7 Task 3.1

Question 7. Describe an architectural approach you will use to implement the desired system. How many modules you plan for the whole UWC 2.0 system? Briefly describe input, output and function of each module

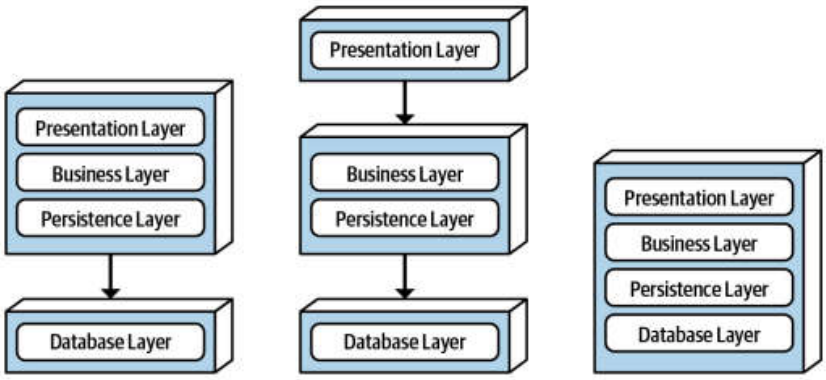
Solution

7.1 Theoretical basis: Layered Architecture Style

Definition: Layered architecture patterns are n-tiered patterns where the components are organized in horizontal layers. This is the standard method for designing most software and is intended to be self-contained. This means that all of the components are linked but not dependent on one another.
This architecture has four layers, each with a connection between modularity and component within it. They are, from top to bottom:

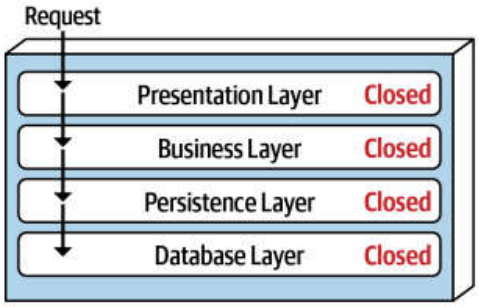
1. **Presentation layer:** It includes categories associated with the presentation layer.
2. **Business layer:** It includes business logic.
3. **Persistence layer:** It is used to handle functions such as object-relational mapping.
4. **Database layer:** This is where all of the data is kept.

There may be, however, some accepted variants of the above model.



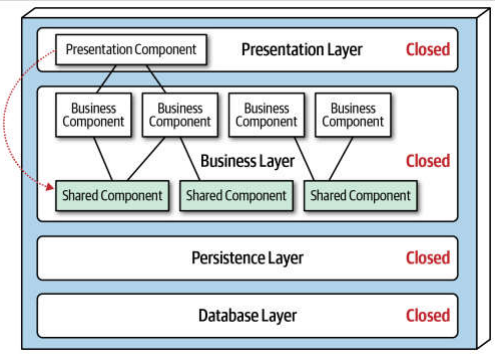
For example: The leftmost variant combines the presentation layer, the business layer and the persistence layer into a single deployment unit, while the database layer is external physical database.

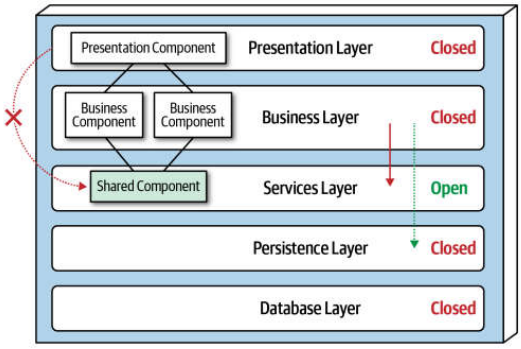
In case that all the layers are closed, which means that a request must traverse all layers from top to bottom.



- The first reason is that all 'similar' components are grouped together. Having 'similar' components together means that everything relevant to a single layer remains in that single layer. This allows for a clean separation of component types and also aids in gathering similar programming code in one location.
- The second reason is that it provides layers of isolation. The layers become independent of one another when they are isolated. Thus, changing the database from an Oracle server to a SQL server will have a significant impact on the database layer but will have no effect on the other layers. Assume you have a custom written business layer and want to replace it with a business rules engine. If we have a well-defined layered architecture, the change will not affect other layers.

Aside from the layers mentioned, the layered architecture pattern can be modified to include additional layers. This is referred to as hybrid layered architecture. A service layer, for example, can exist between the business and persistence layers. This is not an ideal situation because the business layer must now pass through the service layer to reach the persistence layer. Going through the service layer adds no value to this request. This is known as an architecture sinkhole anti-pattern. Requests are routed through layers, with little or no logic performed in each.





Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★
Elasticity	★
Evolutionary	★
Fault tolerance	★
Modularity	★
Overall cost	★★★★★
Performance	★★
Reliability	★★★
Scalability	★
Simplicity	★★★★★
Testability	★★

- Cost and simplicity: Monolithic architecture
    1. Don't have the complexities associated with distributed architecture styles.
    2. Are simple and easy to understand
    3. Are relatively low cost to build and maintain.
- Deployability:
    1. Don't have the complexities associated with distributed architecture styles.
    2. Are simple and easy to understand
    3. Are relatively low cost to build and maintain.
- Testability:
    1. Can mock or stub components (or even an entire layer).
    2. Even small changes can force hours of re-executing the entire regression test suite.
- Reliability:
    1. No network traffic, bandwidth, and latency.
    2. Monolithic deployments.
- Elasticity and scalability:
    1. Monolithic deployments.
    2. No architectural modularity.
    3. Applications can only scale to a certain point.
    4. Selective scaling requires very complex design techniques.
- Performance: Does not lend itself to high-performance systems due to
    1. Lack of parallel processing.
    2. Closed layering.
    3. Sinkhole architecture anti-pattern.
- Fault tolerance:
    1. Due to monolithic deployments and the lack of architectural modularity: a small error can impact and crash the entire application unit.

With the characteristics thoroughly discussed, we came to the conclusion that Layered Architecture is the most suitable approach for the task at hand. We chose it because of several reasons:

- It is a simple architecture, which suits the characteristics of the software requirements (few and simple).
- It is the popular choice for early development of new systems, when we're still unsure of what architecture is the most suitable. Should new requirements appear, we can move to other architectures with ease.
- Despite certain flaws, it is still relevant in the industry. Future maintenance therefore is easier as finding engineers with layered architecture knowledge is a trivial task.
- The software requirements stated no potential additional features, aside from more MCPs. This means the architecture's subpar scalability is not going to affect us too much.
- The architecture's monolithic nature provide us with extremely low latency across the system, as there is no communication latency between distributed services/modules.

While Layered Architecture addresses most of our needs, there are aspects that the architecture underperforms in. Monolithic systems, in general, are very bad at handling faults. To satisfy the availability requirements of 95% uptime, we planned to test the system extensively to minimize bugs and failures on our end. It is not ideal, but given the benefits that this architecture will bring to the system, we think this is a good trade-off.

7.2 Modules of UWC 2.0

Using the layered architecture approach for the implementation of UWC 2.0, our group agree that the UWC 2.0 system can be further divided into separated modules and each of them will be responsible for a specific task. According to our group, there are 8 modules that make up the whole system, each belongs to an appropriate layer. Based on the concept of layered architecture, the UWC 2.0 software will consist of 5 layers.

1. ***Database layer:*** This layer consists of a database to store data, and a basic class to establish connections between the database and the persistence layer. Therefore, it has no module.
2. ***Persistence layer:***
    - **ORM module:**

Function	ORM module consists services for querying and updating data. Upper layer will use this module's services to interact with the database.
Input	Requests from business layer. Each request will contain information specifying the need of business layer and use one or more services.
Output	1. If the requests related to data retrieving, ORM module will return required data in case they exist or a warning message if they don't. 2. If the requests ask for data updating, ORM module will return a confirm message after updating successfully or an error message in case errors occur.

3. ***Service layer:***

   - **External API module:**

Function	Uses external APIs to provide corresponding services for business layer.
Input	Requests from the business layer.
Output	Appropriate responses to the requests.

4. ***Business layer:***

   - **Message module:**

Function	Provides services for dealing with requests related to messaging such as create message, send message, etc.
Input	Requests from the presentation layer.
Output	Appropriate responses to the requests.

   - **Task module:**

Function	Provides services for requests related to tasks like check in/check out, get task overview, assign task, etc.
Input	Requests from the presentation layer.
Output	Appropriate responses to the requests.

   - **User module:**

Function	Provides services for managing users' accounts (log in, log out, overview information, etc)
Input	Requests from the presentation layer.
Output	Appropriate responses to the requests.

- **MCP module:**

Function	Provides services for dealing with MCPs' information like assign MCP, get MCPs' current capacity, get MCP's overview and so on.
Input	Requests from the presentation layer.
Output	Appropriate responses to the requests.

- **Vehicle module:**

Function	Provides services for working with information of vehicles such as get vehicles' overview, assign vehicle, etc.
Input	Requests from the presentation layer.
Output	Appropriate responses to the requests.

5. ***Presentation layer:***

- **View module:**

Function	Provides services for displaying and rendering contents as well as processing users' interactions. For example, when a user types in the required information for logging in and clicks (presses) log in button, some specific services are invoked to pass the requests to business layer.
Input	Users' interaction (click/press button displayed on the screen, etc)
Output	Rendering the contents displayed on the screen according to users' interaction.

8 Task 3.2

Question 8. Draw an implementation diagram for Task Assignment module

Solution

8.1 Theoretical basis: Component diagram

UML Component diagrams are used in modeling the physical aspects of object-oriented systems that are used for visualizing, specifying, and documenting component-based systems and also for constructing executable systems through forward and reverse engineering. Component diagrams are essentially class diagrams that focus on a system's components that often used to model the static implementation view of a system.
A component diagram breaks down the actual system under development into various high levels of functionality. Each component is responsible for one clear aim within the entire system and only interacts with other essential elements on a need-to-know basis.
A component diagram has some common concepts:

- **Component:** represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. In UML 2.x, a component is drawn as a rectangle with optional compartments stacked vertically.
- **Interface:** 2 types:
  - **Required interface:** represents an interface that the component requires, drawn as a socket (half-circle).
  - **Provided interface:** represents an interface that the component provides, drawn as a lollipop (full-circle).
- **Port:** often used to help expose required and provided interfaces of a component, represented as a square placed along the edge of the component rectangle.
- **Relationship:** includes association, composition, aggregation, constraint, links and dependency. Their meaning and representation in UML are the same with those in class diagram.

8.2 Component diagram of Task assignment module

8.2.1 Component diagram

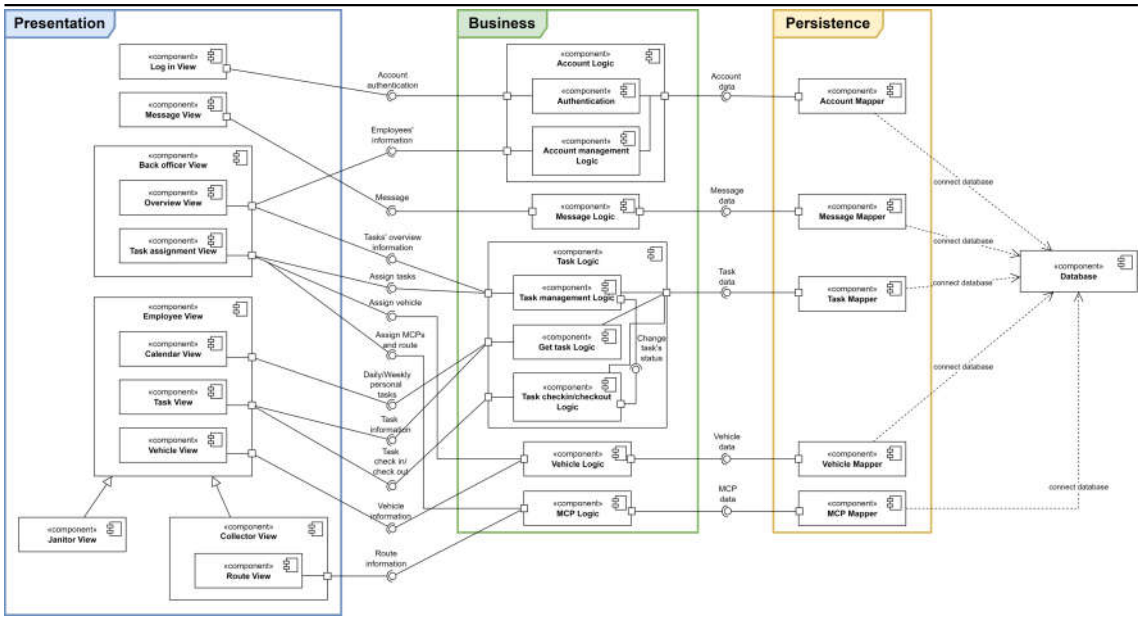


Figure 3: Component diagram of Task assignment module of UWC 2.0

Follow this link for a PNG file with better resolution, or this link for the diagram file.

8.2.2 Description and explanation

The component diagram is divided into 4 different layers corresponding to layered architecture that we chose in task 3.1 (we did not include Service layer in the diagram since it is just a layer separated from the Business layer to include some shared and external API for the components in Business layer), with each layer has the following components:

1. **Presentation layer:**
    - **Log in View:** displays the log in UI to user, uses interface Account authentication from component Account Logic (internal component Authentication) to perform its functionality.
    - **Message View:** displays the message UI to user, uses interface Message exposed by component Message Logic.
    - **Back officer View:**
        - **Overview View:** displays the overview UI for back officer by using interfaces Employees' information provided by component Account Logic (internal component Account Management Logic) and Tasks' overview information provided by component Task Logic (internal component Task management Logic).
        - **Task assignment View:** displays the task assignment view for back officer, requires interfaces Assign tasks, Assign vehicle and Assign MCPs and route from components Task Logic (internal component Task management Logic), Vehicle logic and MCP Logic respectively.
    - **Employee View:**

- **Calendar View:** displays calendar UI for collector and janitor, uses interfaces Daily/Weekly personal tasks from component Task Logic (internal component Get task Logic).
- **Task View:** displays individual task and its details to collector and janitor by using interfaces Task information and Task check in/check out from component Task Logic (internal component Get task Logic and Task check in/check out Logic respectively).
- **Vehicle View:** displays vehicle information to collector and janitor by using interfaces Vehicle information from component Vehicle Logic.
- Component Employee View is inherited by component Janitor View, which displays UI for janitor, and component Collector View, which displays UI for collector. Component Collector View has another internal component Route View displaying the route UI for collector by using interface Route information from component MCP Logic.

2. **Business layer:** consists of 5 major components representing 5 modules of the Business layer we have discussed in task 3.1 which are Account Logic, Message Logic, Task Logic, Vehicle Logic and MCP Logic, each of which requires the data object from its respective data mapper component to perform its business logic.

3. **Persistence layer:** consists of 5 data mappers which are Account Mapper, Message Mapper, Task Mapper, Vehicle Mapper and MCP Mapper, all of which establish a connection to component Database in order to retrieve and perform query on the data.

4. **Data layer:** consists of Database providing data to each data mapper in the Persistence layer.