Lecture 7: Architecture 2

CS 5150, Spring 2026

Previously on 5150…

Design steps

- Given requirements, must design a system to meet them
    - System architecture
    - User experience
    - Program design

- **Ideal**: requirements are independent of design (avoid implementation bias)
- **Reality**: working on design clarifies requirements
    - Methodology should allow feedback (strength of iterative & agile methods)

Design principles

- Design is an especially **creative** part of the software development process
  - More a "craft" than a science
  - Many tools are available; must select appropriate ones for a given project

- Strive for simplicity
  - Use modeling, abstraction to (hopefully) find simple ways to achieve complex requirements
  - Designs should be easy to implement, test, and maintain
- Easy to use correctly, hard to use incorrectly
- **Low coupling, high cohesion**

Software Architecture

Software architecture *is the* ***set of structures*** *needed to reason about a software system and the discipline of creating such structures and systems. Each structure comprises* ***software elements***, ***relations*** *among them, and* ***properties*** *of both elements and relations.* [Bass et al. 2003]

Note: this definition is ambivalent to whether this architecture is known or any good!

Software Architecture: Other Definitions

- Brooks: Conceptual integrity is key to usability and maintainability; architect maintains conceptual integrity
- Johnson: *The shared understanding that expert developers have of the system / The decisions you wish you could get right early in a project*
- Sommerville: Dominant influence on non-functional system characteristics
- "Highest level" organization of system

Levels of abstraction

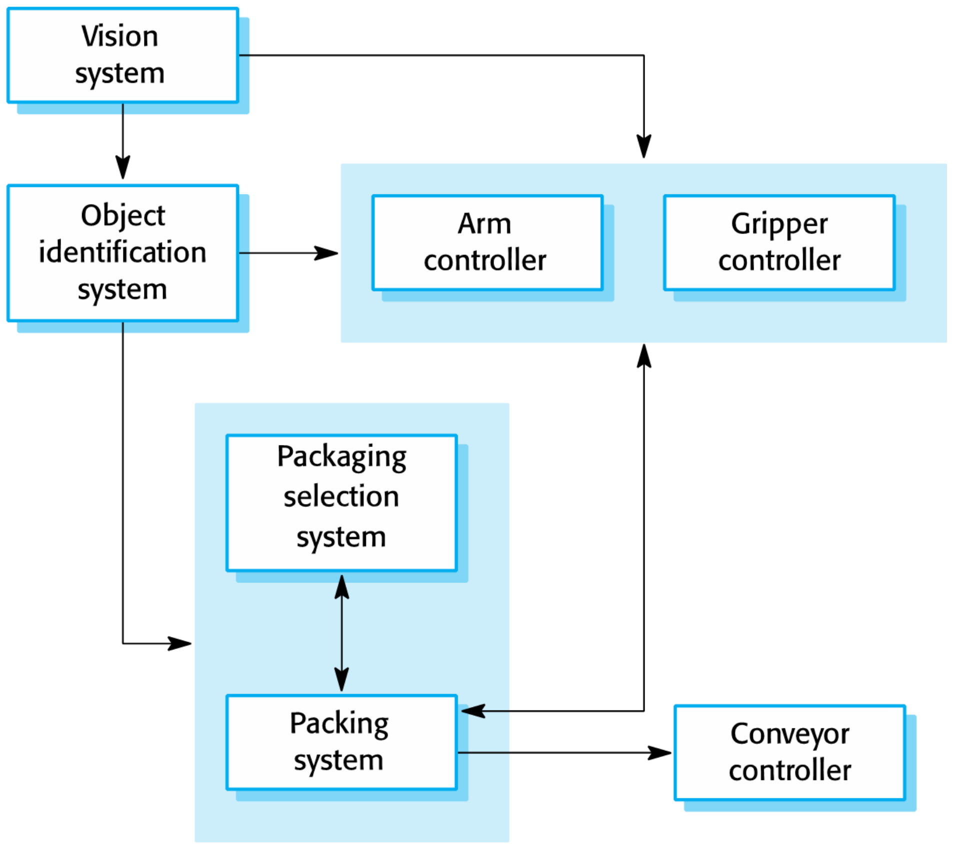
- Requirements
  - High-level "what” needs to be done
- Architecture
  - High-level "how"
  - Mid-level "what"
- Program design (Design patterns)
  - Mid-level "how"
  - Low-level "what"
- Code
  - Low-level "how"

- Documentation for each step should respect its level of abstraction
  - Avoid biasing later steps
  - Avoid redundancy

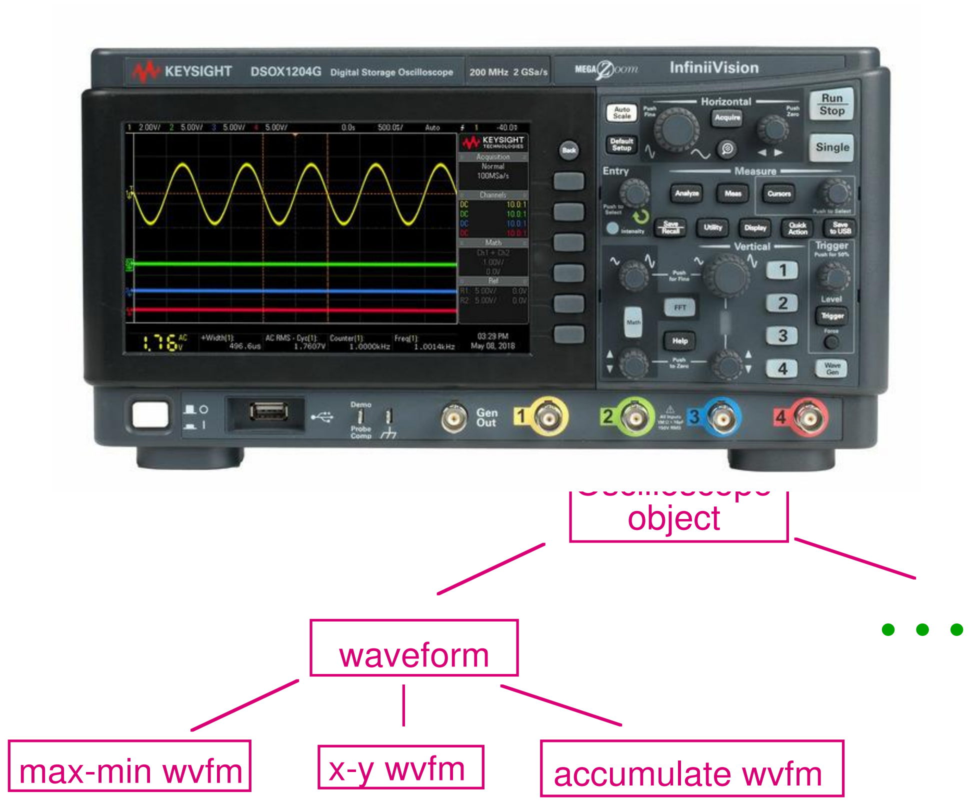
Architectural considerations

- Infrastructure
  - Hardware
  - Operating systems
  - Virtualization
- Interfaces
  - Networks/buses
  - Protocols
- Services
  - Databases
  - Authentication
- Operations
  - Testing
  - Logging/monitoring
  - Backups
  - Rolling deployment
- Product line
  - Staging, A/B Testing, …

Examples



Packing Robot control system



Oscilloscope

Sommerville, *Software Engineering*

Garlan & Shaw, "An Introduction to Software Architecture"

Subsystems

- Improve comprehensibility of system by decomposing into subsystems
- Group elements into subsystems to minimize coupling while maintaining cohesion

- Coupling: Dependencies *between* two subsystems
  - If coupling is high, can't change one without affecting the other
- Cohesion: Dependencies *within* a subsystem
  - High cohesion implies closely-related functionality

Today's Lecture Goals

- Modelling Architecture
- Architectural Styles
- Encapsulate deployments using virtualization

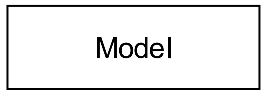
How are you feeling about the project?

PollEv.com/cs5150sp26

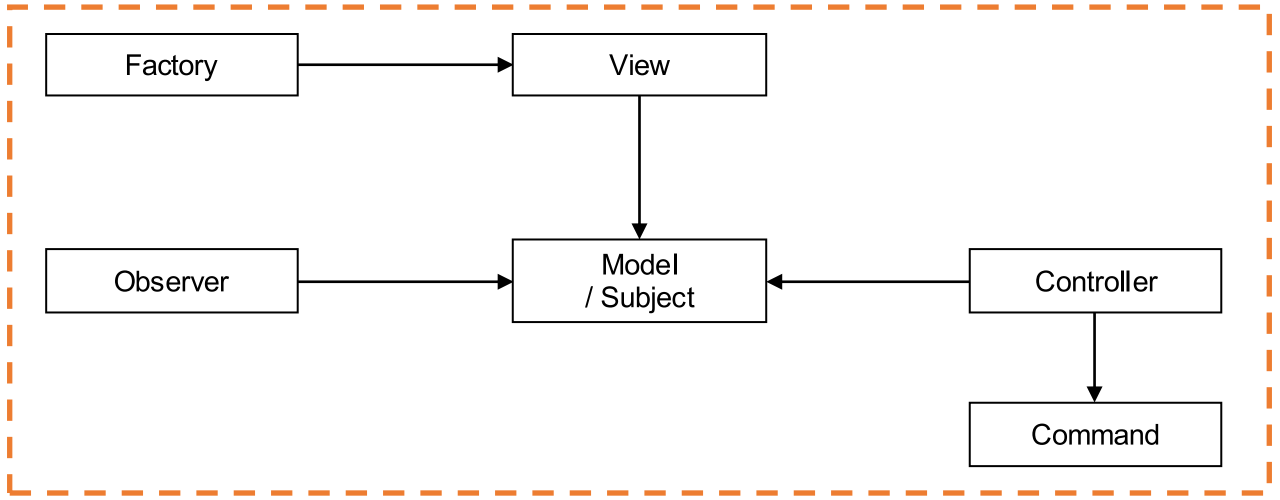
Design vs. Architecture

- Design Questions:
  - How can I add a new menu item in Zulip?
  - How does Google rank pages?
  - What is the interface between objects?
  - How should the communication protocol work?
- Architectural Questions:
  - How do I extend Zulip with a plugin/plugin architecture?
  - How does Google scale to billions of hits per day?
  - Where should I put my firewalls?

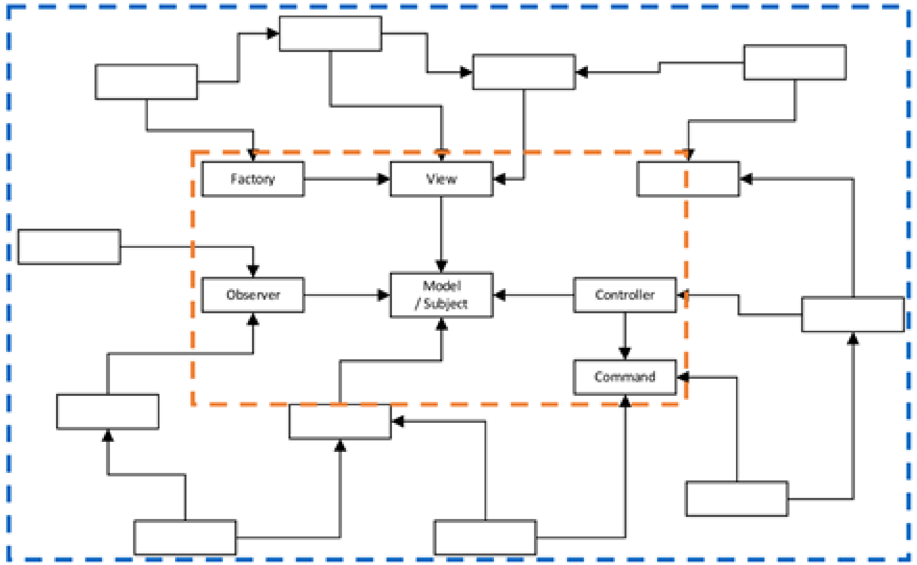
Objects



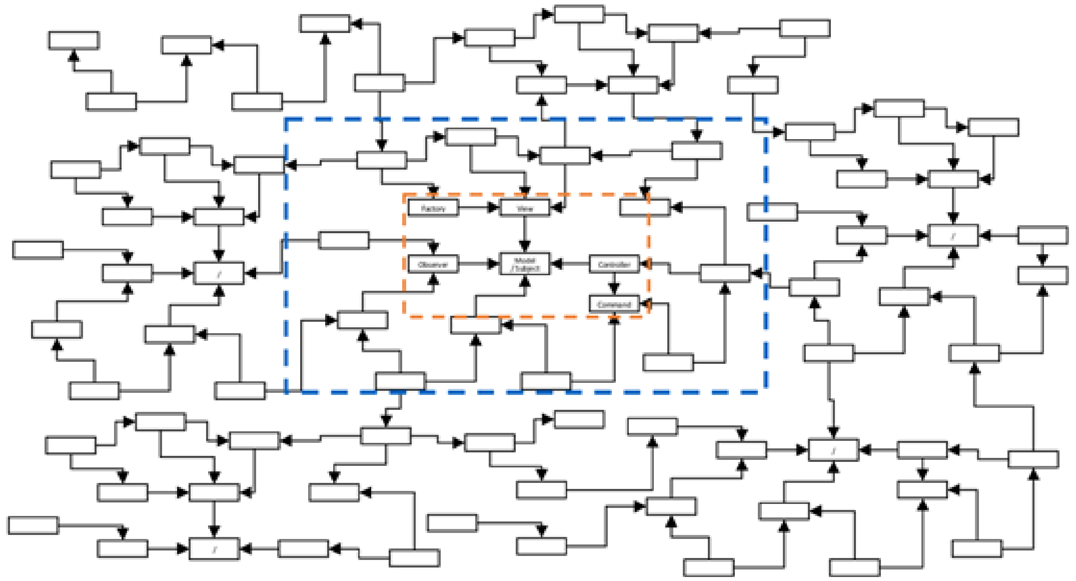
Design Patterns



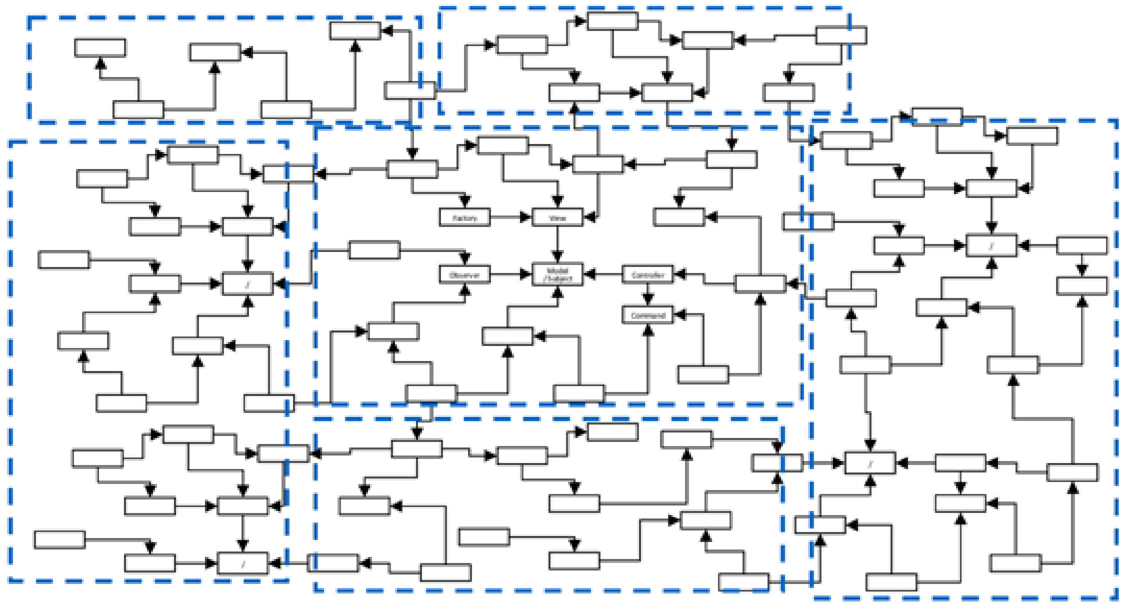
Design Patterns



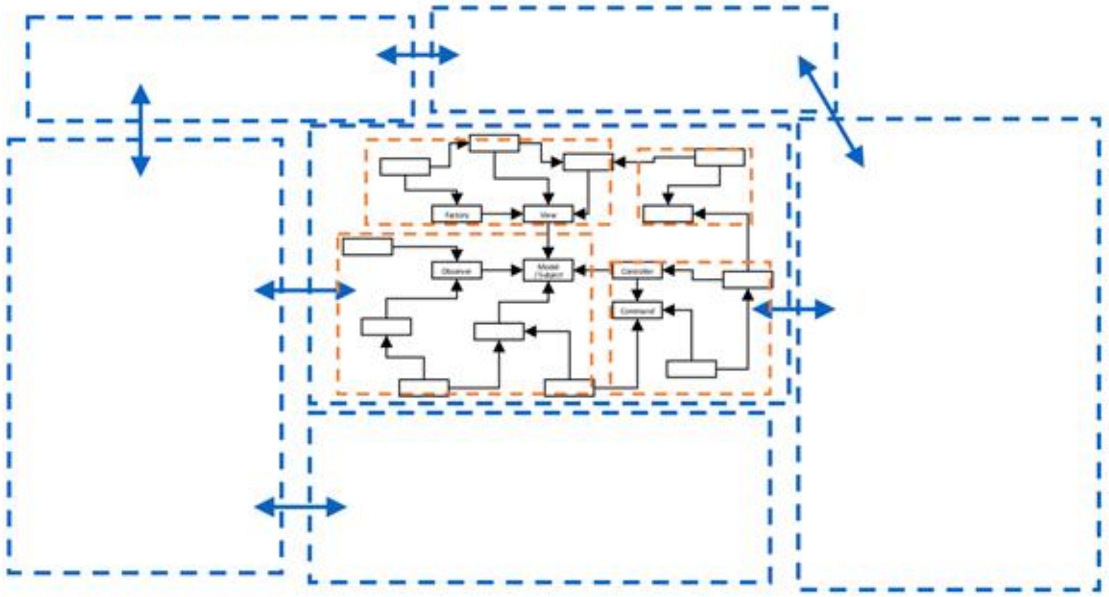
Design Patterns



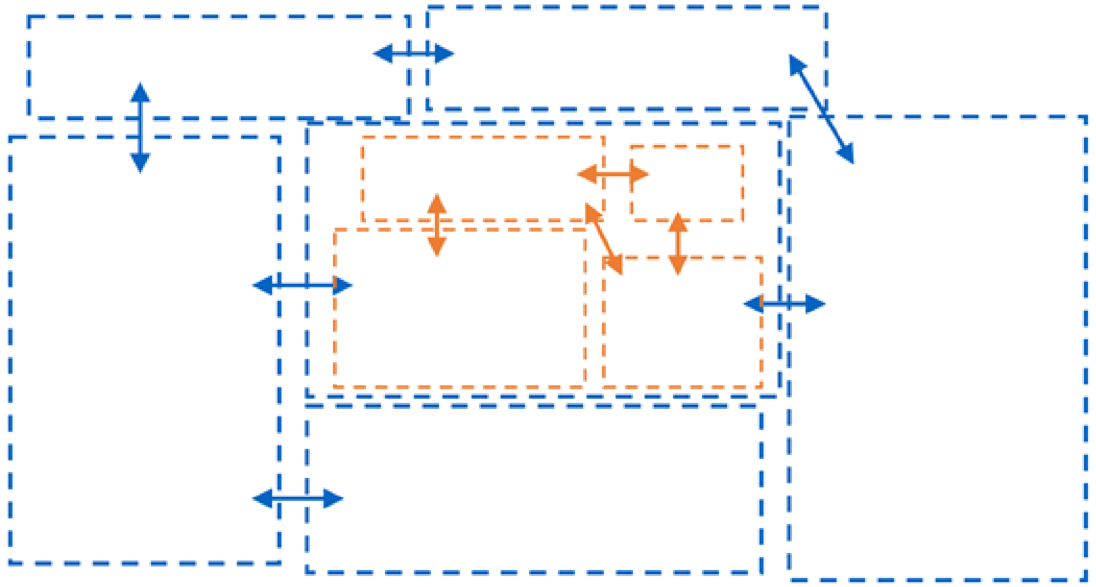
Architecture



Architecture

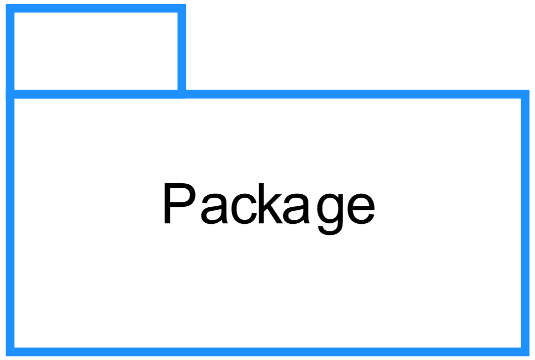


Architecture

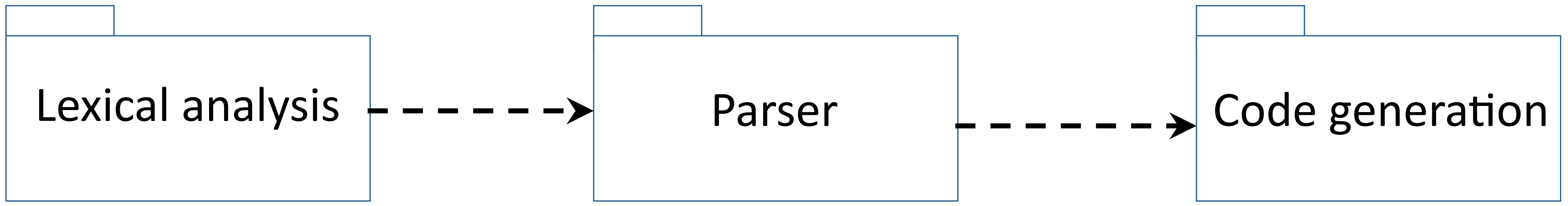


UML: Package

- General grouping of system elements
- Appropriate for denoting subsystem at conceptual level

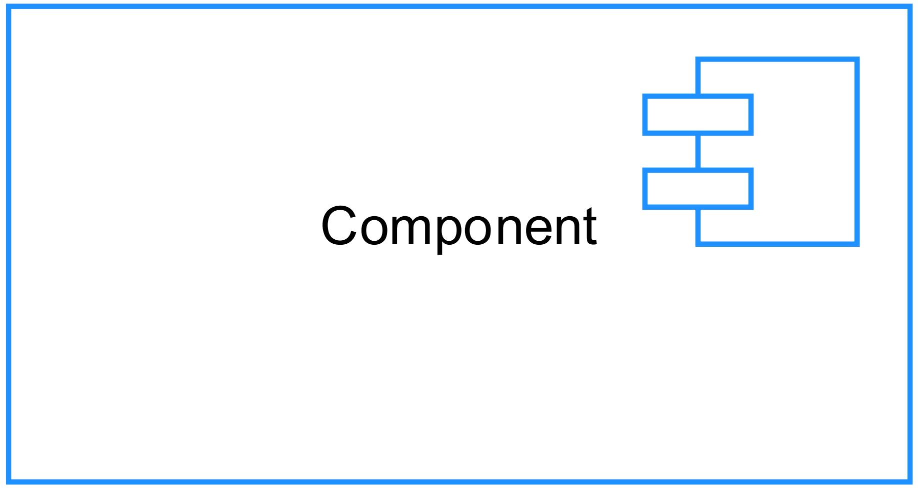


Example: conceptual diagram

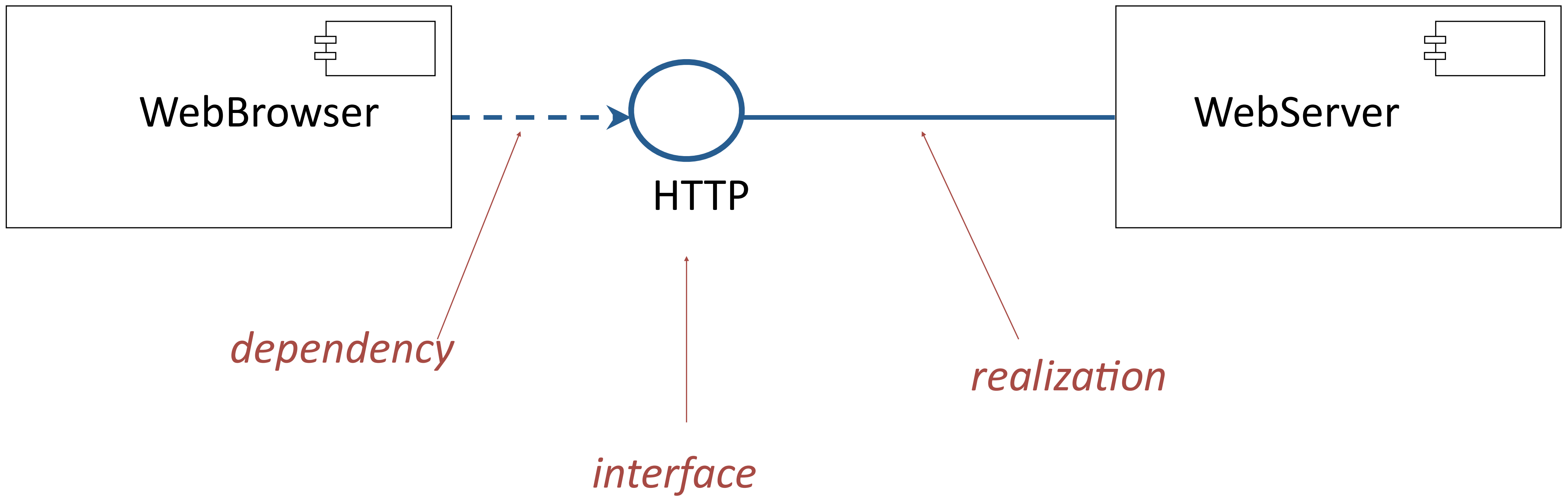


UML: Component

- Replaceable part of a system
  - Conforms to and realizes a set of interfaces
  - An implementation of a subsystem
  - Could be replaced by another component that realizes the same interfaces, and system would still function
- Distinct from classes
  - Classes may have fields, are assembled into programs
  - Components realize interfaces, are assembled into systems

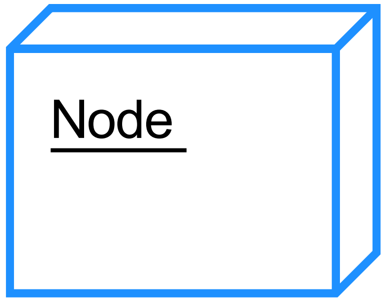


Example: interface diagram

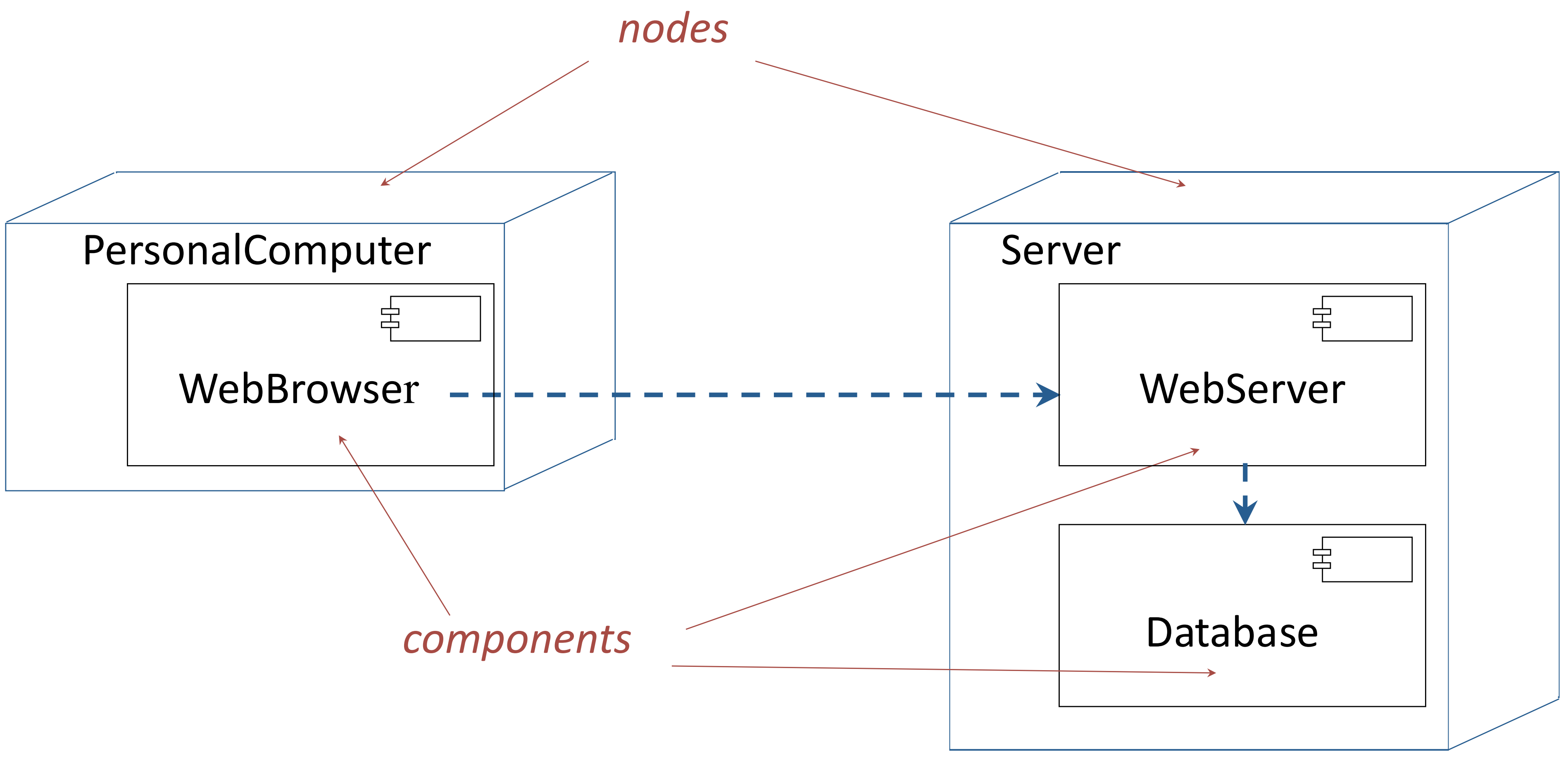


Node

- Physical element that exists at runtime, provides a computational resource
  - Computer
  - Smartphone
  - Network router
- Components live on nodes



Example: deployment diagram



Deployment environments

- Development

- Production
- Staging
- (Acceptance testing)

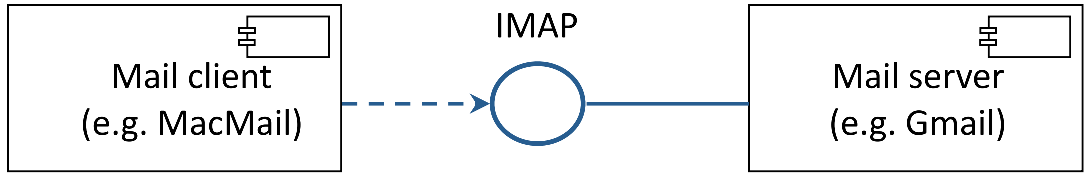
Poll: Account data will be stored in a PostgreSQL database running in a Docker container under Linux on an HPE ProLiant server that also runs the web server. Which architectural diagram is an appropriate place to show these details?

PollEv.com/cs5150sp26

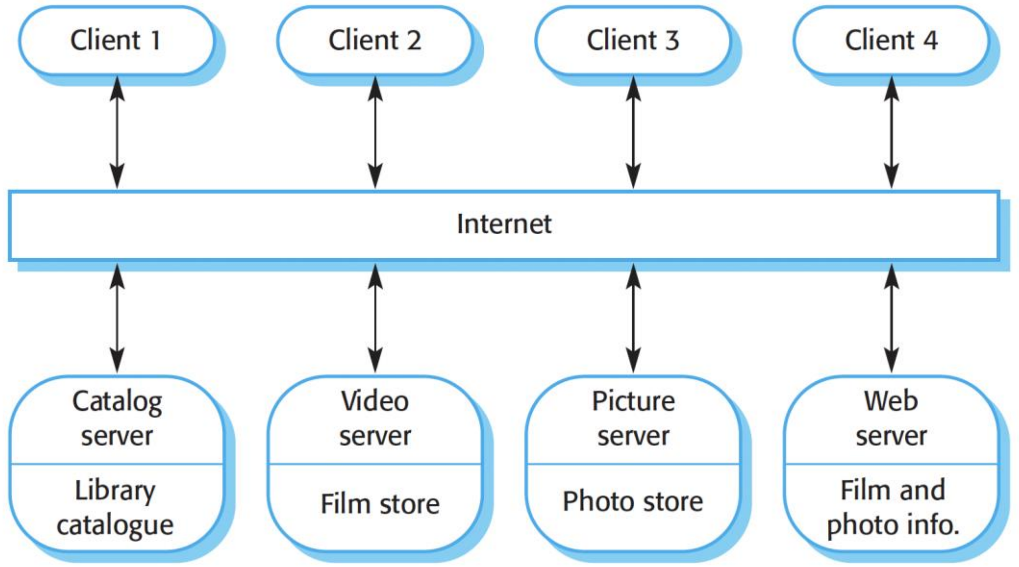
Architectural styles

System architecture (or portion thereof) that recurs in many different applications

Client/Server



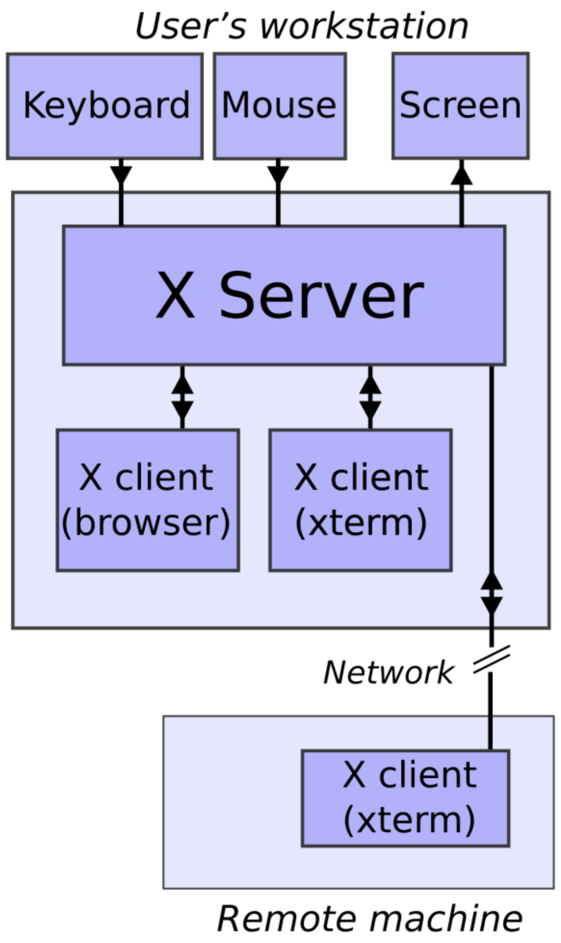
- Control flow in client and server are independent
- Communication follows a protocol
- If protocol is fixed, either side can be replaced independently
- Peer-to-peer: same component can act as both client and server
- Adv: Easy to scale, Clear separation
- Challenges: Latency, Tight API Coupling
- Example: Multiplayer games, Email, Internet servers



Q: Where do cookies and sessions fit in such architectures?

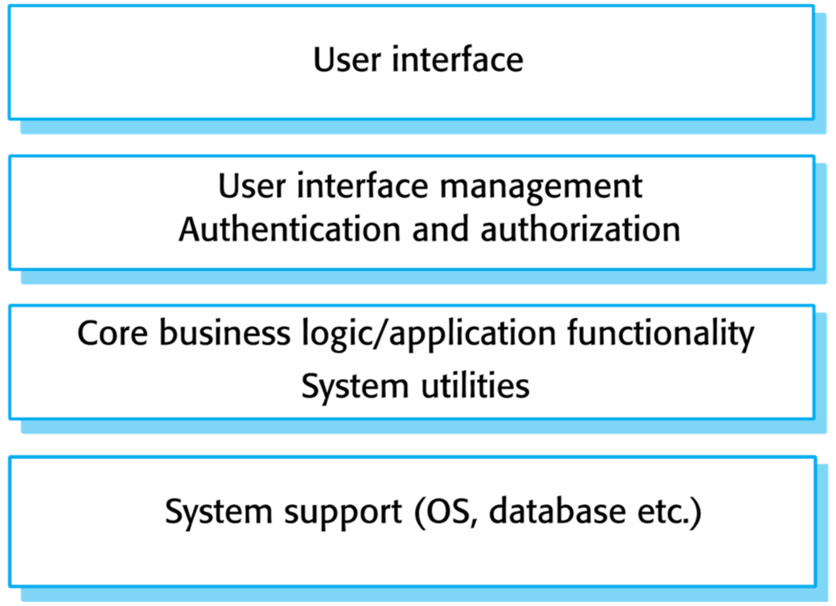
Example: X Window System (X11)

- X server runs on computer w graphic display
- Client application (browser) connects to server via X11 protocol
- Server send input events, client sends drawing commands
- Confusingly, “X11 client” runs on “application server”, while “X11 server” runs on “thin client”



Layered Architecture

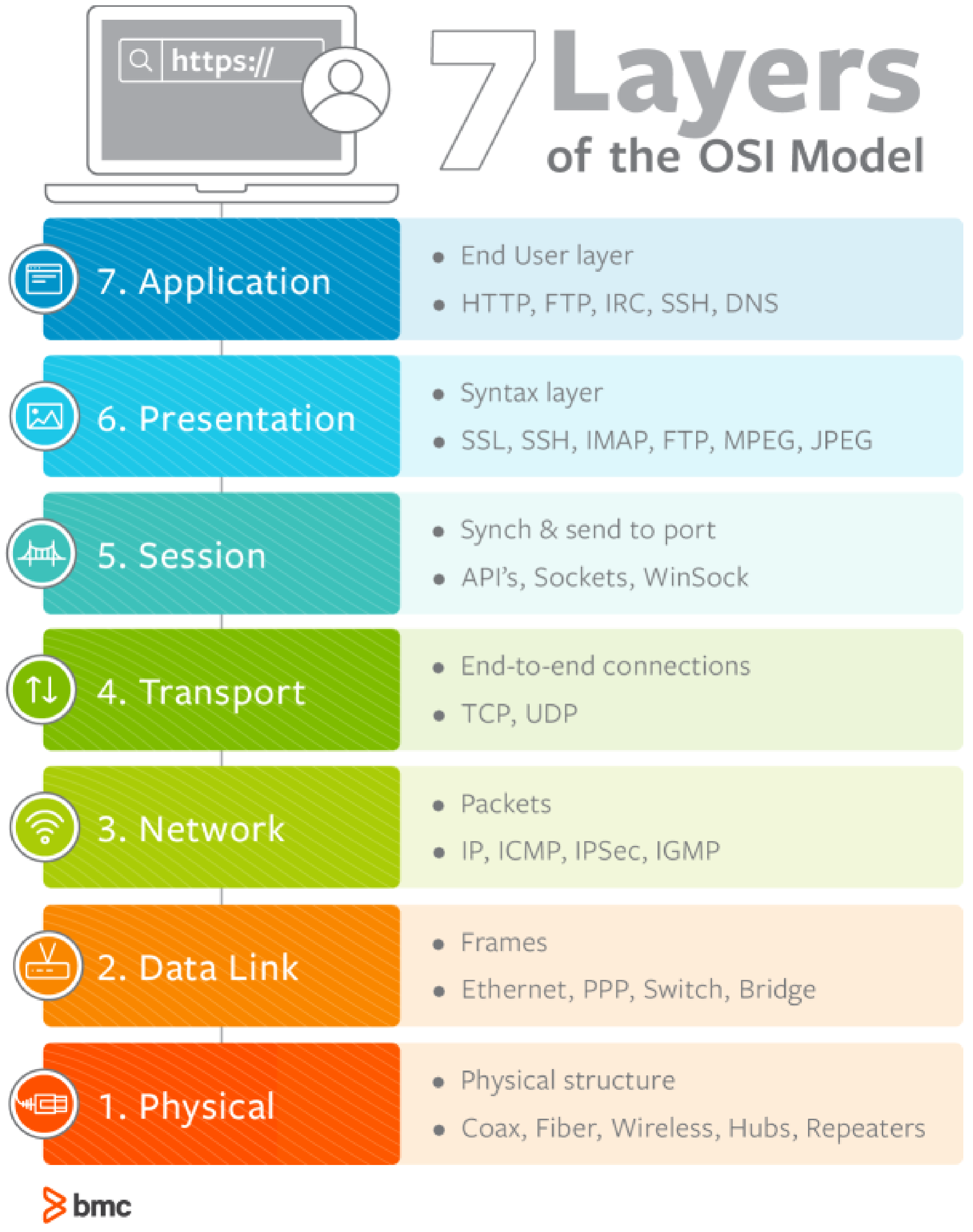
- Partition subsystems into stack of layers
  - Layer provides services to layer directly above
  - Layer relies on services to layer directly below
- Advantage:
  - Constrains coupling, Easy to understand
  - Easy to replace layers
  - Clear ownership/testing boundaries
- Danger: leaky abstractions
  - Clear separation is difficult
  - May need services of multiple lower layers
  - Performance


User interface
User interface management
Authentication and authorization
Core business logic/application functionality
System utilities
System support (OS, database etc.)

Sommerville, *Software Engineering*

Example

- OSI (Open Systems Interconnection) Reference Model
- Used for network protocols (TCP/IP)
- Other examples:
  - Operating Systems

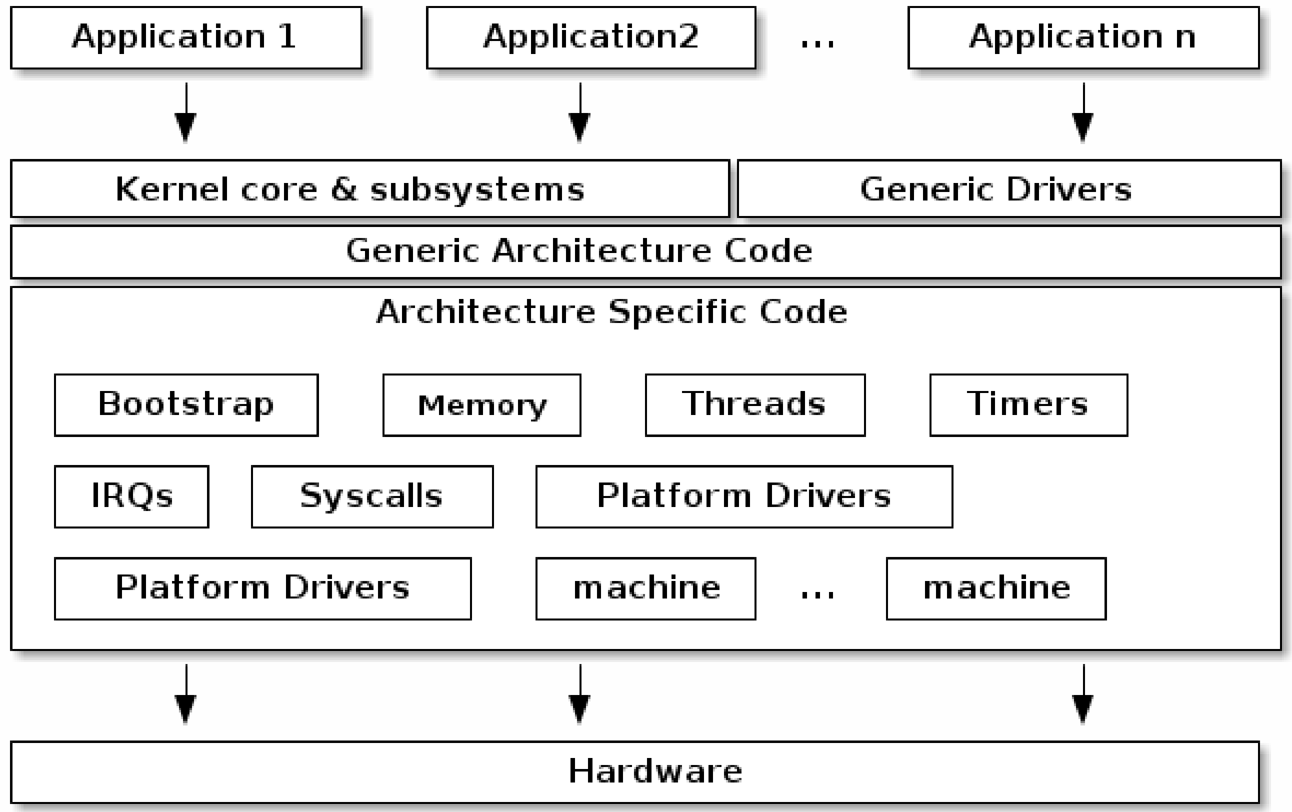


7 Layers of the OSI Model

7. Application	• End User layer • HTTP, FTP, IRC, SSH, DNS
6. Presentation	• Syntax layer • SSL, SSH, IMAP, FTP, MPEG, JPEG
5. Session	• Synch & send to port • API's, Sockets, WinSock
4. Transport	• End-to-end connections • TCP, UDP
3. Network	• Packets • IP, ICMP, IPSec, IGMP
2. Data Link	• Frames • Ethernet, PPP, Switch, Bridge
1. Physical	• Physical structure • Coax, Fiber, Wireless, Hubs, Repeaters

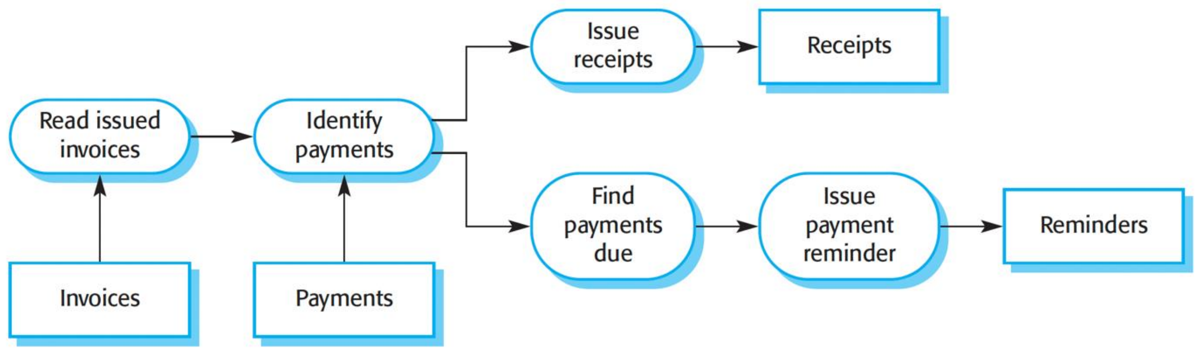
Example

- Operating Systems
    - Clear privilege boundaries
    - Stable interfaces (syscalls)
    - User code cannot directly access hardware -- protection, abstraction, portability

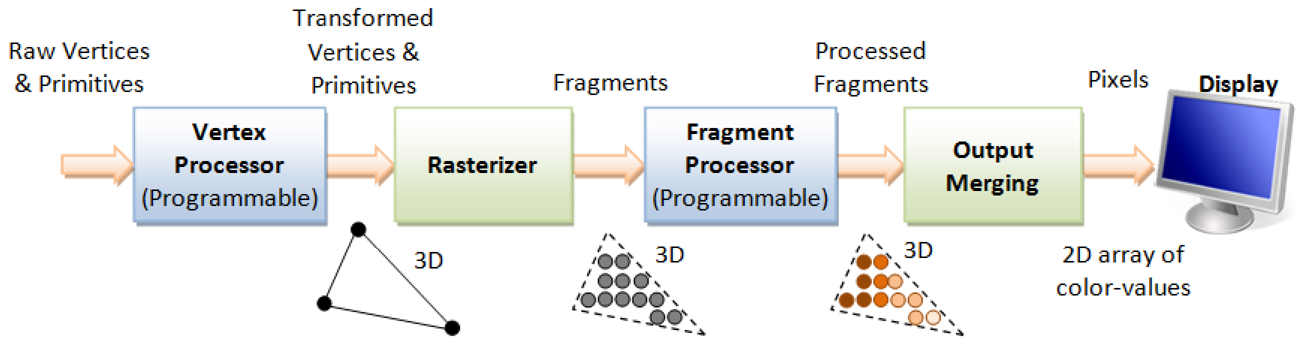


Pipe and Filter

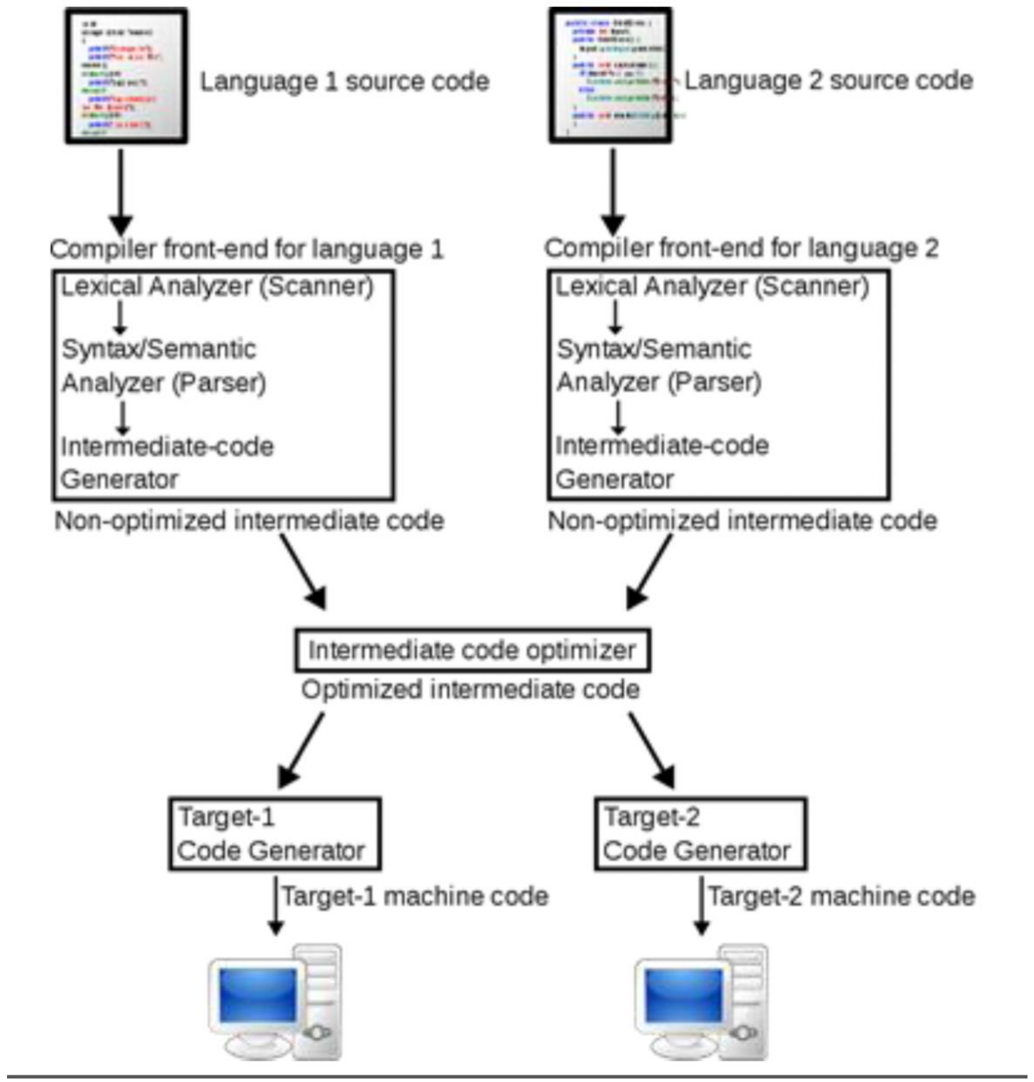
- Transformation components process inputs to produce outputs
  - Subsystems coupled via data exchange
  - Good match for data flow models (sequential/parallel)
  - May be dynamically assembled
  - Limited user interaction
- Applications:
  - Compilers
  - Graphics shaders
  - Signal processing
- Caveats:
  - Awkward to handle events (interactive systems)
  - Rate mismatches if branches merge



Examples

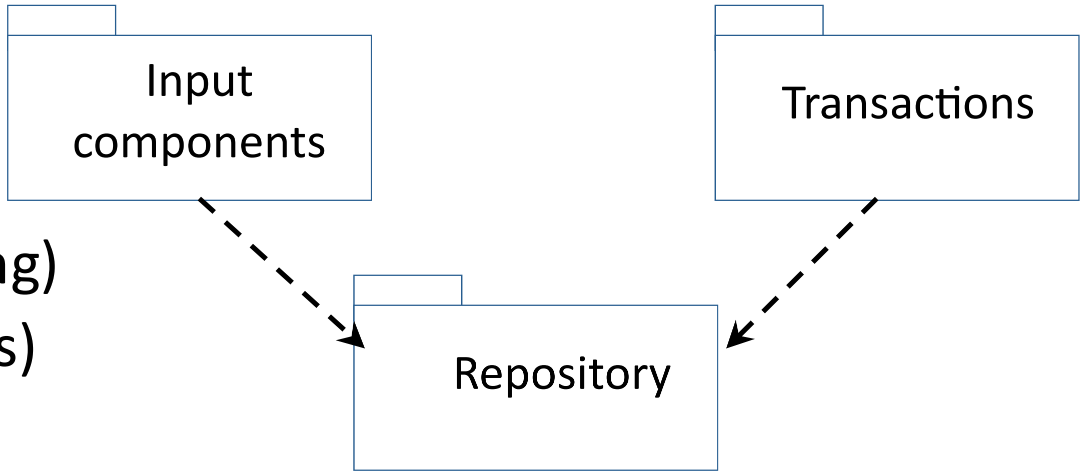

Raw Vertices & Primitives
Transformed Vertices & Primitives
Fragments
Processed Fragments
Pixels
Display
Vertex Processor (Programmable)
Rasterizer
Fragment Processor (Programmable)
Output Merging
3D
3D
3D
2D array of color-values

3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (*x*, *y*, *z*) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.


Language 1 source code
Language 2 source code
Compiler front-end for language 1
Compiler front-end for language 2
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Non-optimized intermediate code
Intermediate code optimizer
Optimized intermediate code
Target-1 Code Generator
Target-2 Code Generator
Target-1 machine code
Target-2 machine code

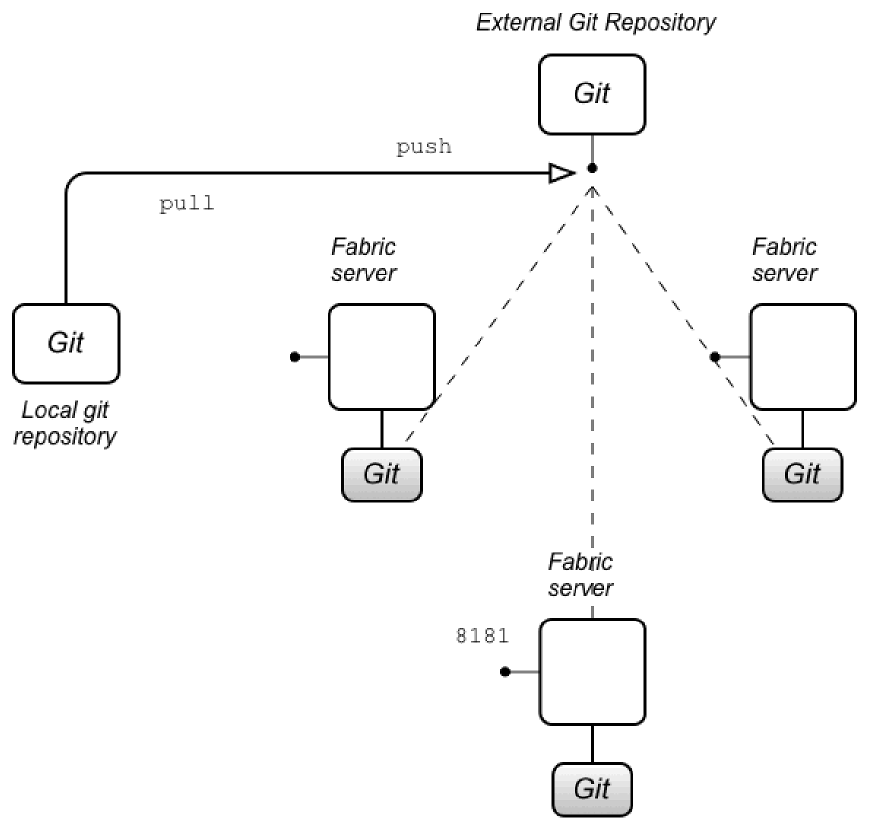
Repository

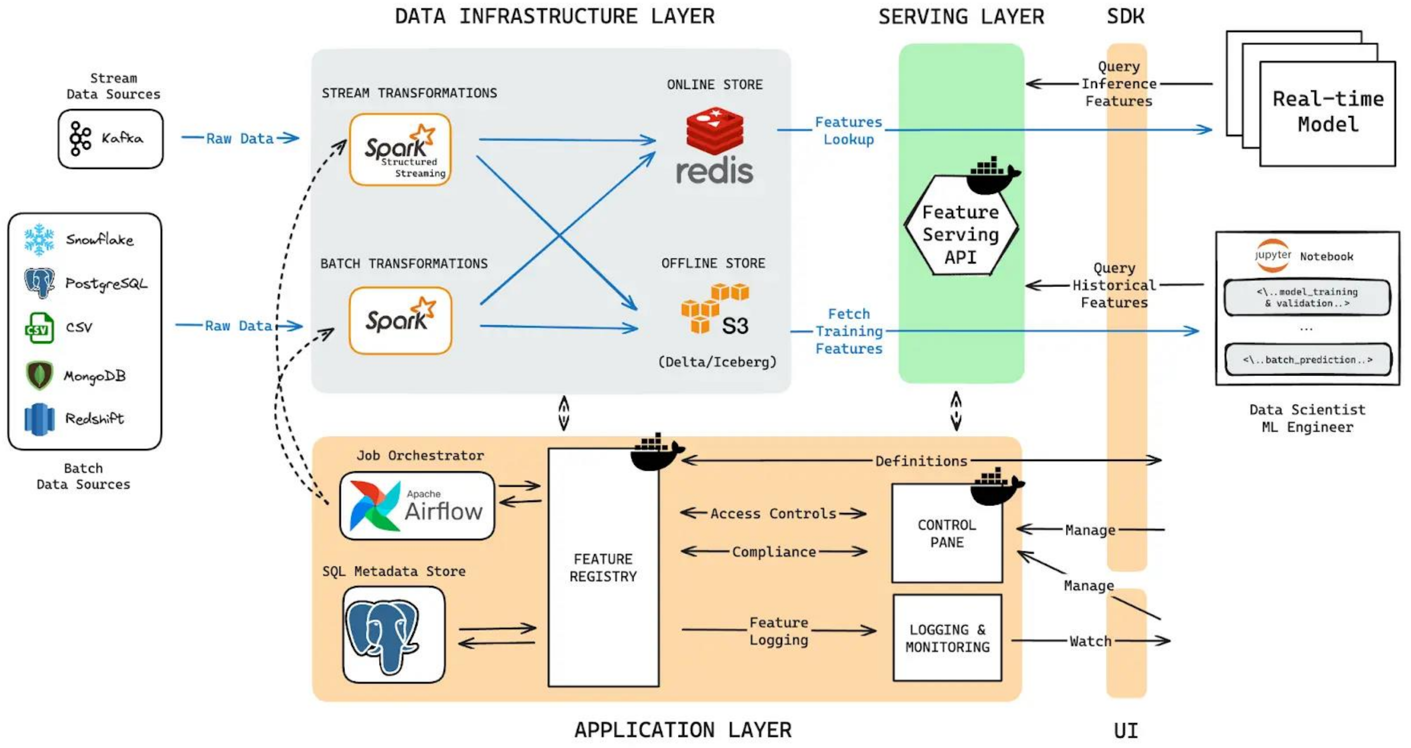
- Couple subsystems via shared data
  - Repository may need to support atomic transactions
- Advantages:
  - Components are independent (low coupling)
  - Centralized state storage (good for backups)
  - Changes propagated easily
- Dangers:
  - Bottleneck / single point of failure
  - Shared data abstractions
  - Replication can be difficult



Example

- Version Control Systems
- Feature Stores in ML



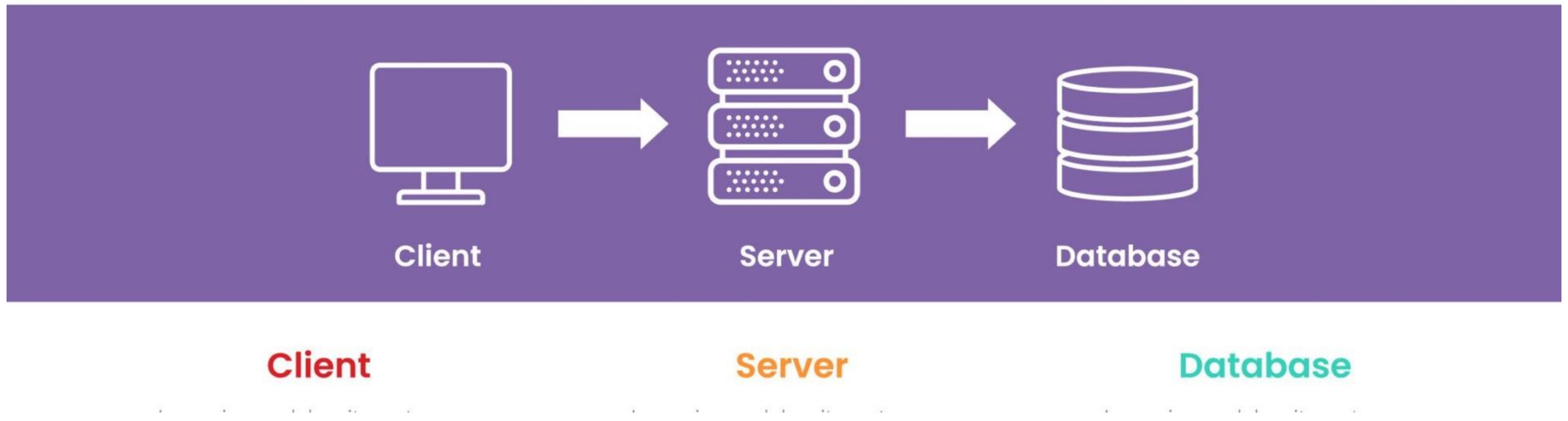


Poll: Consider a real-time data processing system continuously collecting and analyzing log files from multiple distributed servers. The system should filter, aggregate, and store logs while allowing administrators to query historical data efficiently. Which architectural pattern would be most appropriate, and why?

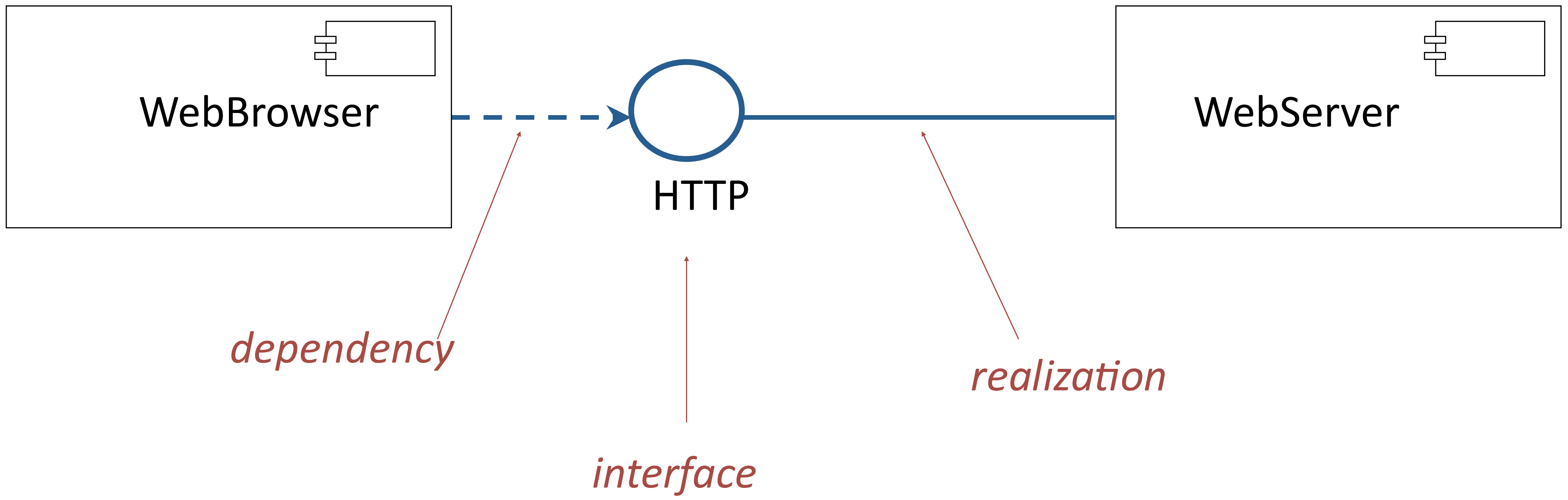
PollEv.com/cs5150sp26

Three tier architecture

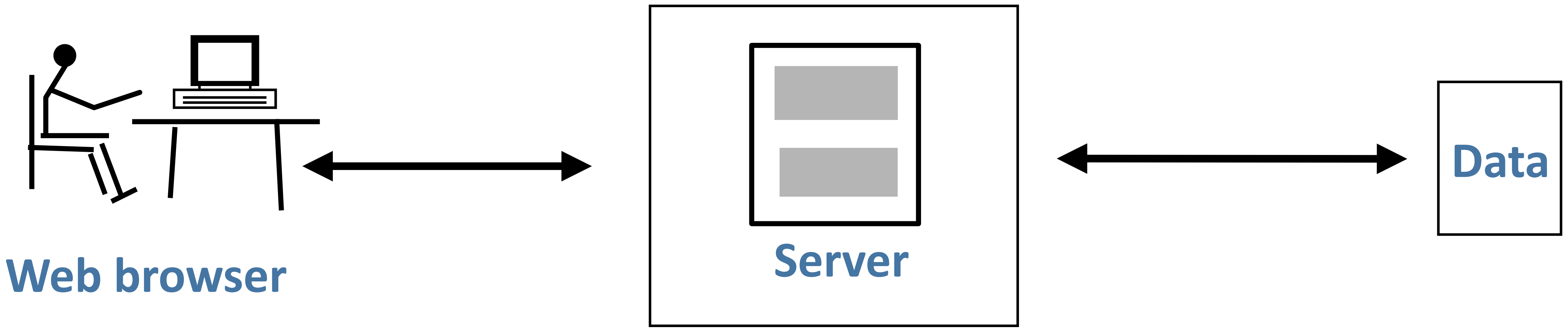
- Extension of client/server model
- Commonly used for small-medium websites
  - Classic example: LAMP stack for web applications (Linux, Apache, MySQL, PHP/Python)



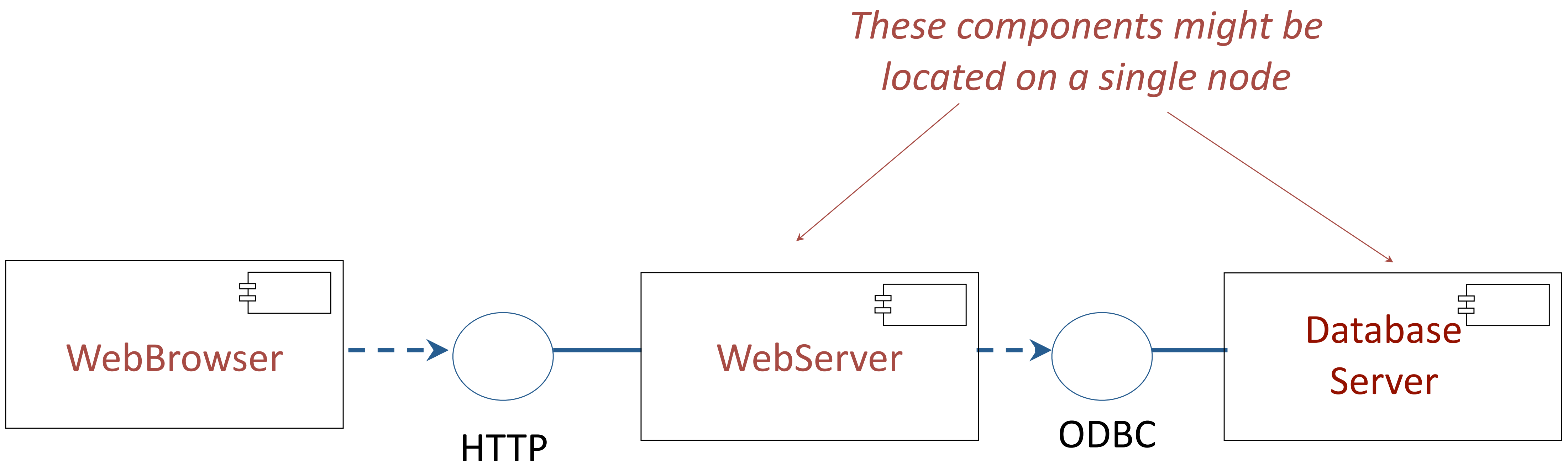
Basic website (client/server)



Extension: data store



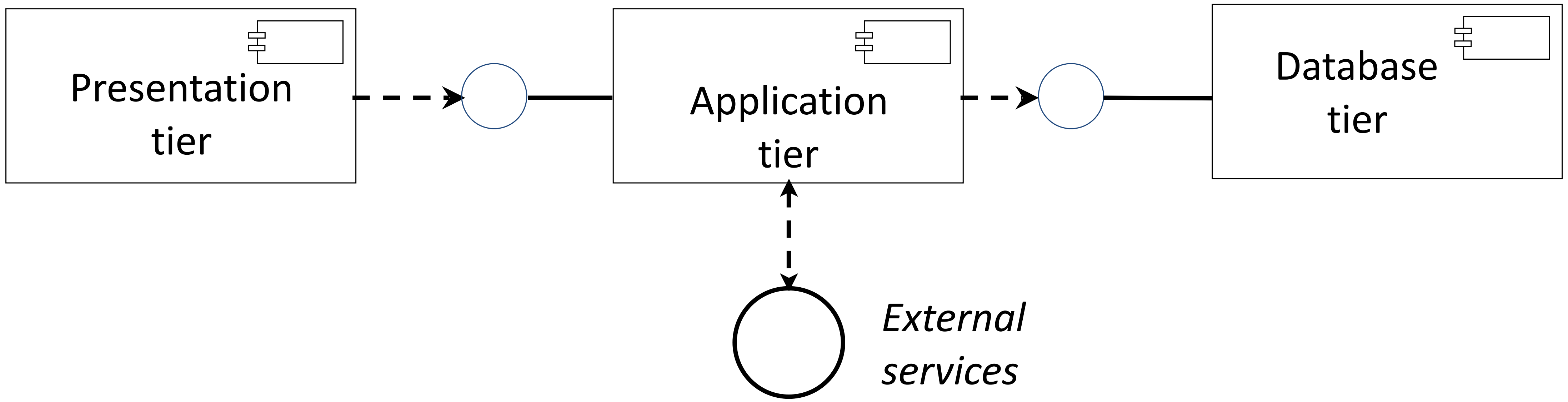
Component diagram



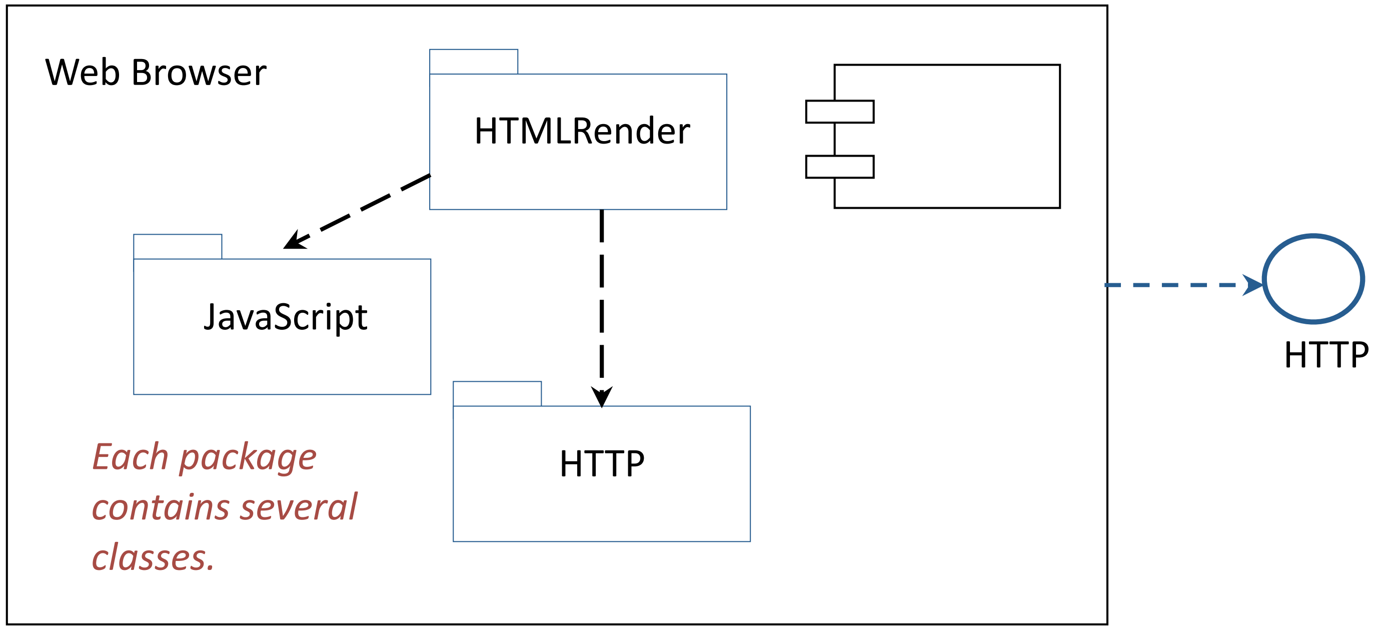
Significance of components (replaceable binary elements):

- Any web browser can access the website
- Database can be replaced by another that supports the same interface

Three tier architectural style



Presentation tier complexity



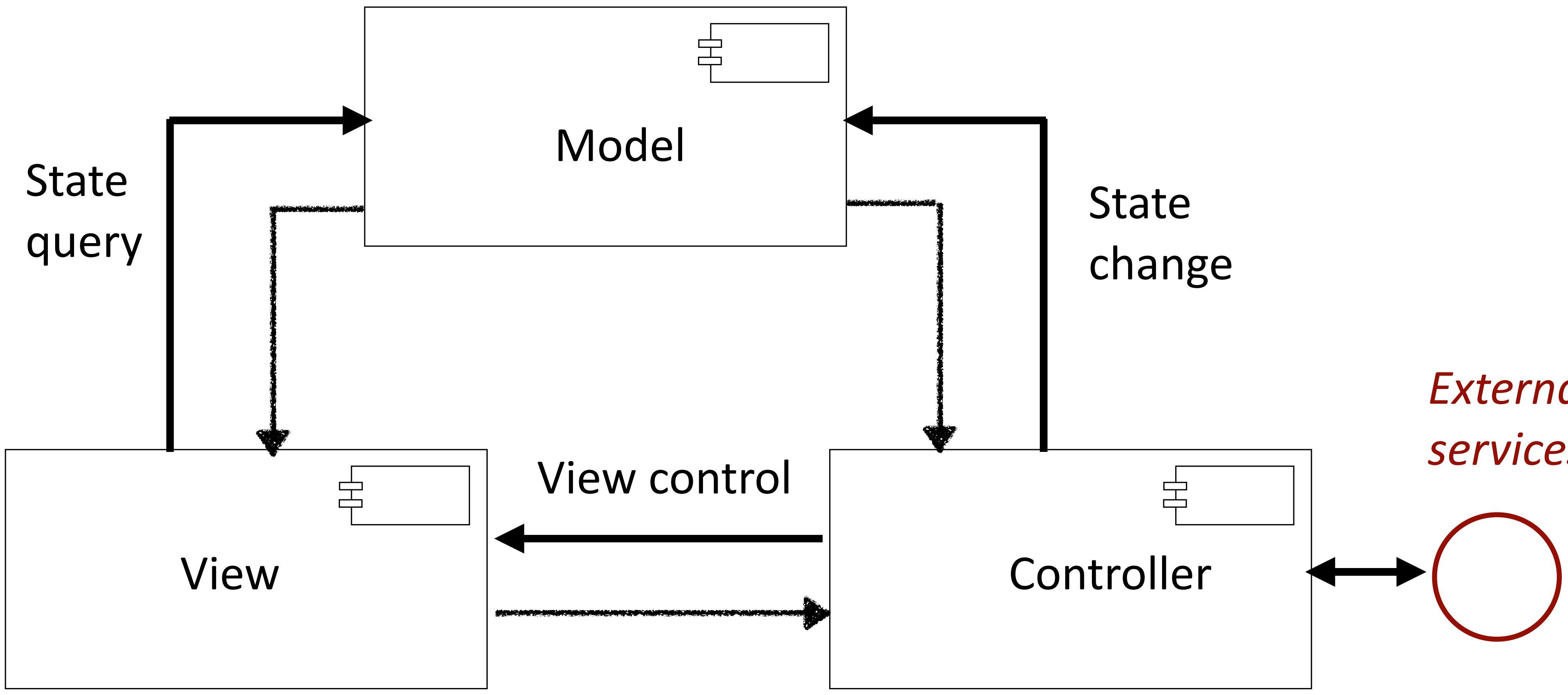
Presentation tier may house internal complexity, but as long as it supports the same interface, it is still a binary-replaceable component

Model-View-Controller

- Beware: many variations
  - Some are architectural styles: system-level responsibilities partitioned into different components
    - Example: **Django**/**Spring Framework/ASP.NET** for building web apps
  - Some are program design patterns: functionality divided between different classes
    - Focus on reusable controls
    - Example: **Swing widgets/React**
    - Variation on which logic is widget-level vs. form-level (MVC vs. MVP)
    - Variation on which classes communicate directly (MVC vs. MVA)
    - Variations in model storage (domain objects, DB record sets, immutable store)

Read more: https://martinfowler.com/eaaDev/uiArchs.html

Component diagram


Model
State query
State change
External services
View control
View
Controller