CPSC 2720 FALL 2021

Life2.0-Logout

Joker

Albert Arrieta, Ayobami Abejide, Hana Kim, Kurt Clarke

2021-10-27

TABLE OF CONTENTS

INTRODUCTION

Simple Version

1) You are a beta tester for a new CAPTCHA system involving terminal commands.
2) This CAPTCHA system involves you passing several tests that will test out how human you are.
3) This is implemented as a text-based adventure where the CAPTCHAs are puzzle rooms where you will have to interact with other characters.
4) If you pass enough of the tests then you win and are allowed to logout of the game, or you will be able to after we fully release the new system improvements.
5) Your questions, comments, and concerns about the new improvements are greatly appreciated!

Fun Version

Dear valued Life2.0 User,

We have recently discovered an exploit using our new terminal interface which viruses have been exploiting to force user logouts. We deeply apologize to those that have been affected via forced logout or by a sense of surreality. We have done our best to maintain a consistent environment, but we understand some of the recent server events have disrupted normal gameplay, we hope things will soon be back to normal.

Here at Joker Industries, we take pride in our reputation for quality and unrelenting professionalism. To prevent further attacks, we are implementing a set of sophisticated CAPTCHA's so that you and only you can decide when your Life2.0 experience ends. Based on the latest research by Nigma et al. (1948) we have decided to use the decisions and errors humans make during text-based adventure games as a starting point for the new system.

You dear user have been selected as a beta-tester for this text-based adventure CAPTCHA system. We would appreciate you trying out the new logout system to provide us with important user data and feedback for us to make further improvements. When you are ready, please run Life2.0-Logout from your nearest terminal, pass the CAPTCHAs, and then provide us any feedback and recommendations on your experience.

We thank you for you for playing Life2.0 and hope you continue to use our services in and out of game. Afterall no one knows you better than Joker Industries, every breath, every move, and every thought has been recorded to enhance your gameplay experience in this world of our making.

Joker Industries, truth through lies.

Sincerely,

Jack M. Napier
Head of Customer Relations
Joker Industries
10534 – Barnaby North, Gotham

PROJECT MANAGEMENT

TEAM ROLES

Team Member	**Design - Draft**	**Design – Final**	**Implementation - Basic**	**Implementation - Final**
Albert	QA Lead	Phase Lead	Design Lead	Reporting Lead
Ayobami	Reporting Lead	QA Lead	Phase Lead	Design Lead
Hana	Design Lead	Reporting Lead	QA Lead	Phase Lead
Kurt	Phase Lead	Design Lead	Reporting Lead	QA Lead

TEAM ROLE RESPONSIBILITIES

Phase Lead:

- Understands what needs to be accomplished in the phase.
- Asks questions/clarification from instructor(s) on behalf of the group.
- Identifies (in collaboration with the team) tasks to be done and distributes them.
- Follows up with team members about assigned tasks.
- Identify problems and lead group discussions about how to solve them.
- Makes sure the team meets deadlines.

Design Lead:

- Propose and collect design ideas from the group.
- Assure (in collaboration with the team) that the software design is following
- SOLID+DRY principles.
- Create or maintain the UML diagrams so they reflect the current design of the software.

Quality Assurance Lead:

- Ensure that the team is making a quality software product.
- Review pull requests to the master branch to ensure that they meet the team's
- quality standards.
- Creates a testing plan / schedule.
- Identifies test cases.

Reporting Lead:

- Organizes the work to be done for the report(s).
- Collects team's contributions to the reports (e.g. design diagrams from Design Lead).
- Records the team's ideas / action items during meetings.
- Writes (in collaboration with the team) the team report.

RISK MANAGEMENT

REQUIREMENTS/DESIGN/ESTIMATION

- We have prioritized classes during implementation in this order:
  1. Entity class

2. Room class
3. Character/NPC class
4. Player class
5. Item class
6. Inventory
7. Interactions class
8. Interaction Subclasses
9. Input and Game classes

- Our team will set soft deadlines during a week and keep track of progress. Our priority is to make sure all the classes work with each other first and then develop more of the plot and story depending on how much time we have left to implement the whole game to ensure that we can deliver a complete game. Entity is prioritized over interactions as in the worst case if we have the entities complete and functioning, we can create subclasses with the specific interactions rather than trying to create an abstraction of interactions.

PEOPLE

- We will use the prioritized development list above to reduce workload if a team member quits or became unavailable due to a major life event or unexpected commitment.
- If a team-member lacks expected technical skills, in the short term they will be assigned to less technical tasks to make sure they can still effectively contribute to the project.
- If a team member is late for a meeting, the meeting will start without them, and we will give a short summary of what the late member has missed. If the member is late for more than 30 minutes of the meeting or absent, we will continue with the meeting and the late group member will be responsible for attaining any information that they might have missed during the meeting.
- If the team member is not meeting deadlines, the phase lead will have to speak with them individually to why this has happened. If deadlines have been missed more than 3 times, the whole team will need to talk to the individual and discuss how they can make up for the missing work.
- If member is unresponsive for one week, the group will meet with Dr. Anvik and let him know that we will be no longer assigning tasks to the individual.

LEARNING & TOOLS

- Before choosing a tool not taught in the labs/class nor recommended by Dr. Anvik an assessor will be chosen to do a quick study of the advantages and disadvantages of using the tool, they will also be responsible for teaching the basics of the tools use to the rest of the team if they decide to recommend the tool.
- If the tool, is one taught in the lab that we do not have sufficient mastery to use, we will make an appointment with the lab instructor to ask for help.
- If the tools we have chosen, do not work together we will meet to decide which tool to keep using and assign a member to assess other options for the other tool.

DEVELOPMENT PROCESS

CODE REVIEW PROCESS

- Team members will email proposals or send proposals through discord for code design changes to the design leader. The design leader will review all proposed changes and prepare a recommendation for the next meeting.
- All trivial changes, changes that are required to bring code into line with the design plan, or substantial changes that have been previously discussed at a team meeting will be approved by the team lead.
- Any changes that are contrary to the design plan and/or will have undesirable consequences will be rejected and added to the list of items to be discussed at the next team meeting.

COMMUNICATION TOOLS

We are using discord to communicate during the final testing phase, group members and any other recruited testers will send bug reports via the GitLab issue tracker.

CHANGE MANAGEMENT

It is something that should not happen, but what if one of the group members gets hit by a bus or arrested by the police? We must prepare for this situation. Each member has a role in each phase. However, the absence of one member should not stop or defer the project. If one member is missing, then another member who took missing person’s role in the previous phase will take charge of it. Other members also need to back up for missing parts. We have solid documentation (project design report) and everyone is following the process from UML and sequence diagram. Also, we are using version control and sharing our work daily base. Therefore, role changing should not create a critical issue in our project.

If the missing member have responsibility of code implementation on certain class, the phase leader will assign the workload to another member who nearly finishes own work. The rest of members should also back up for the missing parts. Filling the hole can be effectively done because all the other member has the most updated version of the missing person’s work through GitLab.

SOFTWARE DESIGN

DESIGN COMPONENTS

The Life2.0-Logout game is designed as a set of components

1. A set of Characters – to be defined in the game based on implementation. The implementation is intended to be designed such that these do not need to be specific sub classes of NPC/Char. This may change if the above is unrealistic.
2. A set of Rooms – to be defined and changed in game based on the implementation. Like characters above this may have to change if this flexibility is a problem.
3. A set of interactions unique to the characters and rooms.
4. A total of 17 rooms are planned. These rooms will be divided into 4 sets of 4 which will each be a set of puzzles and a main room which will include the final puzzle. Each team member is to implement an individual puzzle and record the results in the players stats.
5. Note Items are intended to be interactable through the objects that contain them that is either a room or a character has an object and the presence of this can be required for a character or object interaction.

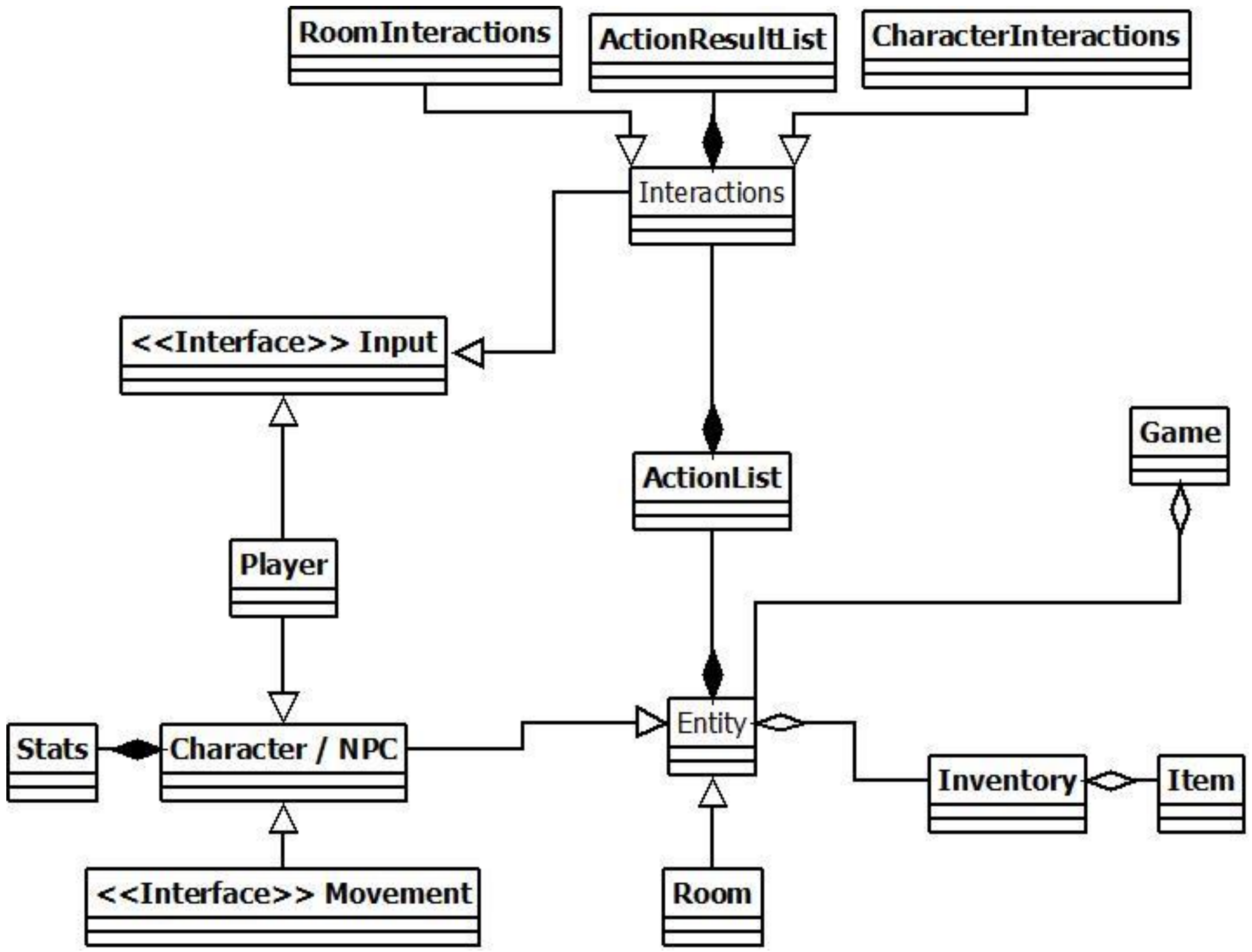
DESIGN RATIONALE

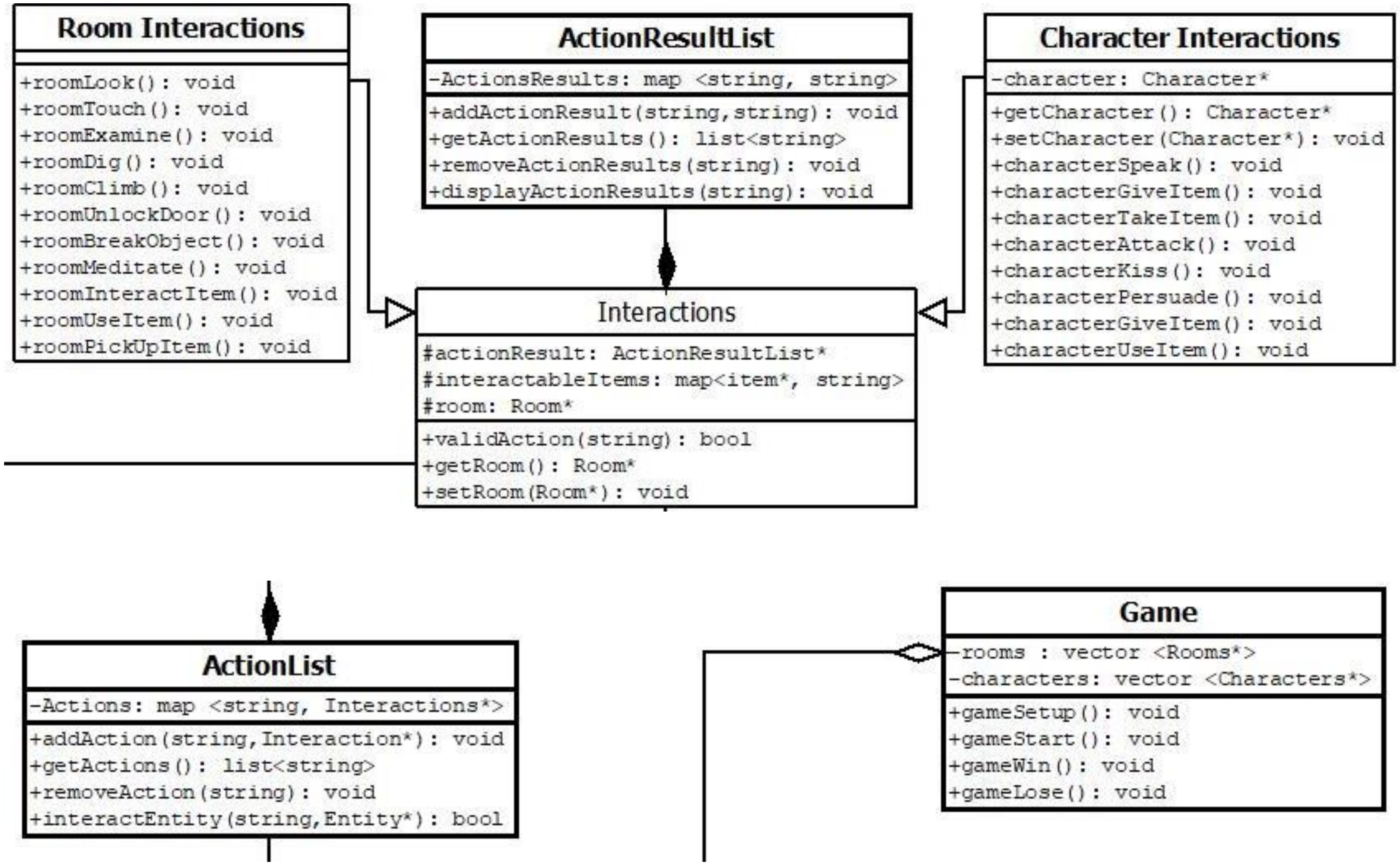
The objective of this project, which informed all design decisions, is to develop examples that illustrate the principles taught in CPSC2720. To this end we are using SOLID, the DRY principal, Agile development techniques, and test-driven development. We are starting by implementing the first three of the above components mentioned independently of each other as they can easily be tested using unit tests. The actual puzzles will require integration tests and be dependent on the first three components working correctly.

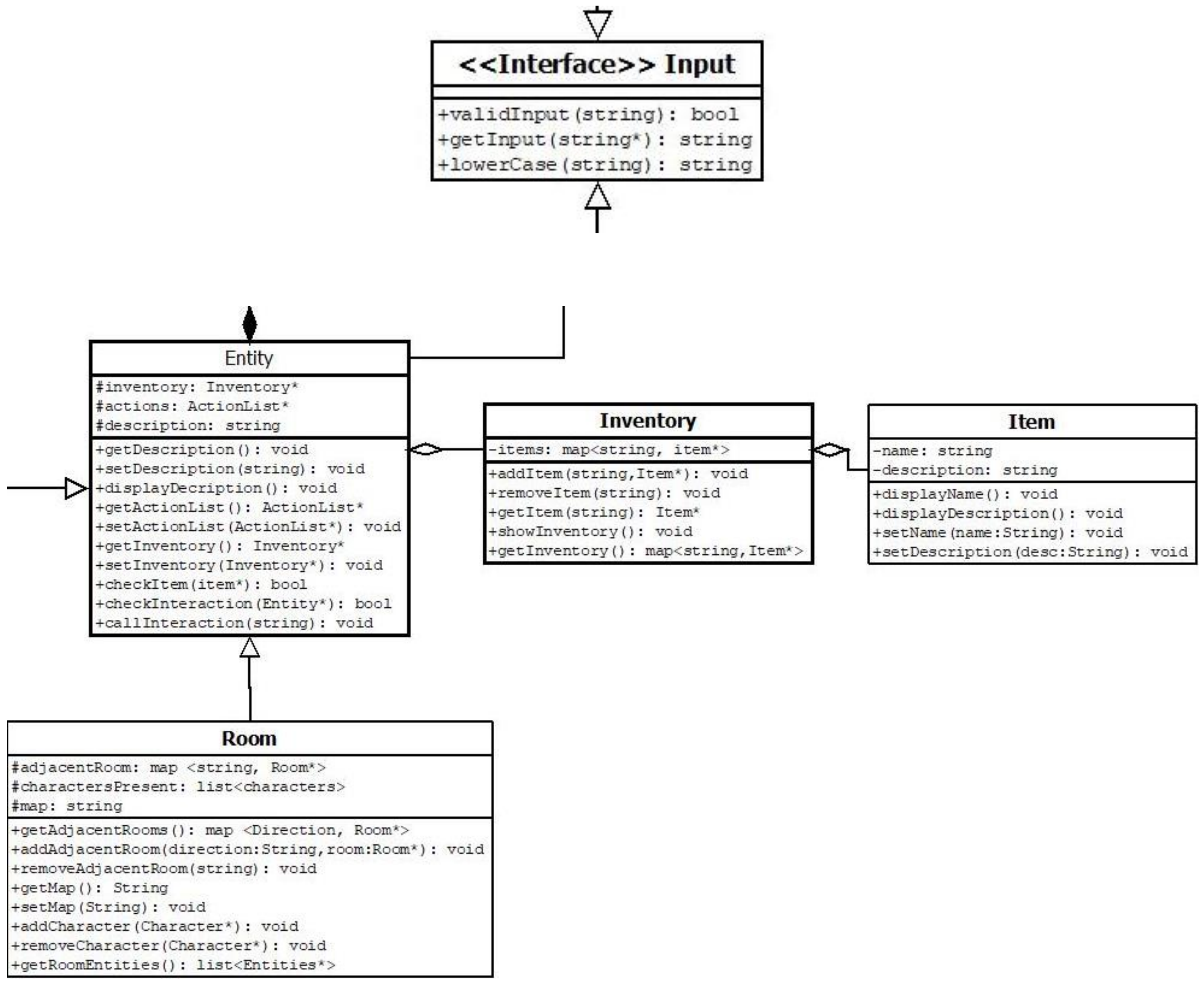
We have attempted to follow the design principles we have covered in class so far including DRY and SOLID. We expect that some level of refactoring will have to be done and which may introduce new design principles as we learn them and during our AGILE leaning development process. The classes currently are intended to have a single responsibility and if this is extended during implementation, we can add additional specialized classes as needed. The current intent is that entities are containers that can be specified in the program rather than requiring specific sub classes though this may ultimately be impractical. If we run low on time, we will implement individual characters and puzzle rooms as subclasses of their respective super classes to simplify implementation but lose flexibility in the program.

The specific outcomes of the design decision can be found in the Design section below.

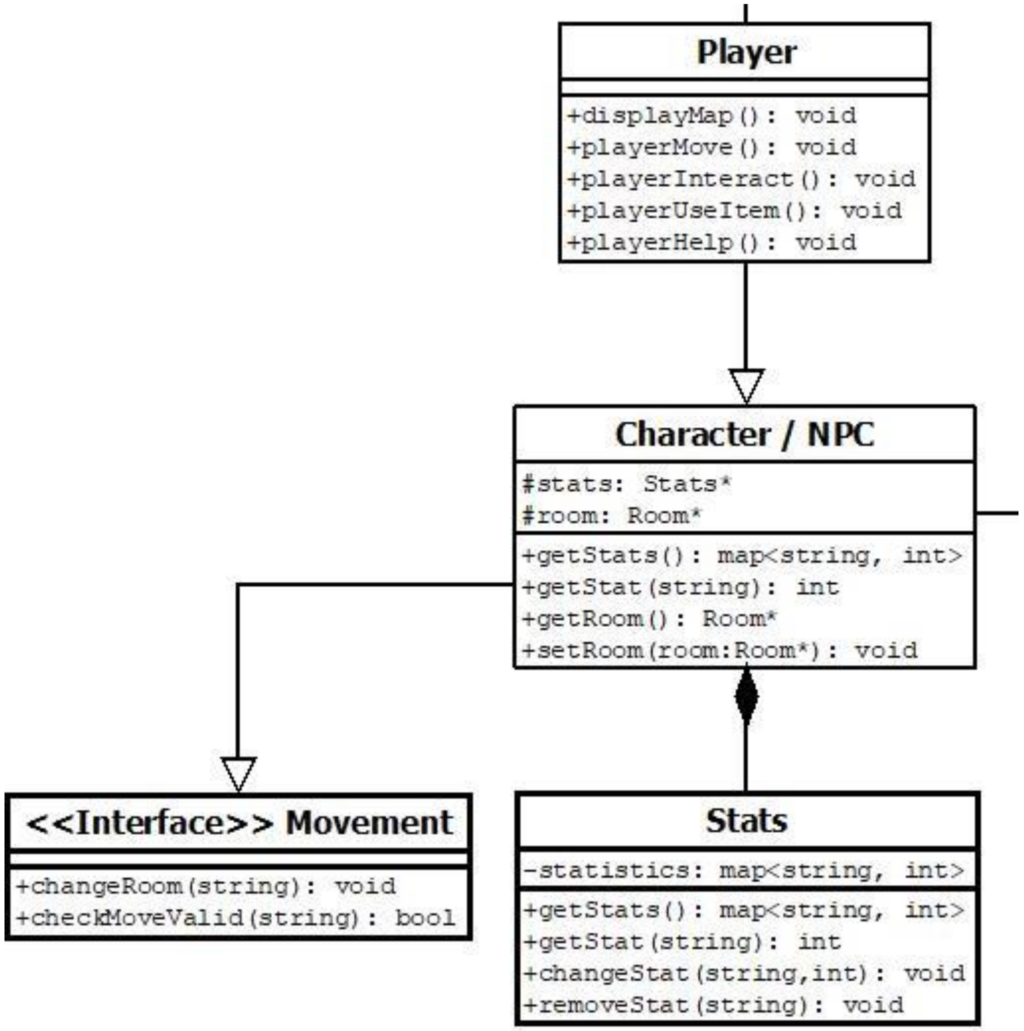
DESIGN – CLASS DIAGRAMS




Room Interactions
+roomLook(): void
+roomTouch(): void
+roomExamine(): void
+roomDig(): void
+roomClimb(): void
+roomUnlockDoor(): void
+roomBreakObject(): void
+roomMeditate(): void
+roomInteractItem(): void
+roomUseItem(): void
+roomPickUpItem(): void
ActionResultList
-ActionsResults: map <string, string>
+addActionResult(string,string): void
+getActionResults(): list<string>
+removeActionResults(string): void
+displayActionResults(string): void
Character Interactions
-character: Character*
+getCharacter(): Character*
+setCharacter(Character*): void
+characterSpeak(): void
+characterGiveItem(): void
+characterTakeItem(): void
+characterAttack(): void
+characterKiss(): void
+characterPersuade(): void
+characterGiveItem(): void
+characterUseItem(): void
Interactions
#actionResult: ActionResultList*
#interactableItems: map<item*, string>
#room: Room*
+validAction(string): bool
+getRoom(): Room*
+setRoom(Room*): void
ActionList
-Actions: map <string, Interactions*>
+addAction(string,Interaction*): void
+getActions(): list<string>
+removeAction(string): void
+interactEntity(string,Entity*): bool
Game
-rooms : vector <Rooms*>
-characters: vector <Characters*>
+gameSetup(): void
+gameStart(): void
+gameWin(): void
+gameLose(): void

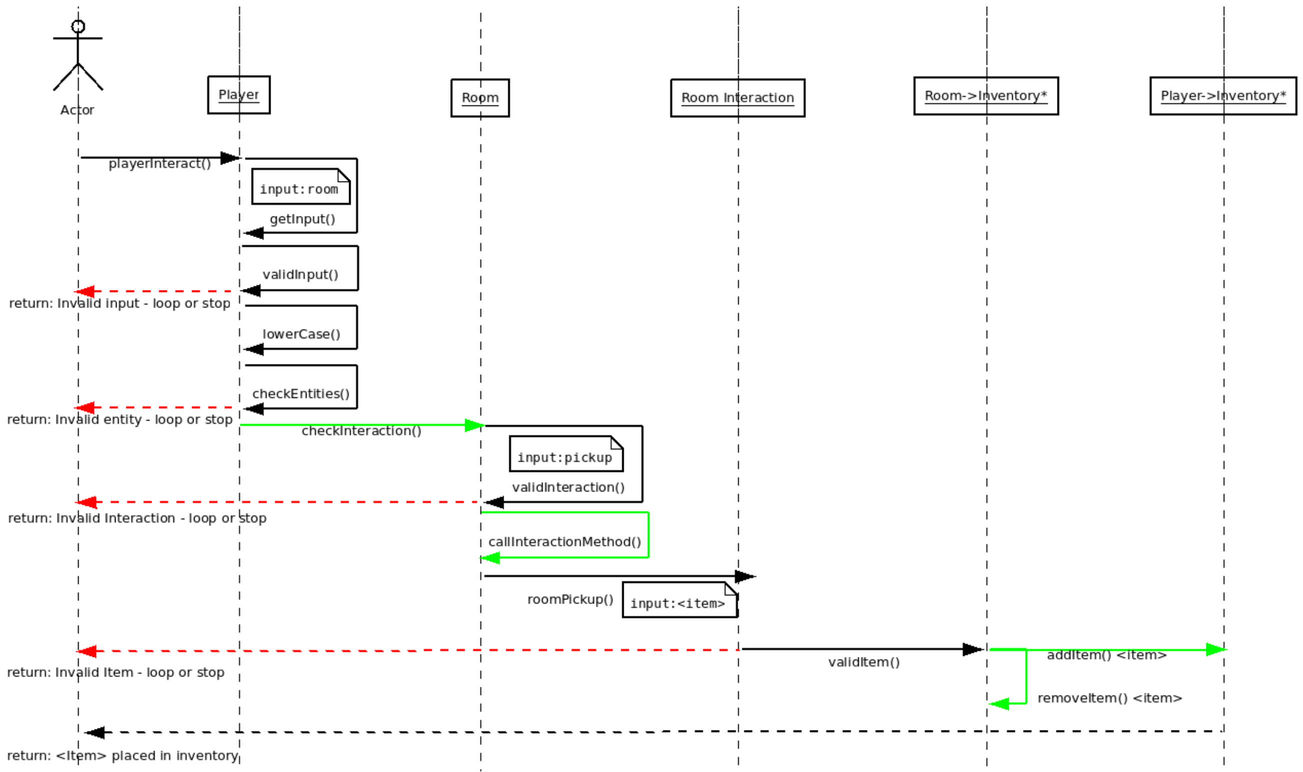


<<Interface>> Input
+validInput(string): bool
+getInput(string*): string
+lowerCase(string): string
Entity
#inventory: Inventory*
#actions: ActionList*
#description: string
+getDescription(): void
+setDescription(string): void
+displayDecription(): void
+getActionList(): ActionList*
+setActionList(ActionList*): void
+getInventory(): Inventory*
+setInventory(Inventory*): void
+checkItem(item*): bool
+checkInteraction(Entity*): bool
+callInteraction(string): void
Inventory
-items: map<string, item*>
+addItem(string,Item*): void
+removeItem(string): void
+getItem(string): Item*
+showInventory(): void
+getInventory(): map<string,Item*>
Item
-name: string
-description: string
+displayName(): void
+displayDescription(): void
+setName(name:String): void
+setDescription(desc:String): void
Room
#adjacentRoom: map <string, Room*>
#charactersPresent: list<characters>
#map: string
+getAdjacentRooms(): map <Direction, Room*>
+addAdjacentRoom(direction:String,room:Room*): void
+removeAdjacentRoom(string): void
+getMap(): String
+setMap(String): void
+addCharacter(Character*): void
+removeCharacter(Character*): void
+getRoomEntities(): list<Entities*>

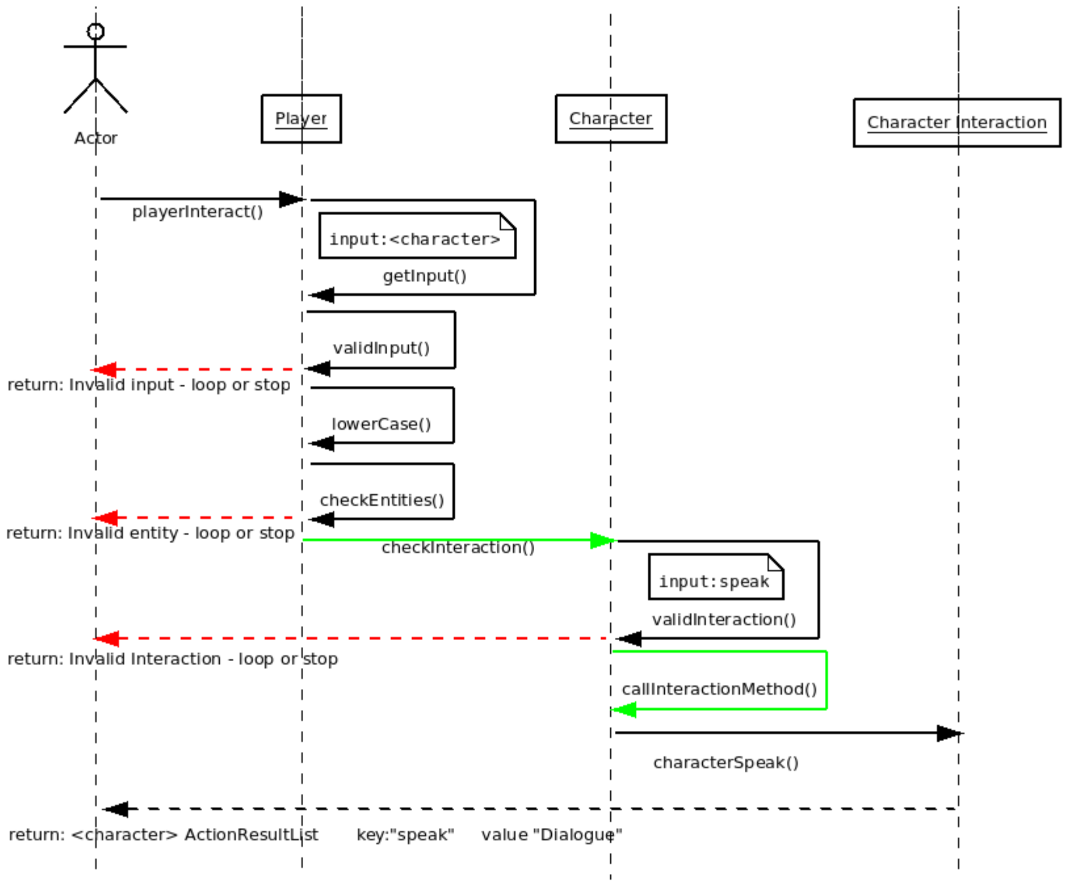

Player
+displayMap(): void
+playerMove(): void
+playerInteract(): void
+playerUseItem(): void
+playerHelp(): void
Character / NPC
#stats: Stats*
#room: Room*
+getStats(): map<string, int>
+getStat(string): int
+getRoom(): Room*
+setRoom(room:Room*): void
<<Interface>> Movement
+changeRoom(string): void
+checkMoveValid(string): bool
Stats
-statistics: map<string, int>
+getStats(): map<string, int>
+getStat(string): int
+changeStat(string,int): void
+removeStat(string): void

DESIGN – SEQUENCE DIAGRAMS

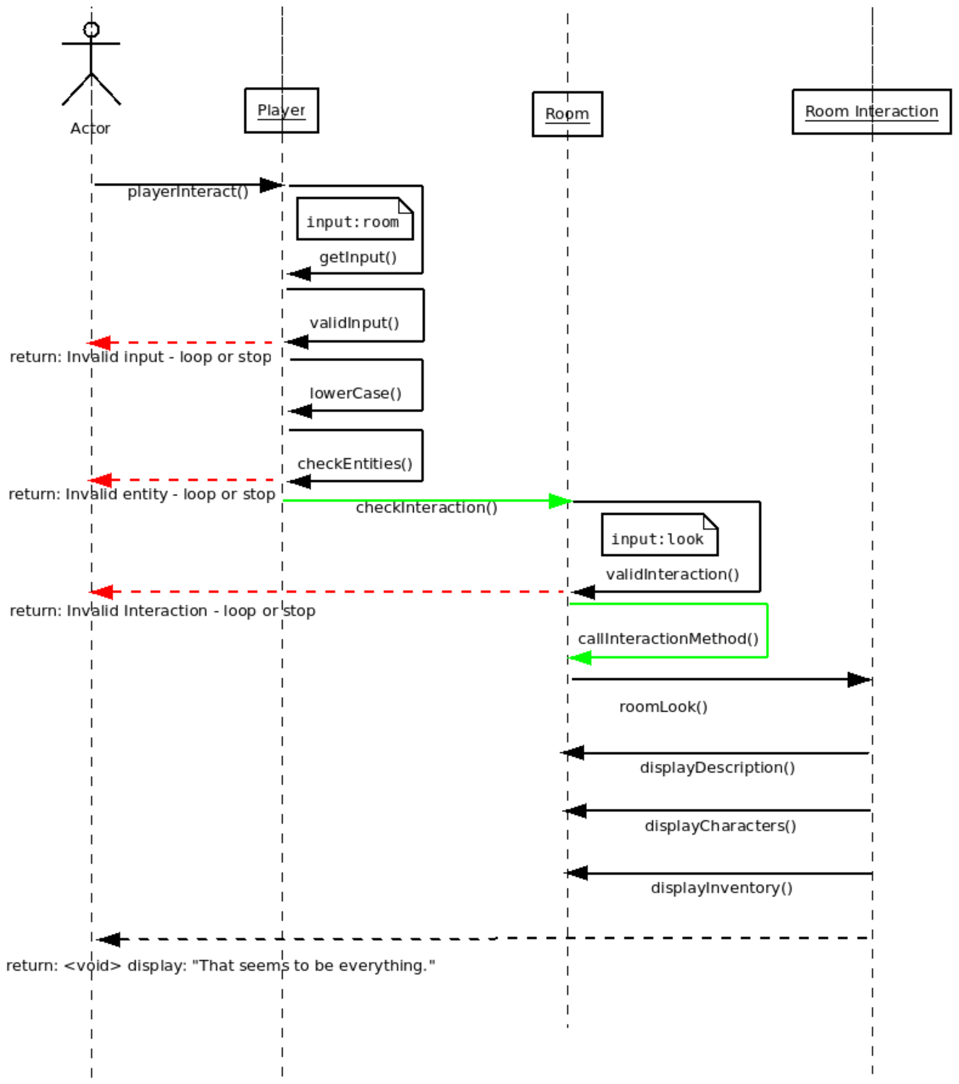
Player pick up an item:



Player Speak with Another Character



Player Look around a Room



CLASS DESCRIPTIONS

NOTE: functions covered by super classes are not included for brevity.

ENTITY – A general container class which contains attributes that will be manipulated by Interactions to move the game forward and keep track of changes / player progress.

getDescription() – returns a description of the entity.

setDescription – changes the description of the entity.

Displaydescription – Display an entities description.

getActionList() – returns a list of available actions for that entity.

setActionList() - Assigns an action list to an entity.

checkInteraction() – tests if an interaction is valid for an entity.

callInteraction() - Initiates an interaction in the action list based on a string input.

checkForItem – checks if an entity has an item in its inventory.

ROOM – A specialized container for room specific attributes.

getAdjacentRooms() – returns adjacent rooms.

addAdjacentRoom() – adds a new room pointer to the map of rooms.

removeAdjacentRoom() – removes a room pointer from the map of rooms.

getMap() – returns a string which acts as a map.

setMap() - sets a rooms map.

addCharacter() – adds a character to the list of characters in a room.

removeCharacter() – removes a character from the list of characters in a room.

getRoomEntities() - gets a list of all the entities in a room.

CHARACTER / NPC – A specialized container for Character/NPC actions for an entity modified by interactions to change the character and move the story forward.

getStats() – returns a map<string, int> of a characters stat.

getStat() – returns the value of a specific stat.

getRoom() – returns a pointer to the room that a character is currently in.

setRoom() – sets the room a character is in.

PLAYER – A specialized container to keep track of all the Player characters information. Contains calls to Player specific actions.

displayMap() - displays the map stored in the room where the player is located.

playerMove() – moves the player to a different room depending on their input and which room they are currently in.

playerInteract() – Asks the player what they would like to interact with and what interaction they would like to perform based on the room they are in and the other characters also in that room (this may include actions self-referential to the players action map).

playerUseItem() – Asks the player what item they would like to use and on what. Interaction will then check if this is a valid interaction and implement any changes that happen because of the interaction.

playerHelp() – Calls player help. This is dependent on the room the player is in and the other characters and items in that room.

INPUT – Used to provide functions for extracting inputs and formatting inputs to prevent errors and simplify the implementations of functions that use inputs.

validInput() – checks that an input is of an appropriate length and character set (this includes ending the input when a space is detected).

getInput() – Asks the user for input and then modifies a pointer with that new value.

lowercase(string) - turns all upper-case letters to lowercase to simplify input matching.

ITEM – a basic unit of inventory used for interaction matching and to provide players with description hints.

displayName() – displays the name of an item.

displayDescription() – displays the description of an item.

SetName() – sets an items name.

SetDescription() – set an items description.

INTERACTIONS – Acts as a set of rules for individual actions and changes entities in response to specific actions.

getRoom() – returns the room the interaction is linked to.

setRoom()- sets the room the interaction is linked to.

validAction() – checks if an interaction is valid for an entity.

ROOM INTERACTIONS – Initiates interactions and consequences in a room in response to player input.

roomLook() – get a rooms description.

roomTouch() – touch an object in a room potentially trigger additional actions.

roomExamine() – look at a rooms inventory and get a specific description of an item.

roomDig() – attempt to dig in a room may trigger additional actions

roomClimb() – attempt to climb in a room may trigger additional actions

roomUnlockDoor() – attempt to unlock a door, will ask a player to use an item and check if that item is able to unlock the door if true then modify room to include other room or new item.

roomBreakObject() – attempt to break an object may trigger additional actions

roomMeditate() – output many blank lines to clear the terminal and "you feel a sense of calm wash over you", or a room specific variation.

roomInteractItem() - allows a player to interact with an item.

roomUseItem() - allows a player to use an item in a room.

roomPickUpItem() - allows a player to pick up an item in the rooms inventory.

CHARACTER INTERACTIONS – Initiates interactions and consequences with another character in response to player input.

getCharacter() – returns the character the interaction is linked to.

setCharacter() – sets a character to the interaction.

characterSpeak() – speak to a character may trigger additional actions.

characterGiveItem() – attempt to give an item to a character may trigger additional actions.

characteTakeItem() – attempt to take an item from a character may trigger additional actions.

characterAttack() – attempt to attack a character may trigger additional actions.

characterKiss() – attempt to kiss another character may trigger additional actions.

characterPersuade() – attempt to persuade another character may trigger additional actions.

characterGiveItem() - attempt to give an item to a character may trigger additional actions.

characterTakeItem() - attempt to take an item from a character.

GAME- Sets a general class for the implementation of the game and the “meta” functions for the entire game.

gameSetup() – sets class and attributes to default values for the start of the game.

gameStart() – initiates the beginning of the game.

gameWin() – displays the results for winning the game.

gameLose() – displays the results for losing the game.

STATS - A class for the organization and management of character stats.

getStats() – return list of stats and their values.

getStat()- return a specific stat and its value.

changeStat()- sets a new value for the specific stat.

removeStat()- removes the stat from the stats list.

INVENTORY- A class for the organization and management of an entity's items.

addItem()- adds item to a list of items the inventory is linked to in room.

removeItem()- removes item to a list of items the inventory is linked to in room.

getItem()- returns the item the inventory is linked to.

showInventory()- displays the items listed in the inventory.

getInventory()- returns the inventory.

MOVEMENT: A class that provides a player with the methods for moving around the map.

changeRoom(string) - allows a player to change a room.

checkMoveValid(string) - checks if a move is valid based on the list of adjacent rooms from the room where the player is.

ACTIONRESULTSLIST: A class for the organization and management of action results.

addActionResults() - adds an action key and action result to the list of action results.

getActionResults() - returns a list of actions which have results.

removeActionResults() - remove an action key and its paired result value.

displayActionResults() - displays and action result in response to an action key value.

ACTIONLIST: A class for the organization and management of interactions.

addAction() - adds an action to the action list.

getActions() - returns the set of actions in an entities action list.

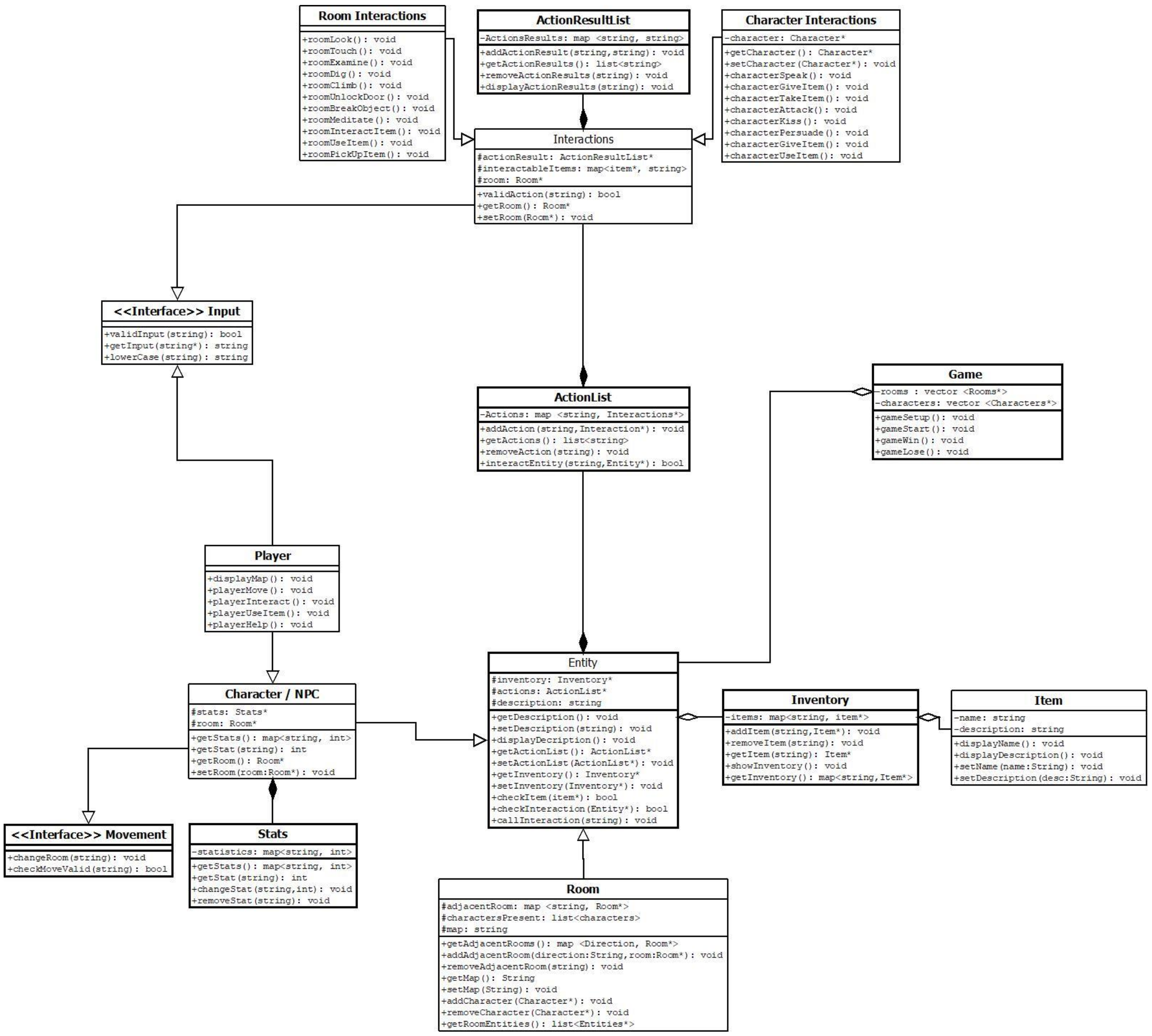
removeAction() - removes an action from the action list.

interactEntity() - Attempts an interaction with an entity.

APPENDICES

APPENDIX A: FIGURES AND TABLES

Bonus: Full Picture of the current class UML.


Room Interactions
+roomLook(): void
+roomTouch(): void
+roomExamine(): void
+roomDig(): void
+roomClimb(): void
+roomUnlockDoor(): void
+roomBreakObject(): void
+roomMeditate(): void
+roomInteractItem(): void
+roomUseItem(): void
+roomPickUpItem(): void
ActionResultList
-ActionsResults: map <string, string>
+addActionResult(string,string): void
+getActionResults(): list<string>
+removeActionResults(string): void
+displayActionResults(string): void
Character Interactions
-character: Character*
+getCharacter(): Character*
+setCharacter(Character*): void
+characterSpeak(): void
+characterGiveItem(): void
+characterTakeItem(): void
+characterAttack(): void
+characterKiss(): void
+characterPersuade(): void
+characterGiveItem(): void
+characterUseItem(): void
Interactions
#actionResult: ActionResultList*
#interactableItems: map<item*, string>
#room: Room*
+validAction(string): bool
+getRoom(): Room*
+setRoom(Room*): void
<<Interface>> Input
+validInput(string): bool
+getInput(string*): string
+lowerCase(string): string
ActionList
-Actions: map <string, Interactions*>
+addAction(string,Interaction*): void
+getActions(): list<string>
+removeAction(string): void
+interactEntity(string,Entity*): bool
Game
-rooms : vector <Rooms*>
-characters: vector <Characters*>
+gameSetup(): void
+gameStart(): void
+gameWin(): void
+gameLose(): void
Player
+displayMap(): void
+playerMove(): void
+playerInteract(): void
+playerUseItem(): void
+playerHelp(): void
Character / NPC
#stats: Stats*
#room: Room*
+getStats(): map<string, int>
+getStat(string): int
+getRoom(): Room*
+setRoom(room:Room*): void
<<Interface>> Movement
+changeRoom(string): void
+checkMoveValid(string): bool
Stats
-statistics: map<string, int>
+getStats(): map<string, int>
+getStat(string): int
+changeStat(string,int): void
+removeStat(string): void
Entity
#inventory: Inventory*
#actions: ActionList*
#description: string
+getDescription(): void
+setDescription(string): void
+displayDecription(): void
+getActionList(): ActionList*
+setActionList(ActionList*): void
+getInventory(): Inventory*
+setInventory(Inventory*): void
+checkItem(item*): bool
+checkInteraction(Entity*): bool
+callInteraction(string): void
Inventory
-items: map<string, item*>
+addItem(string,Item*): void
+removeItem(string): void
+getItem(string): Item*
+showInventory(): void
+getInventory(): map<string,Item*>
Item
-name: string
-description: string
+displayName(): void
+displayDescription(): void
+setName(name:String): void
+setDescription(desc:String): void
Room
#adjacentRoom: map <string, Room*>
#charactersPresent: list<characters>
#map: string
+getAdjacentRooms(): map <Direction, Room*>
+addAdjacentRoom(direction:String,room:Room*): void
+removeAdjacentRoom(string): void
+getMap(): String
+setMap(String): void
+addCharacter(Character*): void
+removeCharacter(Character*): void
+getRoomEntities(): list<Entities*>