Layered Architecture

Static Description of the Style/ Pattern

Layered Architectural Style: each function of an application are separated into logical layers based on functionality and layered on top of each other. Each layer provides a particular service to the layer above and is dependent on the layer below. The procedure calls decide how the layers interact are the connectors. Each layer contains multiple components and a component within a layer can interact with another component within the same layer. Communication between layers is explicit and loosely coupled. However, layers cannot interact with layers that are not directly above or below. They must go through each layer in between. Topological constraints also include limiting the interactions between adjacent layers. Layering aids the application support a strong separation which in turn supports flexibility and maintainability.

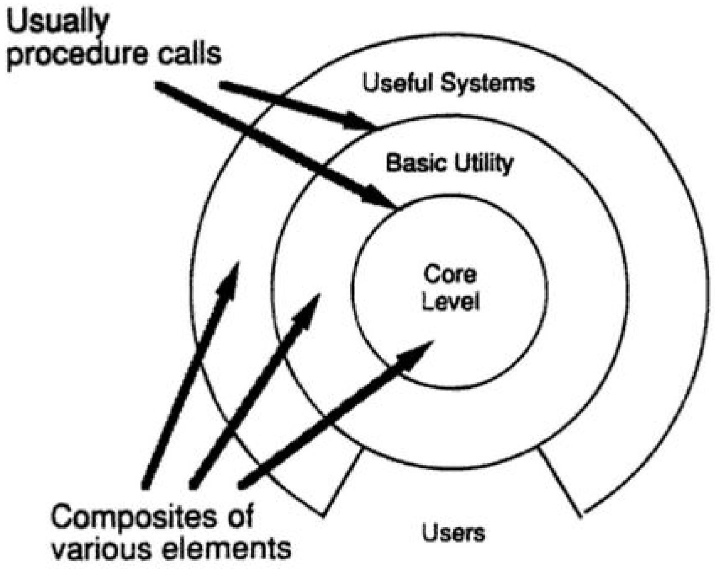


Figure 3: Layered Systems

The 4 most common layers are:

1. **Presentation**: presents the UI to the end-user and sends the response to the client related to view and the UI of the application
2. **Application layer** :contains the logic that the application needs to meet the functional requirements that is not a part of the domain rules. Acts as middleware and often used with 3rd party services
3. **Domain/Business Layer**: Includes the business logic and the domain entities
4. **Data Access Layer**: deals with technical and persistence. This includes networking, logging, persistent data, etc

<u>Common principles for designs:</u>

Abstraction:

- Layered architecture abstracts the view of the system as whole while providing enough detail to understand the roles and responsibilities of individual layers and the relationship between them.

Encapsulation:

- No assumptions need to be made about data types, methods and properties, or implementation during design, as these features are not exposed at layer boundaries.

Clearly defined functional layers:

- The separation between functionality in each layer is clear. Upper layers such as the presentation layer send commands to lower layers, such as the business and data layers, and may react to events in these layers, allowing data to flow both up and down between the layers.

High cohesion:

- Well-defined responsibility boundaries for each layer, and ensuring that each layer contains functionality directly related to the tasks of that layer, will help to maximize cohesion within the layer.

Reusable:

- Lower layers have no dependencies on higher layers, potentially allowing them to be reusable in other scenarios.

Loose coupling:

- Communication between layers is based on abstraction and events to provide loose coupling between layers.

Dynamic description of how the style / pattern is useful over time

<u>Main benefits</u>

Abstraction: changes can be made at the abstract level. Can increase or decrease the level of abstraction in each layer of the hierarchical stack

Modularity: implementation changes can be made within each layer without affecting or breaking functionality of other layers and the entire system

Isolation: allows for upgrades to be isolated to each individual layer. Reduces the risk and minimizes impact of the overall system.

- Differentiate between the different kinds of tasks performed by the components.
- team members can work in parallel on different parts of the application with minimal dependencies

Manageability: helps manage the code by organizing it in a way that separates core concerns and identifies dependencies.

Reusability: possibility of reusable components

Testability: increase testability by having well-defined layers. Can build mock objects that mimic the behaviour of concrete objects.

- Can test the components independently of each other

E.g. All UI changes happen in the Presentation Layer (Other layers don't accidentally break from it) - See example case

<u>Negative Behaviours</u>
Extra overhead: too many layers can cause degradation of performance as changes will pass slowly to higher layers
No built in scalability. Need to find your own way to implement it
Separation of technical aspects: Code is separated by technical aspects which results in classes that do common business and use case scenarios to be far away from each other

- Can also make it hard to assign the "right" functionality to each layer

Complex: cannot be used for simple applications as it adds unnecessary complexity

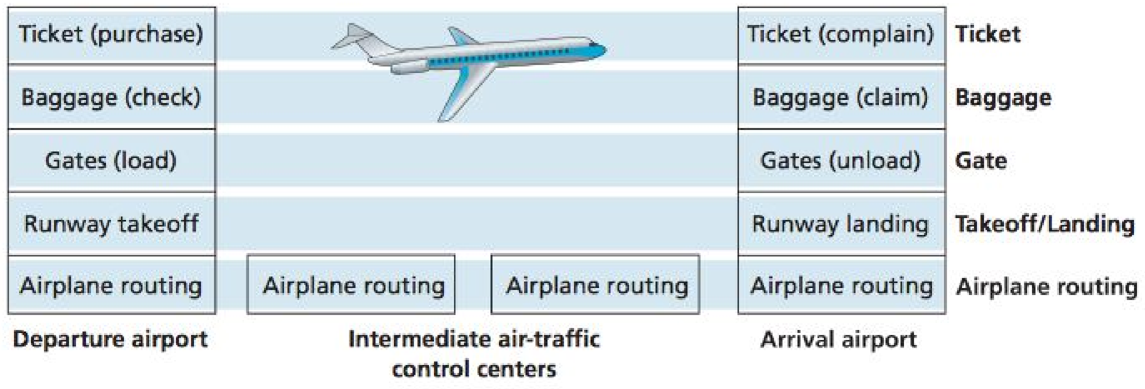
When to Use it:
Good for:

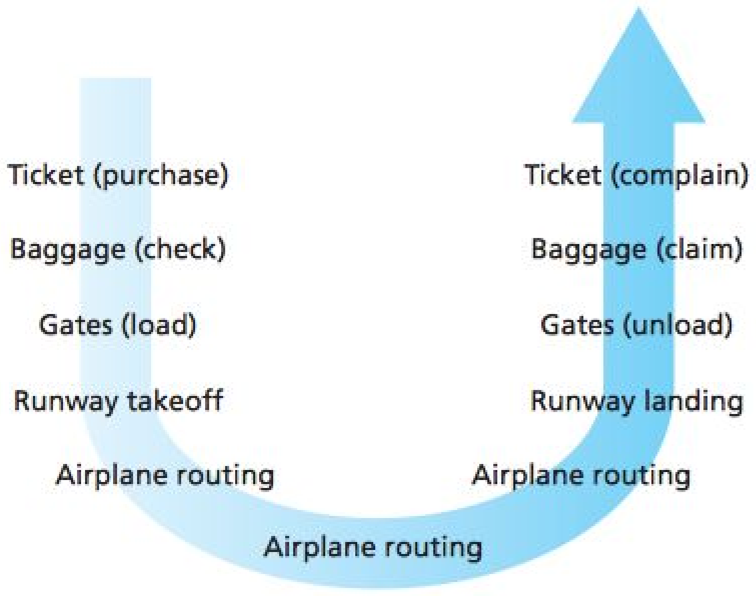
- Idea of layers is simple with no learning curve as it separates tasks by concerns. Moreso useful with low experienced teams
- Use cases are easy/obvious enough
- Minimal knowledge transfer from one layer project to another layer project if components are well defined beforehand
- Don't need a built in scalability

<u>Effect on NFP's</u>
Scalability (inhibits): does not inherently support it (need to implement it on your own)
Adaptability (supports): its modularity allows layers to be changed without affecting other layers
Security (supports): the encapsulation of each layer allows layers to function independently and only provides required services for adjacent layers
Efficiency (inhibits): layers that are not adjacent must go through the layers between, adding additional overhead
Heterogeneity (supports): each layer is independent and serves its own unique functions
Manageability (supports): see above
Reusability (supports): see above
Testability: see above

Examples:

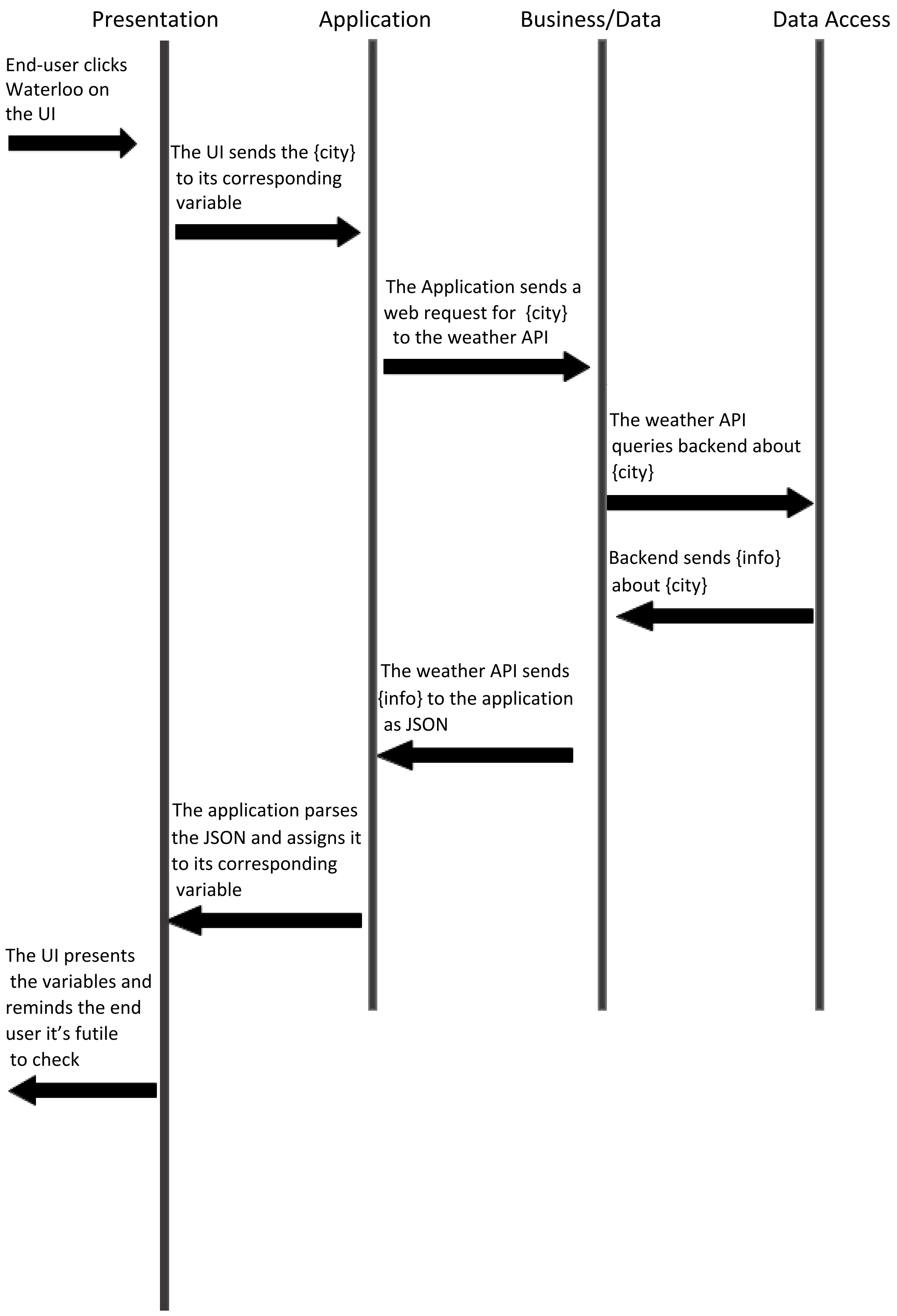
Airport structure:





Layered Architecture Software example:

Knowing how cold it is this week, Lisa searches up the city of Waterloo on her weather app to see if university's classes are cancelled.


Presentation
Application
Business/Data
Data Access
End-user clicks Waterloo on the UI
The UI sends the {city} to its corresponding variable
The Application sends a web request for {city} to the weather API
The weather API queries backend about {city}
Backend sends {info} about {city}
The weather API sends {info} to the application as JSON
The application parses the JSON and assigns it to its corresponding variable
The UI presents the variables and reminds the end user it's futile to check

References

https://en.wikipedia.org/wiki/Multitier_architecture#Layers
https://msdn.microsoft.com/en-ca/library/ee658109.aspx
https://www.eecs.yorku.ca/course_archive/2010-11/F/3213/CSE3213_03_LayeredArchitecture_F2010.pdf
https://www.safaribooksonline.com/library/view/software-architecture-patterns/9781491971437/ch01.html
http://tidyjava.com/layered-architecture-good/
https://msdn.microsoft.com/en-ca/library/ee658117.aspx
https://books.google.ca/books?id=ccK4QJJ4v9YC&pg=PA9&lpg=PA9&dq=topological+constraints+software+layer&source=bl&ots=1FD_Gif9ab&sig=9mQw64Mt_hpp2P7Q3LgTIk8e5cI&hl=en&sa=X&ved=0ahUKEwiqtau0p5DZAhVEymMKHRfnD-4Q6AEIPDAB#v=onepage&q=topological%20constraints%20software%20layer&f=false