ADVANCED SOFTWARE DESIGN
LECTURE 4
SOFTWARE ARCHITECTURE

Dave Clarke

THIS LECTURE

At the end of this lecture you will know

- notations for expressing software architecture
- the design principles of cohesion and coupling
- various different architectural styles

DEFINITION

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

The highest level of design – large-scale structure of solution

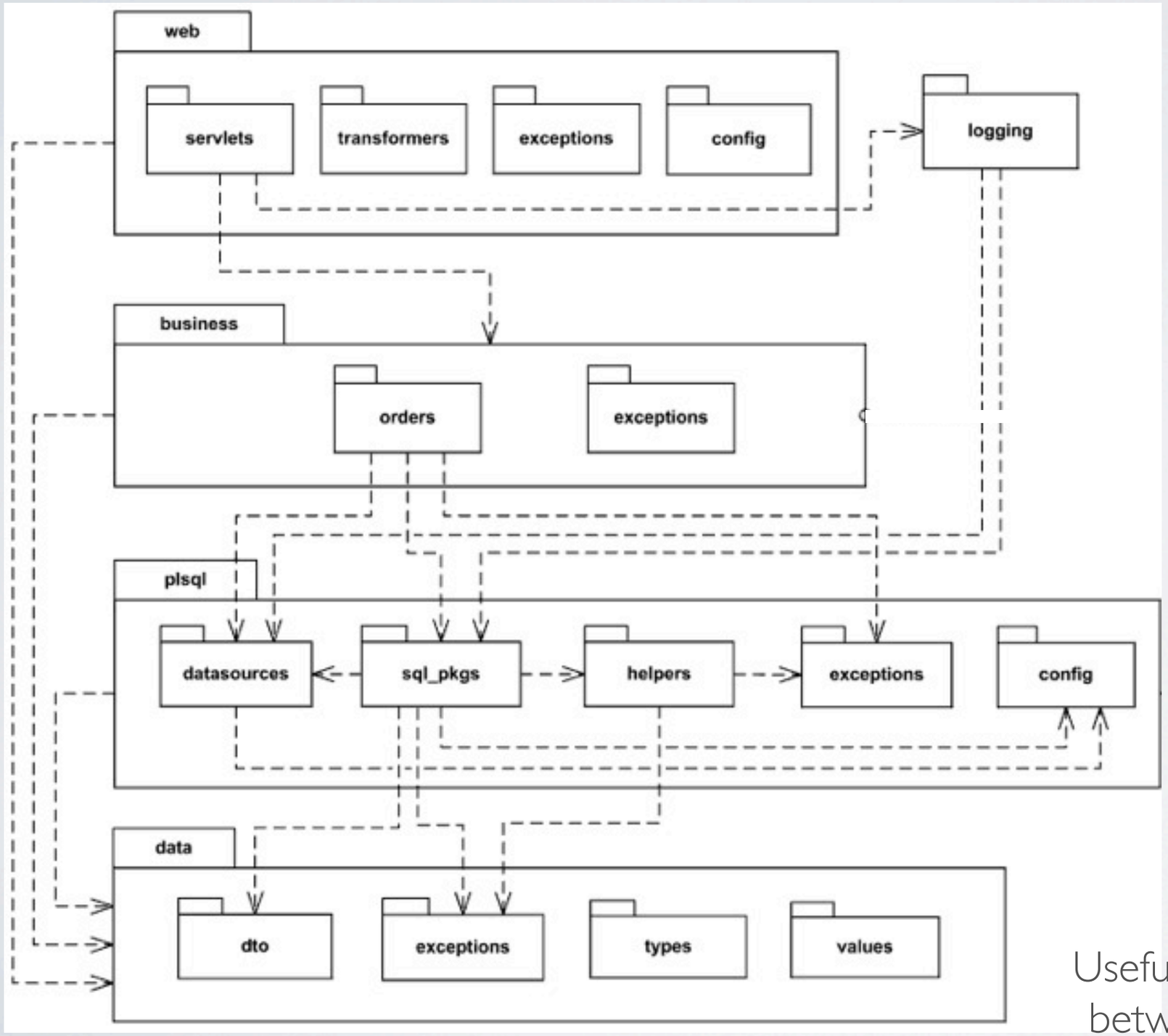
BUILDING BLOCKS

SUBSYSTEMS

- A replaceable part with well-defined interfaces that encapsulates the state and behaviour of its contained classes

- Two UML possibilities: Component and packages diagrams

PACKAGE DIAGRAMS



Packages
Hierarchy
Dependency

Useful for expressing the dependencies between major elements of a system.

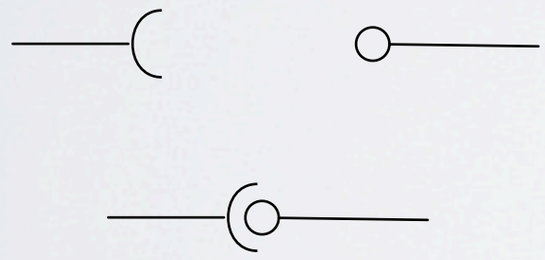
COMPONENT DIAGRAMS

Component – a set of related operations that share a common purpose

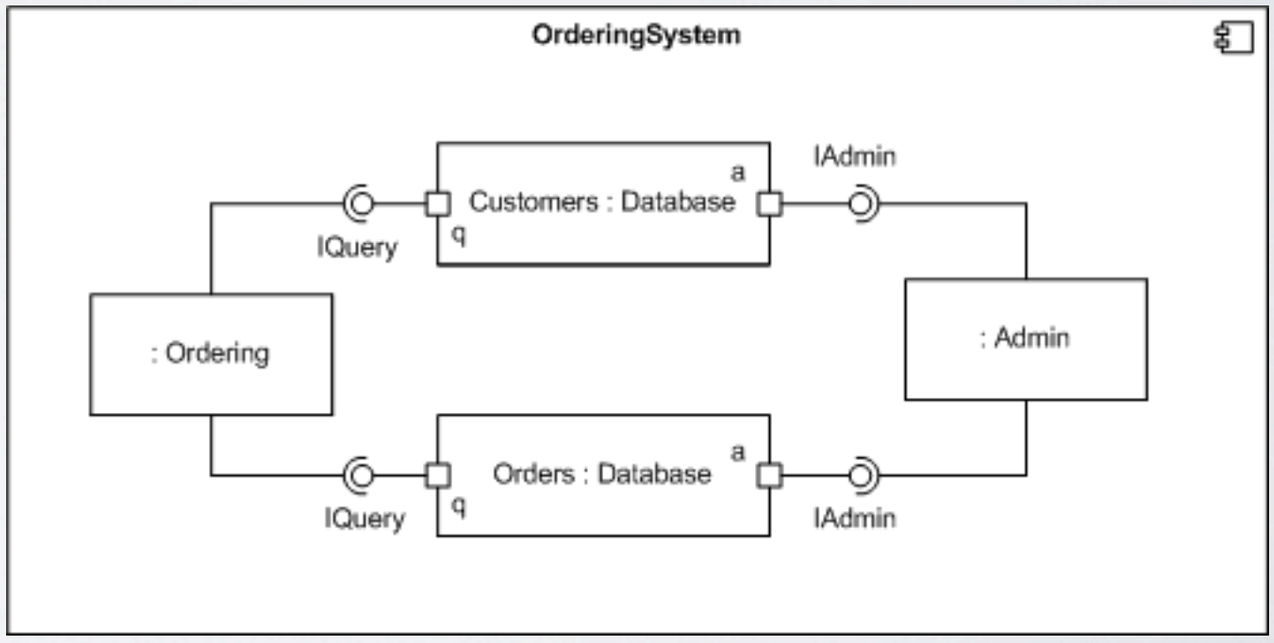
Interface – the set of operations available to other sub-systems

Required interface

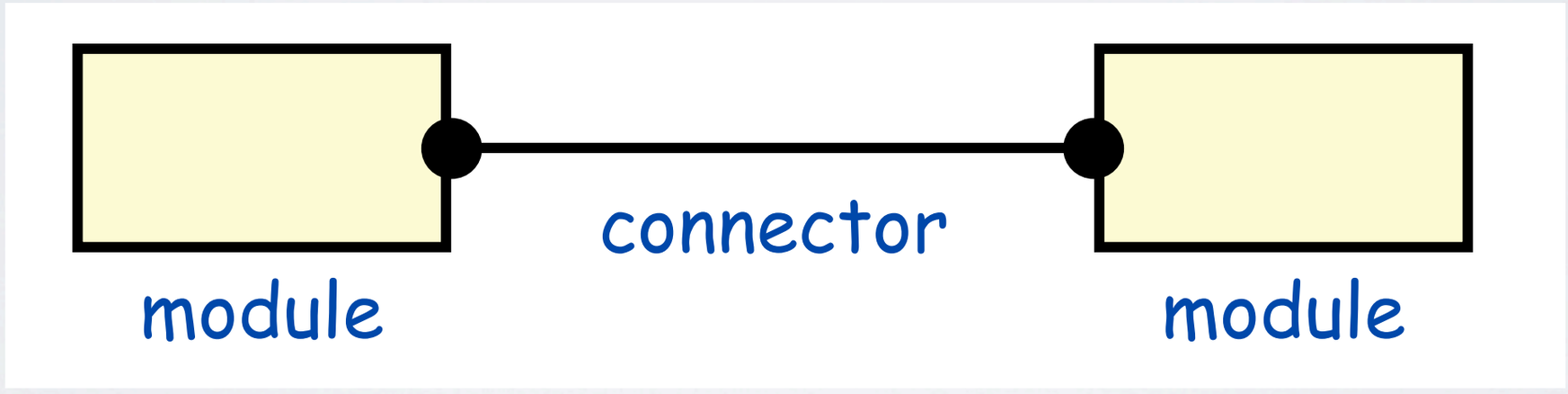
Provided interface



Plugging



NOTATION I'LL USE



COUPLING AND COHESION

COUPLING AND COHESION

- Coupling: Amount of relations between sub-systems
  - Low coupling general design goal: Independence, supportability
  - May lead to extra layers of abstraction
- Cohesion: Amount of relations within a sub-system
  - High cohesion general design goal: Reduce complexity
  - Often a trade-off

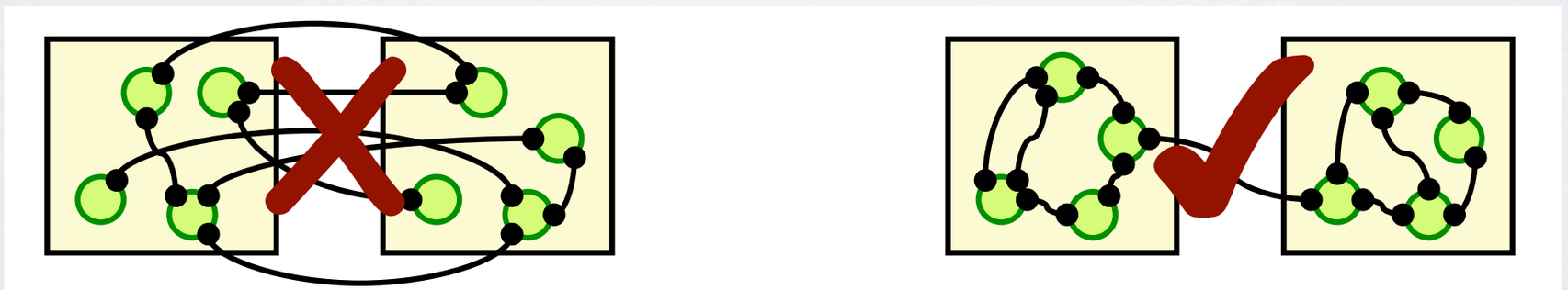
PROPERTIES OF A GOOD ARCHITECTURE

minimises coupling between modules

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

maximises cohesion within modules

- Goal: the contents of each module are strongly inter-related
- High cohesion makes a module easier to understand



Applies also to classes

LOW COUPLING

PROBLEM

How to support low dependency, low change impact, and increase reuse?

COUPLING

- Measure how strongly one element is connected to, has knowledge of or relies on other elements.

- An element with weak low (or weak) coupling is not dependent on too many other elements.

WHEN ARE TWO CLASSES COUPLED?

- Common forms of coupling from *TypeX* to *TypeY*:
  - *TypeX* has an attribute that refers to a *TypeY* instance.
  - A *TypeX* object calls on services of a *TypeY* object.
  - *TypeX* has a method that references an instance of *TypeY* (parameter, local variable, return type).
  - *TypeX* is a direct or indirect subclass of *TypeY*.
  - *TypeY* is an interface and *TypeY* implements that interface.

HIGH COUPLING (BAD)

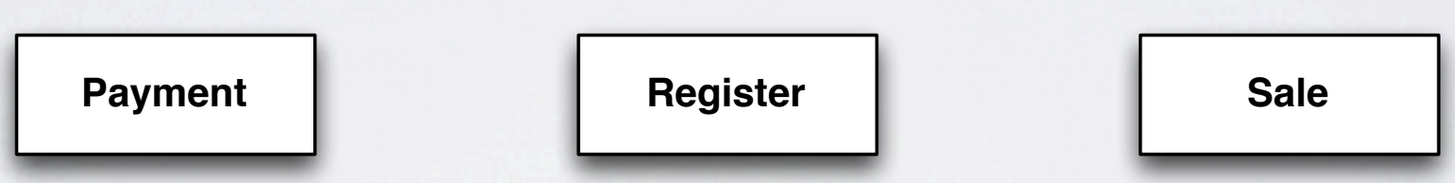
- A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable and suffer from the following problems:
  - Forced local changes because of changes in related classes.
  - Harder to understand in isolation.
  - Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

SOLUTION

- Assign responsibility so that coupling remains low.
- Use this principle to evaluate alternatives.

EXAMPLE

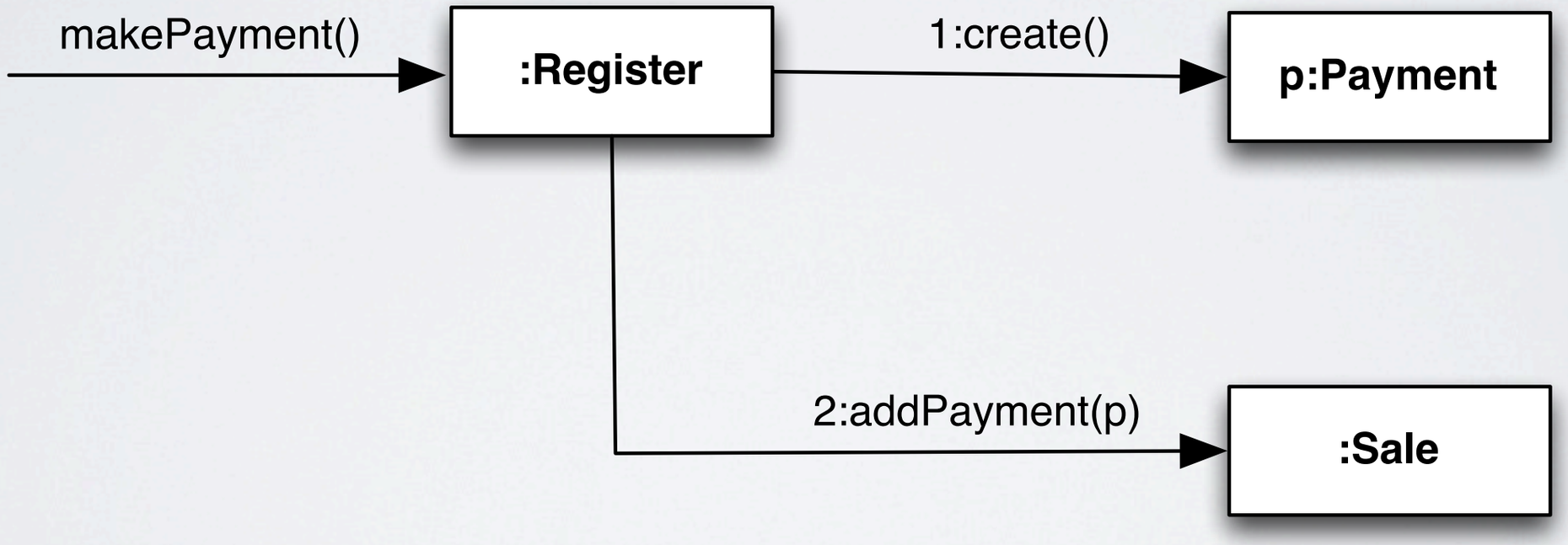
- Consider the following classes.



- We need to create a *Payment* instance and associate it with the *Sale*.

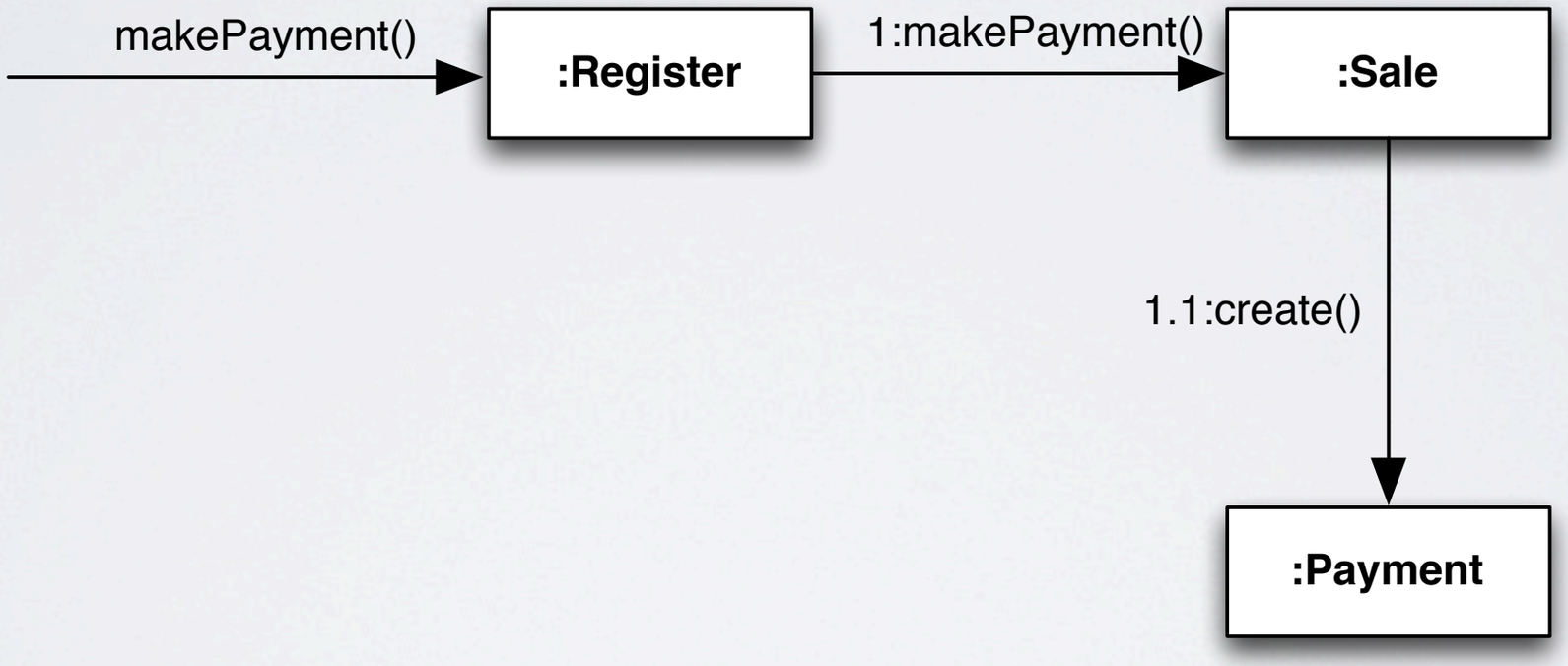
- Which class should be responsible?

POSSIBILITY #1



This is the solution Creator pattern (later) would suggest.

POSSIBILITY #2



WHICH IS BETTER?

Assume that each *Sale* will eventually be coupled with a *Payment*.

Design #1 has the coupling of *Register* and *Payment*, which is absent in Design #2.

Design #2 therefore has lower coupling.

Note that two patterns–Low Coupling and Creator–suggest different solutions.

Do not consider patterns in isolation.

DISCUSSION.

- Low coupling encourages designs to be more independent, which reduces the impact of change.

- Needs to be considered with other patterns such as Information Expert (later) and High Cohesion.

- Subclassing increases coupling – especially considering Domain objects subclassing technical services (e.g., *PersistentObject*)

- High coupling to stable "global" objects is not problematic – to Java libraries such as *java.util.*

PICK YOUR BATTLES

- The problem is not high coupling per se; it is high coupling to *unstable* elements.
- Designers can *future proof* various aspects of the system using lower coupling and encapsulation, but there needs to be good motivation.
- Focus on points of realistic high instability and evolution.

HIGH COHESION

PROBLEM

How to keep objects focussed, understandable, and manageable, and as a side-effect, support Low Coupling?

Cohesion is a measure of how strongly related and focused the responsibilities of an element are.

An element with highly related responsibilities that does not do a tremendous amount of work has high cohesion.

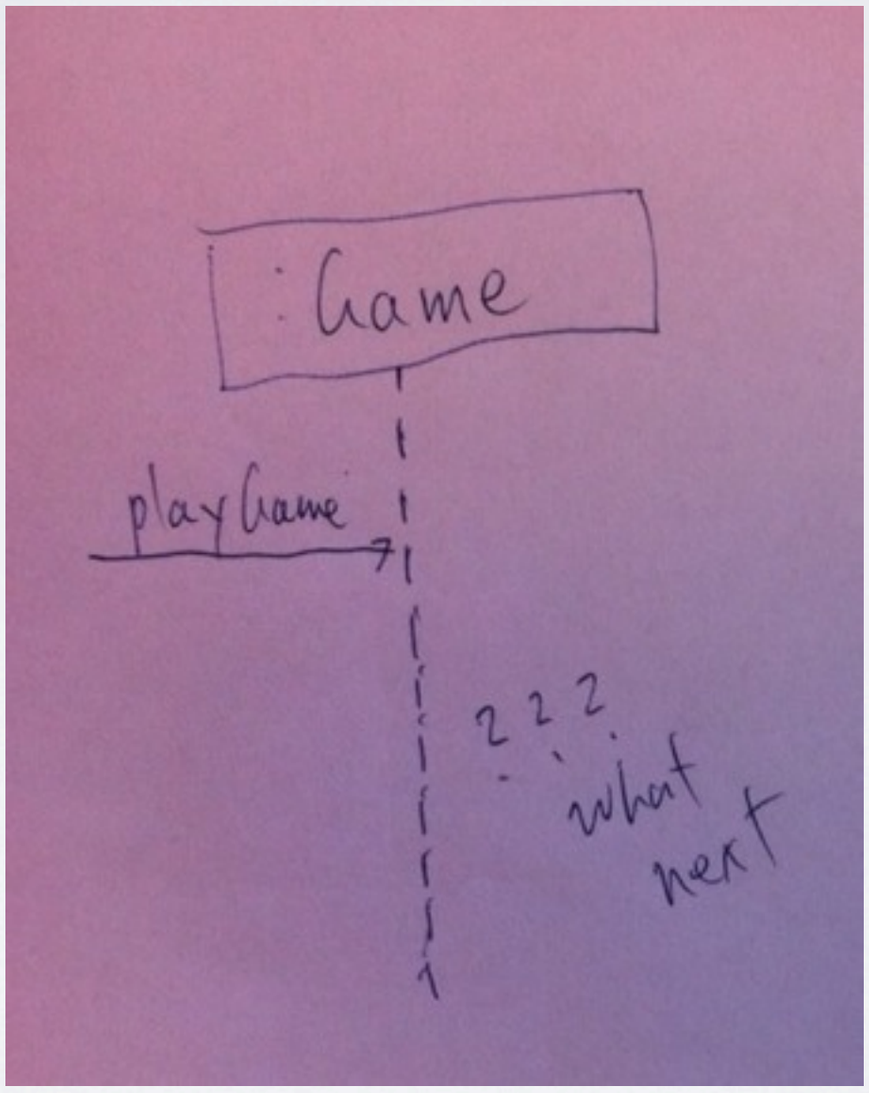
LOW COHESION

- A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems:
  - hard to comprehend
  - hard to reuse
  - hard to maintain
  - delicate; constantly affected by change.

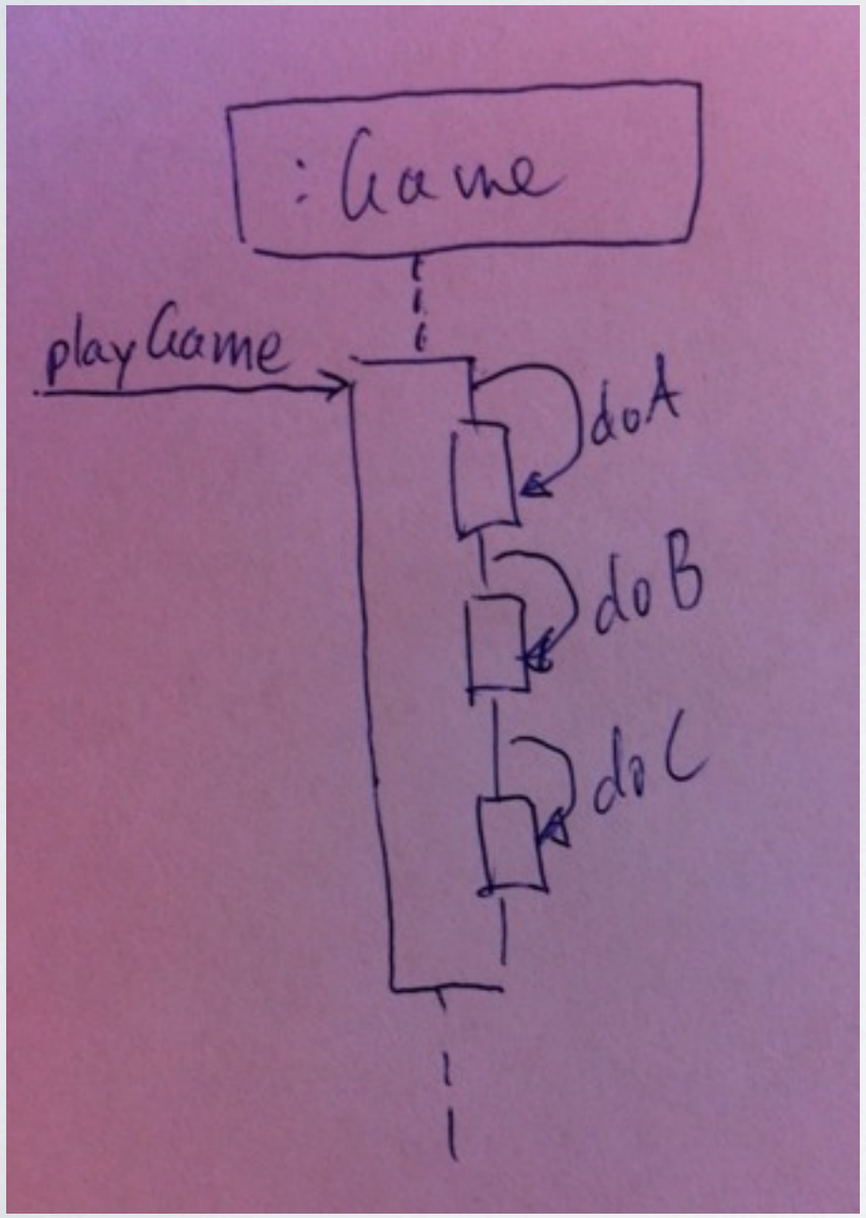
SOLUTION

- Assign responsibility so that cohesion remains high.
- Use this to evaluate alternatives.

EXAMPLE

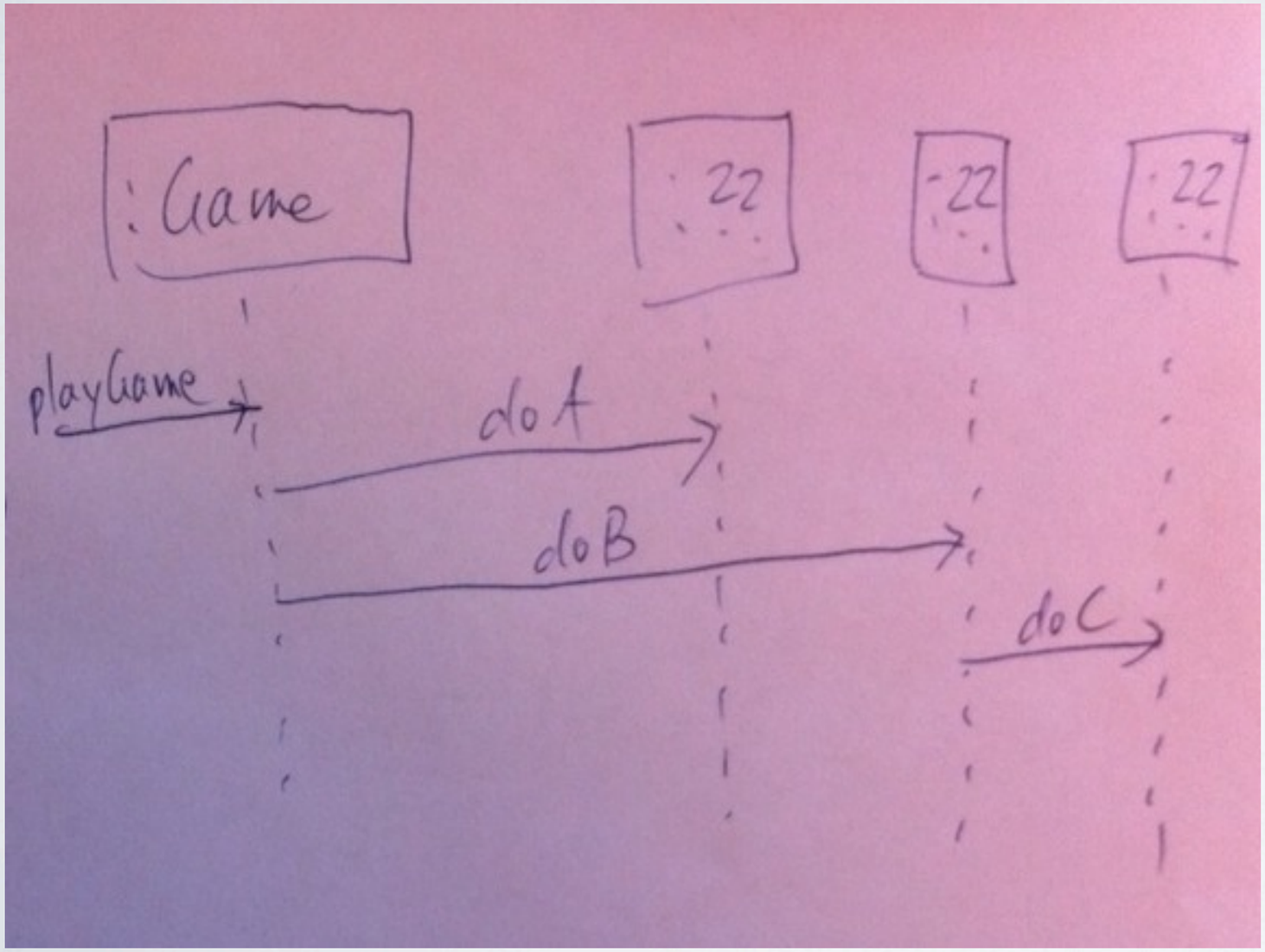


LOW COHESION



Game is doing all the work.
How related are tasks A, B, C?
If not strongly related, then
design has low cohesion.

HIGH COHESION



Delegates and distributes the work, presumably to objects with appropriately assigned responsibilities.

DISCUSSION

- *Very low cohesion*: a class that does two completely different tasks, e.g., database connectivity and RPC.

- *Low cohesion*: a class that has sole responsibility for a complex task in one functional area, e.g., one single interface responsible for all database access.

- *Moderate cohesion*: a lightweight class, solely responsible for a few logically related areas, e.g., *Company* that knows the employees and the financial details.

- *High cohesion*: a class with moderate responsibilities in one functional area that collaborates with other classes.

RULES OF THUMB

- For *high cohesion*, a class must
  - have few methods
  - have a small number of lines of code
  - not do too much work
  - have high relatedness of code.

ARCHITECTURAL STYLES

ARCHITECTURAL STYLES

Object-oriented

Client server; object broker; peer to peer

Pipe and filter

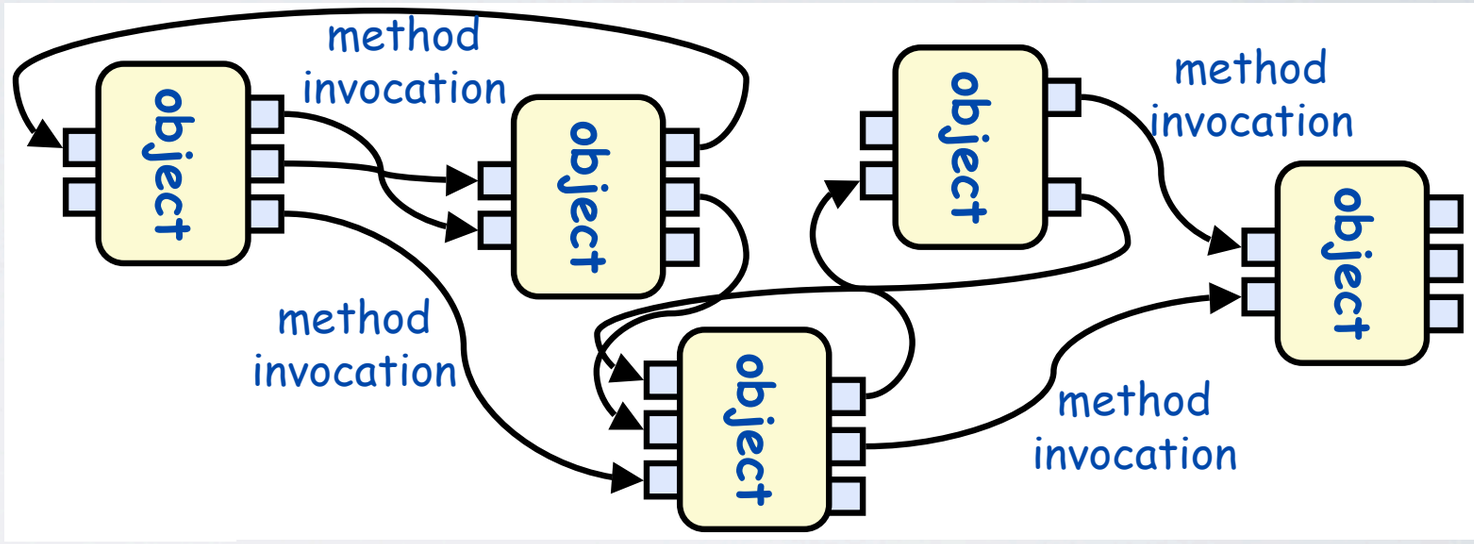
Event based – publish/subscriber

Layered – Three-tier, Four-tier

Repositories: blackboard, Model/View/Controller (MVC)

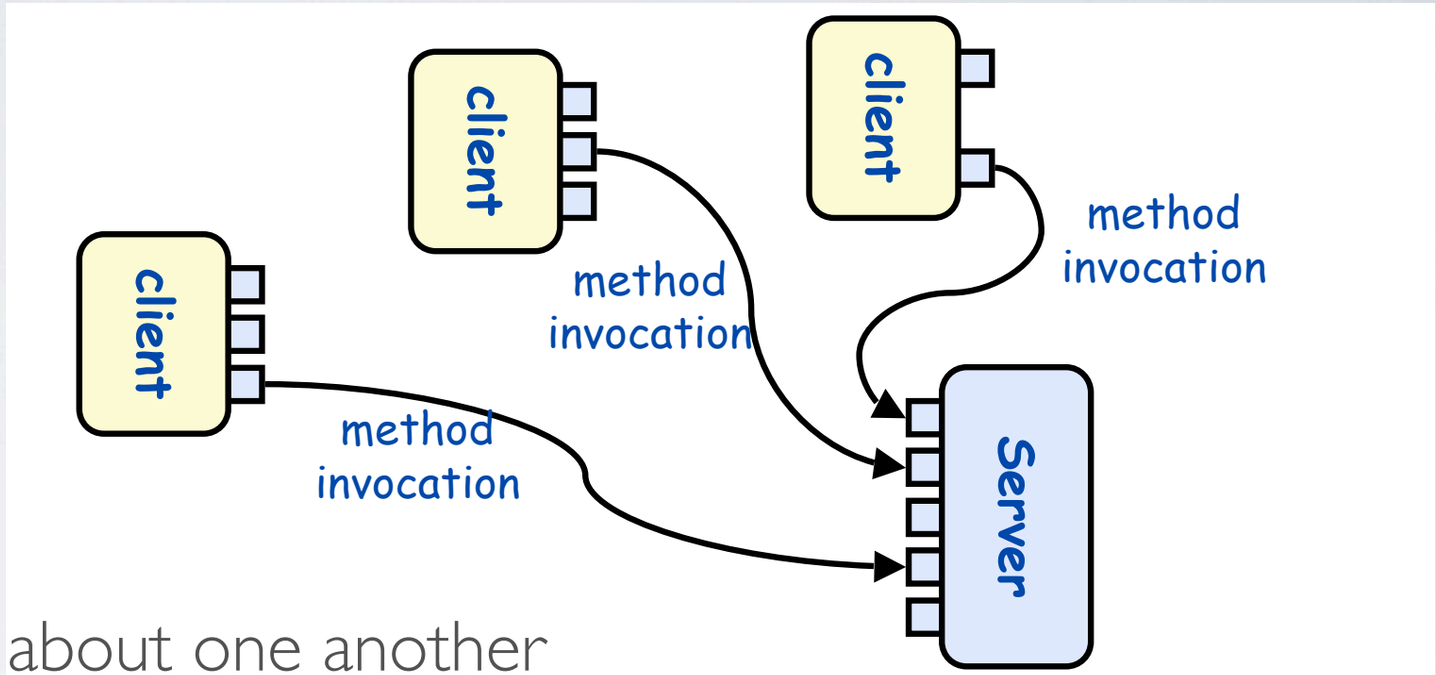
Process control

OBJECT-ORIENTED ARCHITECTURES



- Example
  - Abstract data types (modules)
- 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

VARIANT: CLIENT-SERVER



- Interesting properties
  - Clients do not need to know about one another
  - Breaks the system into manageable components
  - Independent flow of control
  - Server generally responsible for persistence and consistency of data
- Disadvantages
  - Client objects must know the identity of the server

CLIENT-SERVER

Client/Server communication via remote procedure call or common object broker (e.g. CORBA, Java RMI, or HTTP)

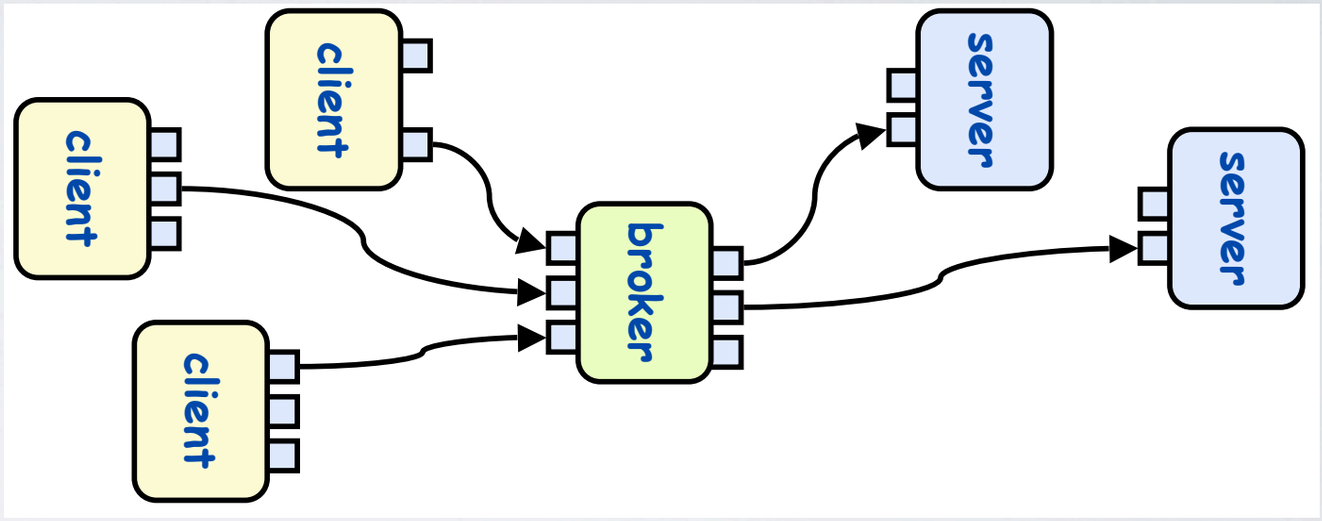
Distributed systems. e.g,. web services, system with central DBMS

Variants

– thick clients have their own services

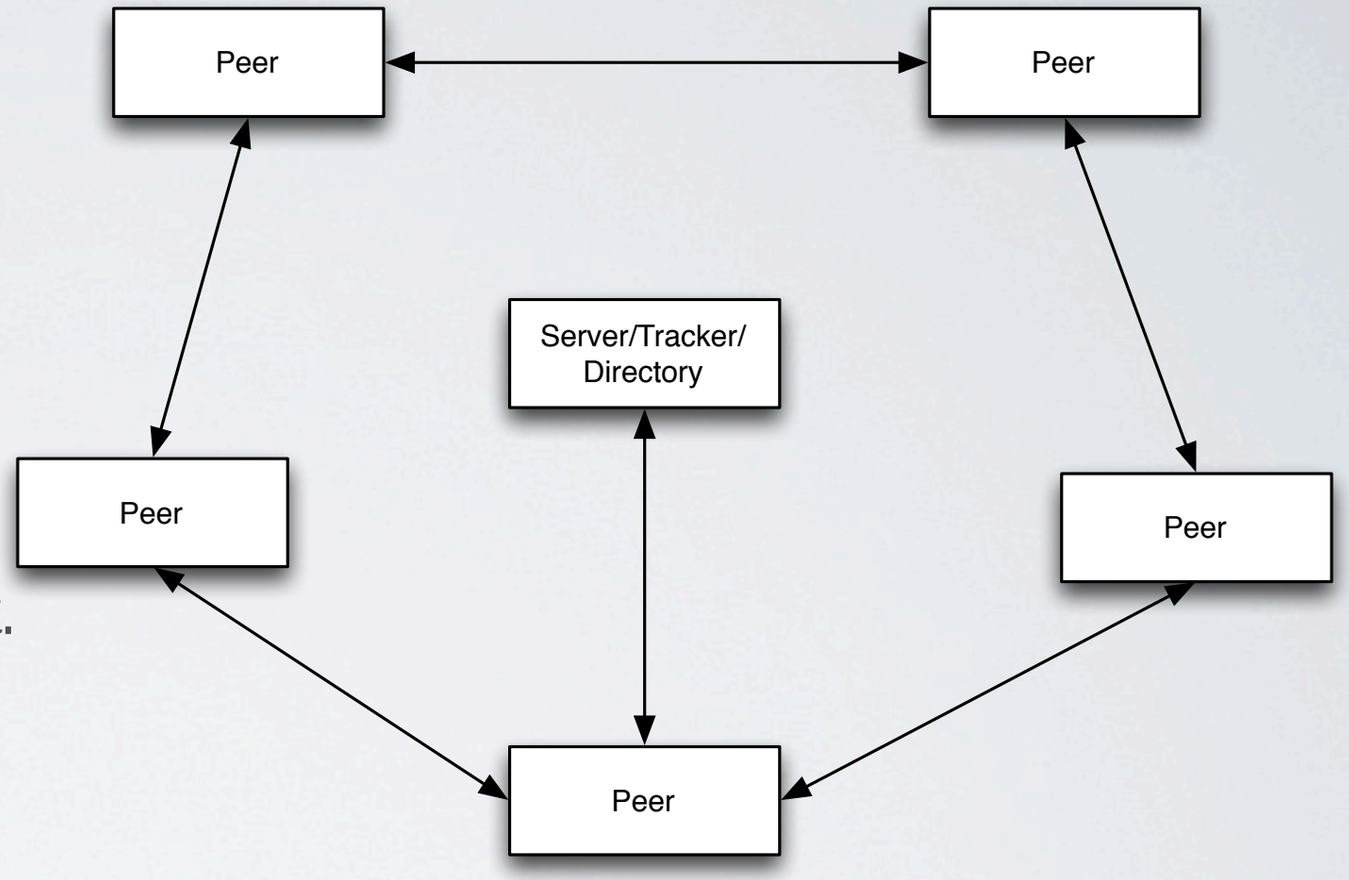
– thin ones get everything from servers

VARIANT: OBJECT BROKER



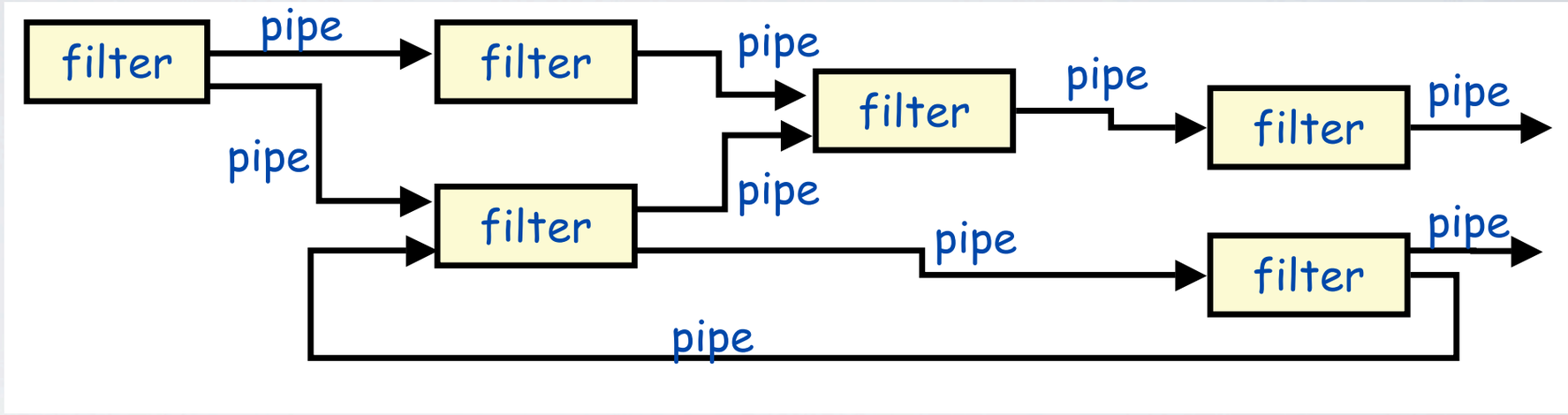
- Interesting Properties
  - Adds a broker between the clients and servers
  - Clients no longer need to know which server they are using
  - Can have many brokers, many servers.
- Disadvantages
  - Broker can become a bottleneck
  - Degraded performance

VARIANT: PEER-TO-PEER



- Interesting Properties
  - Find peers via server or broadcast.
  - Interact subsequently with peers.
  - Reduces bottleneck. Robust to peer failure.
- Disadvantages
  - Server can become a bottleneck
  - Peers have only incomplete picture – synchronisation is (virtually) impossible

PIPE AND FILTER

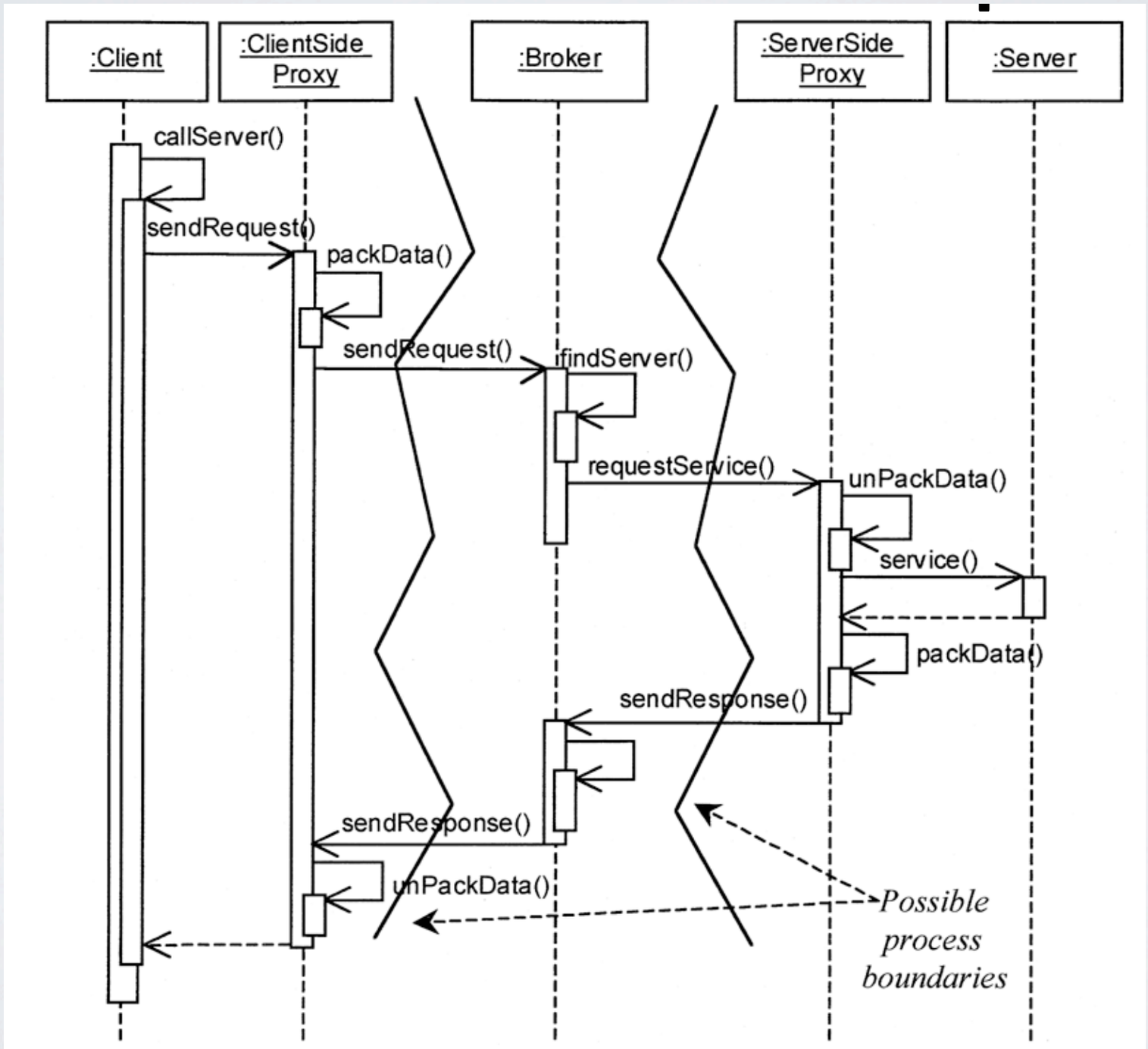


- Examples
  - Unix command shell
  - compiler chain: lexical analysis → parsing → semantic analysis → code generation
  - signal processing

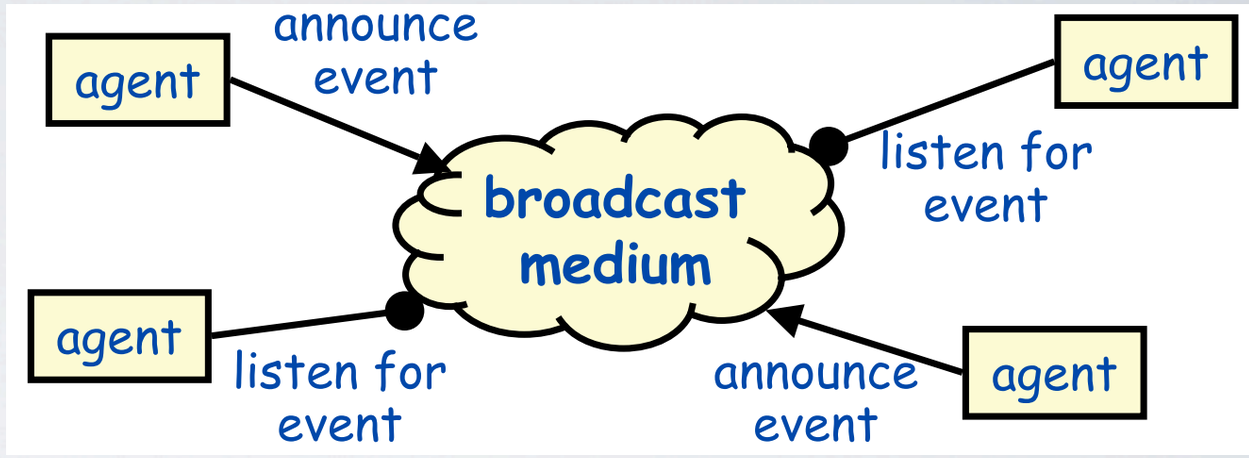
```
grep gustav < foo.txt | sort | cut -f2-3
```

- 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
    - specialised analysis such as throughput and deadlock analysis is possible

BROKER ARCHITECTURE EXAMPLE

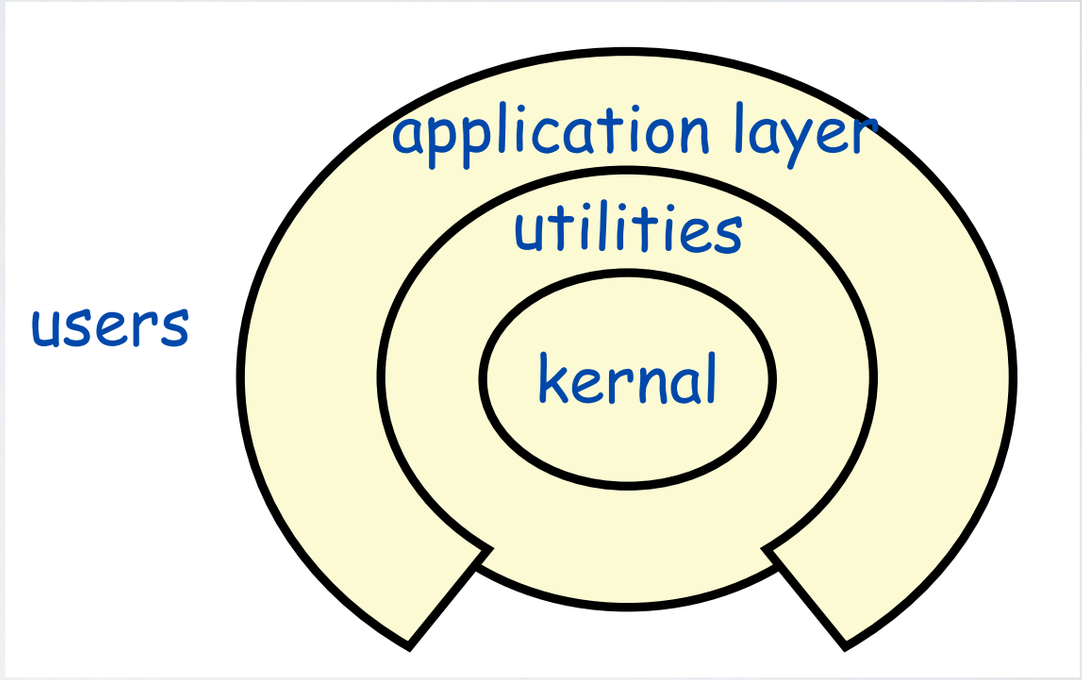


EVENT-BASED (IMPLICIT INVOCATION)



- 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

LAYERED SYSTEMS

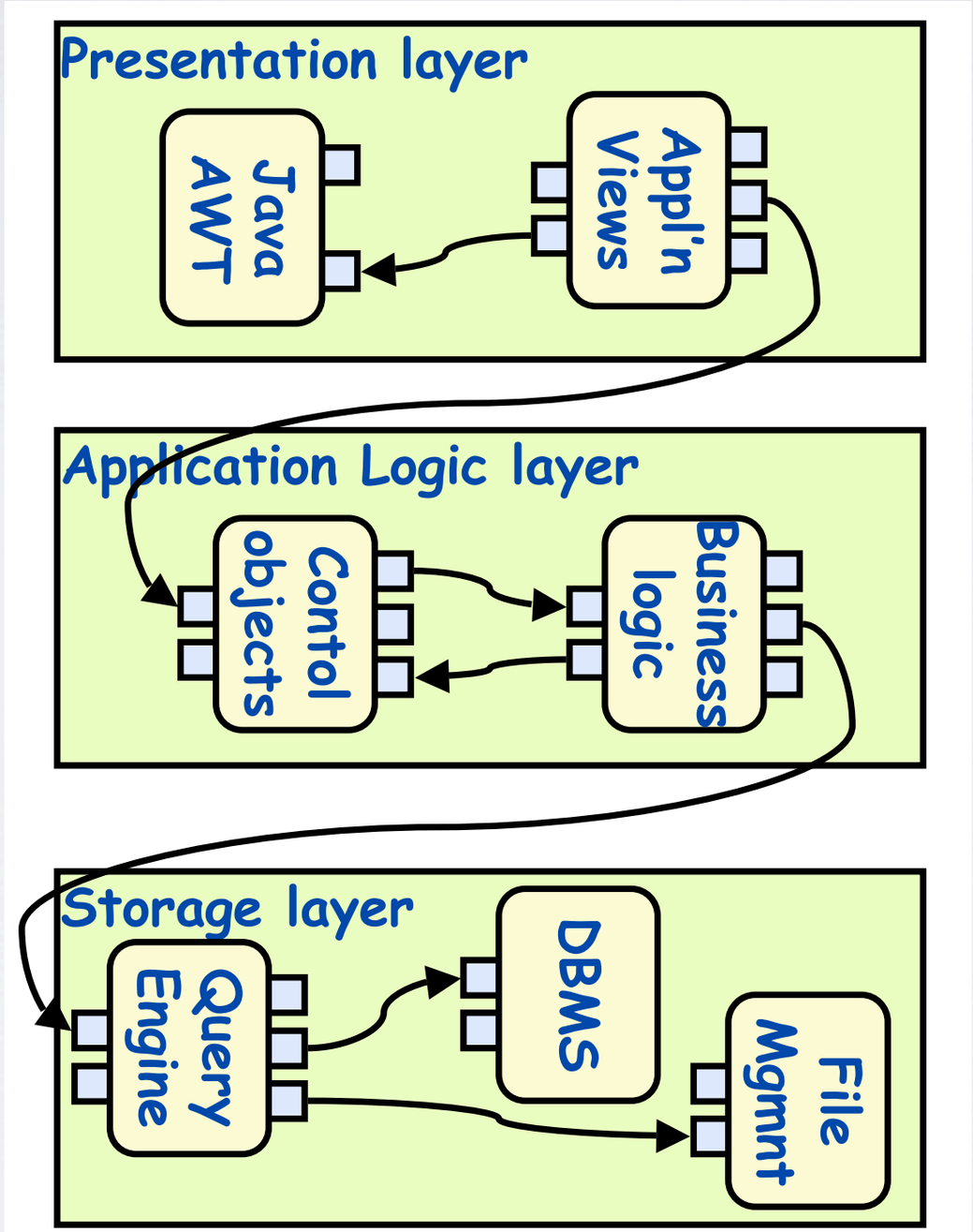


- Examples
- Operating Systems
- communication protocols
- Interesting properties
- Support increasing levels of abstraction during design
- Support enhancement (add functionality) and re-use
- can define standard layer interfaces
- Disadvantages
- May not be able to identify (clean) layers

LAYERS AND PARTITIONS

- Layering
  - Hierarchical decomposition → tree of sub-systems
  - "Horizontal division"
  - Open vs closed layers
- Partitioning
  - "Vertical division": Each set handles a function
  - Extreme form of decoupling: Semi-independent

EXAMPLE: 3-LAYER DATA ACCESS



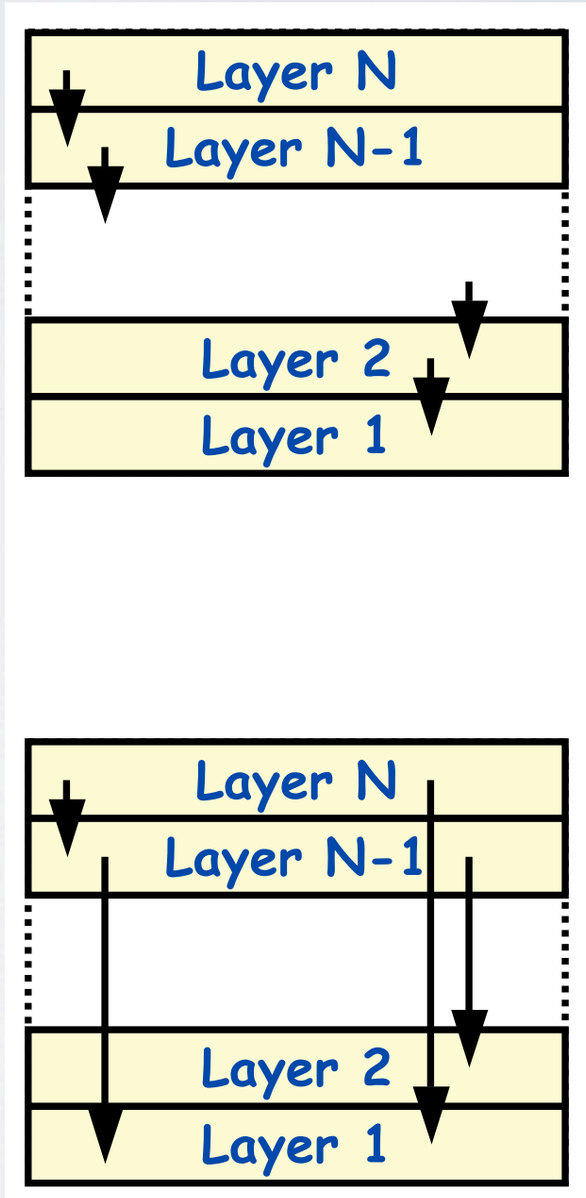
OPEN VS CLOSED ARCHITECTURE

closed architecture

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

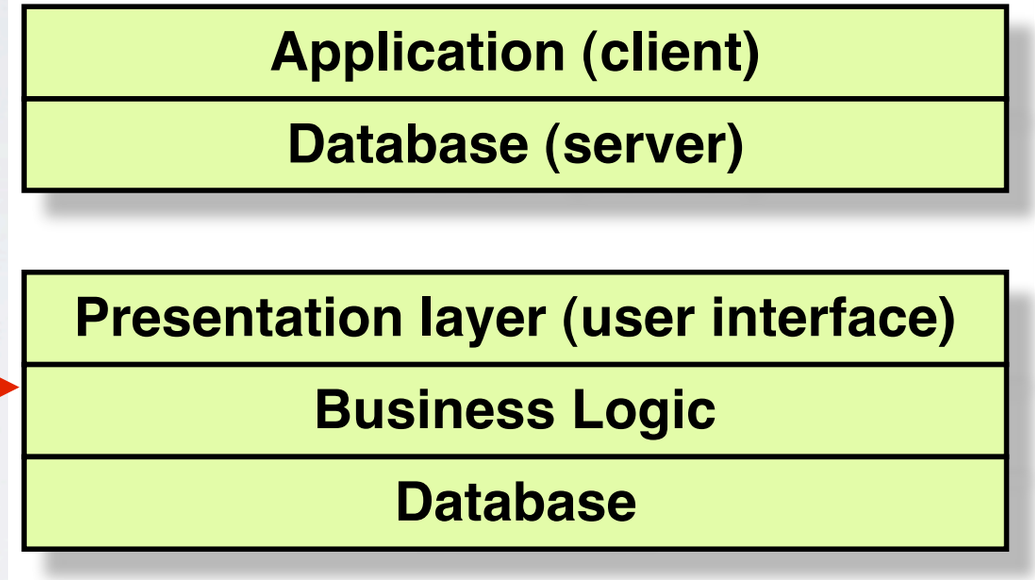
open architecture

- 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



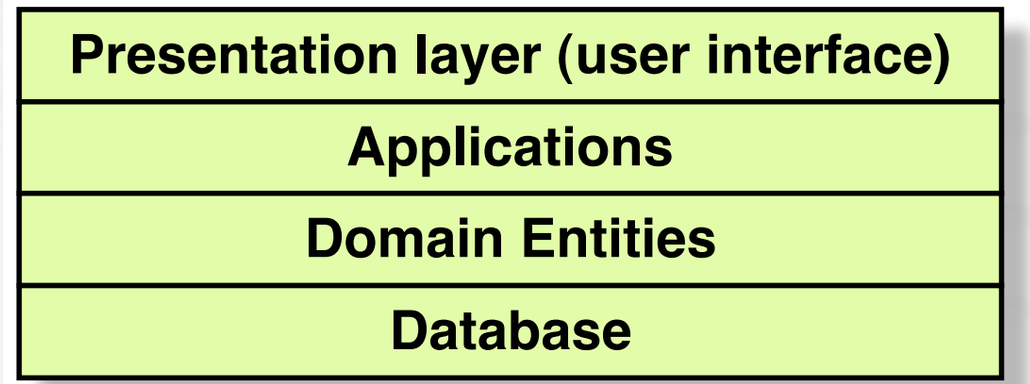
HOW MANY LAYERS?

2 layers:
- application layer
- database layer
- e.g., simple client-server model

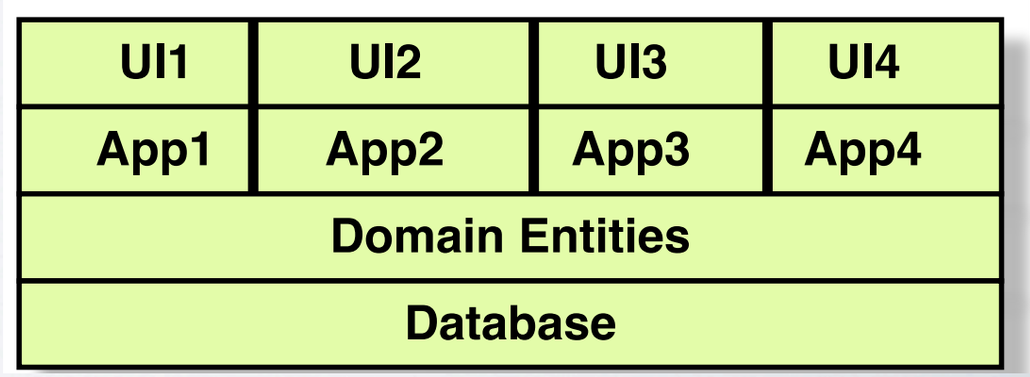


3 layers (three tier):
- separate out the business logic
  - helps make both user interface and database layers modifiable

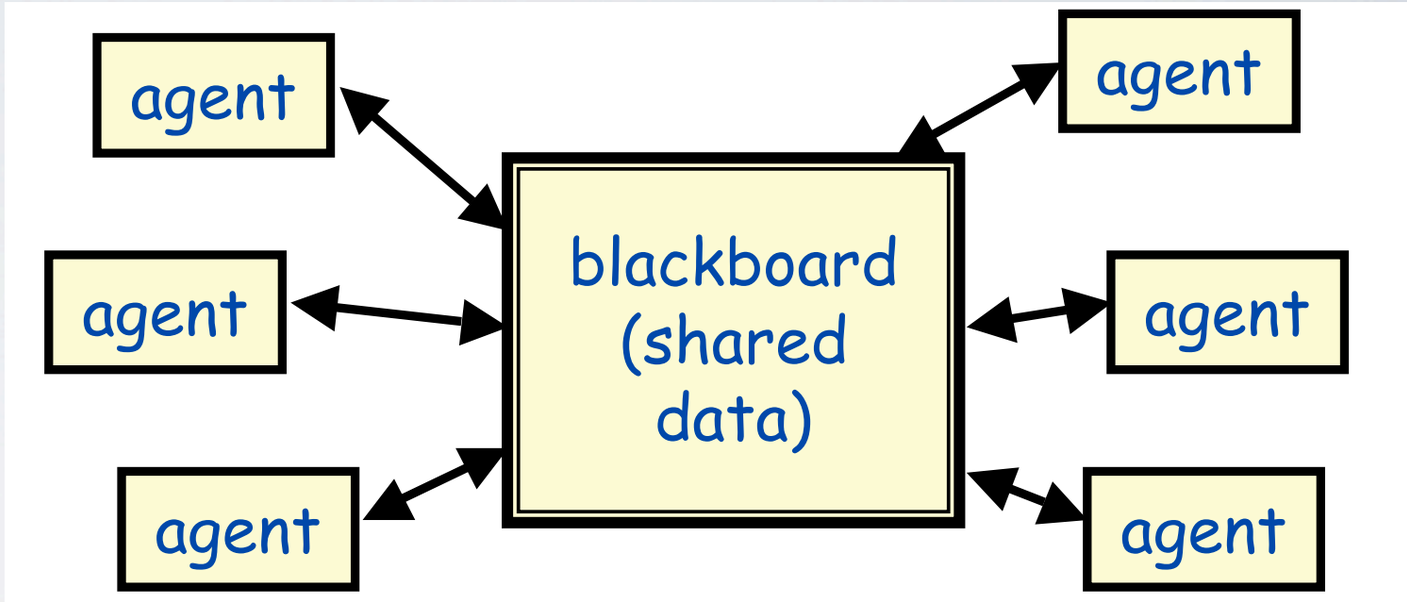
4 layers (four tier):
- separate applications from the domain entities that they use
  - boundary classes in presentation layer
  - control classes in application layer
  - entity classes in domain layer



partitioned 4 layers:
- identify separated applications



REPOSITORIES

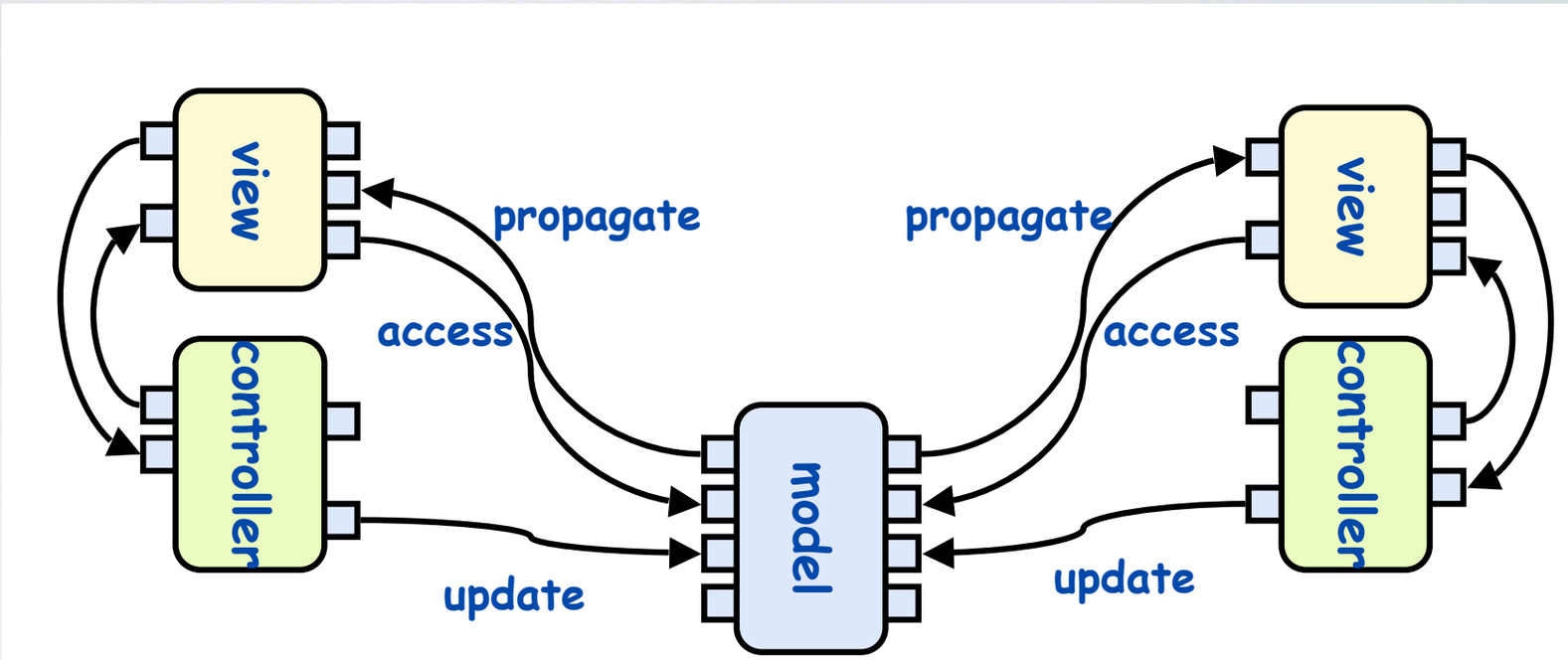


- Examples
- databases
- blackboard expert systems
- programming environments
- Interesting properties
- can choose where the locus of control is (agents, blackboard, both)
- reduce the need to duplicate complex data
- Disadvantages
- blackboard becomes a bottleneck

REPOSITORY

- Sub-systems access and modify a single data structure
- Complex, changing data
- Concurrency & data consistency
- Control by sub-systems or by repository (blackboard)
- Disadvantage: Possibly performance bottlenecks and reduced modifiability
- E.g. databases, IDE's, tuple spaces

MODEL-VIEW-CONTROLLER

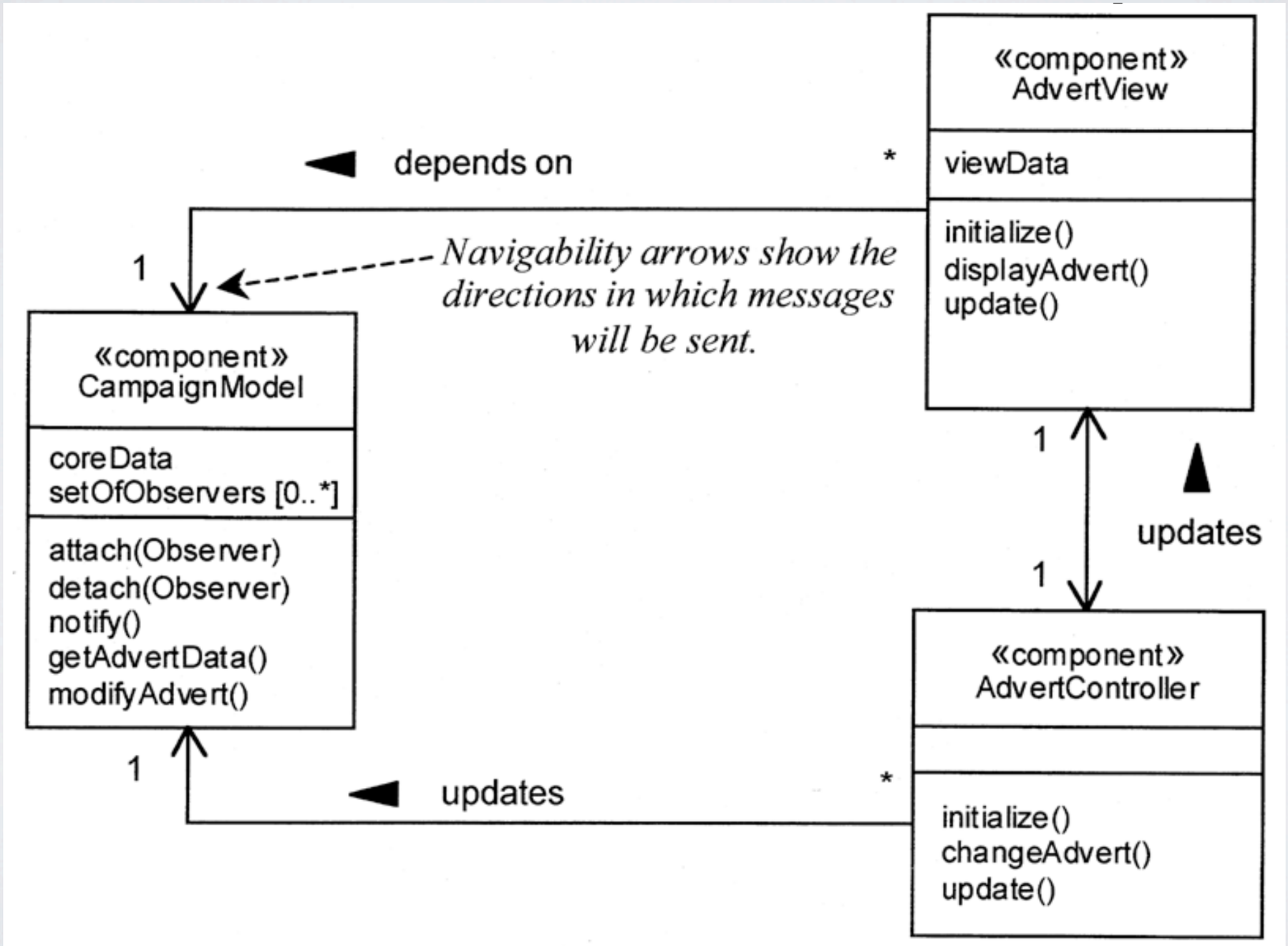


- Properties
  - One central model, many views (viewers)
  - Each view has an associated controller
  - The controller handles updates from the user of the view
  - Changes to the model are propagated to all the views

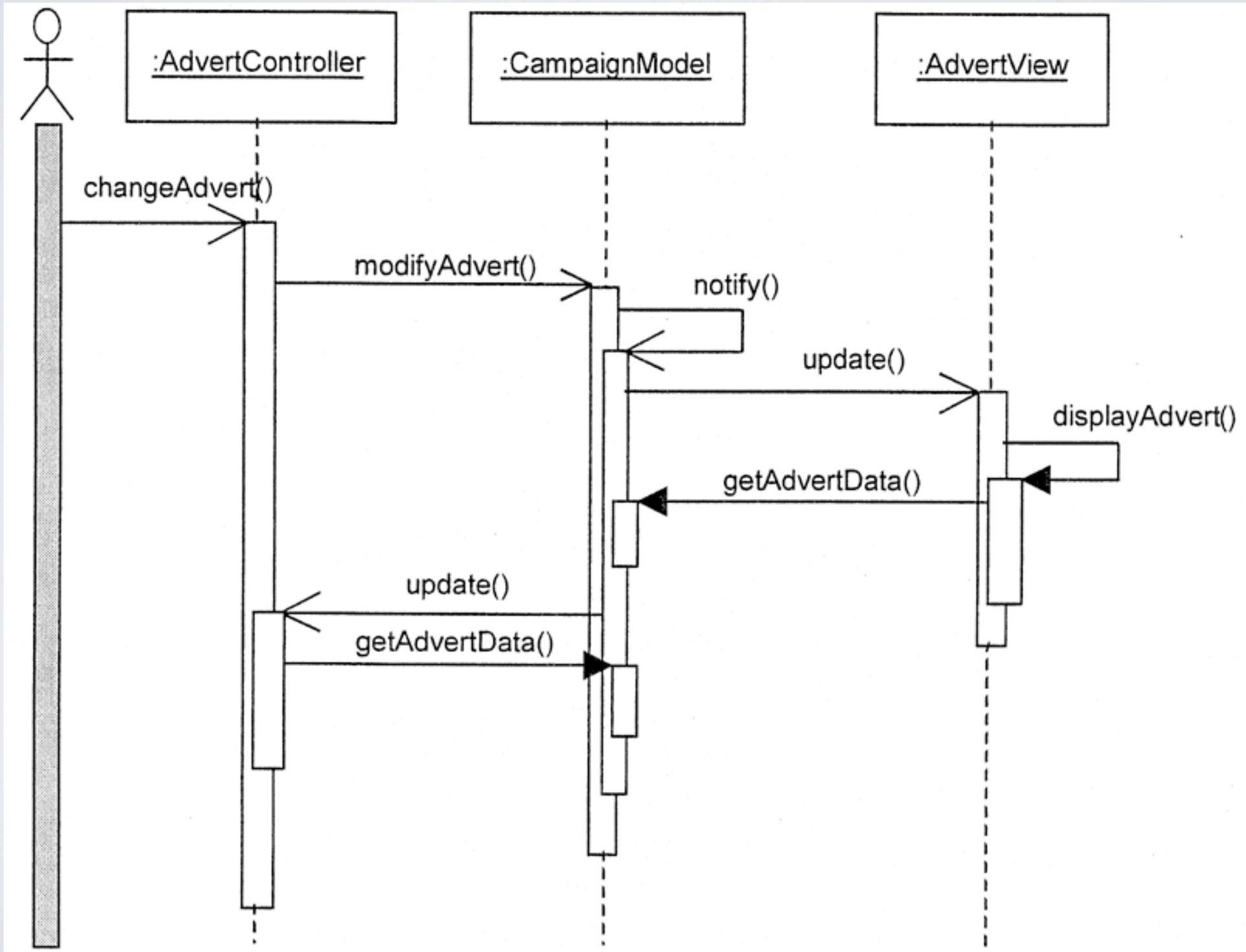
MODEL-VIEW-CONTROLLER (MVC)

- Can be seen as a special case of repository:
  - Model contains domain knowledge (= repository)
  - Views only display data
  - Controllers only manage interaction sequences
- Model does not depend on views or controllers
- Subscribe/notify mechanism
- Interactive systems
- Same potential drawbacks as the repository pattern

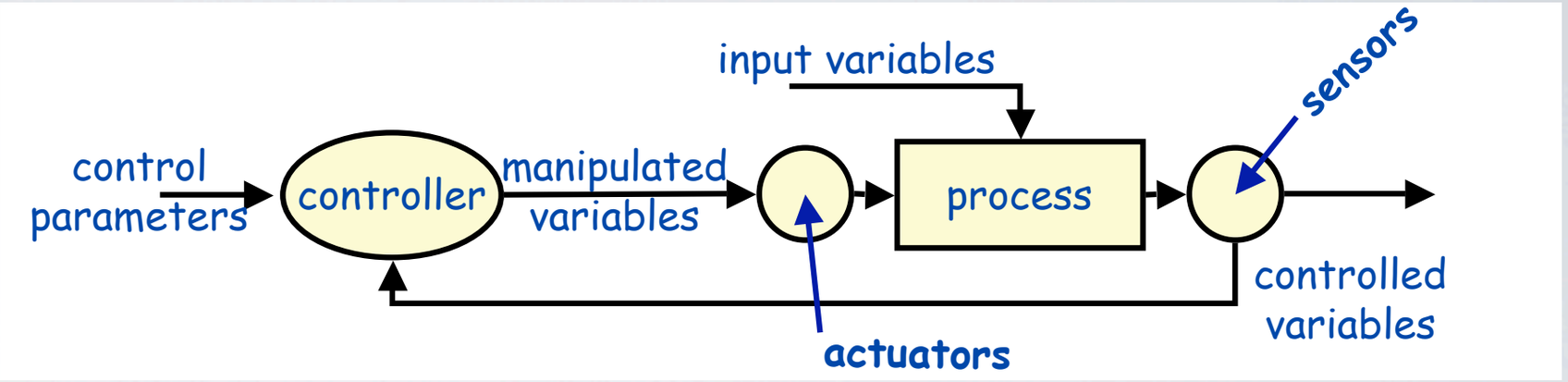
EXAMPLE
MODEL-VIEW-CONTROLLER



MVC INTERACTION



PROCESS CONTROL



- Examples
  - aircraft/spacecraft flight control systems
  - controllers for industrial production lines, power stations, etc.
  - chemical engineering
- Interesting properties
  - separates control policy from the controlled process
  - handles real-time, reactive computations
- Disadvantages
  - difficult to specify the timing characteristics and response to disturbances

THIS LECTURE

This lecture covered

- notations for expressing software architecture
- the design principles of cohesion and coupling
- various different architectural styles