**CO871 – Assignment 1; Bad Zuul**

**Report on bad practise**

|  |  |  |
| --- | --- | --- |
| **Flaw** | **Consequence** | **Solution** |
| All commands within the game are case sensitive, therefore commands entered in capitals do not work. | Results in poor game play experience, commands should work as lowercase, capital or a mixture of the two. | Within the Parser class, the line ‘inputLine = reader.nextLine()’ should be extended to include ‘.toLower()’. This would mean that even if the user inputted uppercase commands, the parser treats them as lowercase and therefore recognises the command. |
| When typing ‘help’ in the console, only three command words appear even though there are more listed within the CommandWords class. | Poor game experience, prevents the player from learning all the commands within the game. | Within the CommandWords class, there should be a public method that returns the string array ‘validCommands’. When the player types help on the command line this ‘getAllCommands()’ should be invoked and the string array should be returned. |
| There is a bug in the ‘take’ command, preventing players from taking legal items. I.e. items that do not have a weight large enough that once added to ‘totalWeight’ does not exceed the MAX\_WEIGHT limit. | The bug is caused in the Game class, within the take() method. There is a set of logic that calculates the new totalWeight by adding the current totalWeight and item weight together. The logic should check to see if the new totalWeight is bigger than or equal to the MAX\_WEIGHT limit, but instead checks to see if the new totalWeight is less than or equal too. | Change the logic from <= (less than or equal too) to >= (more than or equal too.). This way, only when the new totalWeight is more than or equal too the MAX\_WEIGHT limit will the error message “item is too heavy” appear. |
| Room.java contains public fields. This is very bad programming practise. Has the potential to cause serious game issues down the line if another (less experienced) programmer takes over the project and does not know what they are doing. | Since the fields are public, they can be accessed from outside the class – for example in the Game.java class. This can have unforeseen consequences down the line, for example the field ‘Room northExit’ could get deleted/ overwritten during the course of a game and thus make it unplayable. | Simply set the field declarations within the Room class to be private. If the programmer does need access to these fields, a getter or setter method should be written within the Room class. The getter/setter would be made public and the fields themselves should remain private. Getters are good as they return a copy of the original object they are returning. I.e. you can get a value and change it without affecting the original objects value. |
| Within class Command, there is inconsistent naming of the first/second/third fields. In my opinion this is bad practise as it lacks consistency. | Lack of inconsistency could lead to errors in code further down the line. Especially if a less experienced programmer takes over, they might not realise that the field ‘commandWord’ is actually ‘firstWord’. This could lead to two separate fields to represent the first command word entered in by the user playing the game. | Simply change (using a refactor tool) the name of the field ‘commandWord’ to ‘firstWord’, adding consistency into the naming conventions. |
| There is an error in the Command class’s constructor. ‘this.thirdWord’ is being set to the value of ‘secondWord’. | This will prevent the player from ever using three-word commands, as the value of the third word is always going to be set to the value of the second word. | Simply set the value of ‘this.thirdWord’ to be ‘thirdWord’. |
| In the Room class, it is redundant to have all four rooms declared as separate fields. This can cause messy code later on that could lead to performance issues. | Lots of extra code to write, more code to maintain and easier to get things wrong when handing over to new programmers. Could also lead to performance issues. | Would be better to declare a HashMap to hold the exits, using a string as the key and a Room object as the value. This would make checking the users input against the possible exists for any given room easier too. Use the HashMap.containsKey(exitName) method to check if a given exit exists for the current room. |

|  |  |  |
| --- | --- | --- |
| The Room class contains a String and int to represent an item/ itemWeight. This is bad as it limits any given Room to only 1 item. | This could limit game play, for example if a room is required to hold more than one item at a time, the current code does not allow it. | Solution would be to create an Item class, within the class have name, description and weight fields; along with appropriate getters for each. Within the Room class there should be a HashMap that takes a string ‘itemName’ as the key and ‘Item object’ as the value. This makes it possible for each room to house multiple items. Each time the user wants to interact with an item in any given room, the room is first checked to see if the item is there, i.e. doing HashMap.containsKey(itemName). If true, an item object will be returned and other logic can execute. If false, then the console will output the appropriate message. |
| The setExists method within the Room class contains code that is redundant as it is. When the exists are initialized (Room northExit, for example) they by default now have a value of null.  Therefore, there is no need to check the values of the parameters of the method to be not null; as if one of the paramaters is in-fact null, that particular exit is going to be set to what it already is, null. | Repeated code, makes the method messier than needed and could be confusing to new programmers being handed the project. | Within the setExists class, remove the check on the method paramaters to see if they are null or not. Simply set the value of the parameter to the appropriate field.  If the parameter value is null, the field value is already null so will not have any negative affect on game. |
| In Room class, method addItem will not add a new item to a given room, but rather overwrite what the original item was. | Because items and itemWeights are String/int fields, there can only ever be one item in any given room. This means that when addItem() is invoked, it will remove the original item and replace it with the new item.  If this is being done as the result of some sort of in-game trigger, an important item (such as a key for example) could get removed and therefore prevent the player from completing the game. | A solution would be for each room to have a HashMap<String, Item> items. This would allow each room to hold more than one item at a time, using the items name as the key.  One could simply call: ‘items.put(itemName, itemObject)’ to add a new item to a given room. |
| Not a detrimental flaw, but the ‘containsItem’ method within the Room class is odd.  Usually when checking for the existence of something a method would return a Boolean true/false.  This method returns the weight of the item. | When used in the wrong context by the wrong programmer (e.g. a new programmer on the project who was just given the code), this method could throw unexpected results.  For example, if this new programmer didn’t realise that the method returned the items weight, but instead thought it returned Boolean true/false, used within an if statement the method would not work. | Suggest change the return type of the method to be Boolean true/false **or** make the method return to item the player was looking for (provided it exists), that way in the section of code invoking this method the programmer can then proceed to use getters within an Item class to get the items name/desc/weight etc. |
| In Game.java, two ArrayLists have been declared without generics associated to them. | Could cause issues down the line if objects of different types are put into the ArrayList. | Simple add a generic <String>, or <int> to the ArrayLists; e.g. ArrayList<String> items; |
| In the Game class there are two ArrayLists (items and weights) that could be refactored into a single HashMap. | Makes the code messier as the programmer needs to interact with two ArrayLists, not to mention this is a greater memory requirement. | Converge the two ArrayLists into a single HashMap, using the items ArrayList as the HashMap\_Key and the itemWeight ArrayList as the HashMap\_Value.  Better yet, create a new class ‘Inventory’ that contains this information (info on what items the player currently has). Refactoring this out of the Game class makes this concept easier to manage. |
| Within Game.java, there is nothing wrong with defining the max carry weight of the player, MAX\_WEIGHT = 10.  However, this could be added to an ‘Inventory’ class (as mentioned in point above). | As the Game.java class is essentially the games operating logic, it makes more sense to refactor this property of the player out and put it into a class that reflects more of a purpose, i.e. holding items. | Create a new class, ‘Inventory’. This class will be used to hold the current carried weight of the player, what the players max carry weight is. The HashMap of items the player is currently holding. And methods that can get the current carry weight, check the future carry weight **if** a new item is added to the inventory, and get a list of the items in the inventory; in this case returning a list of ‘Item’ objects. |
| In the Game class constructor, the totalWeight field is being set to 0. This is redundant as could have been set to 0 in the field list outside of the constructor. | No consequence really, but could have made the code look a little neater. | Where totalWeight is first declared, re-write the declaration to be “private int totalWeight = 0;” |
| Within Game class, all game rooms (outside, theatre, etc.) are created locally within a method (createRooms()).  This means that at no other point can the properties of any given room change.  For example, certain actions trigger events in certain rooms like a door being unlocked and an extra exit being discovered, or a key dropping down and an extra item being added to a room. | Limits game playability, means there is no feature to change the contents/exits of a room during gameplay. | Within the Game class, create a HashMap of all rooms in the game, using the key as a String ‘name’ and value as Room ‘theRoom’.  This would mean that at any point in the game, the player could perform an action that alters the appearance of any room, e.g. adds a new item to current/previous room. |

|  |  |  |
| --- | --- | --- |
| All over the game’s code there is a large use of System.out.print/println().  This is not terrible however will cause great issue in the future if the developers want messages to output to somewhere other than the STD out console. | Will make it difficult to print message to anywhere other than the STD out console.  For example, if the second iteration of the game contains a GUI and the developers want a message to be displayed as in game text, this could be difficult with the current implementation. | Create a ‘OutputHandler’ class that handles the way the output is treated. This could then be extended in the future to allow the freedom of outputting to a game GUI. |
| Buffer overflow on userinput… |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |