Transition: analysis to design

Now know “what”
Time to focus on “how”

Decided what to do
Might as well do it right

System design

- Goal, in general: solve the problem
  - Goal of OOD: convert OOA results into something that can be implemented
    - e.g, as software (and/or hardware, services, …)
- Key considerations (a.k.a. tradeoffs):
  - Cost-effectiveness of solution vs. design/coding effort
    - Can reduce effort by applying patterns, idioms, 3rd party, …
  - Reusability – *maybe* worth investing effort in
    - Could save lots of effort later
    - But can overly complicate a simple problem if overdone

Design in practice

- No “cookbook” method – no “right” way
  – But have some basic principles for guidance
  – And have a growing knowledge base on patterns
- Is an exercise in problem solving, so attack using the usual strategies
  – Divide/conquer – solve sub-problems to solve whole
  – Top-down approach, with stepwise refinements
- Unlike analysis – leave room for creativity
  – Concentration → incubation → inspiration

“Deep dive”

Design activities

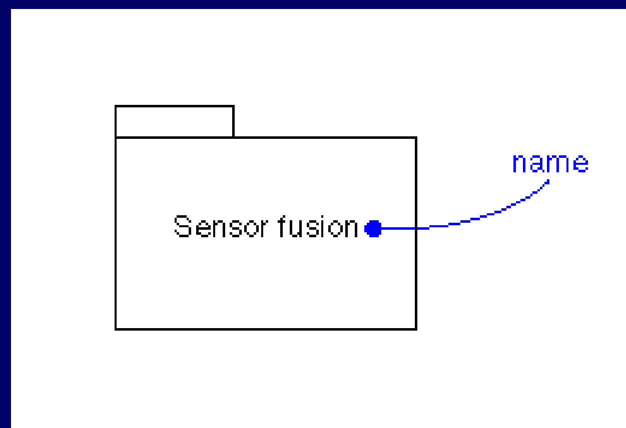
- Consider “real” use cases
  - Sharpen focus to actual technology, specific user interfaces, particular other systems, …
- Package coherent subsystems together
  - And organize the packages into overall system architecture
- Model the interactions between objects
  - Including interactions between packages
- See assignment 3, part 4

System architecture

- High-level descriptions of the system
  - Broad focus on significant structural elements
    - Subsystems, packages, interfaces to other systems
    - At a level of detail all developers and stakeholders can follow
- Often need separate descriptions for various views:
  - Use case views, deployment views, design views, …
    - Design views required for CS 50 project
- Many basic architecture types – vary by purpose
  - Pipes & Filters – for flexibility without user interaction
  - Repository – favor big data storage-retrieval systems
  - Layers ("object-oriented architecture") – most used now

Diagramming packages

- Groups of classes – good for architectural modeling
  - Abstraction benefit: lots of concepts modeled as one
    - A handy way to "divide and conquer" the problem

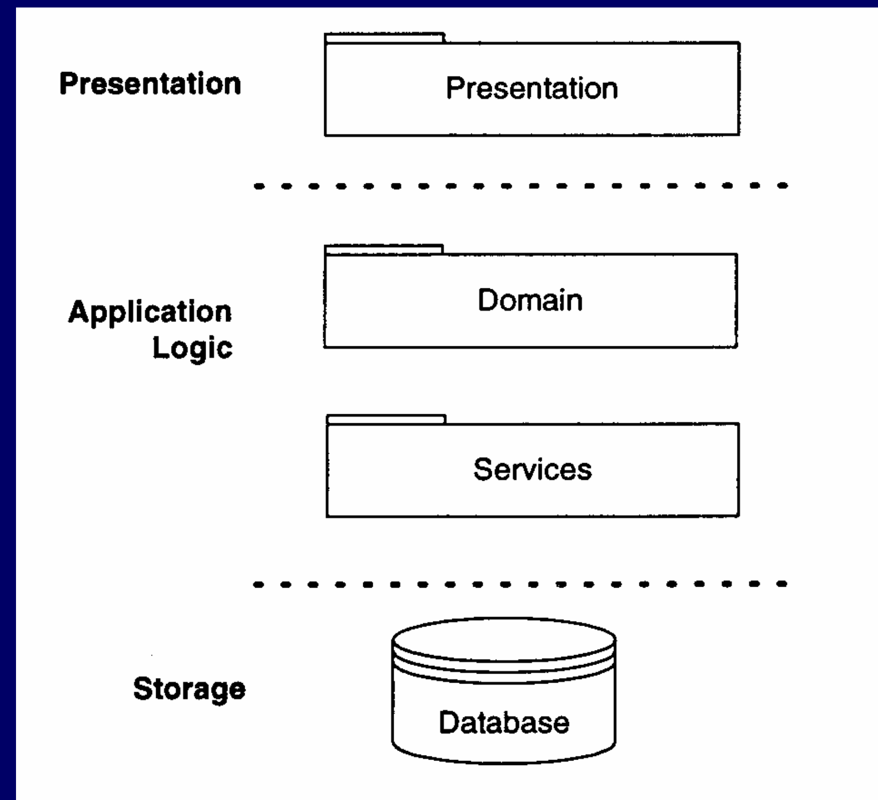


UML package symbolism

  - Idea is to separate functional subsystems
    - Many associations among classes in same package
    - Few associations between packages
  - Side benefit: team members can split work by packages
    - Works best with "clean" interfaces

Basic 3-tier architecture

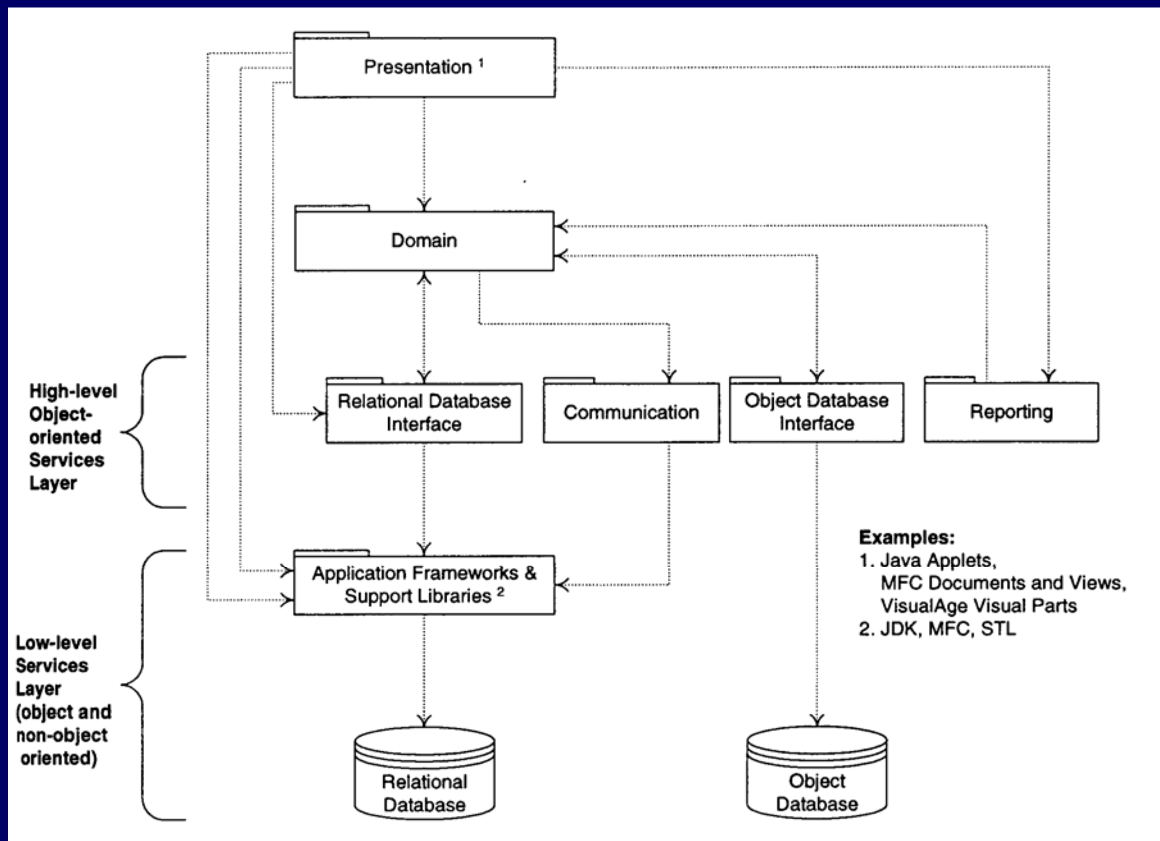
- Can have many layers, but 3 are basic:
  1. Presentation layer – windows, reports, GUIs
  2. Application logic layer – domain, object services
  3. Storage layer – persistent data, basic services



About layered architectures

- Concept – each layer is a base for implementing layers above it
  - Ideally, knowledge and contact is one-way: down ↓
  - Lower layers should not even know layers above
- Lots of good reasons to use layers
  - Reduce complexity – separate the domain from the implementation as much as possible
  - Increase modifiability, and reuse potential
  - Easy to plug in off-the-shelf and 3rd party stuff

Data services sub-layers



- Goal: insulate domain classes from storage details
- How? – interface classes
- Note: often start design by choosing services (inc. software and hardware choices)

Storage and network layer(s)

- The lowest and *least coupled* layers
- 3 main types of storage layer
  - 1. Object database
    - Most abstract, so easiest to adapt (high level access)
  - 2. Relational database
    - Mid-level access (records ←→ objects) – need an interface
  - 3. Do-it-yourself file schemes – lowest level access
- Similar breakdown for network layer types
- Best to decide early
  - And whether to buy or build new, adapt old, …

Separating models and views

- Basic principle: domain (model) never directly contacts the presentation (view)
  - But is ready to answer requests from the view
  - Or can contact indirectly by “broadcasting”
    - See publish-subscribe pattern, text p. 463 (a.k.a., Observer)
- Related idea: view should not control the domain
  - Okay for GUI to signal an event
    - As long as model takes over after that
  - Often best to contact through a mediator, an “application coordinator”

Model-view separation benefits

- Reuse model with different views
- Maybe reuse view with different models
- Have multiple views of the same model
  - Even simultaneously!
  - e.g., view model from many angles
- Side benefit – complexity management
  - Reflects a recurring OOP theme – encapsulation (a.k.a. information hiding)
  - Benefit here – don't have to worry about display while working on the model

Application logic layer partitions

- Partition by logical units (organize as packages)
  - Refer to collaborations on CRC cards – look for:
    - Minimal coupling between packages (few collaborations)
    - Highly cohesive within packages (many collaborations)
- Teamwork benefit too
  - Agree on package interfaces – then split up the work
- CS 50: at least split domain from service classes
  - e.g., report generators, database interfaces, offscreen graphics builders, …