**Design patterns & Solid Principles.**

**Solid Principles:-**

**S**ingle Responsibility Principle (SRP)

**O**pen/Closed Principle (OCP)

**L**iskov Substitution Principle (LSP)

**I**nterface Segregation Principle (ISP)

**D**ependency Inversion Principle (DIP)

**SRP-**

states that a class should have only **one reason to change**, meaning it should focus on a single responsibility or functionality. This principle ensures that the class is easy to maintain, test, and understand.

Imagine we are creating a program to manage employees. We have a class that handles employee details, saving employee data to a database, and generating reports.

Issues with This Design:

**Multiple Responsibilities**: The Employee class handles:

* Employee data management.
* Database operations.
* Report generation.

This violates SRP because:

* Changes in database logic will affect the Employee class.
* Changes in reporting logic will also affect the Employee class.

We fix this by refactor the code by separating responsibilities into different classes.

**(OCP)-**

is a design principle in object-oriented programming that states:  
**"Software entities (classes, modules, functions) should be open for extension but closed for modification."**

This means:

* You can add new functionality to an existing class without altering its existing code.
* This minimizes the risk of introducing bugs in the existing functionality and makes the system more maintainable.

we should wright the code in such a way that if we need to add a new functionality we shouldn't modify the code instead we should wright it in a separate class and just extend it.

**LSP-**

**"Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program."**

In simpler terms, any subclass should be able to stand in for its superclass without altering the desired behaviour of the program.

Eg: <https://github.com/MohammadAasimB/dp-sp/blob/main/Liskov%20Substitution%20Principle%20(LSP)>

**ISP-**

In Interface Segregation Principle **"A class should not be forced to implement interfaces it does not use."**

In simpler terms, interfaces should be small and focused on specific tasks. If an interface has too many methods, classes that implement it may end up with unnecessary methods they don't need.

Eg: <https://github.com/MohammadAasimB/dp-sp/blob/main/Interface%20Segregation%20Principle>

**Design patterns: -**

**Creational-**

Creational patterns deal with the process of creation of the objects of classes.

**Structural-**

Deals with how classes and objects are arranged or composed.

**Behavioural-**

Describes how classes and objects interact and communicate with each other.

iam planning to make a attendance tracker for employees using spring boot and angular and mongodb as database. this application is a monolithic application. the employees should login with their credentials and click mark in and upload their picture as proof and when it is time they should click mark out and again click picture as proof. And the employees can also download their attendance history as document. each employee has their own Supervisor assigned to them. the Supervisor should be able to login with their credentials and they can see all the employees assigned to them and they can view all the data of the employee. the admin is the only one who can create accounts employees or Supervisor. now i want you to give me the detailed packages i should create and the classes in them bearing in mind few design patterns.  
For your monolithic attendance tracker application, I recommend structuring your code using the following package structure and classes, incorporating appropriate design patterns for better maintainability and scalability:

**Package Structure**

arduino

Copy code

com.company.attendancetracker

│

├── config

├── controller

├── dto

├── entity

├── repository

├── service

│ ├── impl

├── util

└── exception

**Explanation of Packages and Classes**

**1. config**

* **Purpose**: To handle application configuration.
* **Classes**:
  + SecurityConfig: Configure authentication and authorization (e.g., roles for Employee, Supervisor, and Admin).
  + MongoConfig: Optional, for MongoDB-specific configurations.

**2. controller**

* **Purpose**: Contains REST controllers for different endpoints.
* **Classes**:
  + AuthController: Handles login and authentication endpoints.
  + EmployeeController: Handles employee-specific endpoints like mark in, mark out, and download attendance history.
  + SupervisorController: Handles endpoints for viewing assigned employees' data.
  + AdminController: Handles admin operations such as creating accounts for employees and supervisors.

**3. dto (Data Transfer Objects)**

* **Purpose**: Defines objects for transferring data between layers.
* **Classes**:
  + LoginRequest: For user login (username, password).
  + AttendanceRequest: For attendance marking (employeeId, timestamp, photo).
  + AttendanceResponse: For attendance history.
  + UserDTO: For user details during admin account creation.

**4. entity**

* **Purpose**: Defines MongoDB collections.
* **Classes**:
  + User: Common user entity with fields for username, password, role (EMPLOYEE, SUPERVISOR, ADMIN), etc.
  + Attendance: Stores attendance data (employeeId, markInTime, markOutTime, photoProof).
  + SupervisorEmployeeMapping: Maps supervisors to employees.

**5. repository**

* **Purpose**: Contains interfaces for database interactions.
* **Interfaces**:
  + UserRepository: For User collection CRUD operations.
  + AttendanceRepository: For Attendance collection operations.
  + SupervisorEmployeeMappingRepository: For mapping operations.

**6. service**

* **Purpose**: Business logic layer.
* **Interfaces**:
  + AuthService: Handles login and authentication logic.
  + EmployeeService: Handles employee-specific logic like attendance marking and history retrieval.
  + SupervisorService: Handles logic for supervisors to view employee data.
  + AdminService: Handles user creation logic.

**7. service.impl**

* **Purpose**: Contains implementations of the service interfaces.
* **Classes**:
  + AuthServiceImpl
  + EmployeeServiceImpl
  + SupervisorServiceImpl
  + AdminServiceImpl

**8. util**

* **Purpose**: Utility classes for common functionality.
* **Classes**:
  + FileUtil: For generating and downloading attendance history documents.
  + PhotoUtil: For processing and validating uploaded pictures.

**9. exception**

* **Purpose**: Handles custom exceptions.
* **Classes**:
  + CustomException: Base class for application-specific exceptions.
  + UserNotFoundException
  + InvalidAttendanceException
  + UnauthorizedException

**Key Design Patterns**

1. **Factory Pattern**: Use for creating different types of users (Employee, Supervisor, Admin).
2. **Strategy Pattern**: Use for different authentication mechanisms, if required.
3. **Builder Pattern**: Use for complex DTO or entity creation, like building a report.
4. **Observer Pattern**: Notify supervisors when an employee marks attendance (optional).
5. **Template Method Pattern**: For consistent data processing (e.g., in services).

Would you like more details on specific classes or implementation guidance for any part of this structure?

4o

**You said:**

what about global exception?

**ChatGPT said:**

ChatGPT

A **Global Exception Handler** is a great addition to handle exceptions in a centralized manner across the entire application. This ensures that your application can provide meaningful and consistent error responses to the client, avoiding exposing stack traces or other sensitive details.

**Adding a Global Exception Handler**

You can create a class in the exception package to handle all exceptions globally. Here’s how you can structure it:

**GlobalExceptionHandler Class**

java

Copy code

package com.company.attendancetracker.exception;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.ExceptionHandler;

import org.springframework.web.bind.annotation.RestControllerAdvice;

@RestControllerAdvice

public class GlobalExceptionHandler {

// Handle custom exceptions

@ExceptionHandler(CustomException.class)

public ResponseEntity<ErrorResponse> handleCustomException(CustomException ex) {

ErrorResponse response = new ErrorResponse(ex.getMessage(), HttpStatus.BAD\_REQUEST.value());

return new ResponseEntity<>(response, HttpStatus.BAD\_REQUEST);

}

// Handle UserNotFoundException

@ExceptionHandler(UserNotFoundException.class)

public ResponseEntity<ErrorResponse> handleUserNotFoundException(UserNotFoundException ex) {

ErrorResponse response = new ErrorResponse(ex.getMessage(), HttpStatus.NOT\_FOUND.value());

return new ResponseEntity<>(response, HttpStatus.NOT\_FOUND);

}

// Handle InvalidAttendanceException

@ExceptionHandler(InvalidAttendanceException.class)

public ResponseEntity<ErrorResponse> handleInvalidAttendanceException(InvalidAttendanceException ex) {

ErrorResponse response = new ErrorResponse(ex.getMessage(), HttpStatus.BAD\_REQUEST.value());

return new ResponseEntity<>(response, HttpStatus.BAD\_REQUEST);

}

// Handle UnauthorizedException

@ExceptionHandler(UnauthorizedException.class)

public ResponseEntity<ErrorResponse> handleUnauthorizedException(UnauthorizedException ex) {

ErrorResponse response = new ErrorResponse(ex.getMessage