* 1. ***Design Patterns***

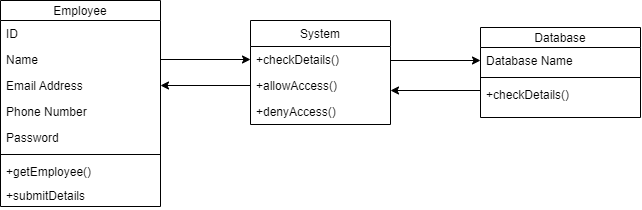
**Mediator**

The Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling. Mediator pattern falls under behavioral pattern category.

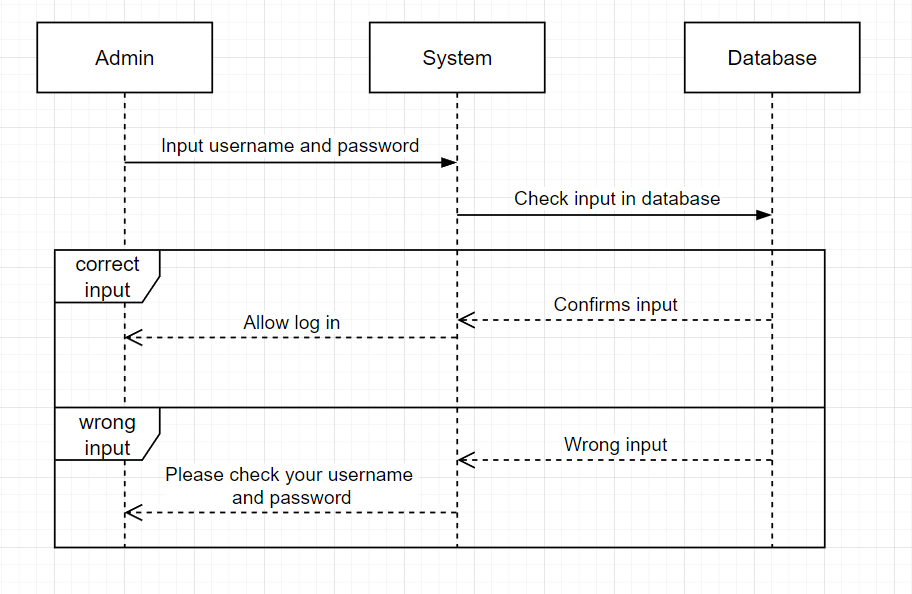
Application:

In this case, the system itself acts as the mediator between the user and the database. The system receives the input from the user and sends it to the database for checking. If the user’s credentials (username and password) are correct, then the user is given access. Otherwise, the user is denied access.

Implementation in Class Diagram:



Implementation in Sequence Diagram: (also applies to other Login sequences)



**Factory Method**

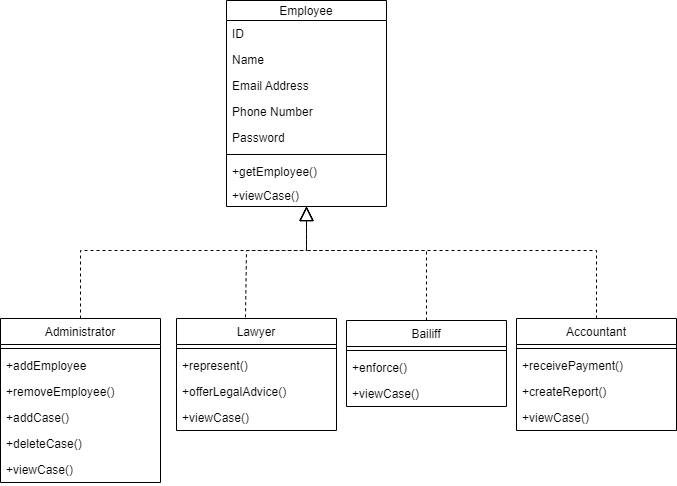
The Factory Method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

Application:

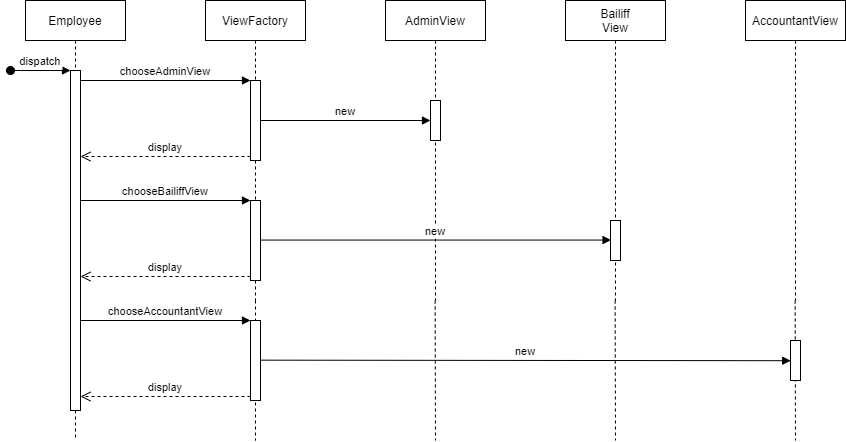
Different employees can all view the cases. But we may want to provide different views. For example, the accountant should not be able to view everything that the administrator can. This is a problem that can be solved with Design Patterns.

By using the Factory Method pattern, we create new subclasses that implement the same interface, but have a different implementation of the viewCase() method. This allows us to split the Employee class into subclasses, each of which can view different things by using the viewCase() method.

Factory Method implementation in Class Diagram:



Factory Method implementation in Sequence Diagram:



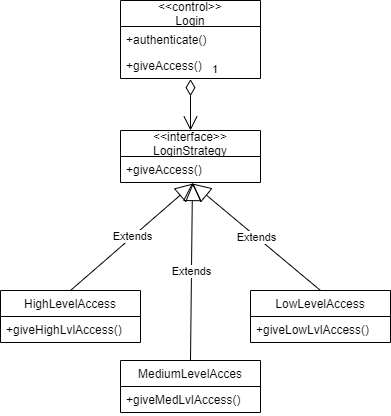
**Strategy**

The Strategy pattern is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use. Strategy lets the algorithm vary independently from clients that use it.

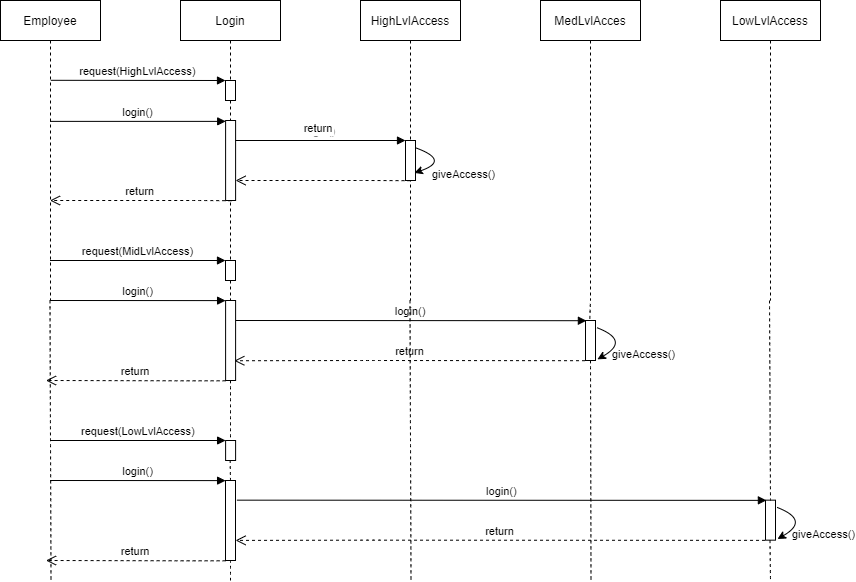
Application:

The Strategy pattern can be used in this project for dealing with user login. As different users will have different access levels, we can implement the Strategy pattern to split the login process into 3 separate algorithms, each of which gives the user a different access level. After the correct algorithm is chosen, only the needed one is executed at runtime.

Strategy implementation in Class Diagram:



Strategy implementation in Sequence Diagram:

******