MARKING GUIDE Name ...Aisha Saleem.....Login ...as2634.....

OVERWRITE YOUR MARKS IN THE LAST COLUMN

Report

1.	Fields in Room are public. This makes the code brittle and hard to change as clients may be tempted to rely on implementation rather than interface. Makes it impossible to guarantee that state invariants are preserved. Lack of encapsulation may lead to changes in many places. Room exits are too tightly coupled with Game.printWelcome. Make all fields (except possibly constants) private or protected	1 1 1 1
2.	There are bugs, e.g. several cut and paste errors in the Command class, comparison tests in Game.take()	1
3.	Duplication in Game Makes changes hard, as we have to remember to make changes in all places. Introduce new methods	1 1 1
4.	Room exits are hard-coded. Not only is this clumsy but it makes it hard to extend the game and encourages tight coupling.	1 1
5.	Separate fields for item names and weights makes for awkward programming. Hard to pass and return items to/from methods. This also reveals too much about items (tight coupling) making it hard to extend the class. Introduce an item class	1 1 1
6.	Game knows too much about rooms. printDetails should be in Room not Game.	1 1
7.	Game.processCommand could be better coded with a Java 7 switch. This design is not very object oriented A much better design is to use the Command pattern, simply calling the exec() method for the command entered.	1 1 1
8.	The code uses only the default package, which is deprecated. Deprecated because it makes it harder to choose names in large programs. Group related classes into packages.	1 1 1
9.	Game has an implicit single Player. This leads to several problems. The Game class is not very cohesive. It muddles generic mechanisms for playing the game with game-specific rules and data. The Game class should be just a generic framework. It would be quite difficult to extend it to make a multi-player game	1 1 1 1 1
10.	The rules of the game (commands) are hard coded into Game. This makes it harder to implement a different game. Each command should be in a separate class, completely uncoupled from the generic game framework.	1 1 1
11.	Game has high implicit coupling with Room through the use of hard-coded strings for directions.	1
12.	Game should be an abstract class, sub-classed by a game-specific one. Alternatively, a configuration class might be possible although this is less flexible.	0
13.	Characters would be better as a separate class. The Character class should be abstract. This would allow them to act (implementation in a subclass of Character).	1 0 0

14.	The code assumes a console throughout. This makes it difficult to add, say, a GUI or allow the tool to be used across a network Separate the I/O methods into a separate class, so that you can use different user interfaces. Change all the printing code to return strings which can then be passed to the I/O methods. Change all printing code to return strings which can then be passed to the I/O methods.	1 1 1 1
15.	CommandWords does not provide a way to iterate through commands. Makes it hard for the tool to provide help	1 1
16.	No final fields	0
17.	Fields which should be immutable could be modified erroneously.	0
18.	Documentation is inadequate Add appropriate Javadoc comments	1 1
19.	Game class is printing commands which should be the responsibility of the CommandWords class. It again violates the responsibility driven design principle	1
20.	Literal string used to refer exits/directions in the whole code.	1
21.	Current code is not restricting to create multiple instances of the Game class.	1

Total 32

Implementation

Initial tidying of the code (visibility, mutability), removal of constraints (e.g. the number of exits from a room, number of items in a room, number of players).

1.	Make all fields private (in Room.description, Room.exits, Room.items, etc)	2
2.	Remove duplication, e.g. introduce printDetails in Game.goRoom and Game.printWelcome.	4
3.	Add an Item class with final fields and accessors.	4
4.	Use a map of exits. Make Room have a Map<String,Room>, new Room.setExit and Room.getExits methods, change Room.setExits, Room.printDetails, and Game.goRoom.	5
5.	Give Room a map of items. Add Room.getItem, Room.addItem, change Room.addItem, Room.removeItem, Game.take, Game.give	5
6.	Make fields final wherever possible, e.g. for the fields	0
7.	Move printDetails from Game to Room	2
8.	Only if a Java 7 switch rather using equals() is used in Game.processCommand. Better solutions have removed this completely in favour of an abstract command class and reflection e.g. see point 12.	2
9.	Place all code in packages	4
10	Add a Player class with the rules. Now the Main class is just a framework	20

Sub-total 48

Now to separate the game framework from the specifics of a *particular* game. One way is through having separate sub-command classes. Reflection can be used to isolate concerns even further. Marks for alternative mechanisms can be awarded but you'll have to justify why they satisfy the requirements.

11.	Make Command abstract and add an abstract execute method.	4
12.	Provide sub-command classes.	20
13.	Use reflection to load and manage sub-command classes. This is nicer than a big switch in the Command class, but a little harder.	20
14.	Make Game abstract and provide a game-specific Game	0
15.	Provide an abstract character class. Add characters e.g. as inner classes in Game.createRooms or as separate classes	6

Sub-total 50

User interface and language independence. Again, marks can be awarded for alternative mechanisms provided you can justify that they meet the requirements.

16.	I/O independence: delegate I/O to In and Out classes	6
17.	Change print statements to return strings; Game.printWelcome, HELPcommand	4

Sub-total 10

Miscellaneous

18.	Thorough use of Javadoc	10
19.	Implement singleton design pattern in Game class which restricts creation of multiple instances of a class and enforce all clients to use the single instance.	1
20.	Create Enumerated types for the Exits.	1
21.	Give Command tells us that Items can be given to Characters which shows there should be a relationship between the two classes but at present there is no relation between the two classes. HAS-A relationship should be created between Character and Item class as Characters can have item or items(in future).	1
22.	Drop item method removing element from Array list without using Iterator. Use iterator to remove an item from a collection.	1
23.	Type not specified when declaring Collections to hold items and weights in Game class. Collections are generic classes. It is a good programming practice to explicitly state which Object a collection is going to hold. Specify type in angle brackets.	1
24.	Parser class should have only one instance throughout the application but	1

	current code doesn't enforce it. Implement singleton design pattern for the parser class.	
25.	Exceptions not handled in the code. It is a good programming practice to handle exceptions and can also be used to show customized error messages to users in a more user-friendly way	1
26.	Implicit constructor not found in some classes. It is a good programming practice to include constructors even if the body is empty. Add constructors wherever missing.	1
27.	Separate collections are used for storing Items and their relevant weights. Using more appropriate collection for the purpose of storing related information in a key-value pair i.e. HashMap instead of separate array lists.	1
28.	Game class is coupled with the logic to create players which also shows poor cohesion. Implement Factory design pattern and encapsulate the logic to create players into a separate class to provide loose coupling and high cohesion. It will also support company's goal to create Computer Controlled Players in future with minimum change in game class.	1

Sub-total 20

Total 128

Grand total 160