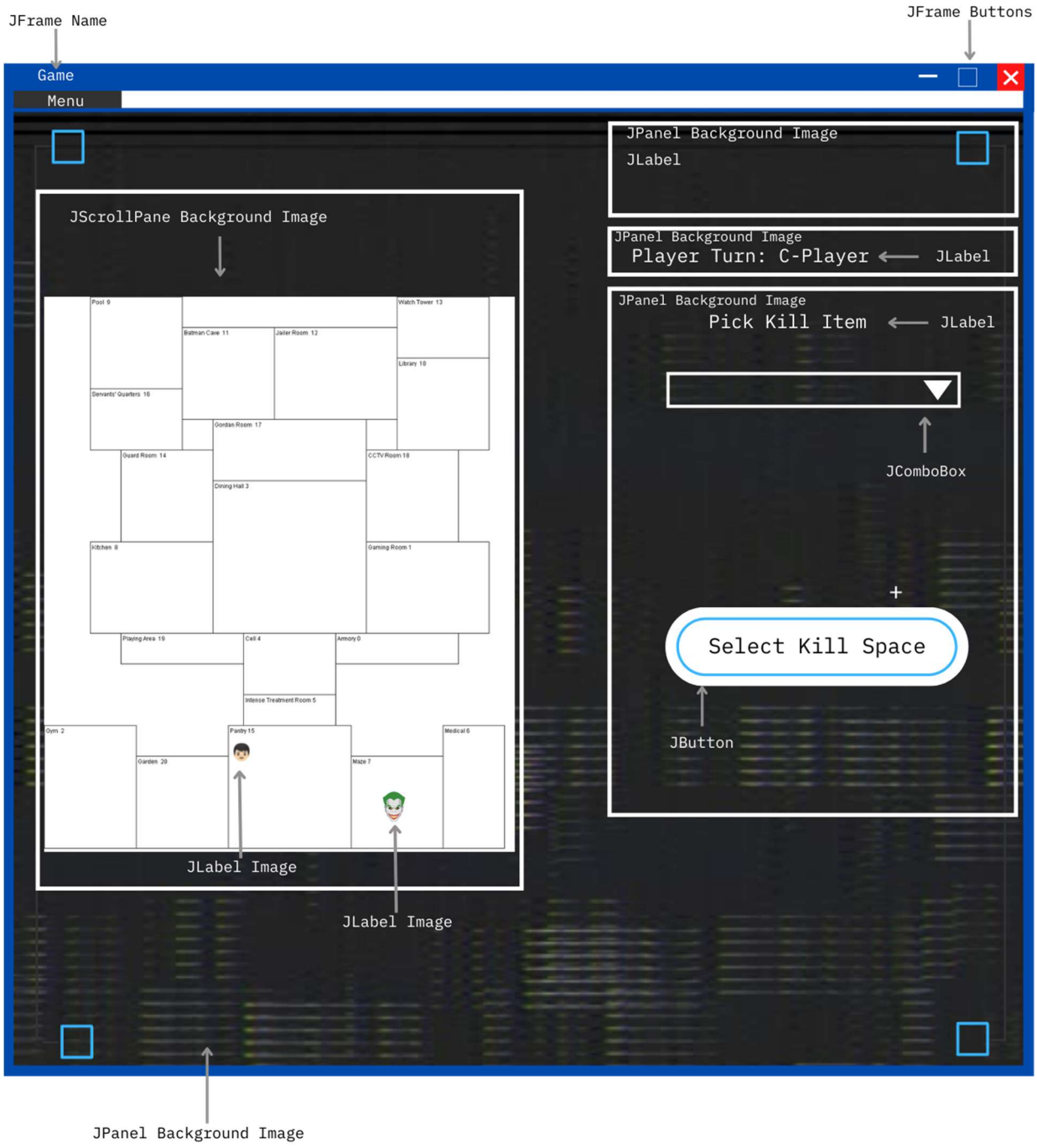
10. Pick Item View:



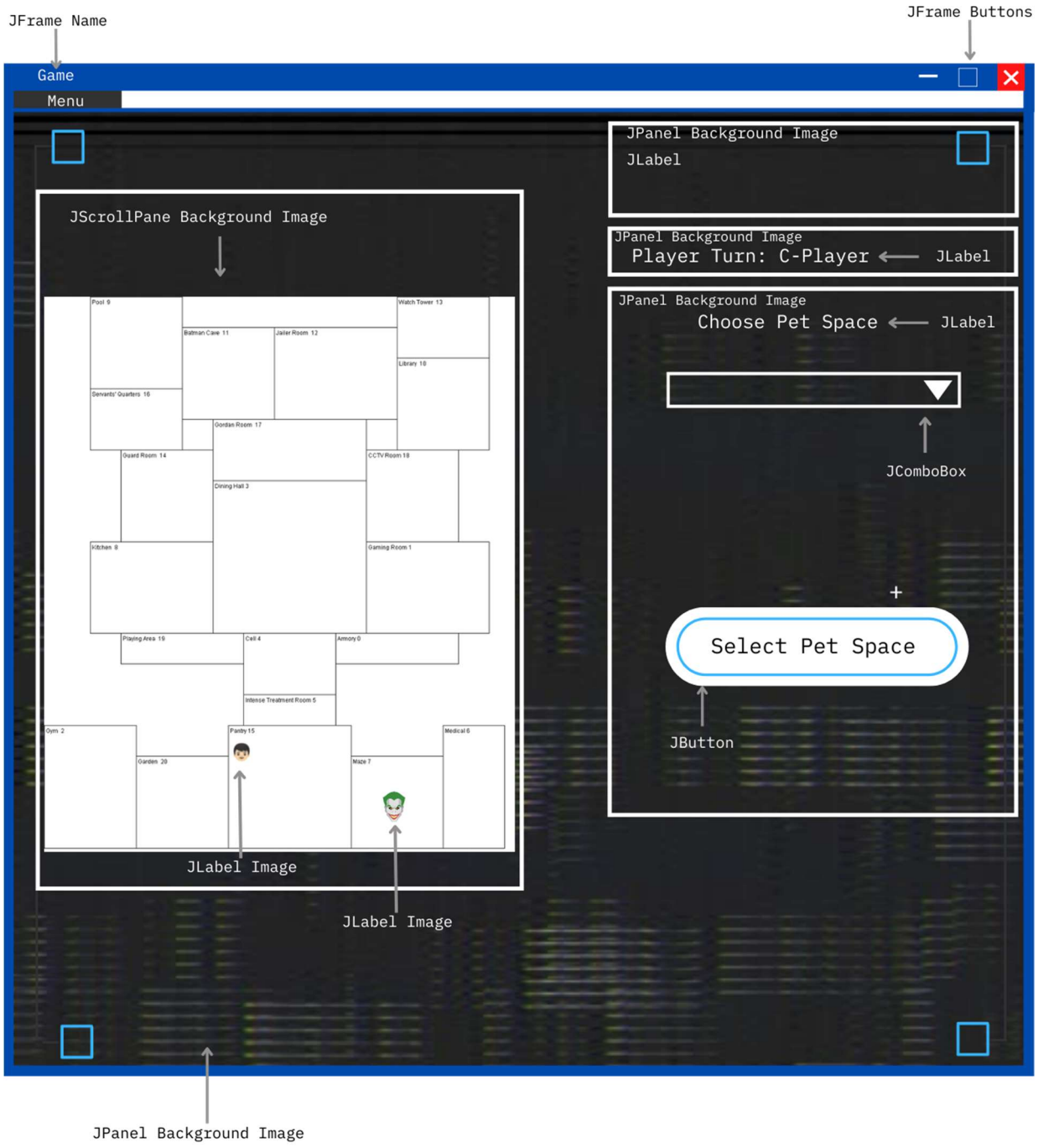
11. Look Around View:



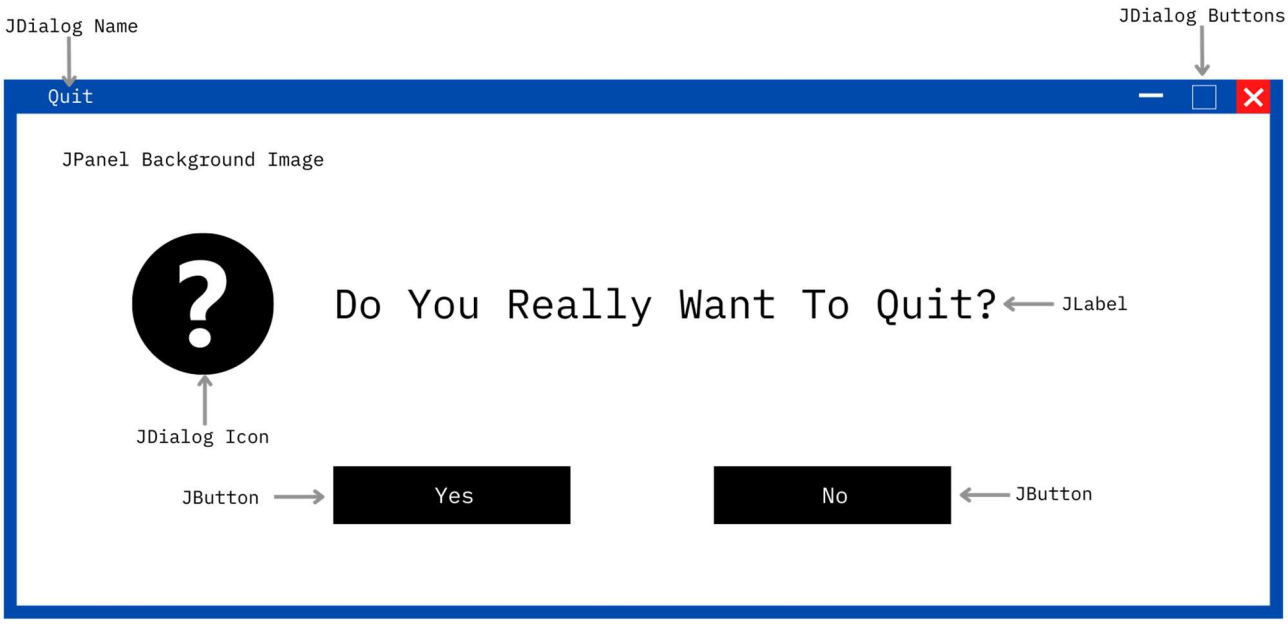
12. Kill Target View:



13. Move Pet View:



14. Quit Dialog:



15. End Game Dialog:

Controller Test

In the testing of the controller, we need to test with mock models to check if our parameters are correctly passed or not. The mock models are also used to check if the function is being called correctly or not. We are going to use3 mock model :
1. WorldMockModel: This mock model will be used to test all the valid test cases.
2. WorldInvalidMockModel: This mock model will be used to test all the invalid test cases.
3. MockView: This mock model will use test view functionalities.

We can use different mocks for failure cases when world or space objects are supposed to throw exceptions.

Example of execution sequence:
For testing the pickUpItem() function, first showPickItemOptions() will be invoked from the view and then pickAnItemByPlayer(String itemName) will be invoked in the model. I am checking if the sequence of the function is being invoked correctly or not and also if the respective functions are invoked properly or not with the provided parameters.

Scenario	**Function**	**Input Value**	**Expected Value**
Testing: pickUpItem(item: String) When a user presses "P". Here, I am testing whether the sequence of the function from the view to the controller to the model is being invoked properly or not.	pickUpItem(item: String)	Input: "Knife"	First in MockView: UniqueCode: 98765. showPickUpItemDialogBox() invoked successfully. Second in MockModel: UniqueCode: 12345. pickUpItemByPlayer() invoked Successfully with input parameter "Knife". Third in MockView: UniqueCode: 6473. showSuccessfullPickUpItemLabel() invoked Successfully with input parameter "Knife".
Testing: Invalid pickUpItem(item: String) When a user presses "P". Here, I am testing for invalid pickUpItem() whether the sequence of	pickUpItem(item: String)	Input: "" Note: Here an empty input is being	First in MockView: UniqueCode: 98765. showPickUpItemDialogBox() invoked successfully. Second in InvalidMockModel: UniqueCode: 12345. IllegalArgumentExcexception error. Third in MockView:

the function from the view to the controller to the model is being invoked properly or not. In to show an unsuccessful message in the view.			UniqueCode: 6473. showUnSuccessfullPickUpItemLabel() invoked Successfully.
Testing: movePlayer(roomName : String) When a user presses "M". Note: Current location of the room: Kitchen. With neighbours: Armoury and garden Here, I am testing for invalid movePlayer() whether the sequence of the function from the view to the controller to the model is being invoked properly or not.	movePlayer(roomName : String)	Input: "Armoury"	First in MockView: UniqueCode: 98765. showMovePlayerDialogBox() invoked successfully. Second in MockModel: UniqueCode: 12345. movePlayer() invoked Successfully with input parameter "Armory". Third in MockView: UniqueCode: 6473. showMovePlayeLabel() invoked Successfully.
Testing: Invalid movePlayer(roomName : String) When a user presses "P". Note: Current location of the pet: Kitchen. With neighbours: Armoury and garden Here, I am testing for invalid movePlayer() whether the sequence of the function from the view to the controller to the model is being	movePlayer(roomName : String)	Input: "Gym" Note: Gym is not one of the neighbours.	First in MockView: UniqueCode: 98765. showMovePlayerDialogBox() invoked successfully. KeyListener() invoked successfully with key press value "M". Second in MockModel: UniqueCode: 12345. movePlayer() invoked Successfully with input parameter "Armory". Third in MockView: UniqueCode: 6473. showMovePlayeLabel() invoked Successfully.

invoked properly or not.			
Testing: movePet(roomName: String) When a user presses “M”. Note: Current location of the pet: Kitchen. Here, I am testing for movePet() whether the sequence of the function from the view to the controller to the model is being invoked properly or not.	movePet(roomName: String)	Input: “Gym”	First in MockView: UniqueCode: 98765. showMovePetDialogBox() invoked successfully. KeyListener() invoked successfully with key press value “P”. Second in MockModel: UniqueCode: 12345. movePet() invoked Successfully with input parameter “Gym”. Third in MockView: UniqueCode: 6473. showMovePetLabel() invoked Successfully.
Testing: movePet(roomName: String) When a user presses “M”. Note: Current location of the pet: Kitchen. Here, I am testing for movePet() whether the sequence of the function from the view to the controller to the model is being invoked properly or not.	movePet(roomName: String)	Input: “Kitchen” Note: The pet is already in Kitchen so it's not a valid input.	First in MockView: UniqueCode: 98765. showMovePetDialogBox() invoked successfully. KeyListener() invoked successfully with key press value “P”. Second in InvalidMockModel: UniqueCode: 12345. movePet() invoked Successfully with input parameter “Gym”. Third in MockView: UniqueCode: 6473. showInvalidMovePetLabel() invoked Successfully.
Testing: attackTarget(itemName: String) When a user presses “A”. Note: Current location of the player: Kitchen	attackTarget(itemName: String)	Input: “Knife” Note: Both target and the player are in the same room and no other player	First in MockView: UniqueCode: 98765. showAttackTargetDialogBox() invoked successfully. Second in MockModel: UniqueCode: 12345. attackTarget() invoked Successfully with input parameter “Knife”. Third in MockView:

and the location of the target is also kitchen Here, I am testing for attackTarget() whether the sequence of the function from the view to the controller to the model is being invoked properly or not.		is watching the attack. Hence the attack is valid.	UniqueCode: 6473. showAttacktLabel(labelText: String) invoked Successfully with input parameter: “Attack successful target health reduced by 10”.
Testing: invalid attackTarget(itemName: String) When a user presses “A”. Note: Current location of the player: Kitchen and the location of the target is also kitchen Here, I am testing for attackTarget() whether the sequence of the function from the view to the controller to the model is being invoked properly or not.	attackTarget(itemName: String)	Input: “Knife” Note: Both target and the player are in the same room but there are other players in the same room so the attack is being watched	First in MockView: UniqueCode: 98765. showAttackTargetDialogBox() invoked successfully. Second in InvalidMockModel: UniqueCode: 12345. IllegalArgumentExceptionError(“Someone is watching the attack”). Third in MockView: UniqueCode: 6473. showAttacktLabel(labelText: String) invoked Successfully with input parameter: “Attack unsuccessful someone is watching the attack”.
Testing: invalid attackTarget(itemName: String) When a user presses “A”. Note: Current location of the player: Kitchen and the location of the target is also Gym Here, I am testing for attackTarget() whether the sequence of the function from the view to the controller to the	attackTarget(itemName: String)	Input: “Knife” Note: Target and the player are not in the same room.	First in MockView: UniqueCode: 98765. showAttackTargetDialogBox() invoked successfully. Second in InvalidMockModel: UniqueCode: 12345. IllegalArgumentExceptionError(“Target and the player are not in the same room”). Third in MockView: UniqueCode: 6473. showAttacktLabel(labelText: String) invoked Successfully with input parameter: “Attack unsuccessful target and the character are not in the same room”.

model is being invoked properly or not.			
Testing: lookAround() When a user presses "L". Note: Current location of the player: Kitchen and the location of the target is also kitchen. Here, I am testing for lookAround() whether the sequence of the function from the view to the controller to the model is being invoked properly or not.	lookAround()	Note: No parameter is required.	First in MockView: UniqueCode: 98765. showLookAroundtDialogBox() invoked successfully. Second in MockModel: UniqueCode: 12345. lookAround() invoked Successfully. Third in MockView: UniqueCode: 6473. showLookAroundLabel(labelText: String) invoked Successfully with input parameter: "Items in the space: Gun. Players in the space: John, Curry". Neighbouring spaces: Gym, Garden. Items in Gym: "RPG". Iem in garden: "No items in the garden". Target Character location: "Kitchen"
Testing: displayPlayerInfo() When a user presses "I". Note: Current location of the player: Kitchen and the location of the target is also kitchen. Here, I am testing for displayPlayerInfo() whether the sequence of the function from the view to the controller to the model is being invoked properly or not.	displayPlayerInfo()	Input: "John"	First in MockView: UniqueCode: 98765. showdisplayPlayerInfoDialogBox() invoked successfully. Second in MockModel: UniqueCode: 12345. displayPlayerInfo(playerName) invoked Successfully. With player Name "John". Third in MockView: UniqueCode: 6473. showLookAroundLabel(labelText: String) invoked Successfully with input parameter: "John Information"
Testing: displaySpaceInfo()	displaySpaceInfo()	Input: "Armoury"	First in MockView: UniqueCode: 98765. showDisplaySpaceInfoDialogBox() invoked successfully.

When a user presses "S". Note: Current location of the player: Kitchen and the location of the target is also kitchen. Here, I am testing for displaySpaceInfo() whether the sequence of the function from the view to the controller to the model is being invoked properly or not.			Second in MockModel: UniqueCode: 12345. displaySpaceInfo(playerName) invoked Successfully. With player Name "Armoury". Third in MockView: UniqueCode: 6473. showdisplaySpaceInfoLabel(labelText: String) invoked Successfully with input parameter: "Armoury Information"
Testing: resetWorld() When the user presses reset world from the menu. Here, I am testing for resetWorld() whether the sequence of the function from the view to the controller to the model is being invoked properly or not.	resetWorld()	No input required	First in MockView: UniqueCode: 98765. showResetWorldDialogBox() invoked successfully. Second in MockModel: UniqueCode: 12345. resetWorld() invoked Successfully.". Third in MockView: UniqueCode: 6473. showResetWorldDialogBox(labelText: String) invoked Successfully.

Testing scenarios of View

Details of MockView

For this, I have created a **MockView** class which implements the IView interface.
The test consists of a unique value which is used to determine if the method of the specific view is being invoked or not.

Scenario	**Function**:	**Input Value**	**Expected Value**
Testing showPlayerDescription() Note: I am using MockView to test if showPlayerDescription is being invoked successfully or not	showPlayerDescription()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showPlayerDescription function is invoked successfully.

Testing showAddPlayerDialogBox() Note: I am using MockView to test if showAddPlayerDialogBox is being invoked successfully or not	showAddPlayerDialogBox()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showAddPlayerDialogBox function is invoked successfully.
Testing startGame() Note: I am using MockView to test if startGame is being invoked successfully or not	startGame()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. startGame function is invoked successfully.
Testing movePlayerToSpecifiedRoom() Note: I am using MockView to test if movePlayerToSpecifiedRoom is being invoked successfully or not	movePlayerToSpecifiedRoom()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. movePlayerToSpecifiedRoom function is invoked successfully.
Testing showPickItemOptions() Note: I am using MockView to test if showPickItemOptions is being invoked successfully or not	showPickItemOptions()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showPickItemOptions function is invoked successfully.
Testing showAttackTargetItems() Note: I am using MockView to test if showAttackTargetItems is being invoked successfully or not	showAttackTargetItems()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showAttackTargetItems function is invoked successfully.
Testing showMovePetLocations() Note: I am using MockView to test if showMovePetLocations is being invoked successfully or not	showMovePetLocations()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showMovePetLocations function is invoked successfully.

Testing setCurrentPlayerTurn() Note: I am using MockView to test if setCurrentPlayerTurn is being invoked successfully or not	setCurrentPlayerTurn()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. setCurrentPlayerTurn function is invoked successfully.
Testing showInvalidItemErrorDialogBox() Note: I am using MockView to test if showInvalidItemErrorDialogBox is being invoked successfully or not	showInvalidItemErrorDialogBox()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showInvalidItemErrorDialogBox function is invoked successfully.
Testing showInvalidLocationErrorDialogBox() Note: I am using MockView to test if showInvalidLocationErrorDialogBox is being invoked successfully or not	showInvalidLocationErrorDialogBox()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showInvalidLocationErrorDialogBox function is invoked successfully.
Testing showInvalidAttackTargetDialogBox() Note: I am using MockView to test if showInvalidAttackTargetDialogBox is being invoked successfully or not	showInvalidAttackTargetDialogBox()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showInvalidAttackTargetDialogBox function is invoked successfully.
Testing showInvalidMovePetDialogBox() Note: I am using MockView to test if showInvalidMovePetDialogBox is being invoked successfully or not	showInvalidMovePetDialogBox()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showInvalidMovePetDialogBox function is invoked successfully.

Testing showInputWorldConfigPath() Note: I am using MockView to test if showInputWorldConfigPath is being invoked successfully or not	showInputWorldConfigPath()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showInputWorldConfigPath function is invoked successfully.
Testing showQuitDialogBox() Note: I am using MockView to test if showQuitDialogBox is being invoked successfully or not	showQuitDialogBox()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showQuitDialogBox function is invoked successfully.
Testing showInvalidMoveDialog() Note: I am using MockView to test if showPlayerDescription is being invoked successfully or not	showInvalidMoveDialog()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showInvalidMoveDialog function is invoked successfully.
Testing showAboutMePage() Note: I am using MockView to test if showAboutMePage is being invoked successfully or not	showAboutMePage()	Note: No parameter is required here	**Expected:** UniqueCode: 12345. showAboutMePage function is invoked successfully.

View constructor Test

Note: We are using a mock view to check if every function in the IView interface is being invoked or not.

Constructor test
The constructor accepts view-only model
ArkhamAsylumWorldReadOnly readOnlyModel: denotes the model for read only values

Scenario	**Function**: **IView**(ArkhamAsylumWorldReadOnly obj) Testing for ?	**Input Value**	**Expected Value**
Initialise the read only object	New IView(obj)	Obj = null	**Expected**: IllegalArgumentException

with a null value In this case it should not accept the value and ask the user for correct read only object			**Message**: Read only model cannot be null
Initialise the read only object with actual value In this case the construct or should be invoked without any errors.	New IView(obj)	Obj = new ArkhamAsylumWorldReadOnly();	**Constructor is invoked successfully**

Model Tests

Function: resetWorld()
Note:
The function takes no arguments and resets the model after a new map or existing map is called from the menu while playing the game.

In world implementation, when the user will click to start the game again with a new or existing map from the options in the menu while playing the game, this method will be called, and the model will be reset.

Scenario	Function: **world.resetWorld()**	**Input Value**	**Expected Value**
The reset world method is called by the controller on click of one of the options.	world.resetWorld()	Note: No parameter is required here	**Expected**: Values of the world are set to the new or existing world as specified by the user Note: As the world is reseted successfully there will be no error given.

PetImpl Test:

Note: The test cases of the Pet class can be broken down into 2 categories. First is the test of the constructor and second is the test of the functions of the PetImp class.

Constructor test
The constructor of the test class accepts 2 values:
String name: denotes the name of the pet.
String roomName: denotes the initial space location of the pet

Scenario	**Function**: **PetImpl**(String name, String roomName) Testing for ?	**Input Value**	**Expected Value**
Initialise the pet's name with an empty String. In this case the it should not accept the value and ask the user to input the name of the pet in correct format	New Pet(name, roomName)	Name = "" roomName = target.getRoomLocation()	**Expected**: IllegalArgumentException **Message**: Name should not be blank.

Initialise the pet's name with a valid string but the pet's roomName is not valid.	New Pet(name, roomName)	Name= "Fortune the Cat" roomName = ""	**Expected**: IllegalArgumentException **Message**: Room name should not be blank.
Initialise the pet's name with a valid String.	New Pet(name, roomName)	Name="Fortune the Cat" roomName = target.getRoomLocation()	Void Note: "As the expected values are correct the constructor won't throw any error"

PetImpl function: getLocation()
Note:
The constructor of the class is initialised with the following values:
name = **"**Fortune the Cat"
roomName=target.getRoomLocation()
The value of target.getRoomLocation = Armory

Scenario	Function: **getLocation()**	**Input Value**	**Expected Value**
Testing the initial location of the pet. Note: At first the initial location of the Target character is Armory as that's the index location of the first Room.	pet.getRoomLocation()	Void Note: No input parameter required for this function	"Armory"
Test pet's location after moving the pet to the neighbouring location. Note: In the starting of the game the pet is in the Armory, then the player do movePet("Kitchen") this will move the pet to the kitchen.	pet.getRoomLocation()	Void Note: No input parameter required for this function	"Kitchen"
Test pet's location after moving the pet to the invalid neighbouring location.	pet.getRoomLocation()	Void	"Kitchen" Note: "The location of the pet is not changed"

Note: After moving the pet to the invalid space movePet("1234") function will throw an illegalArgumentException. Hence the pet will remain in the previous specified space that is Kitchen.		Note: No input parameter required for this function	

Function: movePet(String spaceLocation)
Note:
The function takes a string location. It validates if the location is valid or not and then sets the room location of the pet to the specified room.

In world implementation, when the user will input the command to move the pet to an invalid space. The function will check if the room is valid or not. If the room is not valid then the function throws an IllegalArgumentException error.

Scenario	Function: **world.movePet(String room location)**	**Input Value**	**Expected Value**
Move the pet to an invalid room location.	world.movePet(spaceLocation)	spaceLocation = "!@#$%"	**Expected**: IllegalArgumentException **Message**: The space name should be valid
Move the pet to an invalid room location.	world.movePet(spaceLocation)	spaceLocation = ""	**Expected**: IllegalArgumentException **Message**: The space name should be valid
Move the pet to the same room he is already in. Note: The current location of the pet is in Armory	world.movePet(spaceLocation)	spaceLocation = "Armory"	**Expected**: IllegalArgumentException **Message**: Pet is already in the location.
Move the pet to a valid location.	world.movePet(spaceLocation)	spaceLocation="Kitchen"	Void

			Note: As the location is valid it returns a void hence accepting the parameter

Function: movePetDFT()
Note:
The function moves the pet automatically from one space to another. The function uses depth first search algorithm to move the pet to the adjacent neighbour.

In the Arkham Asylum world, the space are declared in the following order:

1 Armory
2 Gaming Room
3 Gym
4 Dining Hall
5 Cell
6 Intense Treatment Room
7 Medical
8 Maze
9 Kitchen
10 Pool
11 Library
12 Batman Cave
13 Jailer Room
14 Watch Tower
15 Guard Room
16 Pantry
17 Servants' Quarters
18 Gordan Room
19 CCTV Room
20 Playing Area
21 Garden

After each player's turn the function will be called the pet will move the space which is in the next order from the list.

Scenario	Function: **world.movePetDFT()**	**Input Value**	**Expected Value**
After each turn of the player or computer, if movePet is not explicitly called, world.moveDFT() will be called. Suppose the starting location of the pet is Armory and the current player plays a turn to move to	world.movePetDFT()	Void Note: Here no input is required.	Expected output of world.movePet(): Void Note: We can confirm if the pet moved to the correct location by testing

neighbourhood, after the turn movePetDFT() will be called.			pet.getLocation: "Gaming Room"
After each turn of the player or computer, if movePet is not explicitly called, world.moveDFT() will be called. Suppose the starting location of the pet is "Gaming Room" and the current player plays a turn to move to neighbourhood, after the turn movePetDFT() will be called.	world.movePetDFT()	Void Note: Here no input is required.	Expected output of world.movePet(): Void Note: We can confirm if the pet moved to the correct location by testing pet.getLocation: "Gym"
Testing when the pet has reached the last room in the list, i.e Garden Now, the pet should go back to the first room in the list, i.e Armory	world.movePetDFT()	Void Note: Here no input is required.	Expected output of world.movePet(): Void Note: We can confirm if the pet moved to the correct location by testing pet.getLocation: "Armory"
If the makes a command to movePet and specify explicitly where the pet should move then this function should not be called.	world.movePet(String spaceName)	spaceName = "Kitchen"	Void Note: As the location is valid it returns a void hence accepting the parameter We can confirm if the pet moved to the correct location by testing pet.getLocation: "Kitchen"

Function: attackTarget(String itemName)
Note:
The function takes an item as an input. By default Poke will be available with the users to use.

If the item entered by the user is invalid then the function will throw an IllegalArgumentException error. If another player can see the player attacking from the neighbouring room then the function throws IllegalStateExcpeption

Scenario	Function: **world.attackTarget(String itemName)**	**Input Value**	**Expected Value**
Player attacks the target with an Invalid item. Note: Assume the player has following items with himself: "Item: Knife, Damage Value: 50", "Item: Gun, Damage value: 80" and the total health value of the Target is 100	world.attackTarget(String itemName)	itemName= "!@$"	**Expected**: IllegalArgumentException **Message**: The user does not have the specified item
Player attacks the target with an item which he has already used. Note: The player has already used knife, and the player has following items with himself: "Item: Gun, Damage value: 80" and the total health value of the Target is 100	world.attackTarget(String itemName)	itemName= "knife"	**Expected**: IllegalArgumentException **Message**: The player has already used the following item.
Player attacks the target without any item. In this case the player can POKE the target, with damage value of 10 Note: The total health value of the Target is 100	world.attackTarget(String itemName)	itemName= "poke"	Void Note: As the item is valid it returns a void hence accepting the parameter We can confirm if the health of the target decreased by using target,get Health() function: 90
Player attacks the target with a valid item. Note: Assume the player has following items with himself:	world.attackTarget(String itemName)	itemName= "gun"	Void Note: As the item is valid it returns a void hence accepting the parameter

"Item: Gun, Damage value: 80" and the total health value of the Target is 90			We can confirm if the health of the target decreased by using target,get Health() function: 10
Player attacks the target with a valid item but another player can see the player from the neighbouring room. Note: Assume the player has following items with himself: "Item: Knife, Damage Value: 50", "Item: Gun, Damage value: 80" and the total health value of the Target is 100	world.attackTarget(String itemName)	itemName= "gun"	**Expected**: IllegalStateException **Message**: Someone can see you while attacking. Note: We can check if the items is used by the player by using the following player.getItems: "Gun"

Function: isGameOver()

Note: After every turn the controller will check if the game is over or not. The game can only be over in 2 ways:

1. If the target is killed.
2. If the total number of turns gets over.

Scenario	Function: **world.attackTarget(String itemName)**	**Input Value**	**Expected Value**
Player attacks the target with a valid item. Note: Assume the player has following items with himself: "Item: Gun, Damage value: 80" and the total health value of the Target is 80	world.isGameOver()	Void Note: Here no input is required.	true
Player attacks the target with a valid item.	world.isGameOver()	Void Note: Here no input is required.	false Note: The target health is 10

Note: Assume the player has following items with himself: “Item: Gun, Damage value: 80” and the total health value of the Target is 90			hence the game is not yet over.
Player makes a turn. Remaining turns: 0	world.isGameOver()	Void Note: Here no input is required.	True Note: As the remaining turn is over then the game is over.
Player makes a turn. Remaining turns: 2	world.isGameOver()	Void Note: Here no input is required.	false Note: As the remaining turn is not over then the game is not over.

Controller command Tests

In the testing of the controller, we need to test with mock models to check if our parameters are correctly passed or not. The mock models are also used to check if the function is being called correctly or not. We are going to use 2 mock model:
1. WorldMockModel: This mock model will be used to test all the valid test cases.
2. WorldInvalidMockModel: This mock model will be used to test all the invalid test cases.
The constructor of the mock model accepts 3 parameters:
1. Readable readable
2. StringBuffer log
3. Int uuid

We can use different mocks for failure cases when world or space objects are supposed to throw exceptions.

If the user has entered an invalid move, then the user is asked to enter again the same command.

1. **AttackTarget**

Scenario	**Input**	**Expected Output (in out stream)**
Player attacks with a valid Assume the player has the following items with himself: “Item: Knife, Damage Value: 50”, “Item: Gun, Damage value: 80” and the total health value	-> “Attack target” -> “Knife”	**“**Item used successfully**”** “Please enter a new command”

of the Target is 100.		
Player attacks with an invalid Assume the player has the following items with himself: No items and the total health value of the Target is 100.	-> "Attack target" -> "Knife"	"Please enter a valid item"
Player attacks without any item Assume the player has the following items with himself: No items and the total health value of the Target is 100.	-> "Attack target" -> "poke"	"Item used successfully" "Please enter a new command"
Player attacks with a valid Assume the player has the following items with himself: "Item: Knife, Damage Value: 50", "Item: Gun, Damage value: 80" and the total health value of the Target is 50. The current player is Subhankar	-> "Attack target" -> "Knife"	"Item used successfully" "GAME OVER, Subhankar won the game!"

1. **MovePetCommand**

Scenario	**Input**	**Expected Output (in out stream)**
Moving the pet from Kitchen to Armory, which is a valid space.	-> "Move Pet" -> "Armory"	"Pet moved to the space successfully " "Please enter a new command"
Moving the pet from Kitchen to Kitchen. Note: Since the pet is moved to the same room this is an invalid move	-> "Move Pet" - > "Kitchen"	"Please enter valid space to move" "Please enter a valid space"
Moving the pet from Kitchen to !@$%. The user has entered an invalid room hence this is an invalid move	-> "Move Pet" -> "!@$%"	"Please enter valid space to move"

		"Please enter a valid space"

1. **playGame()**

playGame(ArkhamAsylumWorldImpl model)	**Input(World worldModel)**	**Expected Value**
PlayGame if the model is null	null	IllegalArgumentException("Null exception")

1. **AddPlayerCommand**

addPlayer(String playerName, Integer roomId)	**Input (String playerName, Integer roomId)**	**Expected Value**
Add a valid player name and valid roomId	("Add player","Subhankar", 1)	"Added player Subhankar in the room Kitchen"
Add a player in a room which has player Room: Kitchen, Player: {"Josh", "Tom"}	("Add player", "Subhankar", 1)	"Added player Subhankar in the room Kitchen"
Check if the controller is accepting command after adding a new valid player	("Add player", "Harry", 2)	"Added player Subhankar in the kitchen. Input:"
Check if the controller is accepting input by adding a player which has already been added. Player: {"Josh", "Tom"}	("Add player", "Josh", 1)	"Player already exists. Add player name and space id."
Check if the controller is accepting input by adding a player with invalid string value Player: {"Josh", "Tom"}	("Add player", "!@#$", 1)	"Player name not valid. Add player name and space id"

Check if the controller is accepting input by adding a player with invalid room id Player: {“Josh”, “Tom”}	(“Add player”, “Harry”. -1)	“Room id not valid. Add player name and space id”
Check if the controller is accepting input by adding a player with invalid room id and invalid player name Player: {“Josh”, “Tom”}	(“Add player”, “!@#$”. -1)	“Room id and player name not valid. Add player name and space id”

1. **Move Player Command**

movePlayerCommand(Integer roomId)	**Input(String command, Integer roomId)**	**Expected Output**
Move the current player to the neighbouring room. Player: {“Josh”, “Tom”} Current Player: Name: Josh Room: Kitchen Neighbours:Bathroom(1), Dining Room (2)	(“move”, 2)	“Moved Josh to Dining Room Next turn Tom:”
Move the current player to the neighbouring room. Player: {“Tom”, “Josh”} Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2)	(“move”, 2)	“Moved Tom to Dining Room Next turn Josh:”
Check if the controller is accepting input after Moving the current player to the neighbouring room. Player: {“Tom”, “Josh”}	(“move”, 1)	“Moved Tom to Dining Room Next turn Josh:”

Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2)		
Check if the controller is accepting input after Moving the current player to the invalid neighbouring room. Player: {"Tom", "Josh"} Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2)	("move", -1)	"No neighbours found Next turn Tom:"
Check if the controller is accepting input after Moving the current player to the room which is not a neighbour. Player: {"Tom", "Josh"} Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2)	("move", 10)	"No neighbours found Next turn Tom:"
Check if the controller is accepting input after Moving the current player to the room after the game is over. Player: {"Tom", "Josh"} Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2)	("move", 1)	"Game is over. Josh killed the Joker! Input command:"

1. **LookAroundCommand**

LookAroundCommand()	**Input (String command)**	**Expected Output**
Look Around for the current player. Player: {"Tom", "Josh"}	("Look Around")	"Room: Kitchen Neighbours:Bathroom(1),

Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2) Items: Gun, Stick		Dining Room (2). Items in Room: “Gun”. “Stick” Next turn Josh: “
Check if the controller is still working after looking around the command. Player: {“Tom”, “Josh”} Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2) Items: Gun, Stick	(“Look Around”)	“Room: Kitchen Neighbours:Bathroom(1), Dining Room (2). Items in Room: “Gun”. “Stick” Next turn Josh:“
Check if the controller is still working after looking around the command. Player: {“Tom”, “Josh”} Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2) Items: Gun, Stick	(“Look Around”)	“Room: Kitchen Neighbours:Bathroom(1), Dining Room (2). Items in Room: “Gun”. “Stick” Next turn Josh:“
Check if the controller is still working after looking around the command when the game is over. Player: {“Tom”, “Josh”} Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2) Items: Gun, Stick	(“Look Around”)	“Game is over. Josh killed the Joker! Input command:”

1. **PickUpItemCommand**

PickUpItemCommand()	**Input (String command, String itemName)**	**Expected Output**
Pick up items by the current player. Current Player: Tom Player: {“Tom”, “Josh”}	(“Pick Up Item”, “Gun”)	Gun Picked by Tom. Next turn Josh

Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2) Items: Gun, Stick		
Check if the controller is working after picking up an already picked item by the current player. Current Player: Tom Player: {“Tom”, “Josh”} Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2) Items: Gun, Stick	(“Pick Up Item”, “Gun”)	Gun is already been picked. Next turn Tom
Check if the controller is working after picking up an already picked item by the current player. Current Player: Tom Player: {“Tom”, “Josh”} Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2) Items: Gun, Stick	(“Pick Up Item”, “Gun”)	Gun is already been picked. Next turn Tom
Check if the controller is working after picking up an item which has already been used by the current player. Current Player: Tom Player: {“Tom”, “Josh”} Current Player: Name: Tom Room: Kitchen Neighbours:Bathroom(1), Dining Room (2) Items:Stick	(“Pick Up Item”, “Gun”)	“Gun is no longer available Next turn Tom”

1. **DisplayInfoOfPlayerCommand**

DisplayInfoOfPlayerCommand()	**Input(String command, String playerName)**	**Expected Output**
Display info about a valid specific player. Current Player: Tom Player: {“Tom”, “Josh”}	(“Display player info”, “Josh”)	“Player: Josh, Room: Kitchen. Items: Gun, Sticker. Joker Location: Drawing Room Next turn Tom”
Check if the controller is working after displaying info about a valid specific player after the game is over. Current Player: Tom Player: {“Tom”, “Josh”}	(“Display player info”, “Josh”)	“Player: Josh, Room: Kitchen. Items: Gun, Sticker. Joker is dead Game is over. Start new game?:”
Display info about an invalid player. Current Player: Tom Player: {“Tom”, “Josh”}	(“Display player info”, “Curry”)	“Player Not Found. Next Turn: Tom”
Display info about an invalid string name. Current Player: Tom Player: {“Tom”, “Josh”}	(“Display player info”, “!@#$%”)	“Player Not Found. Next Turn: Tom”

1. **DisplayInfoOfSpecificPlace Command:**

DisplayInfoOfSpecificPlace()	**Input(String command, String placeName)**	**Expected Output**
Display info about a valid specific place. Current Player: Tom Place: {“Kitchen”, “Bathroom”}	(“Display place info”, “Bathroom”)	“Player in the room: “Josh, Tom, Subhankar. Items in the room: Knife, Ball, Bat” Next turn Tom”
Display info about a valid specific place after picking up an item from the place. Picked Bat. Current Player: Tom Place: {“Kitchen”, “Bathroom”}	(“Display place info”, “Bathroom”)	“Player in the room: “Josh, Tom, Subhankar. Items in the room: Knife, Ball” Next turn Tom”
Display info about a valid specific place where the target is present. Current Player: Tom Place: {“Kitchen”, “Bathroom”}	(“Display place info”, “Bathroom”)	“Player in the room: “Josh, Joker. Items in the room: Knife, Ball, Bat” Next turn Tom”
Check if the controller is taking input after display info about a valid specific place after the game is over Current Player: Tom Place: {“Kitchen”, “Bathroom”}	(“Display place info”, “Bathroom”)	“Player in the room: “Josh, Tom, Subhankar. Items in the room: Knife, Ball, Bat” The game is over. Tom killed Joker:”
Check if the controller is taking input after display info about a valid specific place Current Player: Tom Place: {“Kitchen”, “Bathroom”}	(“Display place info”, “Bathroom”)	“Player in the room: “Josh, Tom, Subhankar. Items in the room: Knife, Ball, Bat”

		Next turn Tom:”

1. **AddComputerPlayer Command**

AddComputerPlayer	**Input (String command, String playerName)**	**Expected Output**
Check if the controller is accepting input after adding a computer player before the game is starting	“Add computer player, Android”	“Android added as a computer player. Add another player.”
Check if the controller is accepting input after adding a computer player in the middle of the game	“Add computer player, Android”	“Cannot add a player in the middle of the game. Next turn Tom:”
Check if the controller is accepting input after adding a computer player in the game is over.	“Add computer player, Android”	“Cannot add a player after the game is over”. Game is over. Tom killed Joker: Start a new game:”

1. **Restrict The number of Turn Command**

restrictTurn()	**Input**	**Expected**
Check if the controller is accepting input after the maximum number of turn reached	(“Move”, 2)	“Maximum number of turns reached. Next turn: Josh”

Test cases for Player.java class

```
public Player(ArrayList<Strings>[0..3] itemArrayList, String playerName, Integer playerPosition) {
 //...
}
```

Testing construction	Inputs	Expected Value
Player with the valid given name items List and position. Rooms: Kitchen(1), Bathroom(2), Dining Room (3)	((“Gun”, “Knife”), “Josh”, 1)	“Player: Josh, Room: Kitchen. Items: Gun, Sticker.”
Player with the valid given name items List and same position. Rooms: Kitchen(1) ,Bathroom(2), Dining Room (3)	((“RPG”, “Stick”), “Tom”, 1)	“Player: Tom, Room: Kitchen. Items: RPG, Stick.”
Player with the empty items List and position. Rooms: Kitchen(1) ,Bathroom(2), Dining Room (3)	((“”), “Tom”, 1)	“Player: Tom, Room: Kitchen. Items:”
Player with the same given name items List and position. Rooms: Kitchen(1), Bathroom(2), Dining Room (3)	((“Gun”, “Knife”), “Josh”, 1)	IllegalArgumentException(“Player already exists”)
Player with the given name items List and invalid position. Rooms: Kitchen(1), Bathroom(2), Dining Room (3)	((“Gun”, “Knife”), “Josh”, -1)	IllegalArgumentException(“Location does not exists.”)
Player with the given name items list and valid position, but invalid Name.	((“Gun”, “Knife”), “!@#$”, -1)	IllegalArgumentException(“Player name does not exists.”)

Rooms: Kitchen(1), Bathroom(2), Dining Room (3)		

Test addPlayer(String nameOfPlayer, Integer roomId)

Testing addPlayer(String playerName, Integer roomId)	**Inputs (String nameOfPlayer, Integer roomId)**	**Expected Value**
For a valid player name and valid room number.	(“Subhankar”, 0)	true
For a valid player name and the same room number.	(“Tom”, 0)	true
For a valid player name and the different room number.	(“Cat”, 1)	true
For the same player name and the different room number.	(“Cat”, 1)	IllegalArgumentException(“Player is already added”)
For a valid player name and the different room number.	(“Cat”, 1)	IllegalArgumentException(“Player is already added”)
For an invalid string character	(“!@#$”, 1)	IllegalArgumentException(“Player name should valid”)
For an integer value in the name of the player	(123, 1)	IllegalArgumentException(“Player name should valid”)
For an empty value in the name of the player	(“”, 1)	IllegalArgumentException(“Player name should valid”)
For invalid room id but valid name	(“Subhankar”, -1)	IllegalArgumentException(“Room id should be value”)
For room which does not exists but valid string	(“Subhankar”, 100)	IllegalArgumentException(“Room id should be value”)

Test addComputerPlayer()

Testing addComputerPlayer()	Input	Expected Value
Invokes the function	void	No.of player = 1(increased by 1)

Test movePlayer(Integer roomId)

Testing movePlayer(Integer roomId)	Input (Integer roomId)	Expected Value
Move to valid space id	(2)	true
Move to valid space id	(3)	true
Move from last space id to first space id. If the player is 20.	(1)	true
Move to invalid space id	(-100)	IllegalArgumentException("Space id should be valid")
Move to space id which is out of bound	(100)	IllegalArgumentException("Space id should be valid")
Move to space that is not the neighbour. If the player is in room 1.	(20)	IllegalArgumentException("Cannot move to non neighbours")
Move to space id in which the player is already in	(2)	IllegalArgumentException("Player is already in the space")

Test pickItem()

Testing pickItem(String itemName)	Input (String itemName)	Expected Value
Pick an item in the current space	("Gun")	true
Pic an item in the room which has multiple rooms	("Broom")	true
Pick an item in the room. Before reaching the maximum limit	("stick")	true

Pick an item in the room after reaching maximum value	(“stone”)	IllegalArgumentException(“Maximum number of items reached”)
Pick an item which is in the other room	(“Knife”)	IllegalArgumentException(“Item should be in the same player”)
Pick an item which is already picked	(“Gun”)	IllegalArgumentException(“The item is already picked.”)
Pick an item which is not available	(“Bazzuca”)	IllegalArgumentException(“The item is not available”.)

Test displayDescription(String playerName)

Testing displayDescription(String playerName)	**Input (String playerName)**	**Expected Value**
Display for a valid player with 2 items	(“Subhankar”)	“Location: Kitchen, Items: Knife, Spoon”
Display for a valid player with 0 items	(“Subhankar”)	“Location: Dining Room, items:”
Display for a valid player with maximum items	(“Subhankar”)	“Location: Dining Room, Items Knife, spoon, rock, gun”
Display for an invalid player String	(“!@#$”)	IllegalArgumentException(“Player name should be valid”.)
Display for a computer controlled player	(“Josh”)	“Location: Dining Room, Items Knife, spoon, rock, gun”

Test lookAround(“String playerName”)

Test LookAround(String playerName)	**Input (String playerName)**	**Expected Value**
Look around for a valid player with 1 neighbour.	(“Josh”)	“Location: Dining Room”

Look around for a valid player with 2 neighbours.	(“Tom”)	“Location: Dining Room, Kitchen”
Look around for a valid player with 0 neighbours.	(“Cat”)	“”
Look around for an invalid player	(“!@#$%”)	IllegalArgumentException(“Player name should be valid”.)

Test getTurn()

getTurn()	**Input**	**Expected Value**
Get turn for 5 players in the list (“Tom”, “Josh”, “Rick”, “Call”)	getTurn()	“Tom”
Get turn for 4 players in the list (“Josh”, “Rick”, “Call”,“Tom”)	getTurn()	“Josh”
Get turn for 1 players (“Josh”)	getTurn()	“Josh”
Get turn for 0 players	getTurn()	“No players added yet”

Test cases for Target.java class

Constructor of the StevensonReading class:

```
public Target(ArrayList<Integer>[0..3] targetPosition, Integer targetHealth) {
 //...
}
```

Testing construction	**Inputs**	**Expected Value**
Target with a given location and health	((22, 19, 23, 26), 100)	“Target location(30,20,5,15)”
Target with a given location and health	((16 21 21 28),80)	“Target Location(16,21,21,28)”
Target with a given location and negative health	((16 21 21 28), -10)	IllegalArgumentException
Target with a given negative location and negative health	((-30,20,5,15), 100)	IllegalArgumentException
Target with a given negative location and negative health	((-30,-20,-5,-15), 100)	IllegalArgumentException

Target with a given location and health as zero	((22, 19, 23, 26), 0)	IllegalArgumentException
Target with an Invalid given location and valid health	((22, 19, 23, Integer.MAX), 100)	IllegalArgumentException
Target with given location and invalid health	((22, 19, 23, 26), Integer.MAX)	IllegalArgumentException
Target with given location and given health more than total weapon damage	((22, 19, 23, 26), 1000)	IllegalArgumentException
Target with given location which is out of bound of board and given health more than total weapon damage	((330,30,15,0), 10)	IllegalArgumentException

Testing getCharacterType()	**Inputs**	**Expected Value**
For valid position and health		CharacterType.Target

Testing getCharacterHealth()	**Inputs**	**Expected Value**
For valid position and health	((330,30,15,0), 10)	10
For valid position and health	((22, 19, 23, 26), 100)	100

Testing getCharacterPosition()	**Inputs**	**Expected Value**
For valid position and health	((22, 19, 23, 26), 100)	1
After Moving the character to next highest int	((16,21,21,28), 100)	2

Testing getCharacterPosition()	**Inputs**	**Expected Value**
For valid position and health	((22, 19, 23, 26), 100)	1
After Moving the character to next highest int	((16,21,21,28), 100)	2

Testing moveToNexSpace()	Inputs	Expected Value
Move from current room to next subsequent highest int	1	2
Move from current room to next subsequent highest int	50	51
If at the highest space number	100	100

Testing decreaseTargetHealth()	Inputs	Expected Value
Decrease target health from current health 100	10	90
Decrease target health from current health 1000	900	100
Decrease target health from current health 100	100	0
Decrease target health from current health 100	200	0

Test class for Weapons.java class

```
public Weapon(String weaponName, Integer damage Value, List<Integer>[0..3] ItemRoomLocation) {
 //...
}
```

Testing construction	Inputs	Expected Value
Weapon with valid damage value and location	(“Knife”, 20, (22, 19, 23, 26))	“Knife, 20, 1”
Weapon with valid big damage value and location	(“Fork”, 100, (22, 19, 23, 26))	“Fork, 100, 1”
Weapon with invalid damage value	(“Screw”, -100, (22, 19, 23, 26))	IllegalArgumentException

Weapon with invalid room location	(“Spoon”, 100, (-22, 19, 23, 26))	IllegalArgumentException
Weapon with invalid room location	(“Broom”, 100, (22, -19, 23, 26))	IllegalArgumentException
Weapon with invalid room location	(“Lemon”, 100, (22, 19, -23, 26))	IllegalArgumentException
Weapon with invalid room location	(“Gun”, 100, (22, 19, 23, -26))	IllegalArgumentException
Weapon with invalid room location coordinates	(“Axe”, 100, (10000, 19, 23, 26))	IllegalArgumentException

Weapon with invalid room location coordinates	(“Blade”, 100, (10, 1900, 23, 26))	IllegalArgumentException
Weapon with invalid room location coordinates	(“Sword”, 100, (10, 19, 2300, 26))	IllegalArgumentException
Weapon with invalid room location coordinates	(“Machine Gun”, 100, (10, 19, 23, 2600))	IllegalArgumentException

Testing getItemType()	**Inputs**	**Expected Value**
Weapons with given name and location coordinates	(“Fork”, 100, (10, 19, 2300, 26))	(“Fork”, 1)
Weapons with given name and location coordinates	(“Knife”, 10, (22, 19, 23, 26))	(“Fork, 2)

Testing getDamageValue()	**Inputs**	**Expected Value**
Damage value when values are given in constructor	(“Fork”, 100, (10, 19, 2300, 26))	100
Damage value when values are given in constructor	(“Knife”, 10, (22, 19, 23, 26))	10

Testing getItemRoomLocation()	**Inputs**	**Expected Value**

Get room item id for valid item and valid location	(“Fork”, 100, (10, 19, 2300, 26))	1
Get room item id for valid item and valid location	(“Knife”, 10, (22, 19, 23, 26))	2

Test Room

```
public Room(Integer roomId, List<Integer> roomPosition ) {
 //...
}
```

Testing construction	**Inputs**	**Expected Value**
Room with valid with position id and room position	(1, (22, 19, 23, 26))	“Room location(22, 19, 23, 26)”
Room with valid with position id and room position	(2, (24, 20, 24, 30))	“Room Location(24, 20, 24, 30)”
Room with invalid with position id and room position	(3, (-24, 20, 24, 30))	IllegalArgumentException
Room with invalid with position id and room position	(3, (24, -20, 24, 30))	IllegalArgumentException
Room with invalid with position id and room position	(3, (24, 20, -24, 30))	IllegalArgumentException
Room with invalid with position id and room position	(3, (24, 20, 24, -30))	IllegalArgumentException
Room with invalid with position id and room position	(Integer.MAX, (24, 20, 24, 30))	IllegalArgumentException

Testing getSpaceId()	**Inputs**	**Expected Value**
Room with valid with position id and room position	(2, (24, 20, 24, 30))	2
Room with valid with position id and room position	(3, (1, 5, 24, 30))	3

Testing getSpaceLocation()	**Inputs**	**Expected Value**

Room with valid with position id and room position	(2, (24, 20, 24, 30))	(24, 20, 24, 30)
Room with valid with position id and room position	(3, (1, 5, 24, 30))	(1, 5, 24, 30

Testing getNeighnoringSpanceLocation()	**Inputs**	**Expected Value**
Room with valid with position id and room position	(2, (24, 20, 24, 30))	(24, 20, 24, 30) , (1, 5, 24, 30)
Room with valid with position id and room position	(3, (1, 5, 24, 30))	(1, 5, 12, 32), (24, 20, 24, 30)

Testing getNeighnoringSpanceLocation()	**Inputs**	**Expected Value**
Room with valid with position id and room position	(2, (24, 20, 24, 30))	(24, 20, 24, 30) , (1, 5, 24, 30)
Room with valid with position id and room position	(3, (1, 5, 24, 30))	(1, 5, 12, 32), (24, 20, 24, 30)

Testing ifAnyNeighbours()	**Inputs**	**Expected Value**
Room with valid with position id and room position	(2, (24, 20, 24, 30))	True
Room with valid with position id and room position	(3, (1, 5, 24, 30))	False

Test ArkhamAsylumWorld

```
public ArkhamAsylumWorld( List<Space> spaceList, Target targetCharacter, List<Items> itemList) {
 //...
}
```

Testing construction	**Inputs**	**Expected Value**
Room with valid Space location, Target character, and item List	((1, (22, 19, 23, 26),), (2, (24, 20, 24, 30)), (3, (21, 22, 40, 22))))	"DrLuckyWorld((1, (22, 19, 23, 26),), (2, (24, 20, 24, 30)), (3, (21, 22, 40, 22)))))"
Room with valid Space location, Target character, and item List	((2, (22, 19, 23, 26),), (2, (24, 20, 24, 30)), (3, (21, 22, 40, 22))))	"DrLuckyWorld((2, (22, 19, 23, 26),), (2, (24, 20, 24, 30)), (3, (21, 22, 40, 22))))"

Room with Invalid Space location, Target character, and item List	((-2, (22, 19, 23, 26),), (2, (24, 20, 24, 30)), (3, (21, 22, 40, 22))))	IllegalArgumentException
Room with Invalid Space location, Target character, and item List	((2, (-22, 19, 23, 26),), (2, (24, 20, 24, 30)), (3, (21, 22, 40, 22))))	IllegalArgumentException
Room with Invalid Space location, Target character, and item List	((2, (22, -19, 23, 26),), (2, (24, 20, 24, 30)), (3, (21, 22, 40, 22))))	IllegalArgumentException
Room with Invalid Space location, Target character, and item List	((2, (22, 19, -23, 26),), (2, (24, 20, 24, 30)), (3, (21, 22, 40, 22))))	IllegalArgumentException
Room with Invalid Space location, Target character, and item List	((2, (22, 19, -23, 26)) (2, (24, 20, 24, 30)), (3, (21, 22, 40, 22))))	IllegalArgumentException
Room with Invalid Space location, Target character, and item List	((2, (22, 19, 23, -26)) (2, (24, 20, 24, 30)), (3, (21, 22, 40, 22))))	IllegalArgumentException
Room with Invalid Space location, Target character, and item List	((2, (22, 19, 23,6)) (-2, (24, 20, 24, 30)), (3, (21, 22, 40, 22))))	IllegalArgumentException
Room with Invalid Space location, Target character, and item List	((2, (22, 19, 23,6)) (2, (24, 20, 24, 30)), (-3, (21, 22, 40, 22))))	IllegalArgumentException