Introduction to Data Engineering

Unit 1.5:
Basics of Software Architecture

Lesson Objectives

01 Understand the principles of enterprise architecture

02 Know the basics of software architecture

03 Choose between different types of data

04 Decide when and which existing solutions to use

Contents

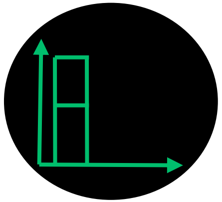
General Architectural Concepts

Reliability

- **Providing all the expected functions** - the solution meets its functional requirements
- **Performance** - providing the functions in expected time, under expected load and volume
- **Resistance to hardware and software faults** - redundancy and testing
- **Resistance to user mistakes** - interface design and validation
- **Security** - preventing unauthorized access

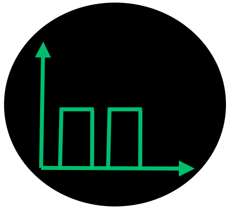
Scalability

Ability to expand the system in such a way that it works with an acceptable capacity



Vertical scaling

adding more resources to a node



Horizontal scaling

adding more nodes

Maintainability

- **Monitoring** - observing logs, metrics and traces
- **Recovery** - procedures for restoring the functions
- **Configuration** - enables customization and tuning without updating the software
- **Simplicity** - easy for further development
- **Avoiding technical debt** - workaround/hot fixes vs. long-term solutions

Managing Complexity

- **Separation of concerns**
  - Dividing solution into smaller parts
  - Modular design
- **Abstraction**
  - Separate essential elements from details
  - Divide into interface and implementation
- **Encapsulation** - exposing interfaces and hiding implementation details
- **Loose coupling** - reducing dependencies for flexibility
- **Stateful vs. stateless**
  - Saving the information
  - Impact on the flexibility and performance

Processing paradigms

Compute-intensive

- Problem requiring more computation
- Distribution of processing

Data-intensive

- Problem operating on a large dataset
- Distribution of data

Domain-driven design

- **Core domain** - the actual problem to be solved
- **Domain logic** - the logic that makes business-critical decisions
- **Building blocks**
  - Entity - defined by identity
  - Value object - immutable object with attributes but no identity
  - Aggregate- binds together with other blocks
  - Service - performs complex use cases
  - Factory - constructs objects
  - Repository - mechanism for encapsulating storage, retrieval, and search behavior
- **Ubiquitous language** - vocabulary shared by everyone involved in a project

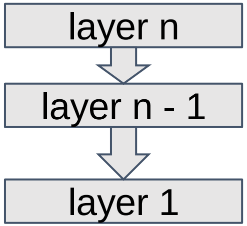
Enterprise Architecture

- **Business architecture** - business strategy, governance, organization, and key business processes
- **Software architecture** - individual application systems to be deployed, their interactions, and their relationships to the core business processes
- **Data architecture** - structure of an organization's logical and physical data assets and data management resources
- **Technology architecture** - logical software and hardware capabilities that are required to support the deployment of business, data, and software services

Software Architecture

Patterns:
Layered

- Structures programs that can be decomposed into groups of subtasks
- Each layer provides services to the next higher layer

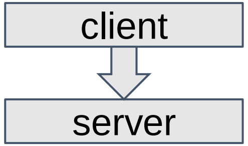


Used to break down a problem into smaller manageable parts, that can be developed separately.

Example: OSI model

Patterns:
client-server

- The server component provides services to multiple client components
- Clients request services from the server
- The server provides relevant services to those clients

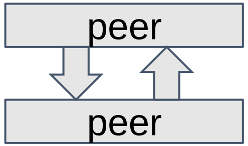


Used to partition tasks or workloads between the providers of a resource or service, and requesters.

Example: HTTP protocol

Patterns:
peer-to-peer

- Peers may function as a client or as a server or as both



- The role may be changed dynamically with time

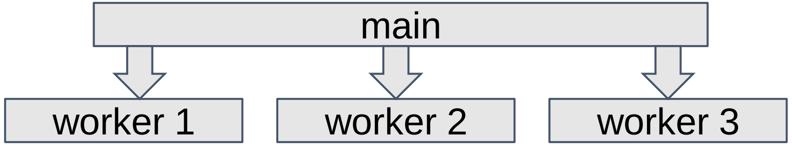
Used to decentralize a system and make it resistant to outages.

Example: cryptocurrencies

Patterns:
main-worker

- The main component distributes the work among identical worker components

- Computes a final result from the results which the workers return



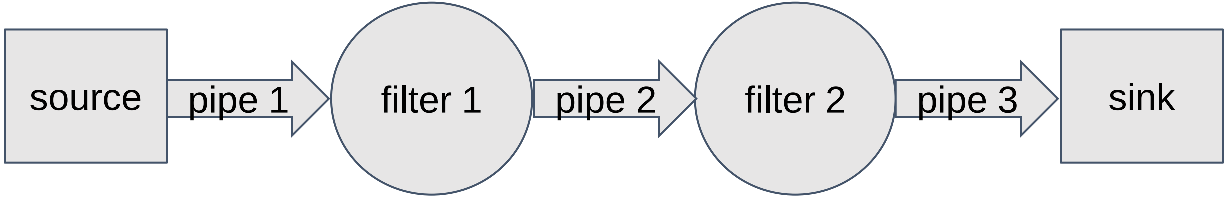
Used to scale up a system.

Example: database replication

Is also named master-slave

Patterns:
Pipes and filters

- Structures systems which produce and process a stream of data
- Each processing step is enclosed within a filter component

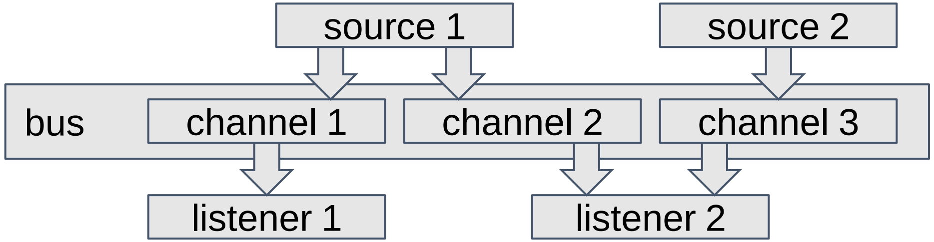


Used to split processing into smaller scalable and reusable parts.

Example: compilers

Patterns:
event bus

- Sources publish messages to channels on an event bus
- Listeners subscribe to particular channels, and are notified of published messages

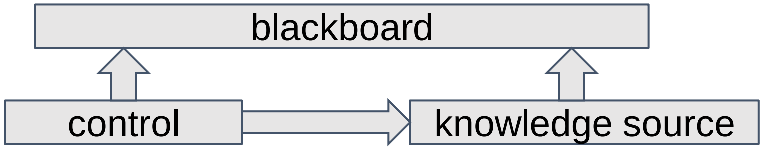


Used to decouple components of a system or integrate multiple systems.

Example: ESB solutions

Patterns: blackboard

Blackboard - a structured global memory containing objects from the solution space
Knowledge source - specialized modules with their own representation
Control component - selects, configures and executes modules

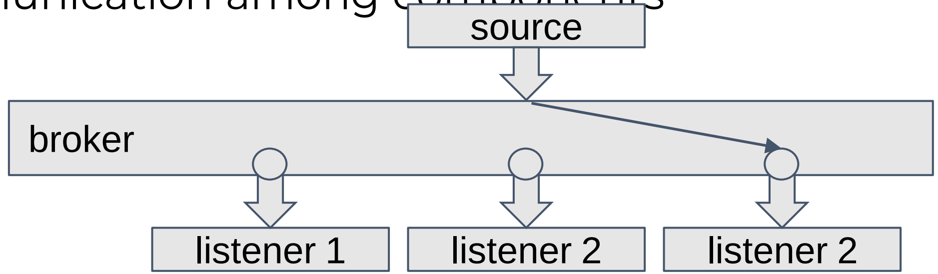


- Components may produce new data objects that are added to the blackboard
- They look for particular kinds of data on the blackboard, and may find these by pattern matching with the existing knowledge source

Used for problems for which no deterministic solution strategies are known.
Example: speech recognition

Patterns:
broker

- Structures distributed systems with decoupled components, that can interact with each other by remote service invocations
- The broker component is responsible for the coordination of communication among components

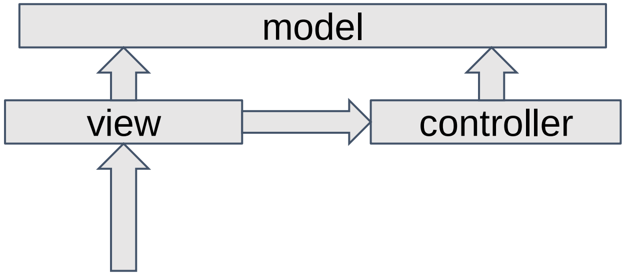


Used to coordinate communication between components.

Example: Kafka

Patterns:
MVC

Model - contains the core functionality and data
View - displays the information to the user
Controller - handles the input from the user



- Separates internal representations of information from the ways information is presented to, and accepted from the user
- Decouples components and allows efficient code reuse

Used to decompose interactive systems.

Example: web applications

Evolution of Software Architecture

Monolith

Single indivisible unit, usually database, user interface, and server-side application

Microservices

Multiple independent components connected with APIs

Serverless

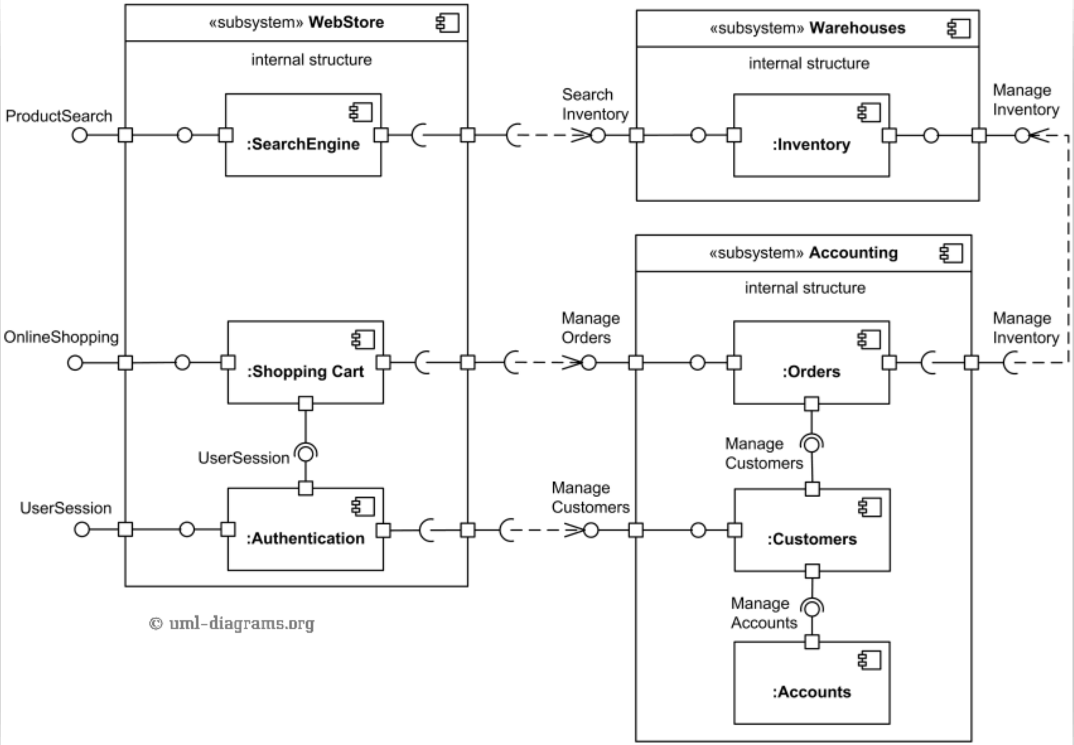
Apps and services in a cloud, without the need for infrastructure management

Unified Modeling Language (UML)

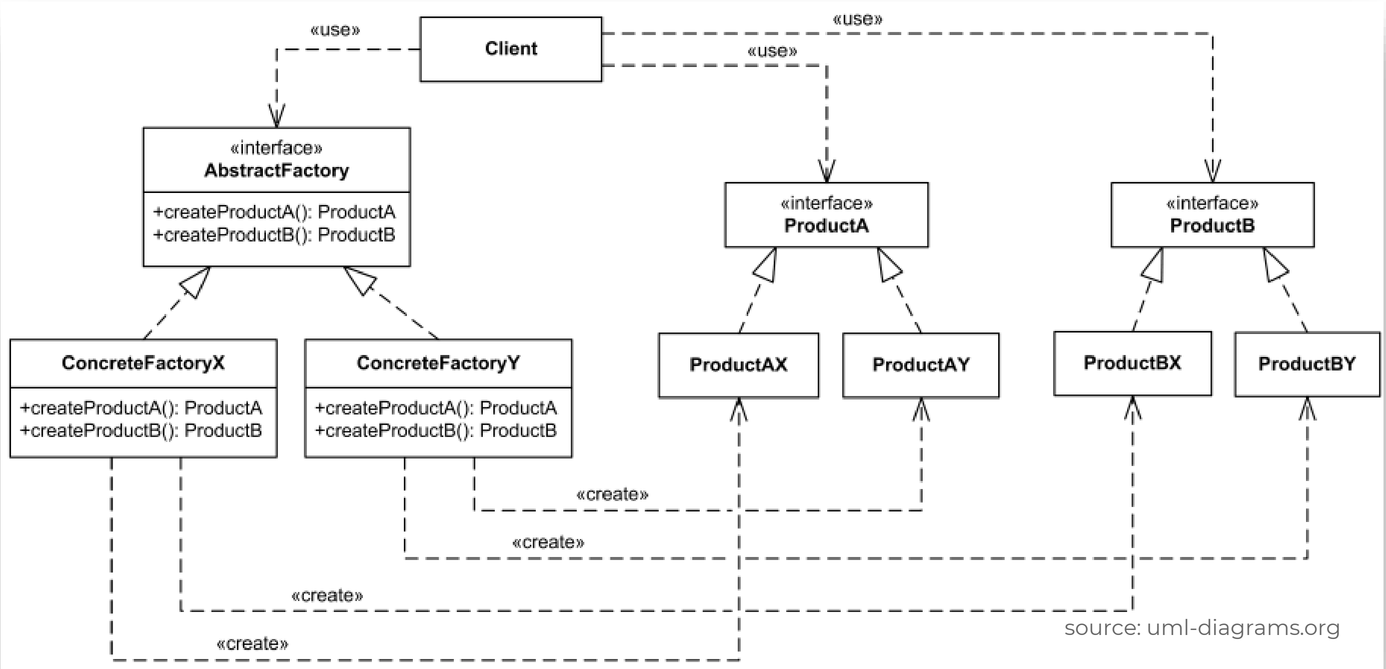
Structural diagrams - describe static structures

- **Component** - displays the structural relationship of components of a software system
- **Class** - shows the classes in a system, attributes, and operations of each class and the relationship between each class
- **Object** - same as the class diagram, but for concrete instances

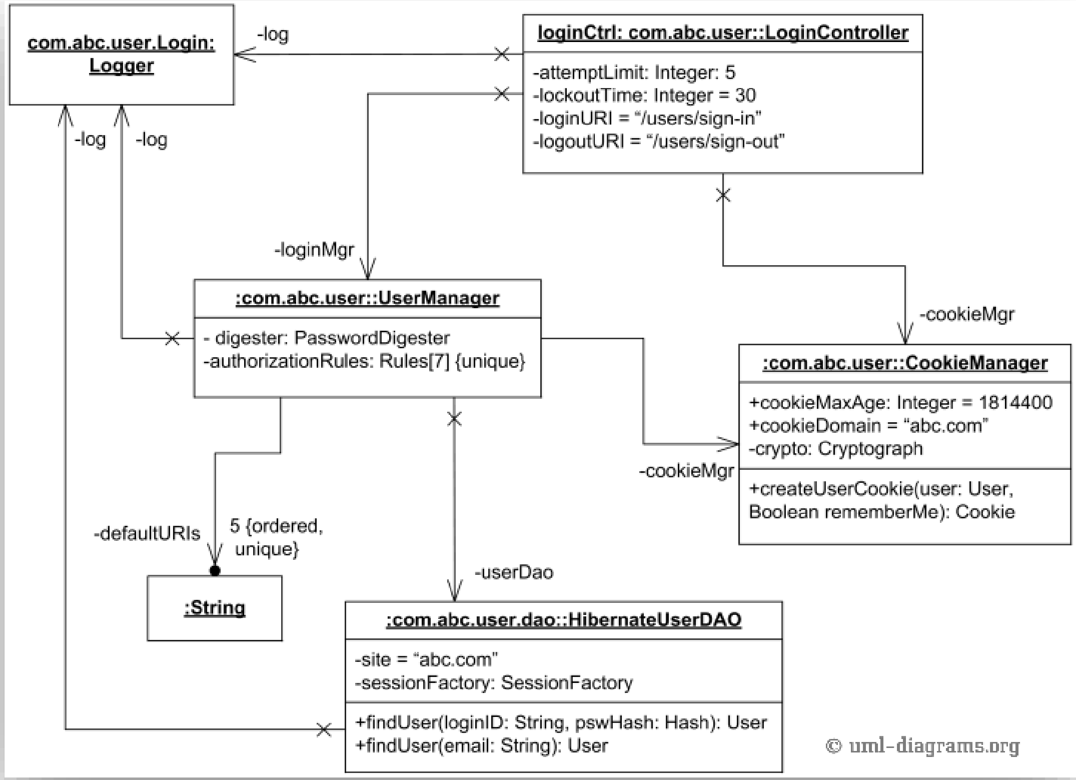
UML - Component Diagram example



UML - Class Diagram example



UML - Object Diagram example

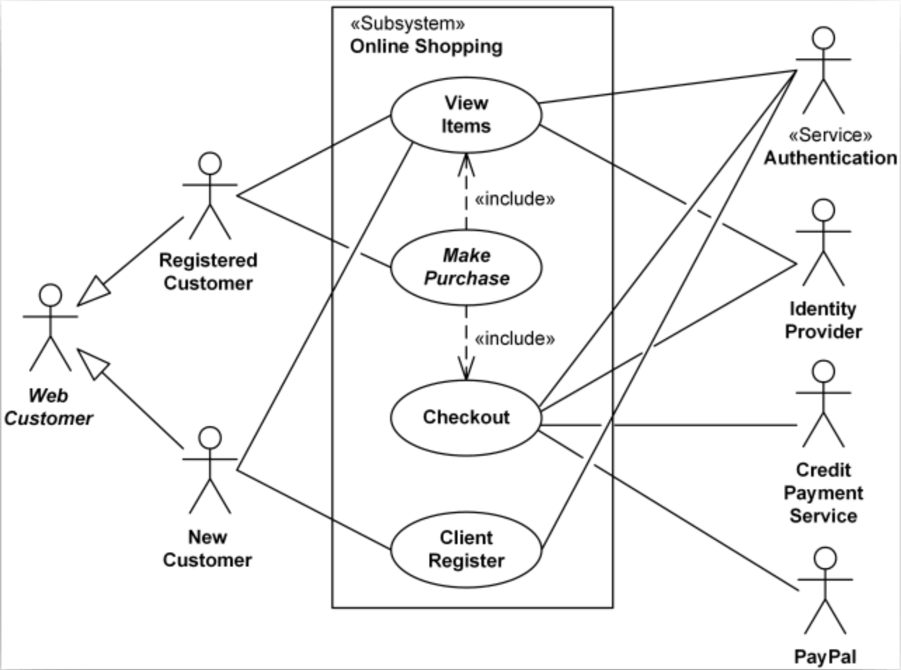


Unified Modeling Language (UML)

Behavioral diagrams - describe dynamic aspects of a system: objects' collaboration and changes

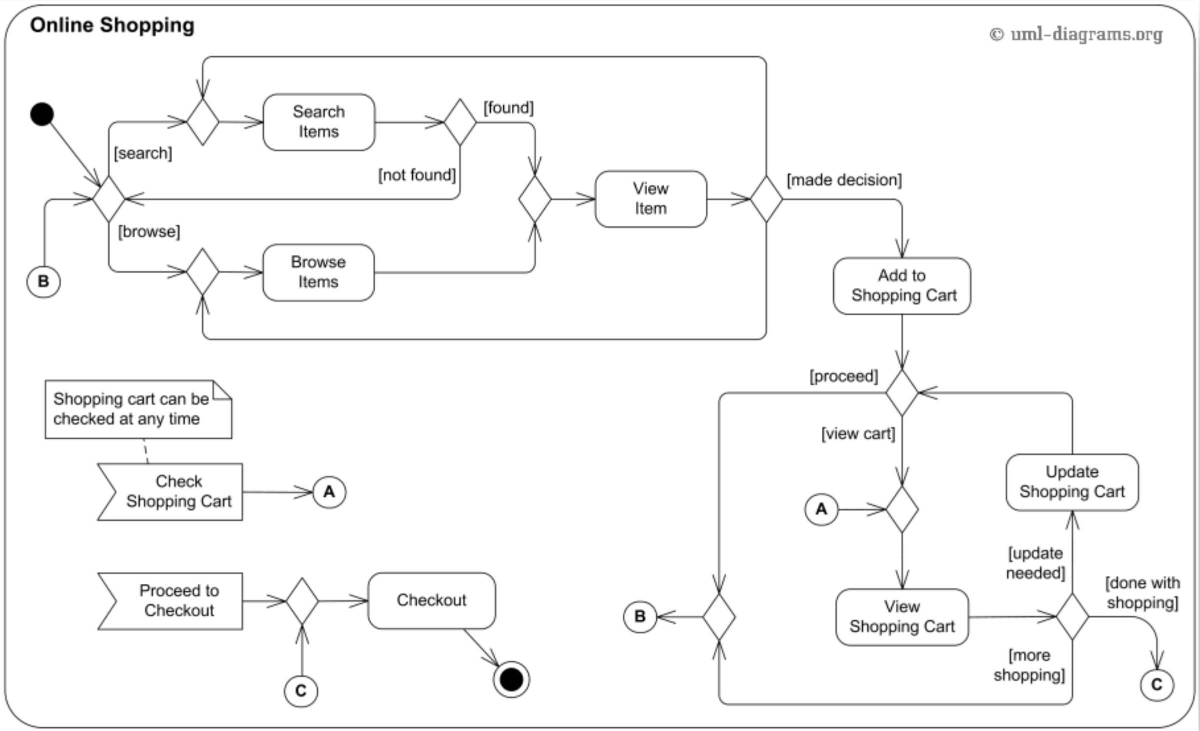
- **Use case** - gives a graphic overview of the actors involved in a system, different functions needed by those actors and how these different functions interact
- **Activity** - represents workflows in a graphical way
- **State** - describes the behavior of objects that act differently according to the state they are in at the moment

UML - use case diagram example

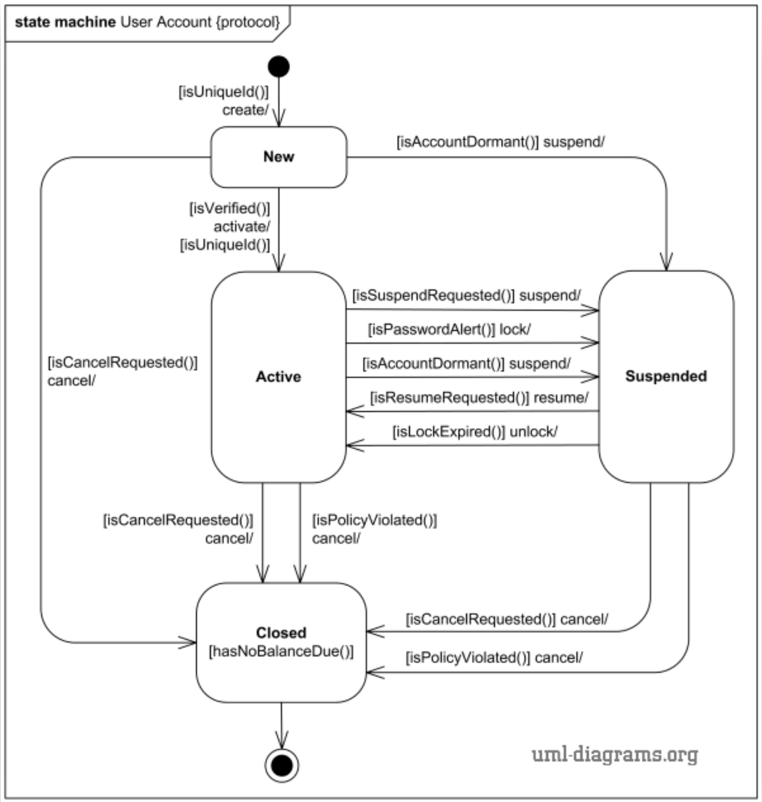


source: uml-diagrams.org

UML - Activity Diagram example



UML - State Diagram example

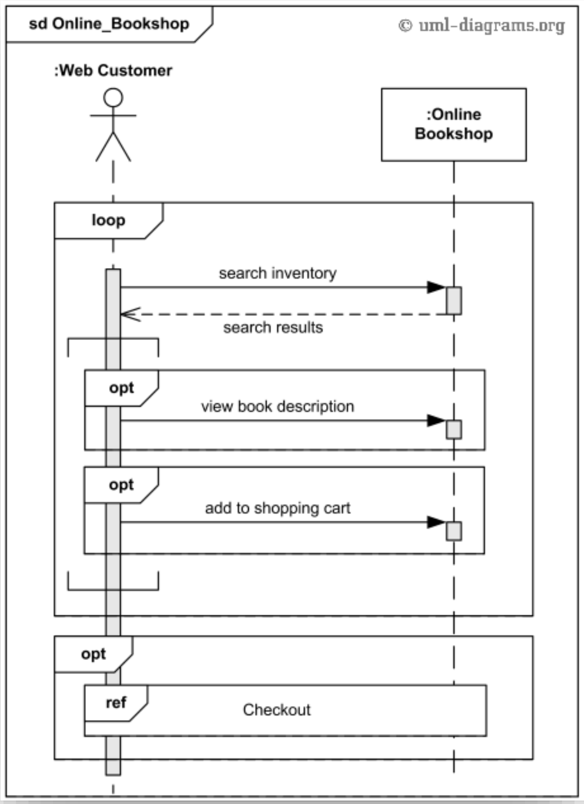


Unified Modeling Language (UML)

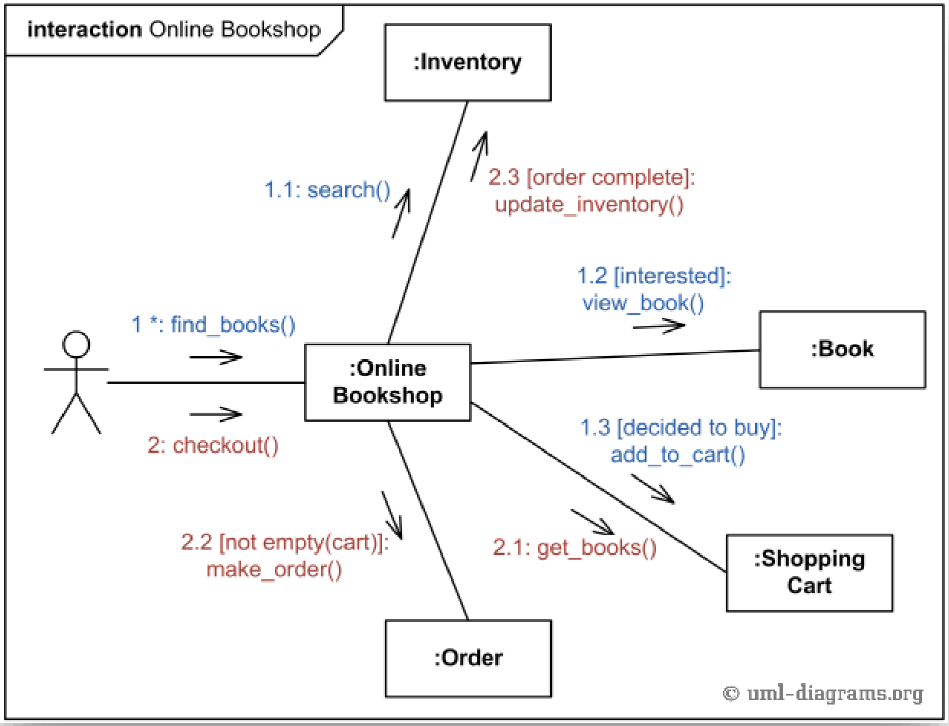
Interaction diagrams - describe interactive behavior of a system

- **Sequence** - shows how objects interact with each other and the order those interactions
- **Communication** - similar to sequence diagrams, but the focus is on messages passed between objects

UML - Sequence Diagram example



UML - Communication Diagram example



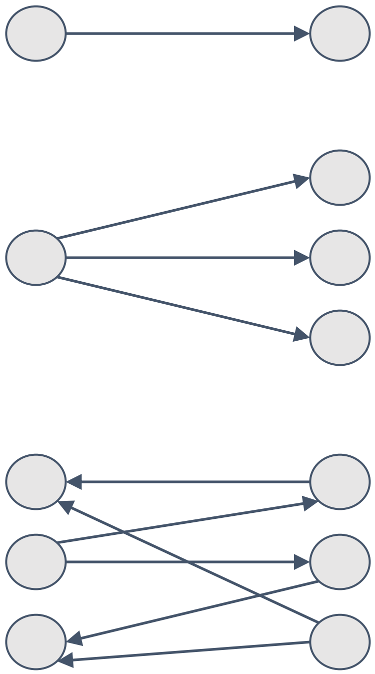
Data Architecture

Data
Relations

- **One-to-one** - connects one entity to another

- **One-to-many** - connects one entity to one or more other entities

- **Many-to-many** - connects many entities to other many entities



Evolution of Data Representation Models

Relational

- Static model
- Structured
- SQL
- Can be scaled vertically

NoSQL (Not only SQL)

- Dynamic model
- Unstructured
- Specialized languages
- Can be scaled vertically and horizontally

CAP Thereom

- **Consistency** - every read receives the most recent write or an error
- **Availability** - every request receives a (non-error) response, without the guarantee that it contains the most recent write
- **Partition tolerance** - ability to continue processing data even if subsystems can not communicate

NoSQL databases prefers availability over consistency

Relational:
The relational model

- Organizes logically related data into **relations** (tables)
- Each **attribute** (column) keeps up with a particular kind of data
- Each **tuple** (row) holds all of the data about a particular entity

	attribute 1	attribute 2		attribute N
tuple 1				
tuple 2				
tuple N				

Relational:
Keys - identify tuples and the relationships

- **Primary key** - attribute (or a set of attributes) whose values uniquely identify a tuple in the relation
- **Foreign key** - attribute (or a set of attributes) whose values correspond to the values of the primary key in another relation

Relational:
SQL - querying relational data

- **Selection** - retrieves particular tuples from a relation that satisfy some condition
- **Projection** - retrieves particular attributes from a relation
- **Joining** - combines information from multiple tuples, basing on shared attributes

Relational:
ACID guarantee

- **Atomicity** - each transaction is a single unit, which either succeeds completely or fails completely

- **Consistency** - transaction can only bring the database from one consistent state to another

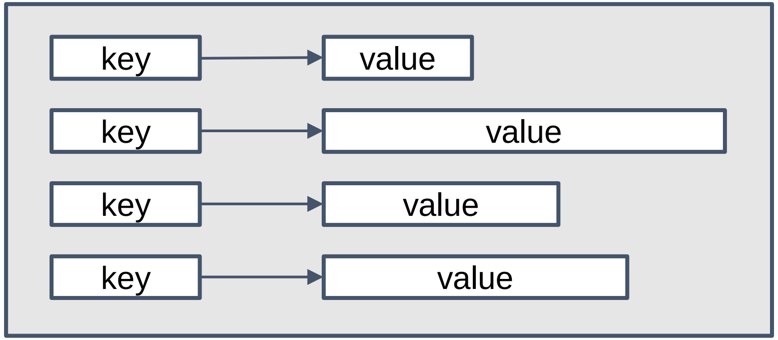
- **Isolation** - concurrent execution of transactions leaves the database in the same state as executed sequentially

- **Durability** - once a transaction has been committed, it will remain committed even in the case of a system failure

Brainstorm! What are the consequences of violation of each rule? What are the dangers?

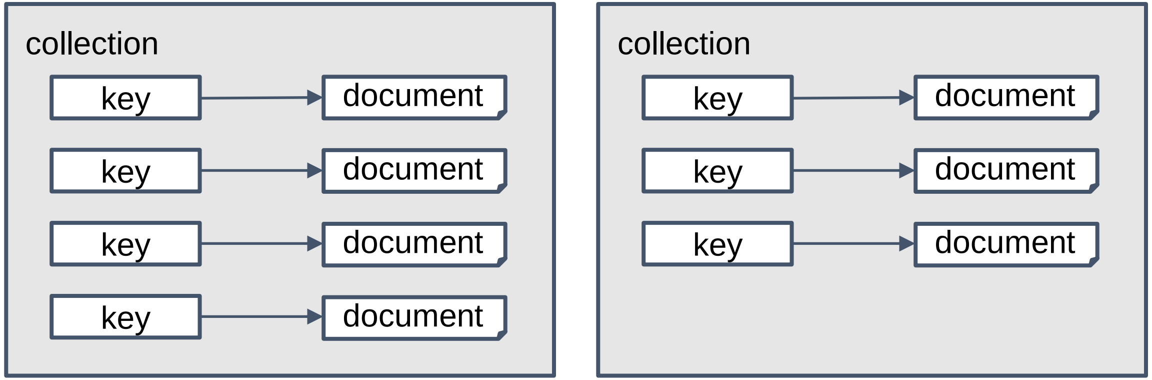
NoSQL: Key-value store

Collection of objects stored and retrieved using a key that uniquely identifies the object. The structure of the objects is not defined.



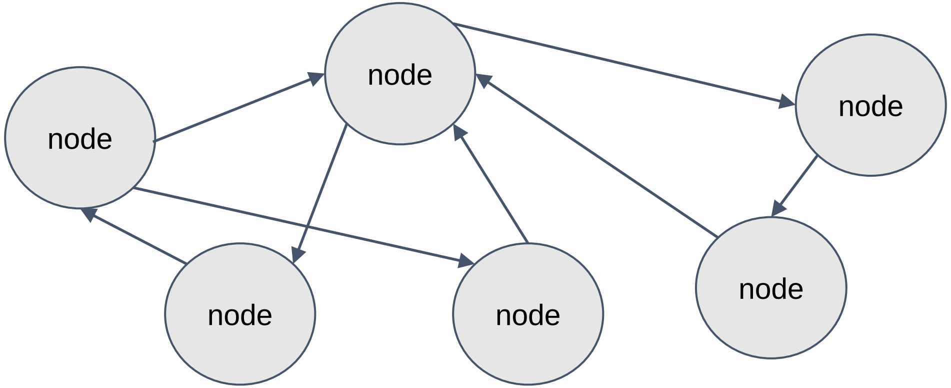
NoSQL:
The document model

- Extends the key-value store
- **Document** - object that encapsulates data in a specific format (JSON, YAML, XML, BSON, ...)
- **Collection** - groups documents of same type (similarly to tables)



NoSQL:
The graph model

- Relations are well represented as a graph consisting of elements connected by a finite number of relations
- **Nodes** represent entities (values, tuples, documents)
- **Edges** represent the relations between nodes



The structure is very flexible, and can reflect complicated relationships such as road network.

Types of Data Architecture

- **Warehouse** - centralized repository of structured data that contains information from many sources
- **Mart** - subset of a data warehouse oriented to a specific domain
- **Lake** - centralized repository that allows to store all structured and unstructured data at any scale
- **Mesh** - distributed, domain-specific data, with each domain handling their own data pipelines

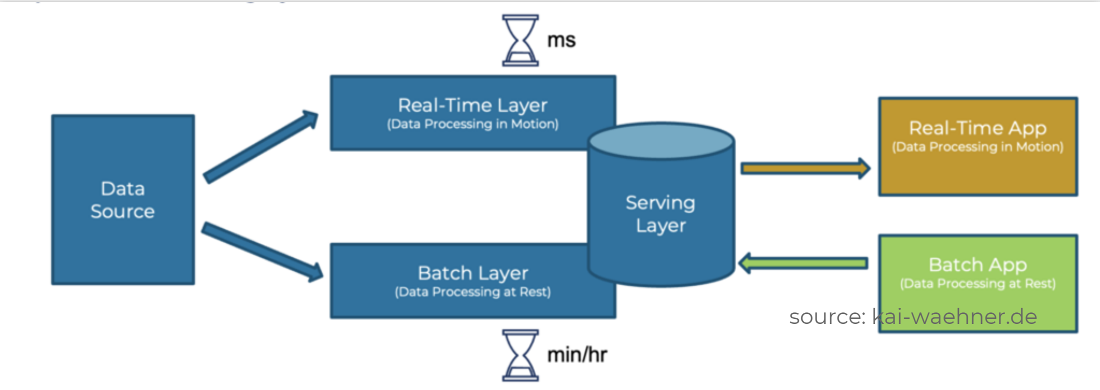
Types of
Data Architecture

	centralized	domain-specific
structured	warehouse	mart
unstructured	lake	mesh

- Centralized - when need to keep all the information in one place
- Domain-specific - helps to focus on the most critical areas of interest or problem areas, and keeps it customized
- Structured vs. unstructured - depends on the format and purpose

Lambda
Architecture

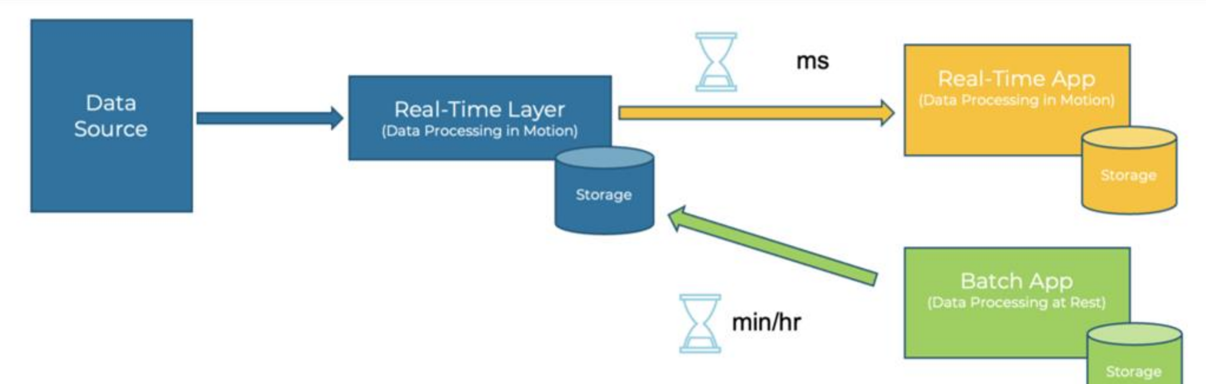
- **Batch layer** - precomputes results using a distributed processing system that can handle very large quantities of data
- **Stream layer** - processes data streams in real time and without the requirements of fix-ups or completeness
- **Serving layer** - responds to ad-hoc queries by returning precomputed views or building views from the processed data



source: kai-waehner.de

Kappa Architecture

Avoids maintaining two different code bases for the batch and stream layers. The key idea is to handle real-time data processing, and continuous data reprocessing using a single stream processing engine meeting the standard quality of service.



source: kai-waehner.de

Making Architectural Decisions

Build vs. Buy:
Aspects to take into account

- **Scope** - is the functionality the core part of business?
- **Time** - how quickly the solution must be deployed?
- **Cost** - how much would it cost to develop own solution?
- **Control** - is the solution configurable?
- **Security** - can we use an open source solution?
- **Maintenance** - what support is provided?

What technology to use:
Factors

- Skills and experience of the people involved
- Legacy system dependencies
- The timescale for delivery
- Tolerance of different kinds of risk
- Regulatory constraints

Comparing Technologies

- **Features** - provided functionality and its usability
- **Performance** - how fast it works and what resources it needs
- **Cost** - how expensive it is
- **Viability** - how popular the technology is and how big its community is
- **Support** - how quickly are issues resolved and how often are new versions delivered

Daily Assignment

Daily
Assignment

Draw a UML state diagram of a mobile phone.

At minimum, it must contain the following states:

- Off
- Idle
- Active
- Ringing
- Call
- texting