CSE 443 Object Oriented Analysis and Design Midterm Report

Ahmed Semih Özmekik
171044039

December 8, 2020

Abstract

Detailed explanation of design choices with class diagrams along with the design patterns used in the homework.

1 Question 1

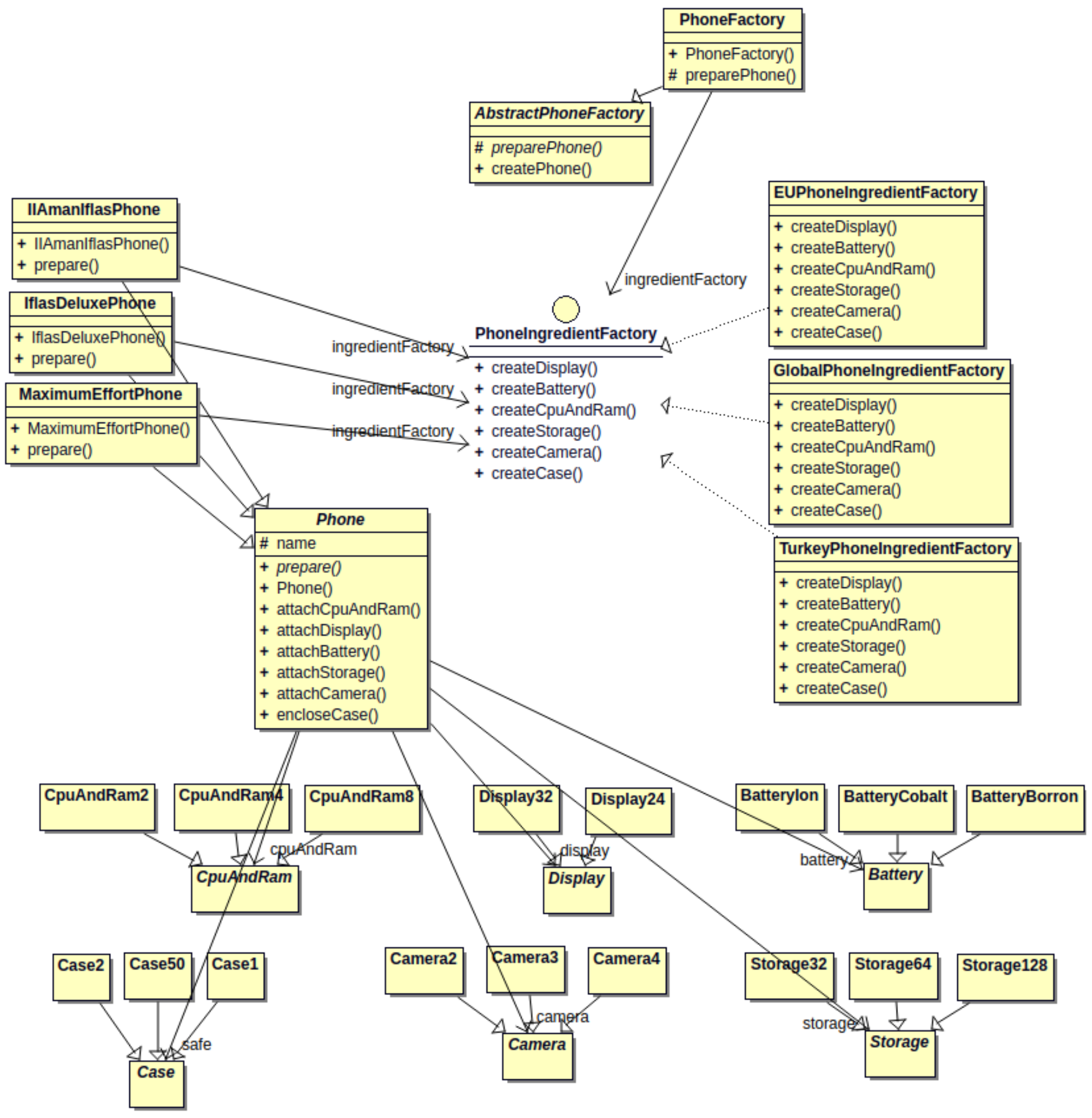

Figure 1: Question 1 Design Class Diagram

In this question, phone factories, different phone models and different details applied to these models in different countries, the application of the abstract factory design pattern is inevitable as it is already requested in the homework. So we have implemented a simple implementation of this pattern.

Besides, it should be added that, as can be seen in the diagram, we proceeded by choosing to derive a class from a base class for each ingredient. This may be seen as an overhead design choice. But we have to realize in this question that many classes actually make an imitation. In other words, in order to realize the abstract factory design pattern, we are faced with an example that is not directly realistic, just like in the book, that simulates the situation. Therefore, all of these ingredient derive classes do not define a new behavior, and they even specify almost no additional attributes. In a real design, we would not make such a choice. But we chose to do it this way, by going parallel with the example in the book.

Apart from that, everything about the Abstract Factory Design Pattern can be seen in detail in the diagram.

2 Question 2

This is an example of the need for an Adapter Design Pattern that is very useful, naive and simple. In this example, since the *TurboPayment* interface exists as *bytecode*, we also wrote it as an interface to verify the situation. And we're adapting this old *TurboPayment* interface to our new interface.

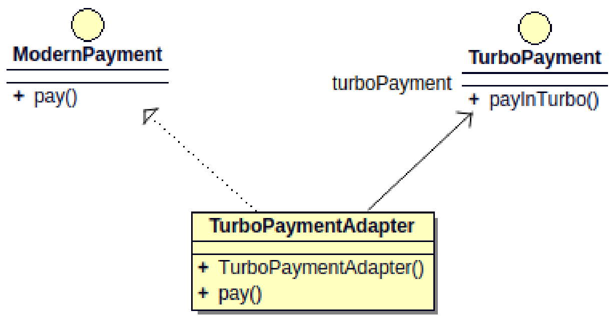

Figure 2: Question 2 Design Class Diagram

3 Question 3

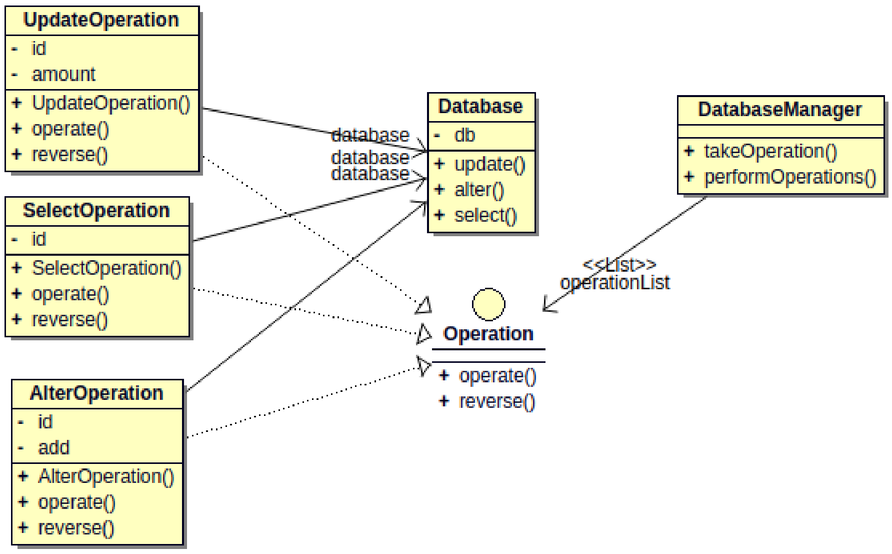

Figure 3: Question 3 Design Class Diagram

Design Solution

In this example, we used the *Command Design Pattern.*

We could also use the *Memento Design Pattern,* which we noticed as a result of our research and may be used again for the example here. Let's detail our design by explaining why we chose *Command* in our selection with a comparative method.

First of all, the important thing for us is that a query sequence will be executed on the database and if the array fails at any time, the entire array will be rollbacked. This is an important detail because the design revolves around solving this problem.

At first glance, we envisioned that we could solve it in two ways: First, either we had to hide the commands we executed, and when there was an error, we would rollback those commands we executed and kept in reverse order in reverse order. Second, we would save a state of a database in every transaction and return to the state we wanted in case of an error. The first represents the Command Design Pattern and the second represents the *Memento Design Pattern.*

Although this example is in the form of a simulation, if we were working in a real database, we would not have preferred the second one, considering the size of the database. Because getting a record of a database in every process is inefficient in terms of memory. Maybe it would provide the efficiency of being able to instantly return to that state, but it wouldn't be a huge profit besides this memory inefficiency.

Therefore, when we go with the first selection, namely *Command Pattern*, we do the operation by taking a sequence of commands and then applying them all. And when there is a problem, we can undo it with a reverse method for the operation we define in our interface.

Rollback of Transactions

In rollback of transactions, simple procedures take place. As mentioned, we put the commands in the script given to us sequentially, each command sends a variable in *boolean* format, as defined in the interface. If it sends *true*, then there is an error. If there is an error, we stop applying the operations at that moment and we end up with the applied commands that we kept in the *Stack* data structure up to that point, by pop operation, that is, by applying the reverse order and reverse operation. Thus, when the process is terminated, database rollback is made.

4 Question 4

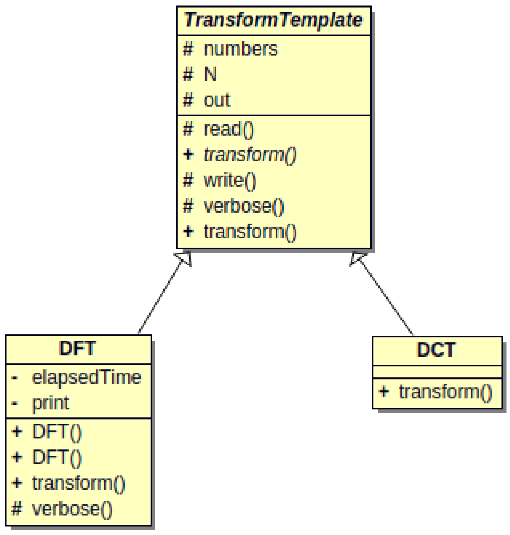

Figure 4: Question 3 Design Class Diagram

In this last example, there is a simple implementation of the Template Design Pattern.

As stated in the homework report, we have a template. And it is implemented in a fixed order by the procedure. Only the transform operation may differ.

Since it is a problem that revolves around these assumptions, it is inevitable to use Template Design Patterns.

To mention something specific to our design, we do not leave *read* and *write* methods *abstract*. So, we could not define these methods in the *template*, but define them to subclasses. But it would be a big code duplication, because the input doesn't change at all. Therefore, read will always remain the same. But of course we did not define it as a *final*, because maybe the person who will transform may want to read in a very different way. Only the *numbers* variable needs to satisfy the condition that it does not leave *null* after reading, which is already done by the *template*.

Apart from that, the *write* method was also defined in the template, not defined as *final*. The *write* method could actually be different for both transform methods. But with choosing a better design, we expect the user to fill in only one string variable in the *transform* method, and then write this string to the file at once in the write method. Therefore, our design has a very generic structure in this state.