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.

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
```

✓ Used by all DAO classes
✓ Prevents multiple DB connections

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()
```

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

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()
```

✓ Supports future DAO changes
✓ Improves decoupling

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
```

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

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")
```

✓ Coordinates DAO + Strategy
✓ Holds business rules

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)
```

✓ Controller handles user interaction
✓ Business logic stays in service

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

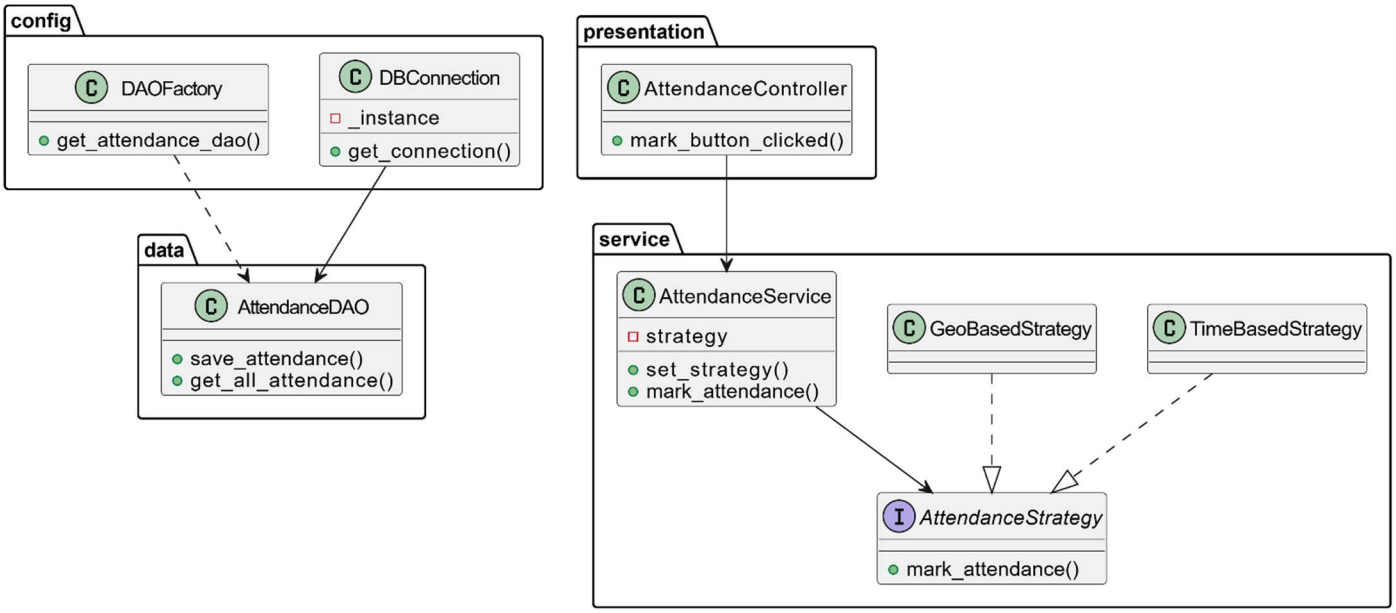
AttendanceService --> AttendanceStrategy

TimeBasedStrategy ..|> AttendanceStrategy

GeoBasedStrategy ..|> AttendanceStrategy

AttendanceController --> AttendanceService

@enduml



Design Overview

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