UNIVERSITY OF NORTH CAROLINA CHARLOTTE

Lecture

Software Design

ITSC 3155 – Software Engineering
Department of Computer Science
College of Computing and Informatics

Agenda

- Software Design
- Software Design Principles
- High-Level Design
- Software Architecture
- Architecture Patterns
- Low-Level Design
- Design Activities
- Casual Software Design in Practice
- Design Software Using UML

Software Design

User requirements are derived from discussions with stakeholders, which are then refined into system requirements to define the system's functional and non-functional requirements.

System software designs is a blueprint of the system that will be developed.

High-Level Design

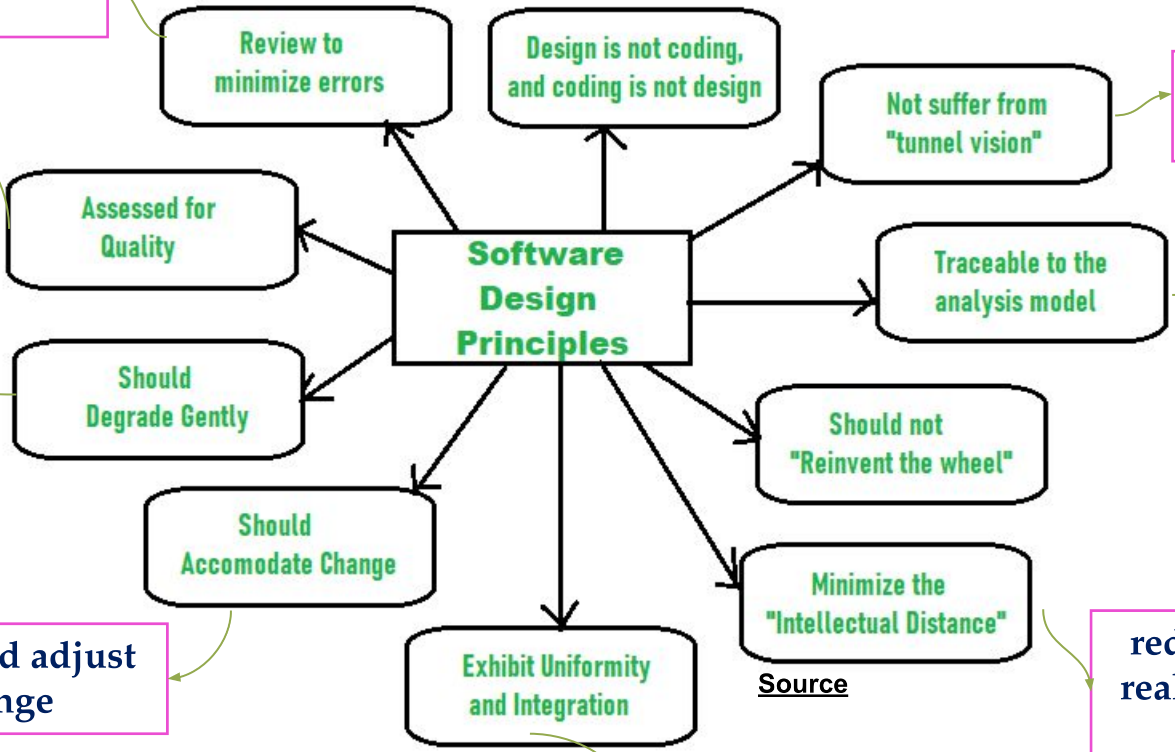
Includes the description of system architecture, database design, brief description on systems, services, platforms and relationship among modules.

Low-Level Design

Describes detailed description of each and every module means it includes actual logic for every system component and it goes deep into each modules specification.

Design is the step where we start applying our plan and idea to real world solutions.

Software Design Principles


Software Design Principles
Review to minimize errors
Design is not coding, and coding is not design
Not suffer from "tunnel vision"
Assessed for Quality
Traceable to the analysis model
Should Degrade Gently
Should not "Reinvent the wheel"
Should Accomodate Change
Minimize the "Intellectual Distance"
Exhibit Uniformity and Integration

Source

The design should be reviewed

during the evaluation, the quality of the design needs to be checked

focusing on both the goal and other effects.

should satisfy all the requirements

it should work properly even if an error occurs

software should adjust to the change

reduce the gap between real-world problems and software solutions

uniform throughout the process without any change and all parts of software should be mixable

Design Activities

1. Architecture Design of system with all subsystems
2. Abstract specification of each subsystem
3. Interface Design for each subsystem
4. Component Design
5. Data Structure Design
6. Algorithm Design

High-Level Design

High-level design shows how the major pieces of the final application will fit and interact at an abstract level.

Software development is a process that chops up the system into smaller and smaller pieces until the pieces are small enough to implement.

High-level design is the first step in the chopping up process.

Image Link

Software Architecture

- Is often modelled with block diagrams
- Very top level of design.
- How software pieces fit together at a high level.
- Link between idea, and reality.

Good Architecture:

- Faster development.
- Reduce overall idle time.
- Maintainable software.

Architecture mistakes cannot be corrected once coding has begun.

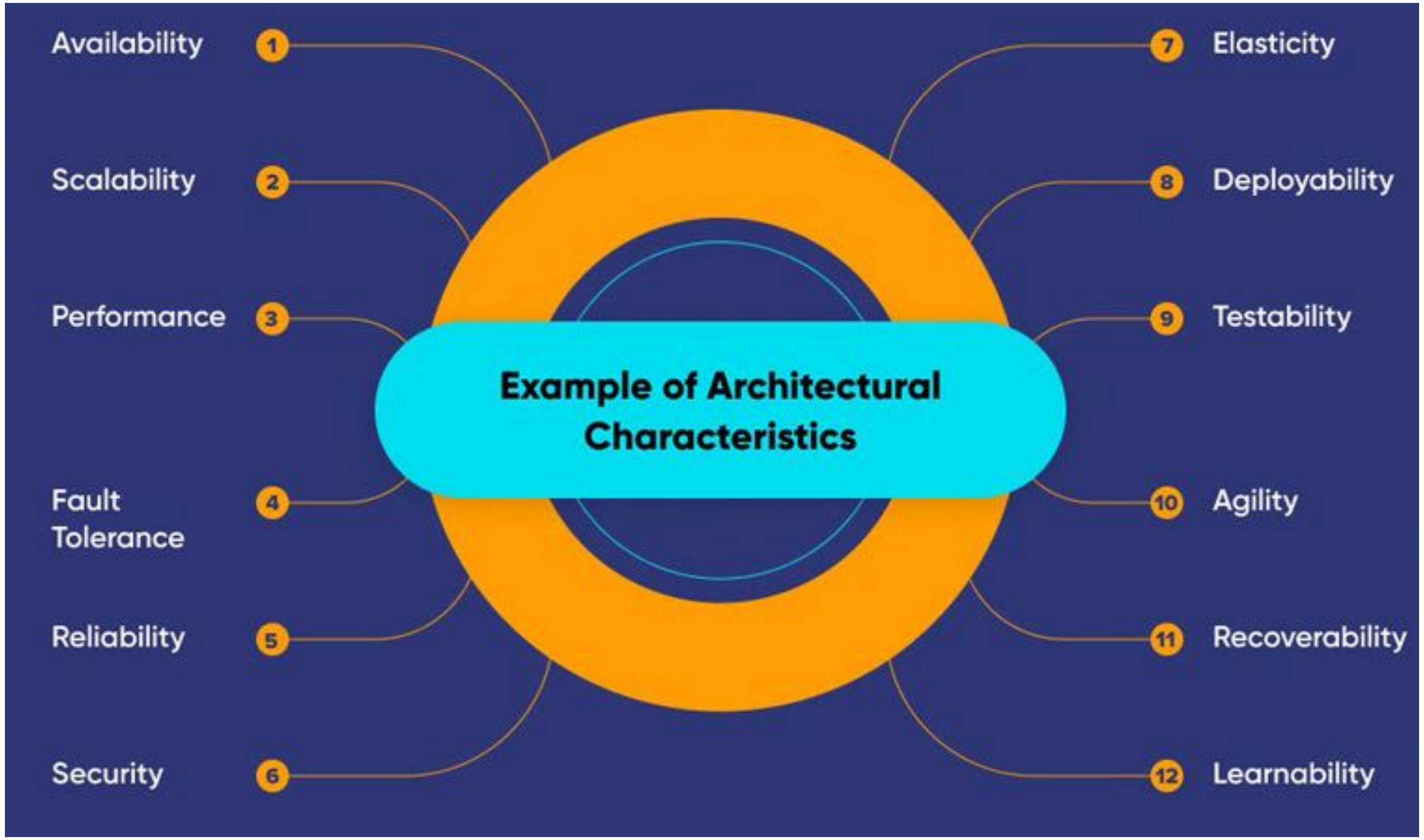
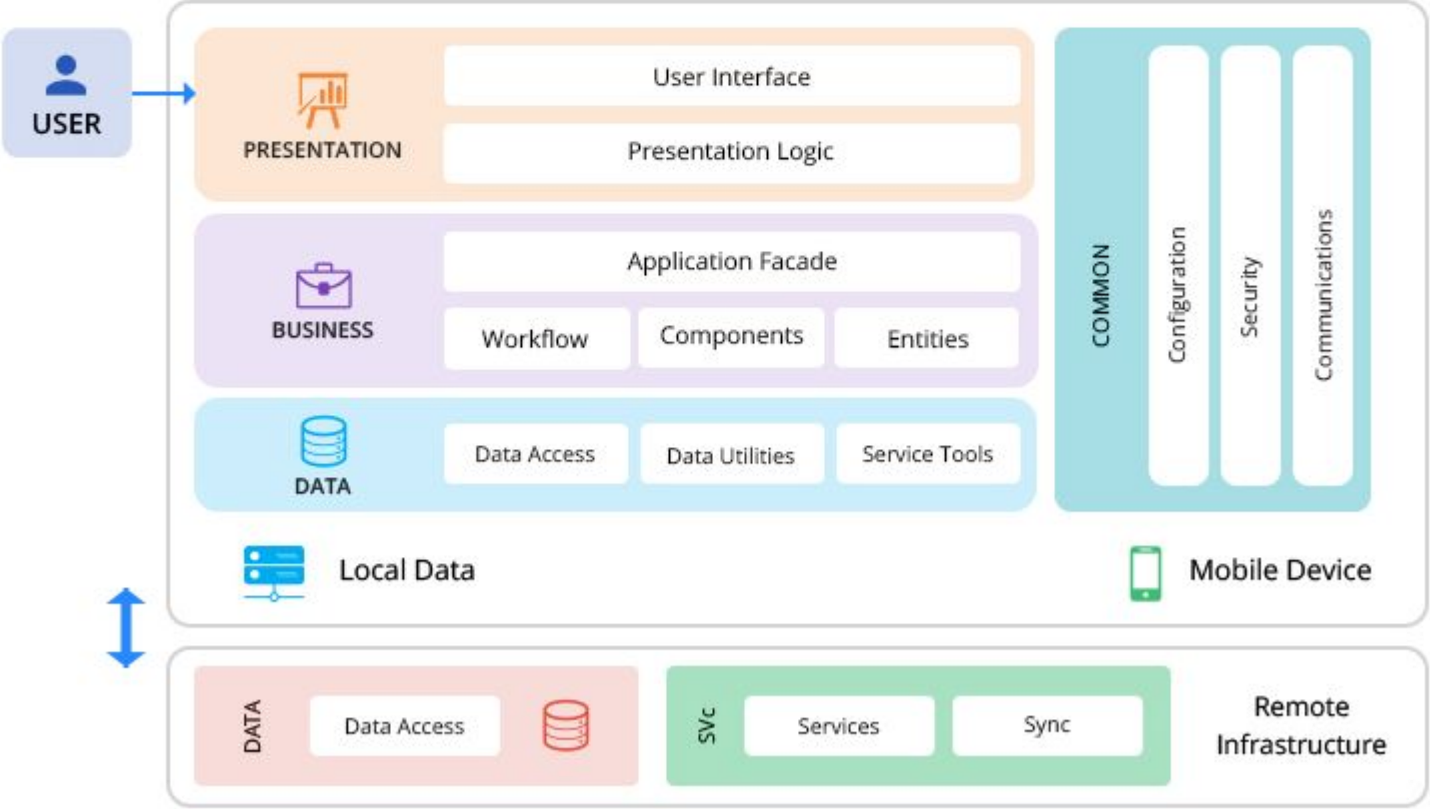


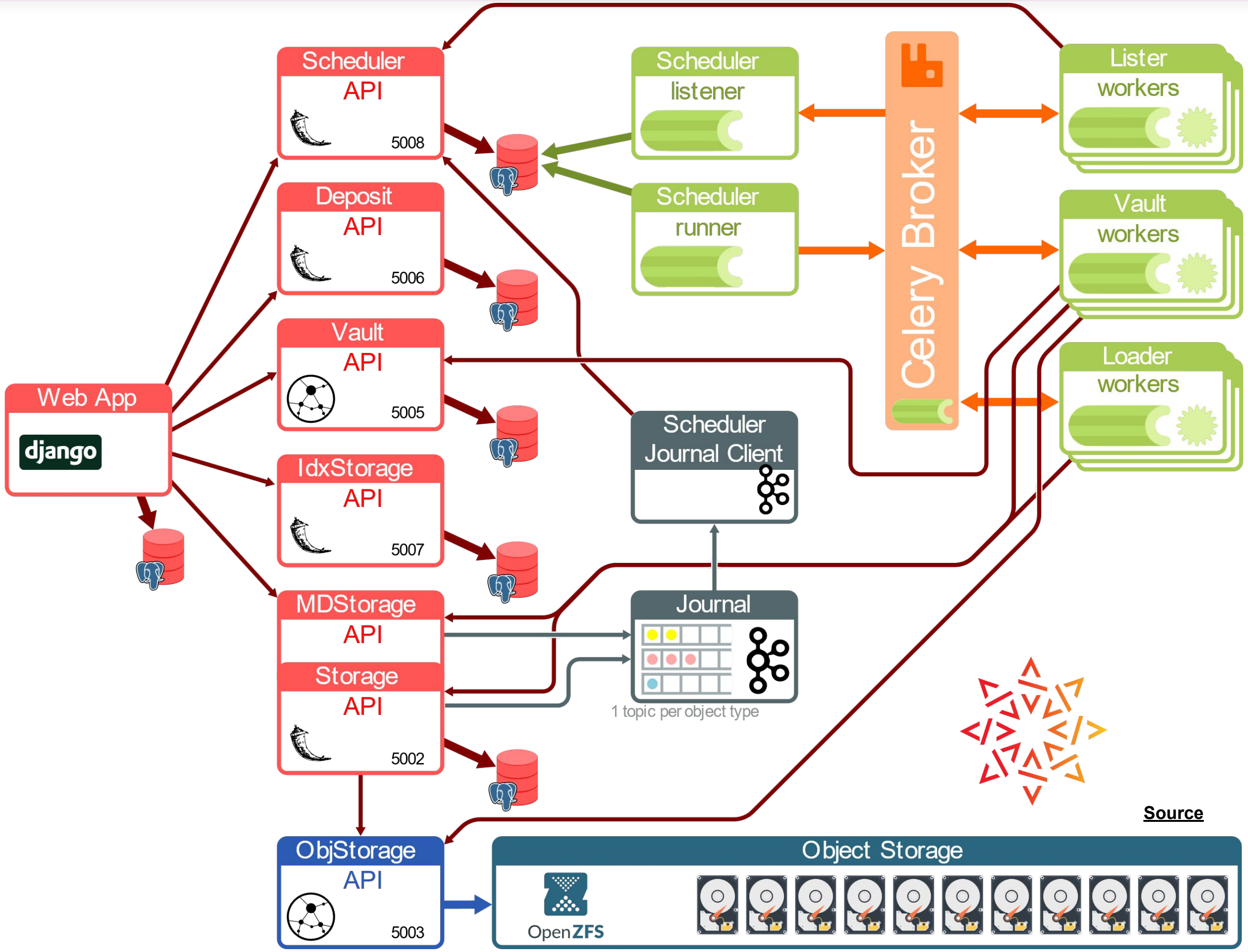
Image link

Software Architecture



Source

Software Architecture



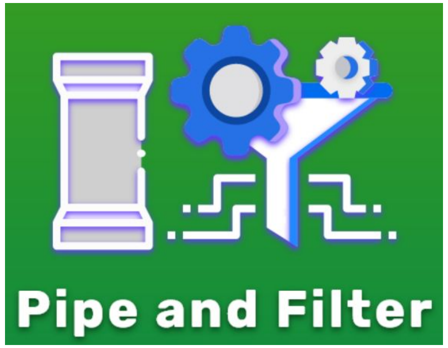
Source

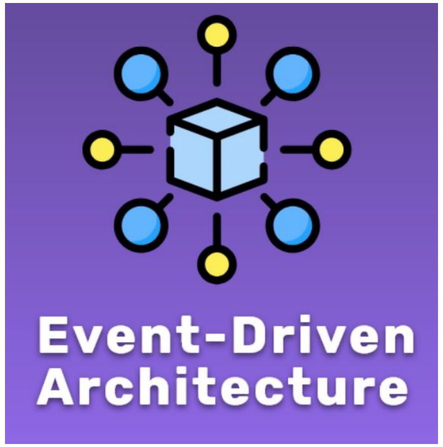
Global,
Read-Only View

Architecture Patterns

“If you think good architecture is expensive, try bad architecture!”

Brian Foote & Joseph Yoder





An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are similar to software design pattern but have a broader scope.

Layered

Architecture Pattern

- Program can be decomposed into groups of subtasks, each of which is at a particular level of abstraction.
- Each layer provides services to the next higher layer.
- E-commerce, desktop, and other applications that include groups of subtasks that execute in a specific order.
- Easy to write applications quickly, but it can be hard to split up the layers later.

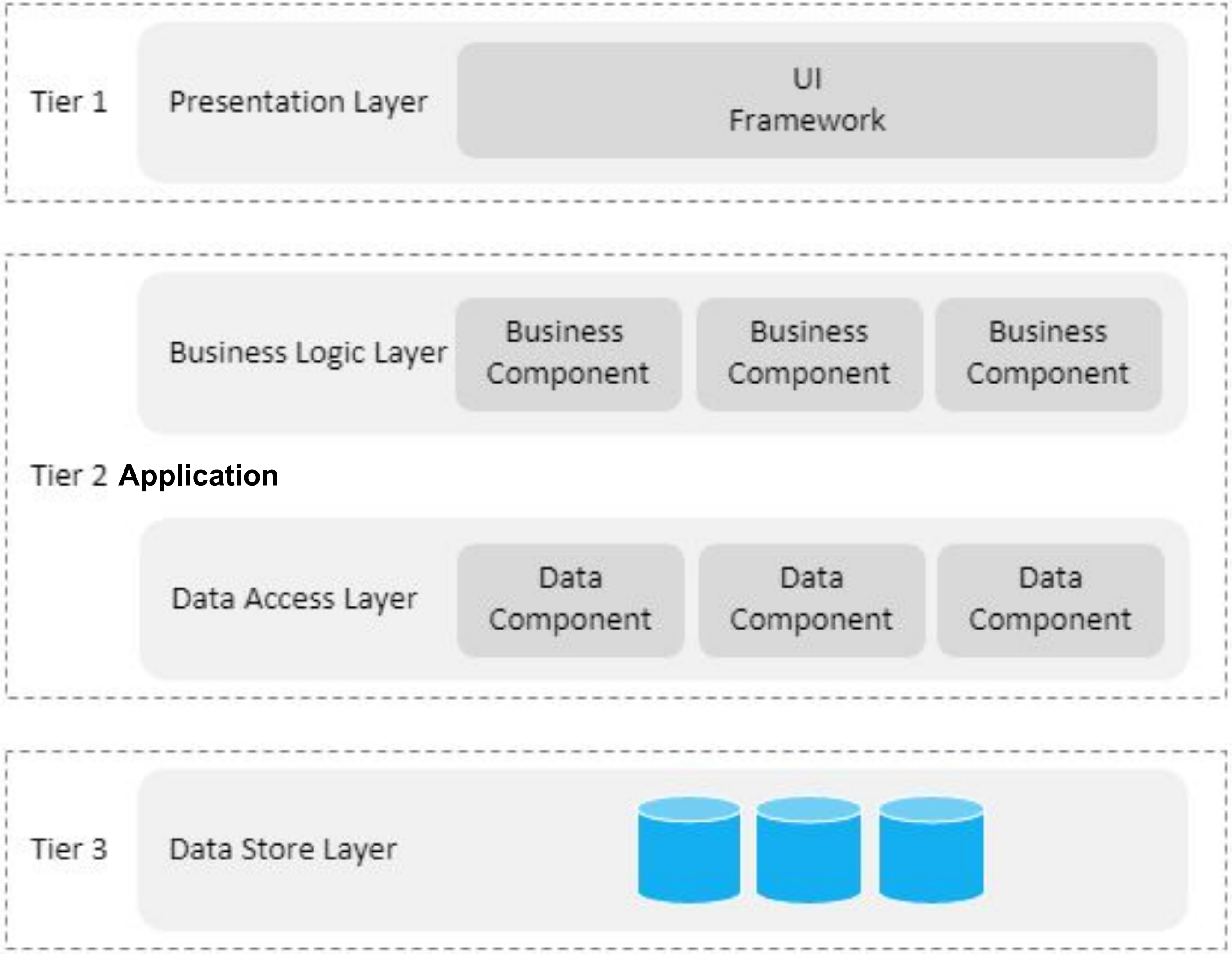


Image link

Layered

Architecture Pattern

Advantages

- We only need to understand the layers beneath the one we are working on.
- Each layer is replaceable by an equivalent implementation, with no impact on the other layers.
- Layers are optimal candidates for standardization.
- A layer can be used by several different higher-level layers.

Disadvantages

- Layers can not encapsulate everything (a field that is added to the UI, most likely also needs to be added to the DB.
- Extra layers can harm performance, especially if in different tiers.

Client-Server

Architecture Pattern

- This architecture pattern helps to design distributed systems that involve a client system, a server system, and a connecting network.
- Online applications such as email and document sharing.
- Increased overhead, Shared server is often a performance bottleneck.

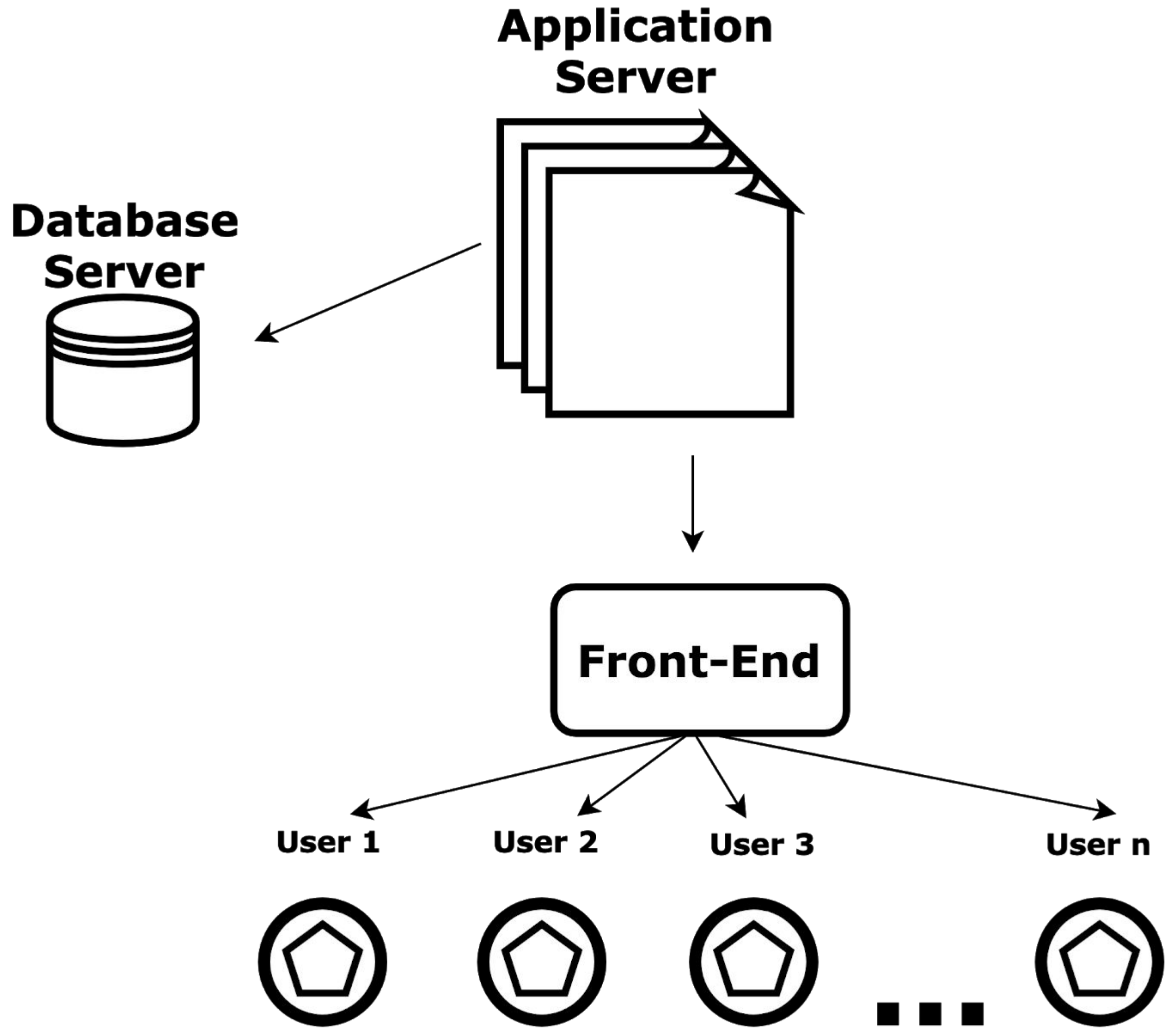


Image link

MVC
Architecture Pattern

- Web frameworks such as Django Laravel,…
- Organizes large-size web applications.
- Easily Modifiable
- Faster Development Process.
- Easy planning and maintenance.
- Supports TDD (test-driven development).

MVC Architecture Pattern

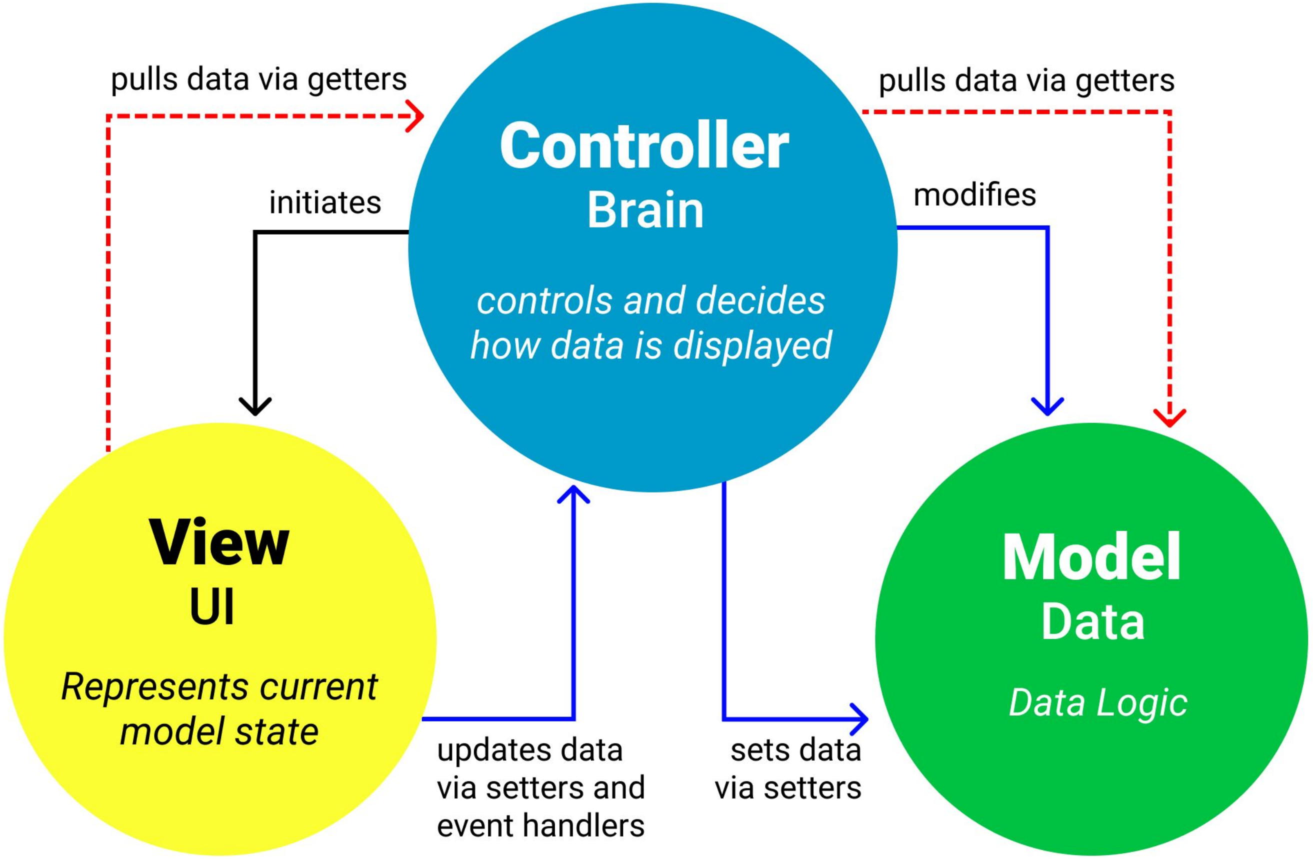
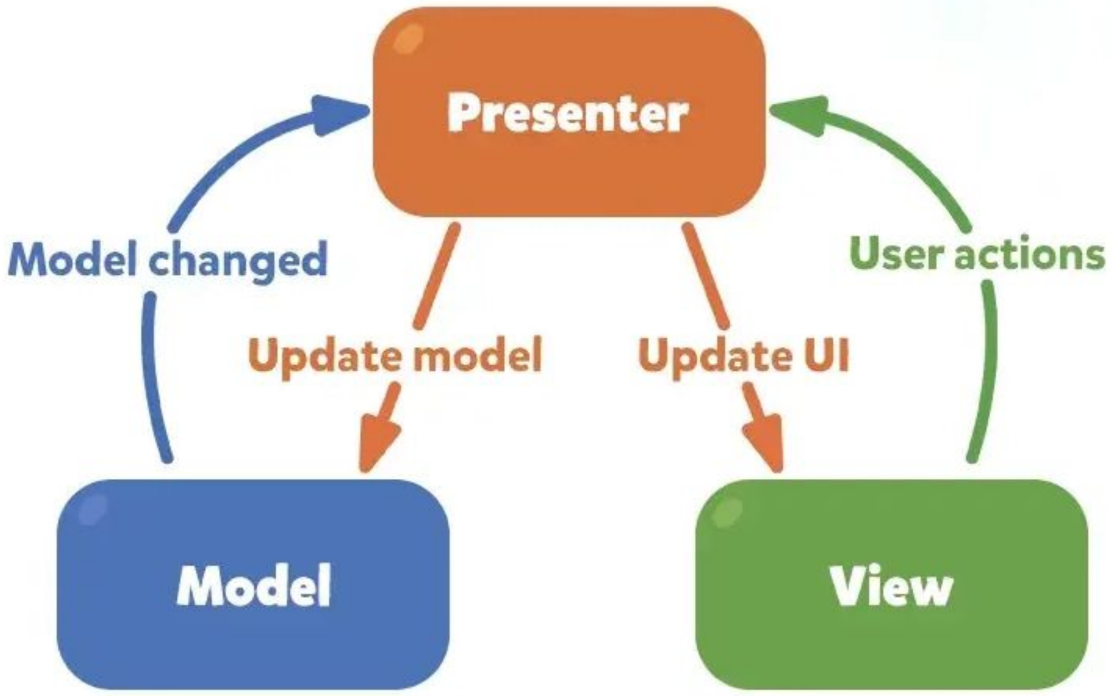
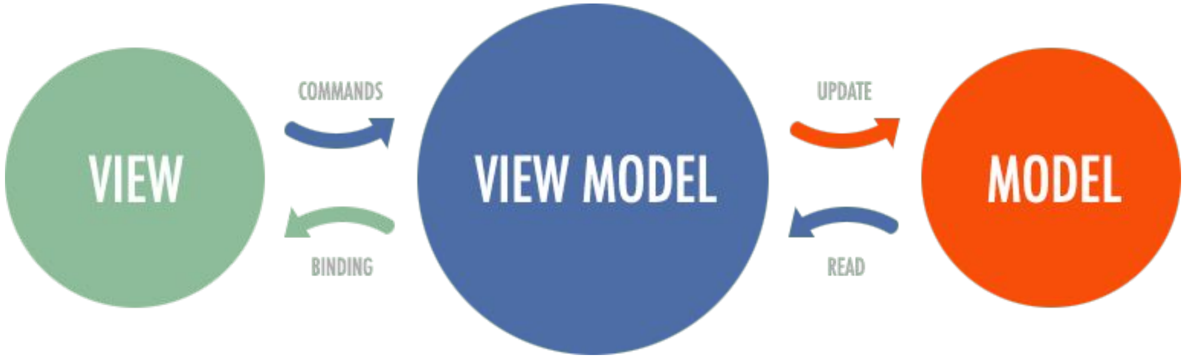


Image link

MVP and MVVM Architecture Pattern

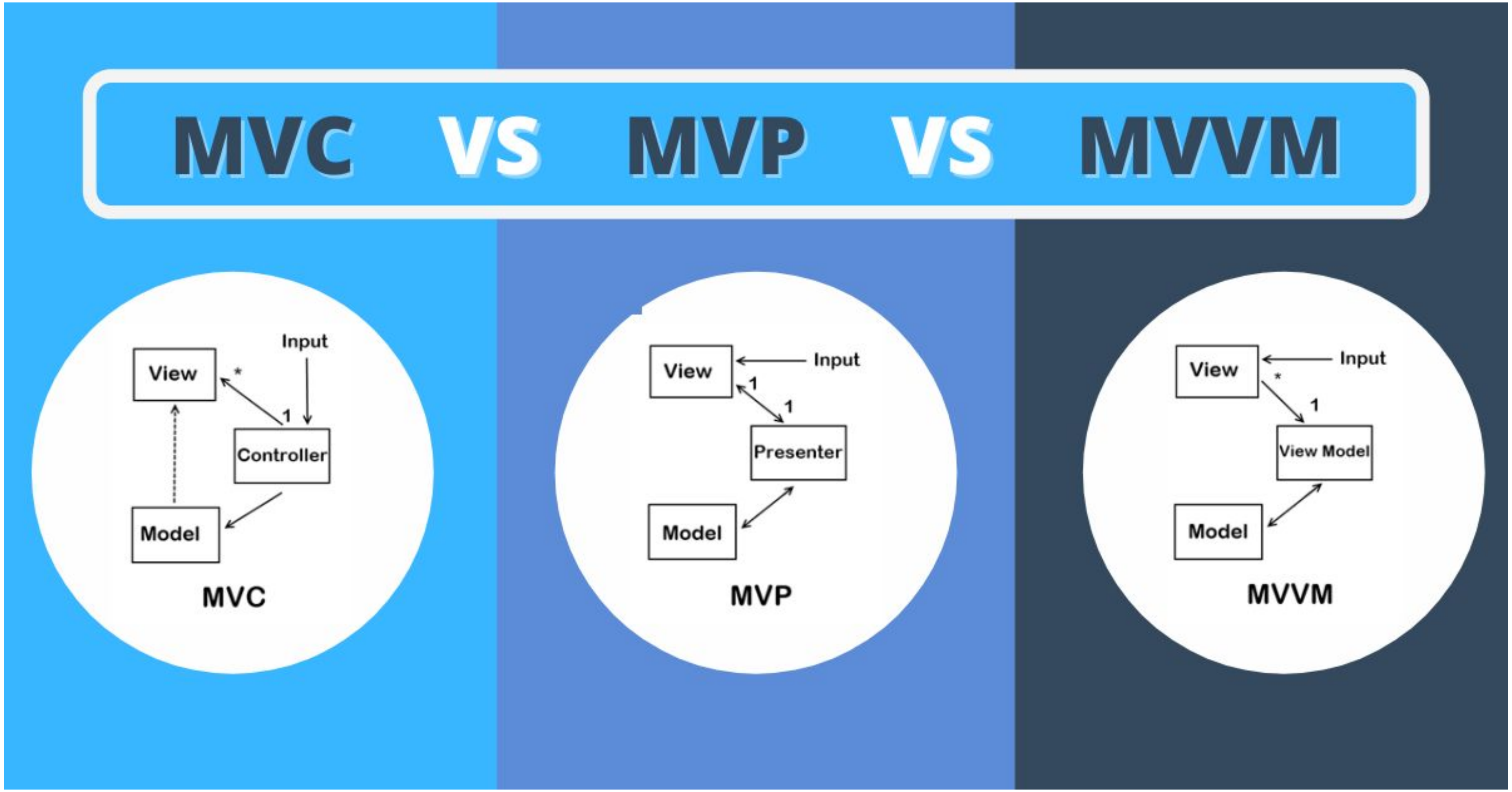


In Android, we have a problem arising from the fact that Android activities are closely coupled to both UI and data access mechanisms.



- Does not hold any kind of reference to the View.
- Many to-1 relationships exist between View and ViewModel.
- No triggering methods to update the View.

MVP and MVVM Architecture Pattern



Microservices

Architecture Pattern

Microservice are small business services that can work together and can be deployed independently.

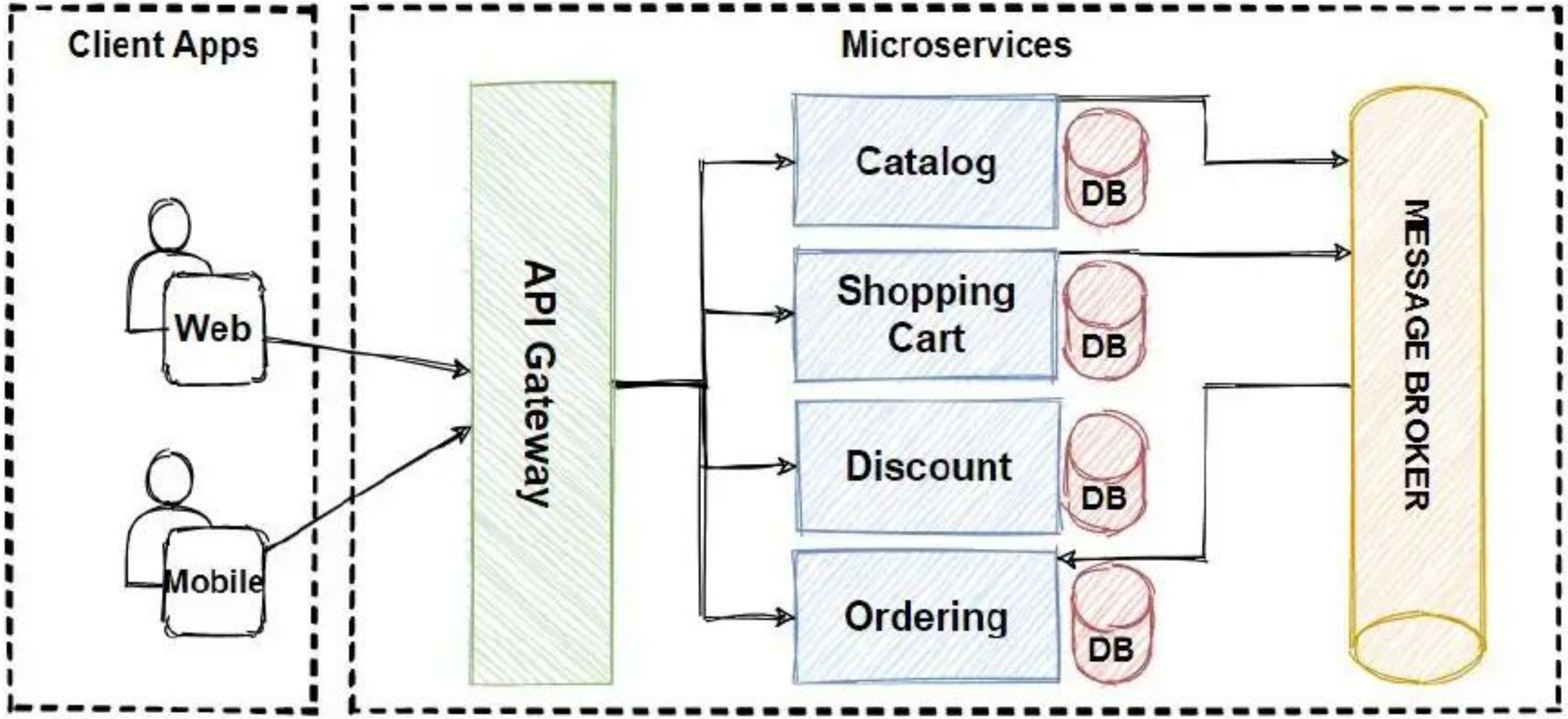


Image link

UNIVERSITY OF NORTH CAROLINA CHARLOTTE

Microservices
Architecture Pattern

Benefits

- Agility
- Small, focused teams
- Small and Separated Code Base
- Right tool for the job
- Fault Isolation
- Scalability
- Data isolation

Drawbacks

- Complexity
- Network problems and latency
- Development and Testing
- Data Integrity

Low-Level Design

Low‑level design moves the high‑level focus from what to a lower level focus on how

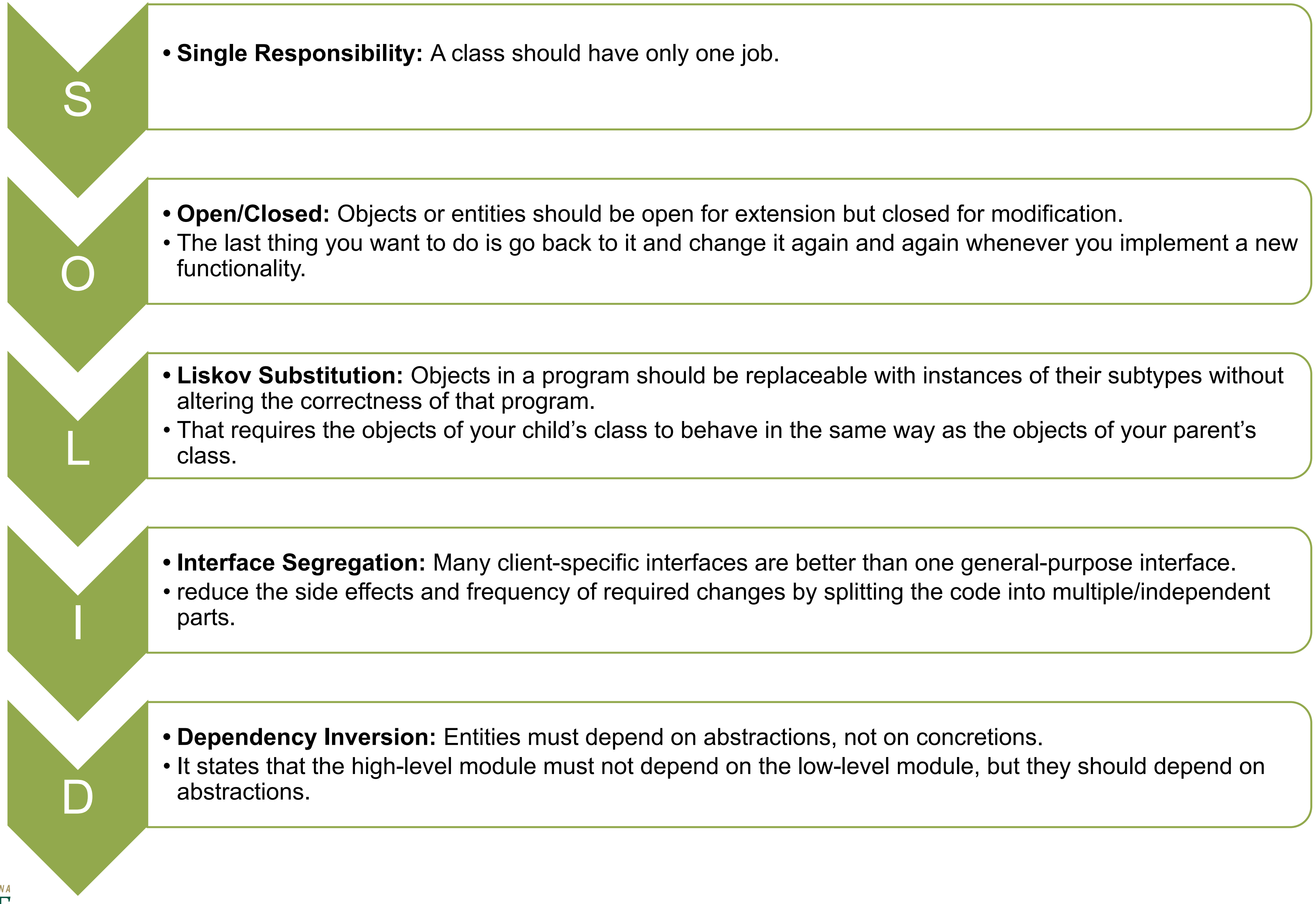
- **Function-Oriented Design:**
  - Involves starting with a high-level view of the system and refining it into a more detailed design.
  - The system state is centralized and shared between the functions operating on that state.
- **Object-Oriented Design:**
  - The system is viewed as a collection of objects rather than functions, with each object managing its own state information.
  - The system state is decentralized and an object is a member of an object class.
- **User Interface Design:**
  - The user interface is the boundary between the user and the system.

Low-Level Design

Low-level design moves the high-level focus from what to a lower level focus on how

- **Open-Source Design:**
  - The idea is that the source code is not proprietary, but is freely available for software developers to use and modify as they wish.
  - It offers a way to speed up software development, as well as potentially providing a high-quality cost-effective solution.
- **Database Design:**
  - Determines what tables the database contains and how they are related.

SOLID Principles



S

- **Single Responsibility:** A class should have only one job.

O

- **Open/Closed:** Objects or entities should be open for extension but closed for modification.
- The last thing you want to do is go back to it and change it again and again whenever you implement a new functionality.

L

- **Liskov Substitution:** Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- That requires the objects of your child’s class to behave in the same way as the objects of your parent’s class.

I

- **Interface Segregation:** Many client-specific interfaces are better than one general-purpose interface.
- reduce the side effects and frequency of required changes by splitting the code into multiple/independent parts.

D

- **Dependency Inversion:** Entities must depend on abstractions, not on concretions.
- It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

Casual Software Design in Practice

Break larger problem into smaller into smaller problems

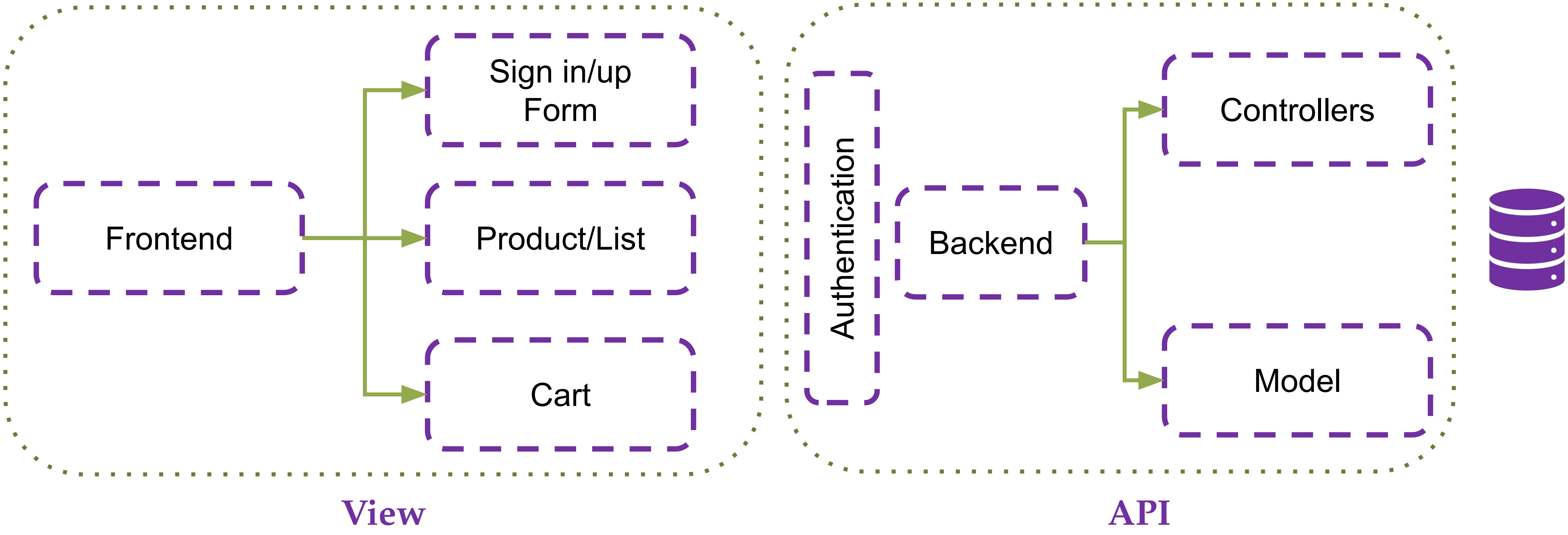
Frontend

Backend

Database

Casual Software Design in Practice

Design architecture and component based on requirements



Casual Software Design in Practice

Identify potential solutions

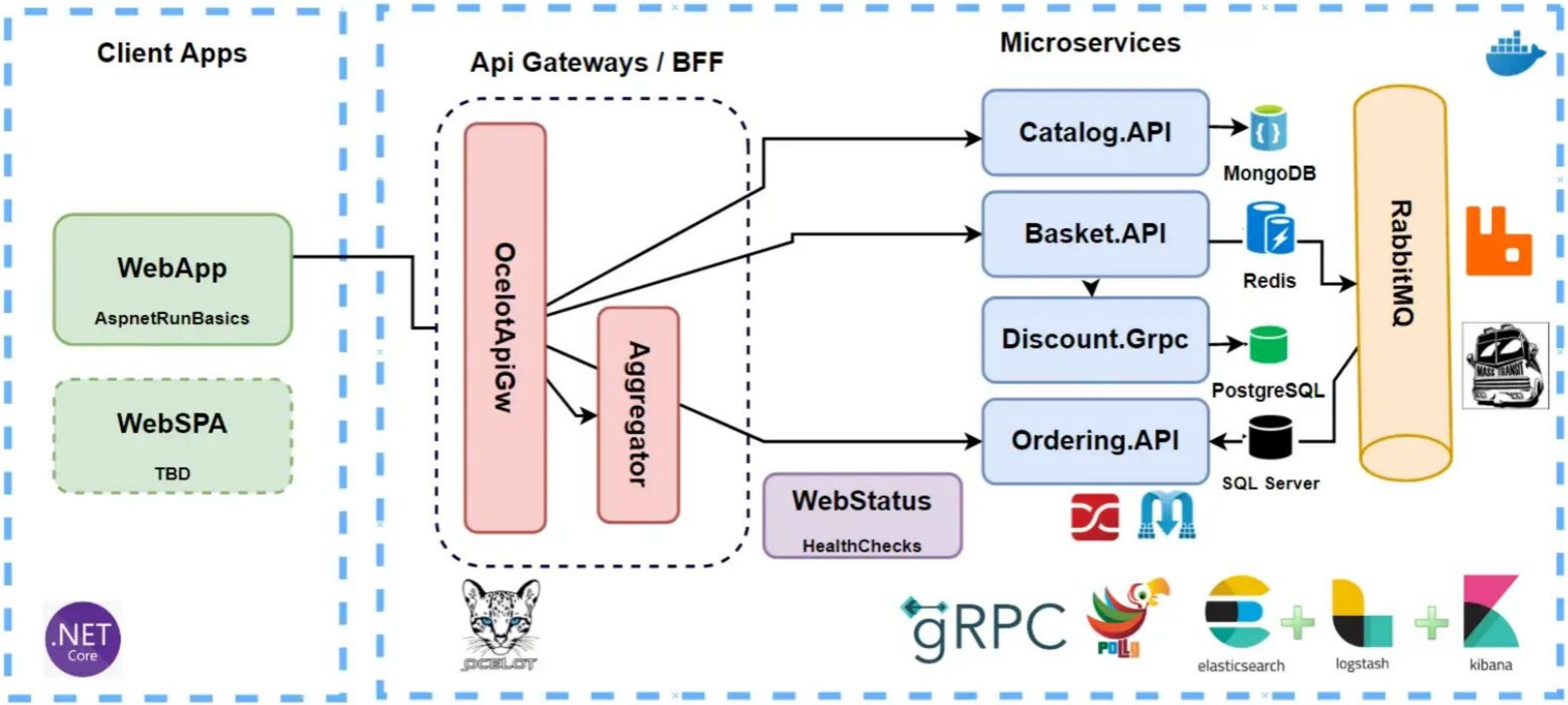


Image link

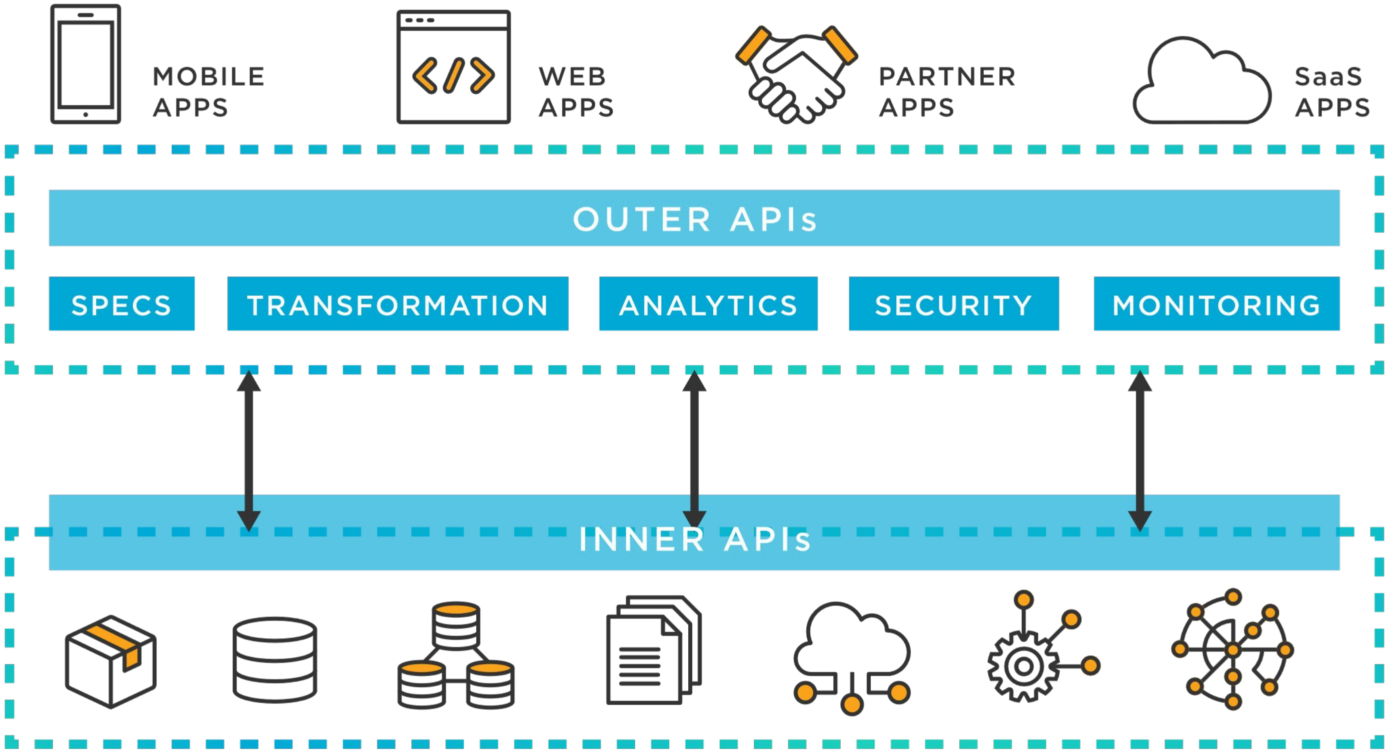
Casual Software Design in Practice

Identify potential solutions

aws

MySQL

redis



MOBILE APPS

WEB APPS

PARTNER APPS

SaaS APPS

API MANAGEMENT SYSTEM

OUTER APIs

SPECS

TRANSFORMATION

ANALYTICS

SECURITY

MONITORING

INNER APIs

DATA SOURCES

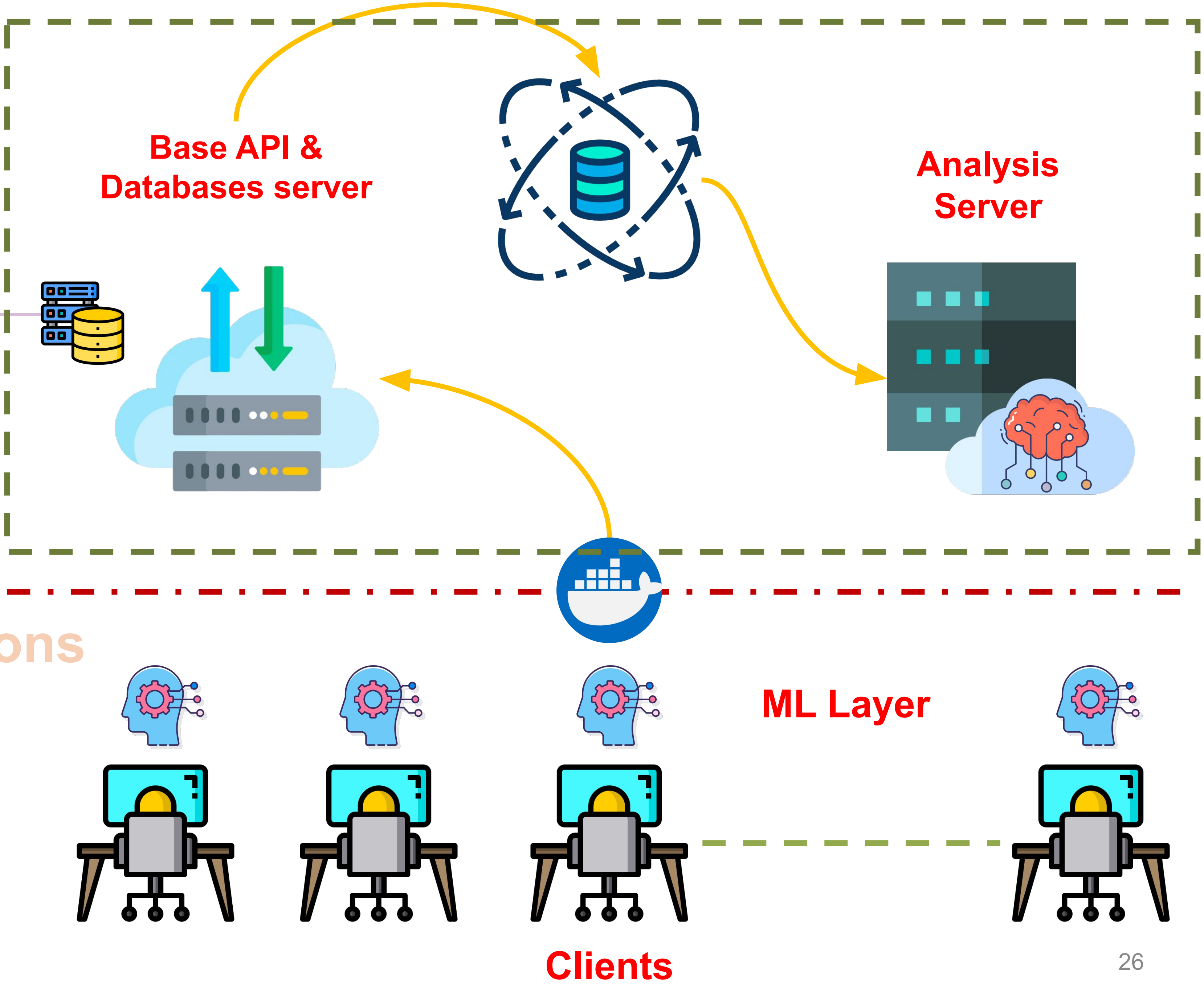
React JS

Flask

JWT

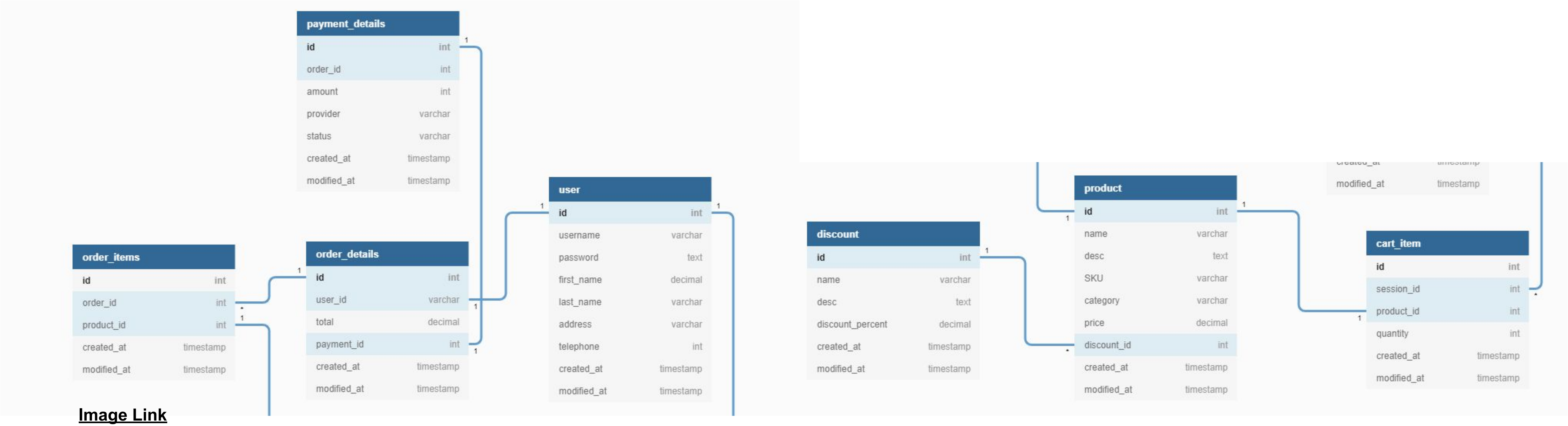
Casual Software Design in Practice

Identify potential solutions



Casual Software Design in Practice

Describe solution abstractions and design database tables



payment_details	
id	int
order_id	int
amount	int
provider	varchar
status	varchar
created_at	timestamp
modified_at	timestamp

order_items	
id	int
order_id	int
product_id	int
created_at	timestamp
modified_at	timestamp

order_details	
id	int
user_id	varchar
total	decimal
payment_id	int
created_at	timestamp
modified_at	timestamp

user	
id	int
username	varchar
password	text
first_name	decimal
last_name	varchar
address	varchar
telephone	int
created_at	timestamp
modified_at	timestamp

discount	
id	int
name	varchar
desc	text
discount_percent	decimal
created_at	timestamp
modified_at	timestamp

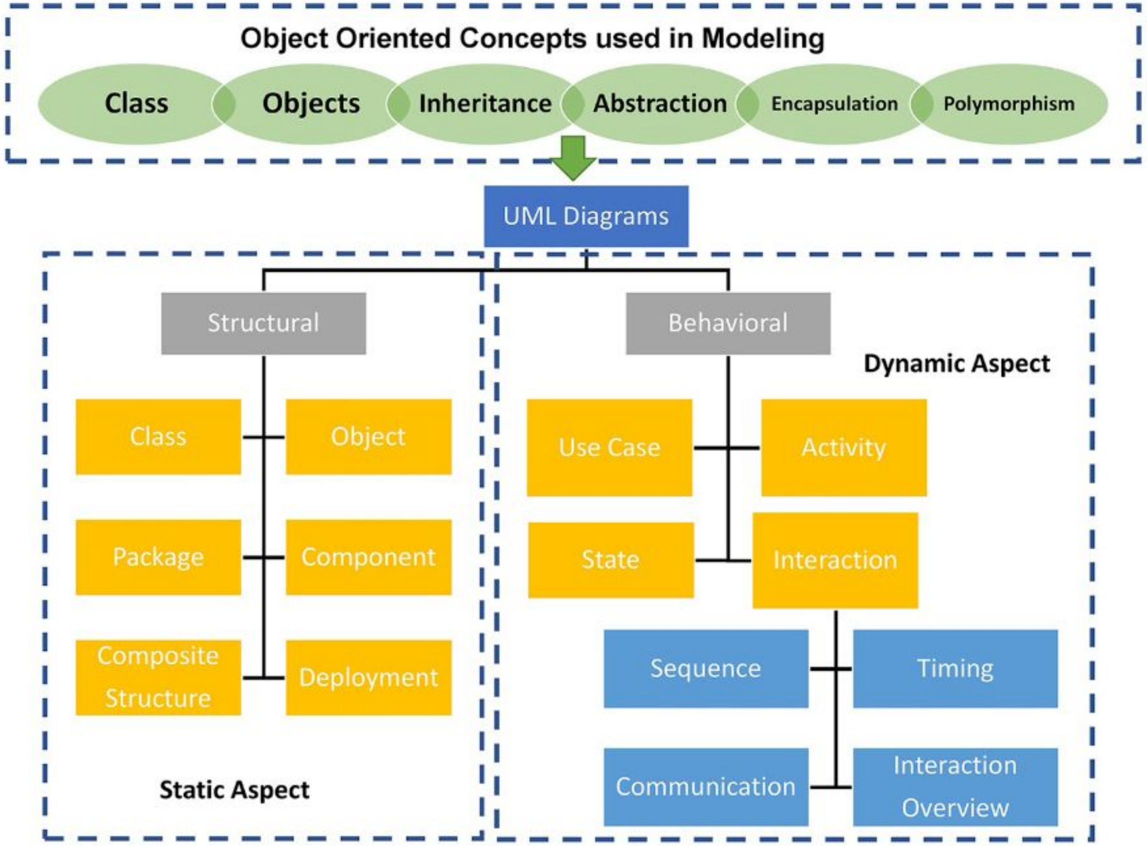
product	
id	int
name	varchar
desc	text
SKU	varchar
category	varchar
price	decimal
discount_id	int
created_at	timestamp
modified_at	timestamp

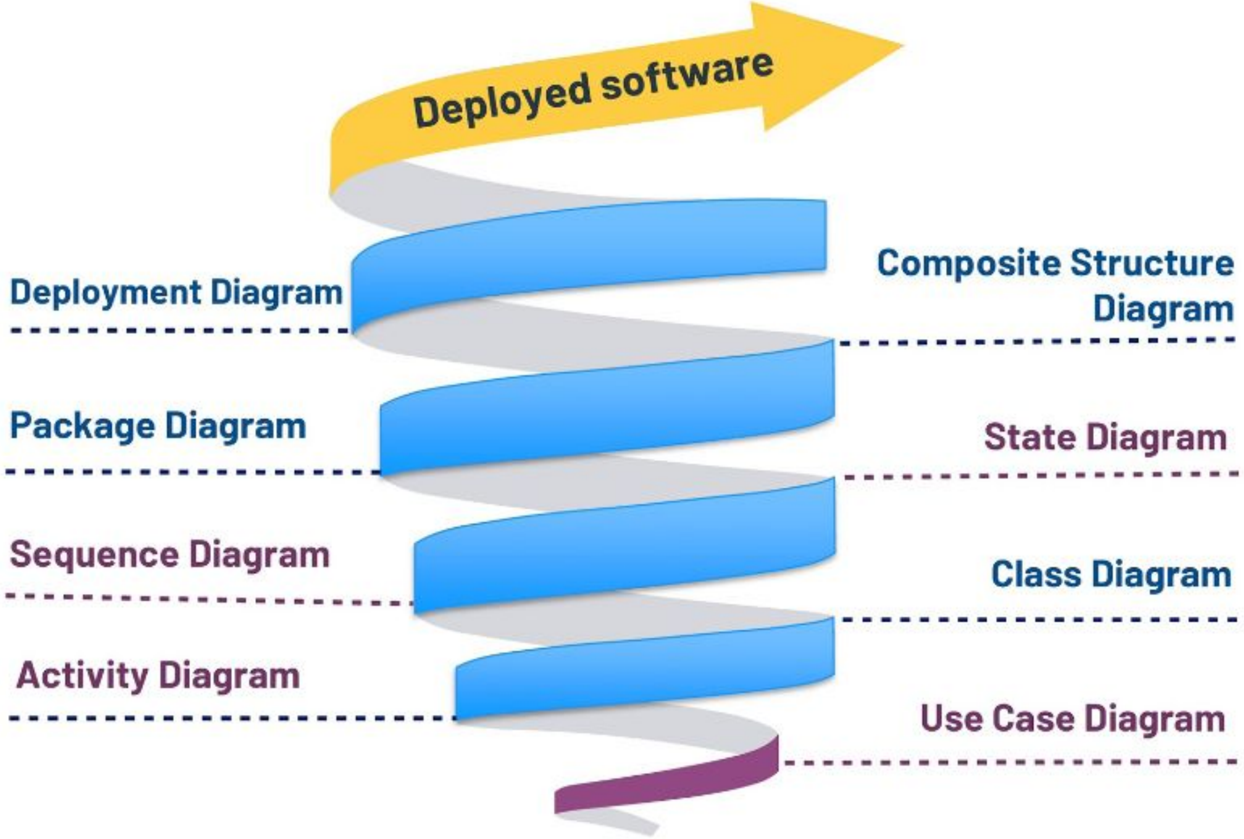
created_at	timestamp
modified_at	timestamp

cart_item	
id	int
session_id	int
product_id	int
quantity	int
created_at	timestamp
modified_at	timestamp

Image Link

Design Software Using UML

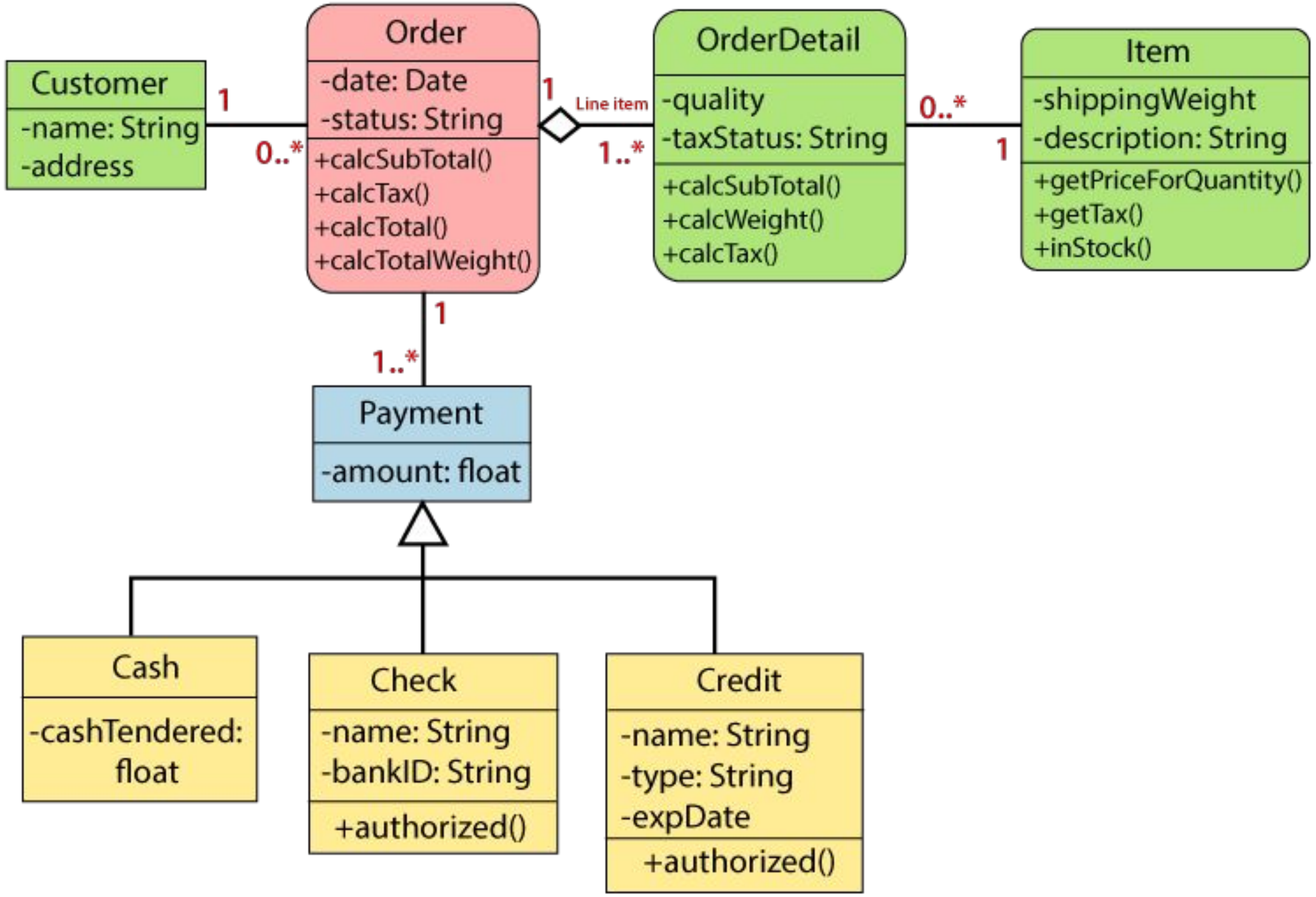




Reference

Class Diagram

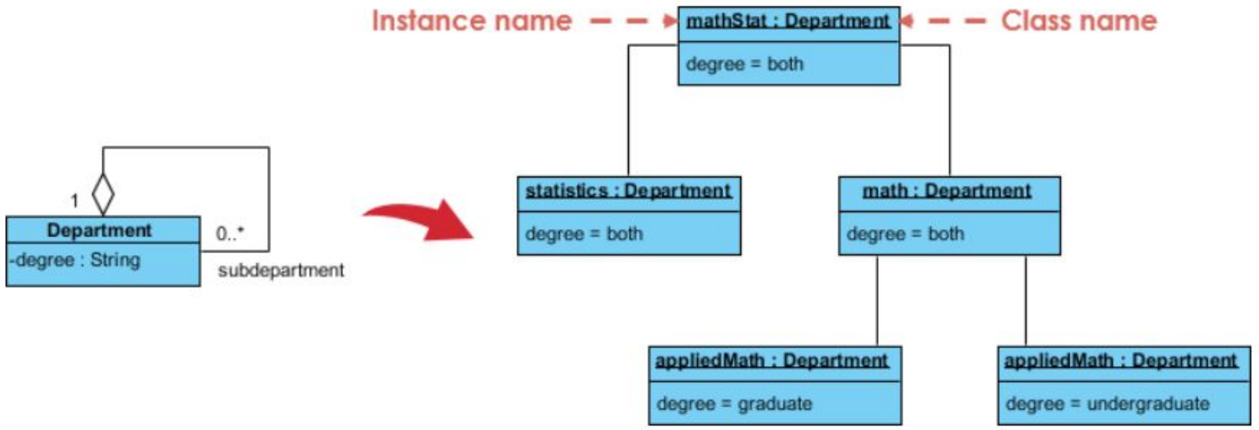
A class diagram represents a static view of the system. It describes the attributes and operations of classes.



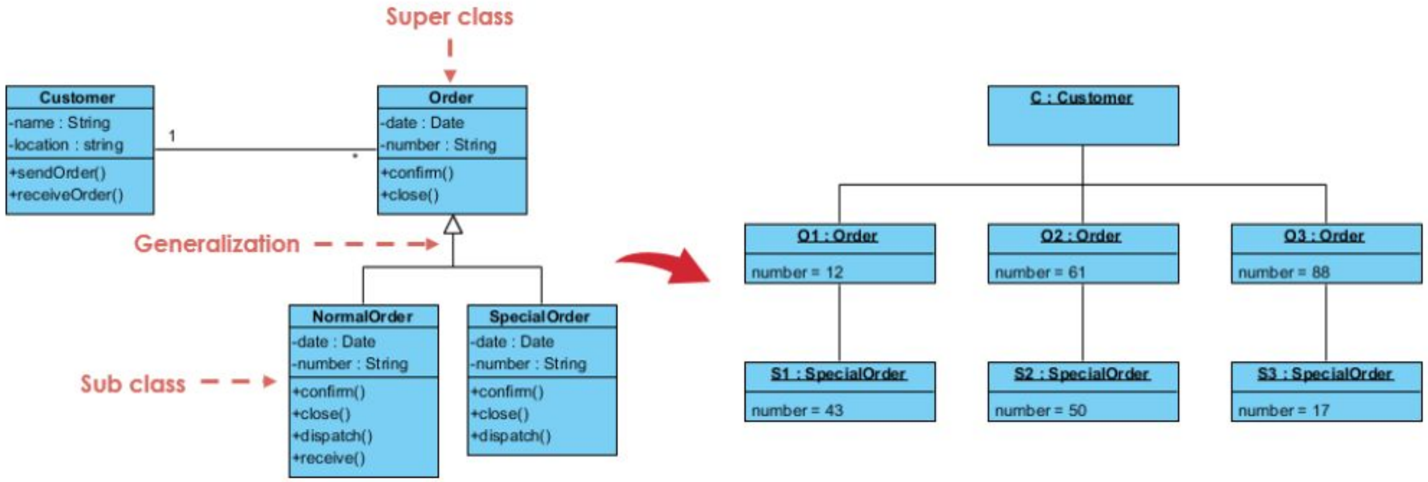
Source

Object Diagram

The basic concepts of object diagram are similar to a class diagram. These diagrams help to understand object behavior and their relationships at a particular moment.



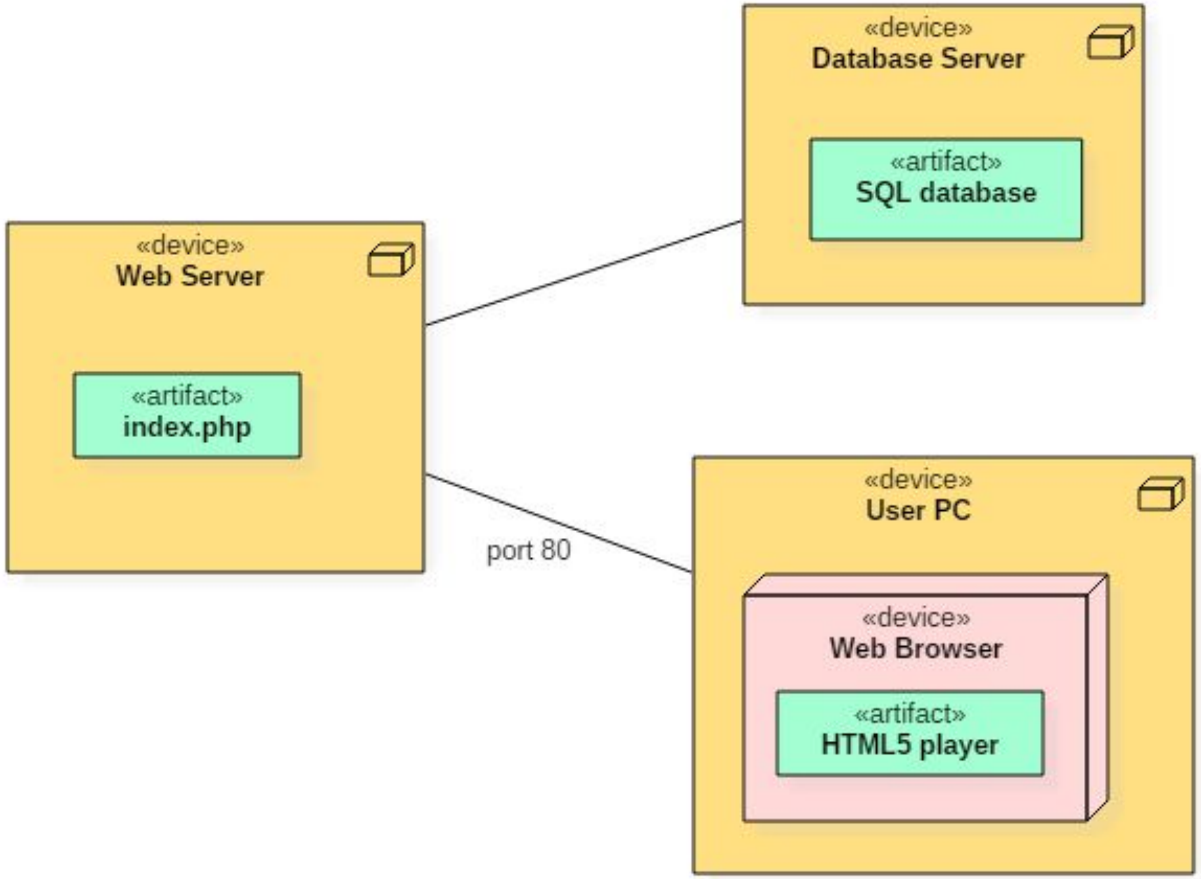
Class to Object Diagram Example - Order System



Source

Deployment Diagram

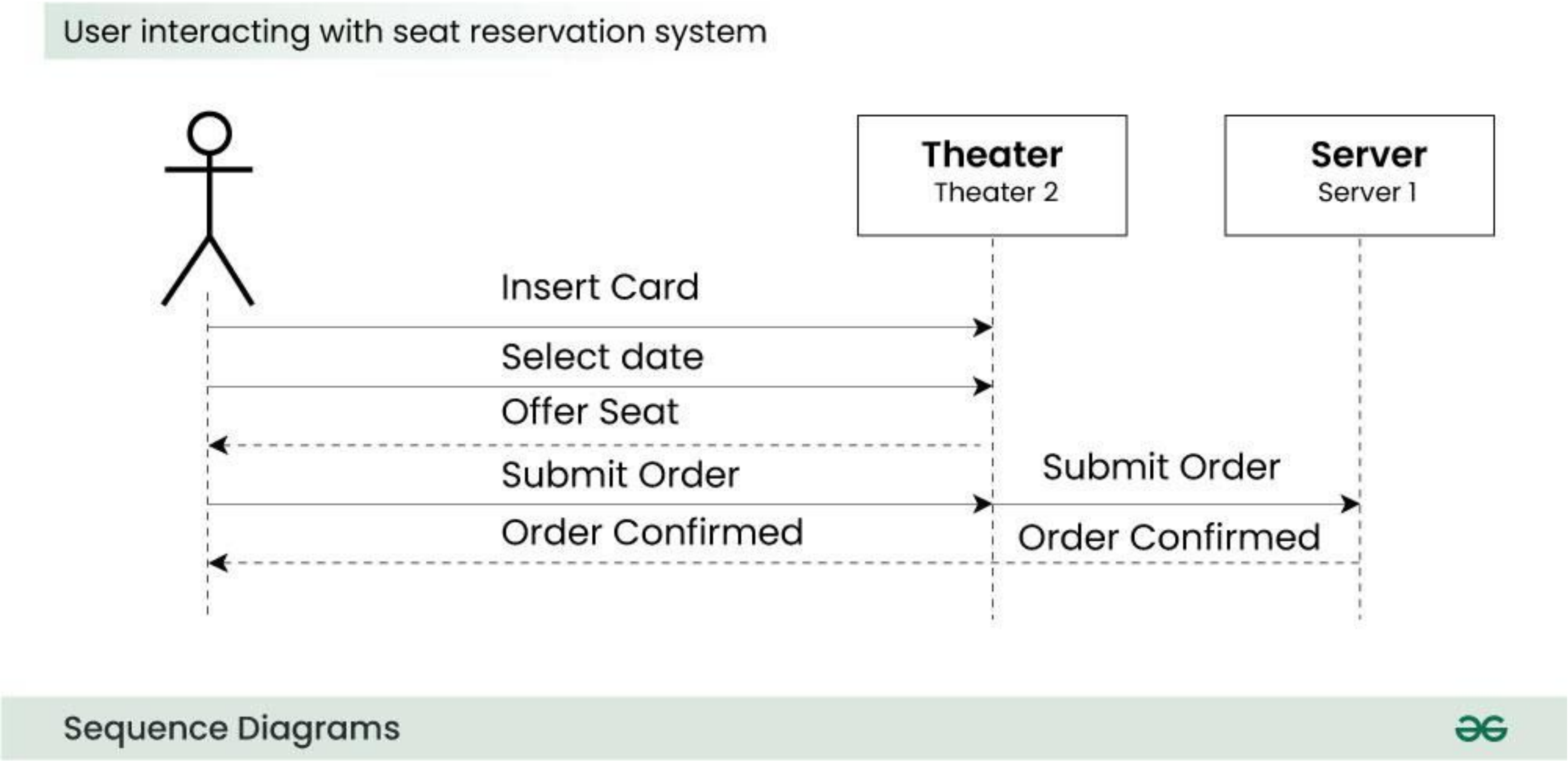
Specifies the physical hardware on which the software system will execute and how the software is deployed on the underlying hardware.



Source

Sequence Diagram

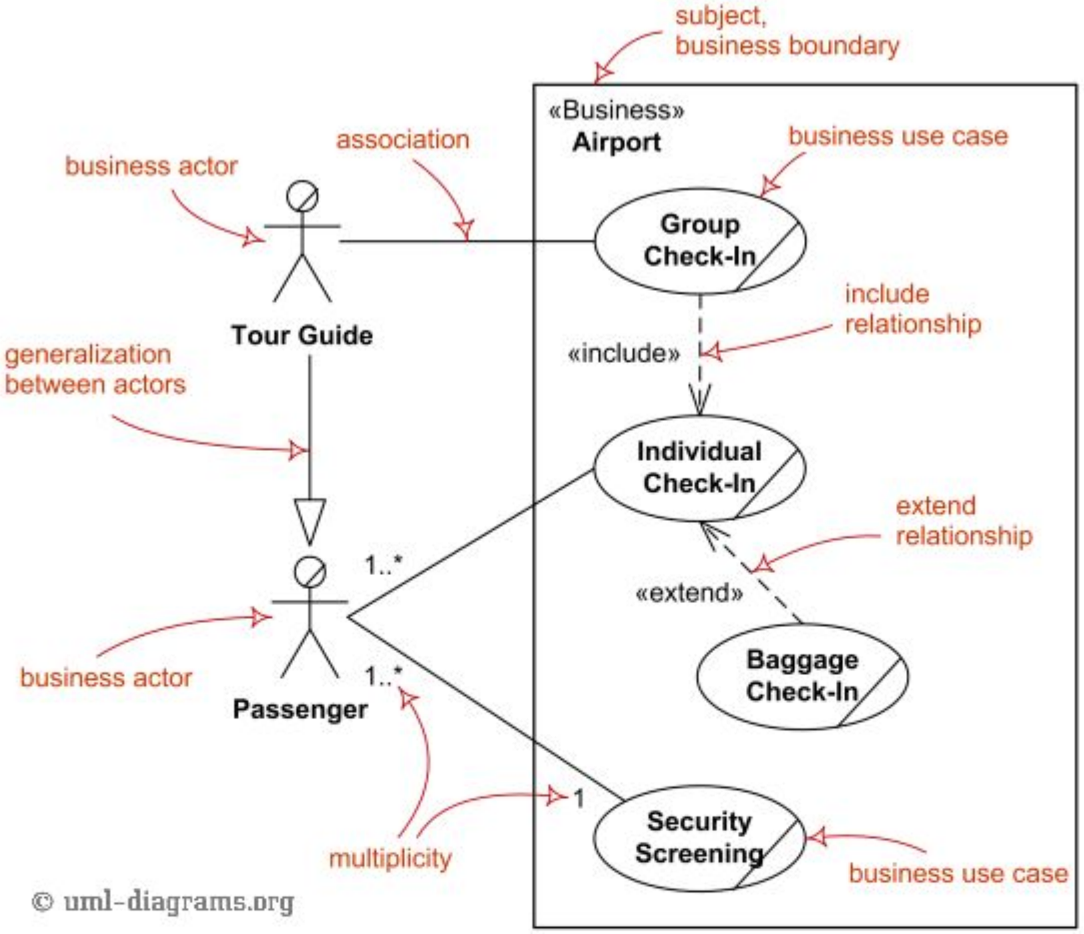
Used to visualize the interaction between objects in a sequential order, focusing on how objects communicate with each other over time.



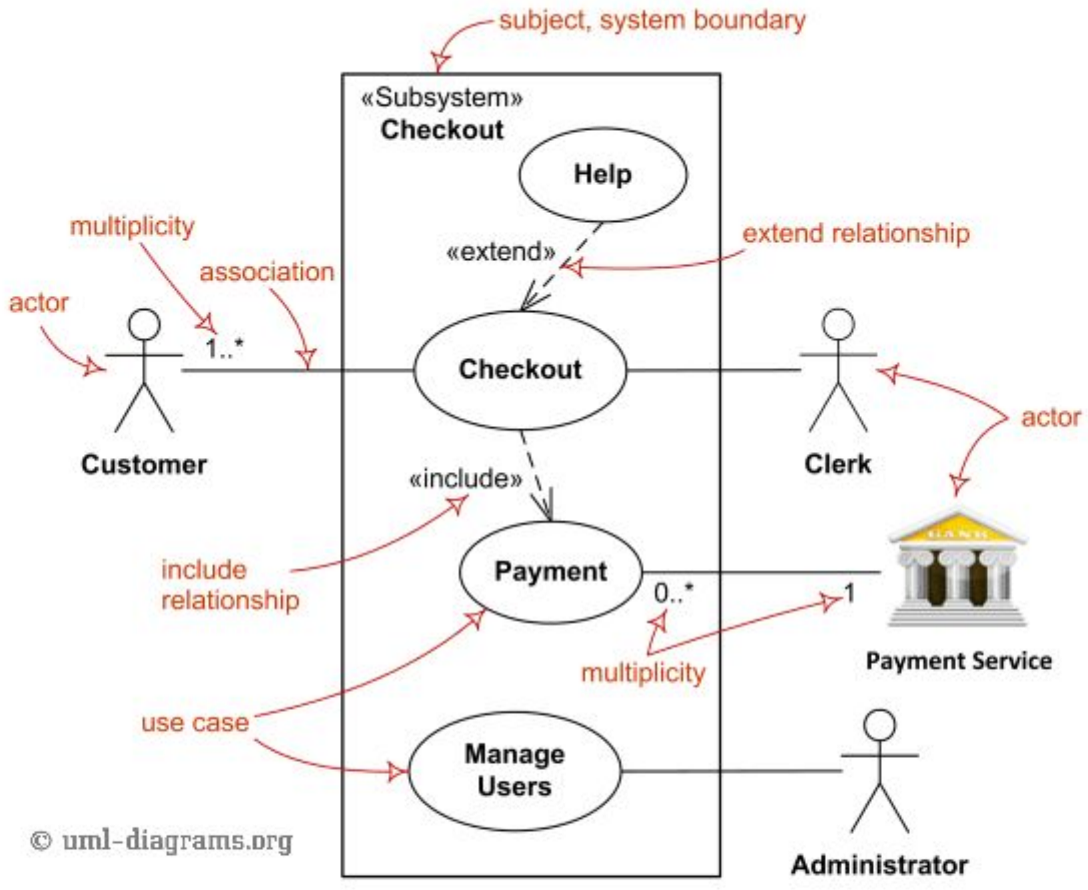
Source

Use Case Diagram

Summarize the details of your system's users (also known as actors) and their interactions with the system (x)



Business Use Case



System Use Case

Source