In the design class diagrams of Wave, our group use three design patterns, namely visitor pattern, strategy pattern and command pattern. The detailed description of the design patterns is listed below.

# Strategy pattern:

In the “Sign In” use-case, we need to encrypt the passwords of users and organizations. But there may be different kinds of encryption methods. The algorithms vary. We use the Strategy pattern in this use case.

Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors. We need different variants of an algorithm. An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures. A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

In this use case, we use an interface named “incode”. Two classes, namely “incode mehod1” and “incode method2”, implement this interface. Then, the controller uses the interface to encode the passwords. More details can be found in the diagram.

As a result, this design pattern deals with families of related encoding algorithms and provides an alternative implementation to sub classing. This pattern also eliminates conditional statements.

# Command pattern:

In many use cases: “Manage Activity Information”, “Manage Org Information” and “Manage User Information”. There are three operations namely “add”, “delete” and “modify”. The parameters of the three operations are almost the same.

We want to parameterize objects by an action to perform , specify, queue, and execute requests at different times, support undo, support logging changes so that they can be reapplied in case of a system crash and structure a system around high-level operations built on primitives operations.

So we use the Command pattern. We use an interface, which has an operation named “execute”. The three entity classes, namely “add”, “delete” and “modify” realize the interface. The controller uses the “execute” of the interface to execute the commands.

As a result, command decouples the object that invokes the operation from the one that knows how to perform it. Commands are first-class objects. They can be manipulated and extended like any other object. Users can assemble commands into a composite command. It's easy to add new commands, because we don't have to change existing classes.

# Visitor pattern:

In the use case “Search”, there are different users. The admin, organization, user and guest can search the information. The results of different users are not the same. So in this situation, we decide to the Visitor pattern.

An object structure contains many classes of objects with differing interfaces, and we want to perform operations on these objects that depend on their concrete classes. Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. The classes defining the object structure rarely change, but you often want to define new operations over the structure

There is an abstract user in the class diagram, which is the father of the class “admin”, “user”, “org”. An entity class has the operation “search”, which can do different type of the search.

As a result, this design pattern makes adding new operations easy. A visitor gathers related operations and separates unrelated ones.