Lecture 8: Architecture 3

CS 5150, Spring 2026

Administrative Stuff

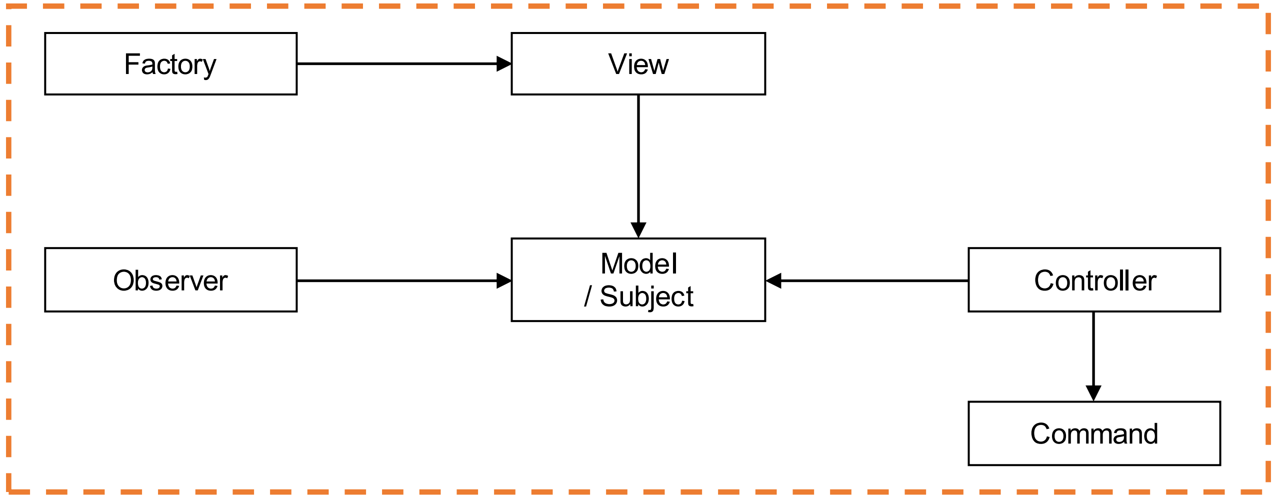
- Assignment 2 released – Due Feb 20, 11.59 PM
- Start working on your projects, meet with your clients
  - Each meeting with your client will be graded, so come prepared
  - How often to meet? Typically once a week
- At the end of each sprint:
  - Sprint report (Feb 26 – Sprint 1 Report)
  - Peer Review
- Midpoint presentation – details coming soon

Lecture goals

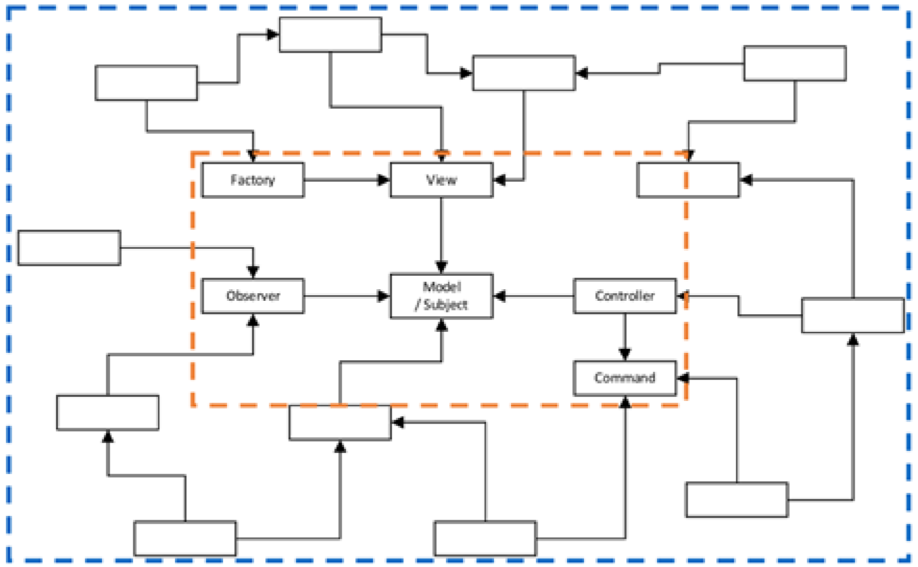
- Review architectures; publish-subscribe
- Encapsulate deployments using virtualization

Previously on 5150…

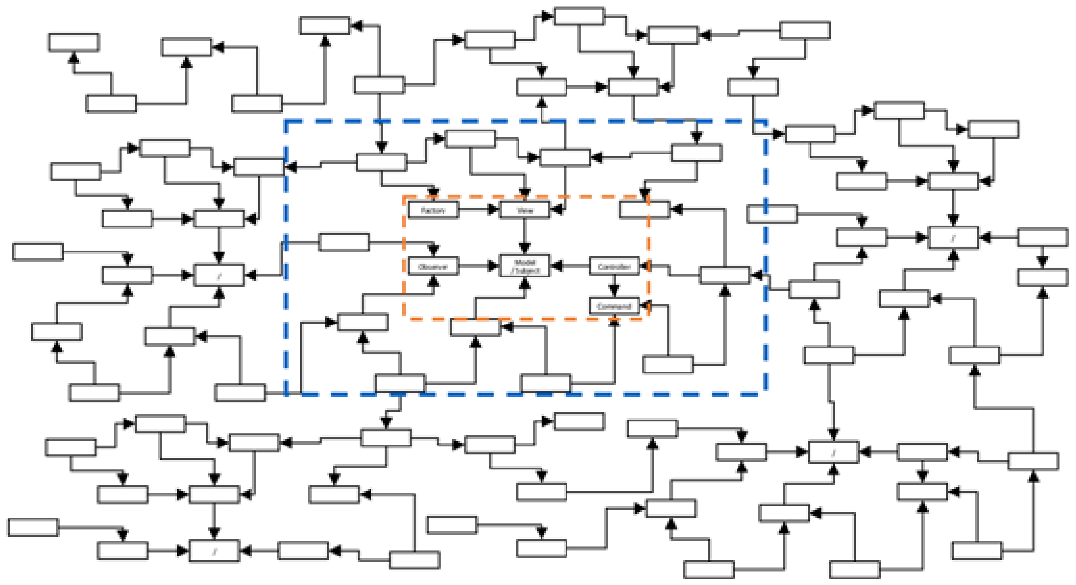
Design Patterns



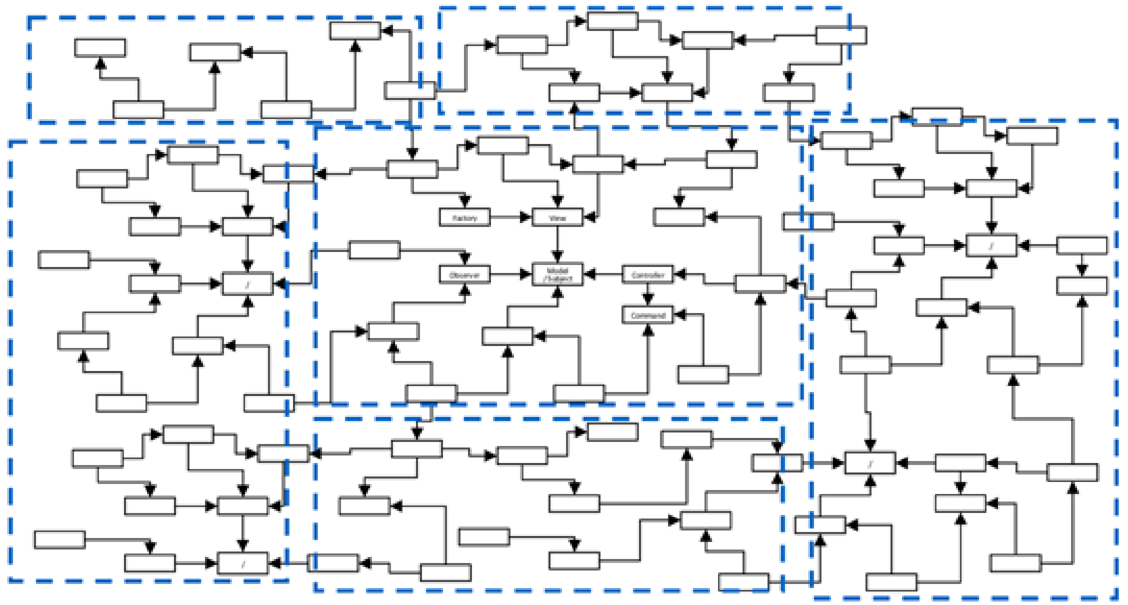
Design Patterns



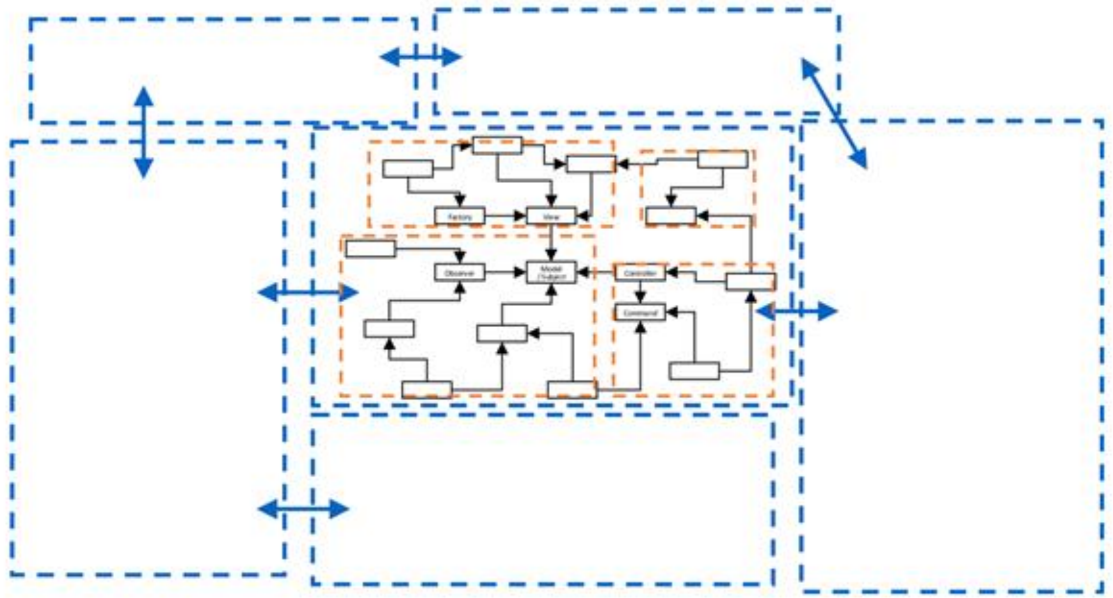
Design Patterns



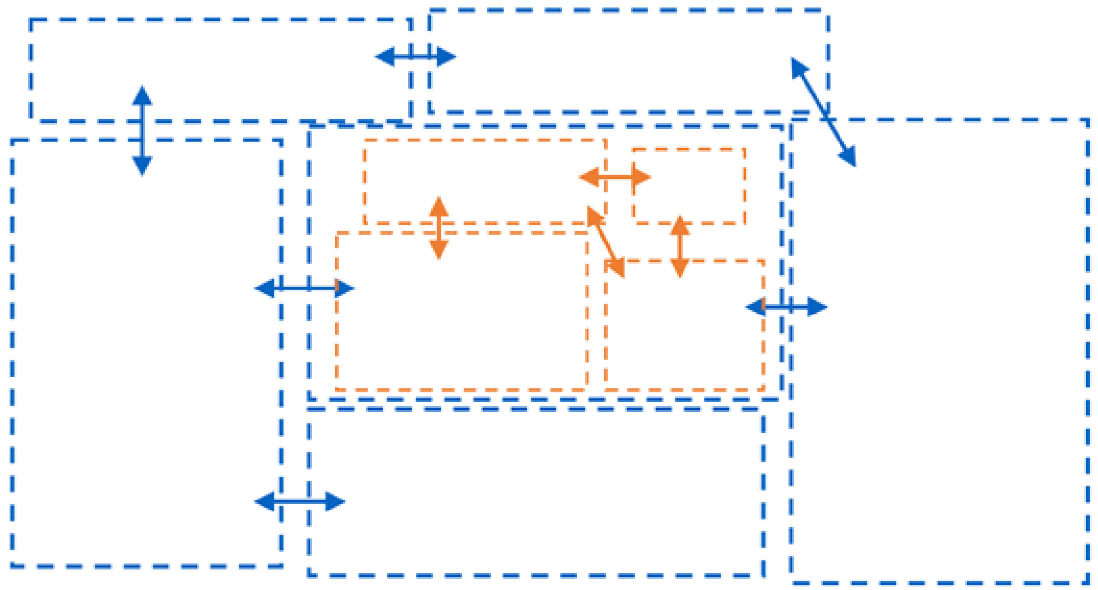
Architecture



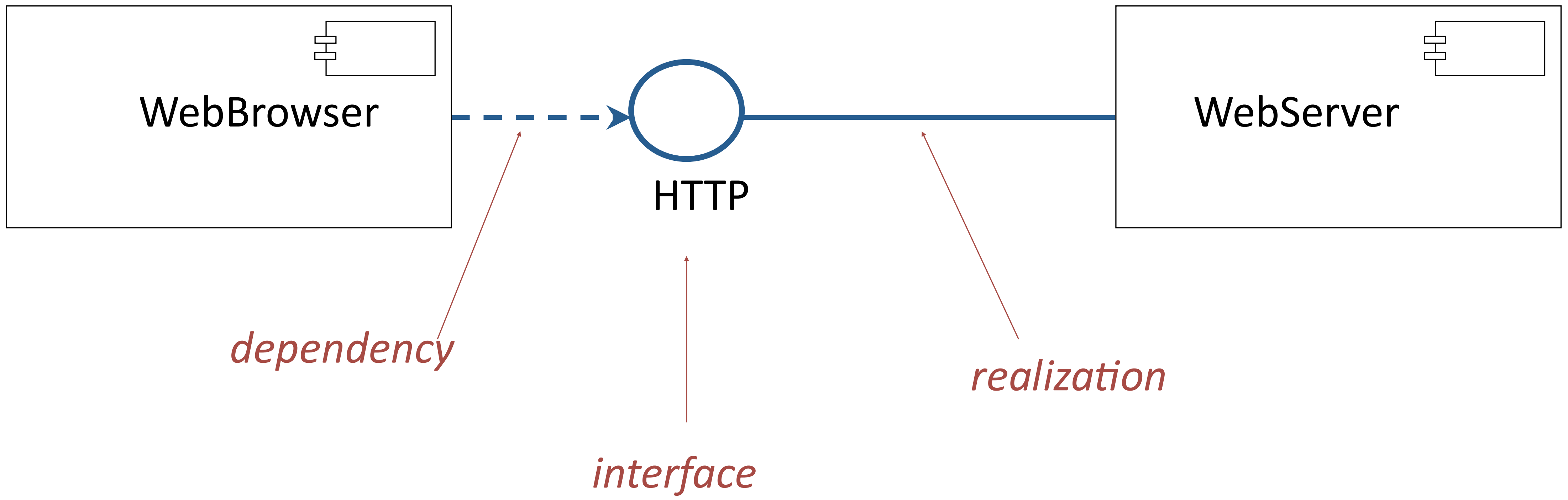
Architecture



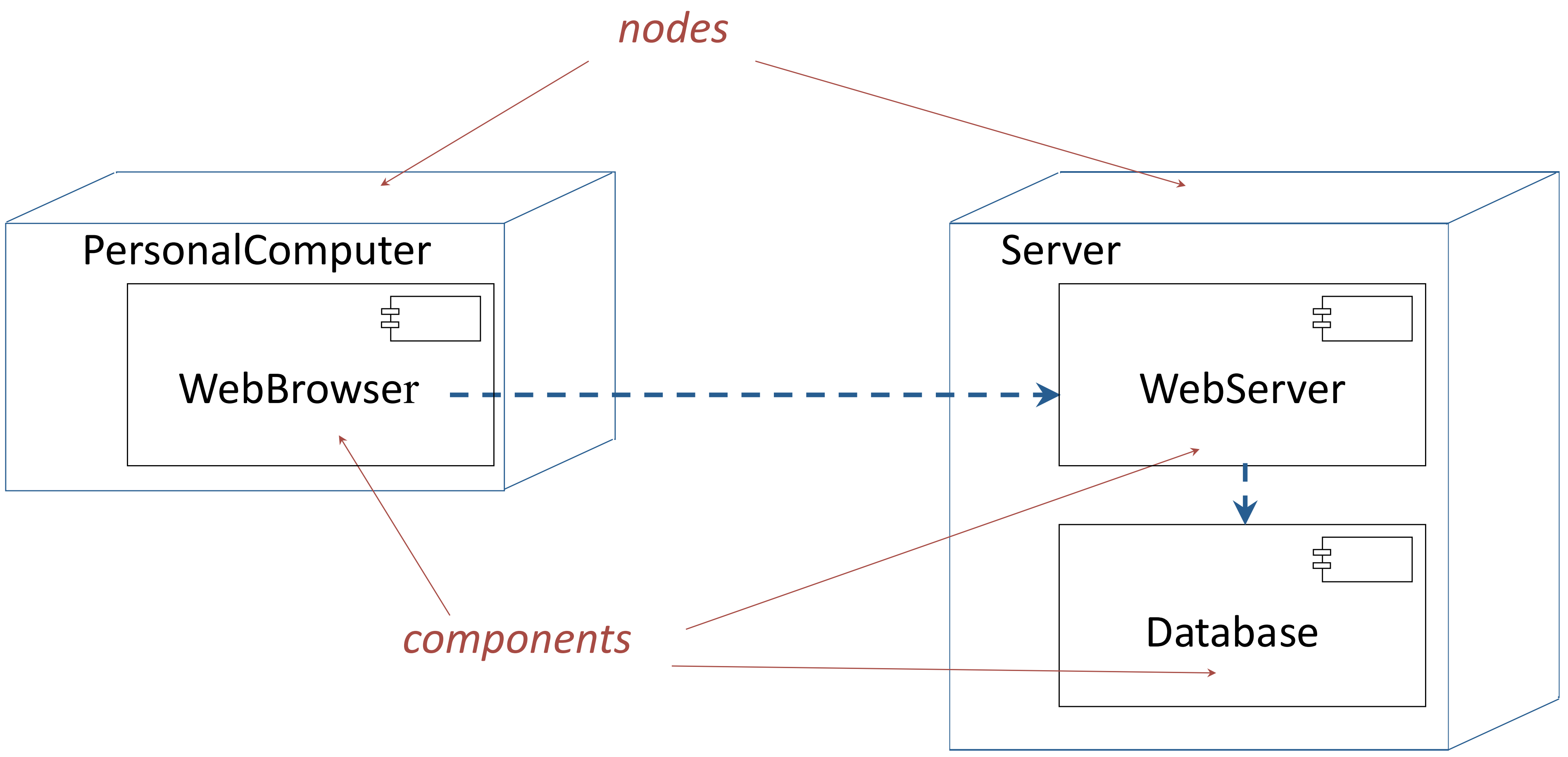
Architecture



Example: interface diagram



Example: deployment diagram



Deployment environments

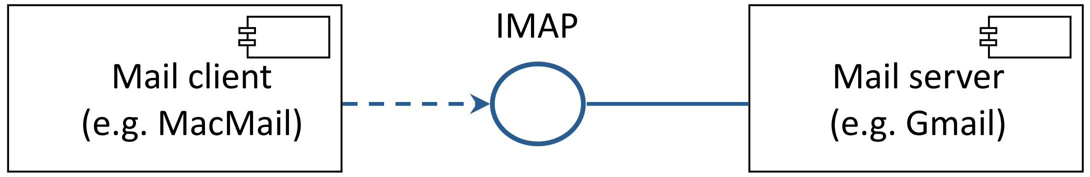
- Development

- Production
- Staging
- (Acceptance testing)

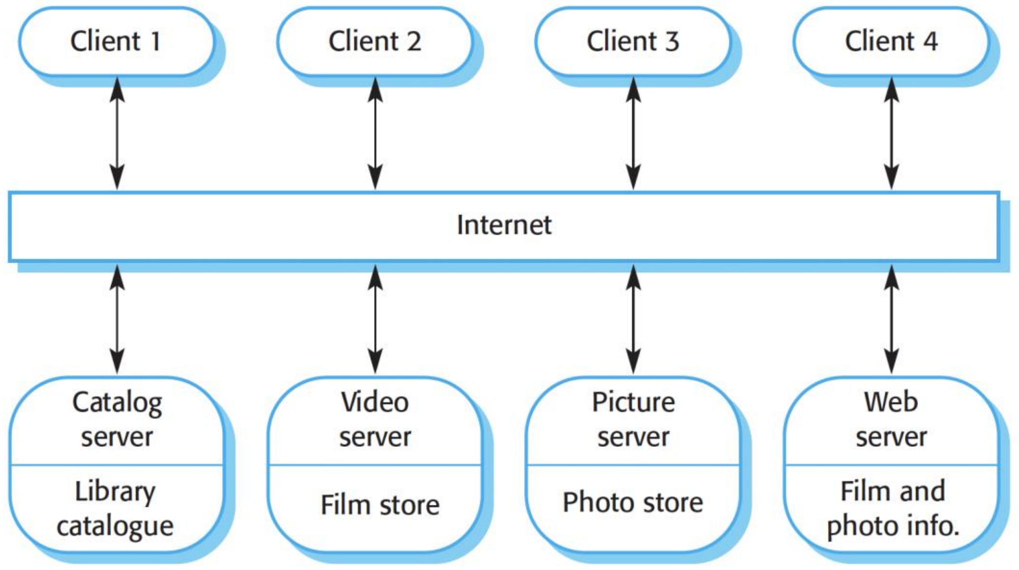
Architectural styles

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

Client/Server



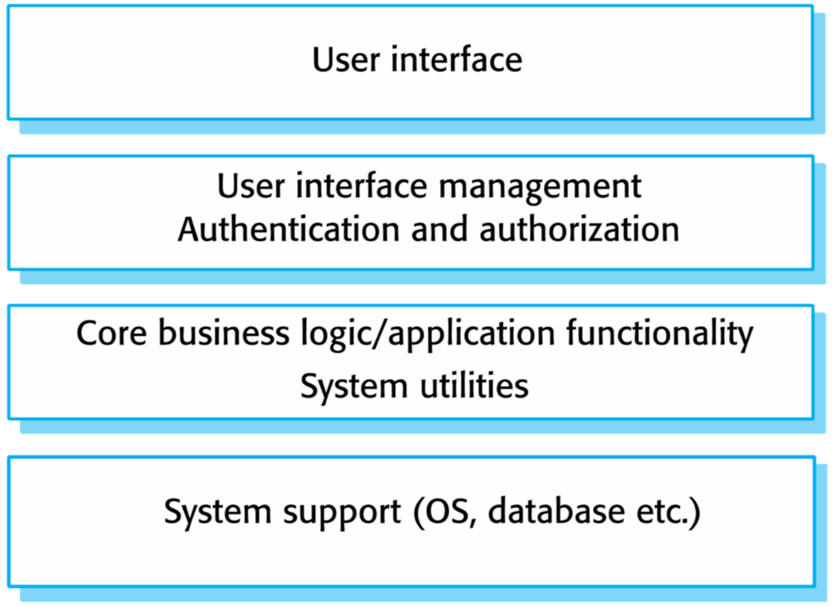
- Control flow in client and server are independent
- Communication follows a protocol
- If protocol is fixed, either side can be replaced independently
- Peer-to-peer: same component can act as both client and server
- Adv: Easy to scale, Clear separation
- Challenges: Latency, Tight API Coupling
-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



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

Layered Architecture

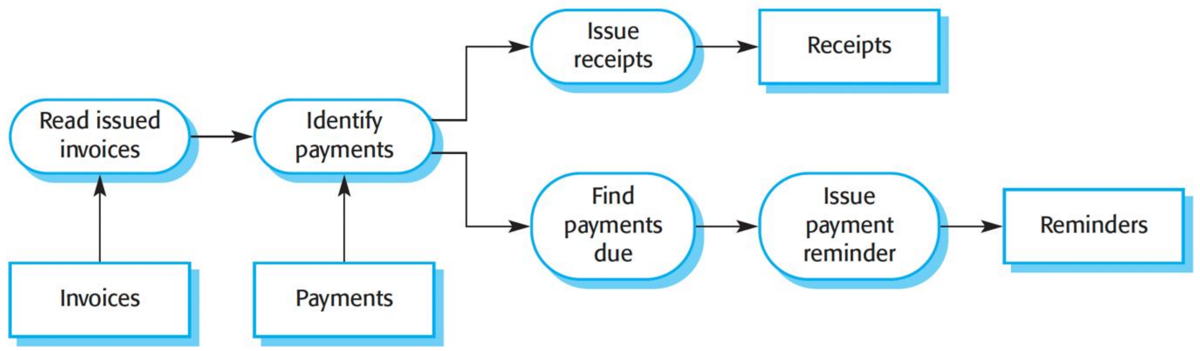
- Partition subsystems into stack of layers
  - Layer provides services to layer directly above
  - Layer relies on services to layer directly below
- Advantage:
  - Constrains coupling, Easy to understand
  - Easy to replace layers
  - Clear ownership/testing boundaries
- Danger: leaky abstractions
  - 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*

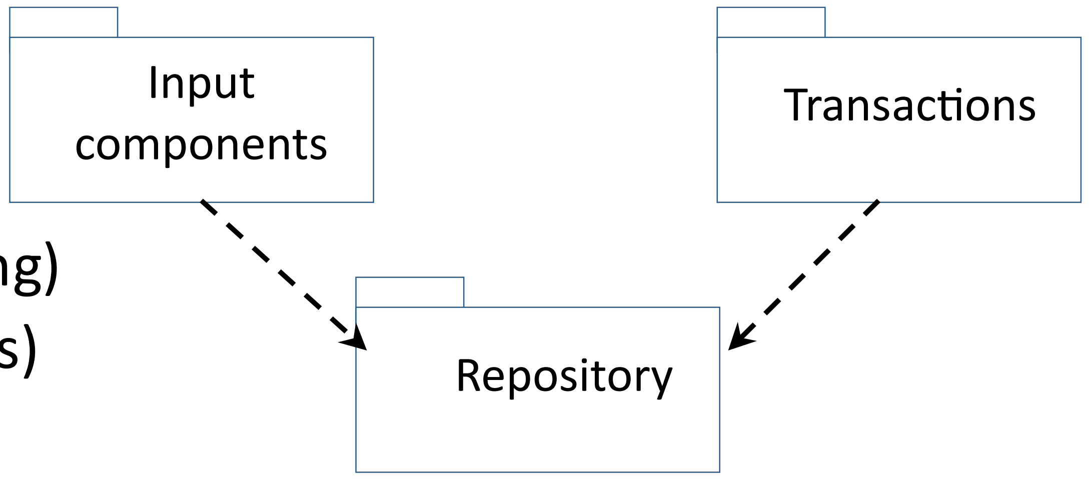
Pipe and Filter

- 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

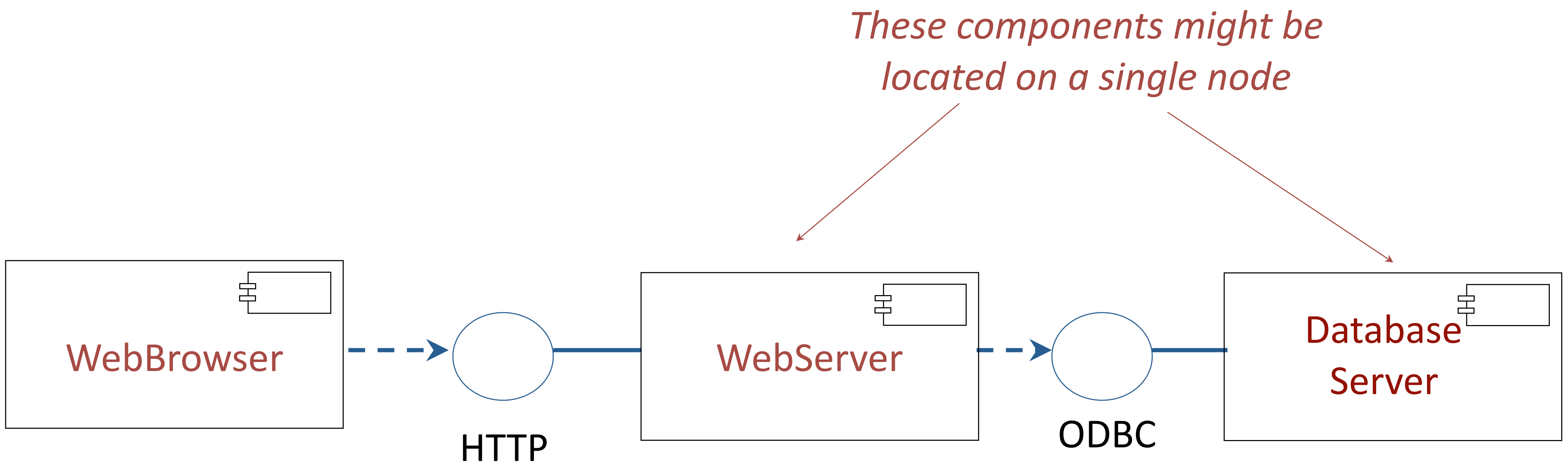


Repository

- 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



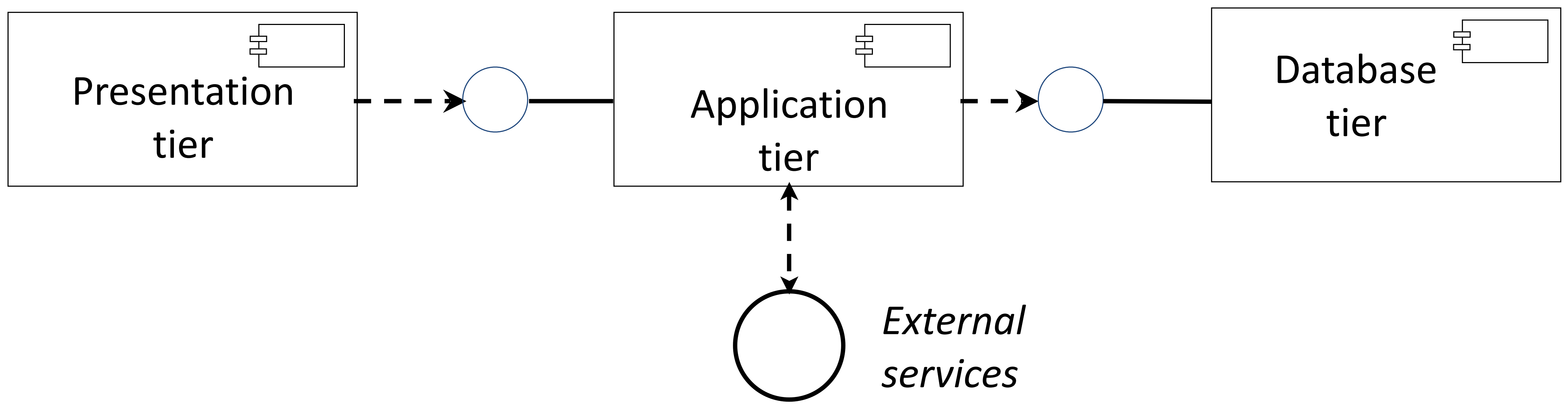
Component diagram



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

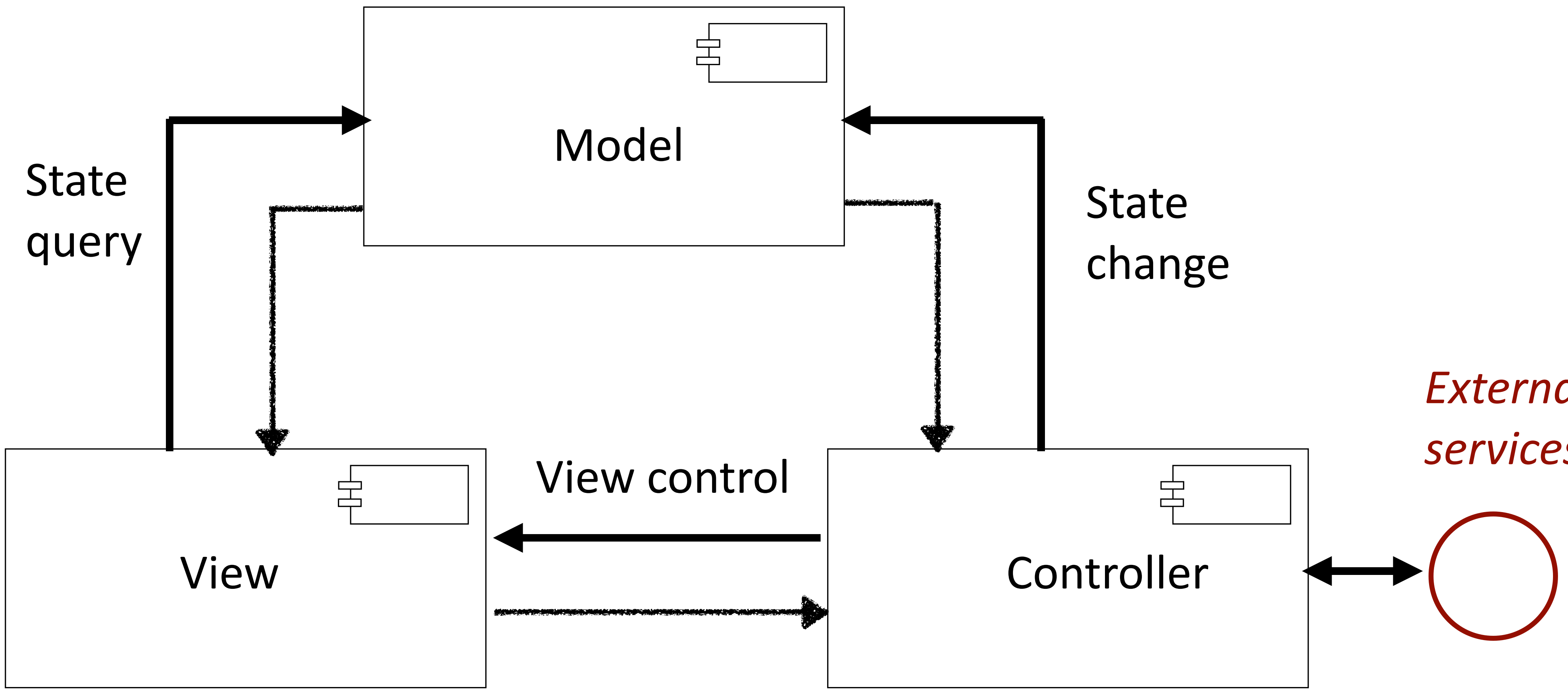


Model-View-Controller

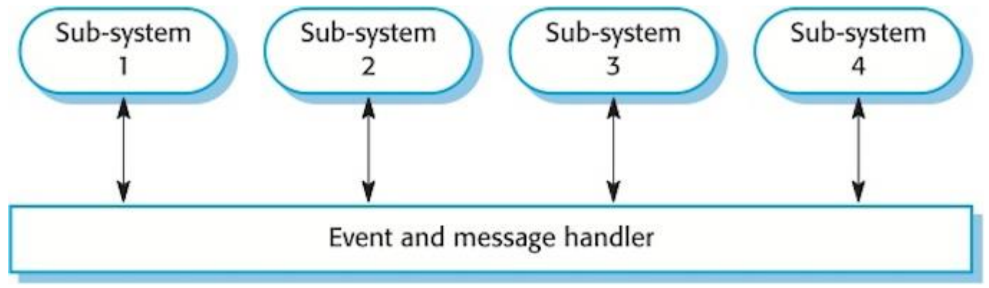
- Beware: many variations
  - Some are architectural styles: system-level responsibilities partitioned into different components
    - Example: **Django**/**Spring Framework/ASP.NET** for building web apps
  - Some are program design patterns: functionality divided between different classes
    - Focus 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



Publish-subscribe



- Event-driven control
  - Application responds to external stimuli and timeouts
  - No centralized orchestration
- Very loose coupling – components communicate via message broker
  - Easy to extend
  - Difficult to analyze (observer pattern)
    - No control over what (if any) code responds to an event
    - Potential for conflicts (multiple components respond in incompatible ways)
    - Potential for silently dropped events
    - Call stacks may not reflect causality

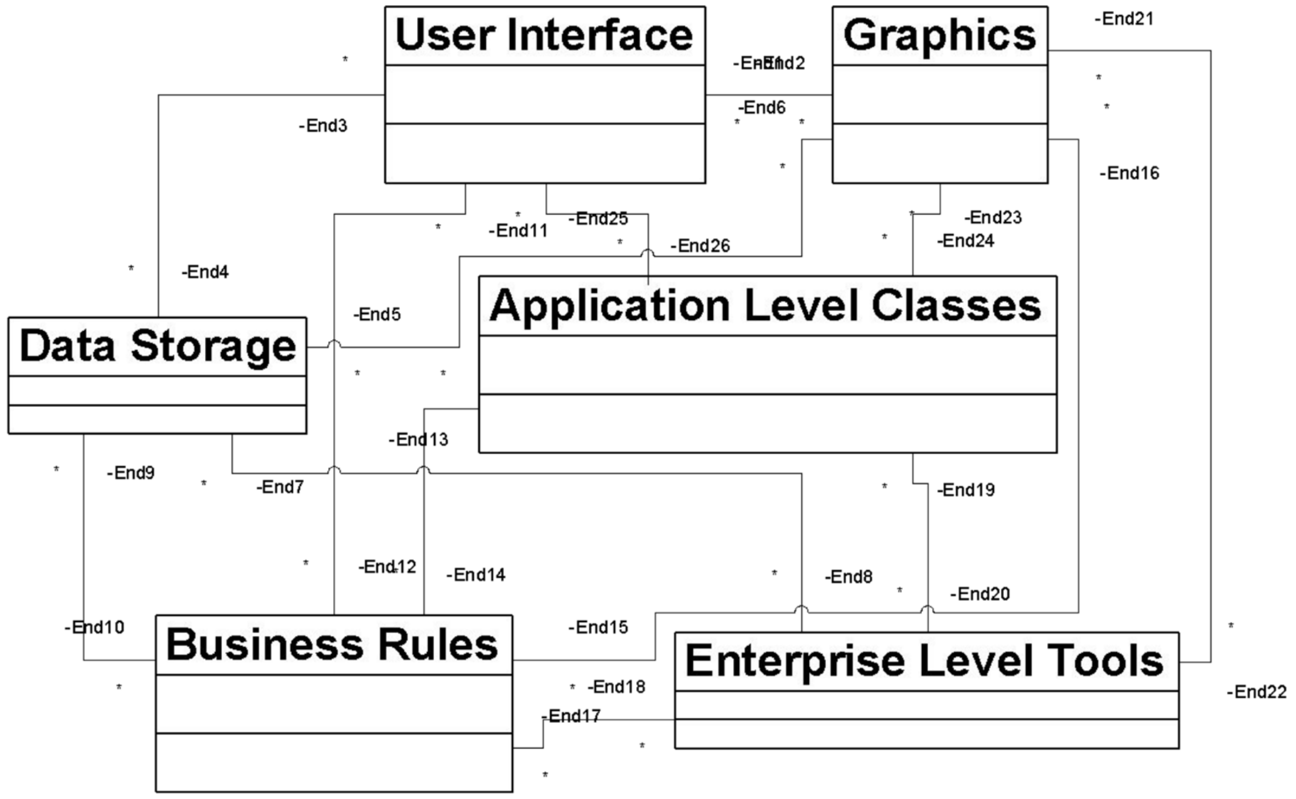
Comparison of Architectures

- Is Client-Server also a Layered architecture?
- Pipe and filter vs Layered?

- Architectural style defines
  - Vocabulary of components and connectors
  - Constraints on the elements and their combinations: Topological (no cycles), Execution (timing, etc.)
  - Non-functional attributes: performance, lack of deadlock, …

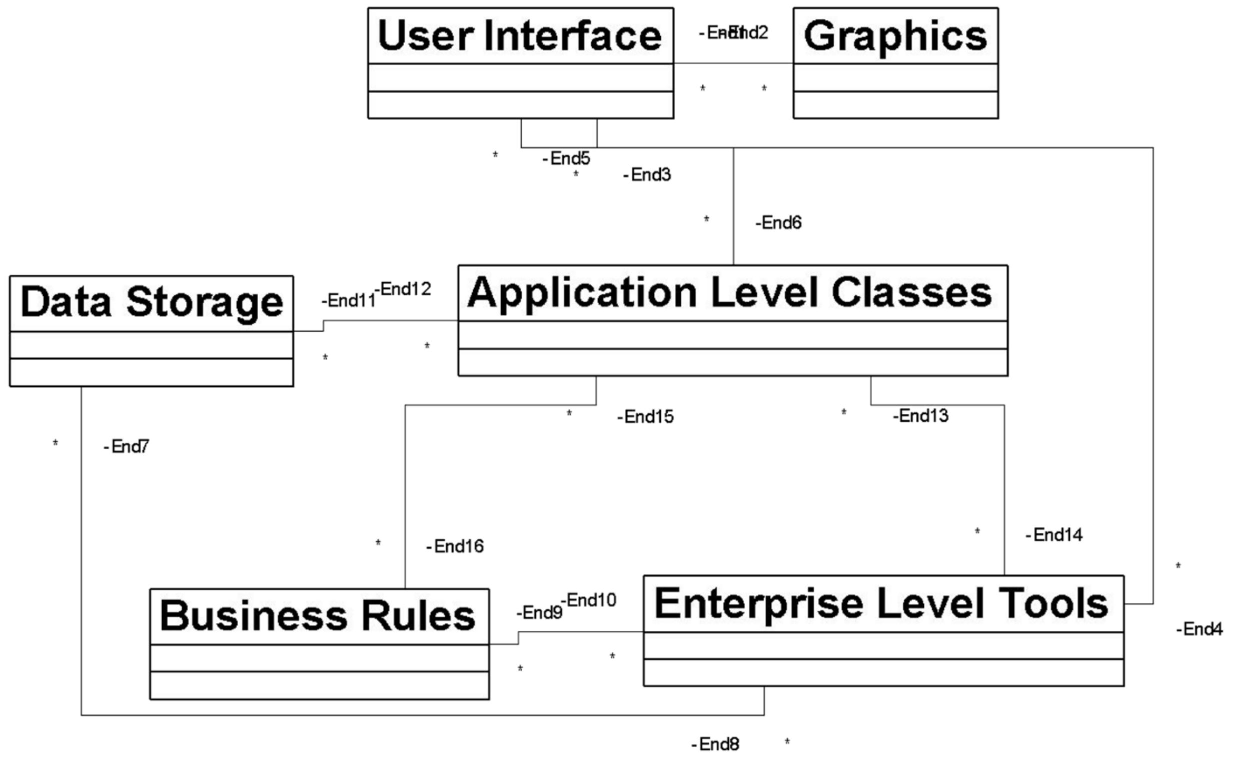
Coupling and Cohesion (Examples)

Is this tightly coupled or loosely coupled?



Coupling and Cohesion (Examples)

Is this tightly coupled or loosely coupled?



Coupling and Cohesion (Examples)

Strong or weak cohesion?

```
class Employee {

public:
  …
  FullName GetName() const;
  Address GetAddress() const;
  PhoneNumber GetWorkPhone() const;
  …
  bool IsJobClassificationValid(JobClassification jobClass);
  bool IsZipCodeValid (Address address);
  bool IsPhoneNumberValid (PhoneNumber phoneNumber);
 …
  SqlQuery GetQueryToCreateNewEmployee() const;
  SqlQuery GetQueryToModifyEmployee() const;
  SqlQuery GetQueryToRetrieveEmployee() const;
 …
}
```

Coupling and Cohesion (Examples)

Strong or weak cohesion?

```
class Employee {
public:
   FullName GetName() const;
   Address GetAddress() const;
   PhoneNumber GetWorkPhone() const;

   JobClassification GetJobClassification() const;

private:
   FullName name;
   Address address;
   PhoneNumber workPhone;
   JobClassification jobClass;
};
```

```
class EmployeeValidator {
public:
   bool IsJobClassificationValid(JobClassification jobClass) const;
   bool IsZipCodeValid(const Address& address) const;
   bool IsPhoneNumberValid(const PhoneNumber&
phoneNumber) const;

   bool IsValid(const Employee& employee) const;
};
```

```
class EmployeeRepository {
public:
   void Create(const Employee& employee);
   void Update(const Employee& employee);
   Employee Retrieve(EmployeeId id) const;
};
```

Virtualization

Deployment concerns

- Dependency conflicts
- Configuration, data sprawl
- OS portability
- Unintended interactions
  - Filesystem has same problems as global variables

- Solution: Encapsulation; but...
  - Deploying on separate machines risks under-utilization

Virtual machines

- Multiple OS instances running on one machine
  - Real hardware is managed by host OS or hypervisor
- Improves hardware utilization, reduces cost
  - Avoids energy consumption by redundant hardware
- Stateful – still risks data sprawl
  - Address with automated administration
- High overhead – software redundancy

- **Examples**: VMware, VirtualBox, Xen, Hyper-V

System configuration management

- Automate deployments
  - Installing dependencies
  - Configuring OS
  - Configuring application
- Combat sprawl
- **Examples**: Ansible, Puppet, Chef, Vagrant

Example (Vagrant)

Sets up two virtual machines:

- db (database server)
  - Postgresql
  - Private network
- web (web server)
  - Private network to contact DB
  - Nginx server
  - Creates simple webpage
- Run: “vagrant up”

```
Vagrant.configure("2") do |config|
config.vm.box = "ubuntu/jammy64"


# --- DB VM ---
config.vm.define "db" do |db|
db.vm.network "private_network", ip: "192.168.56.11"
db.vm.provision "shell", inline: <<-SHELL
apt-get update -y
apt-get install -y postgresql
systemctl start postgresql
SHELL
end


# --- Web VM ---
config.vm.define "web" do |web|
web.vm.network "private_network", ip: "192.168.56.10"
web.vm.network "forwarded_port", guest: 80, host: 8080
web.vm.provision "shell", inline: <<-SHELL
apt-get update -y
apt-get install -y nginx
echo "Hello from Web VM" > /var/www/html/index.html
systemctl restart nginx
SHELL
end
end
```

Containers

- Trade OS heterogeneity for reduced redundancy
- Still isolate filesystem, network without duplicating OS
- Lightweight – new instances start quickly
  - Improves elasticity
- Often encapsulates a single application
- Often treated as stateless (don't write to filesystem)

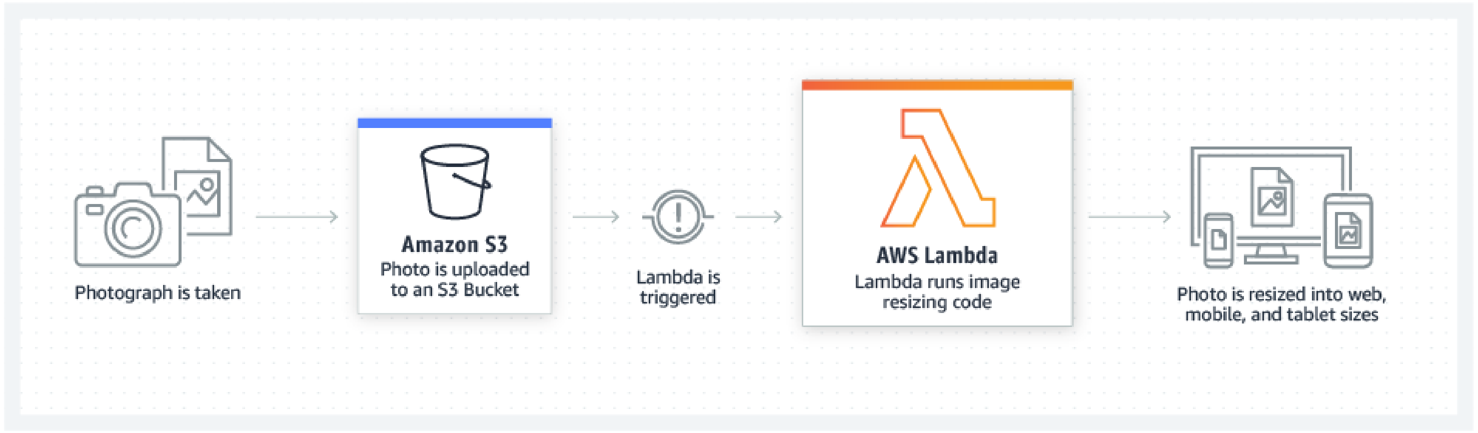
- Examples: Docker, LXC

"Serverless"

- Computation nodes are stateless, ephemeral, and event-triggered
  - Data store services still persist state, but are application-agnostic
- Application decomposed into event-handler functions
  - Event dispatch, container lifetime managed by platform
- Examples: Amazon Lambda (FaaS), Azure Functions

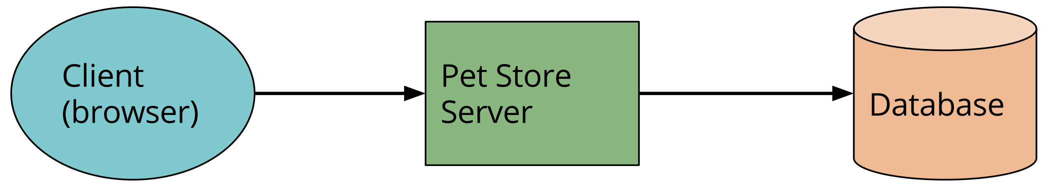
Azure Functions

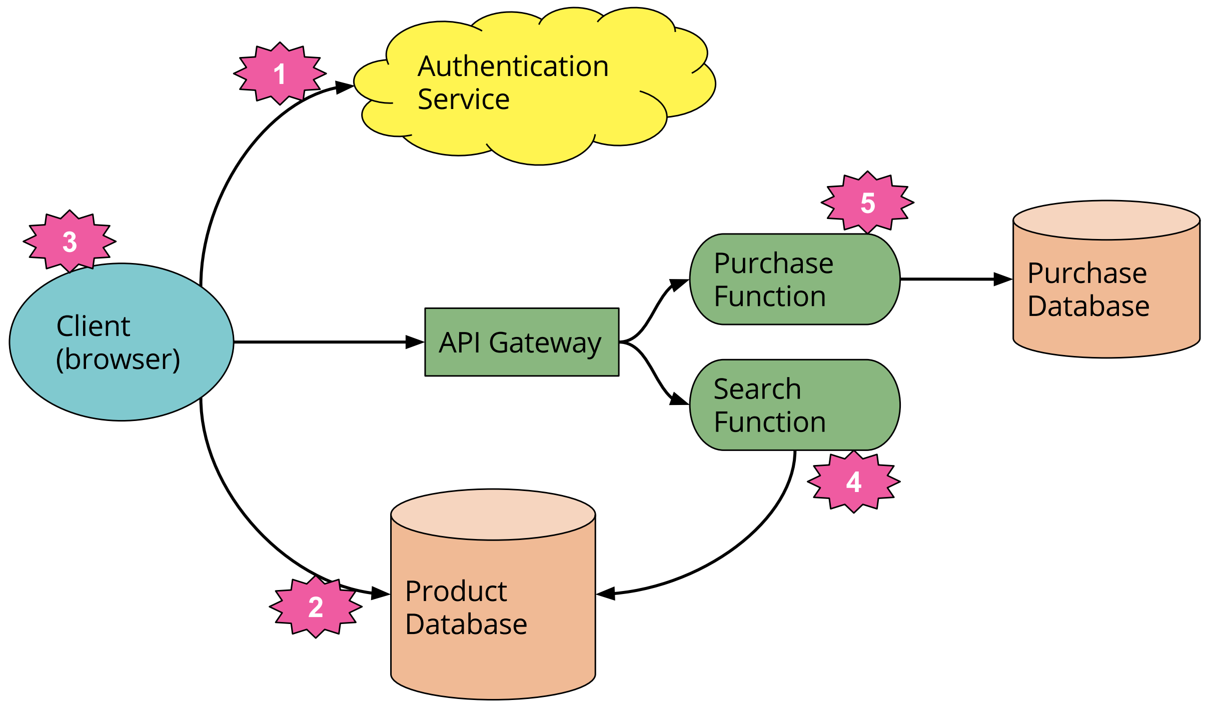
AWS Lambda


Photograph is taken
Amazon S3
Photo is uploaded to an S3 Bucket
Lambda is triggered
AWS Lambda
Lambda runs image resizing code
Photo is resized into web, mobile, and tablet sizes

https://martinfowler.com/articles/serverless.html

Three-tier vs. serverless



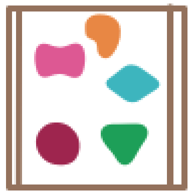


More examples: https://martinfowler.com/articles/serverless.html

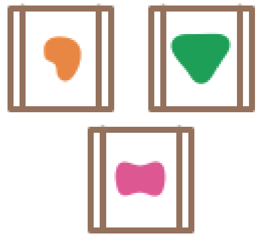
Microservices

- Components encapsulate services and expose them via standard interfaces. Are ideally binary-replaceable
    - In practice, many frameworks for managing modular applications are language-specific (e.g., OSGi for Java)
    - OOP abstractions like objects, methods are complicated at language boundaries and distributed deployment
- Microservices constrain component definition to **reduce coupling**
    - Language-agnostic protocols (e.g., RESTful HTTP)
    - Independently deployable
- Advantage: More scalable, fault tolerant, rapid roll out, distributed database
- Disadvantage: Complex monitoring, more points of failure, network delays, testing is challenging
- Examples: Netflix, Amazon, Uber

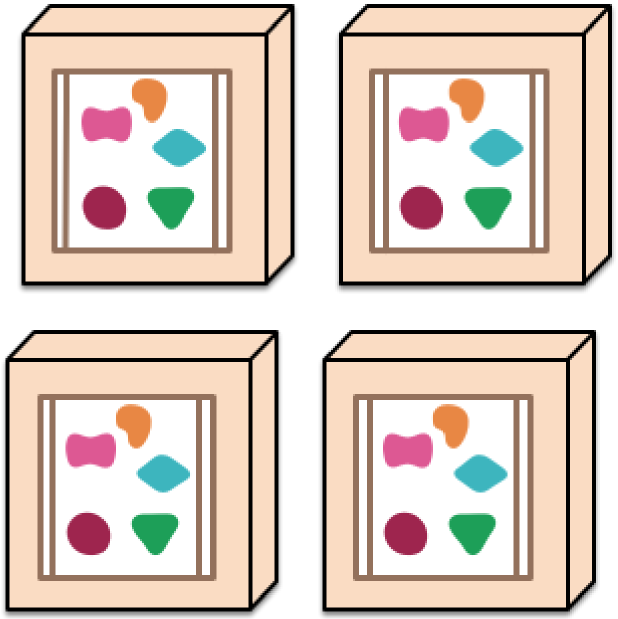
A monolithic application puts all its functionality into a single process…



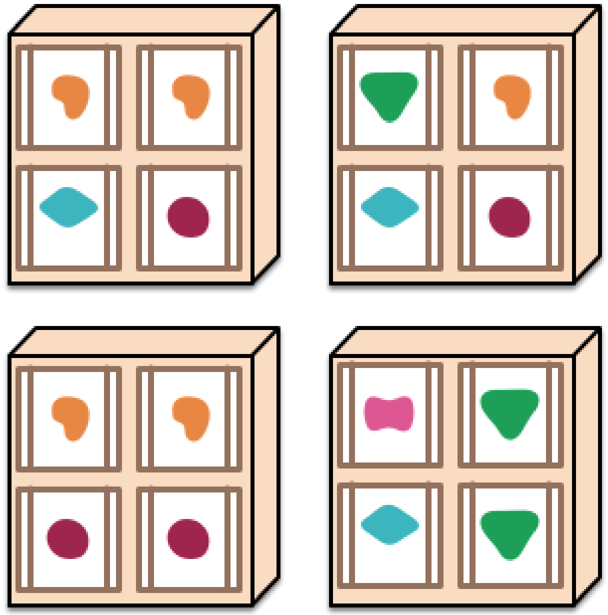
A microservices architecture puts each element of functionality into a separate service…



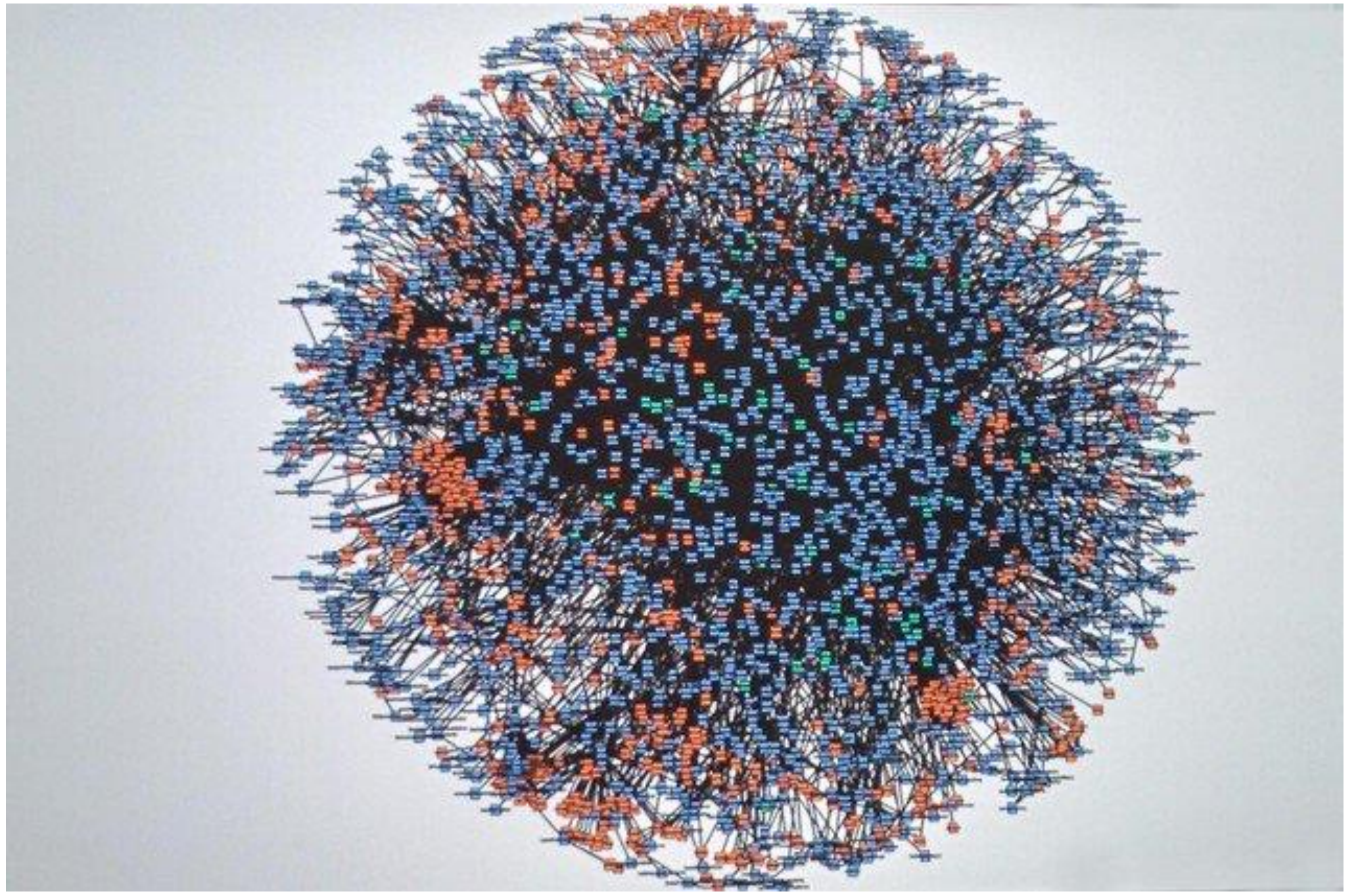
… and scales by replicating the monolith on multiple servers



… and scales by distributing these services across servers, replicating as needed.



Amazon’s Microservices Architecture (2008)



https://x.com/Werner/status/741673514567143424

Software Architecture Resources

- An Introduction to Software Architecture: David Garlan and Mary Shaw
- Software Engineering, Ian Sommerville: Chapter 6
- https://martinfowler.com/architecture/