

## **Laboratory Activity – Design Pattern**

### Applying Design Patterns in Software Development

#### **Objectives**

Understand the purpose and benefits of using design patterns in software development.

Identify appropriate situations for using specific design patterns

Apply common design patterns (Singleton, Factory, Repository, Strategy, Observer) in a real project.

Improve code flexibility, maintainability, and scalability using patterns.

#### **Background/Theory Review**

Design patterns are proven solutions to common software design problems.

They provide reusable templates rather than complete implementations.

#### **Design Patterns Covered**

Singleton Pattern-Ensures only one instance of a class exists.

Factory Pattern-Handles object creation logic

Repository Pattern-Separates data access from business logic.

Strategy Pattern - Allows dynamic selection of algorithms.

Observer Pattern-Enables automatic notification on state changes.

#### **When and Why to Use Design Patterns**

Singleton: shared resources like configuration or logging.

Factory: when object creation is complex or varies

Repository: cleaner data access and easier testing.

Strategy: interchangeable behaviors

Observer: event-driven systems.

## Part A: Pattern Identification & Justification

Design Pattern	Applied Area	Justification
MVC Pattern	presentation, service, data modules	Separates UI logic, business logic, and data handling for better maintainability.
DAO Pattern	AttendanceDAO	Encapsulates database access and isolates SQL logic from services.
Service Layer Pattern	AttendanceService	Centralizes attendance rules and validation logic.
Singleton Pattern	Database connection	Ensures only one database connection instance exists.
Factory Pattern	DAO creation	Decouples object creation from usage.
Strategy Pattern	Attendance marking logic	Allows different attendance rules without modifying core logic.

## **Part B: Implementation of Three Design Patterns**

### **Singleton Pattern — Database Connection**

**Purpose:** Ensure a single database connection instance across the system.

```
# config/db_connection.py
import sqlite3
```

```
class DBConnection:
```

```
    _instance = None
```

```
    def __new__(cls):
```

```
        if cls._instance is None:
```

```
            cls._instance = super(DBConnection, cls).__new__(cls)
```

```
            cls._instance.connection = sqlite3.connect("attendance.db")
```

```
        return cls._instance
```

```
    def get_connection(self):
```

```
        return self.connection
```

### **DAO Pattern — AttendanceDAO**

**Purpose:** Encapsulate all database access logic.

```
# data/attendance_dao.py
```

```
from config.db_connection import DBConnection
```

```
class AttendanceDAO:
```

```
    def __init__(self):
```

```
        self.conn = DBConnection().get_connection()
```

```
    def save_attendance(self, student_id, status):
```

```
        cursor = self.conn.cursor()
```

```

cursor.execute(
    "INSERT INTO attendance (student_id, status) VALUES (?, ?)",
    (student_id, status)
)
self.conn.commit()

def get_all_attendance(self):
    cursor = self.conn.cursor()
    cursor.execute("SELECT * FROM attendance")
    return cursor.fetchall()

```

## Factory Pattern — DAO Factory

**Purpose:** Centralize DAO object creation.

```
# config/dao_factory.py
from data.attendance_dao import AttendanceDAO
```

```
class DAOFactory:
```

```
    @staticmethod
    def get_attendance_dao():
        return AttendanceDAO()
```

## Strategy Pattern — Attendance Rules

### Strategy Interface

```
# service/strategy/attendance_strategy.py
from abc import ABC, abstractmethod
```

```
class AttendanceStrategy(ABC):
```

```
    @abstractmethod
    def mark_attendance(self, student):
```

```
pass
```

## Concrete Strategies

```
# service/strategy/time_based_strategy.py
from service.strategy.attendance_strategy import AttendanceStrategy
```

```
class TimeBasedStrategy(AttendanceStrategy):
```

```
    def mark_attendance(self, student):
        return True
```

```
# service/strategy/geo_based_strategy.py
```

```
from service.strategy.attendance_strategy import AttendanceStrategy
```

```
class GeoBasedStrategy(AttendanceStrategy):
```

```
    def mark_attendance(self, student):
        return True
```

## Service Layer Pattern — AttendanceService

```
# service/attendance_service.py
from config.dao_factory import DAOFactory
```

```
class AttendanceService:
```

```
    def __init__(self):
        self.dao = DAOFactory.get_attendance_dao()
        self.strategy = None
```

```
    def set_strategy(self, strategy):
```

```
        self.strategy = strategy
```

```
    def mark_attendance(self, student):
```

```
if self.strategy and self.strategy.mark_attendance(student):
    self.dao.save_attendance(student.id, "Present")
```

## Controller (MVC – Presentation Layer)

```
# presentation/controllers/attendance_controller.py
from service.attendance_service import AttendanceService
from service.strategy.time_based_strategy import TimeBasedStrategy
```

```
class AttendanceController:
```

```
    def __init__(self):
        self.service = AttendanceService()
        self.service.set_strategy(TimeBasedStrategy())
```

```
    def mark_button_clicked(self, student):
```

```
        self.service.mark_attendance(student)
```

## **Part C: Documentation**

### **Singleton Pattern — Database Connection**

Used by all DAO classes. It prevents multiple DB connections

### **DAO Pattern — AttendanceDAO**

Keeps SQL out of service/controller and Easier to maintain and test

### **Factory Pattern — DAO Factory**

Supports future DAO changes and Improves decoupling

### **Strategy Pattern — Attendance Rules**

Behavior changes without editing service/controller and Open–Closed Principle compliant

### **Service Layer Pattern — AttendanceService**

Coordinates DAO + Strategy and holds business rules.

### **UML Class Diagram (PlantUML for draw.io)**

```
@startuml
```

```
package config {
```

```
    class DBConnection {  
        -_instance  
        +get_connection()  
    }
```

```
    class DAOFactory {
```

```
        +get_attendance_dao()  
    }  
}
```

```
package data {
```

```
    class AttendanceDAO {  
        +save_attendance()  
        +get_all_attendance()  
    }
```

```
}
```

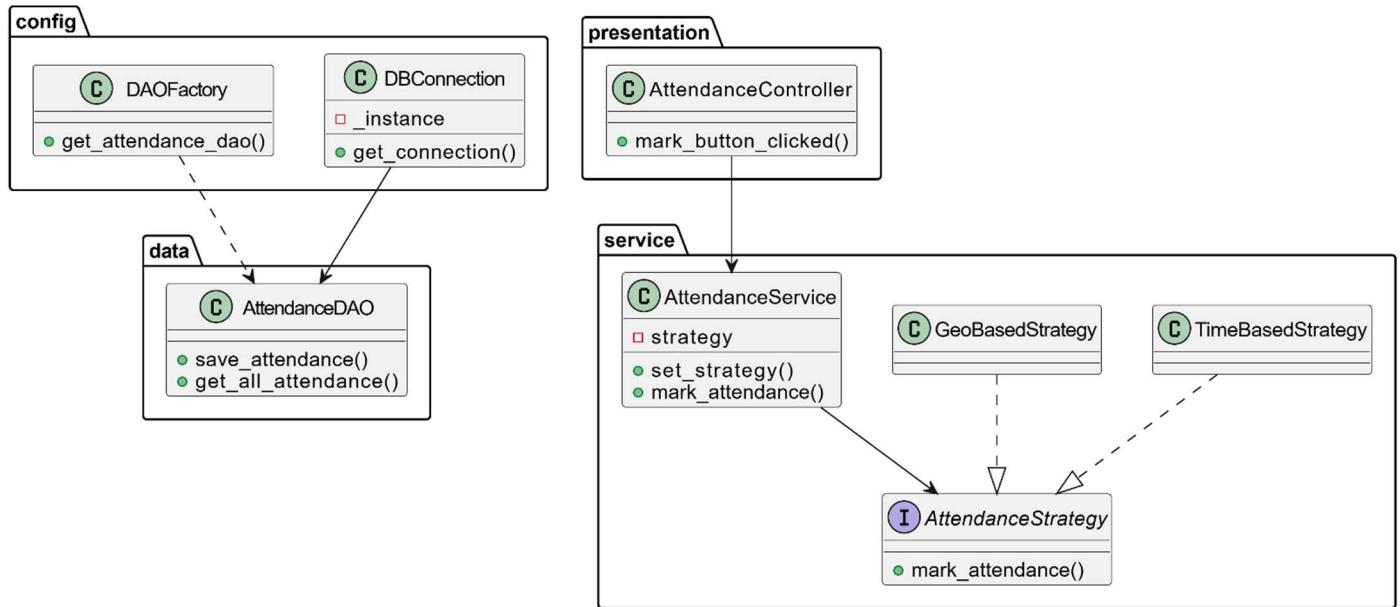
```
package service {  
    interface AttendanceStrategy {  
        +mark_attendance()  
    }  
}
```

```
class TimeBasedStrategy  
class GeoBasedStrategy
```

```
class AttendanceService {  
    -strategy  
    +set_strategy()  
    +mark_attendance()  
}  
}
```

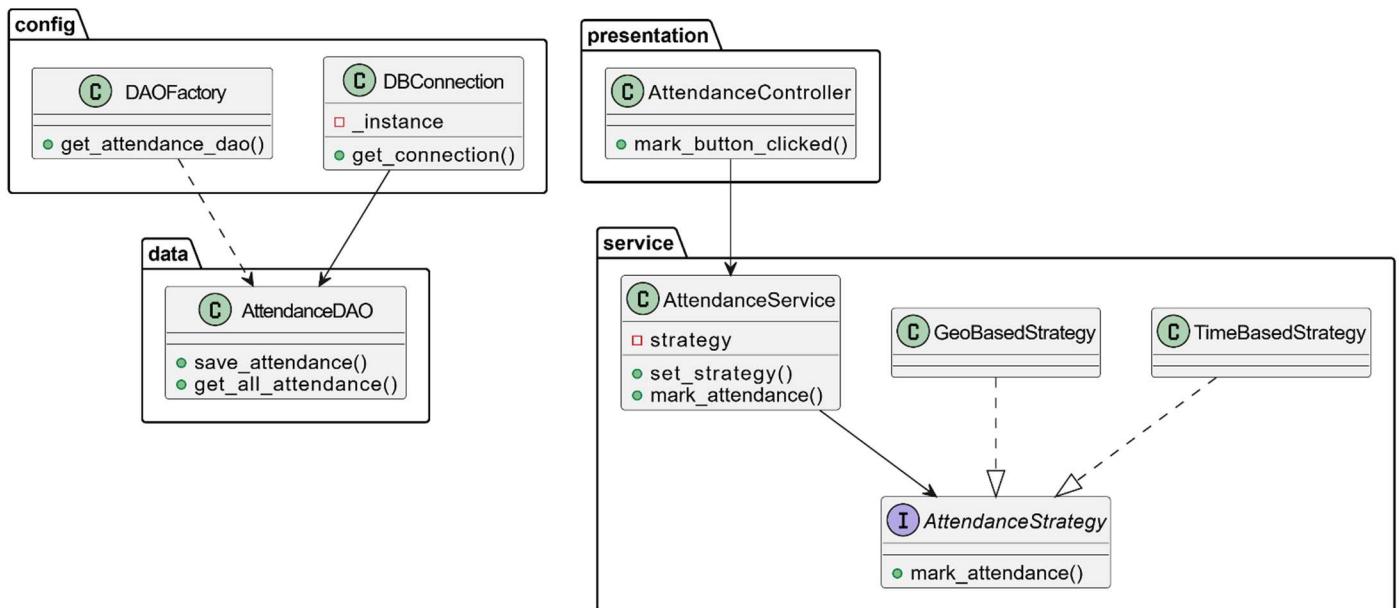
```
package presentation {  
    class AttendanceController {  
        +mark_button_clicked()  
    }  
}
```

```
DBConnection --> AttendanceDAO  
DAOFactory ..> AttendanceDAO  
AttendanceService --> AttendanceStrategy  
TimeBasedStrategy ..|> AttendanceStrategy  
GeoBasedStrategy ..|> AttendanceStrategy  
AttendanceController --> AttendanceService  
@enduml
```



## Deliverables

[https://github.com/Dadacay/AGD\\_StudentAttendanceSystem.git](https://github.com/Dadacay/AGD_StudentAttendanceSystem.git)



The **AGD\_StudentAttendanceSystem** is implemented using **MVC Architecture** in Python. The system separates responsibilities into presentation, service, and data layers to improve maintainability and scalability.

## Design Patterns Applied

- **MVC Pattern** ensures separation of concerns.
- **DAO Pattern** isolates database logic.
- **Service Layer Pattern** centralizes business rules.
- **Singleton Pattern** guarantees a single database connection.
- **Factory Pattern** abstracts DAO creation.
- **Strategy Pattern** allows flexible attendance marking rules.

## System Benefits

- Easy to extend and modify
- Reduced code duplication
- Improved testability
- Clear separation of responsibilities
- Professional, industry-standard design