CS 3110

Modular Design

Prof. Clarkson

Fall 2015

Today's music: Top Down by Fifth Harmony

Review

Previously in 3110:

- language features: modules, structures, signatures, abstract types, includes, functors
- purposes: namespaces, encapsulation, code reuse

Today:

- how to design large programs
- (until after Prelim 1: no new features)

ARCHITECTURE

Architecture

- Highest-level design of software system
- Elements of architecture:
  - Code **components**
    - e.g. modules, libraries
    - *module*: logically separable part of program w.r.t. compiling and loading; in OCaml actually called a module or set thereof; in other languages might be called class or package or...
  - Externally visible **properties** of those components
    - e.g., types, signatures, documentation
  - **Relationships** among components
    - e.g., implemented-by, shares-data-with, independent-of

Why analyze architecture?

- **Understanding**
  - Communicate system design to implementers, testers, maintainers, clients, users
  - Reduce system to a few parts; abstract from details; simplify
    - Working memory: humans can pay attention to only a small number of things at a time (3 or 4? 7?)
- **Reuse**
  - Identify what components can be repurposed from other systems
  - Assembly line model: cheaply produce system out of stock components
    - e.g., web mashups, new 3110 website
- **Construction**
  - Division of (independent) labor
  - How to add new features

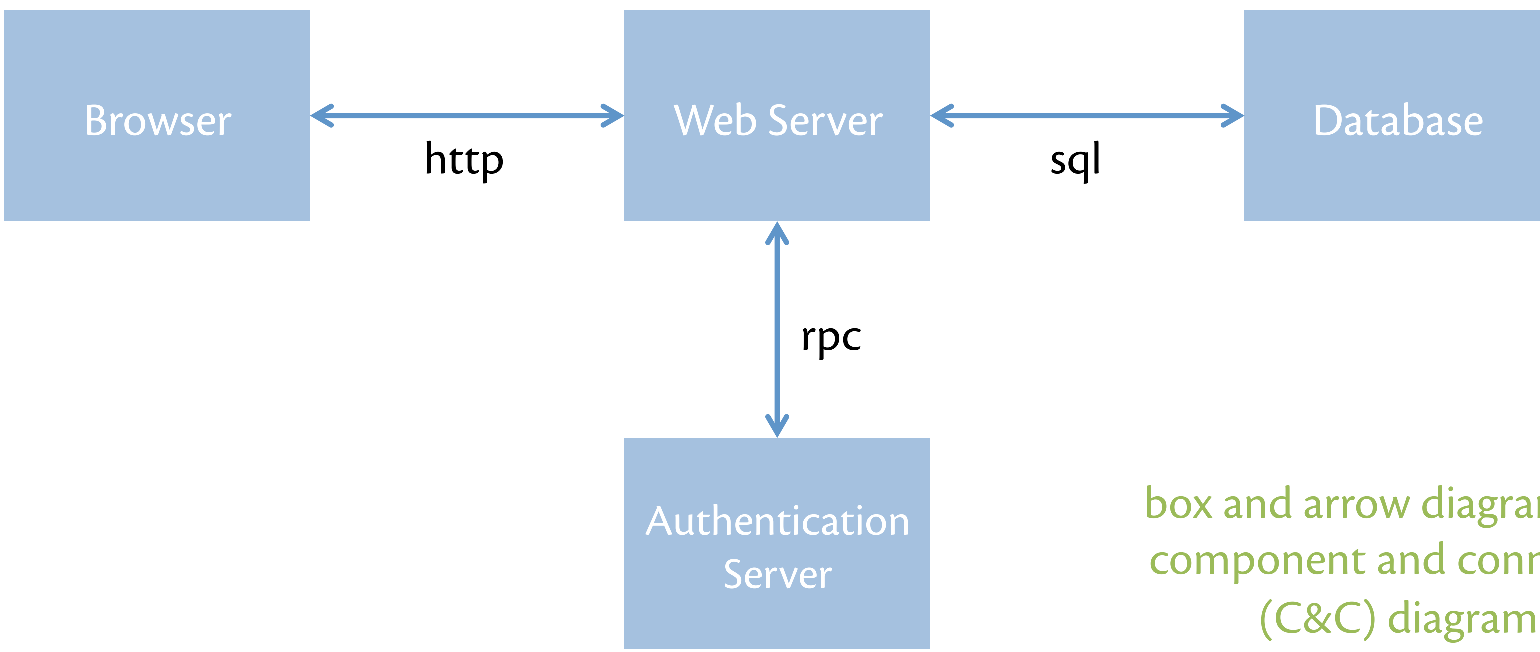
Architecture of survey system

Requirements:

- present multiple-choice questions to user
- collect and store answers
- present results-in-progress to user after they submit

Architecture of survey system

New requirement: only some users are authorized to take survey; must authenticate users before they can register response



box and arrow diagram, aka component and connector (C&C) diagram

Building blocks of architecture

Components:

- Computation elements or data stores
- Primarily from the view of **run time**: what happens while system is executing?
- Not necessarily from the view of **compile time**: how is code physically organized?

Building blocks of architecture

Connectors:

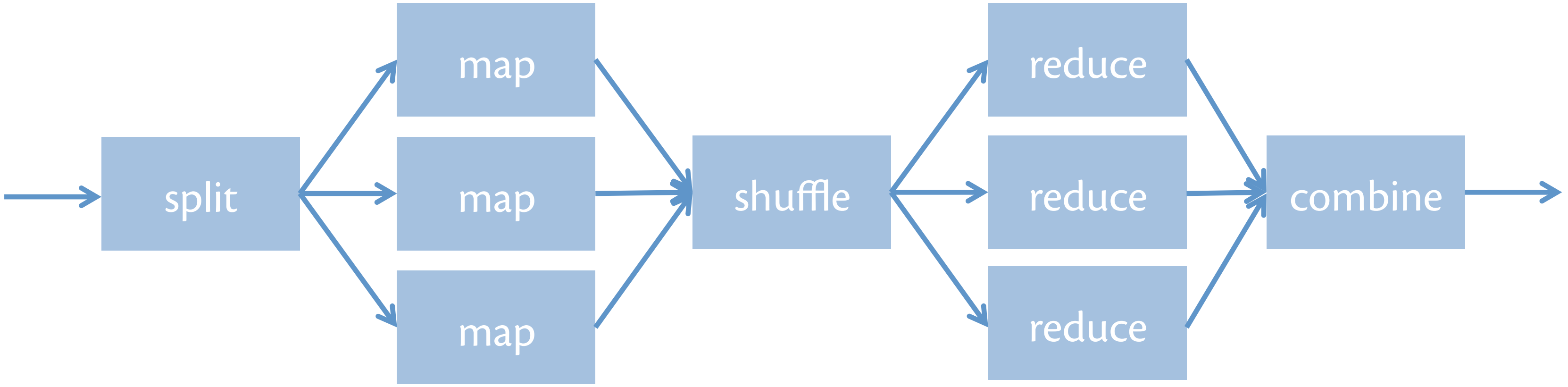
- Protocol: agreed upon means of communication
  - e.g., TCP, function call
- Topology could vary: binary, broadcast, ring, ...

Ex: Pipes and filter architecture

- **Filter:** component that transforms data
  - receives data on input pipes
  - sends output data over pipes to other filters
  - might have >1 inputs, >1 outputs
  - each filter is independent of others and operates concurrently
- **Pipe:** connector that relays data
  - unidirectional
  - does not change data
  - pipes handle storage, synchronization, rate of transfer, etc.

Ex: Pipes and filter architecture

MapReduce as a pipe and filter architecture:



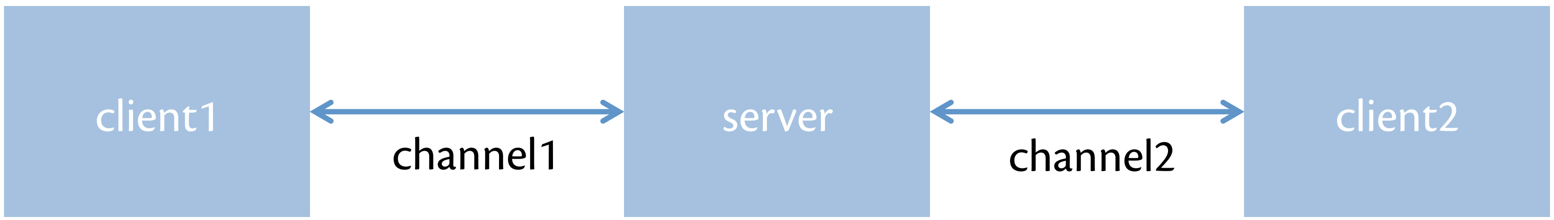
Ex: Shared data architecture

- **Data repository:** component that stores data
  - provides reliability, persistence, access control
  - might be passive or might actively notify accessors about changes in data
- **Data accessor:** component that does computation with data
  - gets data from repository, computes, puts data back to repository
  - accessors do not directly communicate with one another
- **Interfaces:** connectors that gives read/write access to repository

Ex: Shared data architecture

File system as a shared data architecture:

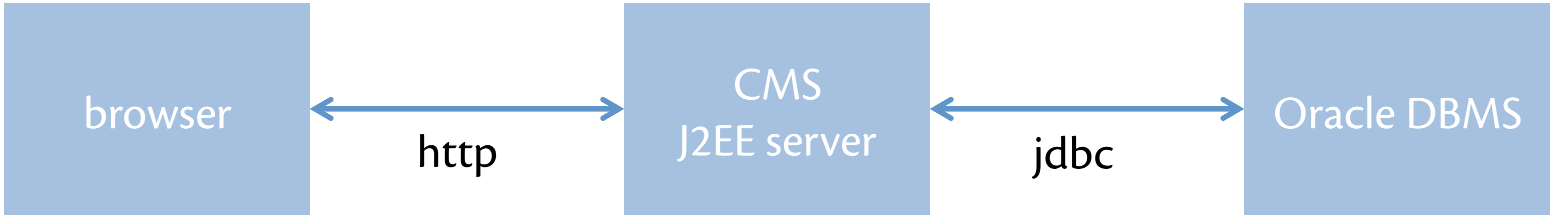
Ex: Client–server architecture



- **Server:** component that provides service/resources
  - When server provides a storage service, might reduce to shared data arch.
- **Client:** component that accesses service/resources
  - clients do not directly communicate with one another
  - clients need not be co-located with server
- **Channels:** connectors that allow client to make request, then server to return response
  - asymmetric: client can contact server, not vice-versa
  - (a)synchronous: client waits for response?
- Generalizes to *n-tier architecture,* in which server acts as client to another server, etc.

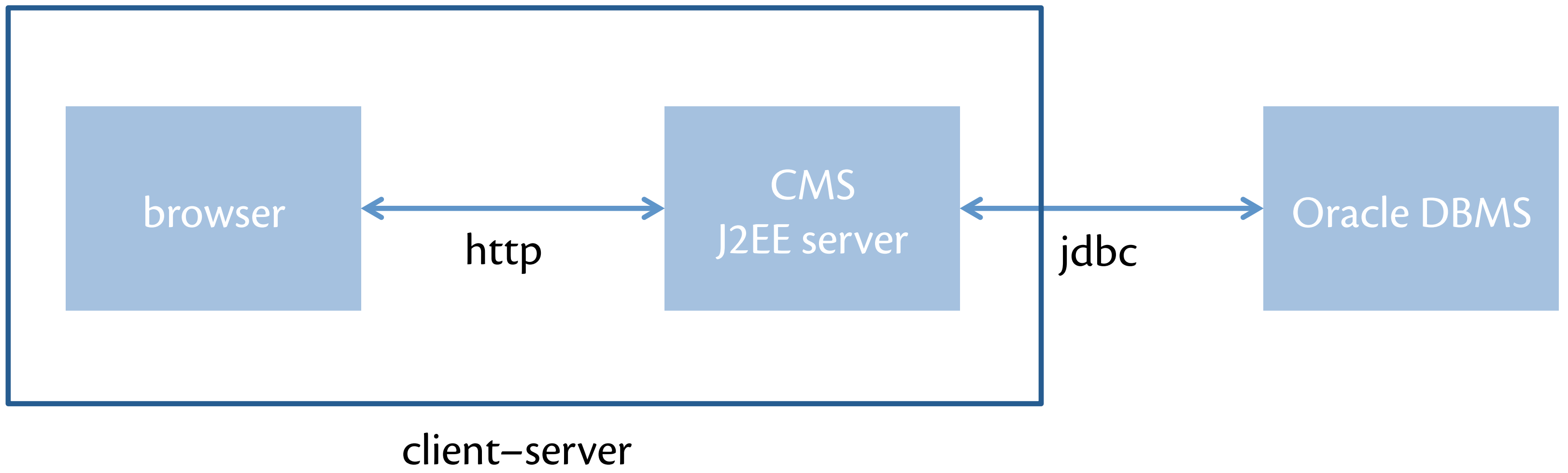
Ex: Client–server architecture

CMS as client-server architecture:



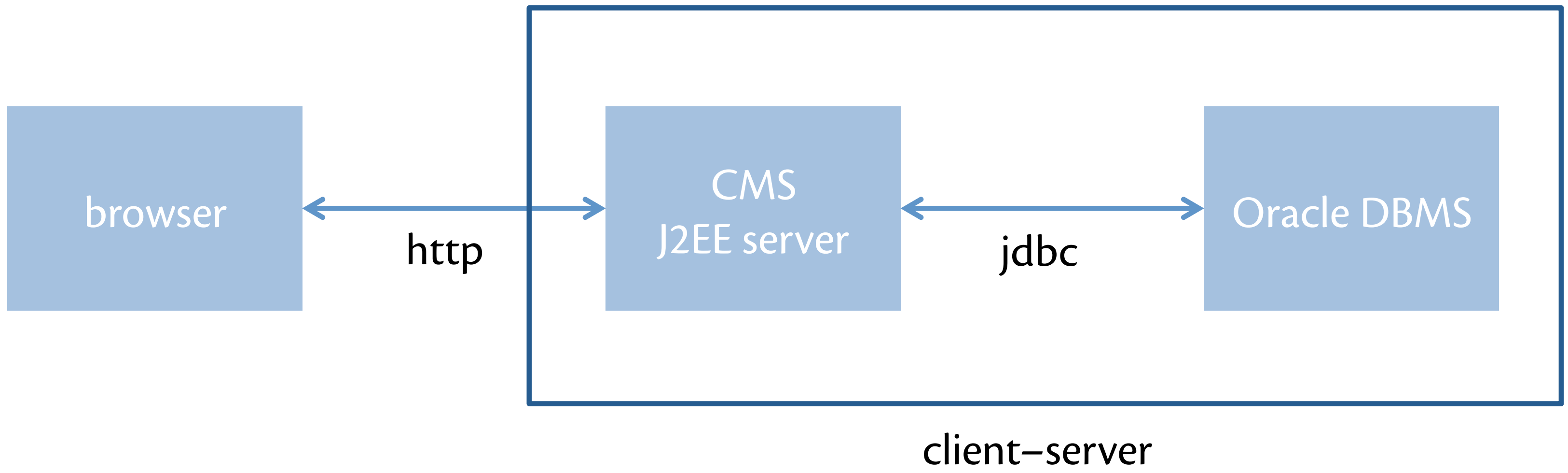
Ex: Client–server architecture

CMS as client-server architecture:



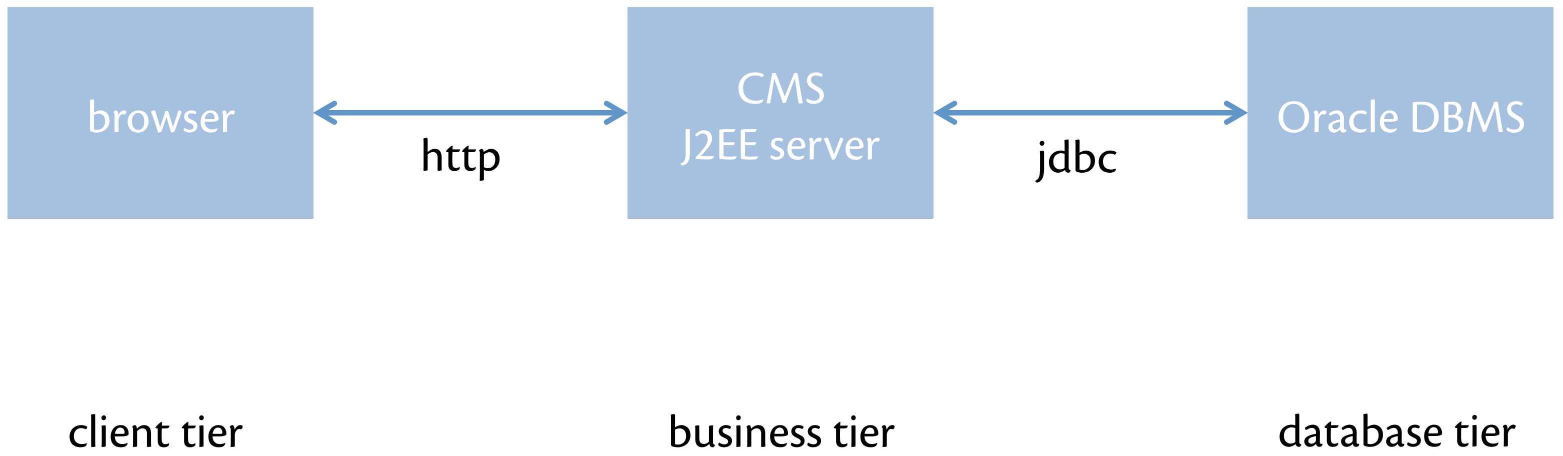
Ex: Client–server architecture

CMS as client-server architecture:



Ex: Client–server architecture

CMS as 3-tier architecture:



Question

Which architecture best describes the Enigma cipher?

A. Pipe and filter
B. Shared data
C. Client–server
D. None of the above
E. YNOXQ

Question

Which architecture best describes the Enigma cipher?

A. Pipe and filter

B. Shared data

C. Client–server

D. None of the above

E. YNOXQ

From architecture to design

- Architecture *is* a kind of design
  - focuses on highest level structure of system
  - based on principle of divide and conquer
- But architecture isn't about code per se
- As the *design process* iteratively proceeds, we get closer and closer to code
- *Design* as a phase of software development has a more specific connotation:
  - **System design:** decide what modules are needed, their specification, how they interact
  - **Detailed design:** decide how the modules themselves can be created such that they satisfy their specifications and can be implemented

SYSTEM DESIGN

Design criteria

- **Simplicity:** easily understood
- **Efficiency:** uses minimal resources
- **Completeness:** solves the entire problem
- **Traceability:** every aspect of design is motivated by some requirement

...not independent

...simplicity by default trumps everything else

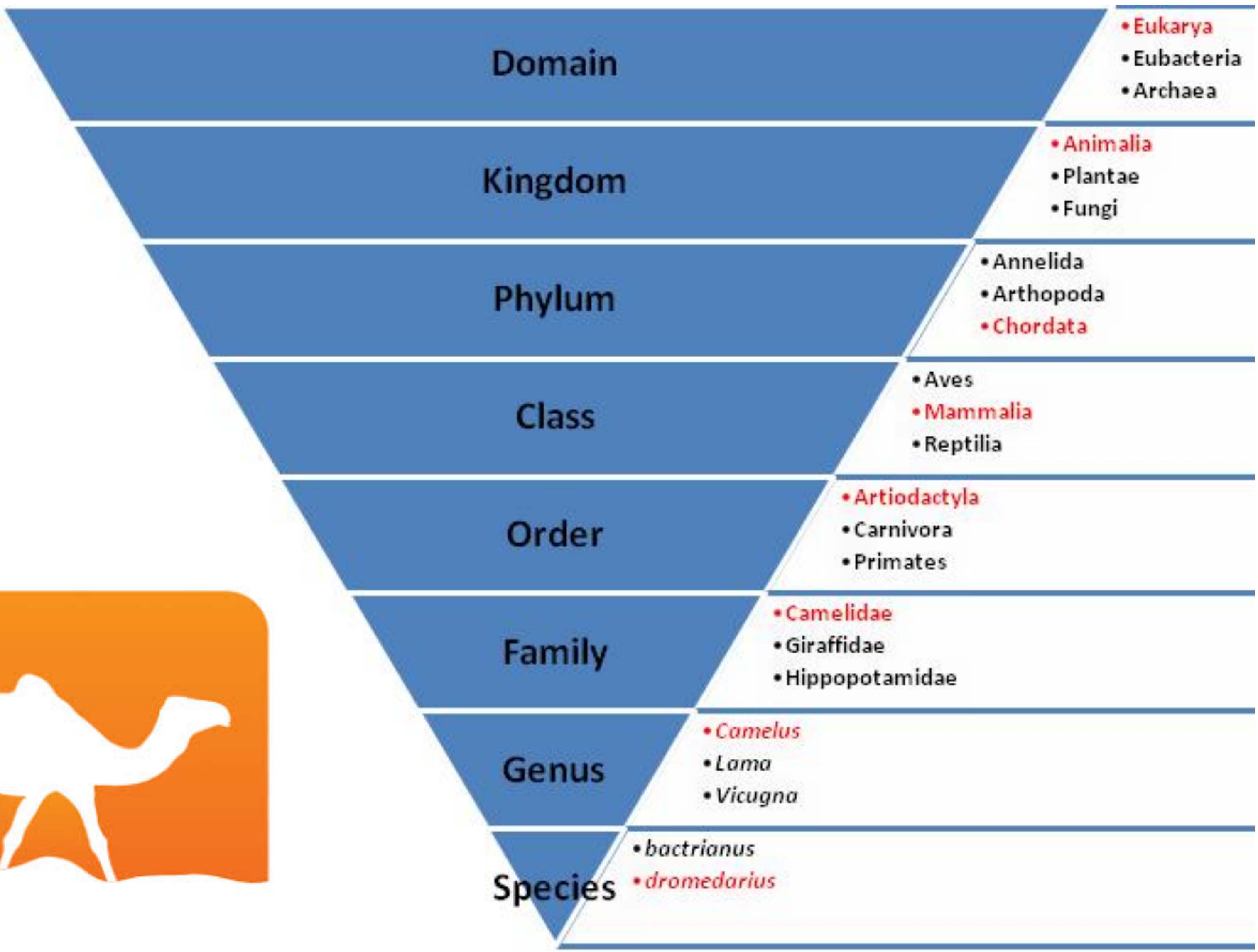
Design principles

- **Divide and conquer (partitioning):** divide problem into smaller pieces, so that each piece can be solved separately
  - but rarely can pieces be completely independent
  - they rather cooperate to achieve goal
  - so software design is typically *hierarchical*: understanding can be deepened as necessary
    - at the bottom level, code units are functions (maybe a couple dozen LoC)
    - at the middle level, code units are modules (maybe a couple dozen functions)
    - at higher levels, code unit is package (at most a couple dozen modules)

Design principles

- **Abstraction:** describe the external behavior of a module, not the internal details that produce the behavior
  - design proceeds from external behavior to internal details
  - great design leaves behind documentation of abstractions (more on that next class)
  - in absence of that, understanding a system involves reinventing the abstractions (be what they may)

Abstraction of the Camel



Abstraction

- Forgetting information
- Treating different things as though they were the same

e.g., animal kingdom

e.g., files vs. block devices, inodes

e.g., high-level programming languages vs. machine instruction set

e.g., floating point arithmetic vs. idealized math

Computational Thinking

Jeanette Wing
Corporate VP, MSR

- *Computational thinking is using abstraction and decomposition when... designing a large, complex system.*
- *Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction.*

https://www.cs.cmu.edu/~15110-s13/Wing06-ct.pdf
http://research.microsoft.com/apps/video/default.aspx?id=179285

Design principles

- **Modularity:** modules are separate
  - design of one module should be able to proceed with only abstractions of other modules
  - changes to a module don't require changes to other modules (even recompilation)
  - *separation of concerns:* implement, maintain, reuse modules independently
  - roughly, modularity = partitioning + abstraction

Design strategies

- **Top down:**
  - start at top, most abstract level of hierarchy
  - proceed downwards, adding more detail to design as you deepen: *stepwise refinement*
  - eventually reach concrete enough design that it can be implemented
- **Bottom up:**
  - start at bottom, most concrete level
  - proceed upwards, creating *layers of abstraction*
  - eventually reach powerful enough modules that they implement the desired system
- In practice, these are nearly always **combined**
  - Design new modules from top down
  - But keeping in mind existing libraries that can be built on from the bottom up

Top down vs. bottom up

- Advantages of **top down**:
  - get high-level design right
  - easier to design abstractions
- Disadvantages of **top down**:
  - harder to test until program is complete
- Advantages of **bottom up**:
  - get low-level implementation right
  - always have testable code
- Disadvantages of **bottom up**:
  - large-scale design flaws don't show up until too late

Upcoming events

- [today] A2 due (soft deadline)
- [Saturday] end of automatic 48 hour extension on A2 (hard deadline)

This is good design.

THIS IS 3110

Acknowledgment

Parts of this lecture are based on this book:

Pankaj Jalote. *An Integrated Approach to Software Engineering*, third edition. Springer, 2005.