Session 2 general feedback

• Need to know deployment infrastructure
  • Have you met with your client?

• Probably need to refine requirements
  • Have you met with your client?

• Remember good meeting strategies
  • Moderator, recorder, agenda (sent ahead of time)
  • Don't just reserve a time – PoC should send an e-mail
Lecture goals

• Identify common architectural styles (continued)
  • Three tier architecture
  • Model-view-controller

• Encapsulate deployments using virtualization
Architectural styles

... continued from Lecture 10
Three tier architecture

• Extension of client/server model
• Commonly used for small-medium web sites
  • Classic example: LAMP stack
Basic website (client/server)
Extension: data store
Review: component & deployment diagrams

PollEv.com/cs5150
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 → Application tier → Database tier

External services
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: Play Framework
  • Some are program design patterns: functionality divided between different classes
    • Focus on reusable controls
    • Example: Swing widgets
    • 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)
Features of MVC

• Separated presentation
  • Decouple model and view (replaceable components)
  • Multiple (possibly simultaneous) views supported
Example: "mission control" terminal
(based on a past CS 5150 project)

• A vehicle (unmanned aircraft) is flown by a pilot interfacing with a computer terminal on the ground

• Vehicle communicates with ground station via radio signals
  • Actuation commands (uplink): change throttle, angle flaps, etc.
  • Sensor measurements (downlink): air speed, GPS position, actuator settings, etc.
Example: View

• Graphical user interface shows model properties (sensor readings, derived state) as instrument dials and provides input widgets for commanding actuators.
Example: Model

• Maintains record of state of vehicle (speed, fuel, ...)
• Computes derived properties of vehicle (rate of turn, predicted trajectory, ...)
• Updates state in response to actions from controller (e.g. new telemetry received)
• Provides view with information to be displayed to user

Different vehicles will need different models but might not require new views or controllers.
Example: Controller

**Scenario:** Pilot wishes to change flap angle to 20 deg to increase lift and accommodate a slower speed for landing.

1. View sends message to controller: `setFlaps(20)`
2. Controller sends radio command to vehicle: `setFlaps(20)`
3. Vehicle acts on command and replies to controller: `flapsSet(20)`
4. Controller relays telemetry to model: `flapsSet(20)`
5. Model updates state and recomputes stall speed
6. Model notifies view of new state (flap setting, stall speed)
7. View displays new state in user interface
View

- Presents application state and controls to user
- Typically subscribes to model for notifications of state changes
  - "Observer pattern"
- Responsible for rendering to a particular interface
  - Drawing graphics, generating HTML, printing text
- Sends user input to controller
- A single model can support multiple views
  - Example: web app, native app

CS 5150 project: create a new view for pilot. Since this was a separate component, it could replace the old view with no other changes to the system.
Model

• Records state of application and notifies subscribers
  • Responds to instructions to change state (from controller)
• Does not depend on either controller or view
• State may be stored in objects or databases
• May be responsible for some application logic (e.g. input validation)
Controller

• Manages user input and navigation
• Defines application behavior
• Maps user actions to changes in state (model) or view
• May interact with external services via APIs
• May be responsible for some application logic (e.g. input validation)

• Variety in distribution of duties between model and controller
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

Activity: system decomposition

• What happens when I tap "send" in a mail app on my phone?
  • Draw a hardware block diagram
  • Draw layers of system software
Closing remark

• Beware software architectures that resemble corporate hierarchy
  • Refactoring more disruptive than reorgs
  • Be aware of and accommodate political context, but architecture should serve
    the application more than the developer
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
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
Three-tier vs. serverless

- See 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
Credits

• Architectural diagrams adapted from work by William Arms