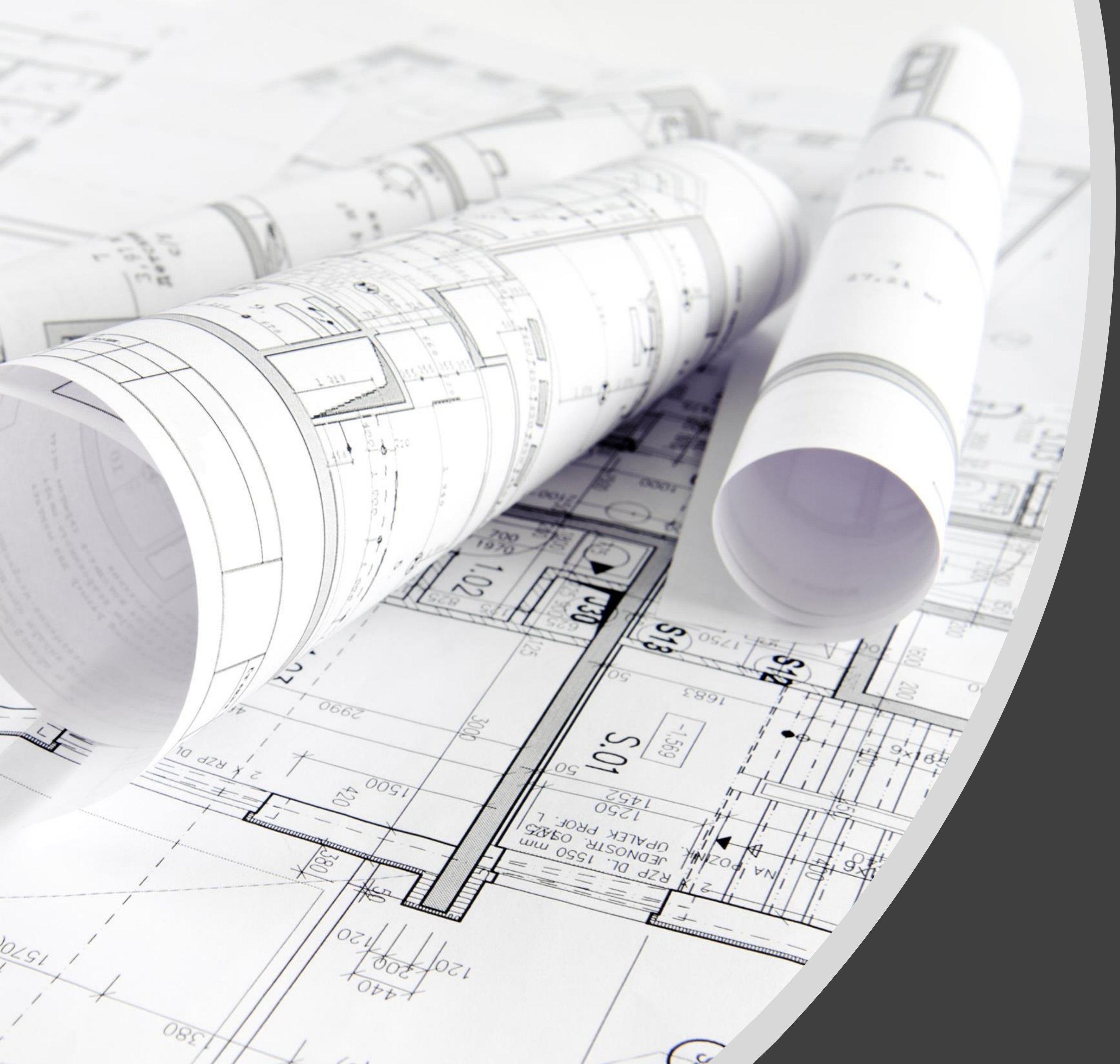


Lecture 7: Architecture

CS 5150, Spring 2025

Lecture goals

- Understand the importance and need for software architectures
- Visualize structural models with deployment and interface diagrams
- Identify common architectural styles

Administrative Reminders

- Project Plan is due today (Feb 11, 11.59 PM)
- Meet with your client for the first sprint – your grade depends on it!
- Assignment A2: Some issues with different OSes

Poll: How are you feeling about the project?

PollEv.com/cs5150sp25

System design

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!

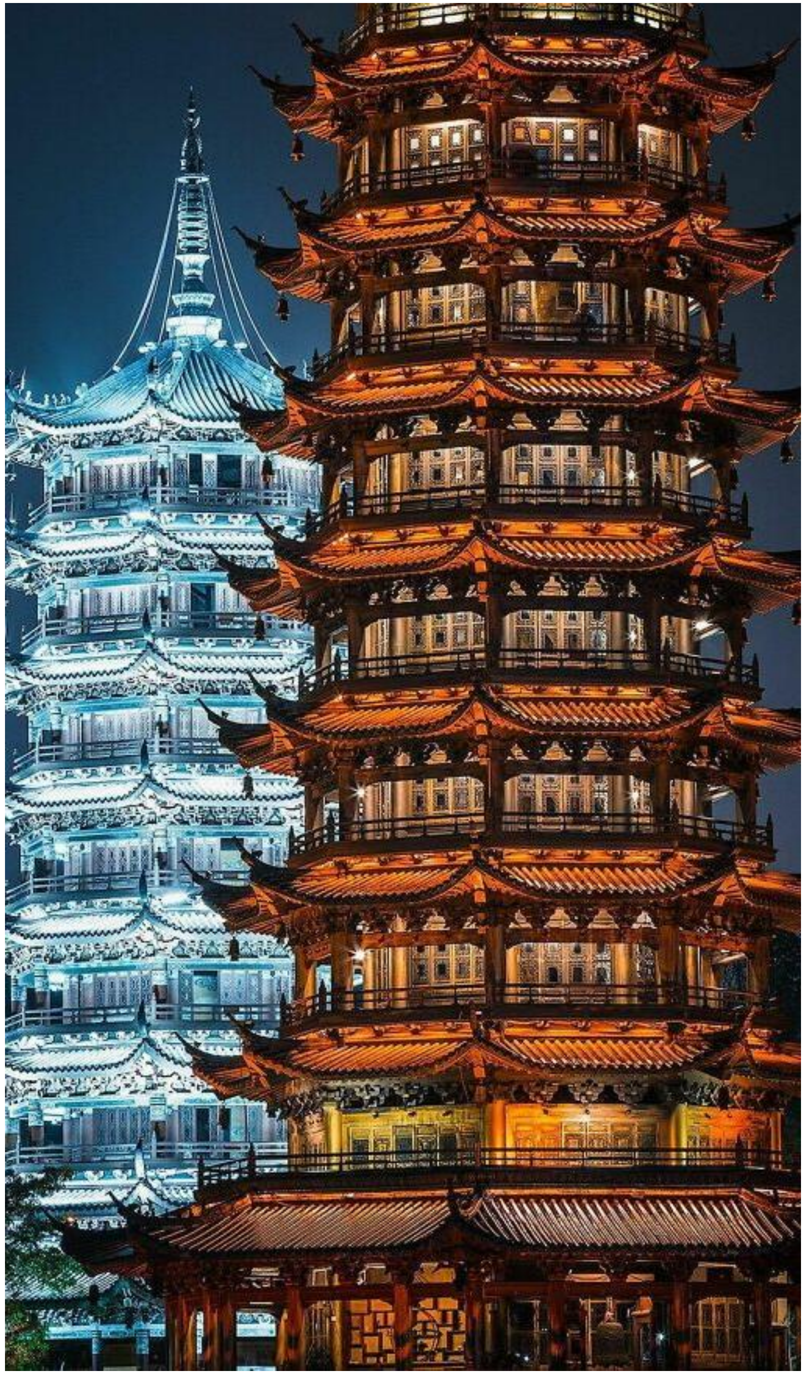
Software Architecture: Other Definitions

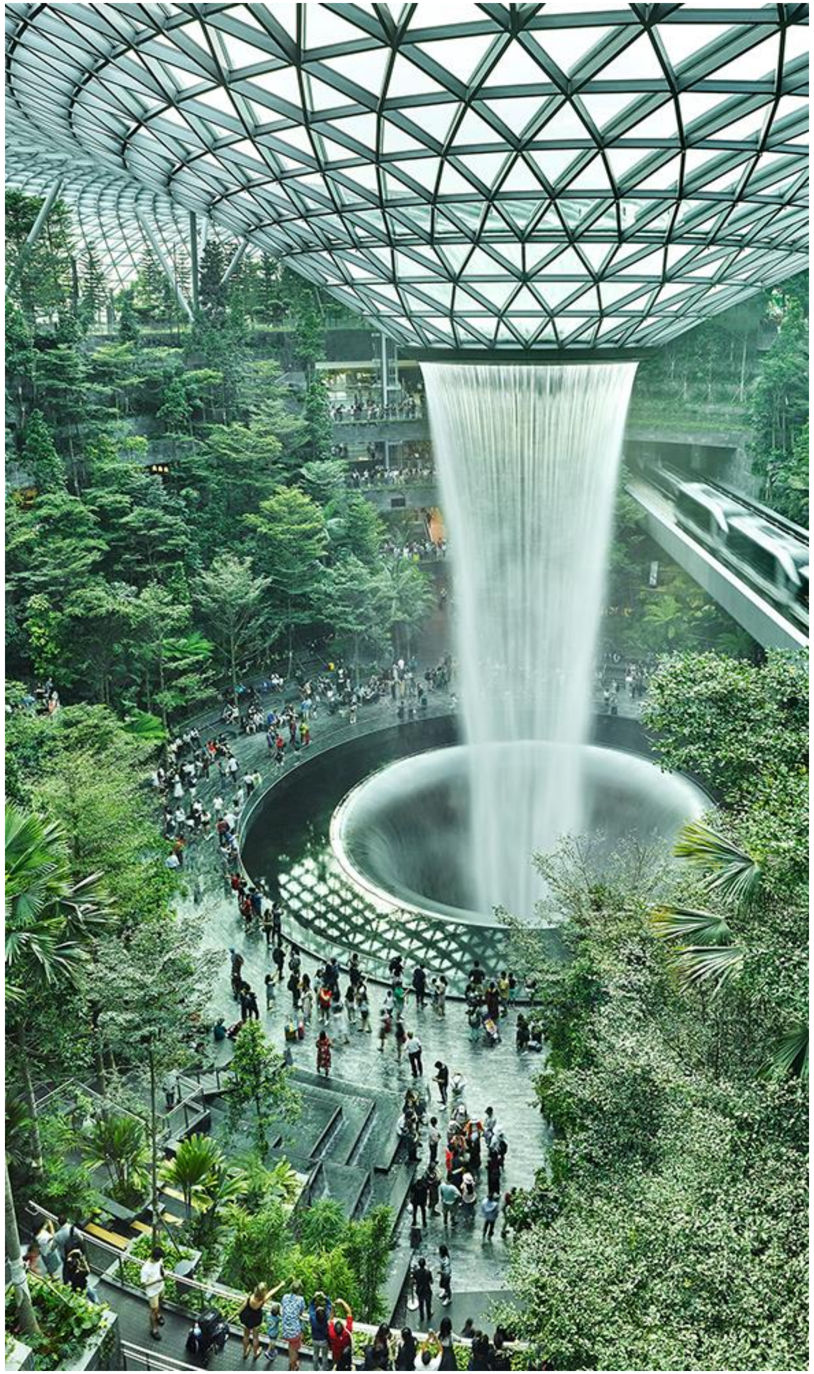
- Brooks: Conceptual integrity is key to usability and maintainability; architect maintains conceptual integrity
- Johnson: *The shared understanding that expert developers have of the system / The decisions you wish you could get right early in a project*

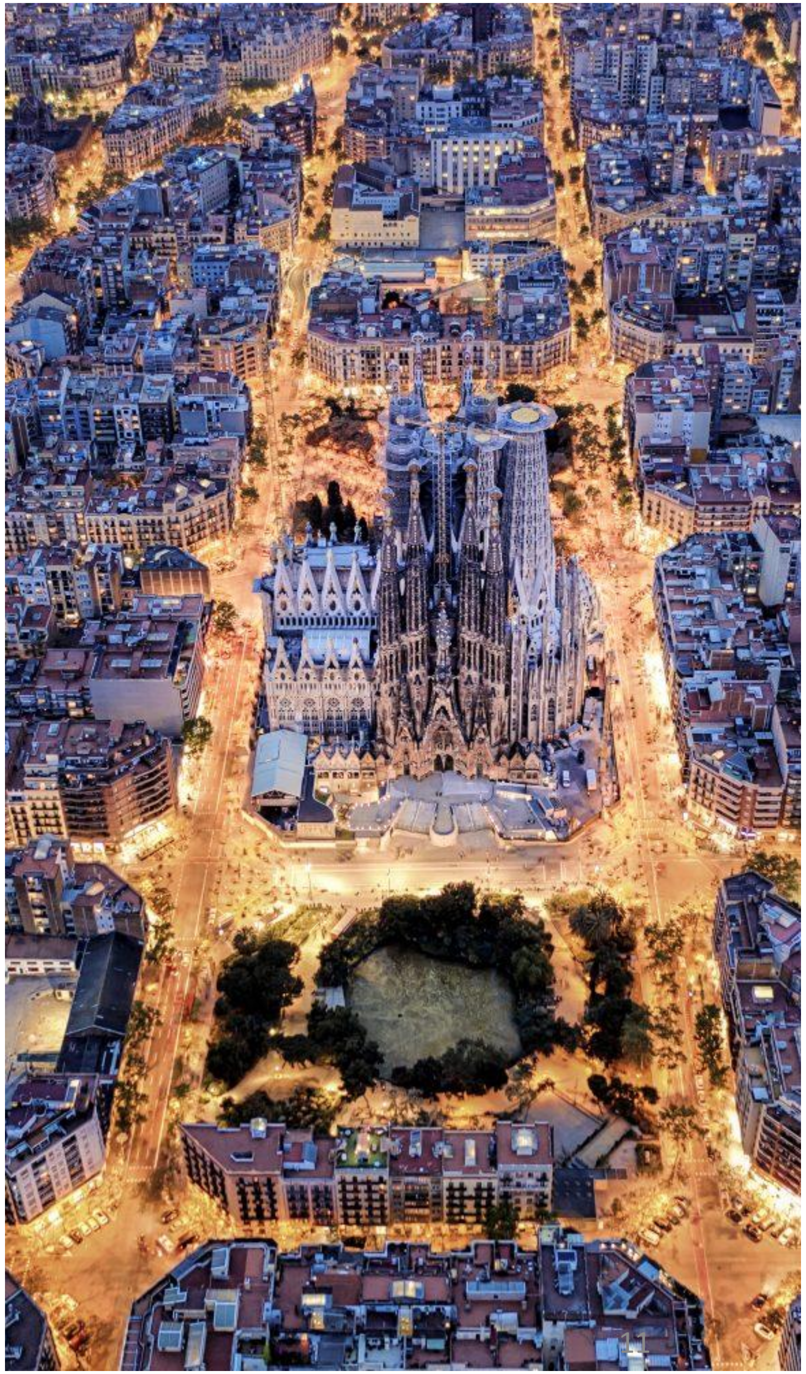
- Sommerville: Dominant influence on non-functional system characteristics
- "Highest level" organization of system

Why document Architecture?

- Blueprint for the system
  - Artifact for early analysis/communication
  - Primary carrier of quality attributes: performance, robustness, reusability
  - Key to post-deployment maintenance
- Documentation speaks for the architect, today and 20 years from today
  - As long as the system is built, maintained, and evolved according to its documented architecture







Every software system has an architecture, whether you know it or not!
If you don’t consciously elaborate the architecture, it will evolve by itself!

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

Example

- Requirements:
  - Drone should hover stably
- Architecture
  - Sensing → navigation → control → actuation
  - Radio input
- Program design
  - PID controller, low-pass filter
  - Gain registry

- Code

  -
    ```
    def lpf(x1, y0, a):
        """exp filter w/ smoothing factor a"""
        return a*x1 + (1-a)*y0
    ```

Architectural considerations

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

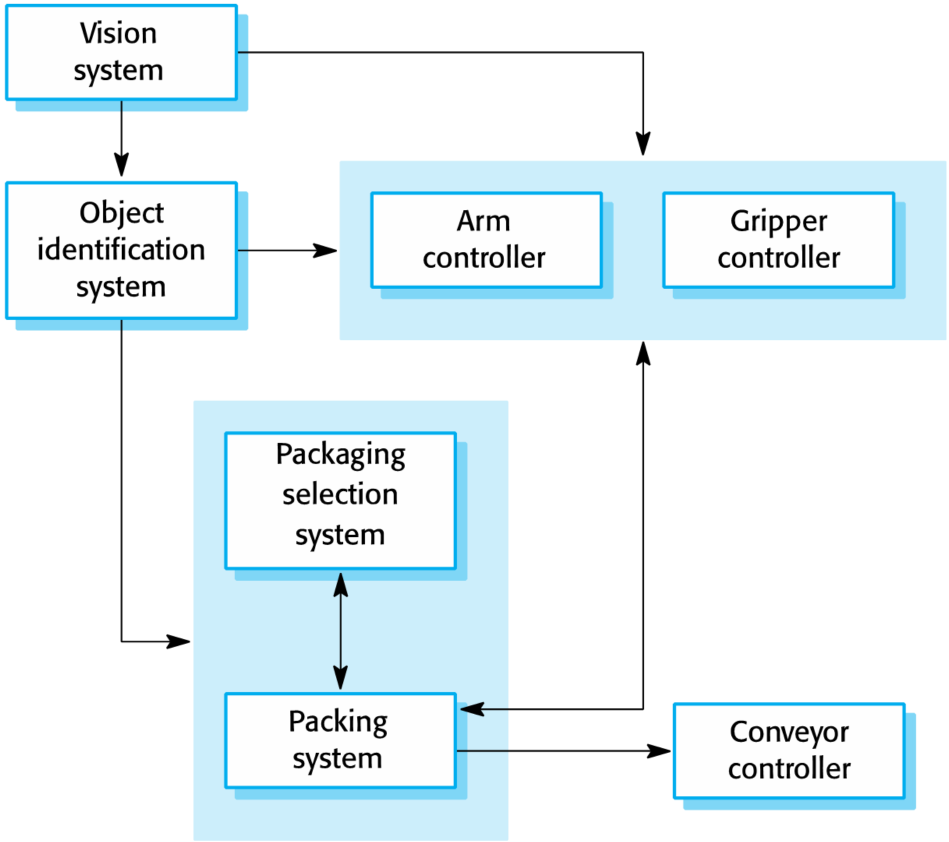
Architectual models

- Diagram **and** supporting specification
  - Be *specific* with notation
- Multiple perspectives
  - Conceptual
  - Static (subsystems)
  - Dynamic (data flow)
  - Physical (deployment)

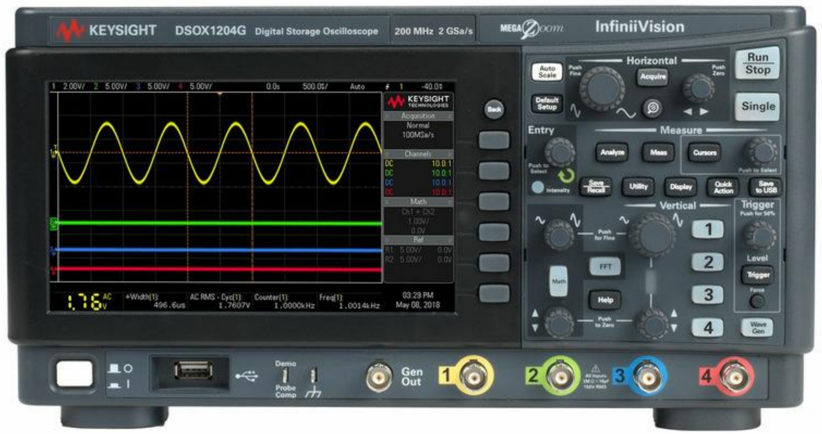
- Appropriate level of detail
  - A single diagram should fit cleanly on one page

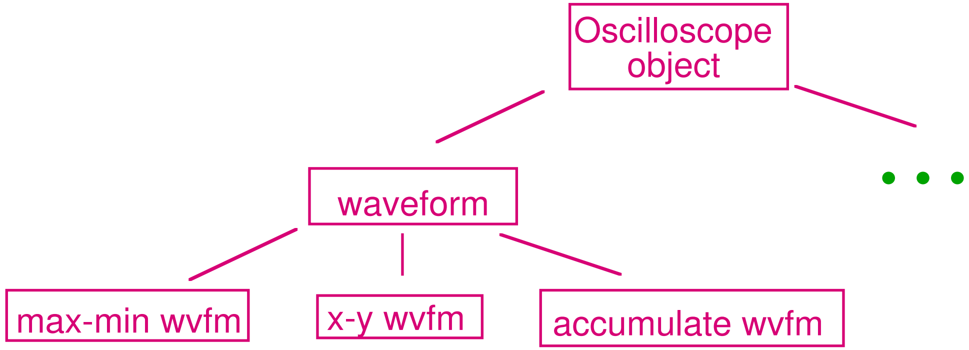
- Distinct from program models
  - Inheritance diagrams don’t show interactions

Examples



Packing Robot control system





Oscilloscope

Sommerville, *Software Engineering*

Garlan & Shaw, "An Introduction to Software Architecture"

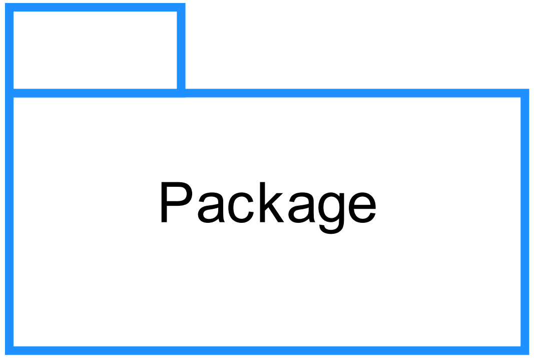
Subsystems

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

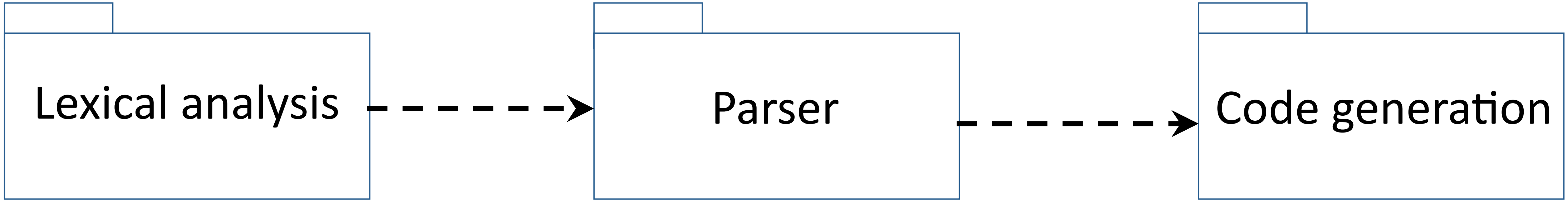
- Coupling: Dependencies *between* two subsystems
  - If coupling is high, can't change one without affecting the other
- Cohesion: Dependencies *within* a subsystem
  - High cohesion implies closely-related functionality

UML: Package

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

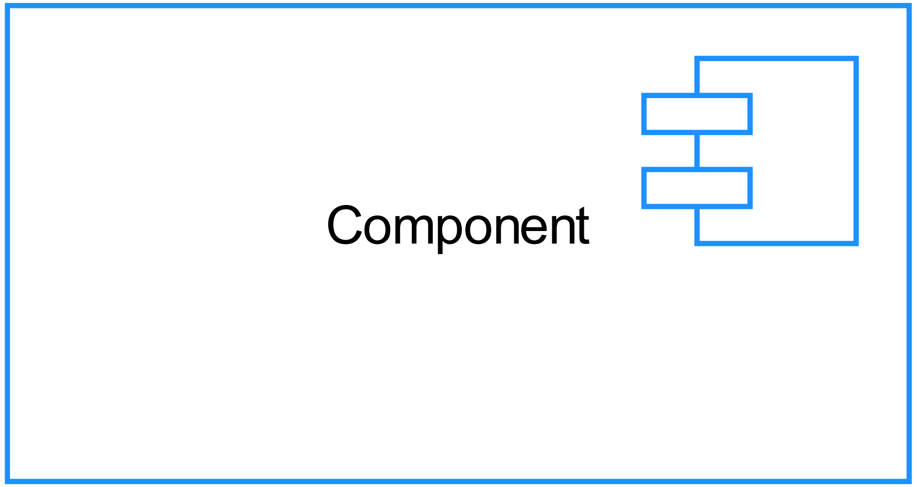


Example: conceptual diagram

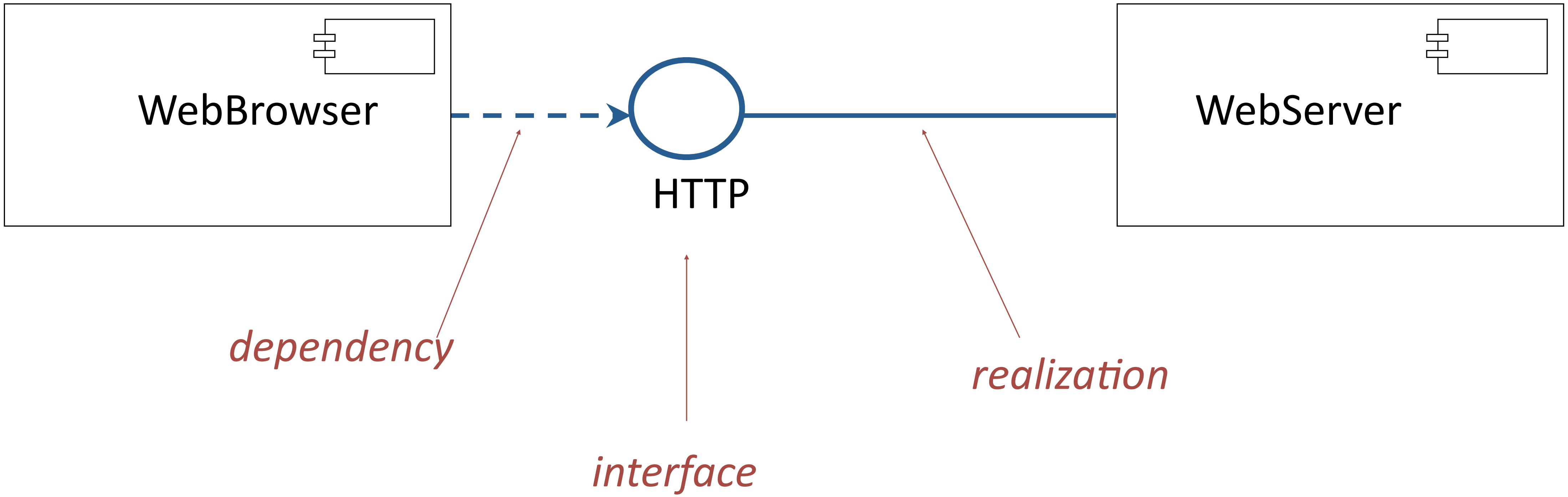


UML: Component

- 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

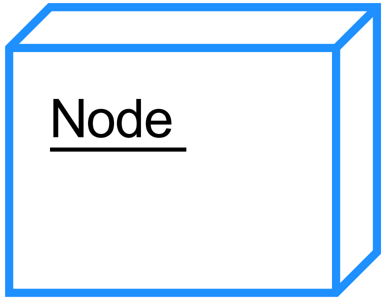


Example: interface diagram

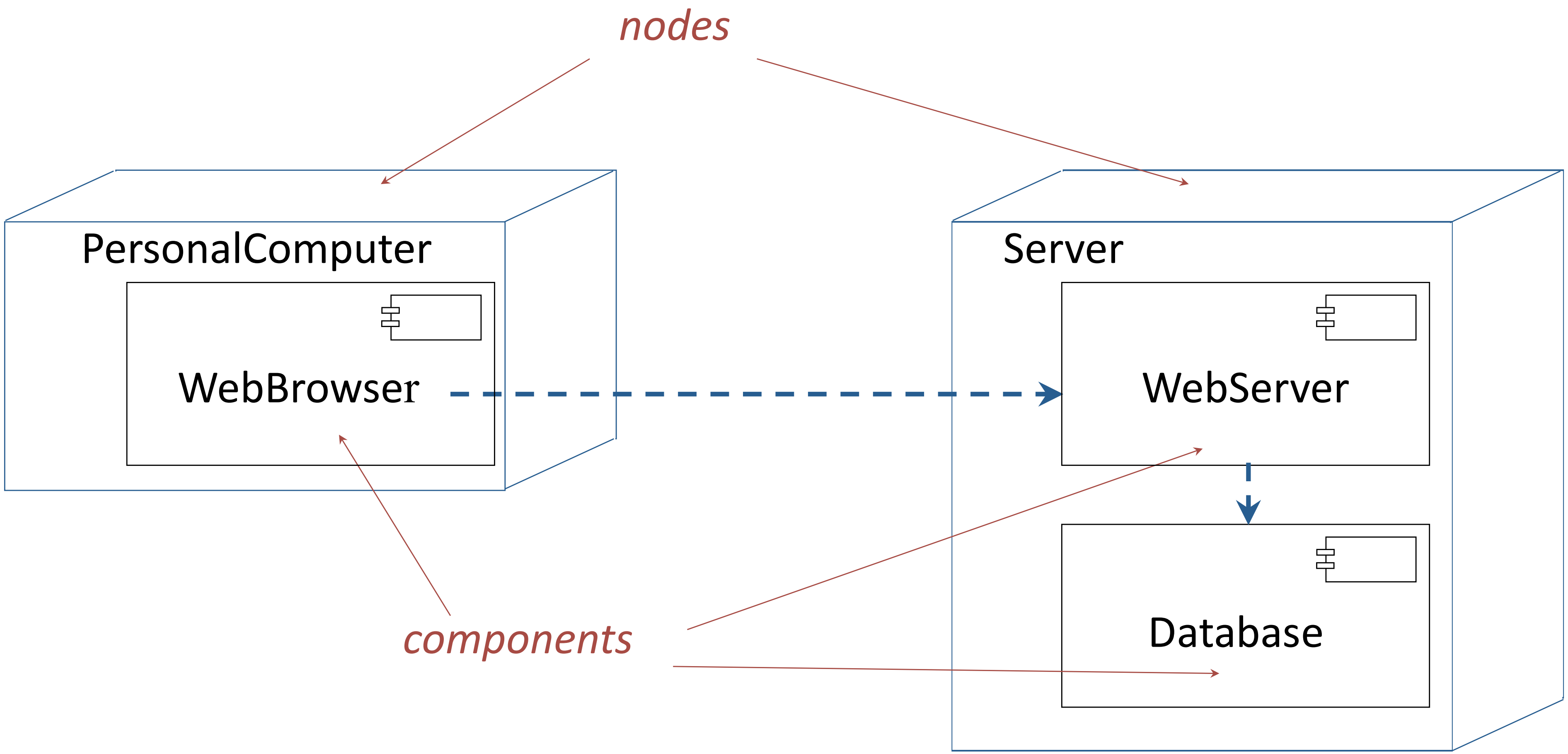


Node

- 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: A web server is designed to communicate with a database via a JDBC driver. Which architectural diagram is the best place to show this constraint?

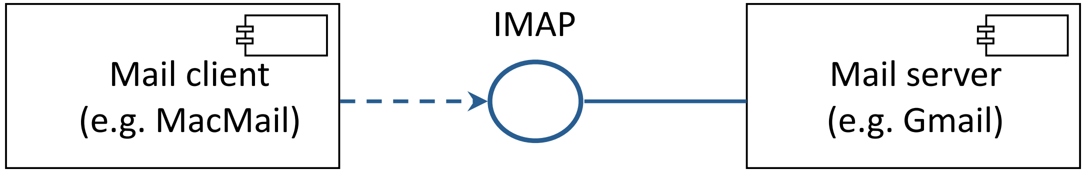
PollEv.com/cs5150sp25

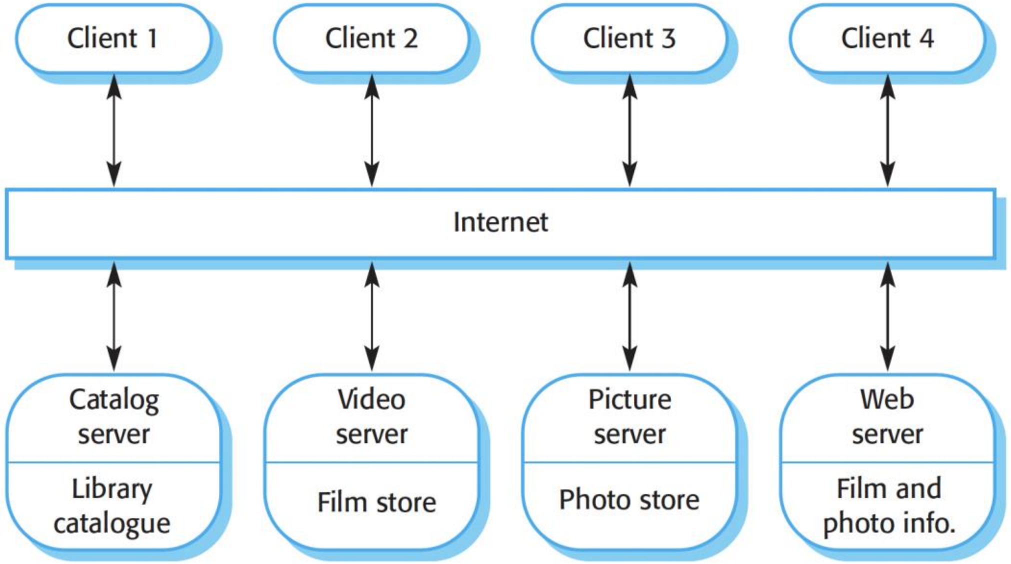
Architectural styles

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

Client/Server

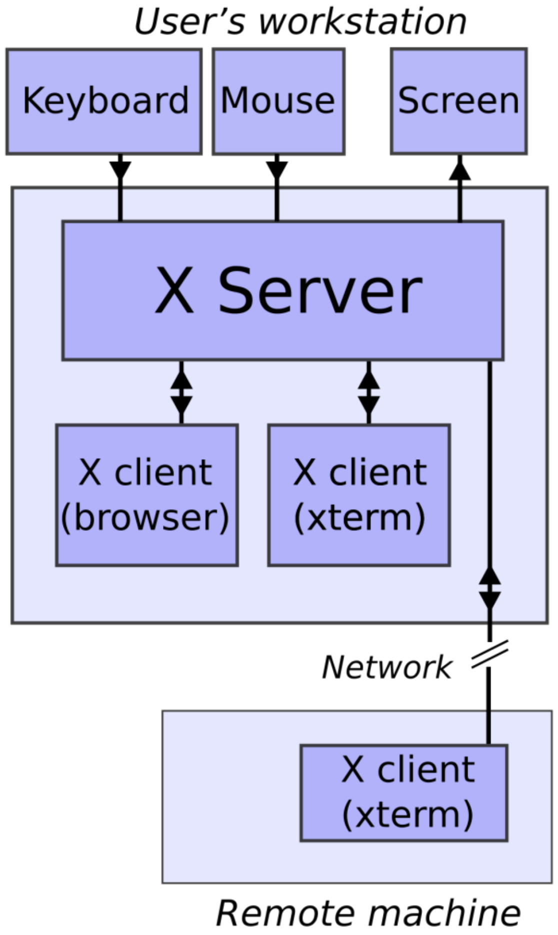
- Control flow in client and server are independent
- Communication follows a protocol
- If protocol is fixed, either side can be replaced independently
- Peer-to-peer: same component can act as both client and server





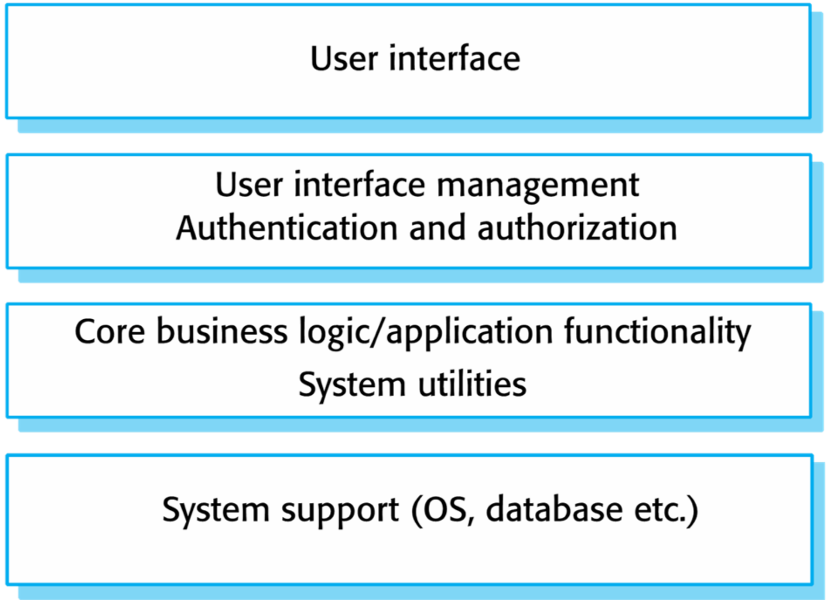
Example: X Window System (X11)

- 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

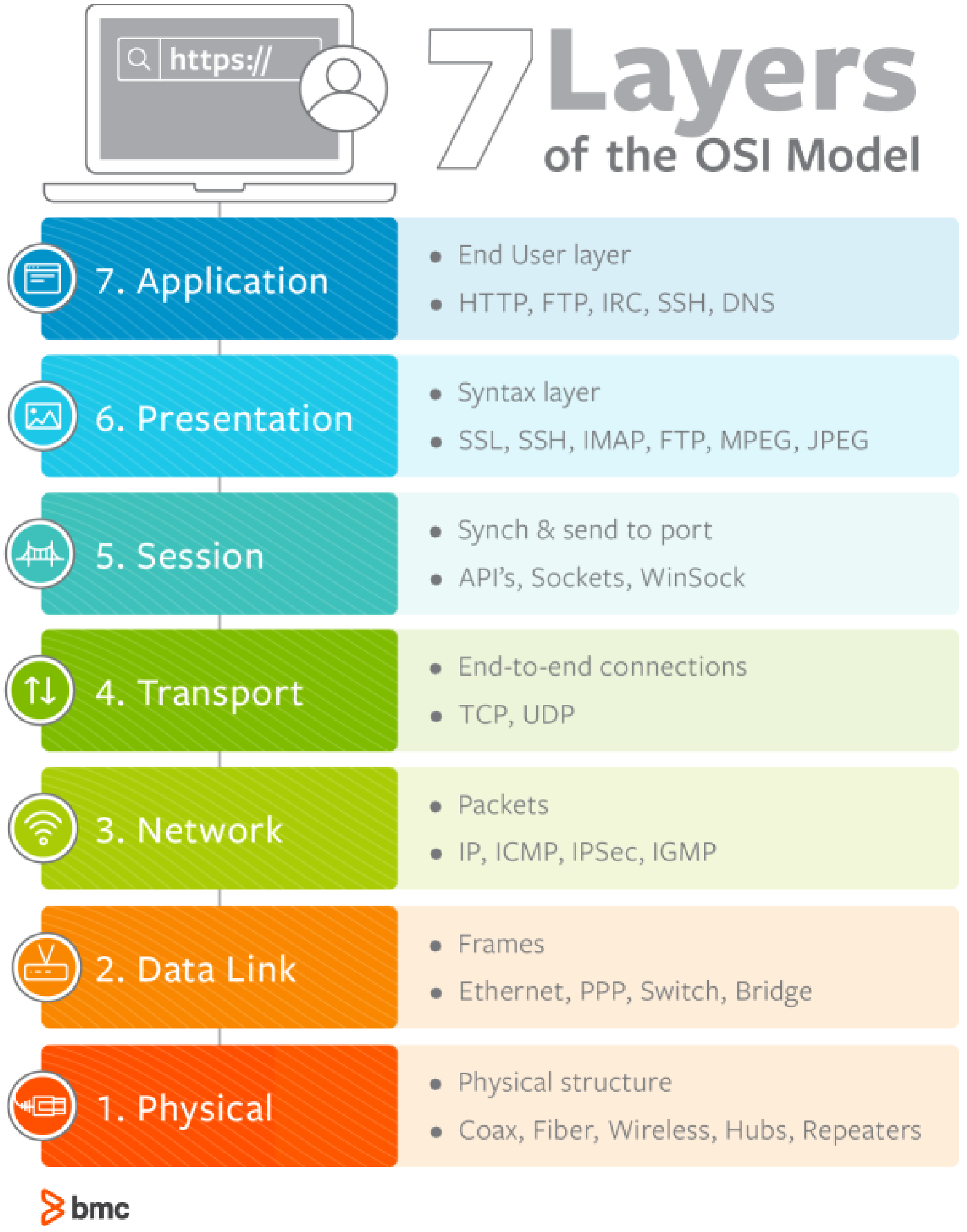
- Partition subsystems into stack of layers
  - Layer provides services to layer directly above
  - Layer relies on services to layer directly below
- Advantage: constrains coupling
- Danger: leaky abstractions
  - Clear separation is difficult
  - May need services of multiple lower layers
  - Performance


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 Reference Model
- Used for network protocols (TCP/IP)



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

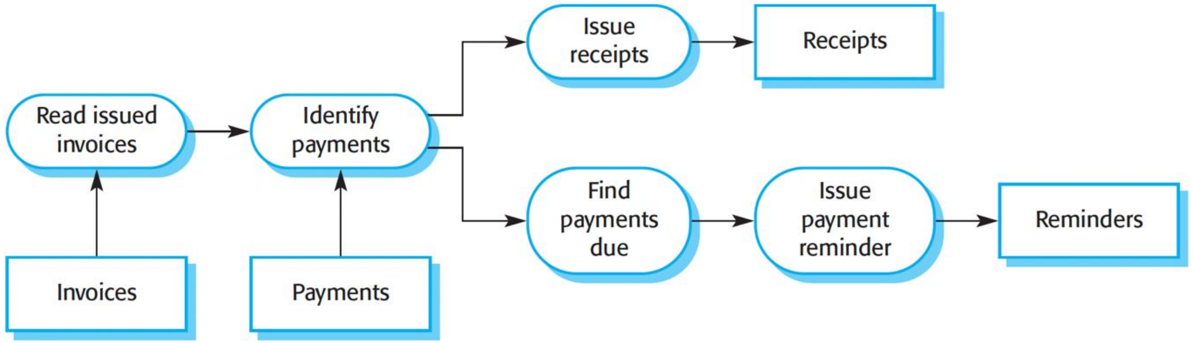
Examples

- OSI reference model architecture
  - See Sommerville, *Software Engineering, Tenth Edition*, Web sections: Reference Architectures

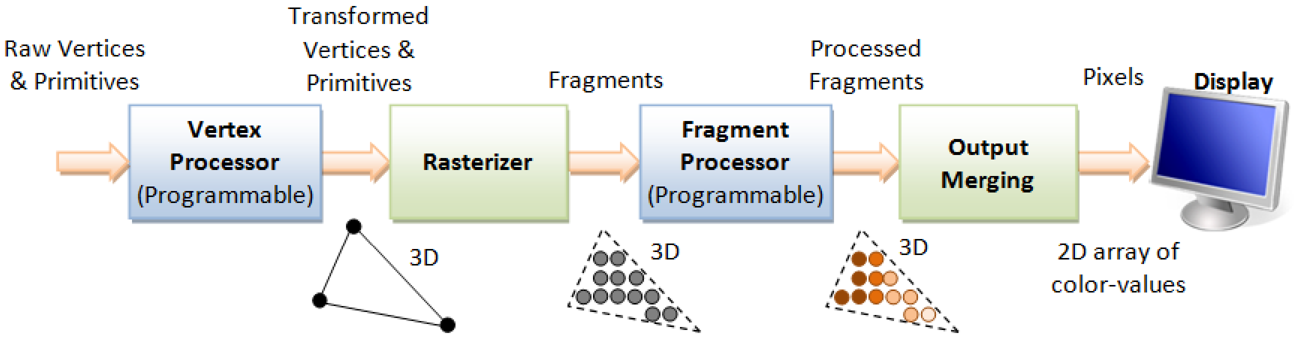
- Oscilloscope: Layered approach
  - See Garlan & Shaw, "An Introduction to Software Architecture"

Pipe and Filter

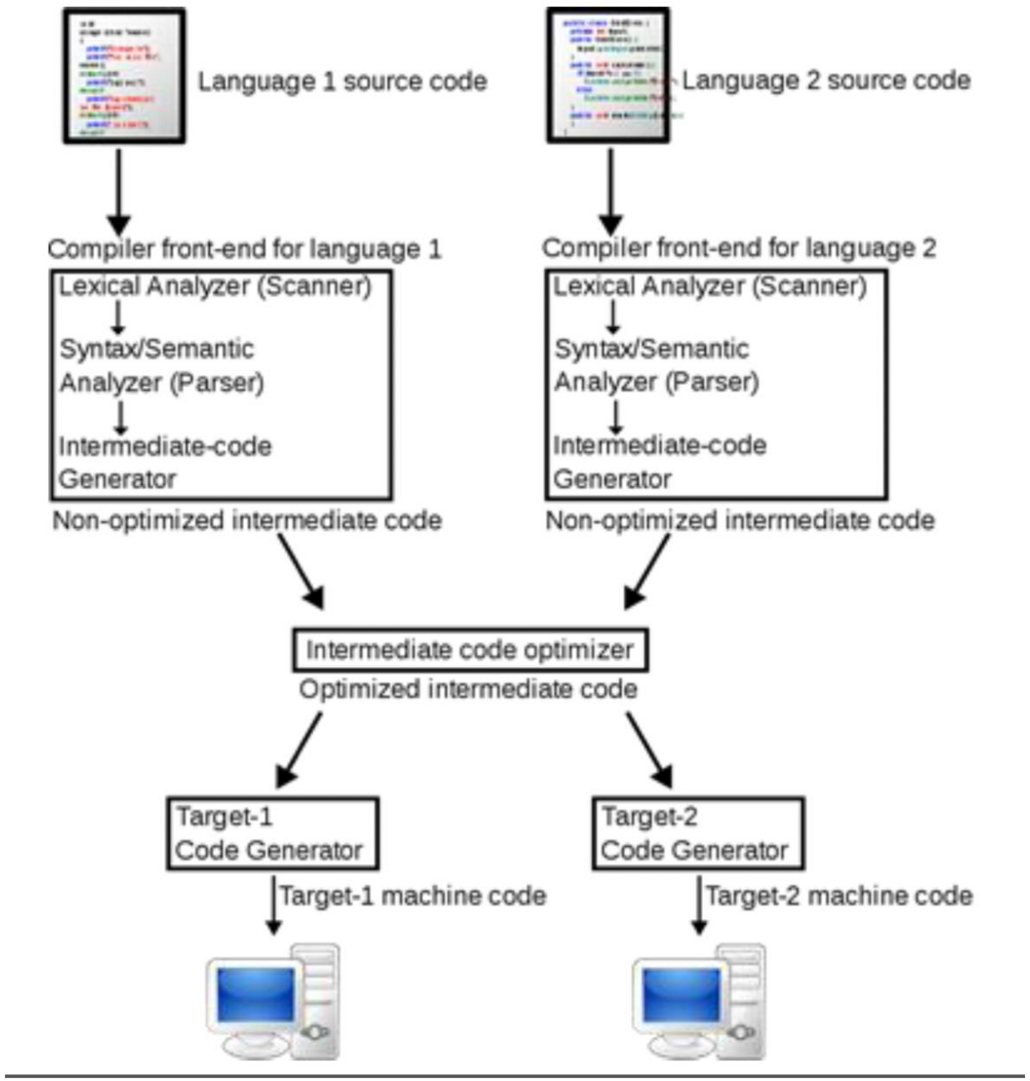
- Transformation components process inputs to produce outputs
  - Subsystems coupled via data exchange
  - Good match for data flow models
  - May be dynamically assembled
  - Limited user interaction
- Applications:
  - Compilers
  - Graphics shaders
  - Signal processing
- Caveats:
  - Awkward to handle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



Examples

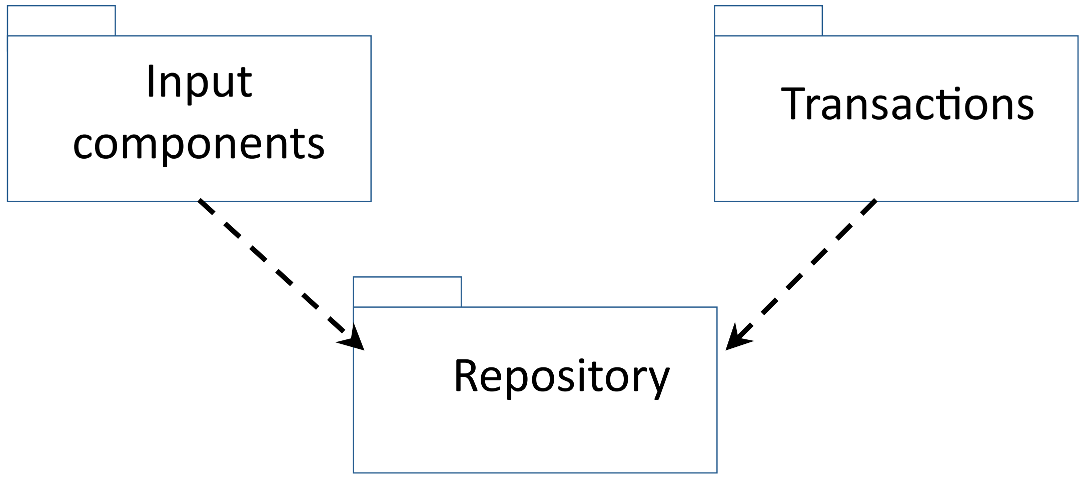

Raw Vertices & Primitives
Transformed Vertices & Primitives
Fragments
Processed Fragments
Pixels
Display
Vertex Processor (Programmable)
Rasterizer
Fragment Processor (Programmable)
Output Merging
3D
3D
3D
2D array of color-values

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
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Compiler front-end for language 2
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

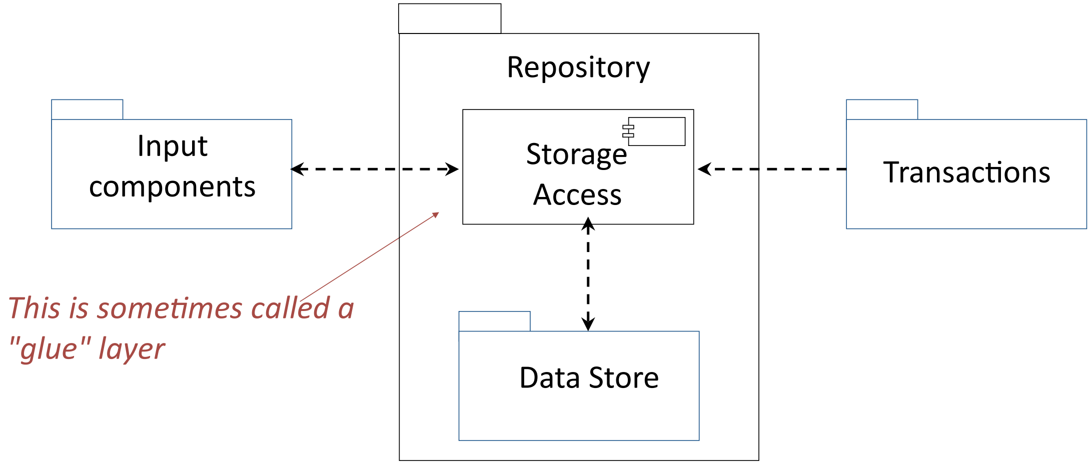
Repository

- 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

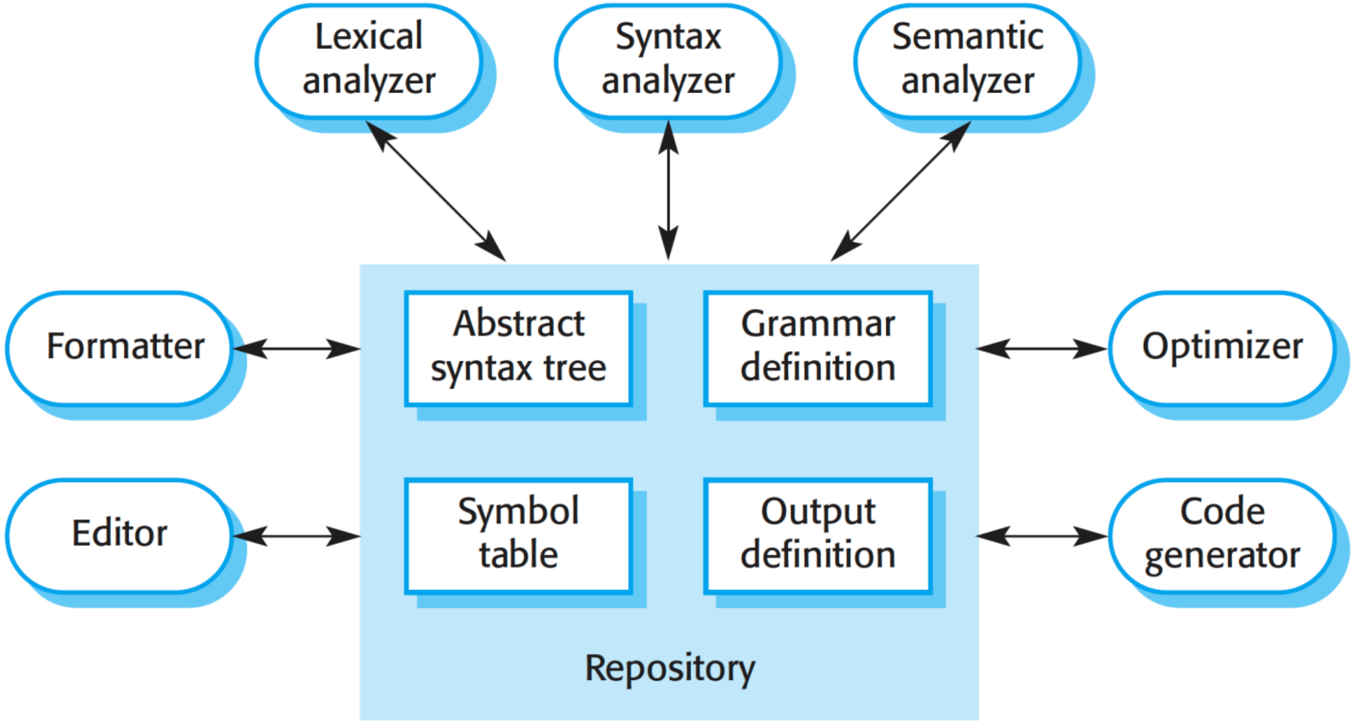


Flexibility through indirection

- Repository is highly coupled – difficult to change data store
- By defining higher-level storage access interface, data store is now lightly coupled



Example: Compilers (Language Processing System)


Lexical analyzer
Syntax analyzer
Semantic analyzer
Formatter
Abstract syntax tree
Grammar definition
Optimizer
Editor
Symbol table
Output definition
Code generator
Repository