Design Question 1

Factory Method

Step 1: Explain in words how the pattern works (in at least five sentences). Include any roles that classes play.

Say we need to implement a class in a framework or a library that is used by others, we should consider "hiding" the constructor by making it private or protected. Instead, we can define a factory method (maker) in such class to make the constructor call to return a newly created object. As a result, no one can call the "new" operator from an external class, and those developers can only use a unified interface (the factory method) to instantiate new objects. There are more benefits when we have more than one such class. We can define a static method in a superclass to instantiate such objects, unifying the maker interface into one and letting the superclass to decide which constructor to call and which type to return. Without the "harmful" "new" operator, the developers should be able to write clean code and avoid creating wrong objects if no proper documentation is in place; instead, now the overhead of such checking is transferred into the factory method in the superclass.

Step 2: Discuss how this design pattern could be implemented in part of your project. If it isn't applicable, explain why and suggest an alternative design pattern that you might be able to implement instead.

In our project, the children classes of Message class can use this design pattern. They include TextMessage, ImageMessage, and ActionMessage. Each class shares very similar instance attributes, such as the "Sender" of the message, which is a User, and also the timestamp of when the message is sent. Currently they define their own constructors, and our developers need to explicitly call the "new" operator to create a new object of them.

Instead, we can make the constructors of Message, TextMessage, ImageMessage and ActionMessage all private. We should define a factory method in each class except the superclass Message. The method can be named "createMessage()", which calls the private constructor of its own. Even better, we can make that method protected so that it is only callable in the superclass Message. Now we define another factory method in the superclass, which takes a sender User, a sending timestamp, and also the actual message which can be the "Object" type, as the arguments. In the factory method, we can check the type of "Object" and decide which factory method of the subclasses to call.

In that way, we save the overhead of educating the other developers in our group about which object to create with. Rather, they can just call the public static factory method in the Message class to create an object with very similar interfaces to other subclasses. That should accelerate our development process and my teammate developers can focus on their own assigned classes.