Design Patterns - Hallmarks of Good Architecture

SOLID Principles of **Object-Oriented Programming**

Practice Session

Dependency Inversion Principle (DIP)

Design Patterns - Hallmarks of Good Architecture

```
class User {
  String name;
  // other properties...

  User(this.name);
}

class MySQLDatabase {
  void saveUser(User user) {
    print('Saving ${user.name} to MySQL
database...');
    // Actual implementation...
  }
}

class UserService {
  MySQLDatabase database;

  UserService(this.database);

  void saveUser(User user) {
    database.saveUser(user);
  }
}
```

Dependency Inversion Principle (DIP)

Hints:

1. Identify direct dependencies between high-level and low-level modules in your code.
2. Introduce an interface or abstract class to **decouple these modules**.
3. Modify the high-level module to **depend on the abstraction**, not on the low-level module.
4. Implement the abstraction **in each low-level module**.
5. Use **dependency injection** to provide the low-level module to the high-level module.

Design Patterns - Hallmarks of Good Architecture

```
abstract class Database {
  void saveUser(User user);
}

class MySQLDatabase implements Database {
  @override
  void saveUser(User user) {
    print('Saving ${user.name} to MySQL
database...');
    // Actual implementation...
  }
}

class PostgreSQLDatabase implements Database {
  @override
  void saveUser(User user) {
    print('Saving ${user.name} to PostgreSQL
database...');
    // Actual implementation...
  }
}
```

```
class UserService {
  Database database;

  UserService(this.database);

  void saveUser(User user) {
    database.saveUser(user);
  }
}
```

Design Patterns - Hallmarks of Good Architecture

Dependency Inversion Principle (DIP)

1. In the refactored solution, we create an abstract class `Database` that declares the `saveUser` method.
2. Both `MySQLDatabase` and `PostgreSQLDatabase` implement this interface.
3. The `UserService` class depends on the `Database` abstraction, **not on a specific database class**.
4. This way, **we can easily switch** between different database systems without changing `UserService`.
5. The original code violates the **Dependency Inversion Principle** because `UserService` **directly depends** on a specific database class which is the `MySQLDatabase`.
6. This makes `UserService` **less flexible** and harder to adapt to changes (like switching to another database system).