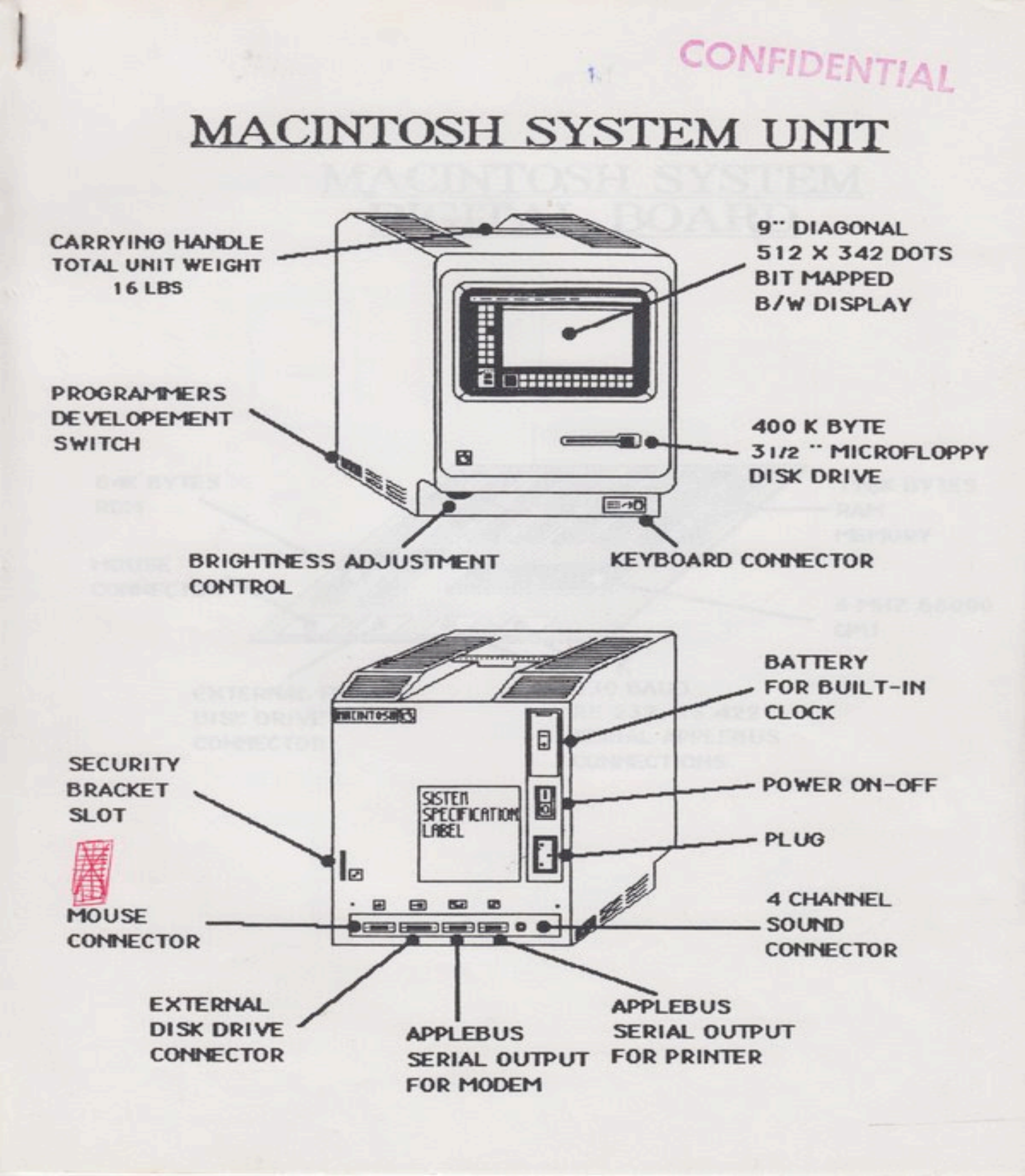
CONFIDENTIAL

MACINTOSH SYSTEM UNIT



CPEN 321

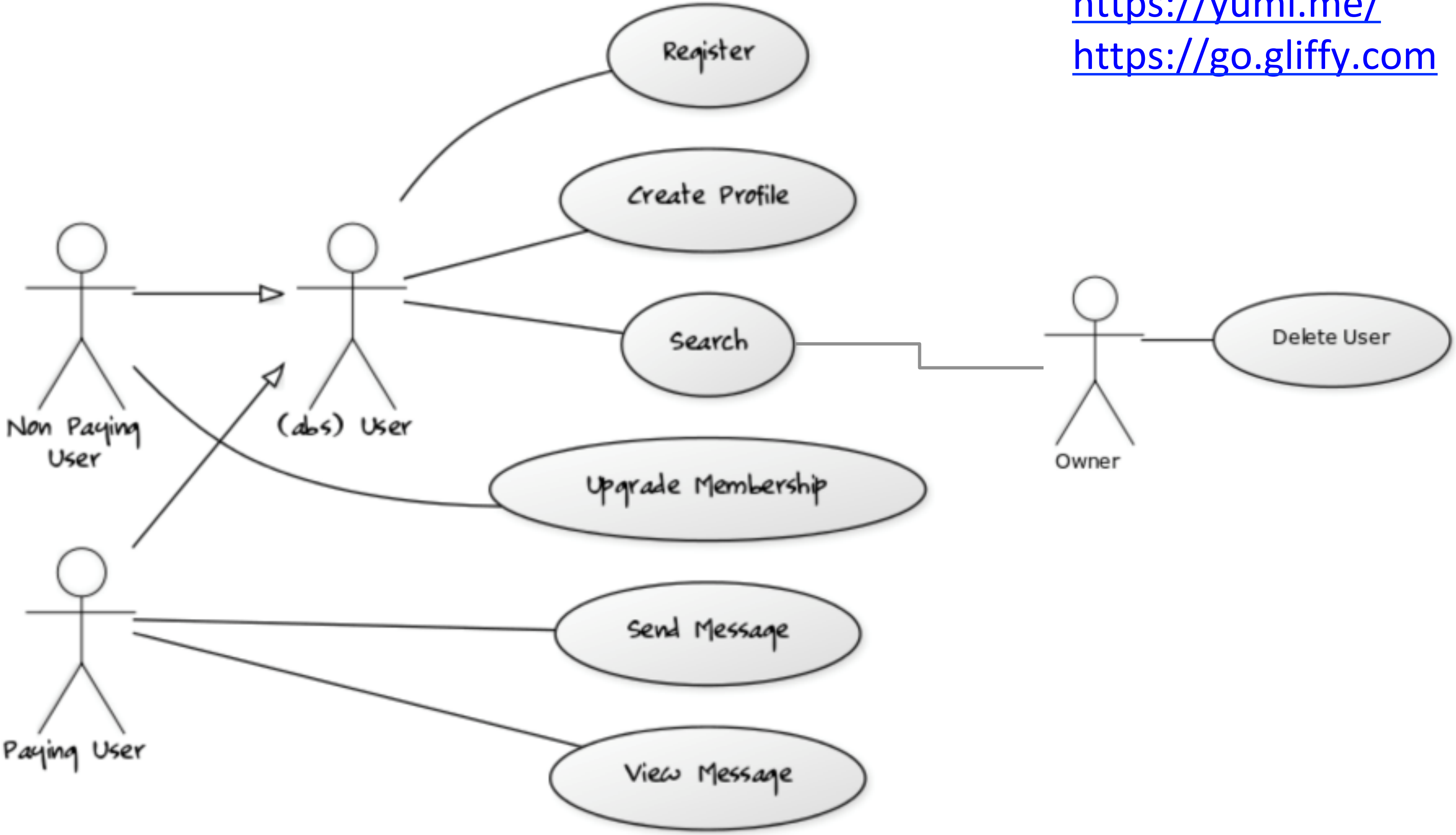
Architecture and Design (REST, Microservices)

Recap: Example

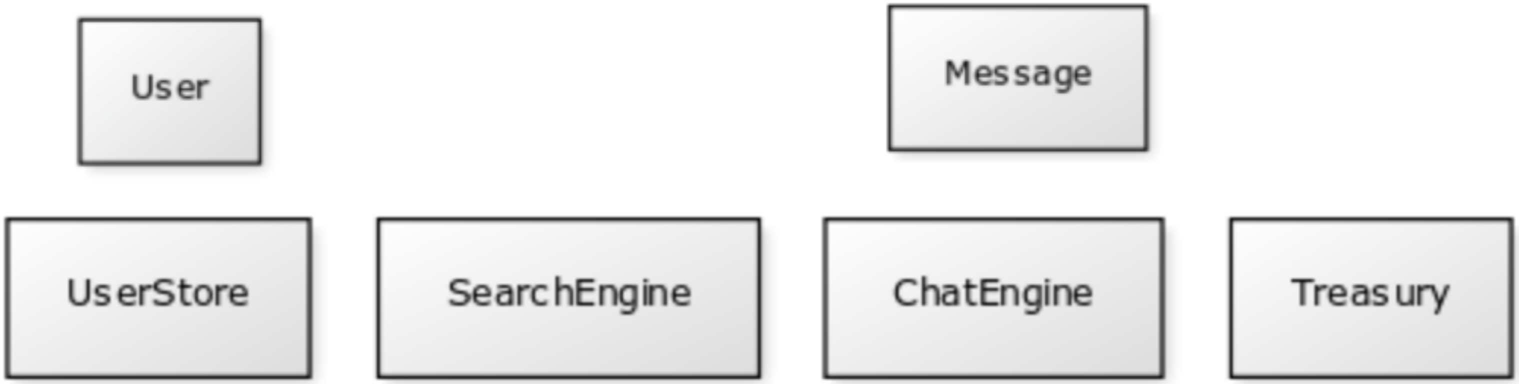
- You build an online dating system.
- The client plans to attract customers by providing free browsing and matching functionality, but charging for allowing users to contact other users.
- For example, a user should be able to register, create a profile, and search for “soul-mates”, all without paying.
- Then, if they want to send a message to another user, or receive a message from another user, they need to upgrade their membership by making a payment.

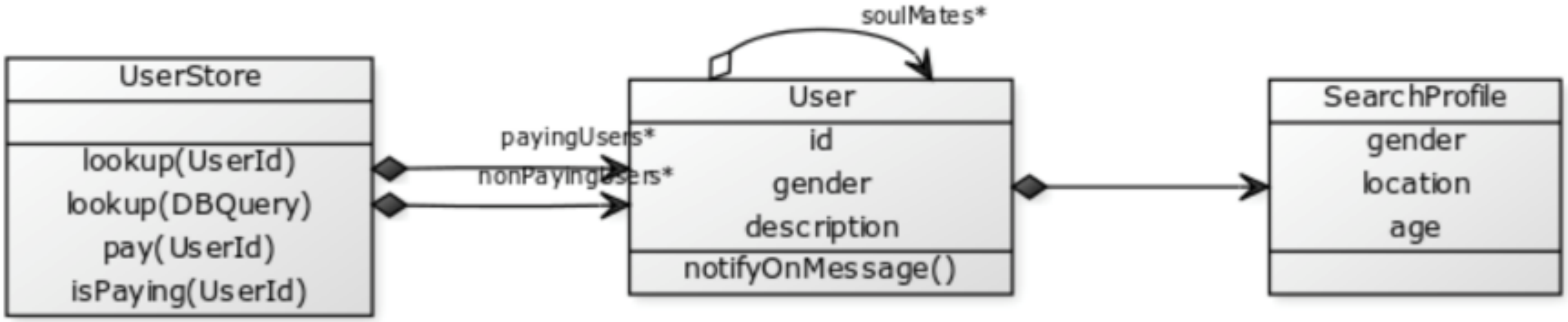
Use Case Diagram

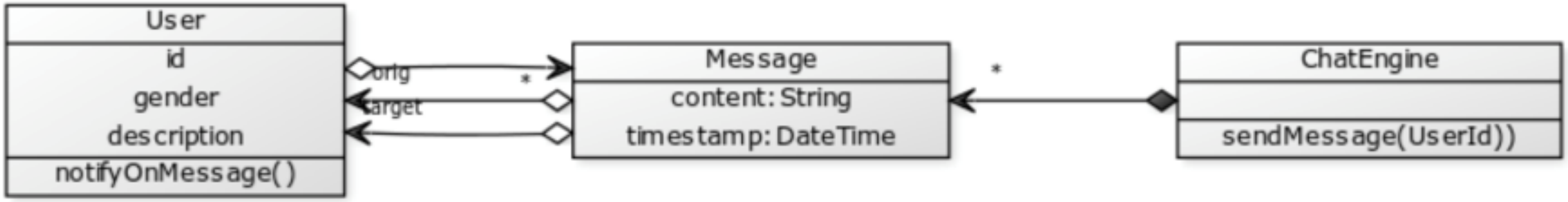
https://yuml.me/
https://go.gliffy.com

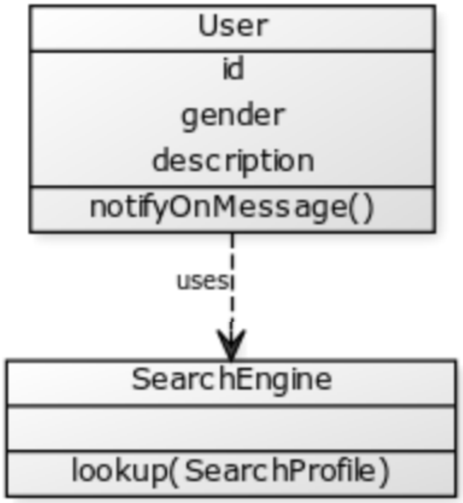


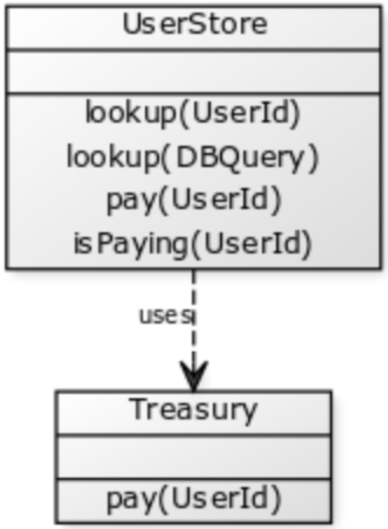
Identify the main modules needed to implement your system. Model their relationships using a class diagram



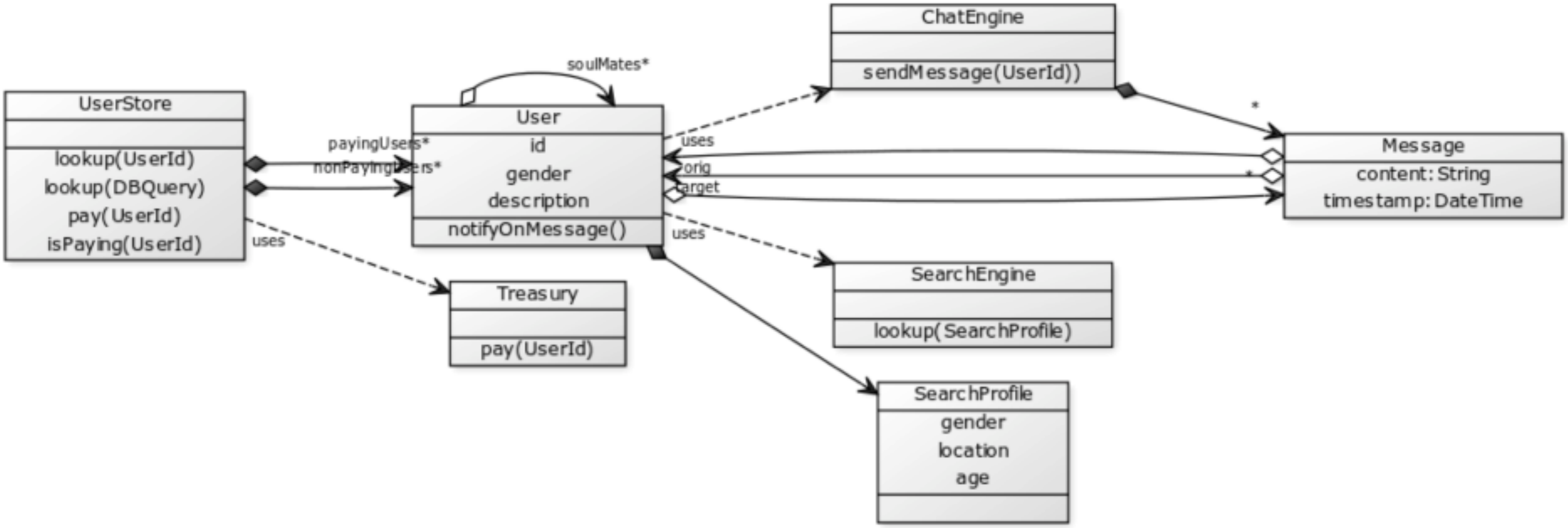

soulMates*
UserStore
lookup(UserId)
lookup(DBQuery)
pay(UserId)
isPaying(UserId)
payingUsers*
nonPayingUsers*
User
id
gender
description
notifyOnMessage()
SearchProfile
gender
location
age


User
id
gender
description
notifyOnMessage()
orig
target
*
Message
content: String
timestamp: DateTime
*
ChatEngine
sendMessage(UserId))


User
id
gender
description
notifyOnMessage()
uses
SearchEngine
lookup(SearchProfile)


UserStore
lookup(UserId)
lookup(DBQuery)
pay(UserId)
isPaying(UserId)
uses
Treasury
pay(UserId)

Putting it all together…



UserStore

lookup(UserId)
lookup(DBQuery)
pay(UserId)
isPaying(UserId)

User

id
gender
description

notifyOnMessage()

Treasury

pay(UserId)

ChatEngine

sendMessage(UserId))

Message

content: String
timestamp: DateTime

SearchEngine

lookup(SearchProfile)

SearchProfile

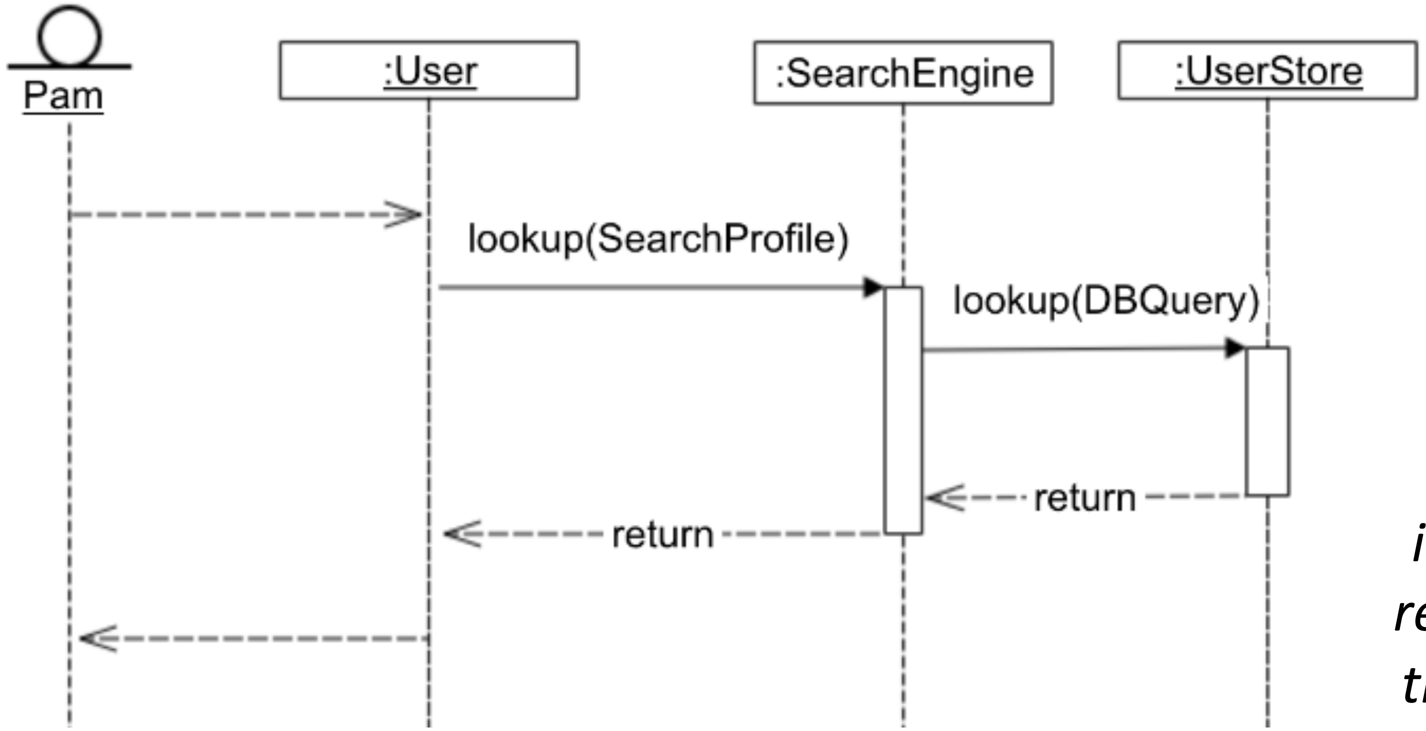
gender
location
age

Sketch a sequence diagram showing a sequence of steps needed to implement "Send a message" use case of your system

Notes

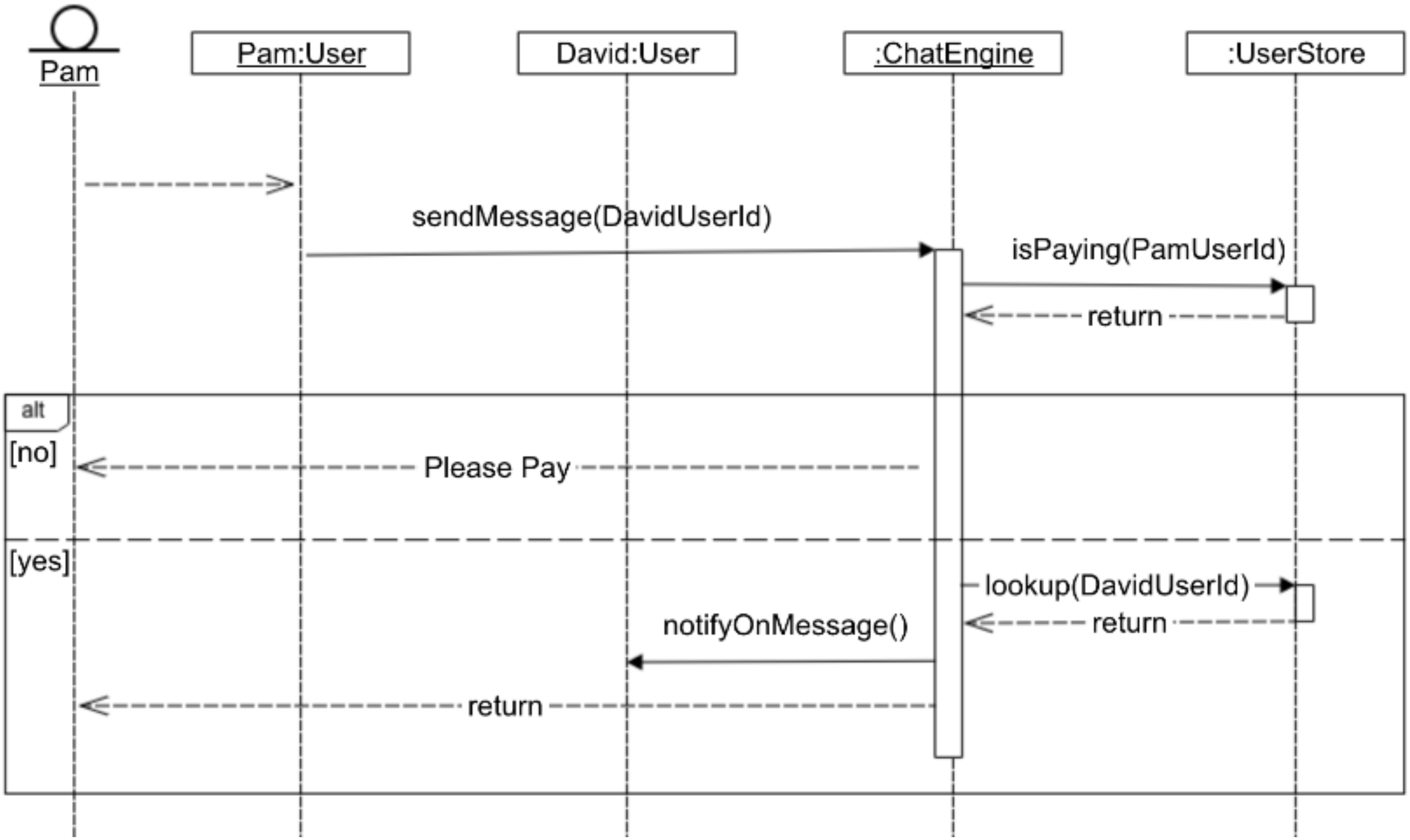
- Your diagram should only include classes and methods you defined in the previous question

(a) Search for Soul-Mates

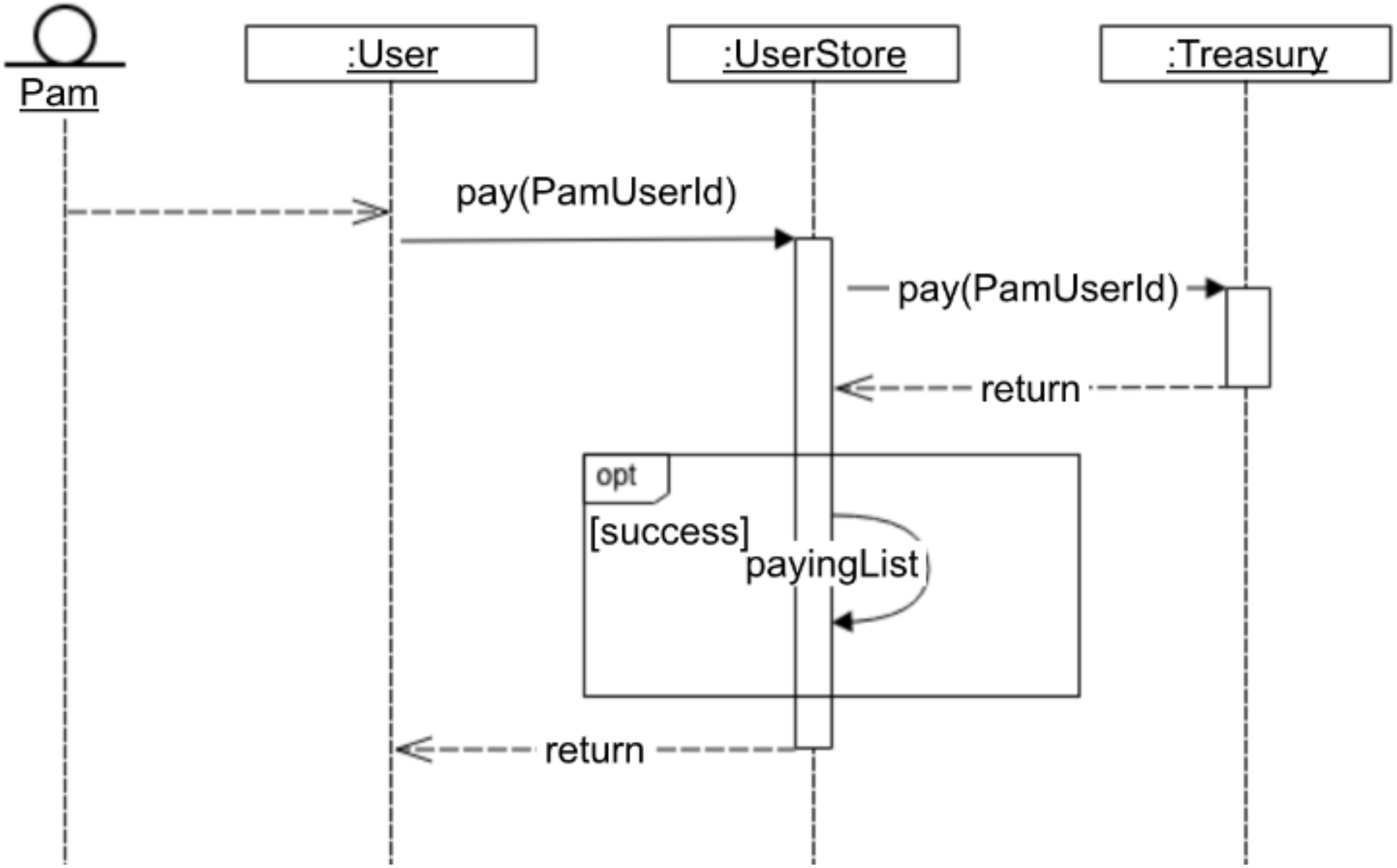


Can optionally iterate over the result and add to the SoulMate list

(b) Send a Message

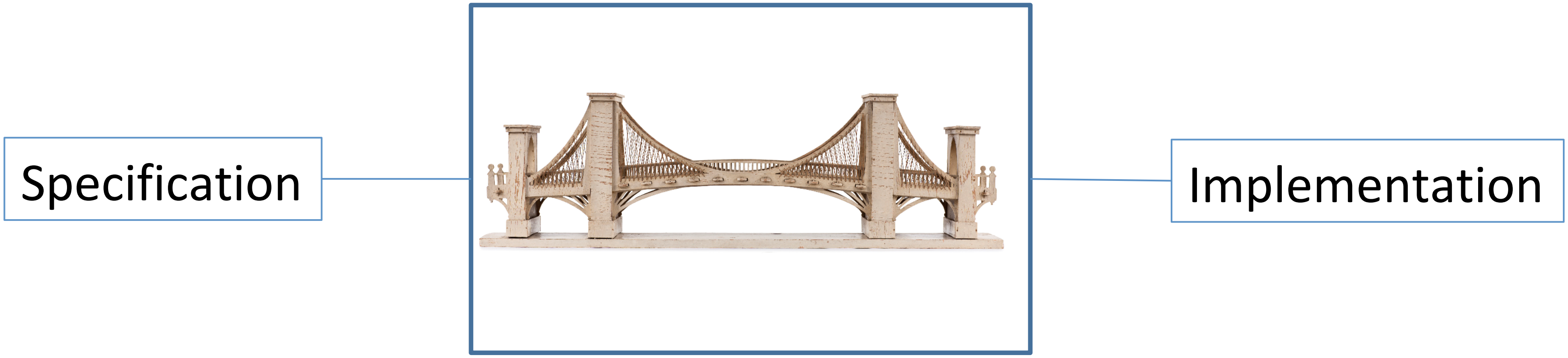


(c) Register and Pay



Why is a user hierarchy a bad idea here?

Software Architecture and Design



Specification

Implementation

Software Architecture and Design

Software Architecture and Design

Architecture: big picture

- Architectural patterns
  - High-level modules
  - Their interaction principles
- Data storage paradigms
- Recovery systems in place
- Used frameworks, tools, and languages
- etc.

Design: detailed view

- Functions of each module (internal and external)
- Implementation of interactions
- Design patterns
- Programming idioms
- Algorithms
- etc.

Terminology

- “Architecture” and “High-level design” are terms that often used interchangeably
- The term “low-level design” is often used to describe the detailed design of individual modules

Agenda

- Architectural principles
- Architectural patterns
  - Layered architecture
  - Client-server architecture
  - Pipe-and-filter architecture
  - Model-View-Controller (MVC)
  - Model-View-ViewModel
  - Message bus
  - REST, Microservices

Architecture: What to Include?

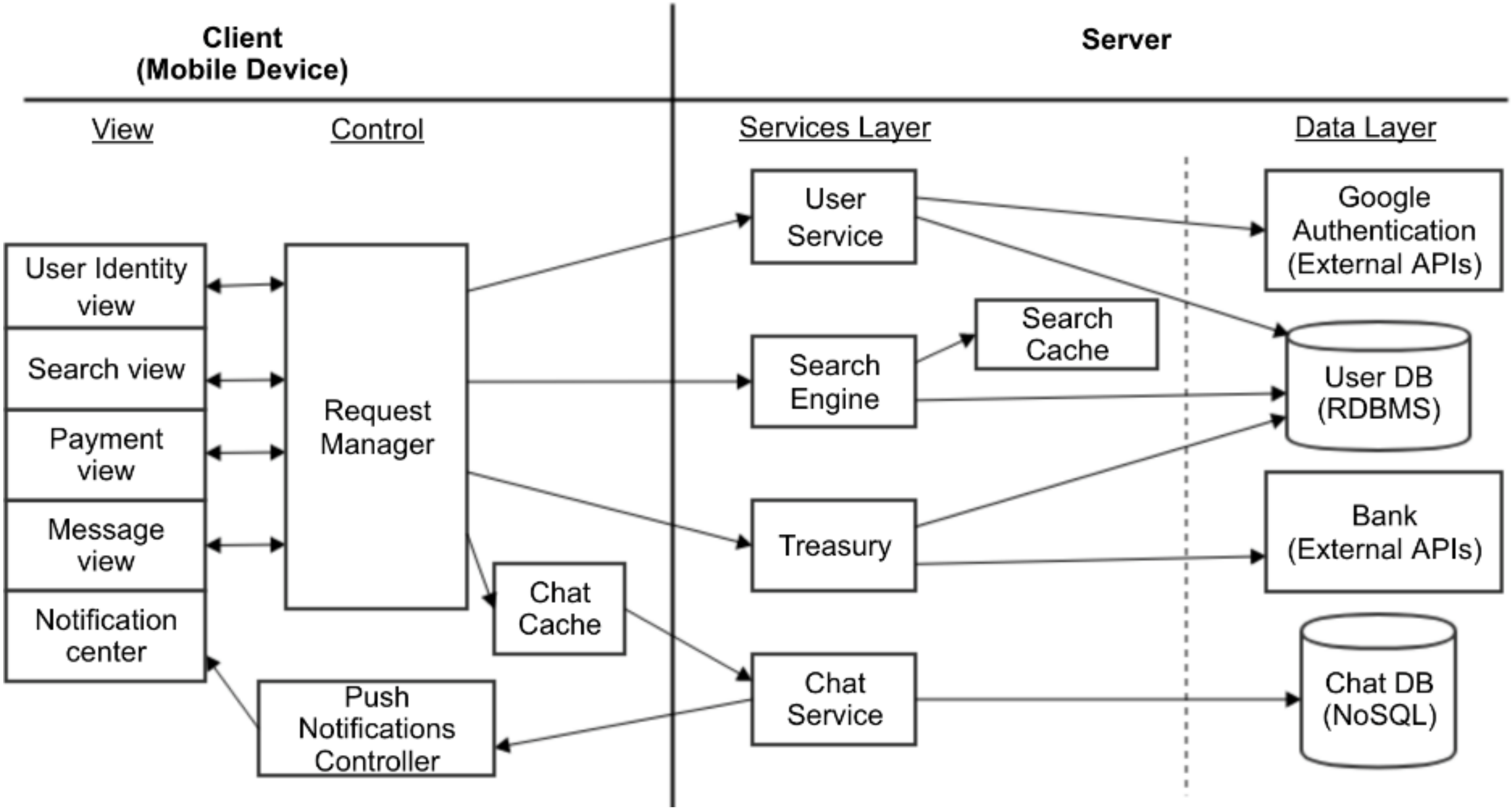
Architecture: What to Include?

- Major systems and sub-systems
- Interfaces and communication protocols
- Database and data structures (relational, graph, etc.)
- Deployment environment and computers
- Security mechanisms
- Fault tolerance mechanisms
- …

Expectations for the Next Milestone

- Main components, roles, and responsibilities
- Which architectural patterns are used and why (for both client and server)
- Interaction protocols (which and why)
- Data store -> which data, type, why and what were the alternatives
- Main frameworks and tools used, why, what were the alternatives
- How your non-functional requirements are realized
- Main algorithms

Example: High-Level Architecture of the Dating App



Choosing Subsystems

- Single responsibility principle:
  - Every subsystem should have a single responsibility
  - The responsibility should be entirely encapsulated by the subsystem
  - All subsystem services should be aligned with that responsibility

- Why?
  - easier to understand
  - easier to test
  - easier to maintain
  - easier to replace

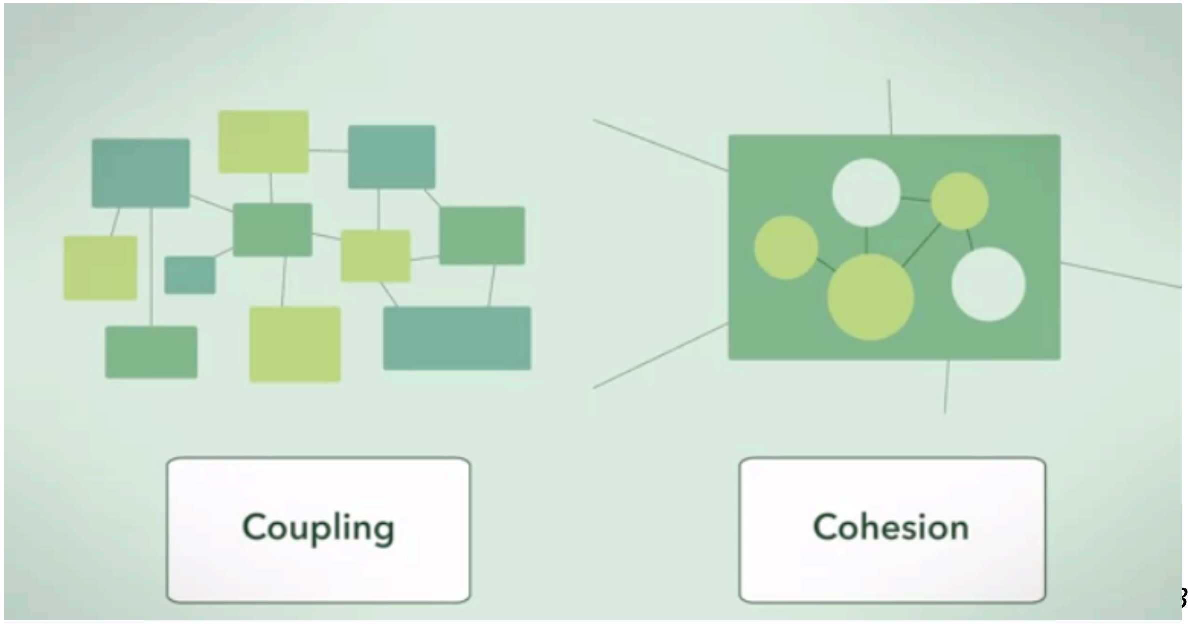
Choosing Subsystems

- Loose coupling principle:
  - Keep connections between parts of the system minimized
    - Avoid n^2 interactions

- Why?
  - easier to understand
  - easier to test
  - easier to maintain
  - easier to replace

Low Coupling / High Cohesion

- **Cohesion**: the degree to which the elements of a module belong together (related code should be close to each other)
- **Coupling**: the degree to which the different modules depend on each other (modules should be independent as far as possible)



Choosing Subsystems

- High fan-in low fan-out principle:
  - Have a system used by many others
  - Do not use many other systems

- Why?
  - complexity management
  - …

Choosing Subsystems

- Minimal complexity principle:
  - Favor simple over “clever” solution

- Why?
  - easier to understand
  - easier to test
  - easier to maintain
  - easier to replace

Core Architectural Principles (Test of Time)

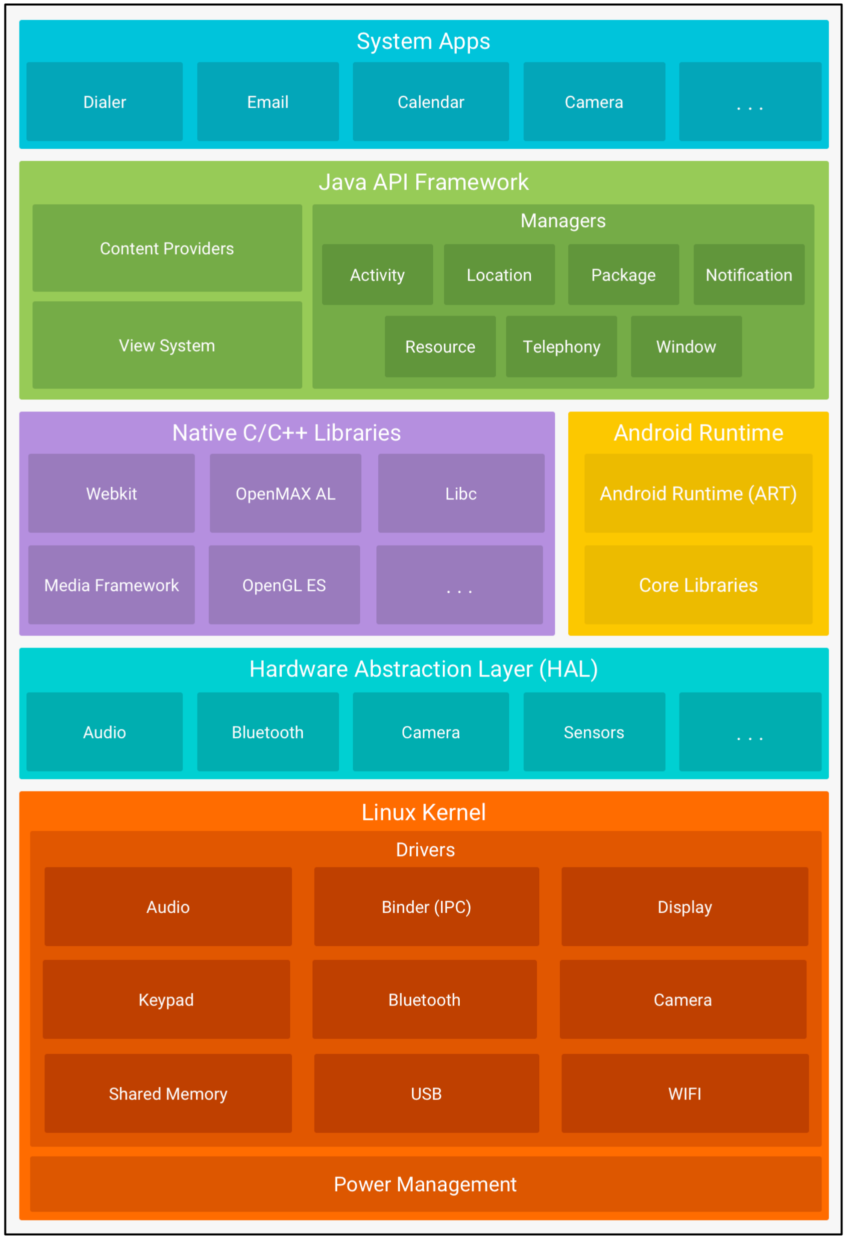
- **Single Responsibility Principle:** Each module should be responsible for only a specific feature or functionality, or aggregation of cohesive functionalities.
- **Separation of Concerns:** Minimize interaction points to achieve high cohesion and low coupling.
- **Principle of Least Knowledge:** A module should not know about internal details of other modules.
- **Don’t Repeat Yourself (DRY)**: Do not duplicate functionality.
- **KISS**: Make it simple. Only focus on what is needed.

Architectural Pattern

- An **architectural pattern** is a stylized description of good design practice, which has been tried and tested in different environments.
- Based on experience of successful implementations
- Include information about when they are, when to use them, and when they are not useful.

Popular Architectural Patterns

- Layered architecture



System Apps

Dialer | Email | Calendar | Camera | . . .

Java API Framework

Content Providers

View System

Managers

Activity | Location | Package | Notification

Resource | Telephony | Window

Native C/C++ Libraries

Webkit | OpenMAX AL | Libc

Media Framework | OpenGL ES | . . .

Android Runtime

Android Runtime (ART)

Core Libraries

Hardware Abstraction Layer (HAL)

Audio | Bluetooth | Camera | Sensors | . . .

Linux Kernel

Drivers

Audio | Binder (IPC) | Display

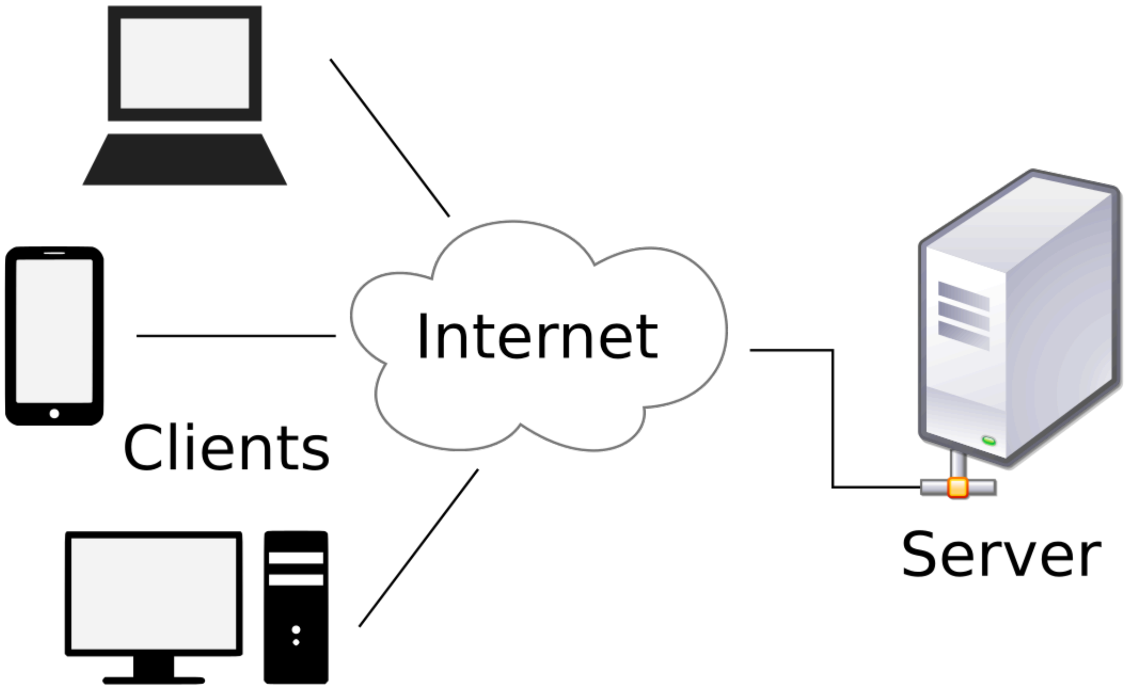
Keypad | Bluetooth | Camera

Shared Memory | USB | WIFI

Power Management

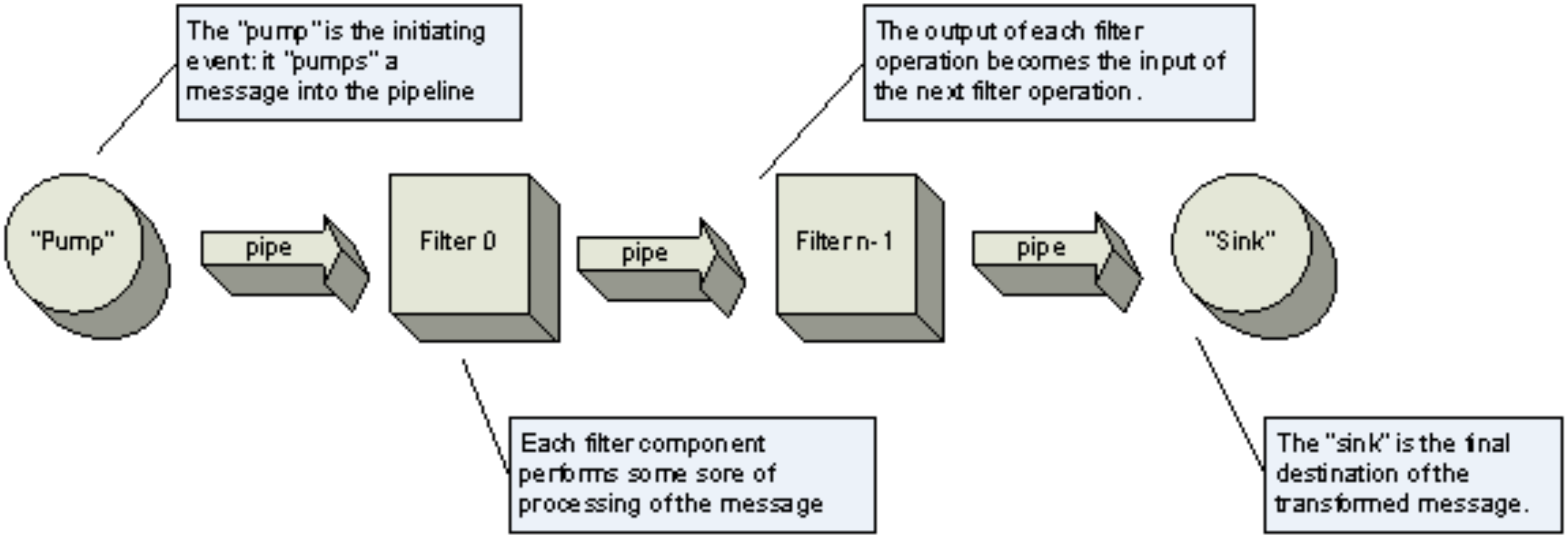
Popular Architectural Patterns

- Layered architecture
- Client-server architecture



Popular Architectural Patterns

- Layered architecture
- Client-server architecture
- Pipe-and-filter architecture



The "pump" is the initiating event: it "pumps" a message into the pipeline

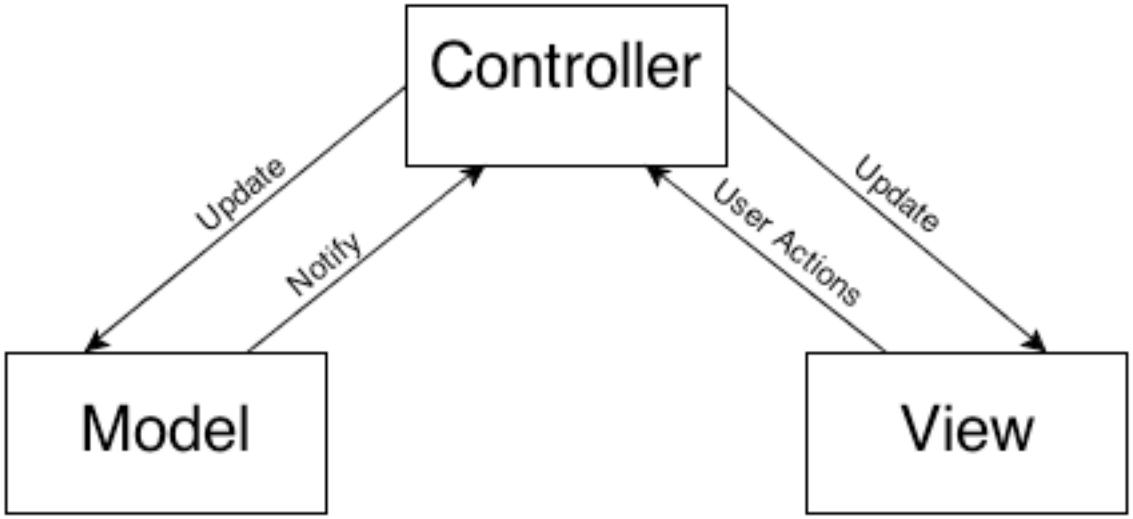
The output of each filter operation becomes the input of the next filter operation.

Each filter component performs some sore of processing of the message

The "sink" is the final destination of the transformed message.

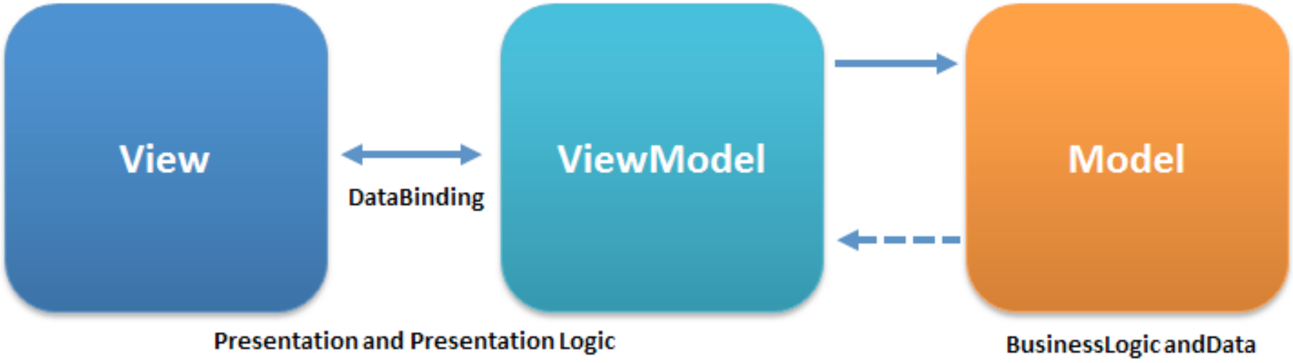
Popular Architectural Patterns

- Layered architecture
- Client-server architecture
- Pipe-and-filter architecture
- Model-View-Controller (MVC)



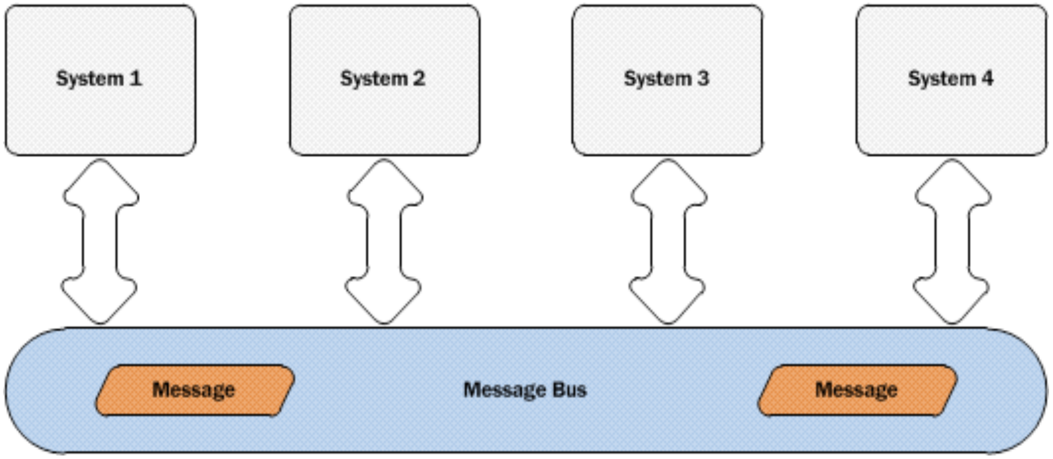
Model-View-ViewModel

- A variation of the MVC Pattern
- Developed by Microsoft architects to simplify event-driven programming of user interfaces
- Facilitates the separation of the view layer
  - Usability experts write GUI code (View) while application developers write and maintain ViewModel
- The separation of roles allows interactive designers to focus on UX needs rather than the programming of business logic.



Popular Architectural Patterns

- Layered architecture
- Client-server architecture
- Pipe-and-filter architecture
- Model-View-Controller (MVC)
- Message Bus: a software system that sends and receives messages using one or more standard communication channels
  - Applications can interact without needing to know specific details about each other.



Agenda

- Architectural principles
- Architectural patterns
  - Layered architecture
  - Client-server architecture
  - Pipe-and-filter architecture
  - Model-View-Controller (MVC)
  - Model-View-ViewModel
  - Message bus
  - **REST, Microservices**

REST Philosophy

- REST is a “design guideline” for communication in networked systems
  - not a protocol, not a specification)

- Principles:
  - Resource Identification using a URI (Uniform Resource Identifier)
  - Unified interface to retrieve, create, delete or update resources

Reference: RESTful Web Services, L. Richardson and S. Ruby, O’Reilly.

Origins of REST

- REST is an acronym standing for Representational State Transfer
  - First introduced by Roy T. Fielding in his PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures", 2000.
  - The thesis focused on the rationale behind the design of the modern Web architecture and how it differs from other architectural styles.

REST Resources

- A resource has:
  - an unique identifier
  - one or more attributes beyond ID
  - a representation

- Examples:
  - Web Site, resume, aircraft, song, transaction, employee, application, blog post, printer, …

REST Resources

- Resources are identified by a URI (Uniform Resource Identifier)
  - http://www.example.com/software/release/1.0.3.tar.gz

- A resource has to have at least one URI

- Every URI refers to exactly one resource

Resource Representations

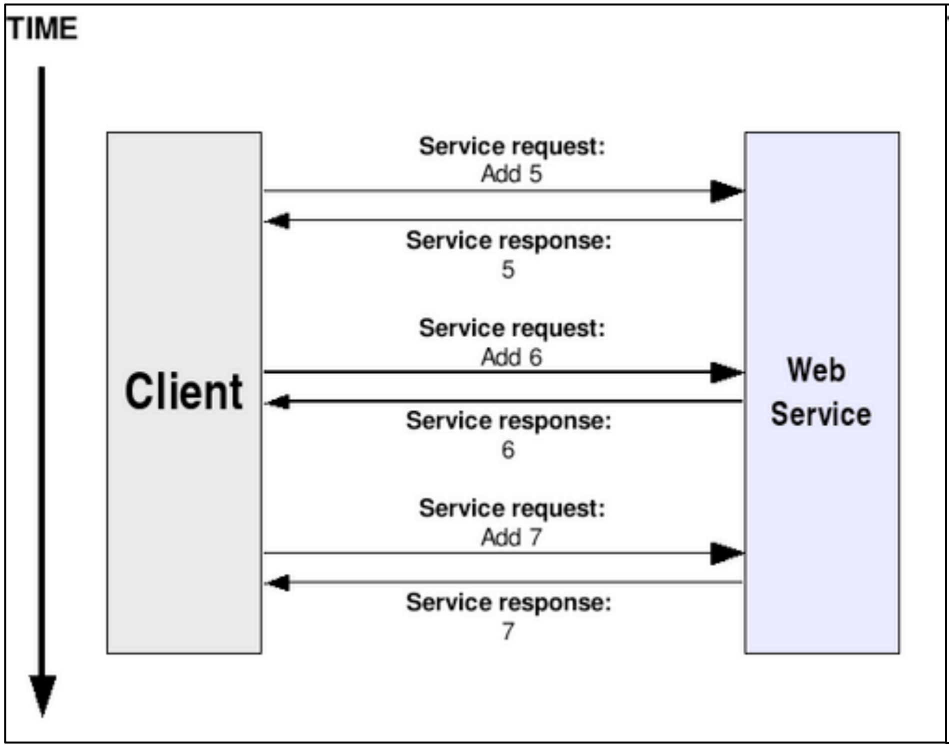
- A resource needs a representation: data forming the "current state" of a resource
  - e.g., a list of open bugs
  - Can be XML, a web page, comma-separated-values, printer-friendly-format, JSON,...
- Representation can flow in two ways:
  - The server returns a resource to the client
  - A client sends a representation of a new resource and the server creates the resource

Statelessness Server

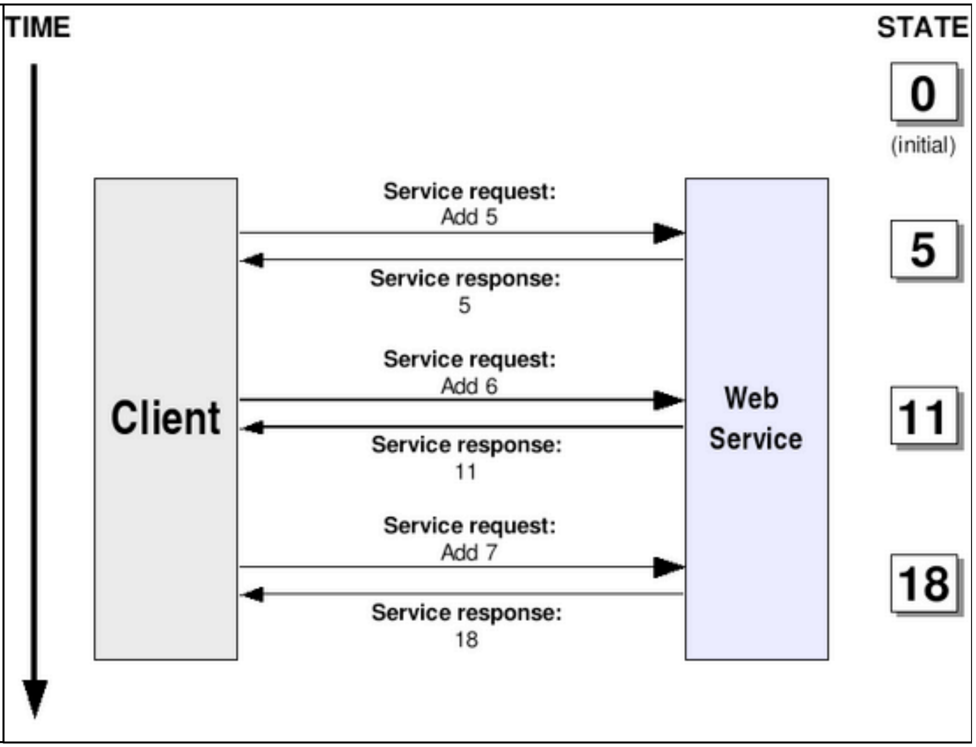
- RESTful service requires that the state stay on the client side
- Server does not keep track of the state
- When a client makes a request, it includes all necessary information for the server to fulfill the request.
  - URI needs to contain the state, not just a key to some state stored on the server
- Server supports the client in navigating the system by sending ’next links’ the client can follow

Statelessness Servers

Stateless



Statefull



What about a database on the server side?

Uniform Interface

- Similar to the CRUD (Create, Read, Update, Delete) databases operations

- REST *Uniform Interface Principle* uses 4 main HTTP methods
  - GET: Retrieve a representation of a resource.
  - POST: Create a new resource.
  - PUT: Update a resource (existing URI).
  - DELETE: Clear a resource, afterwards the URI is no longer valid

Why important?

- Let a client make reliable HTTP requests over an unreliable network.
  - Your GET request gets no response? Retry, it's safe
  - Your PUT request gets no response? Retry – even if your earlier one got through, your second PUT will have no side-effect

- Do not misuse HTTP interface conventions, e.g.,
  - GET https://api.del.icio.us/posts/delete
  - GET www.example.com/registration?new=true&name=aaa&ph=123

Microservices (next lecture)