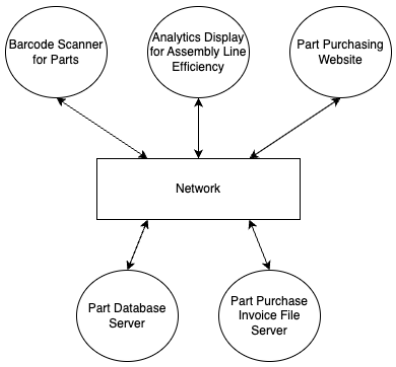
1. Client-Server

For the Client-Server architectural style, a real world non-software example is an inventory management system for a large machining company that manufactures and stores machined parts. There are many inventory clients where the inventory registry can be managed (insert, delete, etc) and explored (analytics, read, etc) virtually, and these clients all interact with a server over the network to interface with a central database. The client sends requests to the database server, the server does the intensive computation and read/write to the database, and then returns the data requested by the client.



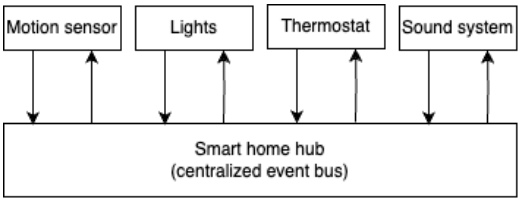
In terms of how this style could be used, consider the inventory management system has multiple clients. Some clients are simple bar-code scanners which log that a new part has been machined. Other clients are display monitors showing the live statistics of machined parts per hour in the assembly line. There is also a web-page client where external users are searching for parts, if the part has available inventory, they can purchase the part. All the clients contact a database server through a network that stores and processes each operation, maintaining a single source of truth. A client-server style is beneficial to this system since it makes it easy to scale as more parts are added to the system, only the database server needs to be scaled up to accommodate the size requirements. Additionally, regardless of user demand through the web-page client, the database server can also be scaled in terms of computing power since the web-page will not and should not be doing the intensive computation to search the large machined part directory, this responsibility would lie in the server. Also, the system has many types of platforms for the clients, ranging from bar-code scanners to modern web-pages, having a server which processes the standardized requests and performs the computations will enable this mix of many heterogeneous platforms into the same database. This style will avoid needing new processing methods for each client platform/type as the processing is all done by some server connected over the network. This is also great from a security perspective because the server can be configured so that certain requests can only be performed from the local network of the machining system (internal to the assembly line), such as inserting new parts into the database system, and other requests can be available to the global network such as purchasing new machined parts from the web-page client, enforcing security.

Client-Server significantly reduces coupling because it separates concerns between client and server. The server can modify their algorithms and business logic, for example improve their part searching algorithm by making the search sorted alphabetically, while keeping the request and data output (interface to the client) the same, ultimately making it possible to improve or change the computation while not having to make any change to the web-page which requests to search the part directory and visualizes them. Additionally, as long as the request remains the same, a diverse set of clients and platforms can be integrated into the system since it is the server that is concerned with the business logic, while the client only needs to send/receive requests from the server.

This style enables many key future changes in the system. For example, if the database server is getting overburdened because the assembly line is ramping up production, it can be scaled up in terms of speed and size to accommodate the new demands without making changes to the clients. Also, client-server allows us to easily add a new generation of bar-code scanners without changing the database server because only the client (scanner) needs to be setup to run the standard requests. New servers such as a file server which stores purchase invoices that can be requested by customers from the web-page client can also be added, without changing other clients or servers.

2. Implicit Invocation

For the Implicit Invocation architectural style, specifically the event-based variant, a real world non-software example is a smart home system which demonstrates a modern way to manage a diverse collection of devices and functionalities in a connected home environment. This architecture style uses decoupling and event-based communication to facilitate interactions among devices without needing the direct knowledge of each other.



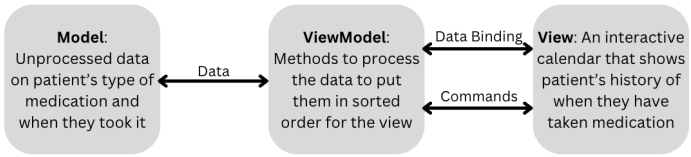
Envision a smart home scenario where a person wants to adjust various settings of a room based on their presence. In the context of this example, the components will be a motion sensor, room lights, a thermostat, and a sound system. The connector will be the centralized smart home hub which serves as an event bus. When the person enters a room, a motion sensor would detect the movement of the person and emit an "motion detected" event. This event would be absorbed by other devices, who are subscribers of the event, that are also connected to the smart home hub. It would trigger the thermostat to adjust the room temperature based on the weather, the lights to turn on to a particular brightness, and the sound system would start playing music based on the time of day. As described above, the devices do not need to communicate directly with each other but function through the emitting and absorbing of events. The approach of incorporating the event-based variant of the implicit invocation architectural style presents many benefits such as flexibility, modularity, and robustness in the system. For example, this style facilitates simplified system expansion with integrating new devices. Also, the devices in the system can be updated independently which enhances total system sustainability. Additionally, since problems with one device do not directly impact other devices, this style minimizes the risks of widespread failure which results in an overall stable system. In a traditional smart home system, devices need to communicate directly with each other to perform actions such as a motion sensor directly telling the thermostat to adjust to a particular temperature. This leads to a tightly-coupled and rigid system architecture which makes overall expansion, of adding new devices, and updating the functionality of devices very difficult.

The use of the implicit invocation architectural style reduces coupling through event-based communication where components interact by emitting and absorbing events instead of direct calls between components. In the provided example, the motion sensor emits the "motion detected" event without having knowledge of which devices in the system will use this information. This guarantees component independence reinforced by a subscription scheme that facilitates dynamic communication without explicit references. By having this decoupling, it allows for easy system dynamic reconfiguration and extension which promotes modularity and encapsulation. In the end, it allows for scalable, flexible systems capable of adapting over time with minimal impact on existing components, fostering separation of concerns and maintainability.

Using implicit invocation in smart systems has many benefits which enable diverse future changes in the system. Firstly, new devices can be easily added to the system without needing to reconfigure existing components. This enhances scalability which allows the system to have new features and devices added in the future. Additionally, the system is able to adapt to various scenarios and user preferences without having to do much re-configuring. For example, reactions to a motion event can be easily changed from changing lights to triggering security cameras. Also, since devices operate independently of each other, if one device fails then it does not affect the entire system. This decentralized approach helps keep the system functional even if some devices experience failure.

3. Model-View-ViewModel

With the Model-View-ViewModel (MVVM) architectural style, a real world non-software example includes a health records system. This architectural style could be used to help manage the history and health records of different patients. It is beneficial to the overall system as it separates the user interface from the data model, and so changes to the model don't affect the user interface. It also helps reduce coupling by using reusable components.



The model encapsulates the application's business logic and data. In the model, unprocessed data is stored that requires interpretation. Given this information, for the health records system, this can include patient data, such as personal information, medical history, allergies, medications, and any other information a hospital may need. In terms of the view model, it encapsulates presentation logic and state. It's purpose is to bridge the gap between raw data in the model and the views. Connecting this to the health records system, the view model could include methods to interpret the data. For instance, a doctor may want to see when a patient has been checked by a nurse. In the view model, there could be a way to sort this information, and send it in a preferred format to the view. In addition to this example, the view model can include ways to create a medication calendar for a patient. Other such examples can be applied given a hospital's needs. Lastly, the view encapsulates the user interface (UI) and its logic. It is also stateless and can be easily destroyed/recreated by the OS. Many different views can be added to a health records system. Some examples include a view for a doctor to see a patient's medical chart, a view for a doctor to see a calendar of a patient's visits, and other views depending on the hospital's desires.

MVVM reduces coupling by separating an application's UI from its logic (separation of concerns). For example, by having a view model, we avoid changes in the model directly affecting the views. The view does not need to know the intricacies of the model, such as how the data is stored. This then allows different teams to be able to independently work and it also makes it easier to unit test. Relating this to the previous example of a doctor needing to see when a patient has taken medication, the view focuses on presenting the calendar and facilitating interactions with the doctor. The view model handles the logic of sorting dates.

As previously mentioned, MVVM helps reduce coupling, which allows one to add future changes more efficiently. By separating the views from the view model and model, changes in the views can be implemented without altering the logic that processes the patient data, which lies in the view model. That is, if a doctor wanted added functionality to see the a different format of a calendar, the view would only need to be changed.