Lecture goals

• Visualize structural models with deployment and interface diagrams
• Identify common architectural styles
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
What is architecture?

• 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"

• Architecture
  • High-level "how"
  • Mid-level "what"

• Program design
  • 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 can hover

• 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
Architectural 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
  - See Sommerville, *Software Engineering, Tenth Edition*, Figure 6.1 (or Chapter 6 slides)

- Oscilloscope user interface
  - See 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

Lexical analysis → Parser → Code generation
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

WebBrowser <- HTTP -> WebServer

dependency

interface

realization
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)
Example: simulator

- **Vehicle state** propagated by integrator
- Integrated equations affected by physics models, actuators
- Sensors read vehicle state
- Sensors and actuators communicate with controller via streams and/or registers
- Register values aggregated by a registry server communicating with registry clients

- Simulator process consists of integrator, models, sensors & actuators, and registry server
- Controller's registry client reads/writes controller's registers via shared memory
- Registry server communicates with clients via UDP
- Streams are sent via TCP
- Controller runs on virtual machine
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)
Layered

• 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
  • May need services of multiple lower layers

• See Sommerville, *Software Engineering, Tenth Edition*, Figure 6.8 (or Chapter 6 slides)
Examples

• OSI reference model architecture

• 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

• Applications:
  • Streaming media
  • Graphics shaders
  • Signal processing

• Caveats:
  • Awkward to handle events (interactive systems)
  • Rate mismatches if branches merge
Example: Oscilloscope

• See Garlan & Shaw, "An Introduction to Software Architecture"
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)

• 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

• See Garlan & Shaw, "An Introduction to Software Architecture"
Credits

• Architectural diagrams adapted from work by William Arms