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).

Design and Code Quality Matter!

Use the Maven project given to you as basis for your code.

Use Java 7. Do not worry about generics or the latest features of the Java language.

Make sure to choose the most appropriate design.

Make sure to write simple and clean code.

Make sure that your code compiles and runs.

Modify the Client class to illustrate via one or more examples your design/implementation.

Package types (interfaces) and classes appropriately.

Make types, classes, methods, and fields visible appropriately.

Name packages, classes, types, methods, fields, parameters, and local variable appropriately.

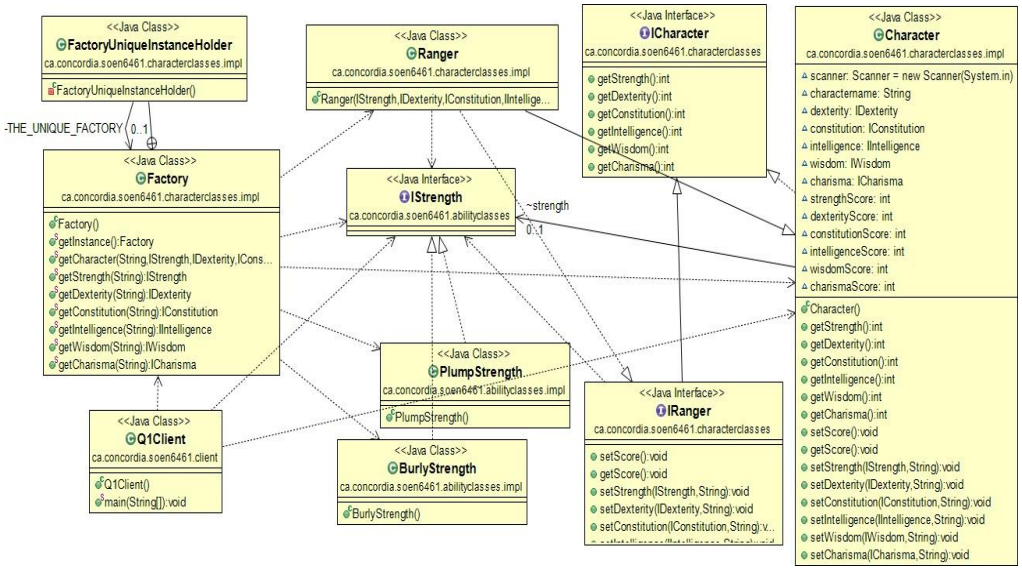
Deadlines and Submission

Submit your answers (this file with your answer in **PDF** as "Report") and your code (five **ZIP** files "Question N.zip") via Moodle before Thur. Apr. 30th, 11:59pm. No other format/names will be accepted.

Question 1 (20 pts): 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 "values" for innate abilities could be added later without having to modify (much) the implementation of the character classes (e.g., a new Constitution: invincible). Before implementing your design, justify your choice in the space below (no more). Then, implement your design based on the given code. ZIP it as "Question1.zip".



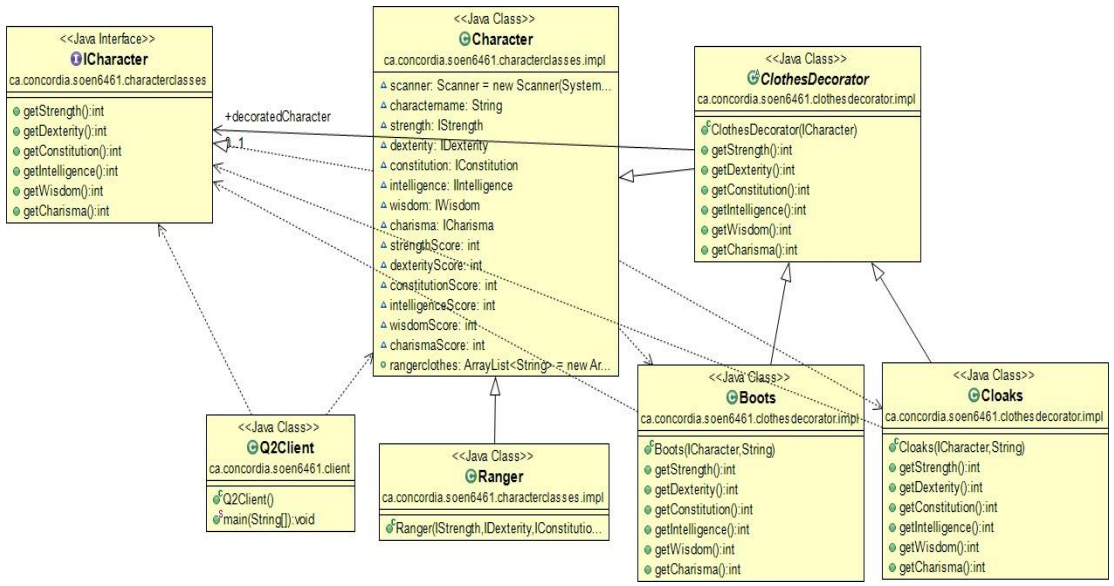
Clients shouldn't be responsible for creating instances of ability types objects such as `IStrength` type.

Factory pattern is decision making class that ensures to instantiate one of several possible subclasses of `ICharacter` based on inputs of clients.

Duck typing is used to favour composition over inheritance. Based on input of client, appropriate object of abilities (`Strength, Dexterity,..`) is composed in `Character` class. Duck typing also ensures to change the abilities at runtime. As I assume `BurlyStrength` might have its own operation to perform therefore any changes made to `BurlyStrength` won't affect `FitStrength`, `PlumpStrength` and so on. However, number of classes are increased.

Finally, Duck typing combined with factory pattern solve this problem.

Question 2 (30 pts): 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. ZIP it as "Question2.zip".



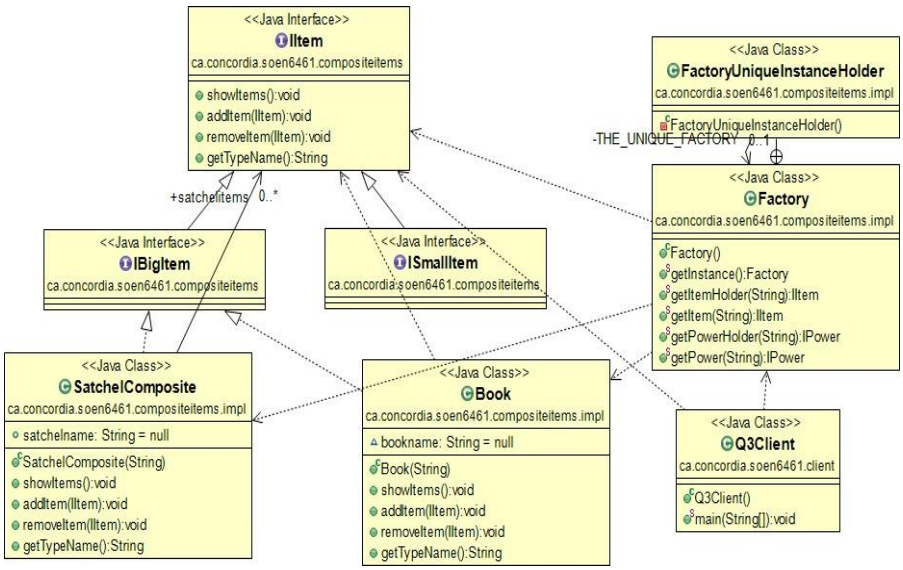
Proposed solution can be `Decorator` or `Extension object` pattern. `Character` will wear different clothing items runtime. Hence, our `character` object has to be decorated or `Character` has to provide extension object.

However, clothes are attributes of `Character`. It should modeled in `Character`. Extension object also does the same job as decorator but when it comes to return strength of decorated objects, decorator pattern seems to be a better candidate to implement because of recursive calls.

Also, when it comes to getting ability score `(Strength, Dexterity,..)` of character wearing `IClothes` decorator pattern does our job fine as `ClothesDecorator` will decorate objects of `ICharacter` and through one level of indirection, ability score `(Strength, Dexterity,..)` of `Character` is fetched.

As clothes are attributes of `Character`. `Character` will perform `setClothes` operation and ensure armour is worn on top of clothing items.

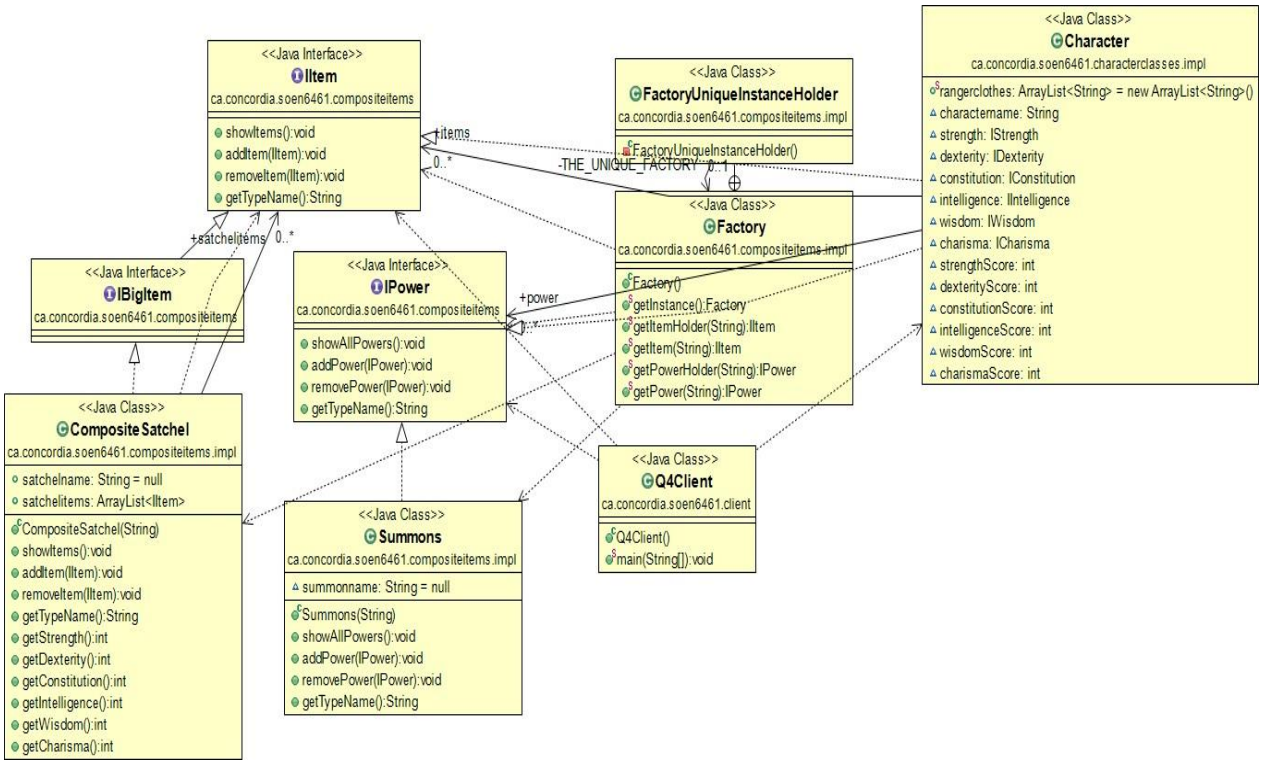
Question 3 (20 pts): Characters can carry various items in satchels or boxes: food items, books, gold coins, rings. Satchels are useful for food items, books, etc. while boxes protect gold coins, magical rings, etc. Separate for the given code, design and implement a hierarchy that offer satchels and 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. ZIP it as "Question3.zip".



Proposed solution can be `composite` or `decorator` pattern. `Character` will possess various items and therefore at runtime changes will be made, therefore decorated objects can be a good candidate to solve the problem. Decorators have the same supertype as object they decorate, but in our case `IItem` differes from `ICharacter`. The objects of `IItems` seems to have multiple hierarchies and will be composed of many objects. Also, we have to check whether the item holder (`Satchel, Box`) can add the Item or not. Since there is object hierarchy composite pattern seems better solution.

Using reflection, I enforced that certain items are placed into certain item holder (`Satchel, Box`) or exception will be thrown.

Question 4 (15 pts): 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. ZIP it as "Question4.zip".



Proposed solution can be `composite` or `extension object` pattern. Here, there is hierarchy of objects (`IPower`) same as previous problem. Also, characters wear items and possess powers therefore, we have to model `IItem, IPower` attributes for character. As these are `Character` attributes, extension object pattern isn't applicable.

There are two separate hierarchy such as `IItem, IPower` and using reflection I made sure to distinguish IItem, IPower. Trade off is that when the `IItem, IPower` will be bloated, then we have to implement `Visitor` design pattern.

Finally, In this design, Character will perform `addItems` and `addPower`. Composite objects are created through `Factoty`. Composite objects are added through Composites (`CompositeSatchel, CompositePower, CompositeBox`).

Question 5 (15 pts): 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

Here, counting innate strength is our concern. Innate strength for (question 1) plus (question 2) is counted using `decorator` pattern which makes sense because when we ask decorated object of `ICharacter` to `getStrength`, it will return sum of decorated object through recursive call since `clothesDecorator` will perform that job.

Possessed items and powers will also add some strength to Character. Therefore, all the leaves (`Food, Book, Spell, Rings…`) will return `getStrength`. Innate strength for (question 3) plus (question 4) can be counted using `CompositeSatchel, CompositeBox, CompositePower`. Our composites can be responsible for returning `getStrength`. The trafe off of using composite pattern is that if the `IItem, IPower` is bloated we need to implement `visitor` design pattern. Also, (question 3) plus (question 4) can be solved using extension design pattern. However, complexity is increased with extension design pattern.

As possessing items and powers are attributes of `Character`, through one level of indirection, `getStrength` will be returned.

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. ZIP it as "Question5.zip"