BCPR301 Assessment 3

# Self-Marking

There are /50 marks in total

## 1.Identifying N design patterns used in the frameworks in this assignment (/10 marks)

|  |  |
| --- | --- |
| 1)The name of the design pattern identified | (N marks) |
| 2)The locations of code involved in the framework | (2 \* N marks) |
| 3)The class diagram of the design pattern identified. All the components in the design pattern class diagram provided in our textbook should be explicitly labelled in your class diagram. | (2\*N marks) |

## 2. Applying N design patterns in your assignment 2 solution (34/40 marks)

|  |  |
| --- | --- |
| 1) The class diagram before your modification | (0 marks) |
| 2) The locations of code involved in your assignment 2 solution | (4 marks) |
| 3) The name of the design pattern applied | (4 marks) |
| 4) The reasons why applying this design pattern is suitable | (4 marks) |
| 5) The class diagram after your modification | (4 marks) |
| 6) Applying the design pattern proposed | (18 marks) |

1. Design pattern identified

Name

Location

Class diagram

Name

Location

Class diagram

2. Design pattern applied

# Builder

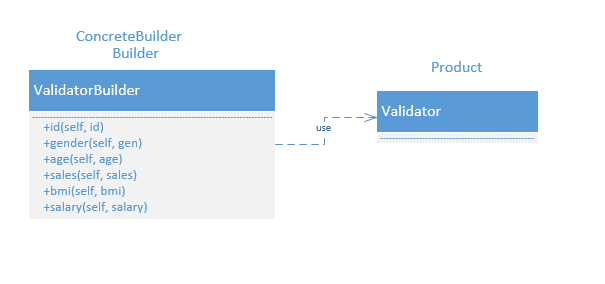
**Class diagram before**

**Location**: Validator.py / ValidatorBuilder, main.py

**Name**: Builder

**Reason**: The reason why I apply this design pattern is that the system can easily modify the existing check\_rules in the future by creating new validator object. For example, if we need to change salary rules in the future, we can use ValidatorBuilder to build a new validator with new salary rules. And then in main.py refer the validator to the new one. If we don’t have builder design pattern, we have to go and check code inside Validator class \_\_init\_\_method and edit the \_\_init\_\_ method. We should not change existing code in the future, it breaks the open-closed principle. After we modify it, we lost the default salary rules forever. But with a ValidatorBuilder, we can set default values for each rule. The system therefore is more stable and have better extendibility.

**Class diagram after**:



# Strategy

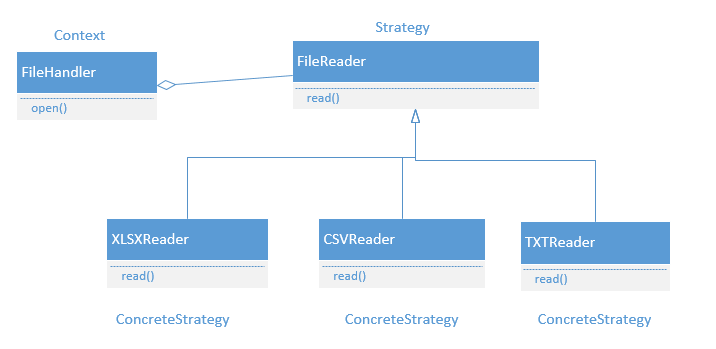
**Class diagram before**

**Location**: file\_handler.py / FileReader, CSVReader, TXTReader, XLSXReader. And file\_handler line 6-12

**Name: Strategy**

**Reason**: In the old design, each file reader has own algorithms, we have to call algorithm by method name. If we can make them interchangeable, it will make the system much simpler. Therefore, I use Strategy design pattern to make each file reader algorithms encapsulated and interchangeable with each other. The client FileHandler don’t need to know what’s inside file reader. File reader is independent from client that use it.

**Class diagram after**



# Factory Method

**Class diagram before**

**Location**: file\_handler.py / FileReader.open, file\_reader.py / CSVReader, TXTReader, XLSXReader

**Name: factory method**

**Reason**: The problem in assignment 2 is that we don’t know what file type the FileHander is going to open in advance. We only know the file type at run time when user inputs the file name. This situation is best solved by factory method design pattern since it can take an input and output the right object. The open method in FileHandler returns a new object depending on the file extension of input. Therefore, the system is more extensible. A new type of file can easily add by adding new file extension and its related Reader class.

**Class diagram after**

