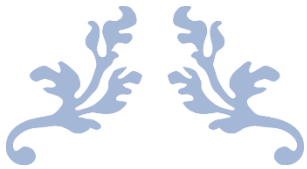

OU2

Application Development (Java)

CS-User: ens20ghn
Gazi Md Rakibul Hasan

DECEMBER 5, 2023

Introduction

A design pattern addresses a frequently encountered issue in an application by providing a reusable solution. Design patterns provide a systematic and practical approach to resolving repetitive problems developers face while creating software. Using design patterns, we implemented multiple applications as part of that assignment. We utilised design patterns including builder, abstract factory, memento, and iterator. The assignment contains four tasks, all of which we have successfully addressed by applying design patterns. The tasks are described below.

Task 1: Builder Design Pattern

Builder design pattern is a creational design pattern in object-oriented programming. The main objective of that design pattern is to create a complex object step by step, and the final step returns the Object. Builder pattern is useful when the creation of the Object takes many parameters in the constructor to instantiate. For that assignment, we constructed hamburgers using the builder design pattern. A hamburger contains various components: meat, bread, vegetables, and sauce. We will use the builder design pattern to construct a hamburger. Figure 1 illustrates the relationships that exist between those classes. Initially, we will create a package **se.umu.cs.apjava.maxdonalds.Burger** and all classes that are included in task 1 will be created within that package.

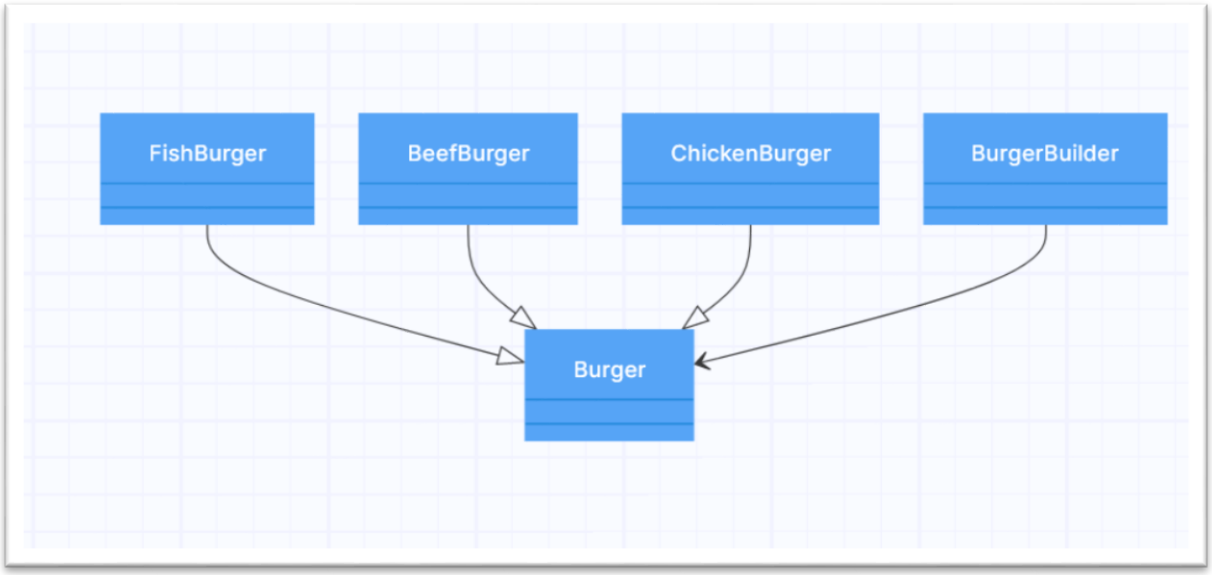

Figure 1: UML class diagram for the classes that created hamburgers using the builder pattern.

Burger

The class will be considered an abstract product. The class provides all the methods that are generally applicable to all burgers. The abstract class includes common characteristics of burgers, including vegetables, sauce, and total price. The class also includes two methods: getCost() and getDescription(). The Burger class is responsible for representing the fundamental properties of a burger.

Concrete Products

The classes Chicken Burger, Fish Burger, and Beef Burger will inherit from the abstract class Burger. The classes will be considered concrete products. These classes will implement the methods inherited from the Burger abstract class. Classes of concrete products are responsible for creating specific types of burgers.

Burger Builder

The Burger Builder class is utilised for the creation of burgers. The class provides the user with the flexibility to construct hamburgers. The user can specify the desired type of Burger using the setMeat() method. Subsequently, the user can incorporate vegetables and sauce into the Burger by employing the corresponding add() methods for each component. The class contains a method that can be used to obtain the final product. The Burger Builder class is responsible for creating a burger object step by step.

Task 2: Abstract Factory Design Pattern

Abstract Factory Design Pattern is a creational design pattern. It is like the factory design pattern. In factory design patterns, the Object is created without exposing the creational logic. Abstract Factory Design patterns put another abstraction layer over the factory design pattern. In the following task, we have implemented the abstract factory design pattern. To implement the given task using the abstract factory design pattern, we will use task 1. The abstract factory design pattern includes four main components: abstract product, concrete product, abstract factory, and concrete factory. Task 1 requires implementing both the abstract product and the concrete product. We will implement a GUI that provides three common types of burgers. It will make it easy for the user to create those hamburgers with minimal interaction with GUI. Figure 2 represents the UML diagram over those classes essential to creating the Abstract Factory Design pattern.

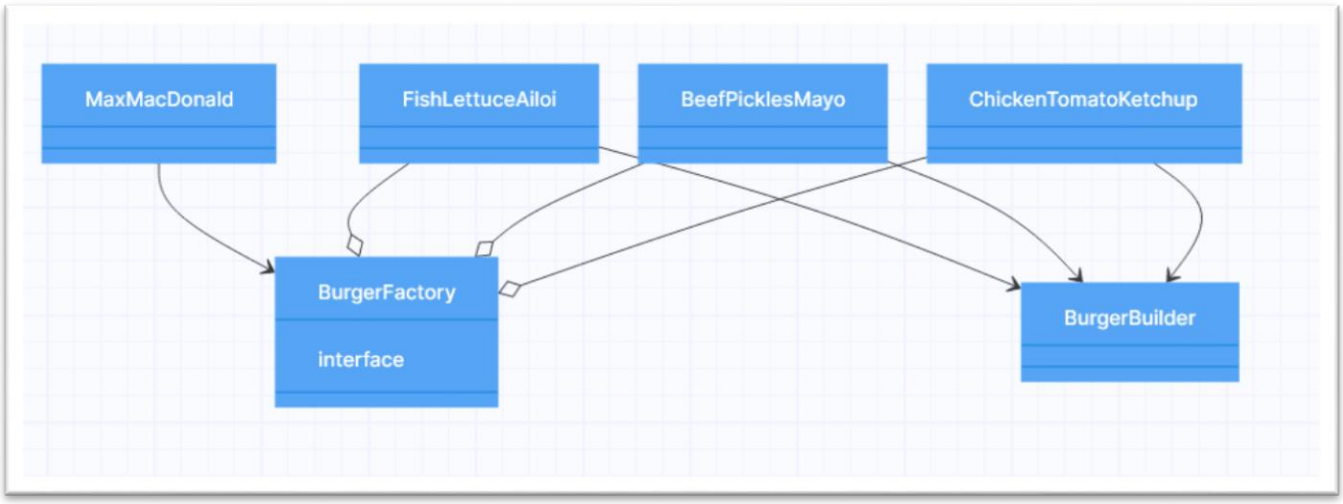

Figure 2: It represents the essential classes to implement the abstract factory design pattern. Those three common types of burgers implement the burger factory, and Max MacDonald's class is the abstraction layer of the Burger Factory.

Burger Factory (Abstract factory)

It is an interface class containing a method, and the method's name is createBurger(). Burger Factory will be regarded as an abstract factory class.

Chicken Tomato Ketchup, Fish Lettuce Ailoi, Beef Pickles Mayo

Each of these classes will implement the Burger Factory class. These classes will utilise the Burger Builder class from task 1. Burger Builder will build the hamburger by providing the appropriate class name for each meat, vegetable, and sauce type.

Max MacDonald (Client)

It takes a Burger Factory class's instance as a constructor parameter. The method orderBurger() will call the method createBurger() method from the Burger Factory class to create the Burger. Max Macdonald's class will be regarded as a client class. It will create a specific burger without exposing the details of the Burger's building's details construction.

Task 3: Memento Design Pattern

The Memento Design Pattern is a behavioural design pattern. It is used for the Object's undoable actions. It is helpful to maintain the Object's history. We will implement a notebook using a memento design pattern in the following task. The user can create notes, and each note is related to a specific ID number. After creating the note, the user will be able to write text on the GUI that will be related to a specific note. The user can save the note's text, and the user can restore the note's previous state text. There will be functionality to delete specific notes from the list. Figure 3 represents a UML class diagram of how classes relate to each other to build that Notebook application.

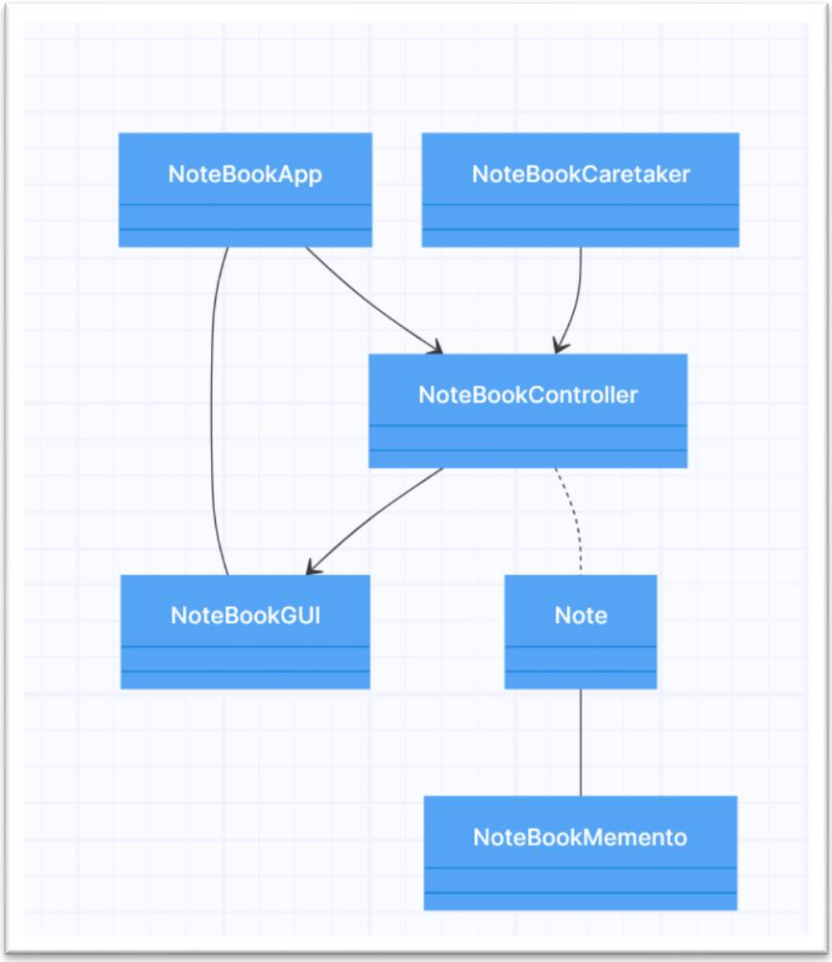

Figure 3: This shows how the classes relate to each other to build the notebook application using the memento design pattern.

Note (Originator)

That class will be responsible for creating a memento. Furthermore, there will be a method to get each note's ID and setter and getter for the note's text.

Note Memento (Memento)

The Note Memento class represents the state of the current notes. Each Memento class holds the string that holds the saved state's text.

Note Caretaker (Caretaker)

That class is responsible for saving the text and getting the previous state. That class holds the state of the Note Memento class. Since each note has a unique ID, we used HashMap to store each note and its saved state. We will use a stack data structure to save the note's states. The getMemento() method will be used to get a Memento of a particular note by using the note's ID number. Similarly, getMementos() will be used to get a List of Mementos of a specific note.

The responsibilities of the Notebook Controller class are to listen to actions for specific button clicks and respond to the user on the GUI. If the delete button is clicked, it will remove a specific note from the GUI. The text of the currently selected note will be saved by clicking the save button. When the restore button is clicked, the GUI will revert to the previous state of the note. Furthermore, a "create" button will create a note. Moreover, the Notebook GUI class is responsible for constructing the GUI.

To test whether the Notebook application works accordingly, we have tried to write some text in the text area. Figure 4 represents the test of how we have done it.

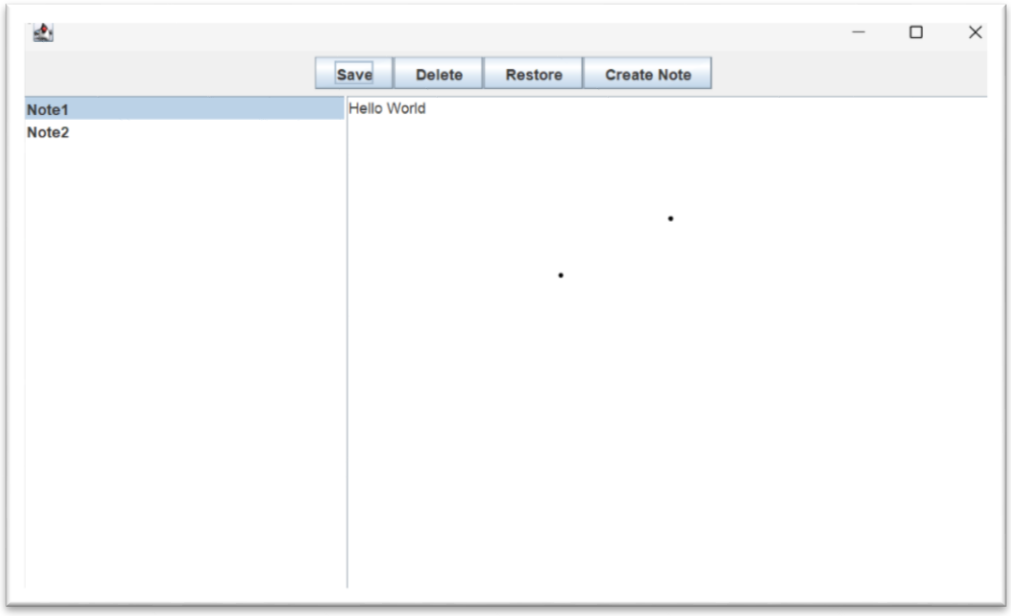

Figure 4: "Hello World" text has been written, and the Save button has been clicked for once.

After saving that text, we will write some text presented in Figure 5. Subsequently, we will click the restore button twice. Then, we will observe that the following GUI text will disappear, as depicted in Figure 6.

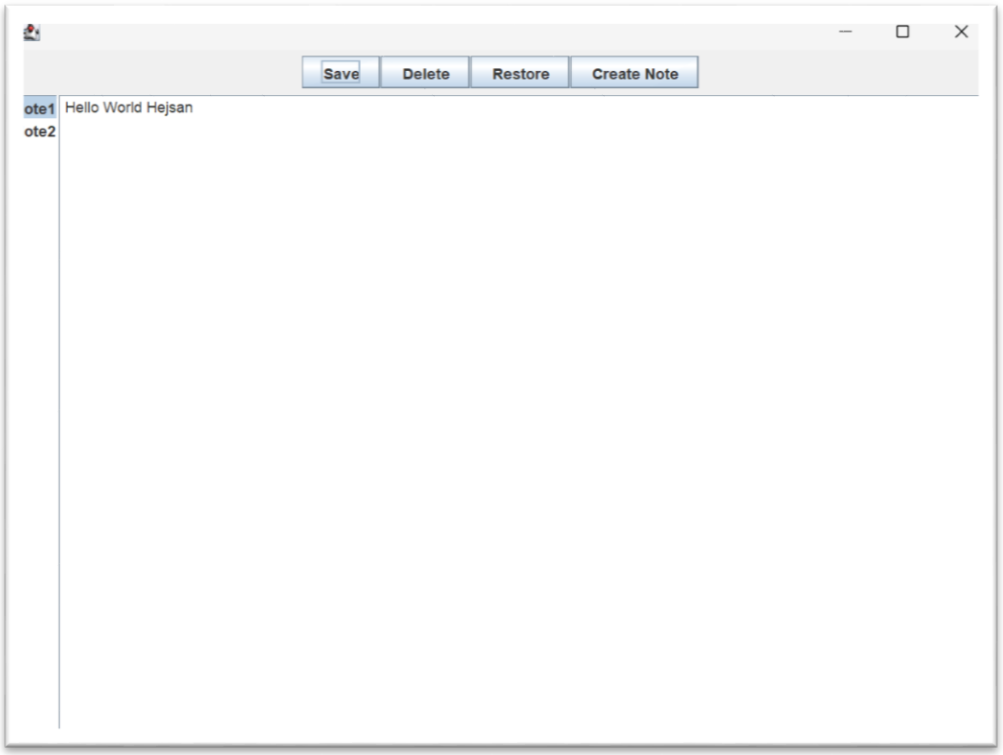

Figure 5: We have added some text after writing the first text, "Hello World," in the text area.

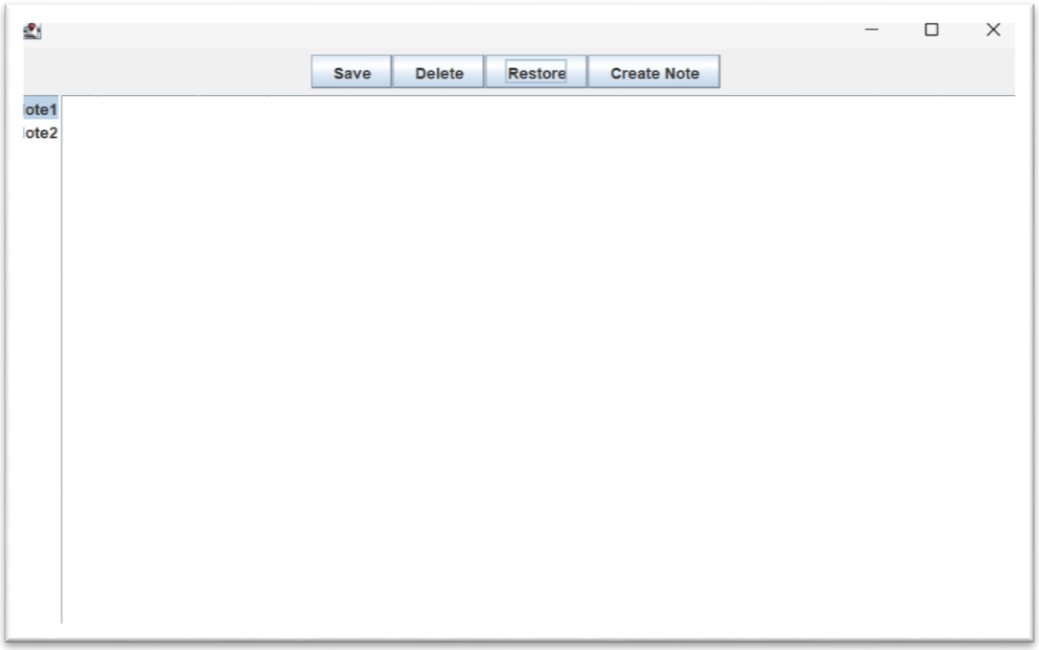

Figure 6: After clicking the restore button twice, the text from the text area will disappear.

Task 4: Iterator Design Pattern

The iterator design pattern is a behaviour pattern that allows the sequential traversal of a collection of objects while not exposing its underlying representations. In the following task, we will iterate over a linked list utilising the iterator design pattern. Figure 7 depicts the relationship of the classes essential to implement the task 4.

Figure 7: It shows how the classes relate to each other to implement task 4.

Iterator

It is an interface class with methods such as hasNext() and next(). Those methods will be helpful for the traversing of the linked list.

Linked List Iterator

That class implements the Iterator interface, and its constructor takes a list of objects as a parameter. An attribute observes the position of the elements in the list. hasNext() method checks if any element exists next to the current element. The next() method gives the current position's value and moves to the next position.

Aggregate

It is an interface class containing a method that creates an instance of the Iterator class.

Concrete Aggregate

That class implements the interface Aggregate. Moreover, that class creates a list of objects, and the add() method adds an element in a specific position in the list.

The Names class will first utilise the iterator design pattern, creating an instance of the Concrete Aggregate class. Then, we will use the add method to add objects in that class. In that case, we will add a string. Afterwards, we will create an instance of the Iterator class. That Object will be created by invoking the create() method from the Concrete Aggregate class. That Iterator class's instance will be used to traverse the linked list, which has been created within the Concreate Aggregate class.