Julia Franco

Software Development I

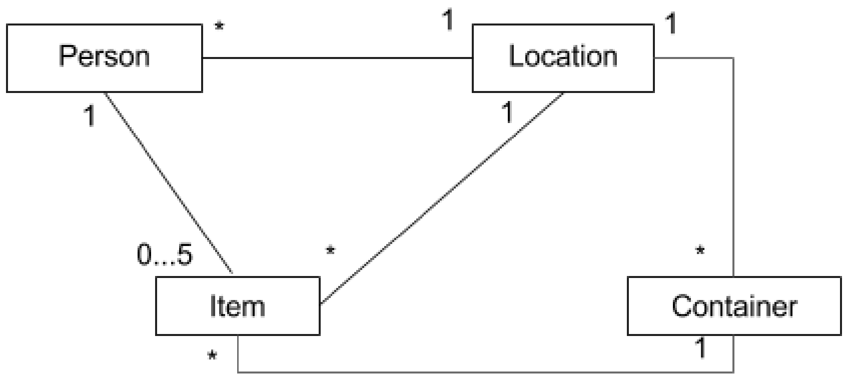
Wednesday 5:00-9:15pm

Abstract: Cove Wrench Drop: The Game is a text based adventure game in which the user navigates around the map to pick up items to combine in hopes of retrieving the last wrench from where it has been dropped.

Introduction: This game was developed to provide a challenge to the developer as well as to create a quirky and fun game for someone to play. It could potentially be defined for a small scale release on the internet if the files can be packaged in such a way suitable for distribution, or simply to exist for future amusement. This paper will include a detailed description of how users interact with it, the requirements it addresses,a literature survey of other works, a user manual, a final summary and references consulted.

Detailed System Description. Describes what the system does and how specific users interact with it. It also describes how classes interact (in UML).: This system only has one user at a time- the player. The player interacts with the system by entering commands. The game is driven by a while loop that runs while true. The first thing the system does upon launch is create all of the items in the game- locations, items, containers, and the player object itself. It then renders the description of the opening location, which is set by the no-args constructor that creates the player object. The system asks the user for a command. The user can enter any number of commands that must match exactly what is specified in a series or if/else if/else statements inside the game loop. The user can also enter a name of an item or a container if the command lends itself to it (ie, take, drop, transfer, etc). The system reads the input as the first word, then the rest of the line, to account for multiple word names. These inputs are stored in an array that the getInput function returns. The system then looks at the first argument for the command. If the command requires secondary input, it first looks to see if the user has supplied any information. If it has not, it then calls the function to get the secondary input. Along with this, it usually prints out the user's options for secondary input (ie, items in the location for the player to take, containers in the location to rummage through, etc). If the user has entered a command requiring secondary input, the system attempts to execute the command. It takes the second input argument as a string and then checks to see if that string matches the name attribute of any known object in a the applicable category. This method either returns an object of applicable type. If it is null, it skips the rest of the function and prints out the correct message. Otherwise, it checks to see it the object of applicable type is in the user's location. If so, it performs the requested action, otherwise displaying an error message. These functions are driven by loops to loop over all the elements in a list of containers and objects, and if/else statements. The player's inventory is an ArrayList of items, making the methods to add and subtract items from the list very simple. If an item is added to the person's inventory, another function is called to remove it from the location. The player enters navigational commands, and depending on the command, passes a specific value to a function to determine where to go. The passed argument specifies the column in a 2D matrix in the location class. The rows represent the locations, and the number at naMatrix[x][y] represents the index of the element in the list Locations.places is in that specific column of the value of the data field of Person.player.locationIndex to determine the row to look in. This function, Person.moveLocation, sets the value of Person.player.locationIndex to the number at this intersection and adds this index to the top of a StackOfIntergers called beento. The system then attempts to render the location. In most circumstances, the location renders fine and returns as true and allows the system to go back to the top of the loop. There are two exceptions at this time- due to the nature of Location.place[5], it immediately causes the player to lose when they go there. The system prints the description of the place, and then prints the appropriate ending from the MainFile.DisplayQuit(int) switch statement that handles what ending is displayed This method also changes Person.player.renderLocation to false, eliminating the need to place it in the code alongside the method call. At the top of the loop, there is an if statement to break out of the loop when Person.player.renderLocation is false. The other instance of Person.renderLocation returning false is when the user goes to Location.places[6]. As a joke, the player can randomly be forcibly ejected from the game. When the system attempts to render this location, it runs a method to determine if it can. It calls the MainFile.getRandomNumber method and if that method returns 12, this method returns true which causes the render method to return false and display the appropriate ending message. When the user leaves the game, either willingly, victoriously, or forcibly, MainFile.DisplayQuit(int) is called, with the integer value corresponding to cases in a switch statement to print the correct ending message before asking the user if they would like to restart. If the player types back, the system looks at the top of Person.player.beento (a StackofIntegers), deletes it, and returns the value of the index left at the top. This gets set as the player's location and runs the Person.renderLocation method. The player can also combine items. There is a subclass of the Item class called CombinedItem. When a user uses the combine command and inputs items, the system goes through a series of checks using loops and if/else statements to determine if the input is valid. It first takes the single string that is supposed to have the names of the items attempting to be combined and splits them using the method MainFile.splitInput according to a split key that gets passed along with it. In the case of combining items, the split key is "with", and the method uses the build in .split method to split the string according the the split key and return a list of String arguments. However, if the improper input is entered (the "with" not separating the two items, or only one item name, etc), the system will throw an ArrayOutOfBounds exception, which is handled by a try/catch clause. If the exception is found, it prints a message instructing the player on the proper input. Otherwise the method then looks to see if the first argument is an item, then if the second one is an item. Then it checks to see if those items are in the player's inventory and then finally if the first item is the same as the second item. If it succeeds in all of these, it then calls the constructor for a combined item, passing these two items into it as arguments. Otherwise, it prints an appropriate error message. The constructor for a combined item takes in the two items being combined and creates an object that exists as a combination of both of them. It sets the String of the item name based upon the Strings of the short names of both items. The description reads "A combination of [item1.itemName] and [item2.itemName]." It then looks to see the type of the item. Type 0 is "uncombinable"- combining it with some else will not be advantageous in achieving the goals of the game. Type 1 is the capture type, meaning that is can grasp something. If an item, regardless of type, is combined with a type 1 item, the type is set to 1. Type 2 indicates that an item possesses significant length while type 3 is the food type. This is uses if the player uses the "eat" command followed by the name of an item. If the item is type 3 and has multiple uses (ie, a package of gum), then it accesses the this.removeItemUses() method and displays a message saying part of it was eaten. Otherwise, it displays a message saying it was eaten and removes the item from the player's inventory. If the item is not type 3, the system displays a message telling the user that they do not want to eat that and goes back to the top of the game loop in the main method. Item type 4 does not show up in the player's inventory. When a player goes to add an item into their inventory, an if statement checks to see if it is type 4, and if it is, it executes without adding that object to the arraylist of the player's inventory. The constructor for the combined item also sets the two base items as "part1" and "part2" respectively, in case the player invokes the "split" command on that item. This combined item is then added to the arraylist of items and the player's inventory. If the player tries to split an item that is a combined item, the combined item is removed from the player's inventory and the original two pieces are placed back in the player's inventory according the the data fields of item1 and item2. Otherwise, it prints an error message. This is determined via a try/catch clause. The item is run through the method to determine its existence (that the string entered that is supposed to be the name of the item matches the name of an item defined in the list of items) and then cast to a CombinedItem. However, if the item is not combined, it throws a caught ClassCastException. Another key feature of the game is retrieving an item from where the player has supposedly dropped it. This item is a wrench, and this place it was dropped is "the depths of the cove". The "depths of the cove" is actually an instance of the SpecialtyContainer class that inherits from the Container class, placed in Location 0 ("Cove Walkway"). When the player uses the "deploy" command, supplemented by the name of the item they wish to deploy, it invokes the deploy method on this SpecialtyContainer (reference variable "depths"). The method checks to see if the item exists and if the item is in the player's inventory, and that the player is in the same place as the container the method pertains to (Location.places[Person.player.locationIndex].receptacle .contains(this).) The method then invokes the method SpecialtyContainer.alternateDeploy, which checks to see if the item being deployed is an item that prints out a specific message for trying to deploy it and has alternate behavior, such as a carton of eggs or a ladder. If it is a predetermined item, it returns true which causes the deploy method to return and go back to the top of the main game loop. Otherwise, the method returns false and the system then calls the GetFoundItem method, if the checkConditions method returns true. The checkConditions method looks to see if the item meets specific requirements- that it is long enough and it can grasp an item, and returns true if it does. The GetFoundItem method first calls the determineReturn method and saves what comes back as an item. The determineReturn method first determines what multiplication factor to use in determining the items. Conditions that influence what the multiplication factor is include if the player has dropped an egg into the container and/or if the player has the worklight. A random number between 0 and the multiplication factor is generated and that number is returned back to the GetFoundItem method. If that random number matches one of the numbers hardcoded to items, then the method returns that item. Otherwise, it prints a message saying the retrieval attempt was unsuccessful. The item is then directly deposited into the player's inventory. In the process of that, the system runs a method called determineWrench to see if the player is trying to add a wrench to their inventory. If it is the one they dropped, the game ends with a message of congratulations. If it is one of two other wrenches in the game, the user is asked if they would like to continue or not. If they want to continue, the game goes back to the top of the game loop. Otherwise it prints the correct ending. If at any point, conditions are unmet, the system informs the user of their transgressions and the game goes back to the top of the game loop.

UML Diagram of Class Interaction:



A person can only have one location, but a location can have many people. A location can have many items and many containers, but each container and item can only have on location. A person can have up to 5 items, but an item can only be owned by one person. A container can hold unlimited items, but an item can only be in one container at a time. Really, an item can only be in one of those places at one time.

<u>Requirements</u>: At the current moment in time, the following proposed requirements have been met. The user can collect items and hold them in an inventory. However, the inventory has to be enabled by finding an appropriate item that can hold an inventory. The user can outright discard items, or put them in a secondary inventory. There is also a map the player can find and display to visually see the layout of the world. The player has specific attributes, including a name, an inventory, and a location. There is a map the a person can navigate, with the option to go in 6 directions- up, down, left, right, forward, and "backward" (acts as a turn around and go forward.) The player can rummage through containers to find other items, as well as being able combine the items together and deploy these combined objects. If the objects deployed have the correct attributes, there is a chance the user will retrieve

the wrench that it is supposed they dropped. Otherwise, it will give the player a message specifying what requirements the item does not meet. When the player retrieves an item, there are a number of items it could possibly be, one being a set of car keys. If the player finds the car keys as well as the car elsewhere on the map, they can use the car to drive to specific locations. When the player has these items and they type drive, if they are in a location that can be driven to, the system will prompt them for a location to drive to, as well as a list of valid locations. One major proposed requirement was removed between the milestone and final reports- the presence of and the ability to interact with other people within the game. Due to time constraints, this feature was removed from the set of requirements, but could be implemented in future releases. The player would become a subclass of person and retain all of the current functionality, while other people would become objects of the Person class. They would have names and randomly generated locations, as well as some pre-programmed phrases for if the player talks to them. Meeting another person would not cause instant forfeiture unless the player was in a predefined set of circumstances that would lend themselves to it (ie getting caught in an unauthorized area). Another feature proposed- the player having a score and points being added as the player found and combined items- was removed because it was deemed extraneous. Another possible requirement that could make game play more interesting would be allowing an item to appear in one of a few predetermined locations. This would probably be achieved by giving the item a data field to hold the predefined locations it could appear in, and then have methods to randomly determine a number between 0 and the length of the list, and assign the location value at that index to the item's location.

<u>Literature Survey</u>: There are many well known text based adventure games. Titles include Zork, The Hitchhiker's guide to the Galaxy, and Spider in Web. Many of these games include commands used here- basic navigation, the ability to hold and use items. These games have unique stories and features that shape the game play- the ability to dig and use swords, dependent upon the items held and the scenario the player faces. This system employs many of the same functionality and styles of these games, in the descriptions and commands usable in the games.

User Manual: The user inputs commands to move the player around the map, and to collect and drop items and hold them in an inventory. The user can also rummage through various containers for additional items. Since the inventory can only hold 5 items, there is an unlimited secondary inventory that the user can transfer items to. Many commands can be used in conjunction with an item name or item short key to perform specific actions.The user can also type help for a list of valid commands and pointers for their use, as well as quit out of the game. It will ask the player if they want to reset. If they choose yes, the game will begin again, Otherwise, it will print the credits and terminate the execution. The user is also be able to combine items, and deploy them in an attempt to accomplish the goal of the game. Additionally, the user can pull a set of car keys and then find a car within the game and "drive" somewhere. Currently, there are only three places you can drive to, and do not hold anything critical to the completion of the game. As a note of caution, the "drive" command will work whether or not the player has picked up the car ("take car") as long as the player is in the same location as the car. If they drive somewhere without having picked up the car, they will effectively be stranded at the location as they will not have a car to get back. They can navigate back using the "back" command. The user can also find various money scattered around the game, which adds to the amount of money they have, but does not show up as an item in their inventory. The user input is not case sensitive.

Conclusion: The system functions as desired and has met all of the goals that it set out to meet. There is full navigation, 28 locations, approximately 25 items than can be collected and combined with each other, and 12 different endings. The game is interesting, quirky, and unique, containing hidden locations and unique surprises. It is complete in terms of setting out to do what it was intended, but there is always more that could be added.

References/Bibliography: Y. D. Liang's Introduction to Java Programming, Comprehensive Ver., 10th Ed., StackOverflow.com, as well as Zork and Spider and Web.