Example use case

Use case: **Open file**

Related use cases:
Generalization of:

- Open file by typing name
- Open file by browsing

Steps:

Actor Actions	**System responses**
1. Choose "Open…" command	2. File open dialog appears
3. Specify filename	
4. Confirm selection	5. Dialog disappears

Use case: **Open file by typing name**

Related use cases:
Specialization of: Open file

Steps:

Actor Actions	System responses
1. Choose "Open…" command	
3a. Select text field	
3b. Type file name	
	2. File open dialog appears
4. Click 'Open'	
	5. Dialog disappears

Use case: **Open file by browsing**

Related use cases:
Specialization of: Open file
Includes: Browse for file

Steps:

Actor Actions	System responses
1. Choose "Open…" command	2. File open dialog appears
3. Browse for file	
4. Confirm selection	
	5. Dialog disappears

Use case: **Attempt to open a nonexistent file**

Related use cases:
Extension of: Open file by typing name

Steps:

Actor Actions	System responses
1. Choose "Open…" command	
	2. File open dialog appears
3a. Select text field	
3b. Type file name	
4. Click 'Open'	
	5. System indicates file does not exist
6. Correct the file name	
7. Click 'Open'	
	8. Dialog disappears

Use case: **Browse for file (inclusion)**

Steps:

Actor Actions	System responses
1. If the desired file is not displayed, select a directory	2. Contents of directory is displayed
3. Repeat step 1 until desired file is displayed	
4. Select a file	

Manage the requirements

- Review them
- Track them
- Verify them
- Control the changes to them

Reviewing requirements

- Each requirement should…
  - Have benefits that outweigh the costs
  - Be important for the solution
  - Be unambiguous
  - Be logically consistent
  - Lead to a quality system
  - Be realistic
  - Be verifiable
  - Be uniquely identifiable
  - Not over-constrain the design of the system

Remember

- The purpose of exploring requirements is to reduce the uncertainty about
  - what was needed and not requested
  - what was asked for and not needed

Problems

- Unknown – missing requirements
- Incomplete requirements
- Ambiguous requirements
- Non-requirements

Sample questions

- Who is the client?
- Who is the user?
- When do you really need it?
- How much time do we have for this project?
- Will real users be available to help test the product?
- Can we copy or modify something that already exists?

Product questions

- What is the skill level of the users?
- What environment is this product likely to encounter?
- What are the performance and resource constraints?
- What are the safety and security needs?

Meta questions

- Am I asking you too many questions at this time?
- Do my questions seem relevant?
- Are you the right person to answer these questions?
- Are your answers official?

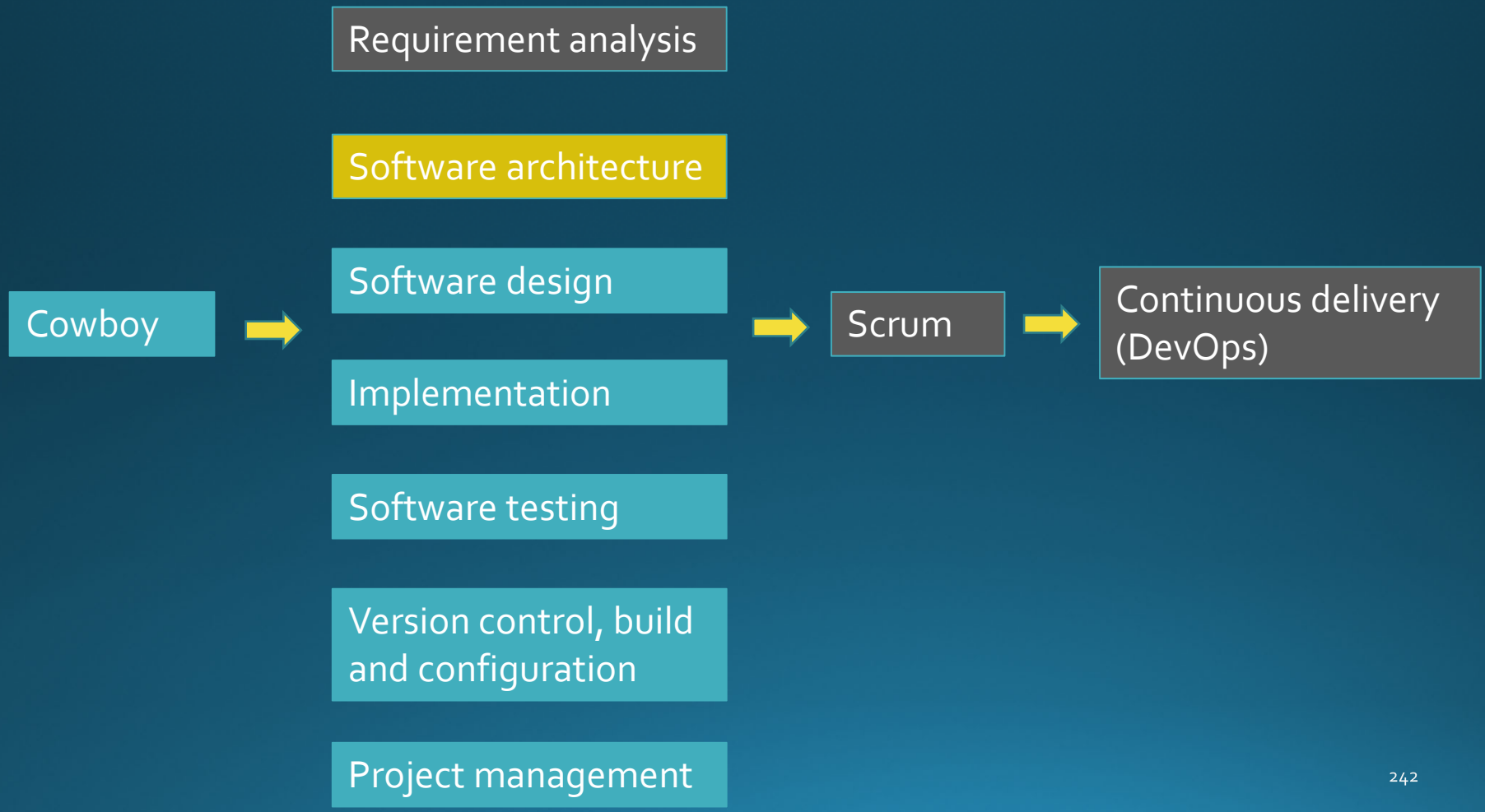
Ending questions

- Is there anything else I should be asking you?
- Is there anything you would like to ask me?
- May I ask you more questions at a later time to help cover things we might have overlooked?

Requirements document (product backlog)

A. Problem
  B. Background Information
  C. Requirements
    a. Functional requirements
    b. Non-functional requirements

What is the course about



Software Architecture

- Software architecture
- Architectural patterns
  - What are they?
  - Multi-layer pattern
  - Client-server architecture
  - Transaction-processing pattern
  - Pipe-and-filter
  - MVC
  - Service-oriented

Software Architecture

- Set of structures that can be used to reason about a system
- Comprises software elements, relations among them, and properties of both
- More abstract than software design

* some slides based on material developed by former CS student Joe Koncel

Importance

- Enables everyone to better understand the system
- Allows people to work on individual pieces in isolation
- Prepares for extension of the system
- Facilitates reuse and reusability
- Determines overall efficiency, reusability, and maintainability of system
- Expensive to change once implemented

Good architectural models

- Contain a logical breakdown of subsystems
  - Separation of concerns (e.g., functionality first and then efficiency, dataflow and control flow views)
- Document interfaces between subsystems
- Capture dynamics of component interactions
- Outline shared data
- Are quality-driven

Stability

- Architectural models should be stable
  - Ensure maintainability and reliability
- Stable means features and components can be added or changed without impacting the overall architecture

Developing an architectural model

- Start by sketching an outline of the architecture
  - Based on principal requirements and use cases
- Determine the main components
- Apply architectural patterns when appropriate

Refine the architecture

- Decide how data and functionality will be distributed among the components
- Identify main ways components interact and their interfaces
- Decide if a framework exists that can be re-used
- Consider each use case and adjust the architecture to make it realizable

Architectural patterns

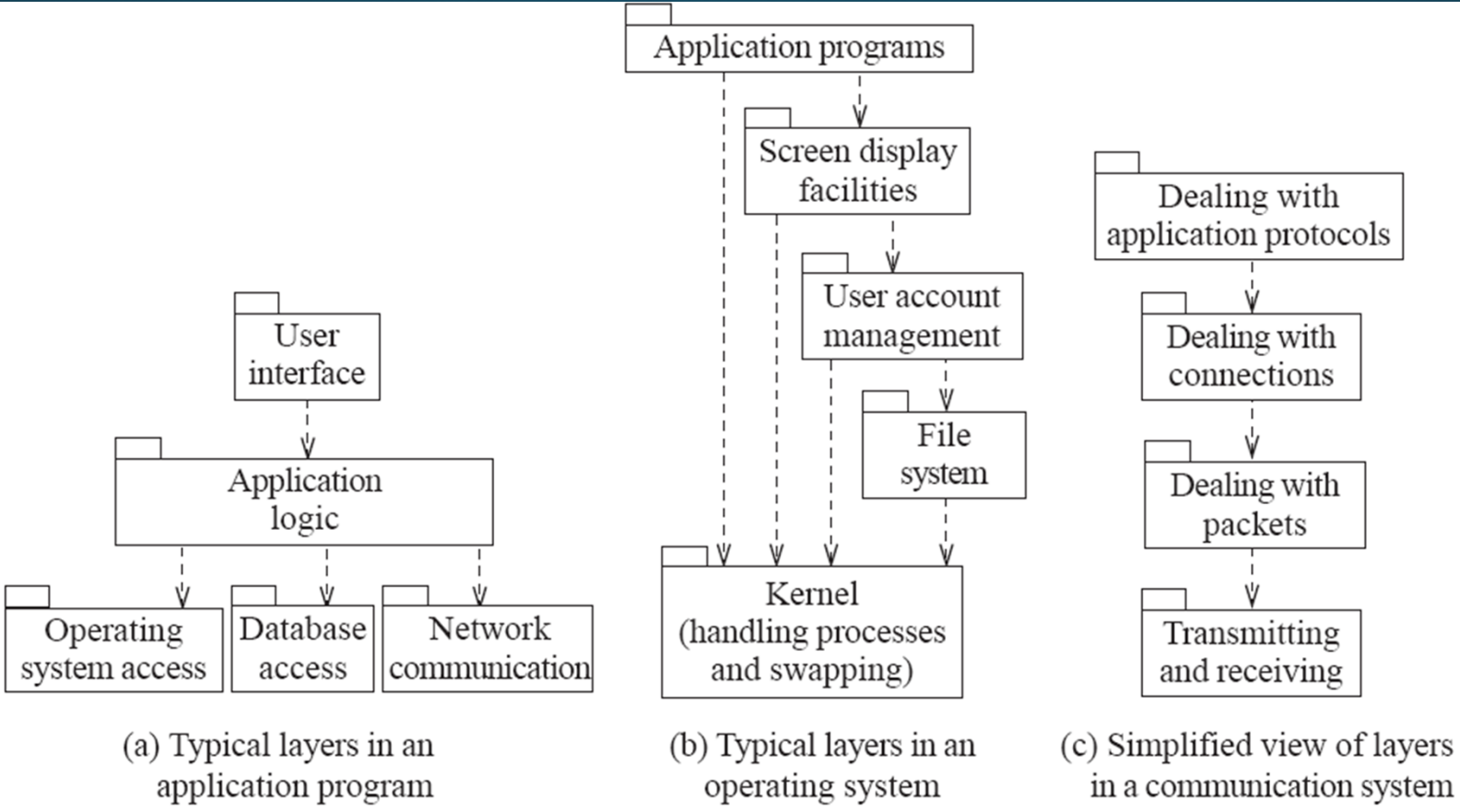
- Architectural patterns/styles that have been proved to be effective (for certain kind of software)
- Each pattern has…
  - A context
  - A problem
  - A solution

Multi-Layer pattern

- Problem – system components need to be built and tested independently
- Solution – define layers (groupings of cohesive modules) and a unidirectional allowed-to-use relation among the layers
  - Often illustrated with stacked boxes representing layers on top of each other

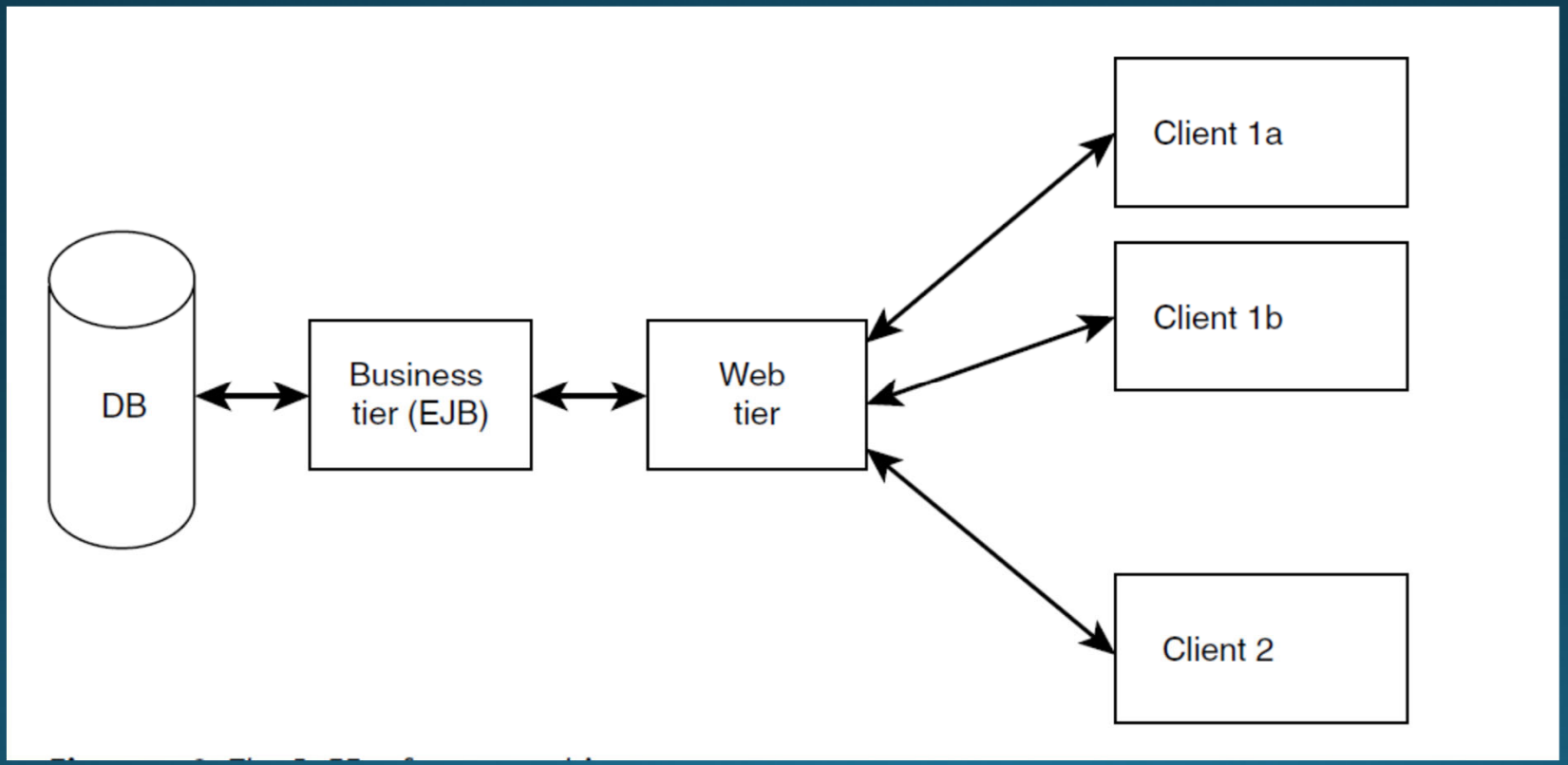
- Separate layer for UI
- Layers below UI provide application functions
  - Determined by use cases
- Bottom layers provide general services
  - Network communication
  - Database access
  - etc

Example



Three-tier architecture

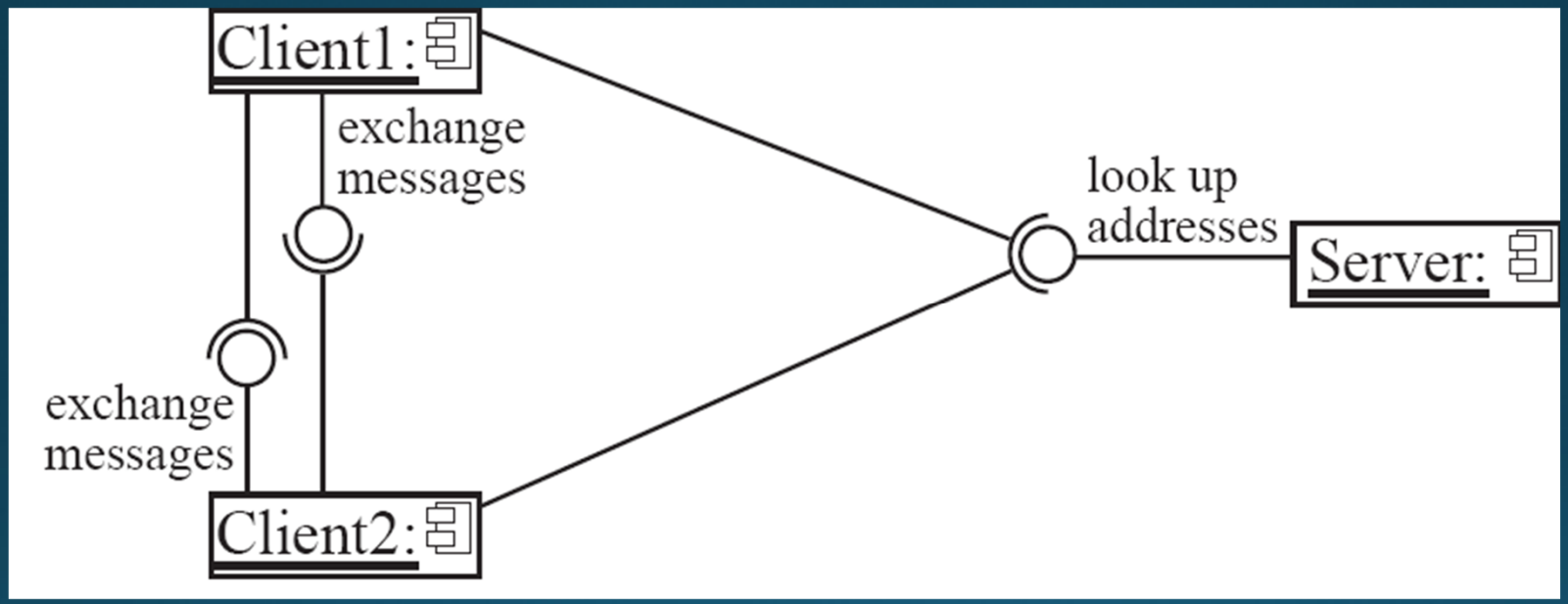
- Popular for web applications



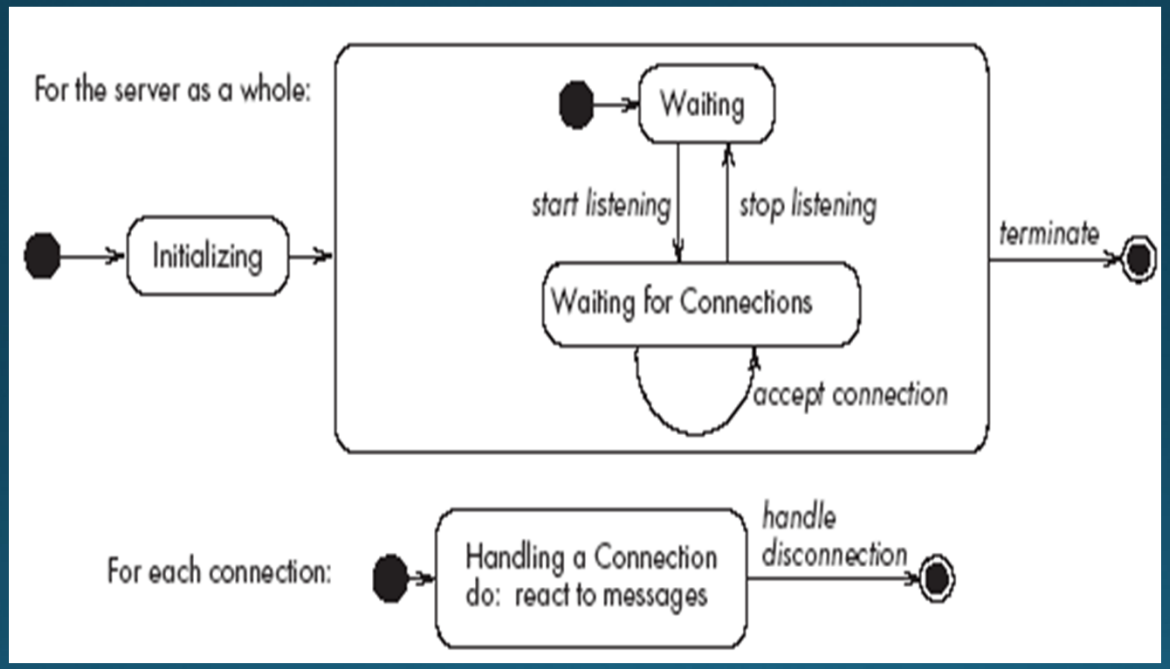
Client-server architecture

- Problem – large number of distributed clients need access to shared resources or services
- Solution – client components initiate interactions with server components, invoking services as needed and waiting on results

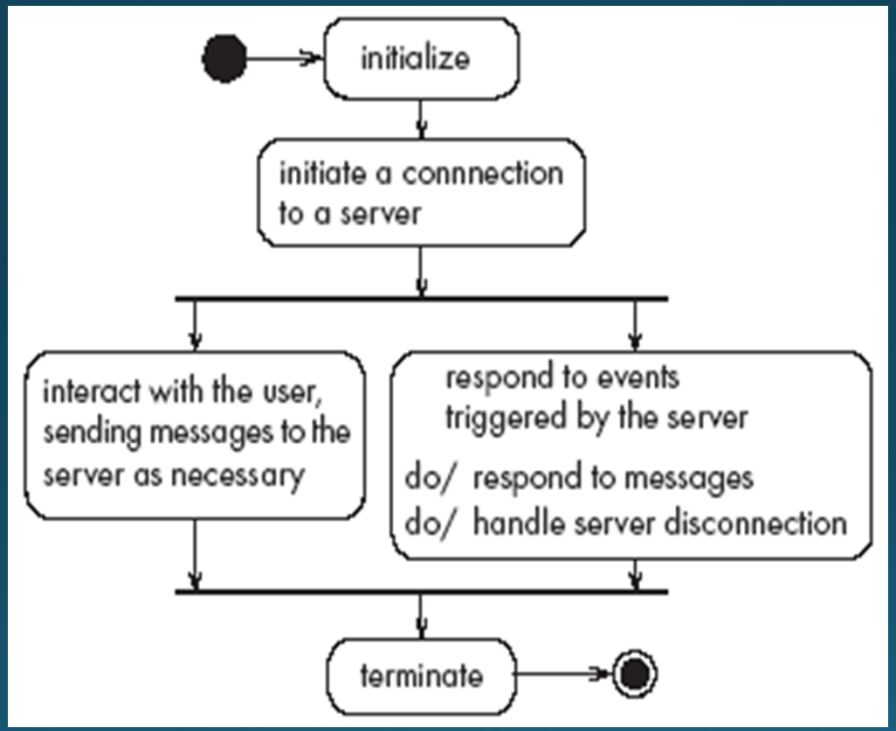
Example



Server



Client

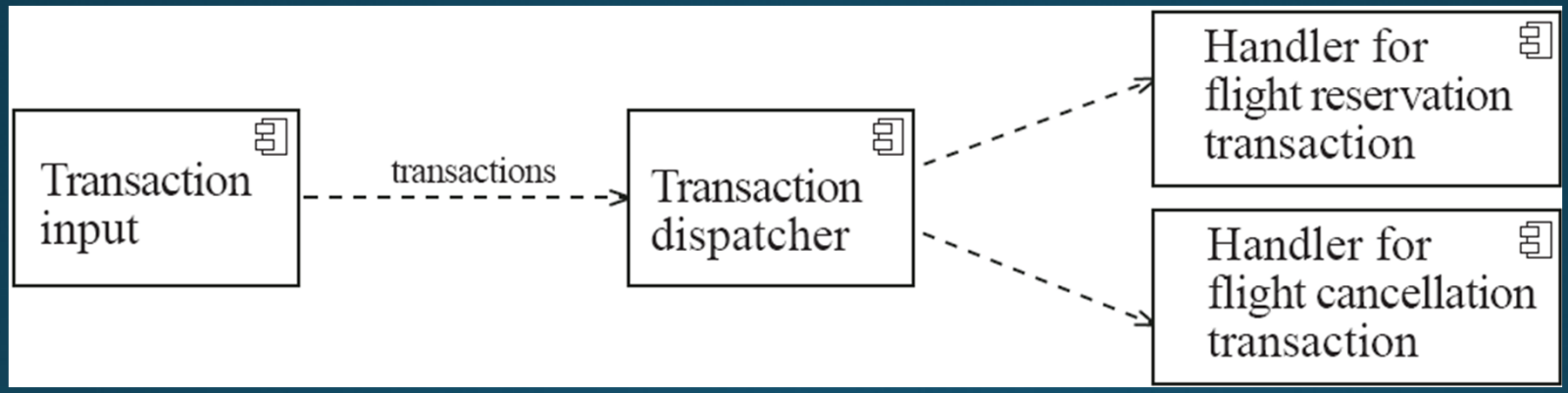


Sequence

- Server starts running
  - Creates a socket
  - Binds the socket to an address
  - Waits for clients (listening)
- Client requests something from the server
  - Creates a socket
  - Attempts to connect to server
- Server accepts the connection
- Send and receive data (read and write)

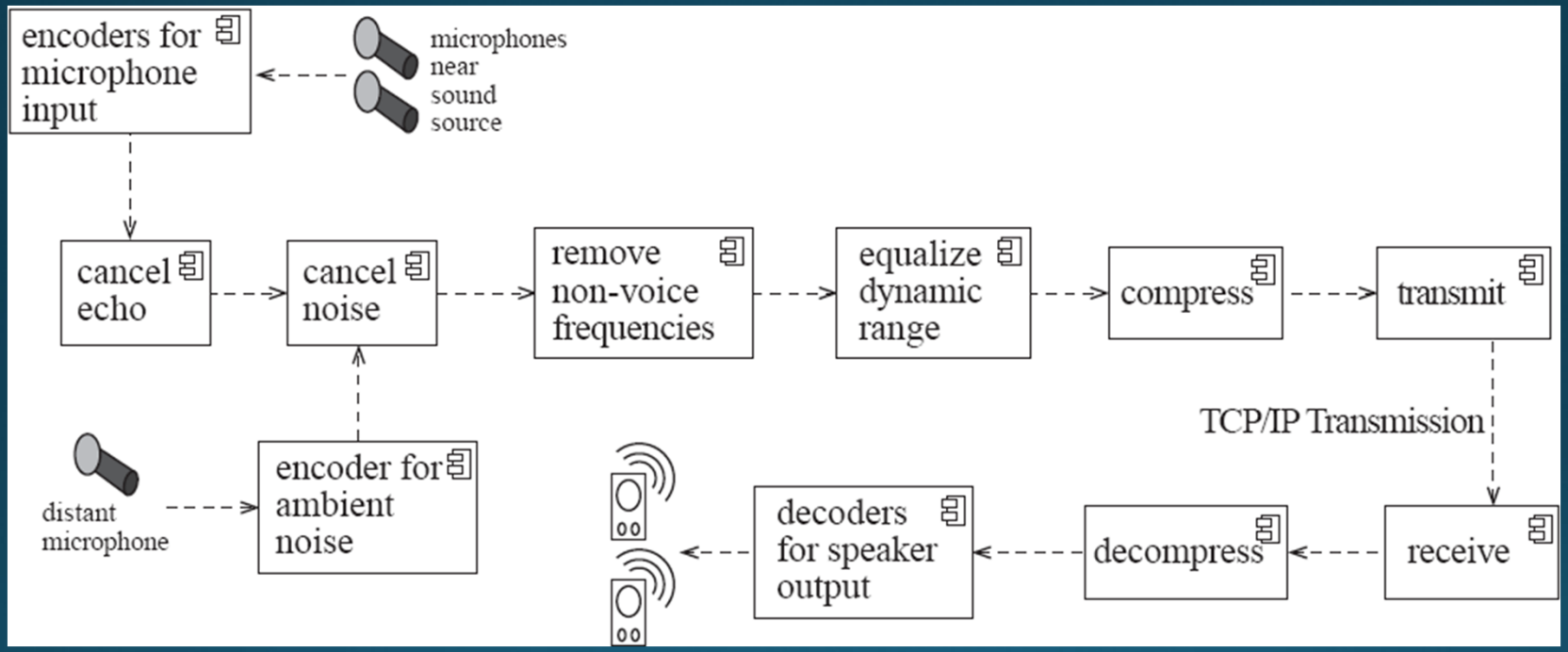
Transaction-processing pattern (event driven architecture)

- Problem – system must read and handle series of inputs that change stored data
- Solution – dispatcher component that decides how to handle each transaction (input), calling a procedure or messaging a component



Pipe-and-filter

- Stream of data is passed through a series of processes
  - Each transforms it in some way
  - Data is constantly fed into the pipeline
  - Processes work concurrently
- Architecture is flexible
  - Components could be added, removed, replaced, sometimes reordered

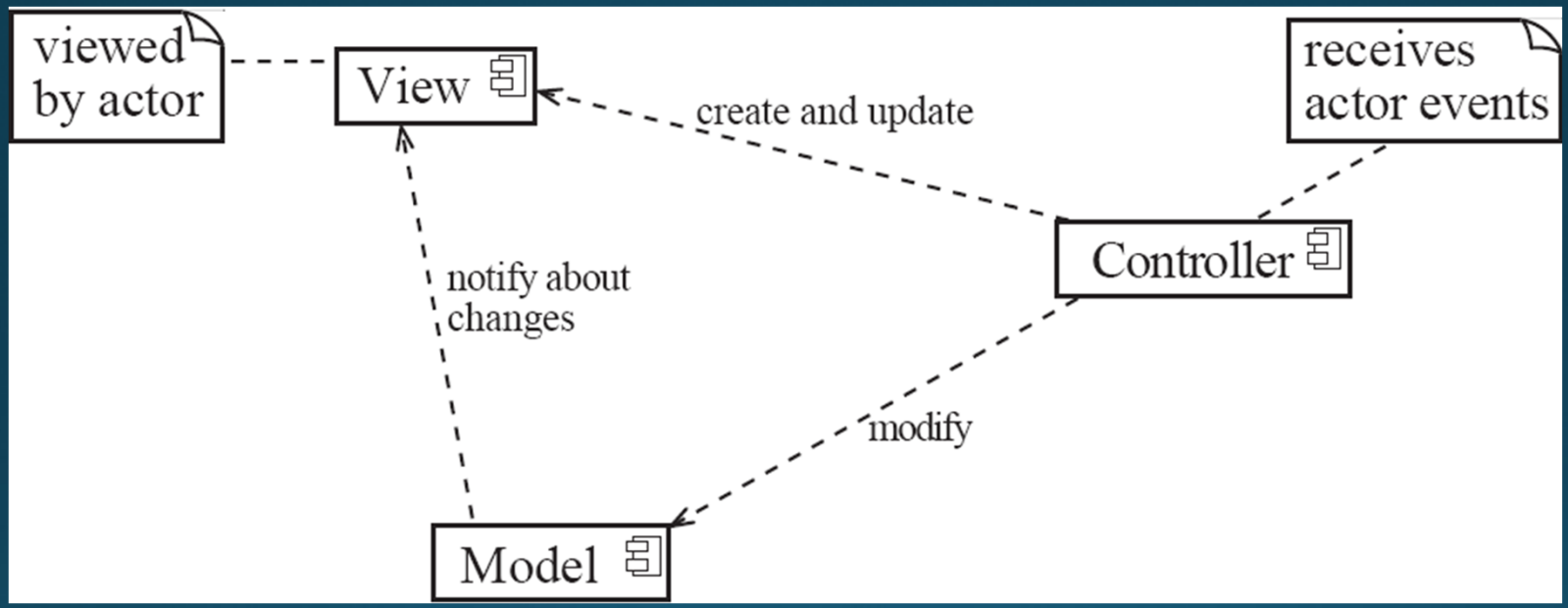


Model-View-Controller (MVC)

- Problem – UI needs frequent modification without impacting system's functionality
- Solution – break system into three components – model, view, and controller
  - controller mediates between the model and the view

MVC

- Model contains underlying classes
  - Instances are viewed and manipulated
- View contains objects that render the appearance (UI) of data from the model
- Controller contains objects that control and handle user's interaction with the view and the model

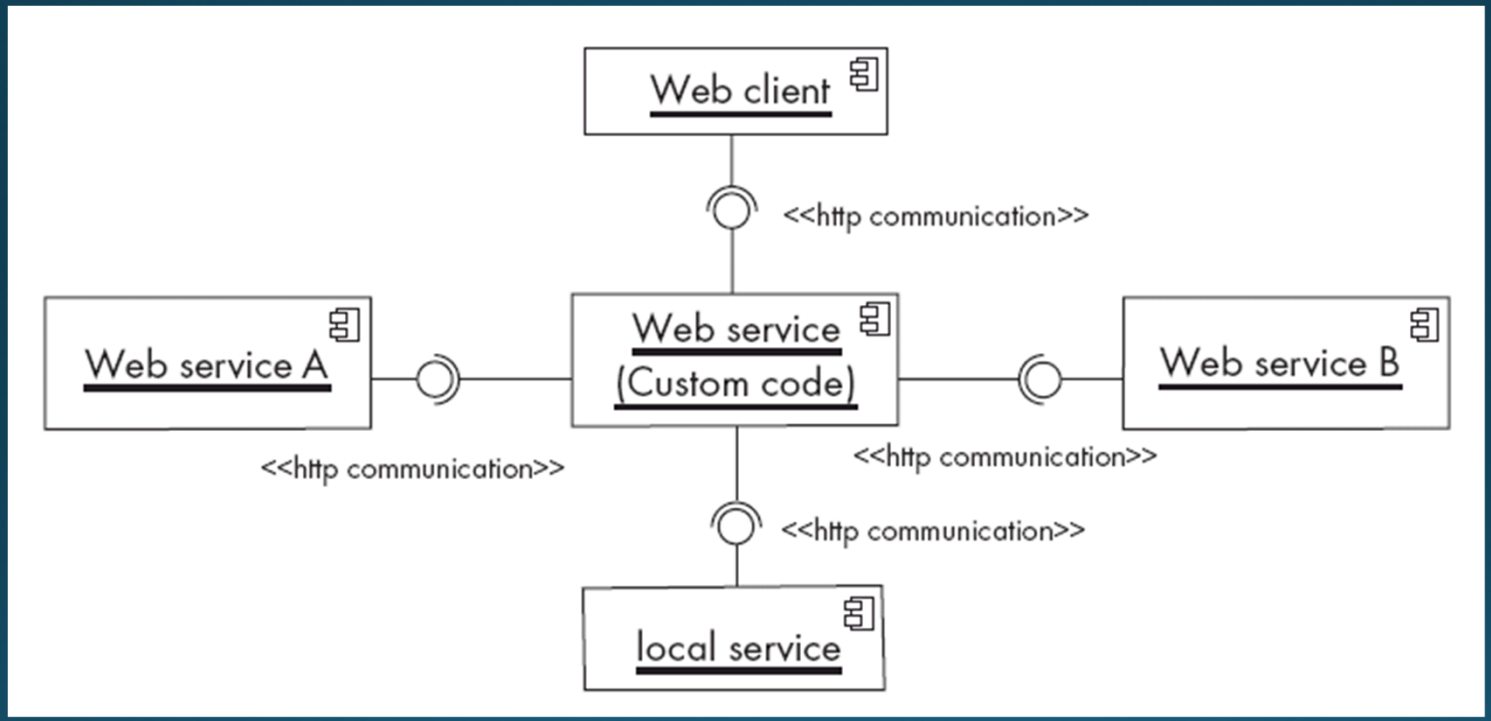


MVC on the WWW

- View component generates HTML
  - Displayed by browser
- Controller interprets HTTP POSTs from the browser
- Model is the underlying system
  - Manages the information

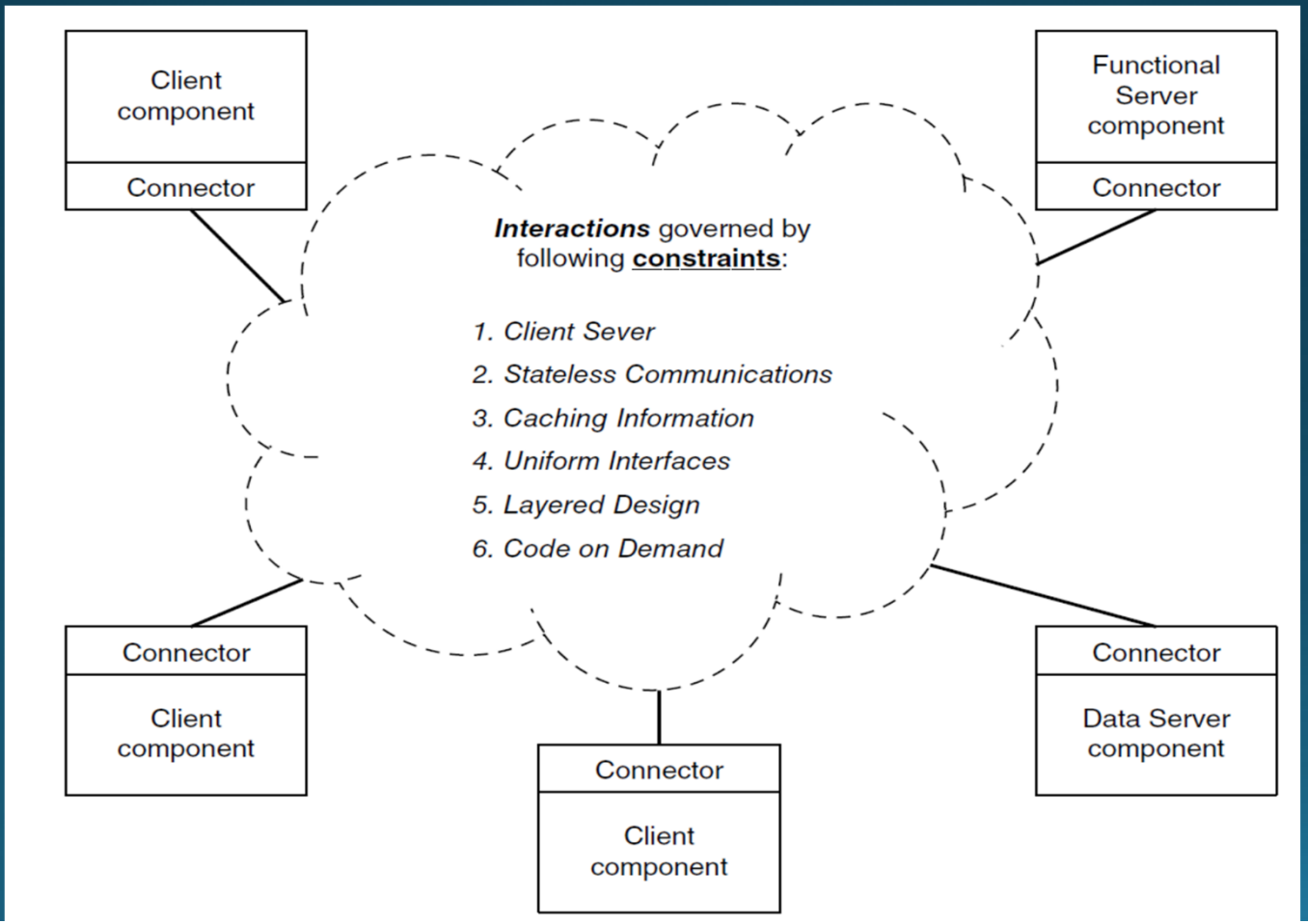
Service-oriented

- Problem – service consumers must be able to use/access a number of service providers
  - Without understanding the implementation
- Solution – cooperating peers that request service from and provide services to one another across a network
  - Called web services on the Internet



REST architecture

- An architectural style by Roy Fielding in his doctorate thesis at the University California, Irvine for distributed web applications



REST architecture

- *Client-server design style:* separating user interface from the rest of the functional and data processing improves the portability and scalability
- *Communications between components must be stateless:* scalability
- *Caching information:* information that is "labeled" as cacheable is reused by the client
- *A uniform interface between components* (e.g., http)
  - resources have unique identification
  - enough information is attached to a representation of resource such that resource manipulation is enabled
  - self-descriptive messages
  - client state transitions occur within the context of hypermedia
- *Layered design style*
- *Code-on-demand*