Designing and Implementing (Some of) Dungeon and Dragons Character Classes

Problem Statement

Wikipedia reports that "Dungeons & Dragons (commonly abbreviated as D&D or DnD) is a fantasy tabletop role-playing game (RPG) originally designed by Gary Gygax and Dave Arneson. It was first published in 1974 by Tactical Studies Rules, Inc. (TSR). The game has been published by Wizards of the Coast (now a subsidiary of Hasbro) since 1997. It was derived from miniature wargames, with a variation of the 1971 game Chainmail serving as the initial rule system. D&D's publication is commonly recognized as the beginning of modern role-playing games and the role-playing game industry. D&D departs from traditional wargaming by allowing each player to create their own character to play instead of a military formation. These characters embark upon imaginary adventures within a fantasy setting. **A Dungeon Master (DM)** serves as the game's referee and storyteller, while maintaining the setting in which the adventures occur, and playing the role of the inhabitants of the game world. The characters form a party and they interact with the setting's inhabitants and each other. Together they solve dilemmas, engage in battles, and gather treasure and knowledge. In the process, the characters earn experience points (XP) in order to rise in levels, and become increasingly powerful over a series of separate gaming sessions." (Emphasis mine.)
We want to create character generator: a tool with which a player can create xyr own character, from a set of given character classes (e.g., the Ranger), a set of abilities, and pieces of equipment. Instead of using numerical values to characterise the character' abilities, the system will offer explicit, first-class abilities (e.g., "intuition" or "leadership"). The system will also offer to equip characters with different internal and external pieces of equipment or capabilities (e.g., armours, weapons, bags… but also "stealth" or "shapeshifting").

Rules

Explanations and Justifications Matter!
In every justification, report the (abstract) design problem to solve, discuss possible alternative solutions, and explain the trade-offs of each solutions before choosing one solution.
Favour short explanations. Be careful of grammar and vocabulary. In particular, use the proper terms as seen during the course.
If necessary, draw short but illustrative class and sequence diagrams. (You can draw these diagrams by hand if it is faster than by computer).

Question 1 : The given code provides examples of character classes (e.g., Ranger). Innate abilities are numerous, but we will consider the following ones:

- **Strength: burly, fit, scrawny, plump**
- **Dexterity: slim, sneaky, awkward, clumsy**
- **Constitution: strong, healthy, frail, sick**
- **Intelligence: inquisitive, studious, simple, forgetful**
- **Wisdom: good judgement, empathy, foolish, oblivious**
- **Charisma: leadership, confidence, timid, awkward**

Redesign the given code to allow adding abilities to the characters so that new innate abilities could be added later without having to modify (much) the implementation of the character classes. Before implementing your design, justify your choice in the space below (no more). Then, implement your design based on the given code.

Explanation and Justification of adding one level of indirection:

In order to encapsulate the abilities to the characters add one level of indirection could be a potential solution for this problem. This design/solution will also be helpful for adding new innate abilities without changing (much) the existing implementation (abilities/character classes) that use any of the abilities from six. Adding one level of indirection (allow favour composition over inheritance) will provide more flexibility, allow changing implementation and safe inheritance. In addition, we also use double-dispatch (Object receive a message, send back a message with itself as parameter).

With this design (adding one level of indirection), other type of objects can reuse our six abilities (e.g., Strength, Dexterity, Constitution, Intelligence, Wisdom and Charisma) because these abilities are no longer hidden away in our Character classes. In this design, the obvious picture on encapsulated characters / abilities are we can think them **as a family of algorithm** instead of thinking the character's abilities **as a set of abilities**.

In terms of coding, *Client* makes use of an encapsulated family of algorithms for all six Strength, Dexterity, Constitution, Intelligence, Wisdom and Charisma abilities. We encapsulated all six abilities in a separate package and used interface for adding one level of indirection. Then we implement all the abilities for specific abilities option as a concrete class (e.g *CharismaAwkward* implements *ICharismaCharacter*). We created abstract class (e.g., *Character*) and added instance variables for each abilities, then delegate to the abilities classes as well as set abilities dynamically (e.g., `public void setCharismaCharacter(ICharismaCharacter cc) { this.charismaCharacter = cc; }`) and from here double-dispatch happened. Questions1.zip has clear view as a potential solution for this problem. We also draw UML-Add One Level Of Indirection using Eclipse additional features and kept in the root folder in the source code (src/main/java).

Question 2 : Characters can wear various types of clothing: boots, hats, helmets, cloaks, armour. Redesign the given code to allow adding types of clothing to the characters in such a way that new types can be added and worn by characters without having to modify (much) the implementation of the character classes. Before implementing your design, justify your choice in the space below (no more) and explain how you enforce that certain types of clothing must be worn before or after other certain types, e.g., how do you enforce that armours must always be on top of clothes? Then, implement your design based on the given code.

Explanation and Justification of using Decorator Design Pattern:

In order to add / modify the behaviour of some methods (characters) of some objects at runtime without having to modify the existing implementation of the character classes, we should use the Decorator Design Pattern. Decorator design patterns are most often used for applying single responsibility principles since we divide the functionality into classes with unique areas of concern. As we need to allow the adding types of clothing to the characters in such a way that new types can be added and worn by characters without being modify (much) the implementation of the character classes, hence we are using the Decorator Design Pattern.

However, decorators have their own cons (Problems) like every other good thing, such as presence of lots of small classes and problems in configuring multi-wrapped properties, these are good to know because of their ability to compose complex objects into simple ones, instead of having monolithic, single point agenda classes. (Pros and Cons Source: Dzone, Blog)

Here, in terms of coding, Ranger class have different character and each character have multiple and different option, so, we can separate those option using enum for collecting constants of each character's different characteristics. We have a *ICharacter* interface, which can include *getStrength, getDexterity, getConstitution, getIntelligence, getWisdom, getCharisma, getClothings.* We have one Concrete class (*Character*) of *ICharacter* to define specific Character. When we need additional characteristics of Ranger class then Decorator Design Pattern to solve this problem without changing the existing implementation of Character classes. We would like to have some additional characters (functionalities) for the *ICharater* like *CharismaDecorator*, *ConstitutionDecorator*, *DexterityDecorator*, *IntellinenceDecorator*, *StrengthDecorator, WisdomDecorator, ClothingsDecorator* classes. Using those concrete classes by extending an abstract wrapper (*CharacterDecorator*, a decorator in this problem) class that implements the *ICharacter* interface, we can add new innate abilities without changing the existing functionalities. A simple main program (Client) will run to execute and test the decoder code.

As we can see, by creating the wrapper and decorator classes, we have added and customized the behavior of *ICharacter* and *Character*. Added clothing's abilities for the characters. We can control that armours must always be on the top of the clothes from the *Client* (see 2nd part).

2nd Part (how do you enforce that armours must always be on top of clothes): We would use enum to ensure that the armours must always be on the top of the clothes. Enum is used for collecting constants item of each character's and in the list of enum we kept it in the first place, so where the option will be called then armours will be on the tops of the clothes list. It is customizable if the requirements changed according to the time. We implemented this portion on our code. Questions2.zip will provide a clear view of this problem as a solution.

We also draw UML-Decorator Design Pattern using Eclipse additional features and kept in the root folder in the source code (src/main/java).

Question 3 : Characters can carry various items in satchels or boxes: food items, books, gold coins, rings. Satchels are useful for food items, books, etc. while boxes protect gold coins, magical rings, etc. Separate for the given code, design and implement a hierarchy that offer satchels and boxes in which various items can be stored. Before implementing your design, justify your choice in the space below (no more) and explain how you enforce that certain items can only be put in satchels, e.g., food items, and not boxes (because boxes are too small). Then, implement your design independently of the given code.

Explanation and Justification of using Abstract Factory (AF) + Composite Design Pattern:

1st Part (Abstract Factory (AF) Design Pattern): Abstract Factory (AF) Design Pattern allow us to explicitly declare interface for each distinct product of the product family (e.g., Boxes, Satchels, Strength, Dexterity, Constitution, Charisma, Clothing, Intelligence, Wisdom). And then we can call all variant of the abilities (i.e. products) following those interfaces. This DP provides a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. In addition, this DP allow packaging code, hiding information and decouple the client code from any particular implementation of an abstraction. The "factory" object has the responsibility for providing creative services for the entire platform family. Clients use the factory pattern to create platform objects but never create them directly. Therefore, we select this design pattern, so that we can separate for the given code, design and implementation along with hierarchy that offer Satchels and Boxes in which various item can be stored. Although it allow loose-coupling, but makes code more difficult to read as all of the code is behind an abstraction that may in turn hide abstractions (Source: Quora).

We also design the UML-Abstract Factory DP (handwritten for this design) and attached/kept in the root folder of the solution. **In terms of coding**, in order to implements the Abstract Factory design, we started with *ICharacter* interface and we developed concrete implementations (e.g., Ranger, Barbarian, Druid and Wizard) of the *ICharacter* interface. I also added several enum in the *ca.concordia.soen6461.character.abilities.option* package to the name of character, abilities of the character and options of the each abilities. Then, we implemented the AbstractFactory abstract class and extends this class with each character abilities/ abilityType (e.g., ***Boxes***, ***Charisma***, ***Clothings***, ***Constitution***, ***Dextrity***, ***Intelligence***, ***Satchels***, ***Strength***, ***Wisdom***) as a concrete class. We also create a FactoryProvider class to return ability instance by getFactory() method according to the abilityType. Then we invoke all the necessary method and classes to generate character instance and type of ability of the characters with specific option of the ability.

2nd Part (Composite Design Pattern): By implementing Composite design pattern we can enforce that certain items can only be put in satchels, e.g., food items, books. Clearer picture has been implemented in the code (Questions3.zip). In terms of Coding, add leaf node Note: Assume that all the items in Satchels are leaf because we only count (group total number) the same/unique item as same item could stay multiple time. And there is no overlap between the items. Suppose, the Satchels can contain only any 5 items (e.g., FoodItem, Books) So, if we provide more that 5 items then the container cannot store or character cannot carry the Satchels, hence it is not allowed to store more than 5 considering saturation point (5). We have completed coding part using Composite DP and the solution zipped by name Questions3.zip, where a clearer view has been provided. We also draw UML-AFDP and Composite DP for those source code using Eclipse additional features and kept in the corresponding packages folder in the source code. **However**, In Composite Design pattern, Composite design pattern is applied to any recursive data structure (Pros). But, once the tree structure is defined, composite design makes tree overly general. Leaf class have to create some methods which has to empty (Cons, Source: Dzone)

Question 4: Redesign the given code to allow characters to carry satchels and boxes (themselves possibly containing various items) and to allow characters to possess powers, e.g., spells, infravision, summons, etc. Before implementing your design, justify your choice in the space below (no more) and explain how you distinguish items (e.g., satchels) from powers (e.g., infravision). Then, implement your design based on the given code.

Explanation and Justification of using Composite + Extension Object Design Pattern:
1st Part (allow characters to carry satchels and boxes): Here, we used **Composite** pattern to allow characters to carry satchels and boxes. In this part, there are two containers (e.g., Satchels, Boxes) and each container have specific composite items that create a tree structure. Therefore, this tree structure helps us to use composite pattern for solving this challenge. We assume root node as *CombinedItemContainer*. Then, we consider other node as like tree structure to allow the character to carry satchels and boxes by using composite design pattern. Hand Writing UML diagram and tree structure for composite and Extension Object DP are provided on the source code root folder (src/main/java) as well as use Eclipse additional features for generating UML diagram in the respective packages name of the Design Pattern.

2nd part (allow characters to possess powers): We already solved Qestions-3 using Abstract Factory (AF) Design Pattern, therefore, we just **expand this solution** in order to allow the character to possess powers, e.g., Spells, infravision, summons, etc using the **Extension Object Design Pattern**. We select this DP because it provides only a minimal interface used to manage the extension itself and allowing character to possess powers indicates adding extension of the existing features (done in questions-3 using AFDP). Therefore, we opt to implement this design pattern to solve this problem as it (**Extension Object DP**) support the addition of new or unforeseen interfaces to existing classes and provide one demand extension like singleton. It Extends objects to control their extension (instantiations and garbage collections). However, Extension Object DP has risk of name clash; possibly, use reflection. In addition client must know the possible extension. We also provide the UML diagram in the root folder of the solution in convenience of this DP.

3rd Part (Distinguish between items and powers of the characters): Generally, both are same because both belongs to the characters where satchels & box need to be carried and power need to be possessed. However, to allow carry the satchels and boxes we used Composite design pattern to allow character to carry both containers, whereas, to allow character to possess powers we used **Extension Object design pattern** because we already developed questions-3 and this is an additional functionality.

In terms of Coding, to solve this problem we combinedly used Composite, Extension Object and Abstract Factory (previously developed for solving questions-3) Design pattern. Each DP pattern has specific package name to solve the problem.

A clearer view has been depicted in the developments (Coding) part and we keep its name as Questions4.zip.

Question 5: Characters can wear different types of clothing, carry items, and possess powers, which all combine with their innate abilities. These combinations result in the final values for a character's abilities. For example, a character's final strength could be the sum of the character's innate strength (Question 1) plus the strength added from wearing (or not) armour, helmets, etc. (Question 2) plus the strength added from carrying special items and having powers (Questions 3 and 4). Combine and redesign the given code and your previous code to allow computing a character's final abilities, e.g., strength, without having to modify your design every time the rules of the game change. Before implementing your design, justify your choice in the space below (no more). Then, implement your design based on the given code.

Explanation and Justification of using Abstract Factory + Extension Object Design Pattern:

1st Part (A character's innate strength (Abstract Factory DP)): We already used Abstract factory Design pattern in questions-3,4 and we are extending the previous solution for solving *"A Character's innate strength"* in this questions-5 (Questions#5_1 section in coding) for extracting strength. So, we just reuse solution from questions-3,4 and extend/added/modify few more codes in this question. So, we will not explain same things again here due to space constraint.
2nd Part (Characters can wear different types of clothing (Extension Object DP)): So, here we used Extension Object DP for implementing "*Characters can wear different types of clothing*" (Questions#5_2 section in coding). As this is new innate ability to add (*Note: This feature is exist in questions-2, but we used Decorator DP there in Questions-2, hence in this solution we are considering this as new feature*) of a character so, we implement this feature using Extension Object DP without changing the existing implementation of questions-4.
3rd Part (A character's carry items special item in Satchels or Boxes (Abstract Factory DP)): Again, we already used Abstract factory Design pattern in questions-3,4 and 1st part of the question. Similarly, we are extending the previous solution (questions-3,4) for solving *"A character's carry items special item"* in this questions-5 (Questions#5_3 section in coding) for extracting carried items in containers (e.g., Satchels, Boxes). So, we just reuse solution from questions-3,4 and extended / added / modified few blocks of codes in this question. So, we will not explain same things again here.
4th Part (Characters can wear different types of clothing (Extension Object DP)): So, here again we used Extension Object DP for implementing "*A Character strength adding by allowing possess power*" (Questions#5_4 section in coding). As this is not new innate ability to add of a character, we reuse this existing implementation from previous Questions (Question-4).

After implementing all the features of a character according to the requirements, we combined all the result in the final values for a character's abilities (i.e., strength). Therefore, we redesign the previous questions code to allow computing a character's final abilities, e.g., strength, without having to modify the design every time the rules of the game change.

Here, in the code we only consider the Ranger Character (others: Barbarian, Wizard, Druid), but each will be invoked in similar way for understanding a character's final abilities.

We added UMLs, auto generated by Eclipse and hand written UMLs are attached in the source code root folder (src/main/java).