SOEN6461: Take Home Exam Designing and Implementing (Some of) Dungeon and Dragons Character Classes

Yann-Gaël Guéhéneuc
Opening date: 2020/04/23
Submission date: 2020/04/30

Terms and Conditions

a) Following material is permitted: lecture notes, Moodle material, StackOverflow, Academic Code of Conduct and Concordia University's Academic Integrity, any Web site about Dungeon and Dragons.
b) Online searching of subject material/related exam questions is NOT permitted.
c) Communicating with classmates regarding any aspect of the exam or course is NOT permitted.
d) Posting or sharing the exam content, including exam questions, or your answers both during and after submission is NOT permitted

Questions During the Exam

Do not hesitate to contact me via e-mail or Moodle at any time. I will do my best to answer any question **within 24h.**

Academic integrity

Concordia University takes academic integrity very seriously and expects its students to do the same. I understand that any form of cheating, or plagiarism, as well as any other form of dishonest

behaviour, intentional or not, related to the obtention of gain, academic or otherwise, or the interference in evaluative exercises committed by a student is an offence under the Academic Code of Conduct. I understand that the above actions constitute an offence by anyone who carries them out, attempts to carry them out or participates in them. By way of example only, academic offences include: Plagiarism; Contribution by a student to another student's work with the knowledge that such work may be submitted by the other student as their own; Unauthorized collaboration; Obtaining the questions or answers to an exam or other unauthorized resource; Use of another person's exam during an exam; Communication with anyone other than the instructor/invigilator during an exam and any unauthorized assistance during an exam; Impersonation; Falsification of a document, a fact, data or a reference. For more information about academic misconduct and academic integrity, refer to the Academic Code of Conduct and Concordia University's Academic Integrity webpage.

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".

Design Patterns: Visitor Design Pattern and Abstract Factory Design Pattern
Based on the question I decided to use Visitor and Abstract Factory Design Patterns. Following one by one I will illustrate in details:
Visitor:

- For starters, I would go with implementing **Visitor Design** pattern and the reason is that, based on our lecture notes "Visitor lets you define a new operation without changing the classes of the elements on which it operates."[1] And that is exactly what I need. I want to add some new values to my current values without having to modify much. (Considered Innate Abilities items in the question are referred to as Values)
- So my current values will be the target of this design pattern.
- Based on the Visitor UML, I need to have an **Interface** for my visitor, which will have all of my visit methods for each of my values. (In the implementation, I would have one concrete class for each of the: Strength, Dexterity, Constitution, Intelligence, Wisdom, and Charisma. By this I would have the ability to separate each value and easier to add new values later.) One class as the **Concrete Visitor** to implement visits and it would be the place to add new values. For **Element** I would create an interface just to have an accept method so inside all of the values I need to override this accept method. And my **Concrete Elements** would be my values.
- At the end, if I create an object form Visitor Interface and use the accept method, compiler would go through the explicit accept method, call the visit, and visit would add the new value.
- "By using the visitor design pattern: Decouple data structure and algorithms – Put the traversal in only one place, in the AST – Allow the addition of new algorithms without changing the data structure". [2]

Factory:

- As I have in the given code, I would need a general interface for gathering around all of the characters. The reason for this would be typing so all of my characters would be the same type and the main reason to have Abstract Factory Design Pattern. Based on lecture notes: "*Problem*: How to remove the dependency on the concrete implementation? *Solution*: Abstract Factory design pattern [3]. So there won't be any implementation exposure. "A system should be independent of how its products are created, composed, and represented – A system should be configured with one of multiple families of products."[4]

- Step by step based on our lecture notes and Abstract Factory UML, there would be a **Factory** class which has singleton pattern and values getters. The **General Interface** that I mentioned earlier would be the design pattern interface and at the end each of my characters would be my **Products**. All this are because of the "A family of related product objects is designed to be used together and you must enforce this constraint – You want to provide a library of products, and you want to reveal just their interfaces, not their implementations."[5]

Alternative Design Pattern: Extension Design Pattern
Trade of: More complex implementation – More messages exchanged.

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".

Design Patterns: Decorator Design Pattern and Template Design Pattern.
Decorator:

- Question is asking to add cloths to our characters so based on: "*Problem:* Add/modify the behaviour of some methods of some objects at runtime. *Solution:* Decorator design pattern."[6] Decorator Pattern would be the proper answer.
- I would start with decorator pattern and the obvious reason is the fact that in this step, layer by layer we are adding cloths to our characters.
- "The important aspect of this pattern is that it lets decorators appear anywhere. That way clients generally can't tell the difference between a decorated component and an undecorated one, and so they don't depend at all on the decoration."[7] relying on this would be one of the main reasons to choose this pattern because the characters would be dressed in order without client's intervention.
- "Decorator subclasses are free to add operations for specific functionality".[8]
- Based on the Decorator UML, there should be a **Component class** which would be our general interface which brings unity. Our characters would be our **Concrete Components** because they are the objects that we want to decorate. There should be an abstract class to play the role of **Decorator** so based on this class characters would be decorated and at the end, the **Concrete Classes** will be the cloths because they are the ones that I want my characters to get decorated with.

Template:

- At the end of the question, it is mentioned that we want to enforce a certain flow for decorating our characters. Based on "*Problem*: Let the framework decided when/how to call some user-defined methods *Solution*: Template Method design pattern". [9] the most appropriate pattern would be template pattern because, indeed it will force the flow without any input mistake to follow a certain order.
- "By defining some of the steps of an algorithm using abstract operations, the template method fixes their ordering, but it lets subclasses vary those steps to suit their needs." [10]
- Based on the Template UML, There will be one **Abstract Class** so it will carry the decorators and enforcing the order and **Concrete classes** which are implementing hooks methods.
- At the end, not only we are forcing the order this will also avoid any mistake in inputs.

Trade-offs: Like all design patterns, complexity

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".

Design Pattern: Composite Design Pattern

- Characters want to carry several items and we want to hide the implementation plus the confusion of adding the items based on satchels and boxes, so relying on "*Problem*: How to let client see similarly one sort algorithm or a set of algorithms? *Solution*: Composite design pattern" [11] composite design pattern would be useful because without any interference it will put the items in the correct place.
- "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly" [12] so this intention would give me the ability to help the client don't get involved with different types of items and treat all objects uniformly.
- "We must build the composite objects - Clients use these composite objects"[13]
- As we are responsible for creating the composites, I would follow the Composite UML step by step to have a clear order. **Composites** would be boxes and satchels because they are the one that we want to look at uniformly and **Leafs** should be each of the items, separately, that we can put in each of our composites.

Trade-offs: we should be careful not to have a bloated interface.

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".

Design Pattern: Abstract Factory Design Pattern

This question asked us to allow characters to have two new attributes and abilities. We can look at this question from 2 aspects.

- First, equally as the question number 3, this question can be a Composite design pattern. "*Problem*: How to let client see similarly one sort algorithm or a set of algorithms? *Solution*: Composite design pattern" [14] based on this context, by using Composite Design Pattern we get the ability to hide the implementation and remove the dependency between the client and implementation. In this scenario, client can look at the objects to carry and objects to possess in a same way and on the bright side, if we also combine the code from question number 3, we also get the specific items that are in bags and satchels, a grand gesture structure. We can look at the to carry and to possess items as our leafs so we can have packaging and a clear, straight forward implementation. Since we also need the Factory Design pattern to create an instance from our characters, at the end we would have a uniform way to create characters and have the carried and possessed items which would give us the combination of these two patterns.
- Second aspect, would be Factory design pattern, if we look at the question in a way that it is implying two new attributes for our characters, the Abstract Factory Design Pattern would be the proper answer. With Abstract Factory Pattern, we need to look at to carry and to possess as the characters abilities. In this scenario we need to get these new abilities as a list and since it is not mentioned in the question that we may add new items in future, we simply add them into our characters usual attributes. So whenever we create a new instance from our characters, these attributes would be a part of our character. In this way, expanding our code from question number one would be a wise choice. As we had before, there would be a **Factory** class which has singleton pattern and values getters and for now we would add two new list in it, to carry and to possess, which in our case since they are not going to be changed and they are similar for each of the characters there is no need to get an input for these two. The **General Interface** would be the design pattern interface and each of my characters would be my **Products** which now have two new abilities.

The reasons behind choosing these two aspects are completely clear. "Using the composite design pattern let clients treat objects and composition of objects uniformly" [15] so there is no need for the client to be cautious about the differences and the compiler would get the separation done. Also, by having Abstract Factory Design Pattern, we avoid exposing our implementation, "Provide an interface for creating families of related or dependent objects without specifying their concrete classes" [16] and that is exactly what we need. At the end, the client would only create an instance from one of our characters and the instance would have all of the needed attributes and abilities or if we go with the Composite Pattern, the leafs would be there so the client can easily understand their differences without any interference.

Trade-off: while using Composite Design Pattern we should always be careful about the bloating interface. Since we are using design patterns it is necessarily to always consider the complexity that we bring into the code, gives us the appropriate flexibility because otherwise it is not the best approach.

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 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".

For this question, I am considering not to use any kind of design patterns. While I was studying the lecture notes, the reason that we use any kind of Design Pattern is the fact that, we want to have more flexibility and for doing that we are adding complexity to our source code. Considering Design Patterns, are "a means to enhance the reusability Of the code written using the pattern + Its flexibility and a means to encapsulate design experience" [17], in my opinion, until now, step by step, for each part we implemented the needed design pattern and now that we reached a place to combine all these things together, implying another Design Pattern, would just add more complexity to our code without adding that much flexibility. "Trade-offs of (most of) design patterns:

- Flexibility
  - ❖ Favouring composition over inheritance
- Complexity
  - ❖ More objects interacting
  - ❖ More messages exchanged "[18]

Based on the above, I think by composition, typing, creating objects and connection between the classes, without any further complexity we could gather all the abilities, cloths, and attributes together and return a proper number to show the character's appropriate number.

Alternative: Since it is mentioned in the question that we need minimum changes when a specific rule is changing, maybe using a Visitor Design Pattern could be the other option.

Trade-off: Since I am not using any design pattern for combining everything, I think it could reduce the code flexibility.

I have neither given nor received unauthorized aid on this exam and I agree to adhere to the specific Terms and Conditions that govern this exam.

Name	Hasti Tajdari
Student ID	40087702
Unique e-mail ID	hasti.tjd@gmail.com
Signature	

References

1- SOEN6461: Software Design Methodologies Some Theory and Practice on Patterns – In Practice - Yann-Gaël Guéhéneuc – Page 92
2- SOEN6461: Software Design Methodologies Some Theory and Practice on Patterns – In Practice - Yann-Gaël Guéhéneuc – Page 110
3- SOEN6461: Software Design Methodologies, The Abstract Factory DP - Yann-Gaël Guéhéneuc – Page 12
4- SOEN6461: Software Design Methodologies, The Abstract Factory DP - Yann-Gaël Guéhéneuc – Page 21
5- SOEN6461: Software Design Methodologies, The Abstract Factory DP - Yann-Gaël Guéhéneuc – Page 21
6- SOEN6461: Software Design Methodologies The Decorator DP - Yann-Gaël Guéhéneuc – Page 5
7- SOEN6461: Software Design Methodologies The Decorator DP - Yann-Gaël Guéhéneuc – Page 6
8- SOEN6461: Software Design Methodologies The Decorator DP - Yann-Gaël Guéhéneuc – Page 11
9- SOEN6461: Software Design Methodologies The Template Method DP - Yann-Gaël Guéhéneuc – Page 6
10- SOEN6461: Software Design Methodologies The Template Method DP - Yann-Gaël Guéhéneuc – Page 11
11-SOEN6461: Software Design Methodologies The Composite DP - Yann-Gaël Guéhéneuc – Page 8
12-SOEN6461: Software Design Methodologies The Composite DP - Yann-Gaël Guéhéneuc – Page 9
13- SOEN6461: Software Design Methodologies The Composite DP - Yann-Gaël Guéhéneuc – Page 20
14- SOEN6461: Software Design Methodologies The Composite DP - Yann-Gaël Guéhéneuc – Page 8
15- SOEN6461: Software Design Methodologies The Composite DP - Yann-Gaël Guéhéneuc – Page 9
16- SOEN6461: Software Design Methodologies, The Abstract Factory DP - Yann-Gaël Guéhéneuc – Page 20
17- SOEN6461: Software Design Methodologies Some Theory and Practice on Patterns – In Practice - Yann-Gaël Guéhéneuc – Page 15
18- SOEN6461: Software Design Methodologies Some Theory and Practice on Patterns – In Practice - Yann-Gaël Guéhéneuc – Page 175