software architecture

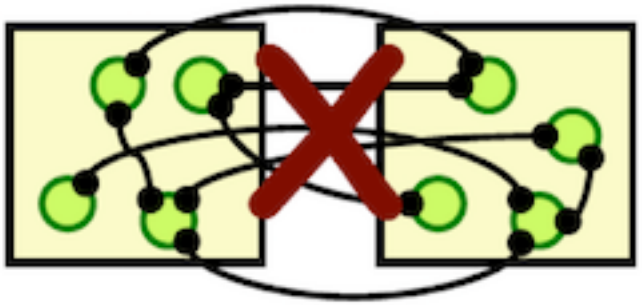
showing the architecture

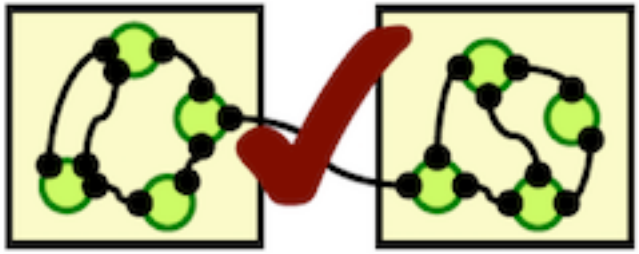
- coupling and cohesion
- uml package diagrams
- software architecture styles
  - layered architectures
  - pipe-&-filter
  - object-oriented architecture
  - implicit invocation
  - repositories

coupling & cohesion

- architectural building blocks

- a good architecture:
  - minimizes **coupling** between modules
    - goal: modules don't need to know much about one another to interact
    - low coupling makes future changes easier
  - maximizes the **cohesion** of each module
    - goal: the contents of each module are strongluy inter-related
    - high cohesion means the subcomponents really do belong together

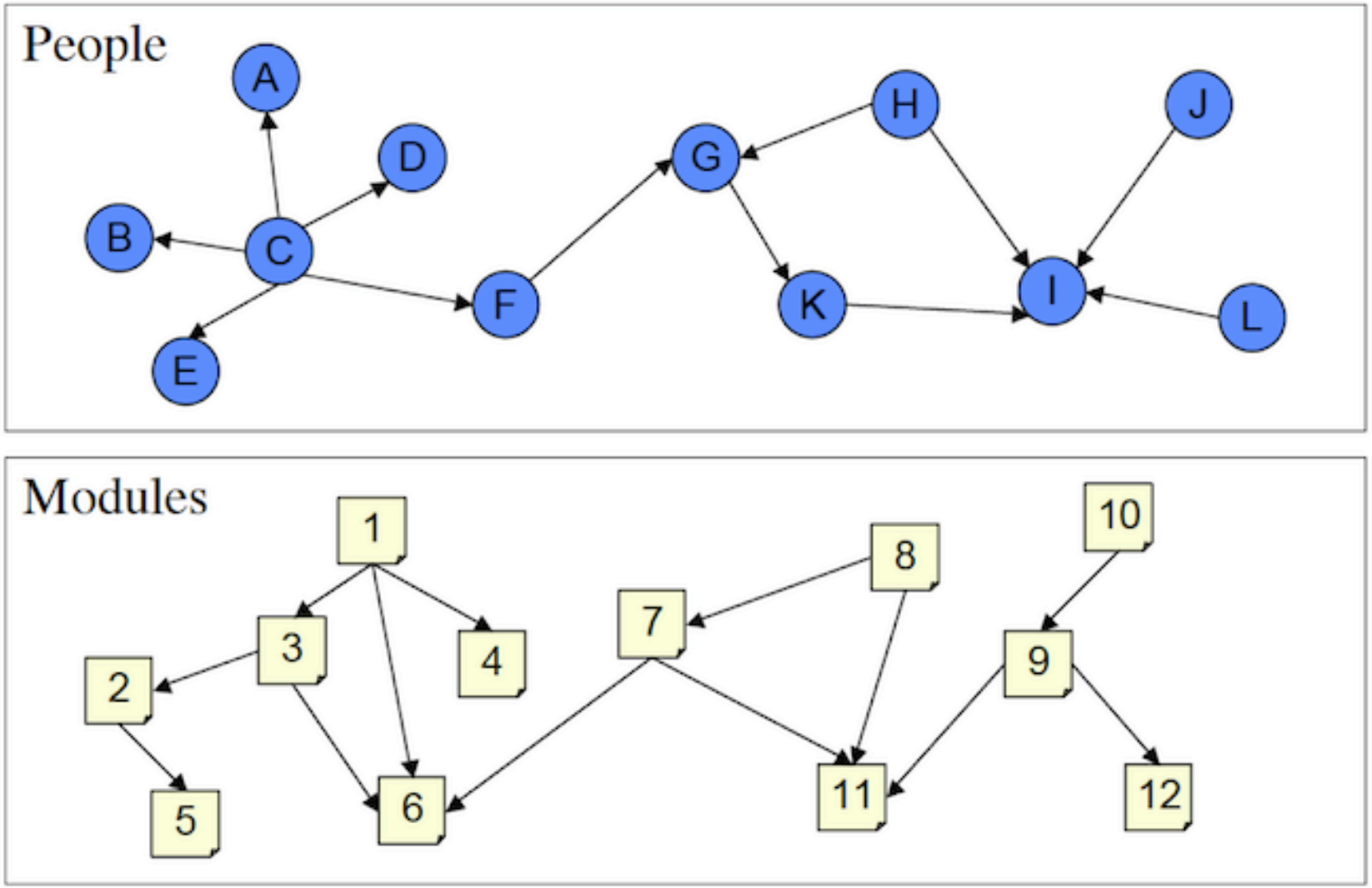




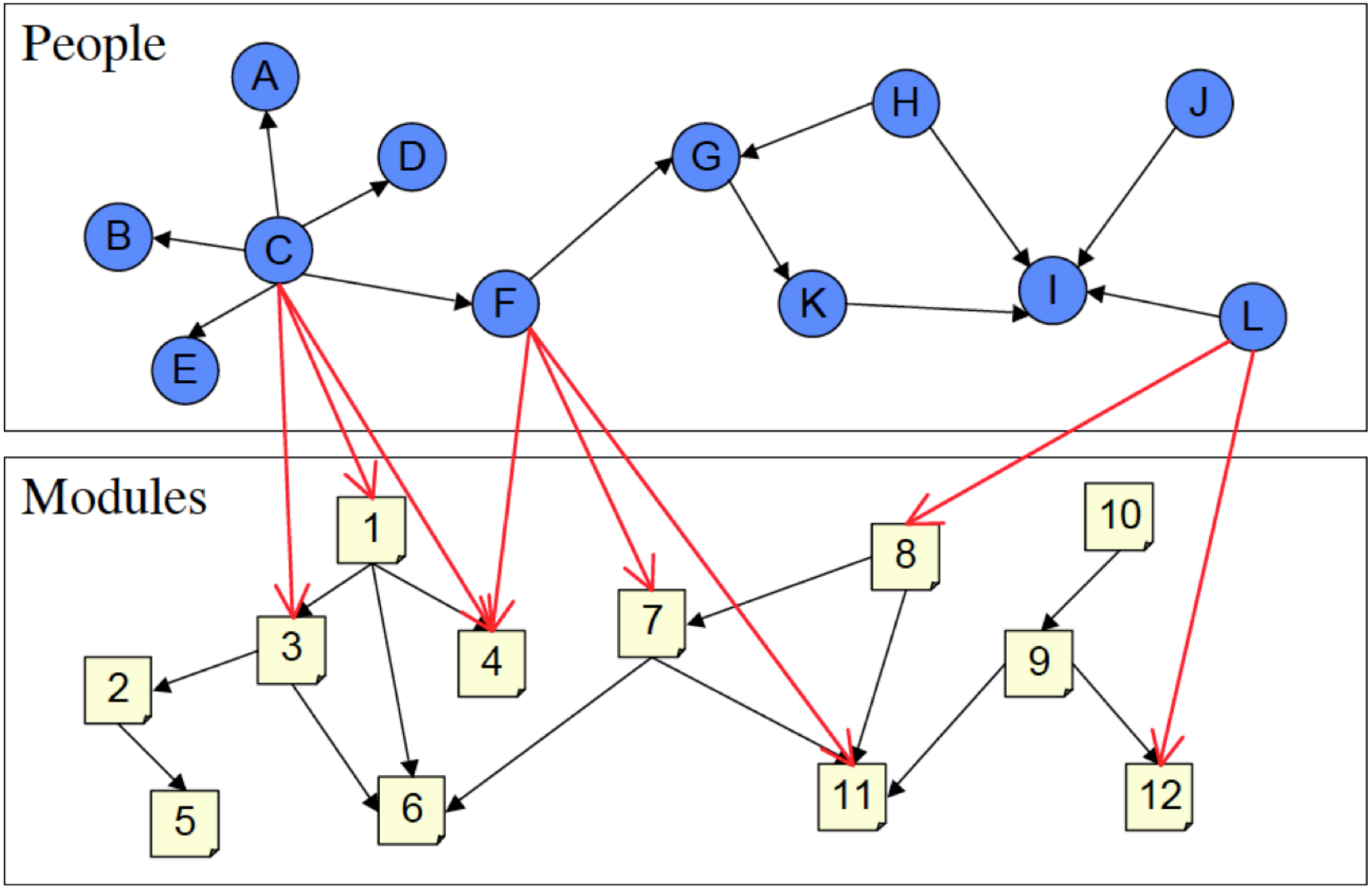
Conway's law

"The structure of a software system reflects the structure of the organization that built it"

socio-technical congruence

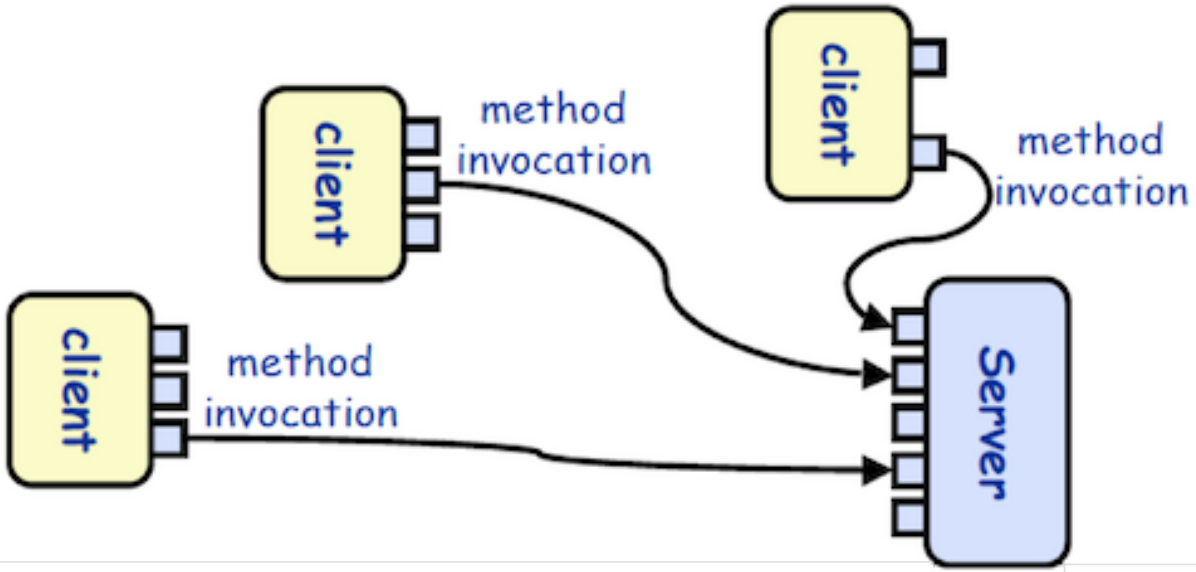


socio-technical congruence (2)

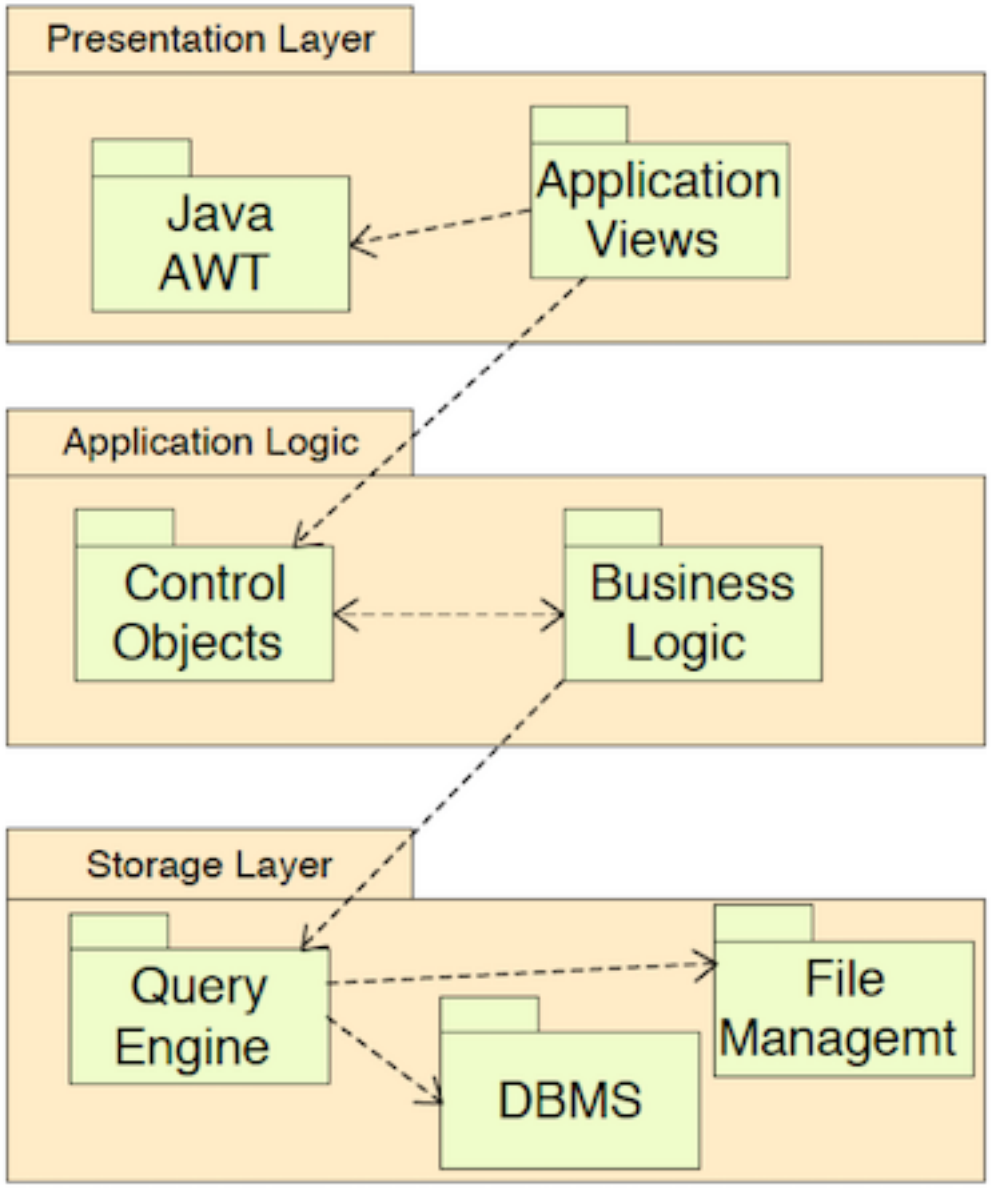


software architecture

- a software architecture defines:
  - the components of the software system
  - how the components use each others functionality and data
  - how control is managed between the components
- an example: client-server
  - servers provide some kind of service; clients request and use the service(s)
  - reduced coupling: servers don't need to know what clients are out there



example: 3-layer architecture



uml packages

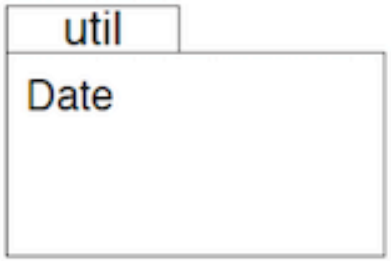
- we need to represent our architectures
  - uml elements can be grouped together in packages – elements may be:
    - other packages (representing subsystems/ modules)
    - classes
    - models (ex. use case models, interaction diagrams, statechart diagrams, etc.)
  - each element of a uml model is owned by a single package

uml packages (2)

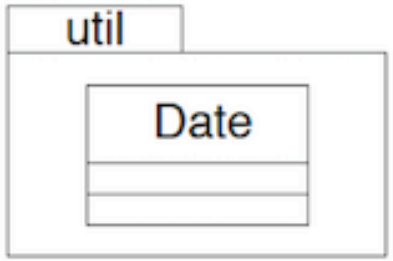
- criteria for decomposing a system into packages:
  - different owners
    - who is responsible for working on which diagrams
  - different applications
    - each problem has its own obvious partitions
  - clusters of classes with strong cohesion
    - ex. course, course description, instructor, student, …
  - or, use an architectural pattern to help find a suitable decomposition

package notation

named package

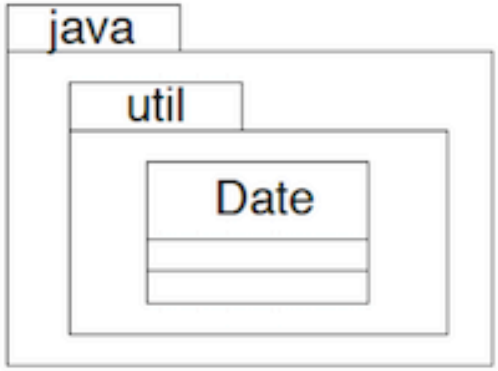


package with list
of contained classes

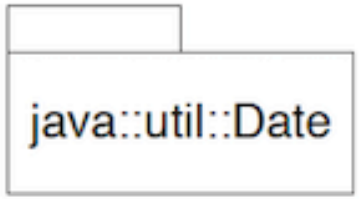


package containing
a class diagram

package with
qualified name

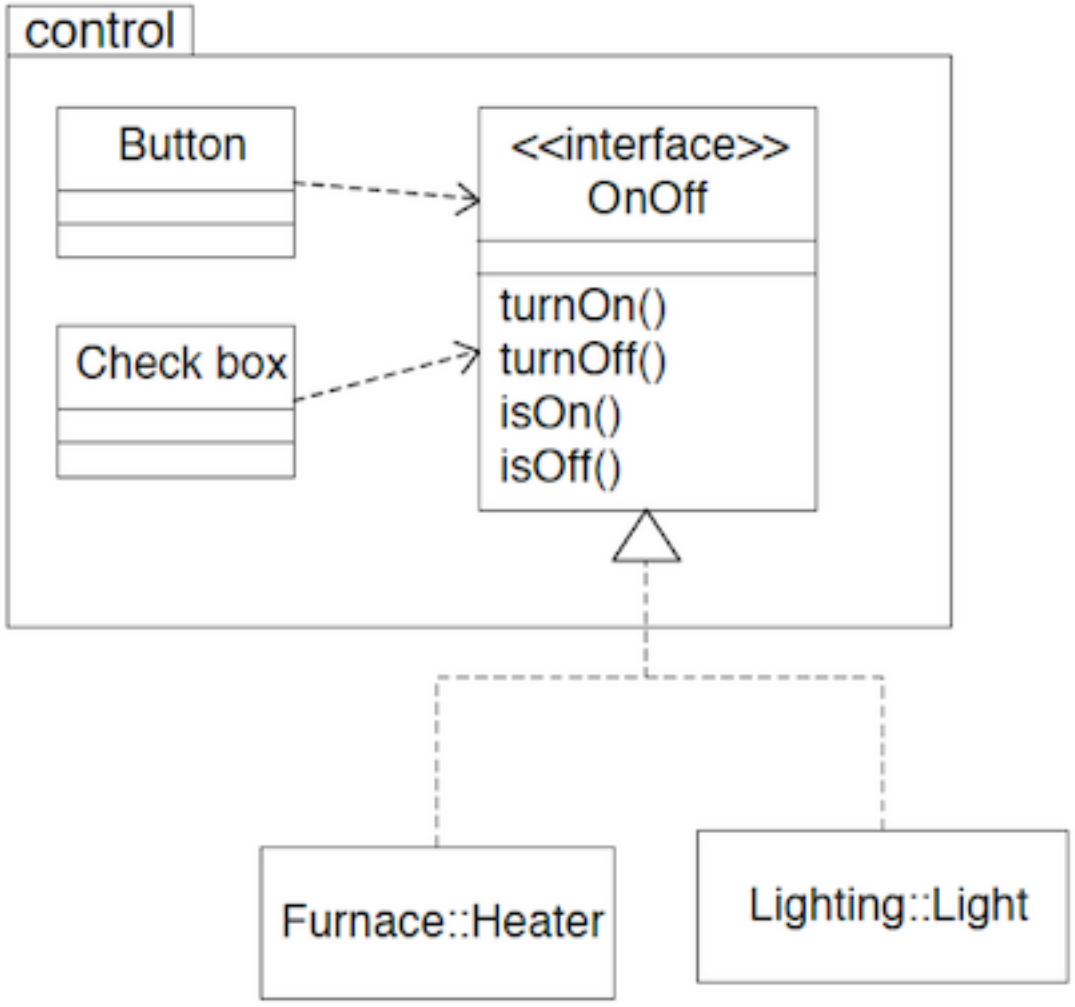


nested packages

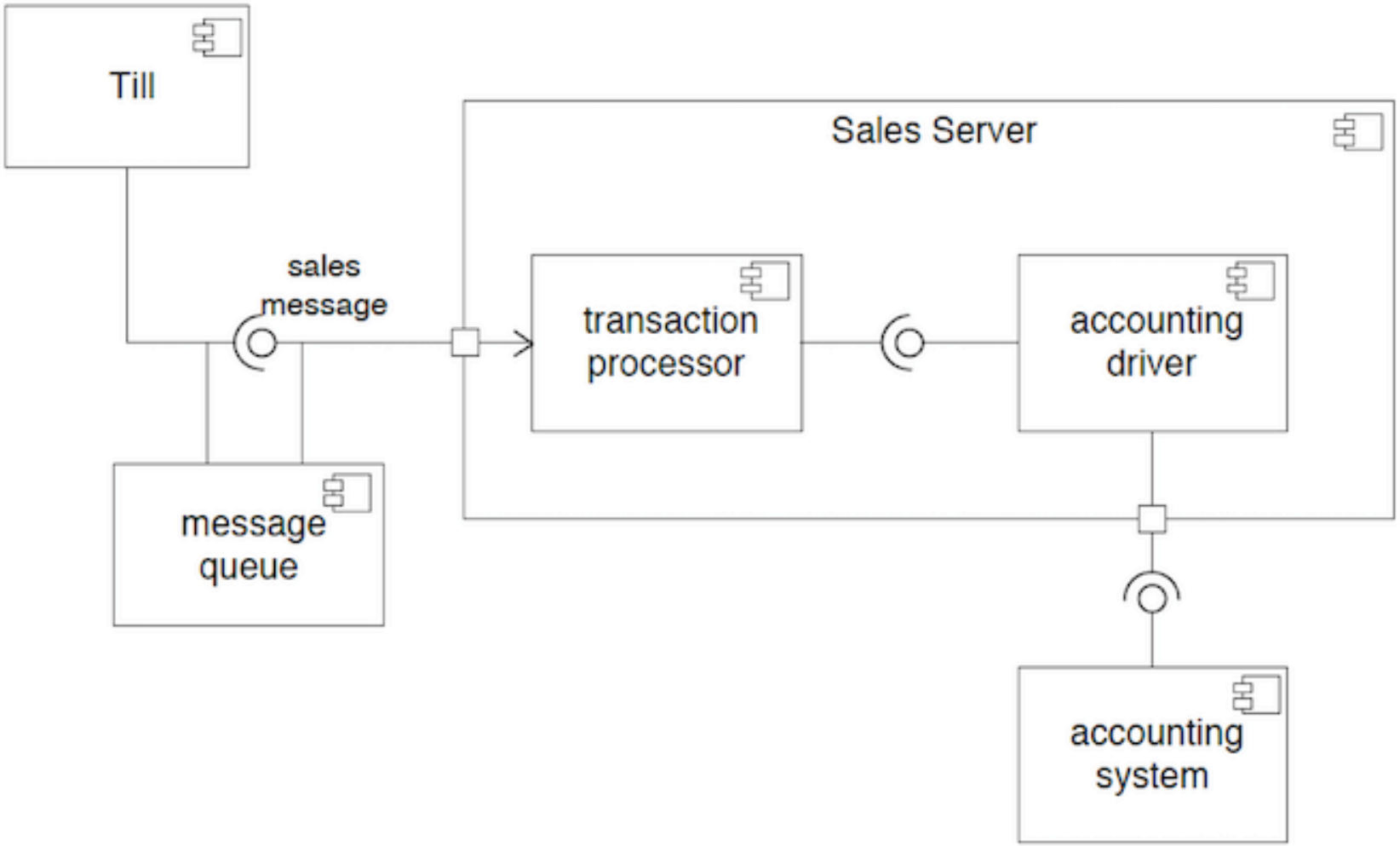


package with
fully qualified name

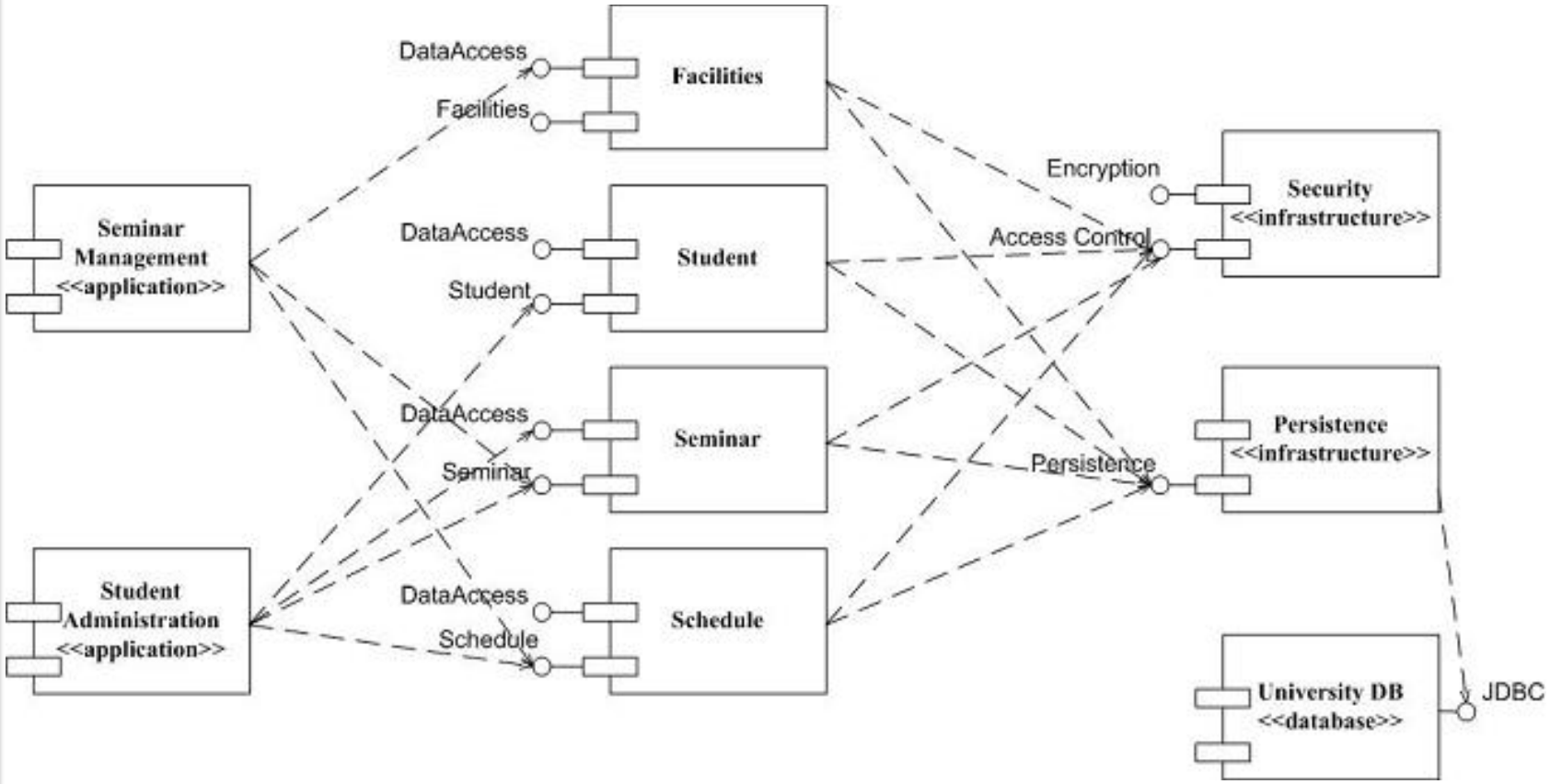
towards component-based design



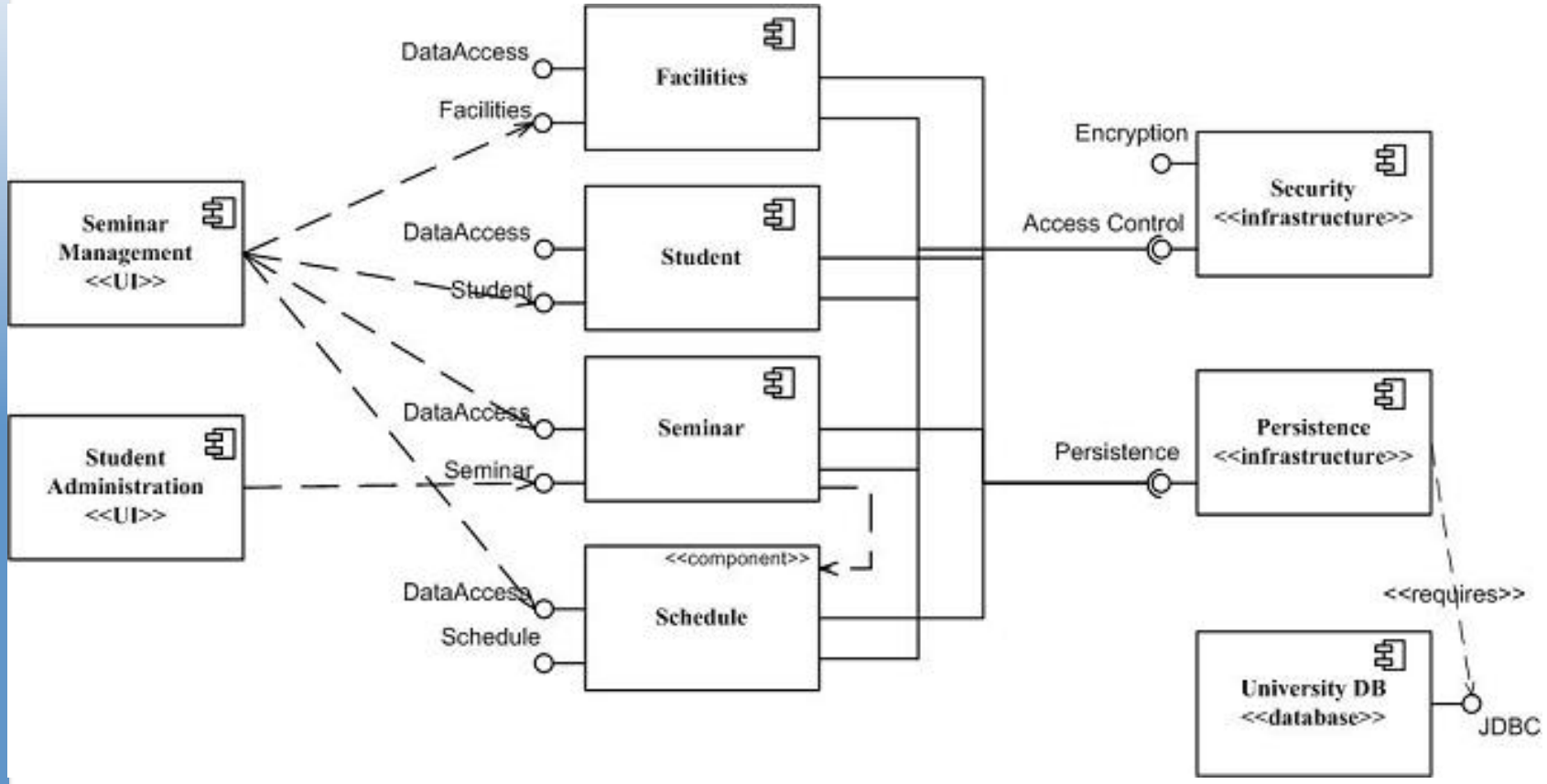
or, use component diagrams



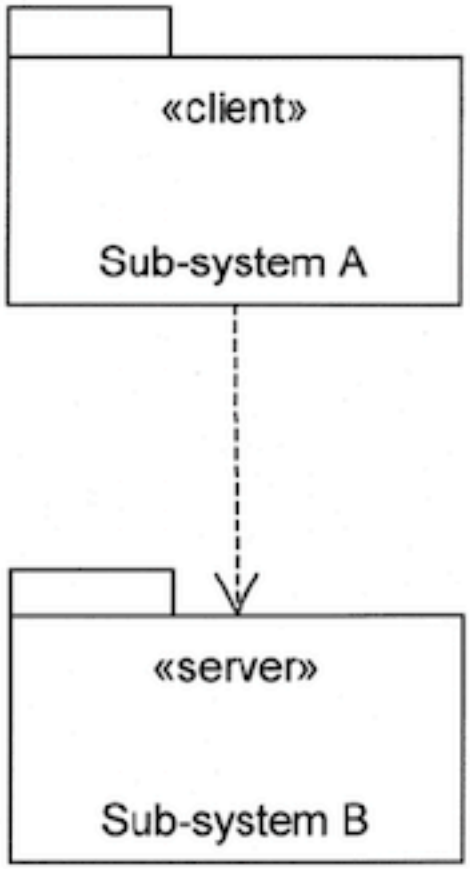
UML v1 component diagram



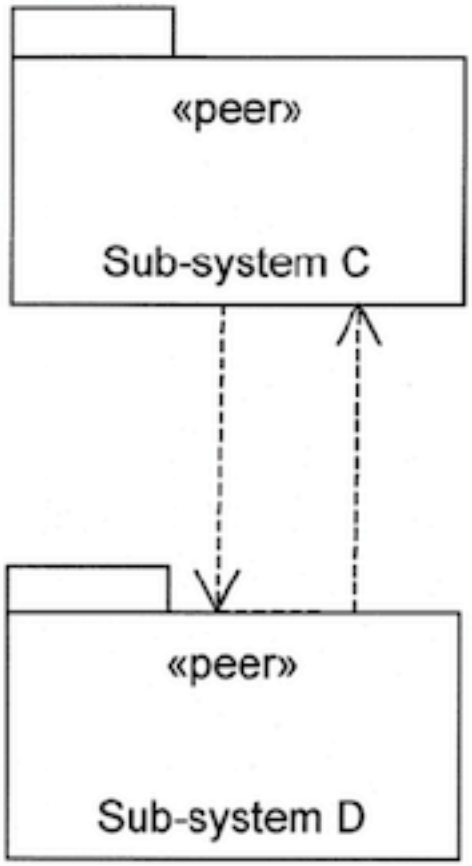
UML v2 component diagram



avoid dependency cycles



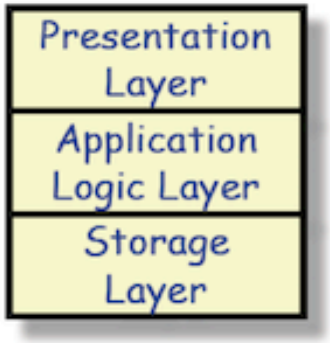
The server sub-system does not depend on the client sub-system and is not affected by changes to the client's interface.

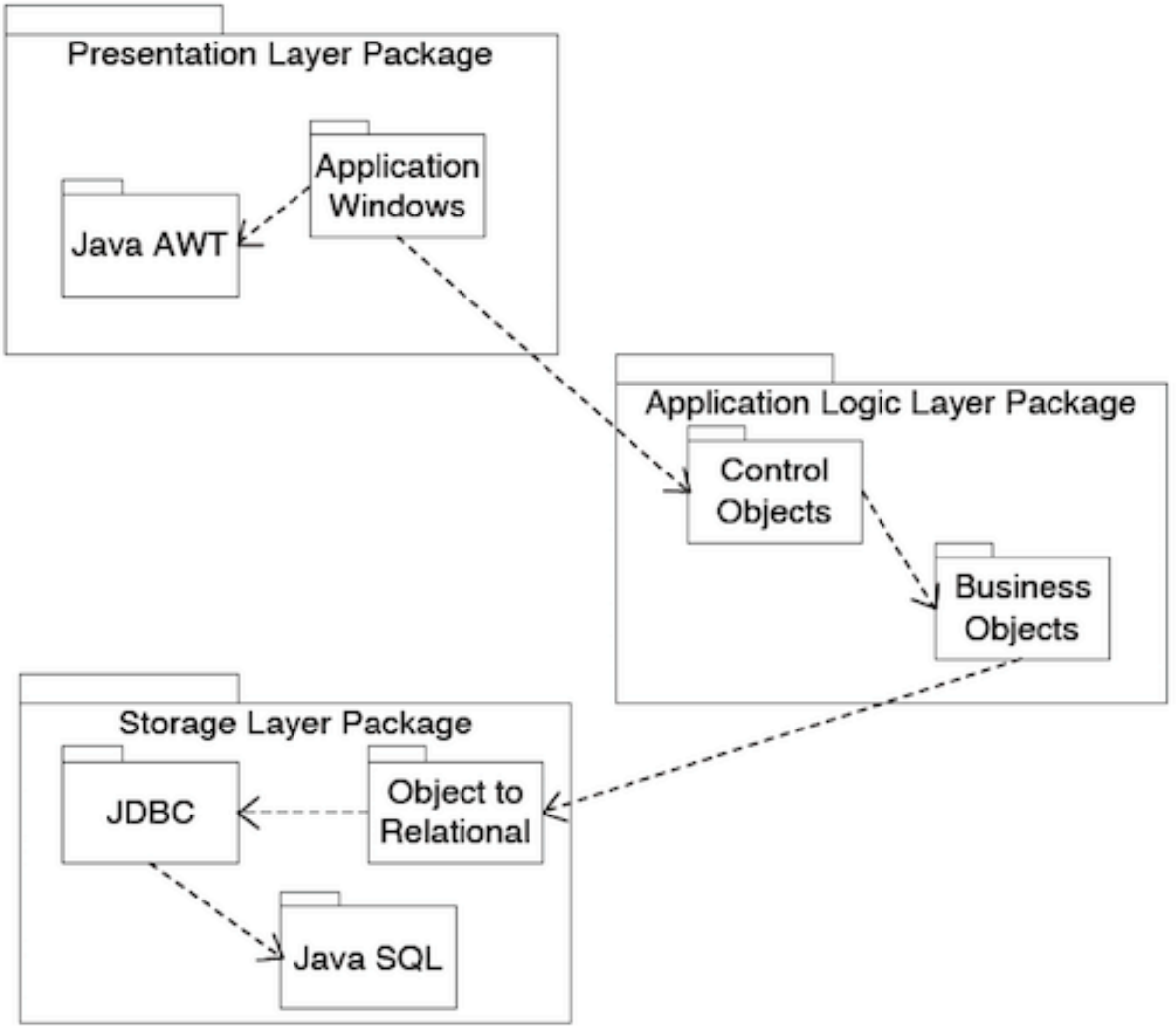


Each peer sub-system depends on the other and each is affected by changes in the other's interface.

architectural patterns

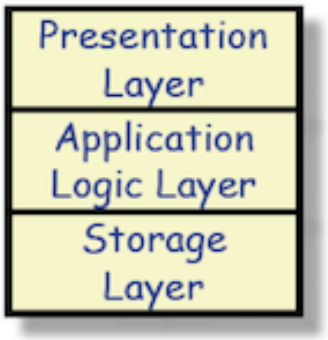
E.g. 3 layer architecture:

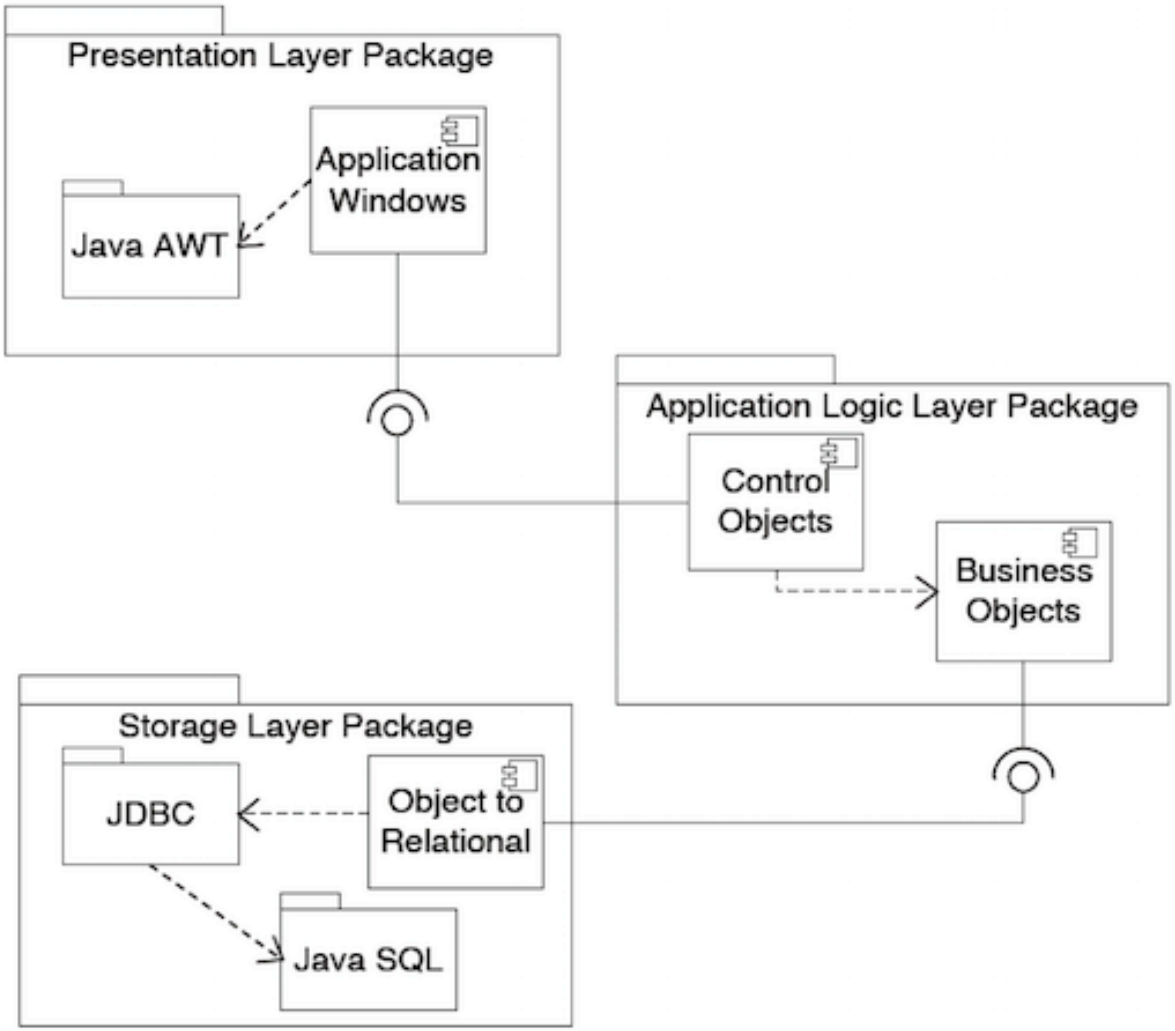




or, to show the interfaces…

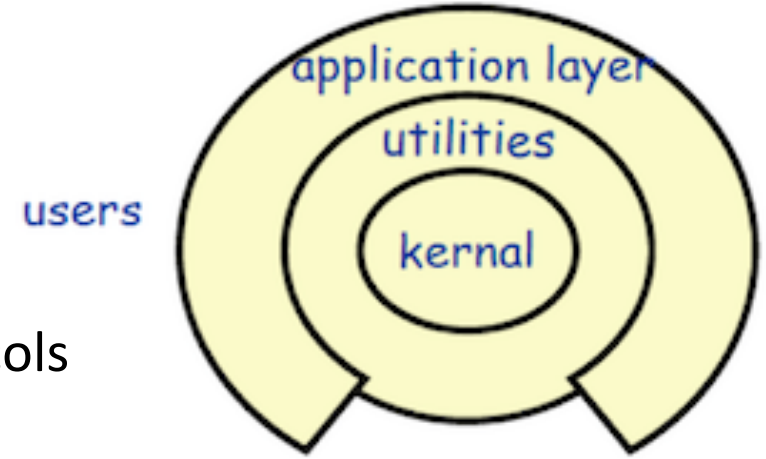
E.g. 3 layer architecture:





layered systems

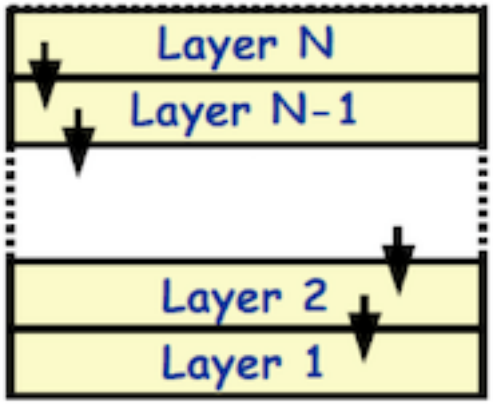
Source: Adapted from Shaw & Garlan 1996, p25. See also van Vliet, 1999, p281.



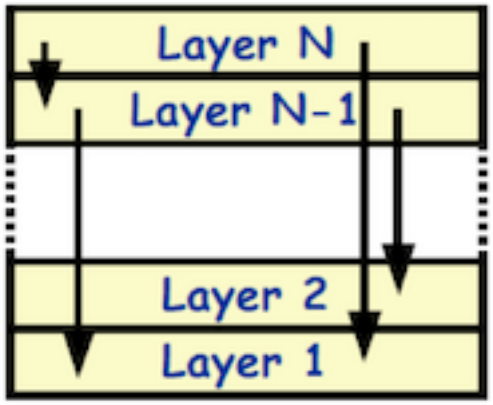
- examples:
  - operating systems
  - communications protpcols
- interesting properties:
  - support increasing levels of abstraction during design
  - support enhancement (add functionality) and re-use
  - can define standard layer interfaces
- disadvantages:
  - may not be able to identify clean layers

open vs. closed layered arch.

- closed architecture
  - each layer only uses services of the layer immediately below
  - minimizes dependencies between layers & reduces impact of change

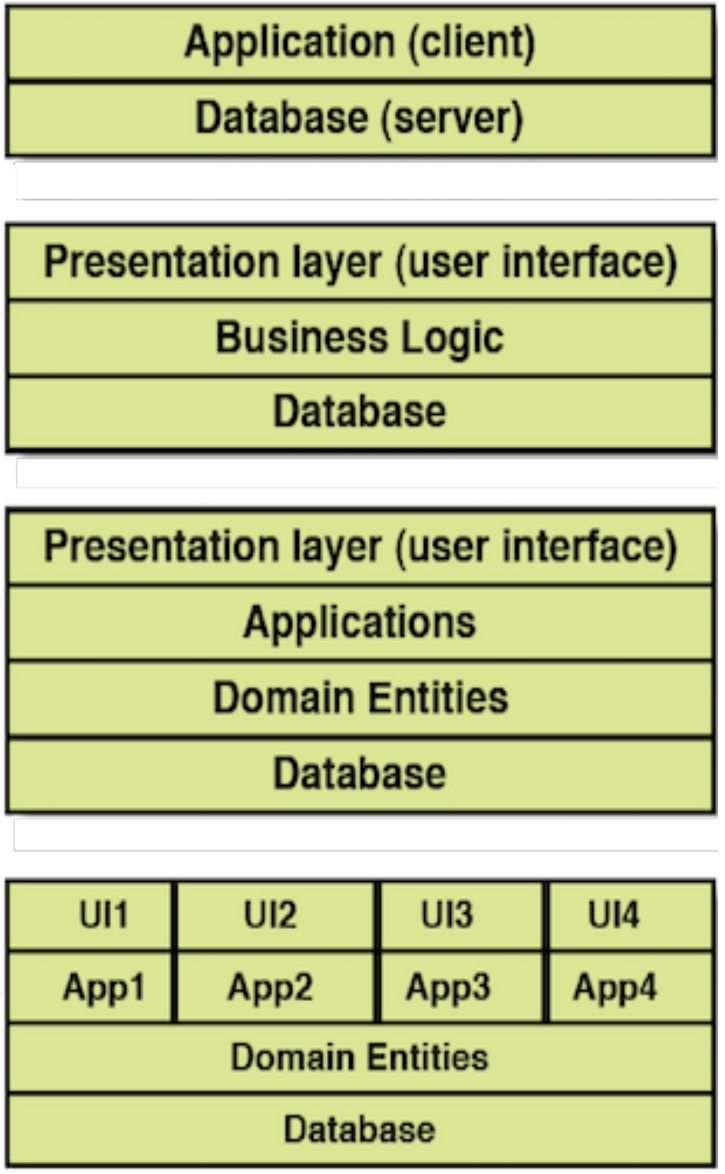


- open architecture
  - a layer can use services from any lower layer
  - more compact code, as services of lower layers can be access directly
  - breaks encapsulation of layers, so increases dependences between layers

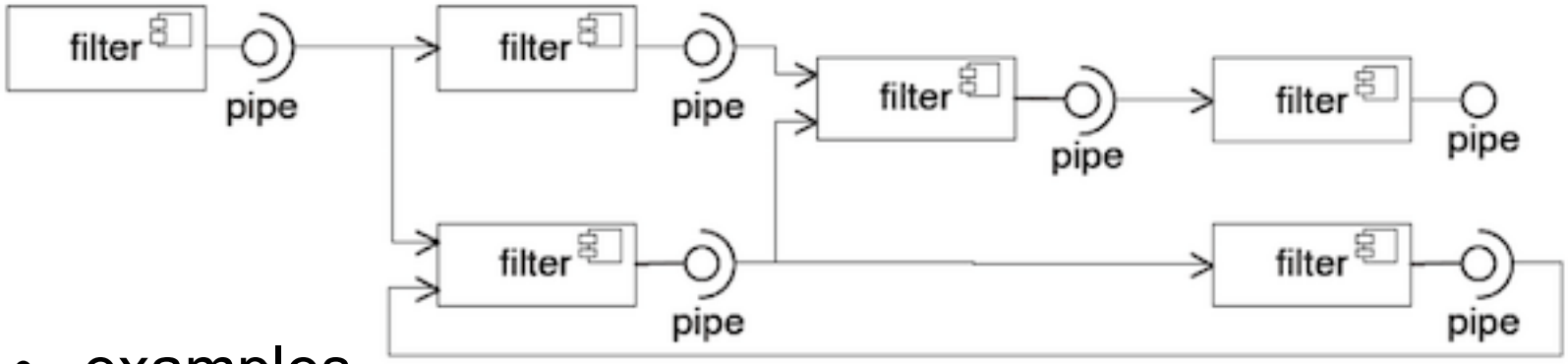


how many layers?

- 2 layers
  - application layer
  - database layer
  - ex. simple client-server
- 3 layers
  - separate out business logic
    - makes UI & DB layers modifiable
- 4 layers
  - separate application from domain
    - boundary classes in presentation layer
    - control classes in application layer
    - entity classes in domain layer
- partitioned 4 layer
  - identify separate applications

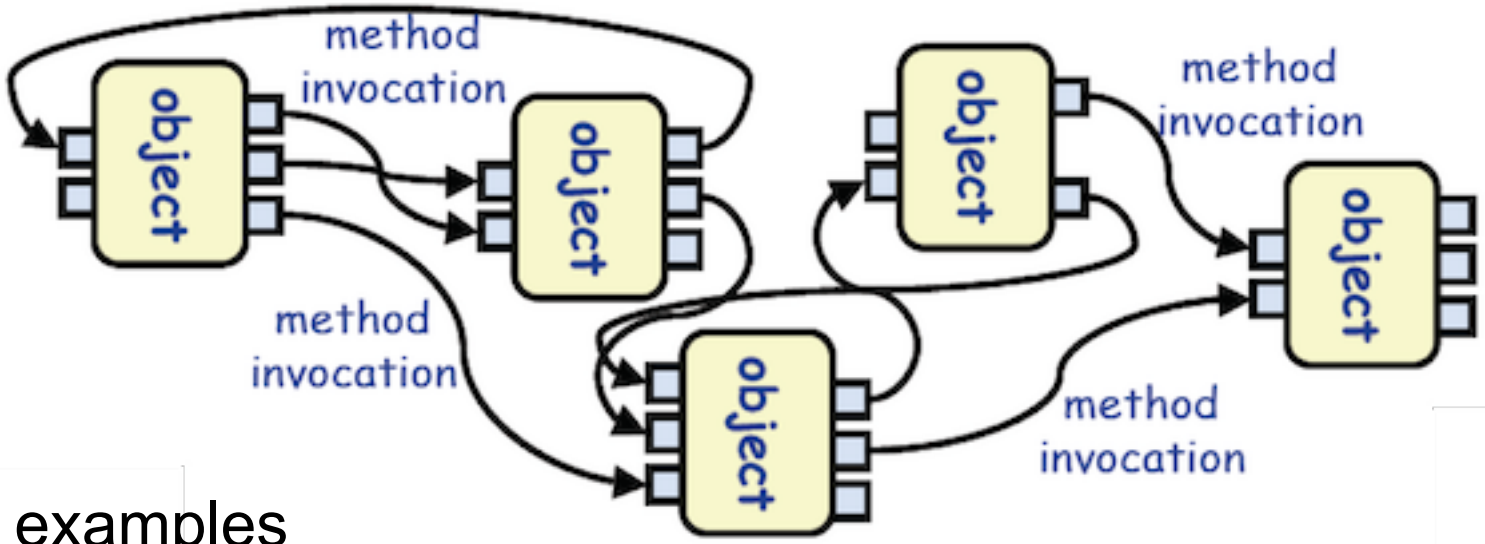


pipe & filter



- examples
  - unix shell scripts
  - compilers
    - lexical analysis -> parsing -> semantic analysis -> optimization (optional) -> code generation
  - signal processing
- interesting properties
  - filters don't need to know anything about what they are connected to
  - filters may be able to be implemented in parallel
  - behaviour of the system is a composition of behaviour of the filters
    - specialized analysis, such as deadlock and throughput, are possible

object oriented architectures

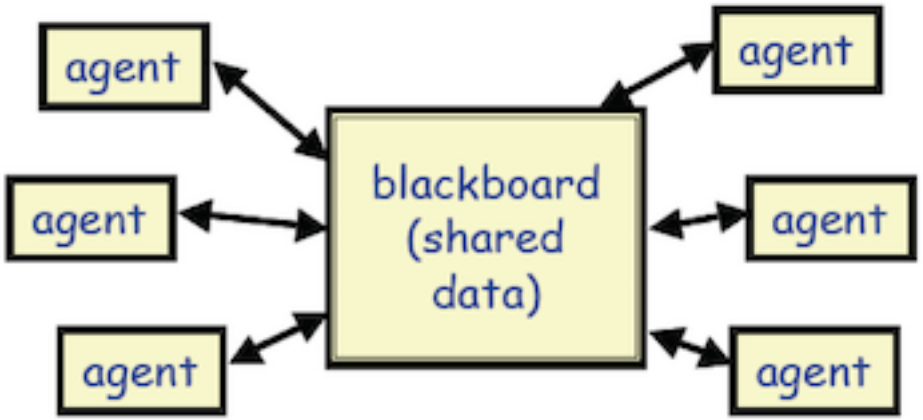


- examples
  - abstract data types
- interesting properties
  - data hiding (internal representation not visible to clients)
  - decompose into set of interacting agents
  - multi-threaded or single thread
- disadvantages
  - objects must know the identify of objects they interact with

event based (implicit invocation)

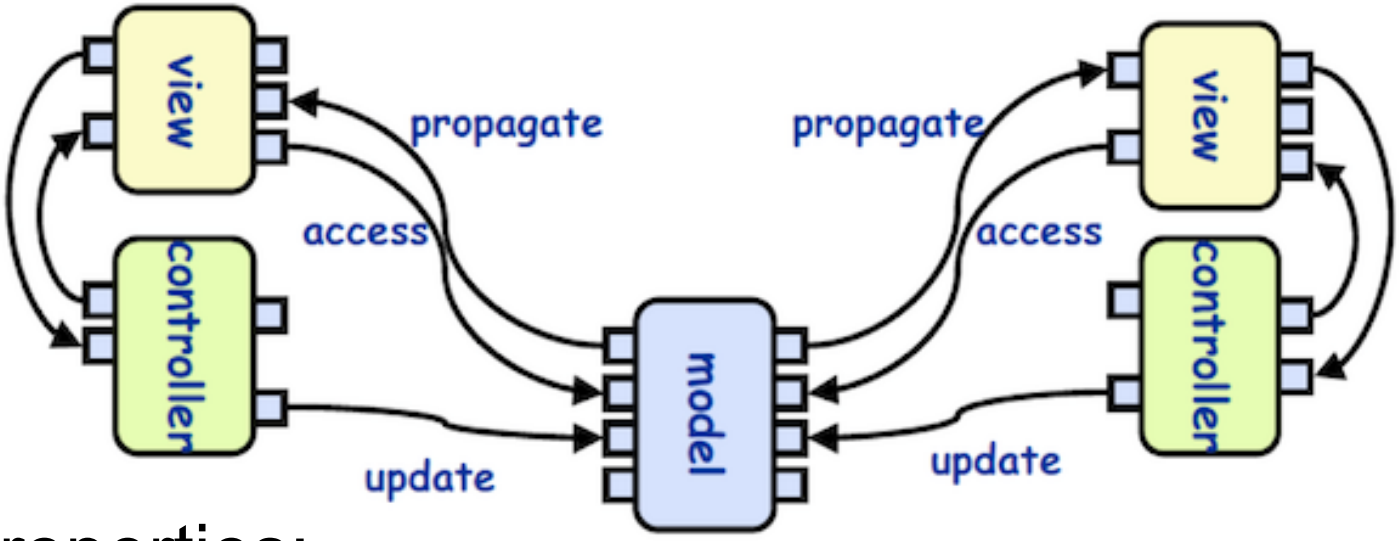
- examples
  - debugging systems (listening for breakpoints)
  - DBMS checking RI, firing triggers
  - GUI
  - publish/subscribe
- interesting properties
  - announcers of events don't need to know who will handle the event
  - supports re-use and evolution of systems (easy to add new agents)
- disadvantages
  - components have not control over ordering of computations

repositories



- examples
  - databases
  - blackboard expert systems
  - programming environments
- interesting properties
  - can choose where control lies (agents, blackboard, both)
  - reduce need to duplicate complex data
- disadvantages
  - bottleneck

model-view-controller



- properties:
  - one central model, many views (viewers)
  - each view has an associated controller
  - the controller handles updates from the user of the view
  - changes to the model are propagated to all views

summary

- avoid unnecessary coupling & cohesion
- if a layered approach, what are the layers? what goes in each
  - following a pattern like MVC, MVP?
- modularize for reusability (well designed public interface)
- uml diagrams for discussing architecture
  - adherence to uml syntax is not the point
  - clearly communicating the architecture is the point

summary (2)

"Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher." – Antoine de Saint Exupéry, Terre des Hommes, 1939

(my) translation: *"perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away"*

summary (3)

- uml books