



Lecture 4: Showing the architecture

- Coupling and Cohesion
- UML Package Diagrams
- Software Architectural Styles:
 - ↳ Layered Architectures
 - ↳ Pipe-and-filter
 - ↳ Object Oriented Architecture
 - ↳ Implicit Invocation
 - ↳ Repositories



Coupling and Cohesion

Architectural Building blocks:



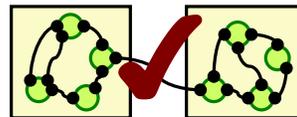
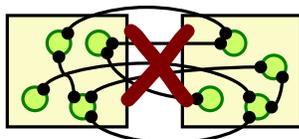
A good architecture:

Minimizes coupling between modules:

Goal: modules don't need to know much about one another to interact
Low coupling makes future change easier

Maximizes the cohesion of each module

Goal: the contents of each module are strongly inter-related
High cohesion means the subcomponents really do belong together



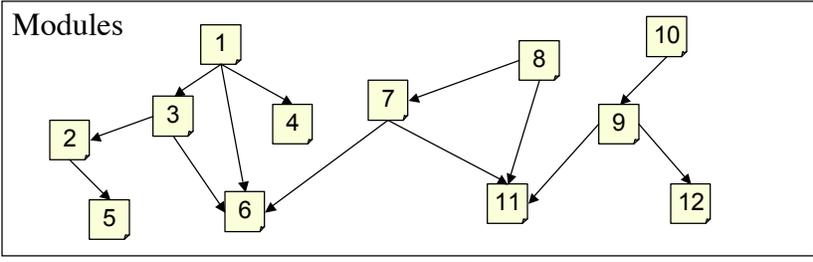
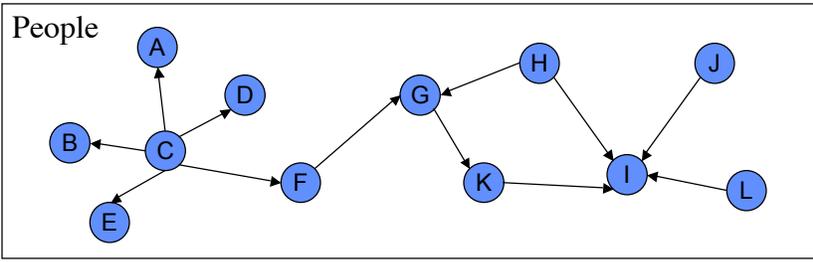


Conway's Law

“The structure of a software system reflects the structure of the organisation that built it”



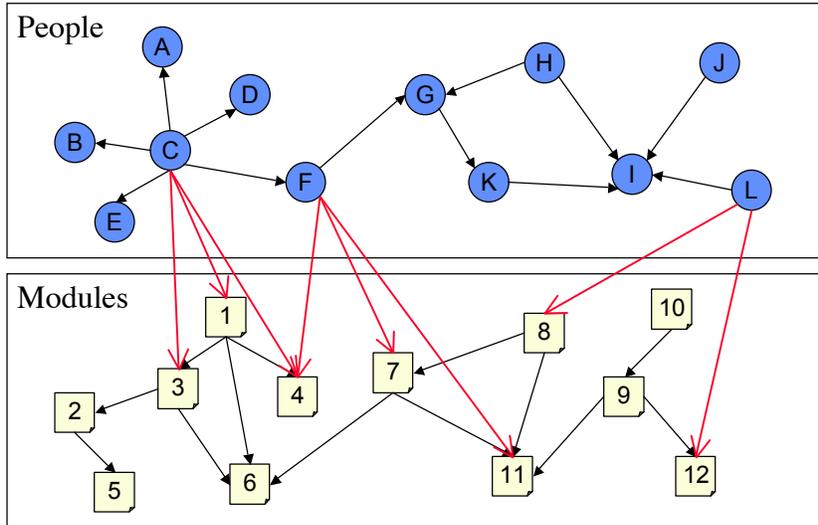
Socio-Technical Congruence



See: Valetto, et al., 2007.



Socio-Technical Congruence



See: Valetto, et al., 2007.



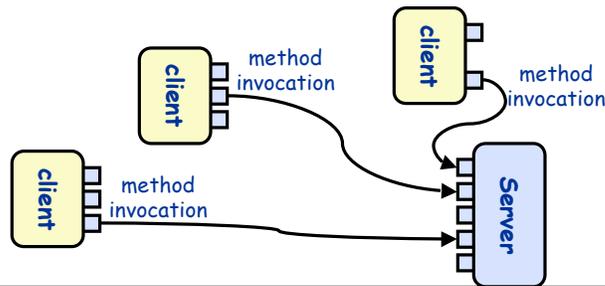
Software Architecture

A software architecture defines:

- The components of the software system
- How the components use each other's functionality and data
- How control is managed between the components

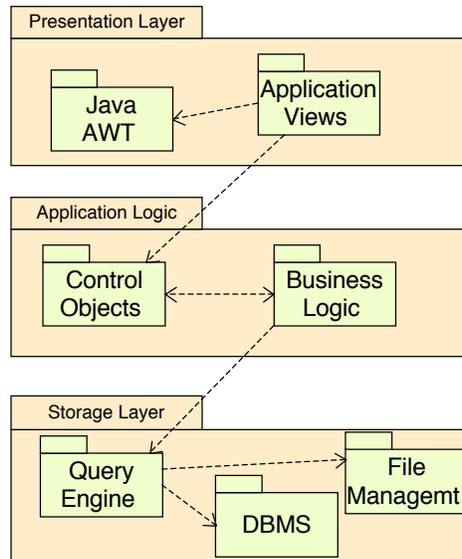
An example: client-server

- Servers provide some kind of service; clients request and use services
- 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 of a package may be:

- > other packages (representing subsystems or modules);
- > classes;
- > models (e.g. use case models, interaction diagrams, statechart diagrams, etc)

Each element of a UML model is owned by a single package

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;

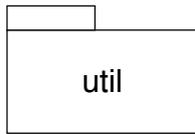
Clusters of classes with strong cohesion

e.g., course, course description, instructor, student,...

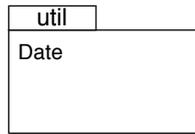
Or: use an architectural pattern to help find a suitable decomposition...



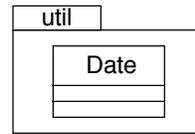
Package notation



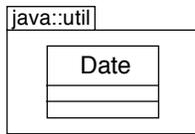
named package



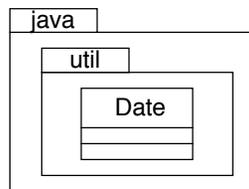
package with list of contained classes



package containing a class diagram



package with qualified name



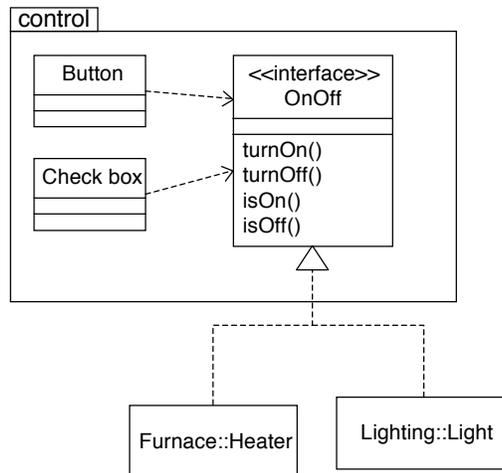
nested packages



package with fully qualified name

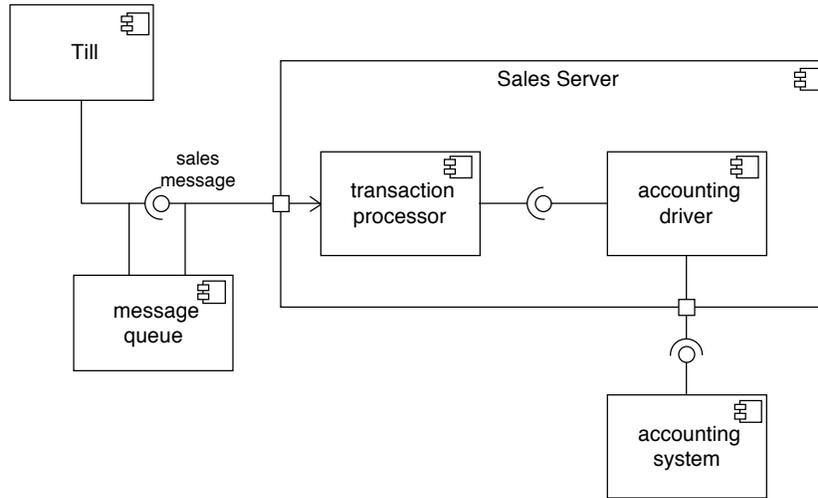


Towards component-based design

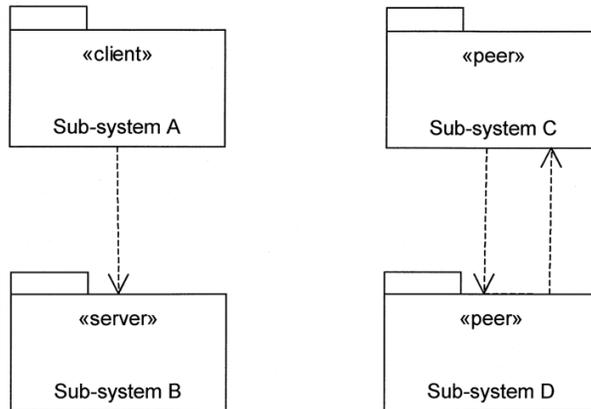




Or use Component Diagrams...



Dependency cycles (to be avoided)



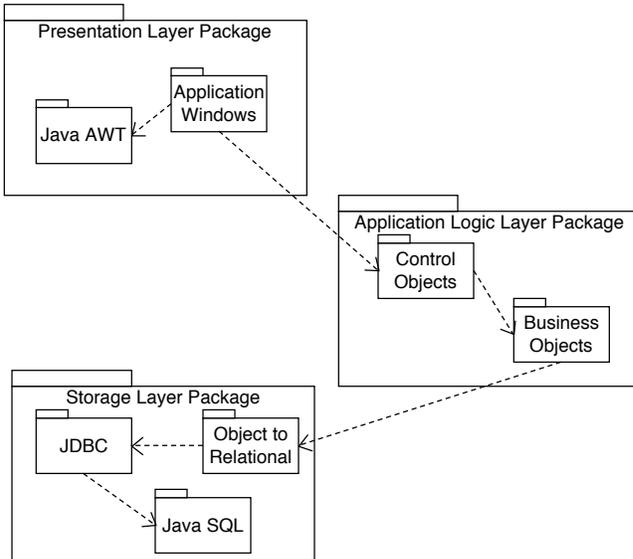
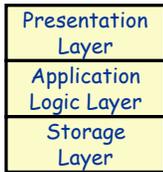
The server sub-system does not depend on the client sub-system and is not affected by changes to the client's interface.

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



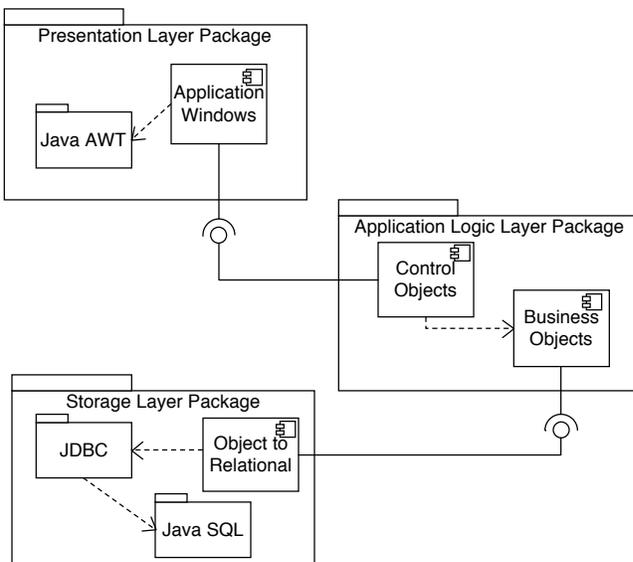
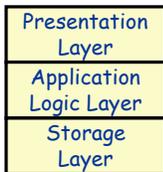
Architectural Patterns

E.g. 3 layer architecture:



Or to show the interfaces...

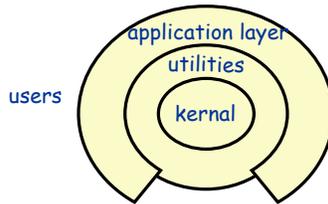
E.g. 3 layer architecture:





Layered Systems

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



Examples

- Operating Systems
- communication protocols

Interesting properties

- Support increasing levels of abstraction during design
- Support enhancement (add functionality) and re-use
- can define standard layer interfaces

Disadvantages

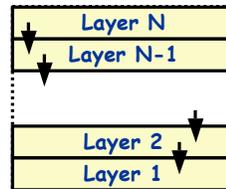
- May not be able to identify (clean) layers



Open vs. Closed Layered Architecture

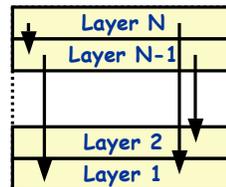
closed architecture

- Each layer only uses services of the layer immediately below;
- Minimizes dependencies between layers and reduces the impact of a change.



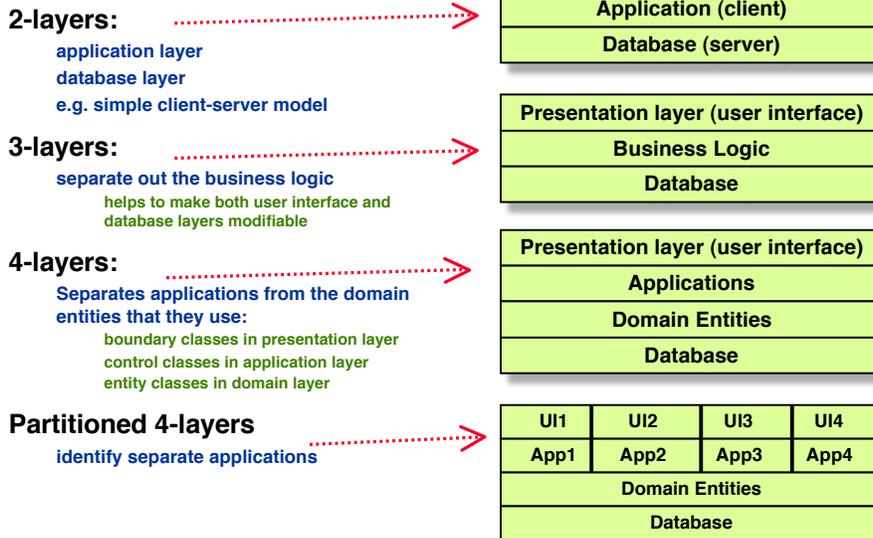
open architecture

- A layer can use services from any lower layer.
- More compact code, as the services of lower layers can be accessed directly
- Breaks the encapsulation of layers, so increase dependencies between layers



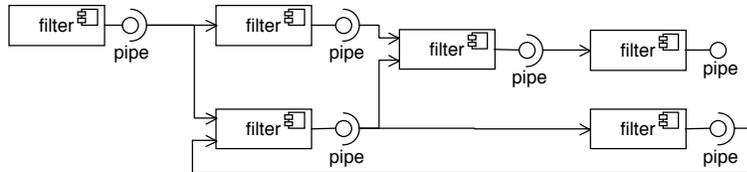


How many layers?



Pipe-and-filter

Source: Adapted from Shaw & Garlan 1996, p21-2. See also van Vliet, 1999 Pp266-7 and p279



Examples:

- UNIX shell commands
- Compilers:
 - Lexical Analysis -> parsing -> semantic analysis -> code generation
- Signal Processing

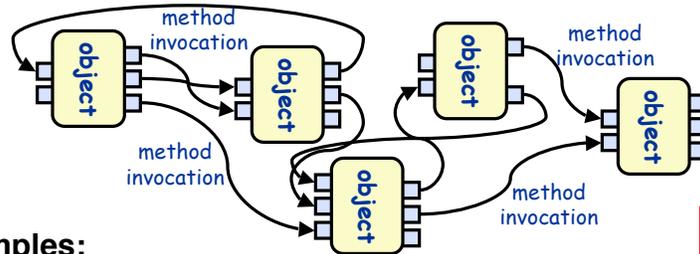
Interesting properties:

- filters don't need to know anything about what they are connected to
- filters can be implemented in parallel
- behaviour of the system is the composition of behaviour of the filters
- specialized analysis such as throughput and deadlock analysis is possible



Object Oriented Architectures

Source: Adapted from Shaw & Garlan 1996, p22-3.



Examples:

abstract data types

Interesting properties

- data hiding (internal data representations are not visible to clients)
- can decompose problems into sets of interacting agents
- can be multi-threaded or single thread

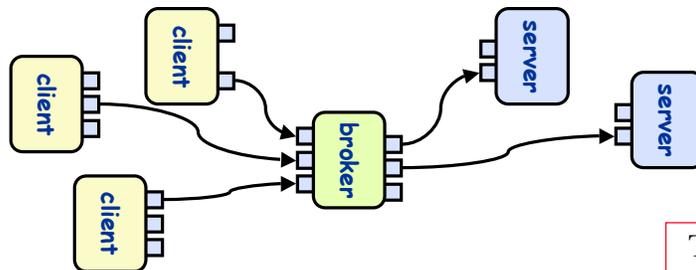
Disadvantages

objects must know the identity of objects they wish to interact with

This is not UML!



Variant: Object Brokers



Interesting properties

- Adds a broker between the clients and servers
- Clients no longer need to know which server they are using
- Can have many brokers, many servers.

Disadvantages

- Broker can become a bottleneck
- Degraded performance

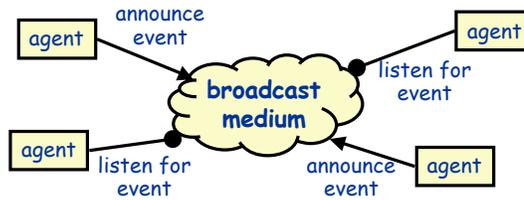
This is not UML!





Event based (implicit invocation)

Source: Adapted from Shaw & Garlan 1996, p23-4. See also van Vliet, 1999 Pp264-5 and p278



This is not UML!

Examples

- debugging systems (listen for particular breakpoints)
- database management systems (for data integrity checking)
- graphical user interfaces

Interesting properties

- announcers of events don't need to know who will handle the event
- Supports re-use, and evolution of systems (add new agents easily)

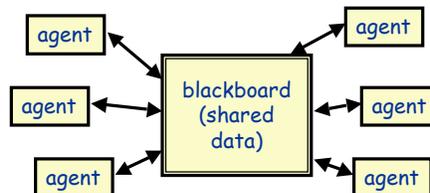
Disadvantages

- Components have no control over ordering of computations



Repositories

Source: Adapted from Shaw & Garlan 1996, p26-7. See also van Vliet, 1999, p280



Examples

- databases
- blackboard expert systems
- programming environments

Interesting properties

- can choose where the locus of control is (agents, blackboard, both)
- reduce the need to duplicate complex data

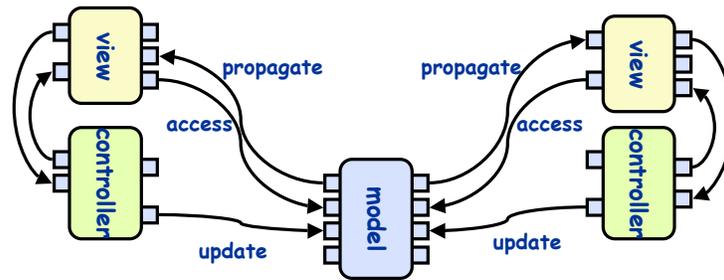
Disadvantages

- blackboard becomes a bottleneck





Variant: 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 the views