ACTIVITY REPORT

Problem Statement: Item Authoring

Problem Description: The authors, having logged in to the system are supposed to create items and subsequently save it in a database.

Design Patterns Used: Singleton Pattern, Adapter Pattern, Decorator Pattern, Factory Method Pattern.

Singleton Pattern:

- **Motivation:** The questions need to be saved to a database, and this shall be required for every author that has logged in to the system. However, there are two things to notice here:
  - A connection is only required when the item is to be saved (which is a very small fraction of duration in the authoring process).
  - Opening and Closing database connections is often very costly, and hence, doing it repeatedly is sub-optimal.
- **Design:** We maintain a set of open connections (as a **Singleton** registry), and for every user that requires it, one of the free connections can be provided from this. The registry is shared across all the users, and after usage, every user explicitly releases the connection.
- For the purpose of our demonstrative code, the registry of connections is simply an ArrayList of booleans, where True refers to a connection being free, and false refers to it being under use. This is implemented in the *Database.QuestionDatabaseConnectionPool.java* class. The *getConnection()* method either allots a free connection to a user, and then returns with its index, or returns with -1. Everytime the user requires to save an item, an infinite loop is run which attempts to fetch for a free connection, if a free connection gets allotted, the control-flow breaks out of the loop.

Adapter Pattern:

- **Motivation:** Every Item has two parts, *Question* and *Answer*, and both of these have predefined interfaces. The adapter is required for every subsequent Item type added as a plug-in, such that they can be made to work with these predefined interfaces.
- **Design:** To this end we use **class adapter**, which adapts the **adaptee** to target by committing to a concrete adapter class. This design uses multiple inheritance, where the **adapter** inherits from both the **target**, and the **adaptee**. We use two adapters for each plugin-type, one for *Question* and another one for *Answer*.
- For the purpose of our demonstrative code, we shall discuss any one Adapter implementation (as all others are essentially identical). Suppose we have *MCQQuestionAdapter*, which extends *MCQQuestion* (**adaptee**) and implements *Question* (**target**), such that both can be made to work together. The class diagram for the same is as follows. Similar strategy is adopted for *MCQ2Answer, MCQ2Question, ComprehensionAnswer*, and *ComprehensionQuestion*. For the case of Comprehension however, a **composition adapter** (instead of **class adapter**) has been used (the **adapter** in that case has a '**has-a**' relation with the **adaptee**, i.e., *Comprehension*)

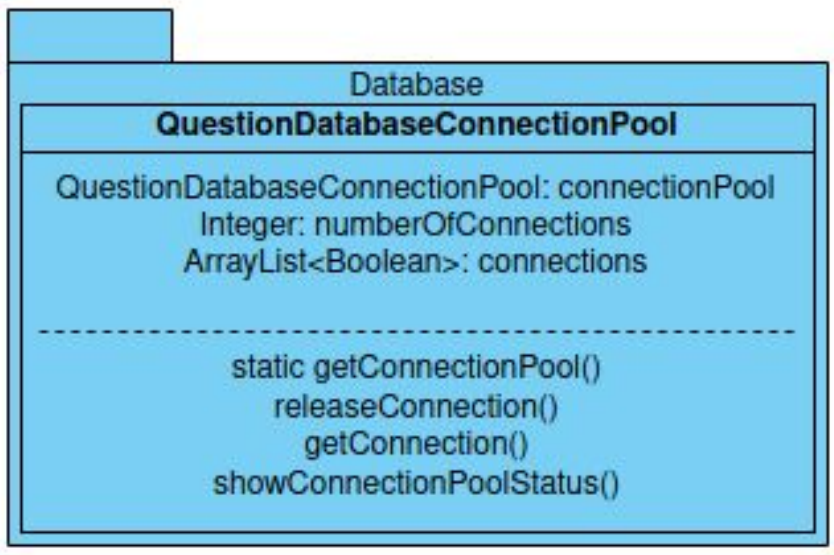
Decorator Pattern:

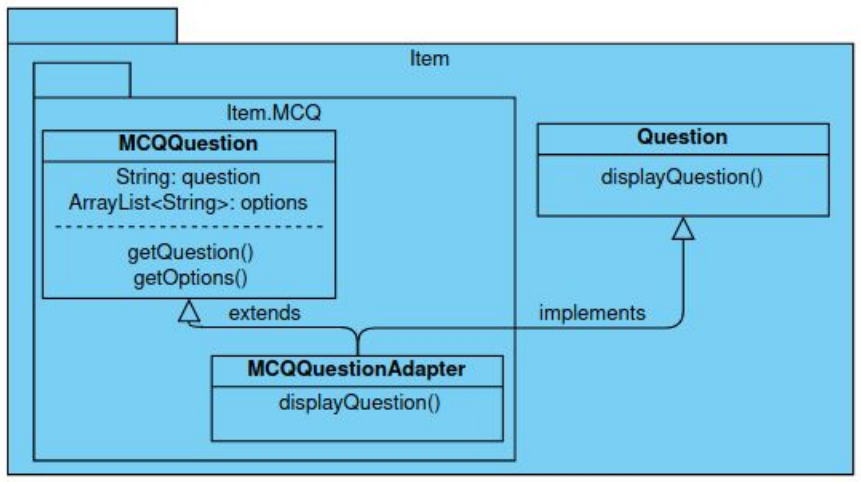
- **Motivation:** We have three kinds of plug-ins for demonstration, MCQ - single choice correct answer, MCQ2 - multi-choice correct answer, and Comprehension - a paragraph followed by a number of MCQ and MCQ2 type questions. The kind of MCQ and MCQ2 type questions associated with a Comprehension question is to be decided dynamically (during the Item authoring phase). Also, catering statically for all the different possible combinations of the same can very easily lead to a blow-up of subclasses.
- **Design:** To this end, we implement the Comprehension plug-in using the **decorator** pattern. The **decorator** pattern provides a flexible alternative to subclassing for extending functionality dynamically. This allows for the addition of an arbitrary number of questions (either MCQ or MCQ2), where the decision making is done at run-time.
- For the purpose of our demonstrative code, all the relevant implementation resides in the Item.Comprehension package. The **component** is *Comprehension,* **concrete component** is *Paragraph,* **decorator** has been named as *decorator,* and the **concrete decorators** are *AddMCQ* and *AddMCQ2*. The class diagram for this is as follows.

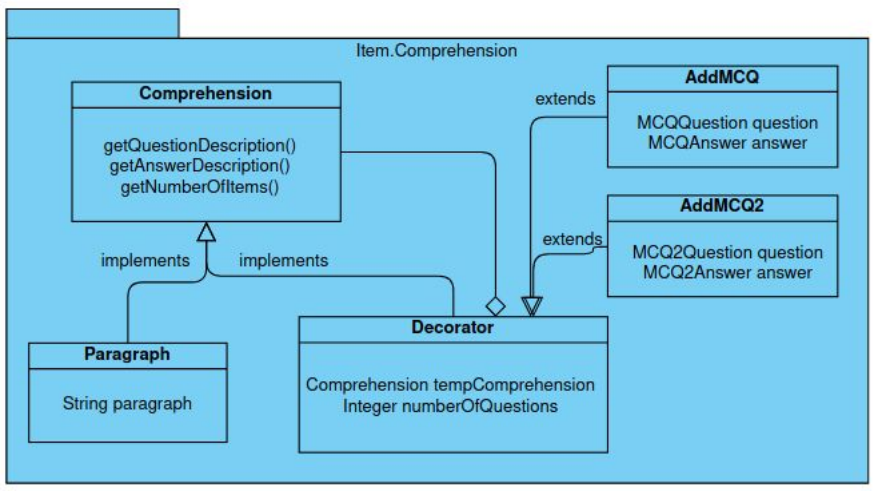
Factory Method Pattern:

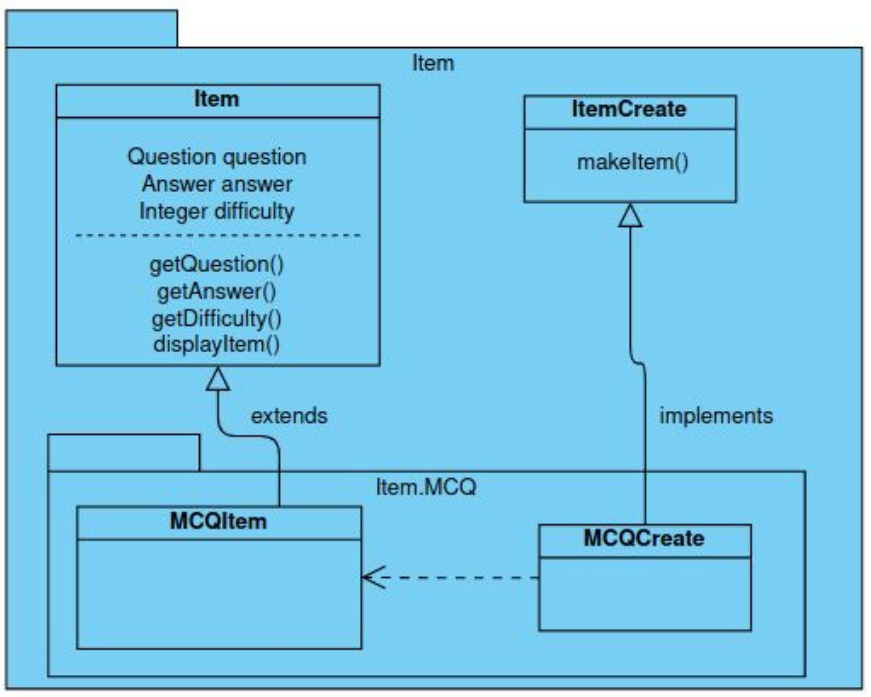
- **Motivation:** We create three kinds of items, MCQ, MCQ2 and Comprehension, and the choice of item to be created is decided dynamically at run-time (by the author). Hence, we need a pattern to provide us with flexibility for creation, and also for managing the creation of all the different types of items.
- **Design:** To this end, we use the **Factory Method** pattern for item creation. It defines an interface for creating an object, but lets the subclasses decide which class to instantiate. This brings all the creation logic at one location (in our case, namely the *ItemFactory* class, in the *Item* package), and also allows for dynamically deciding which subclass to instantiate.
- For the purpose of our demonstrative code, we have an abstract class *Item* (which is equivalent to the **Product**). This is extended by all the concrete Item classes (*MCQItem, MCQ2Item,* and *ComprehensionItem*) which form the **concrete products**. Then, we have our *ItemCreate* interface, which acts as the **creator**, and the *MCQCreate, MCQ2Create* and *ComprehensionCreate* are the corresponding **concrete creators**. For the sake of demonstration, we only show it with respect to one **concrete product**, and one **concrete creator** class, as it is essentially analogous for the rest.

Class Diagrams









- **NOTE: To save space, in the class diagrams, functions/fields that get carried over from the inherited interface/class, are not shown.**
- **All the notations/terminologies (Component, Adapter, Adaptee, Creator, etc) have been taken as-is from the 'Gang of Four' book.**
- **Instructions to run the code:**

  Since the item authoring module is interactive by nature, there are two demonstrations prepared for the same:

  1. Interactive: execute **./IMT2017003.jar** (a dummy set of input has been provided in the inputFile.txt - present in the jar-file)
  2. Dummy Run: **./dummyrun.jar -** present in the jar-file (almost mimics what the above dummy set of input would do otherwise.)