

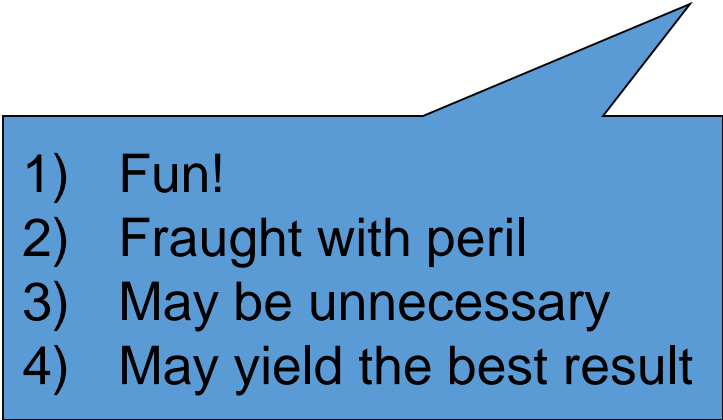
# Architectural Design

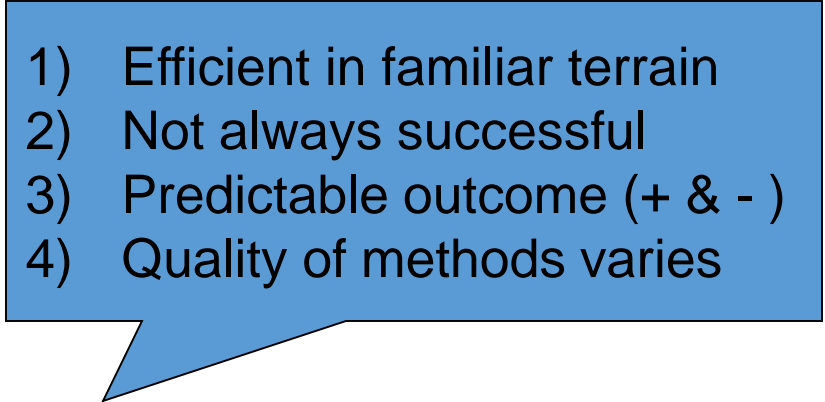
Ack: Slides from Neno

# How Do You Design?

*Where do architectures come from?*

## Creativity

- 
- 1) Fun!
  - 2) Fraught with peril
  - 3) May be unnecessary
  - 4) May yield the best result

- 
- 1) Efficient in familiar terrain
  - 2) Not always successful
  - 3) Predictable outcome (+ & - )
  - 4) Quality of methods varies

## Method

# Objectives

- Creativity
  - Enhance your skill-set
  - Provide new tools
- Method
  - Focus on highly effective techniques
- Develop judgment: when to develop novel solutions, and when to follow established method

# The Tools of “Software Engineering 101”

- Abstraction
  - Abstraction(1): look at details, and abstract “up” to concepts
  - Abstraction(2): choose concepts, then add detailed substructure, and move “down”
    - Example: design of a stack class
- Separation of concerns

# Separation of Concerns

- Separation of concerns is the subdivision of a problem into (hopefully) independent parts.
- The difficulties arise when the issues are either actually or apparently intertwined.
- Separations of concerns frequently involves many tradeoffs
- Total independence of concepts may not be possible.
- Key example from software architecture: separation of components (computation) from connectors (interaction)

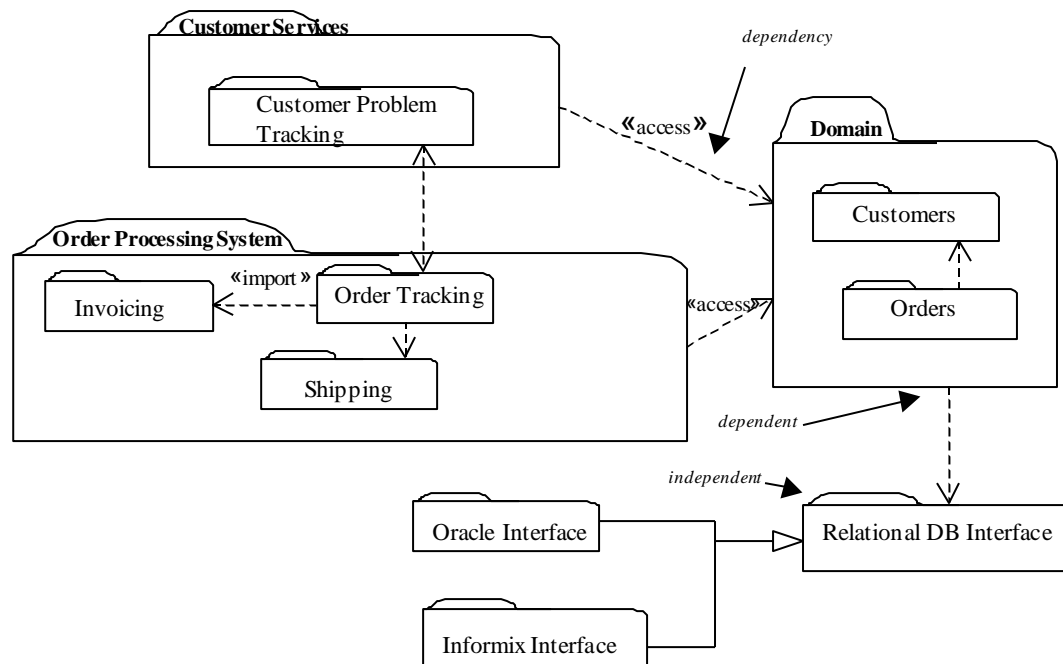
# Software Architecture

- *Software architecture* is process of designing the global organization of a software system, including:
  - Dividing software into subsystems.
  - Deciding how these will interact.
  - Determining their interfaces.
- The architecture is the core of the design, so all software engineers need to understand it.
- The architecture will often constrain the overall efficiency, reusability and maintainability of the system.

# Architecture Types

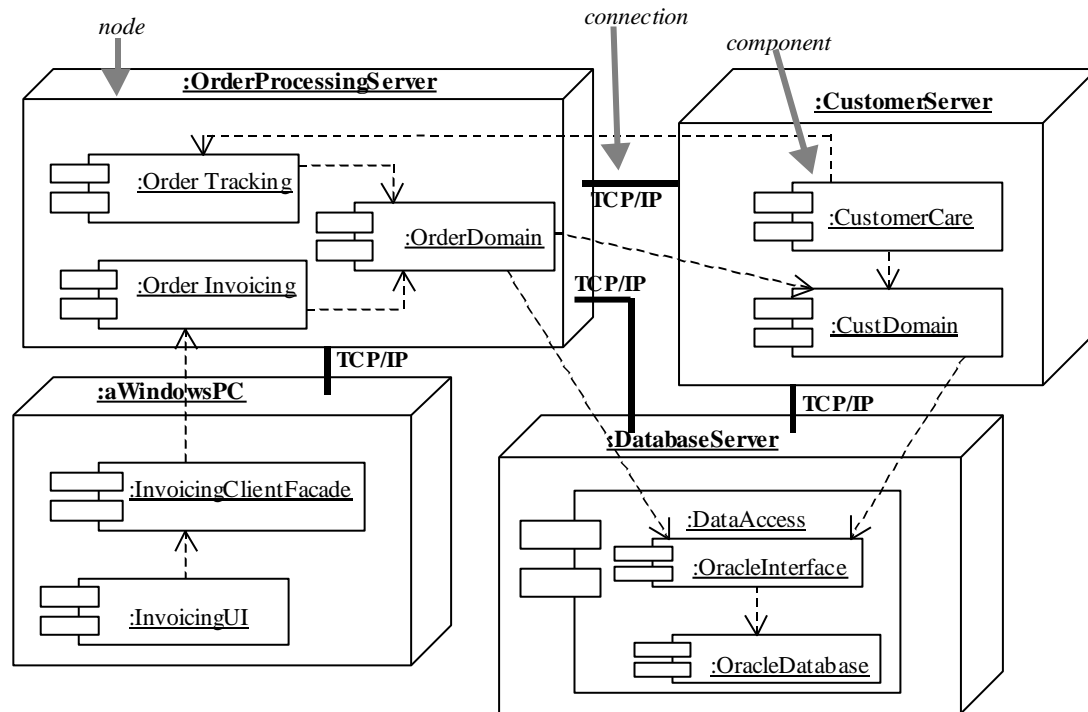
- Logical Architecture
  - Organization of a system into logical units (e.g., layers, subsystems)
  - Logical units do not necessarily result in units of implementation
- Implementation (Deployment) Architecture
  - Organization of a system into physical units

# Example Logical Architecture

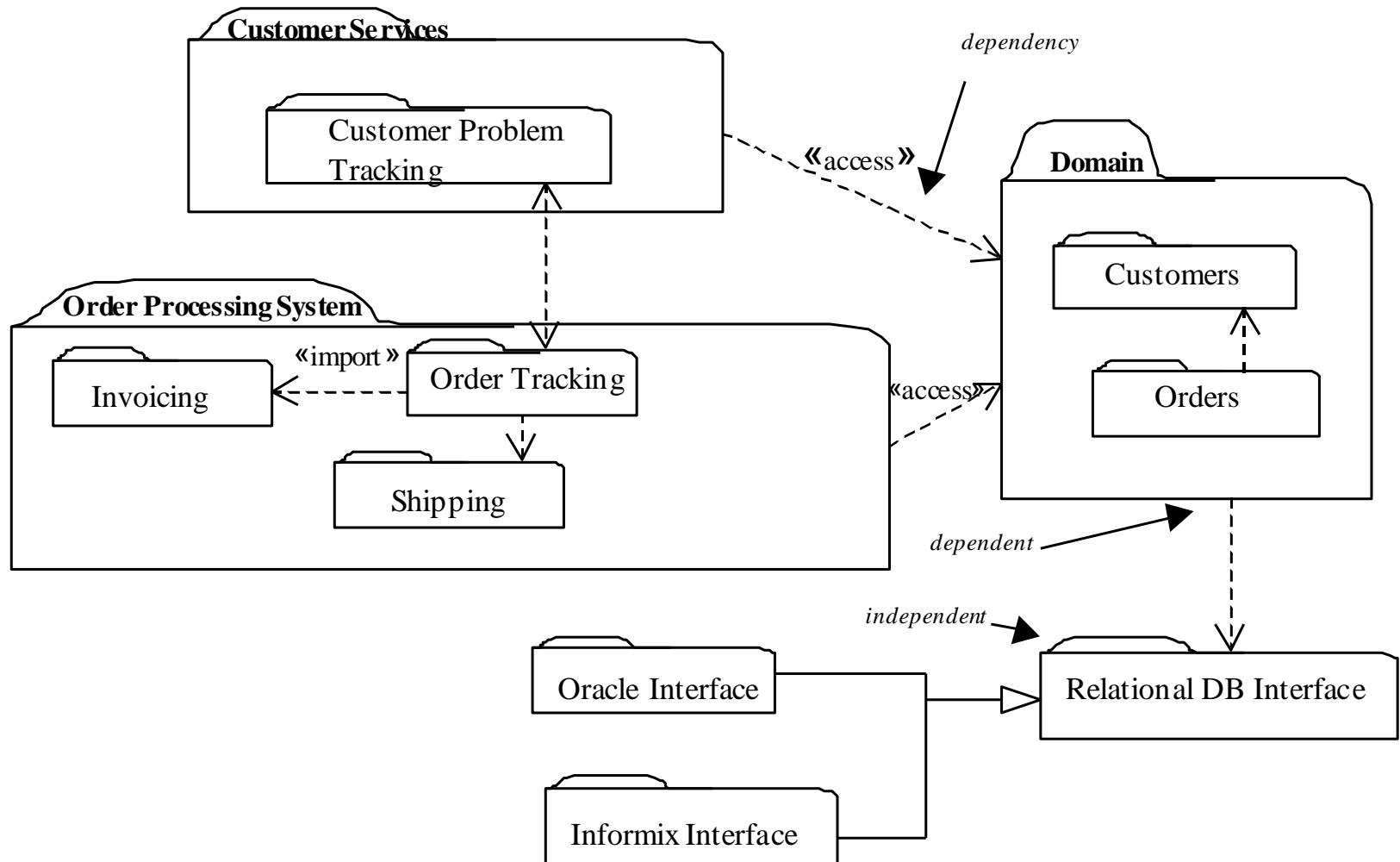




# Example Deployment Diagram



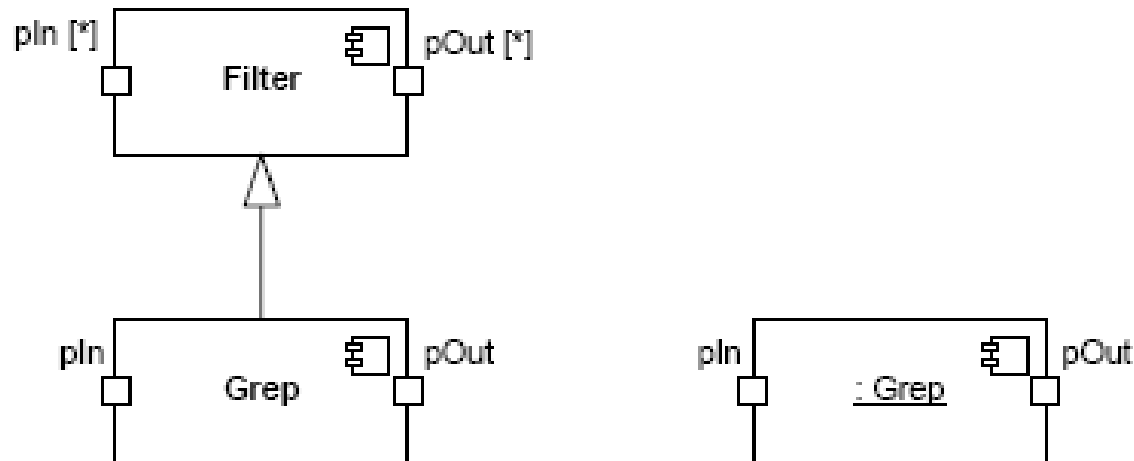
# Example – Module view



# Component and Connector View

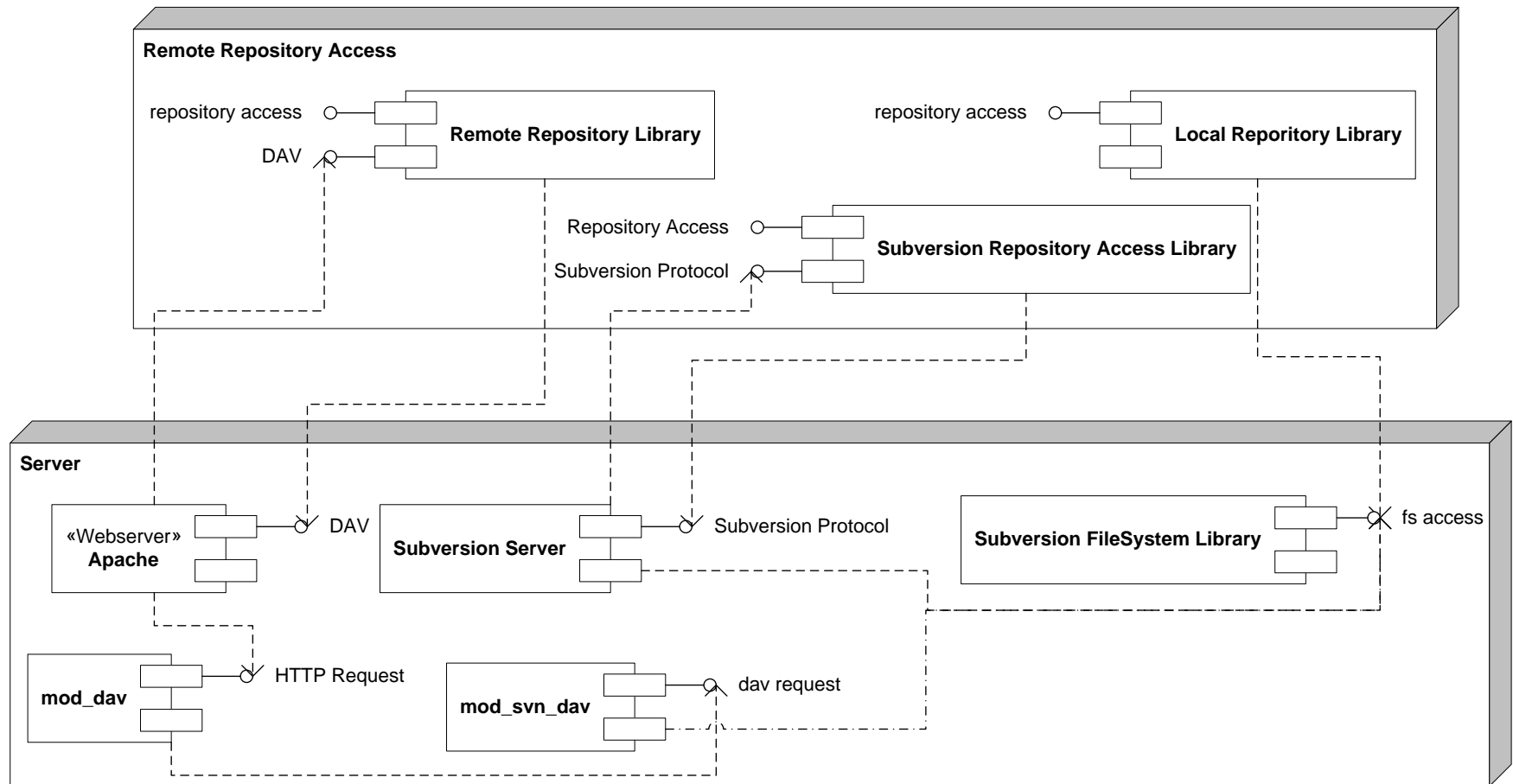


Components

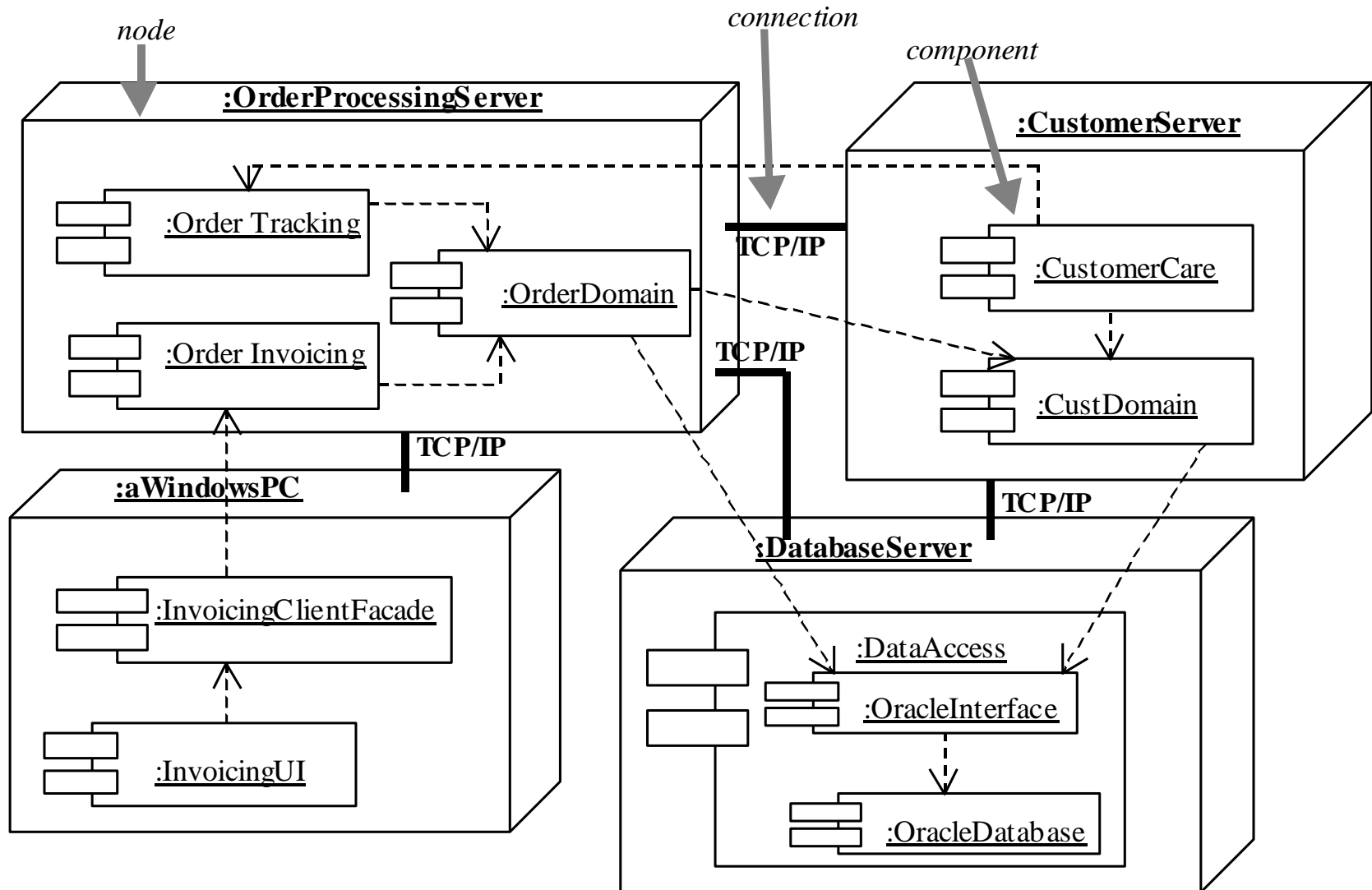


*C&C Types as UML Component Types and C&C Instances as UML Component Instances*

# Component and Connector – Client Server View



# Example Allocation view (Deployment)

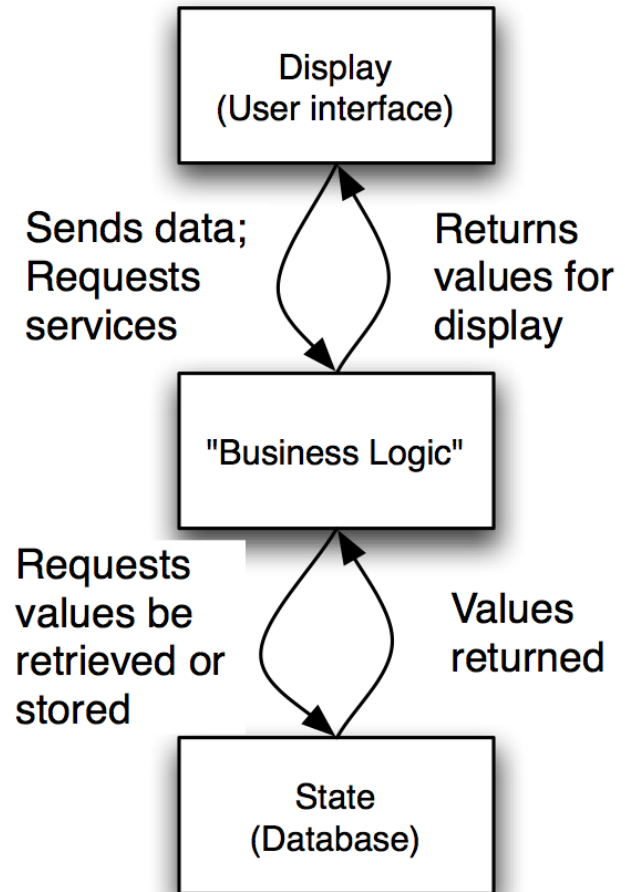


# Architectural Patterns

- The notion of patterns can be applied to software architecture.
  - These are called *architectural patterns* or *architectural styles*.
  - Each allows you to design flexible systems using components
    - The components are as independent of each other as possible.
  - An architectural pattern is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears.
- Common Architectural Patterns
  - Multi-Layer Pattern
  - Client-Server Pattern
  - Broker Pattern
  - Transaction-Processing Pattern
  - Pipe-and-Filter Pattern
  - Model-View-Controller (MVC) Pattern

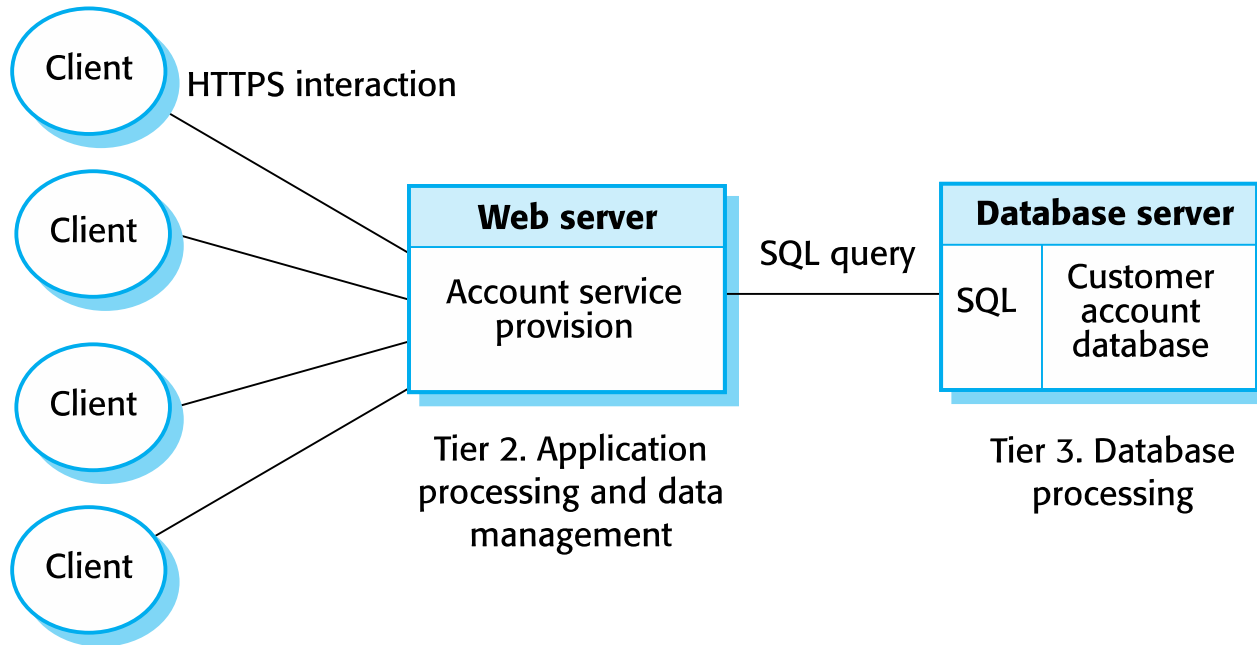
# State-Logic-Display: Three-Tiered Pattern

- Application Examples
  - Business applications
  - Multi-player games
  - Web-based applications



# Three-tier architecture for an Internet banking system

Tier 1. Presentation

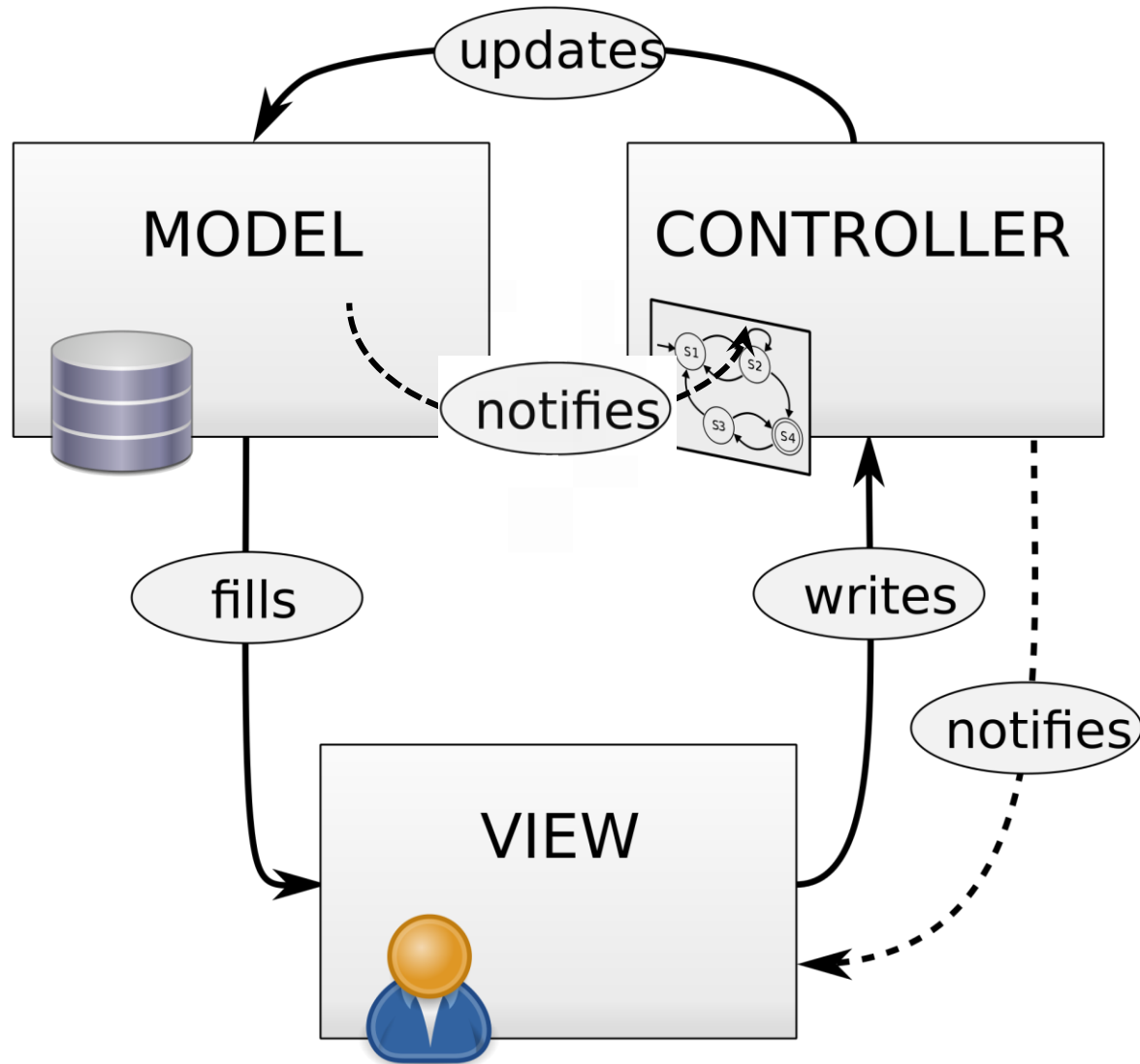




# Model-View-Controller (MVC) Pattern

- An architectural pattern used to help separate the user interface layer from other parts of the system (Separation between information, presentation and user interaction.)
  - The *model* contains the underlying classes whose instances are to be viewed and manipulated
  - The *view* contains objects used to render the appearance of the data from the model in the user interface
  - The *controller* contains the objects that control and handle the user's interaction with the view and the model

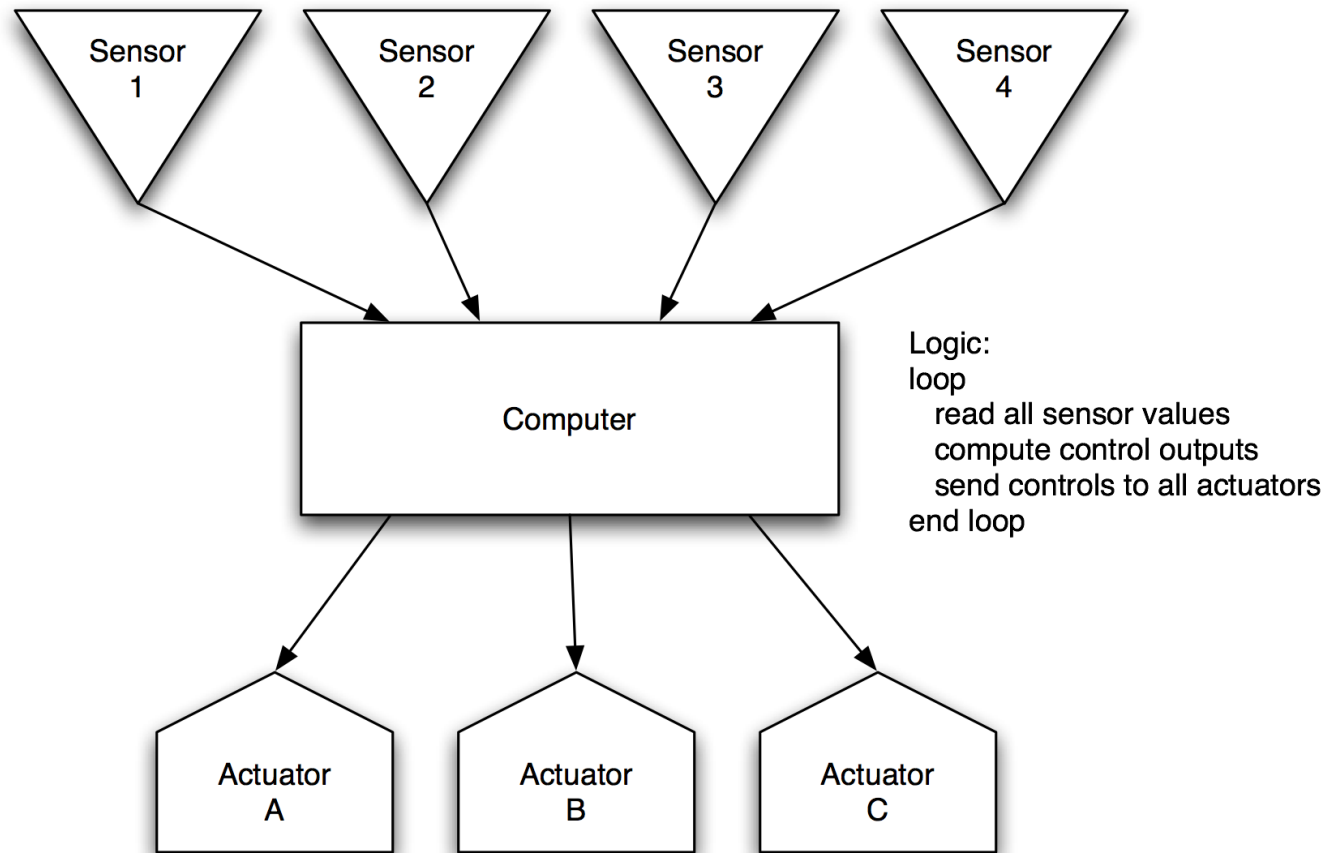
# Model-View-Controller



# The MVC architecture and design principles

- *Divide and conquer*: The three components can be somewhat independently designed.
- *Increase cohesion*: The components have stronger layer cohesion than if the view and controller were together in a single UI layer.
- *Reduce coupling*: The communication channels between the three components are minimal.
- *Increase reuse*: The view and controller normally make extensive use of reusable components for various kinds of UI controls.
- *Design for flexibility*: It is usually quite easy to change the UI by changing the view, the controller, or both.
- *Design for testability*: You can test the application separately from the UI.

# Sense-Compute-Control

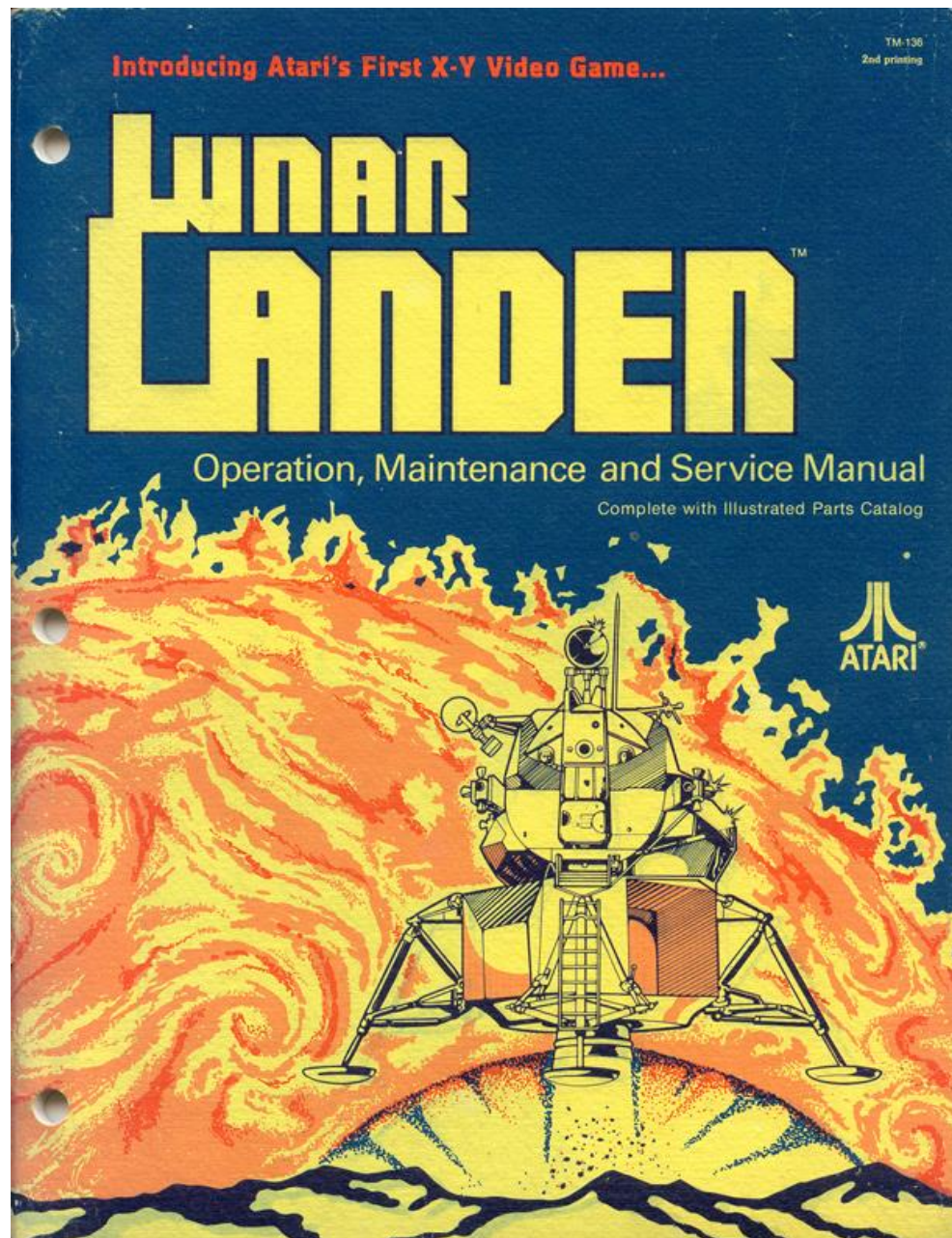


Objective: Structuring embedded control applications

# The Lunar Lander

- A simple computer game that first appeared in the 1960's
- Simple concept:
  - You (the pilot) control the descent rate of the Apollo-era Lunar Lander
    - Throttle setting controls descent engine
    - Limited fuel
    - Initial altitude and speed preset
    - If you land with a descent rate of  $< 5$  fps: you win (whether there's fuel left or not)
  - “Advanced” version: joystick controls attitude & horizontal motion

LL





# LL Arcade Game



# LL UI

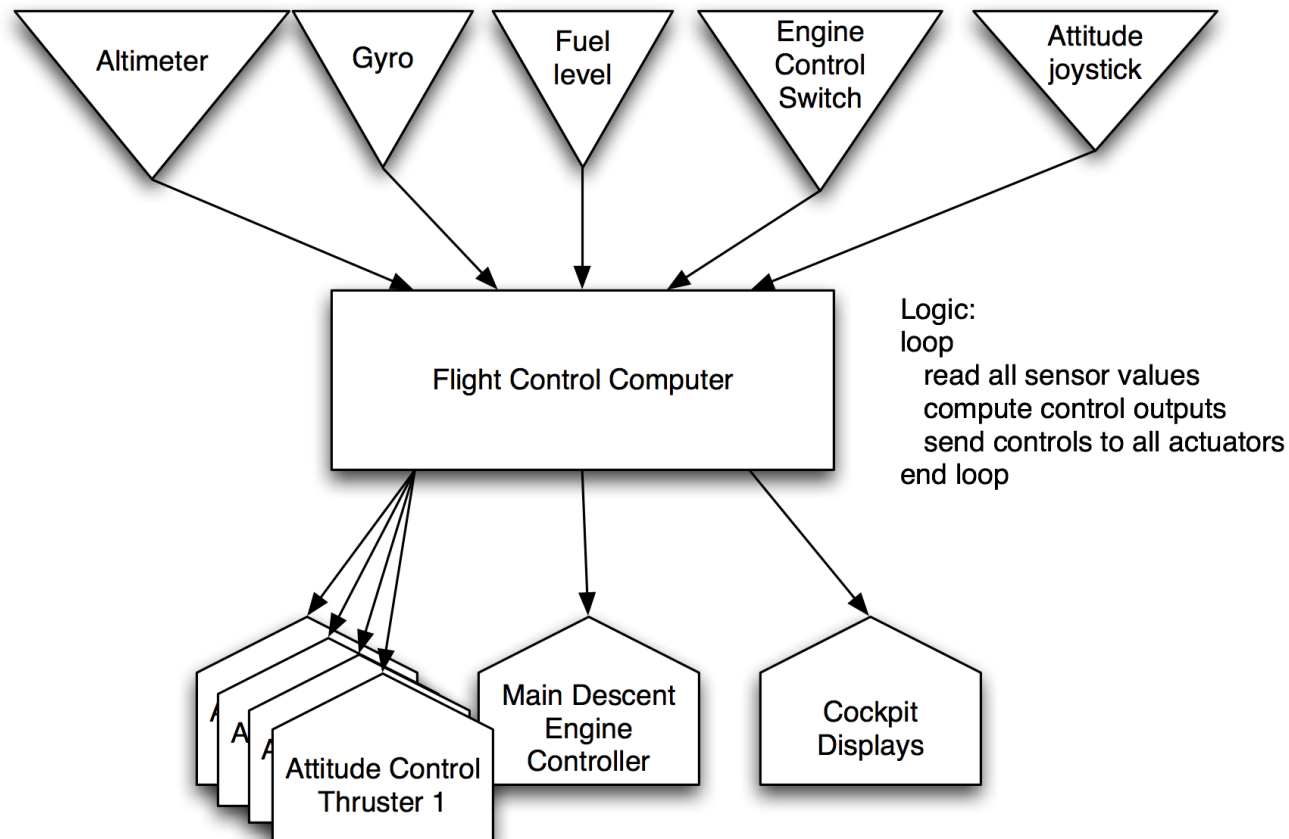




# LL Command Console



# Sense-Compute-Control LL



# Architectural Styles

- An architectural style is a named collection of architectural design decisions that
  - are applicable in a given development context
  - constrain architectural design decisions that are specific to a particular system within that context
  - elicit beneficial qualities in each resulting system
- Reflect less domain specificity than architectural patterns
- Many styles exist

## Building Style



ART DECO



BUNGALOW



CAPE COD



COLONIAL



CONTEMPORARY



CRAFTSMAN



CREOLE



DUTCH  
COLONIAL



FEDERAL



FRENCH  
PROVINCIAL



GEORGIAN



GOthic  
REVIVAL



GREEK  
REVIVAL



INTERNATIONAL



ITALIANATE



MONTEREY



NATIONAL



NEOCLASSICAL



PRAIRIE



PUEBLO



QUEEN ANNE



RANCH



REGENCY



SALTBOX



SECOND  
EMPIRE



SHED



SHINGLE



SHOTGUN



SPANISH  
ECLECTIC



SPLIT LEVEL



STICK



TUDOR



VICTORIAN

# GENERAL ROOFING TERMS

FROM THE INTERNATIONAL LIBRARY OF TECHNOLOGY, 33C, 1909

## Building “Subsystem” Style



SHED, PENT OR LEAN-TO ROOF



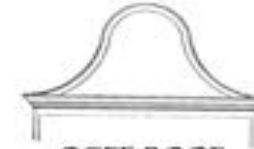
MANSARD ROOF



LOUVRE  
VENTILATOR



GABLE ROOF



OGE ROOF



SURMOUNTED DOME



CURB OR  
GAMBREL ROOF



SEMICIRCULAR OR  
BARREL ROOF



SEMICIRCULAR DOME



HIP ROOF



ELLIPSOIDAL DOME



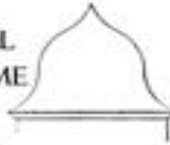
HIP AND VALLEY ROOF



SEGMENTED DOME



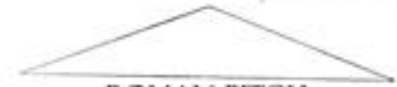
DORMER  
WINDOW



BELL  
DOME



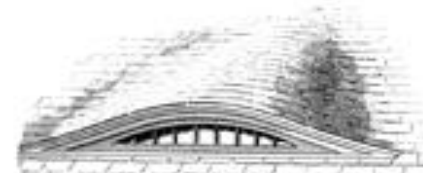
GRECIAN PITCH



ROMAN PITCH



A SQUINT



EYEBROW DORMER

ELIZABETHAN PITCH

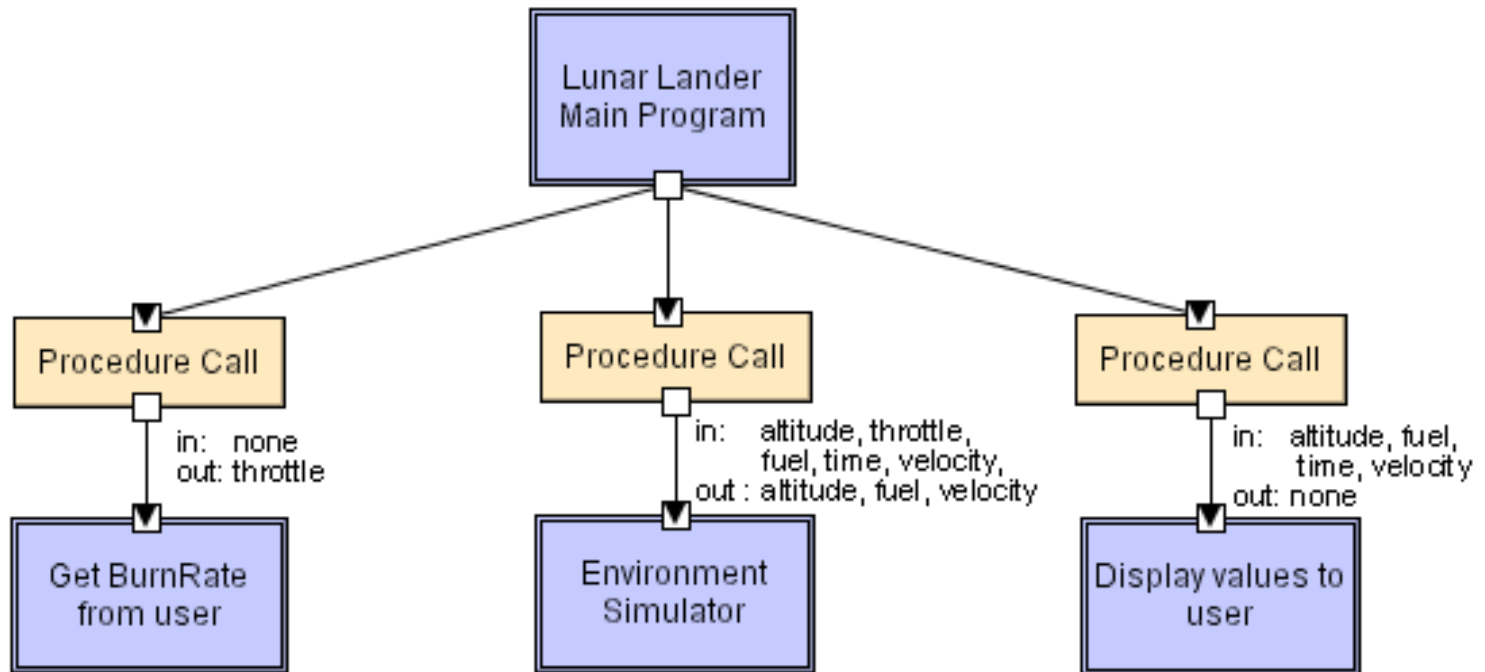


GOthic PITCH

# Some Common Styles

- Traditional, language-influenced styles
  - Main program and subroutines
  - Object-oriented
- Layered
  - Virtual machines
  - Client-server
- Data-flow styles
  - Batch sequential
  - Pipe and filter
- Shared memory
  - Blackboard
  - Rule based
- Interpreter
  - Interpreter
  - Mobile code
- Implicit invocation
  - Event-based
  - Publish-subscribe
- Peer-to-peer

# Main Program and Subroutines LL

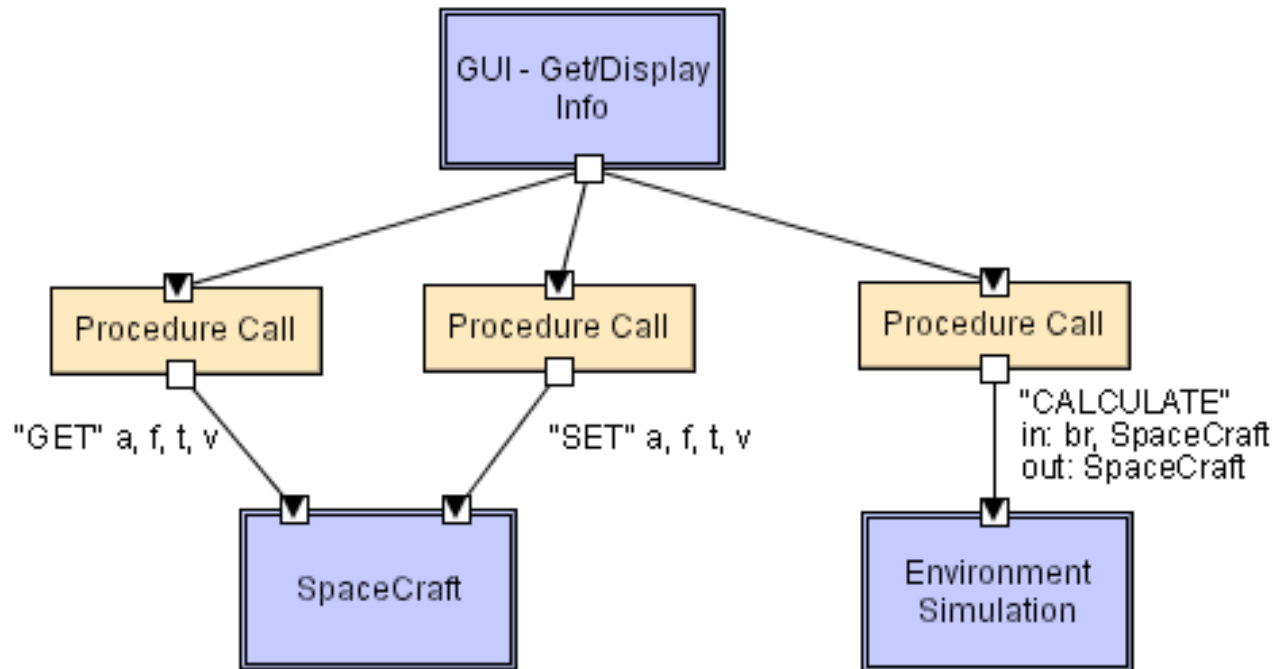


# Object-Oriented Style

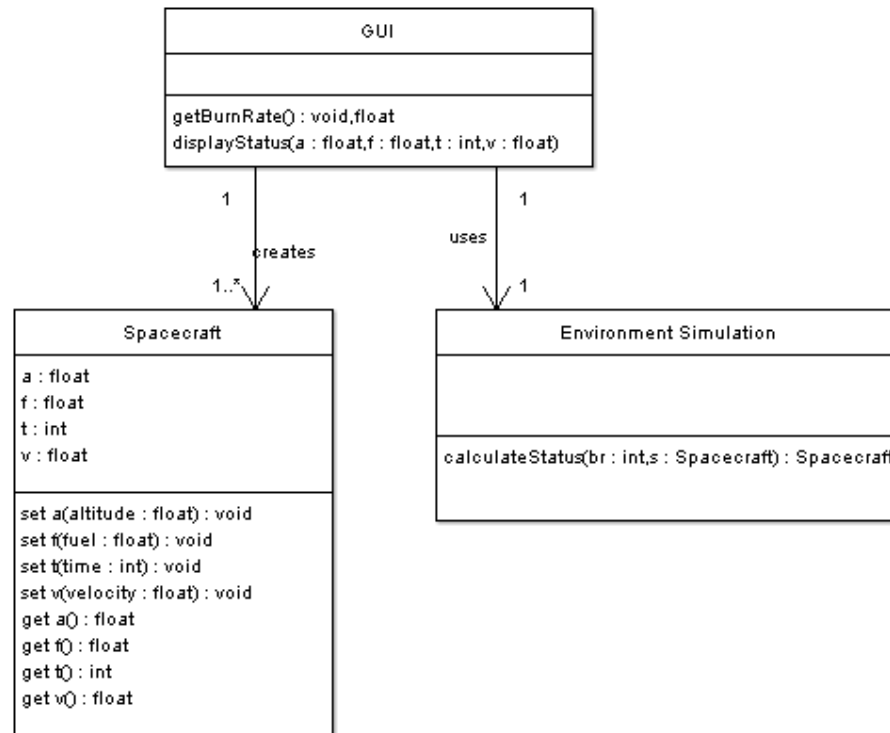
- Components are objects
  - Data and associated operations
- Connectors are messages and method invocations
- Style invariants
  - Objects are responsible for their internal representation integrity
  - Internal representation is hidden from other objects
- Advantages
  - “Infinite malleability” of object internals
  - System decomposition into sets of interacting agents
- Disadvantages
  - Objects must know identities of servers
  - Side effects in object method invocations



# Object-Oriented LL



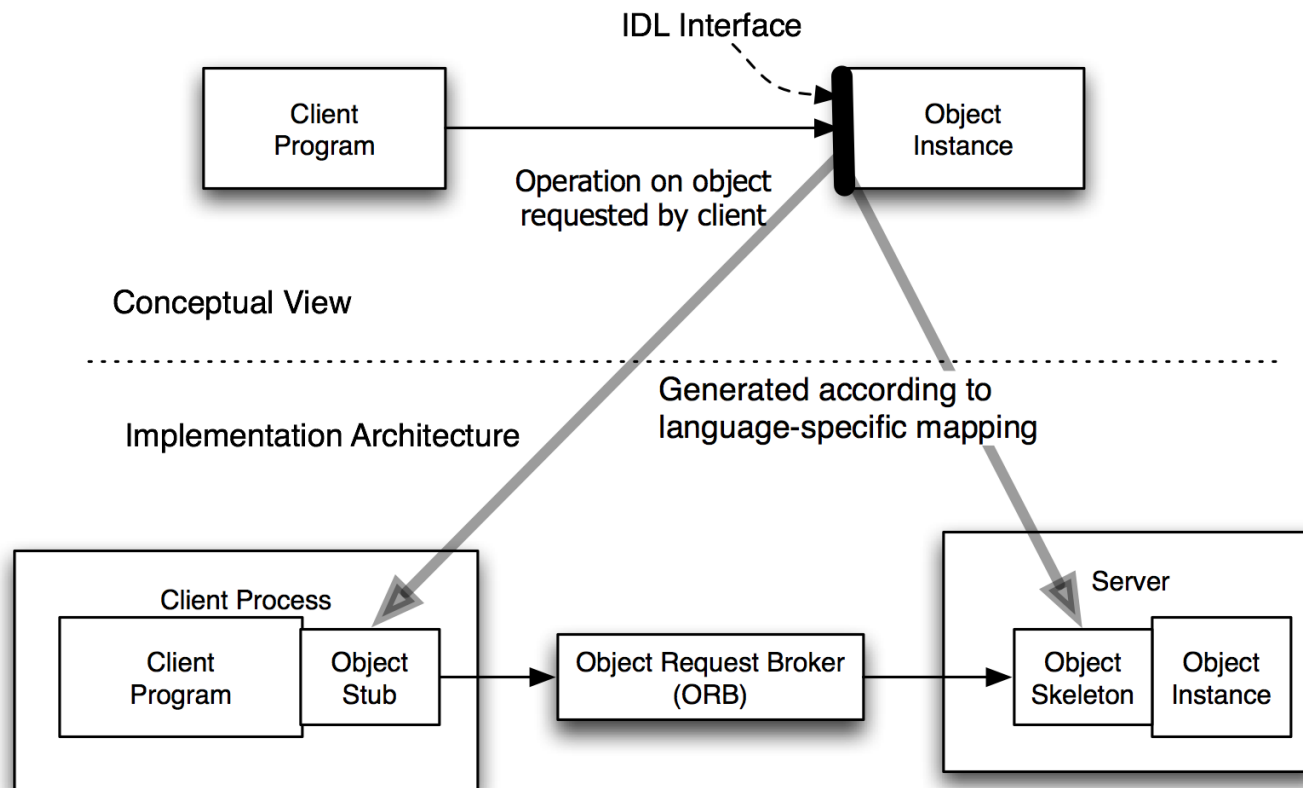
# OO/LL in UML



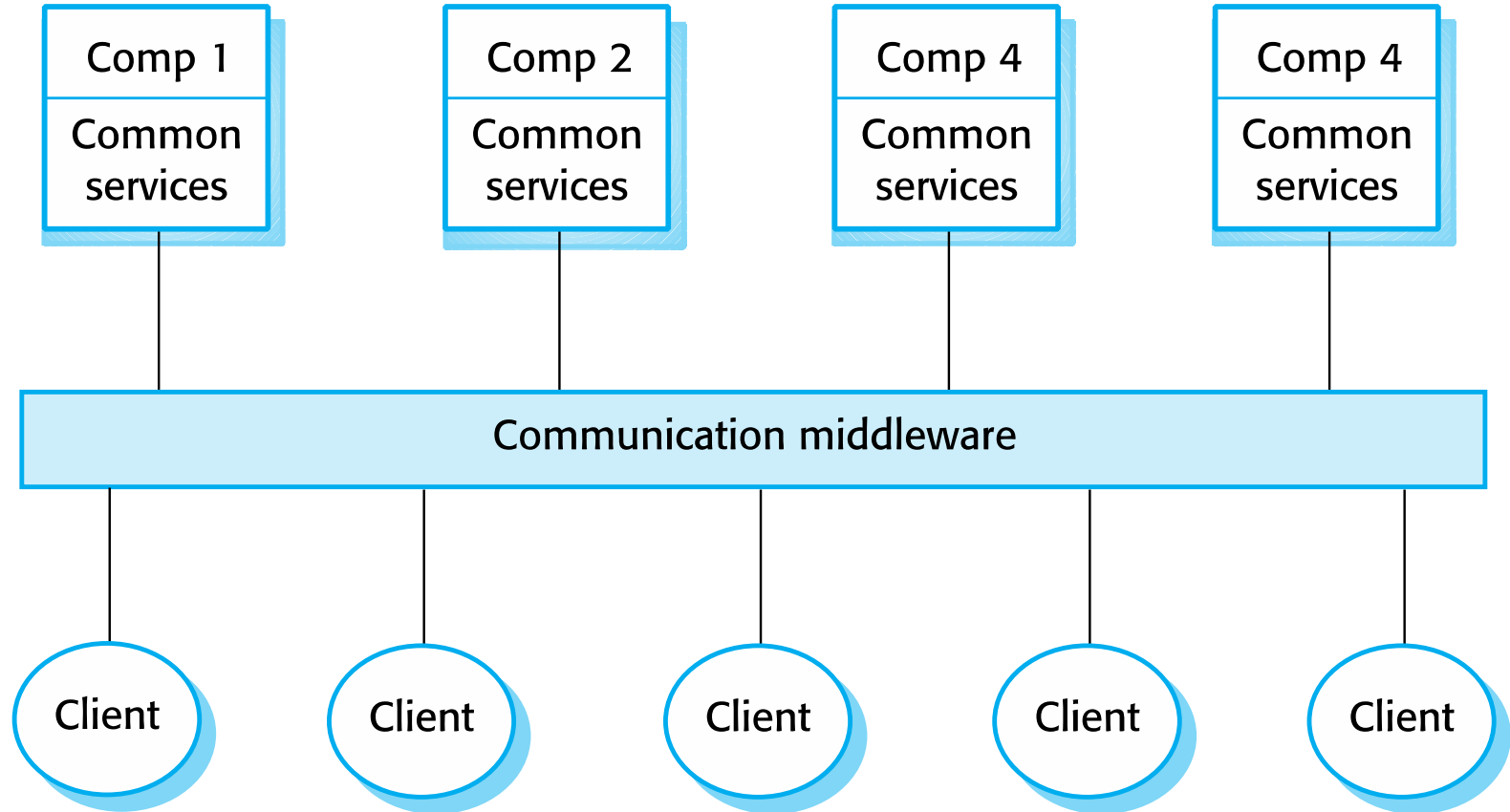
# Distributed Objects/Components

- “Objects” (coarse- or fine-grained) run on heterogeneous hosts, written in heterogeneous languages. Objects provide services through well-defined interfaces. Objects invoke methods across host, process, and language boundaries via remote procedure calls (RPCs).
- Components: Objects (software components exposing services through well-defined provided interfaces)
- Connector: (Remote) Method invocation
- Data Elements: Arguments to methods, return values, and exceptions
- Topology: General graph of objects from callers to callees.
- Additional constraints imposed: Data passed in remote procedure calls must be serializable. Callers must deal with exceptions that can arise due to network or process faults.
- Location, platform, and language “transparency”. **CAUTION**

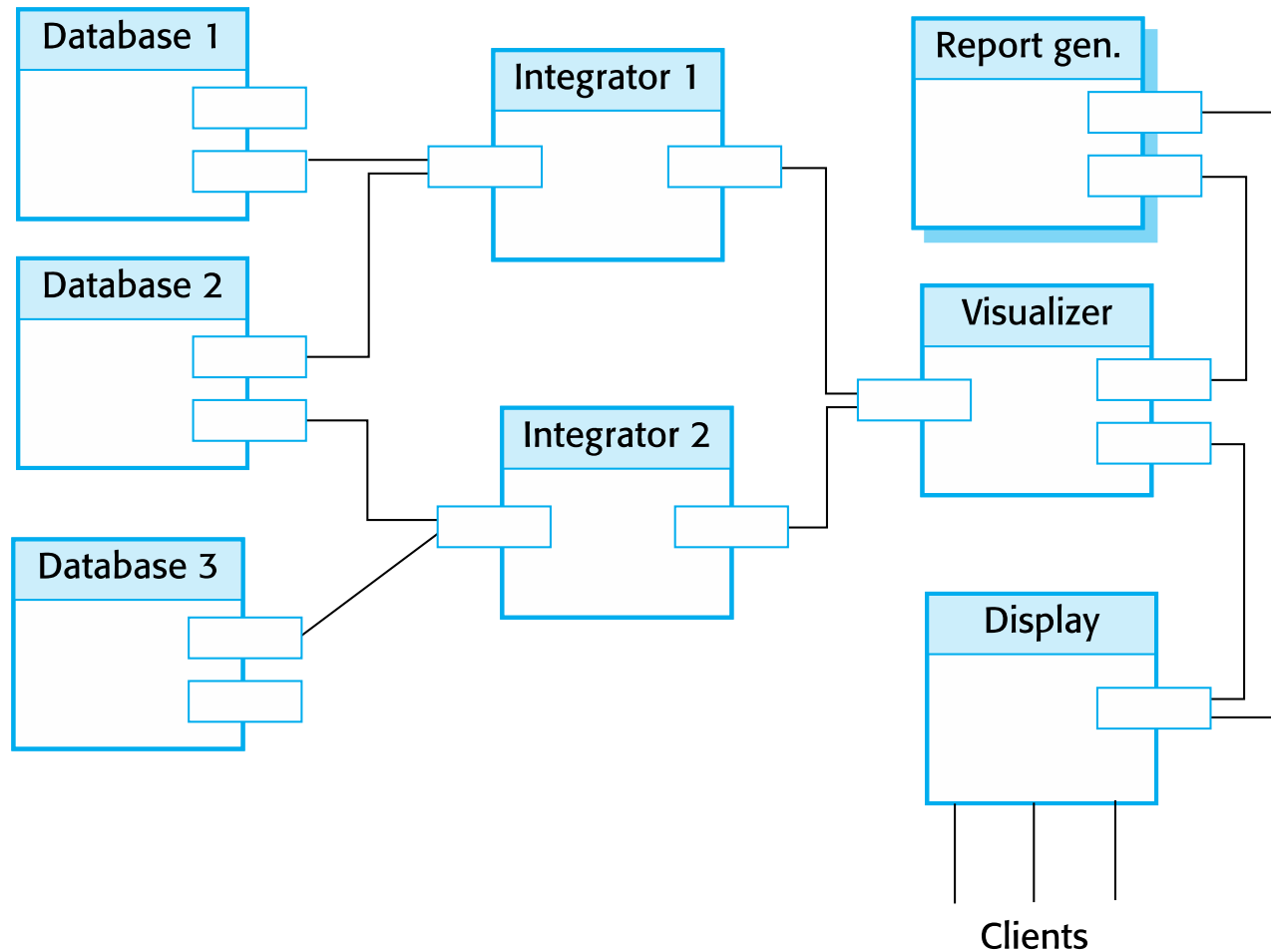
# Example DCA: CORBA



# A More General View



# DCA for a data mining system



# Layered Style

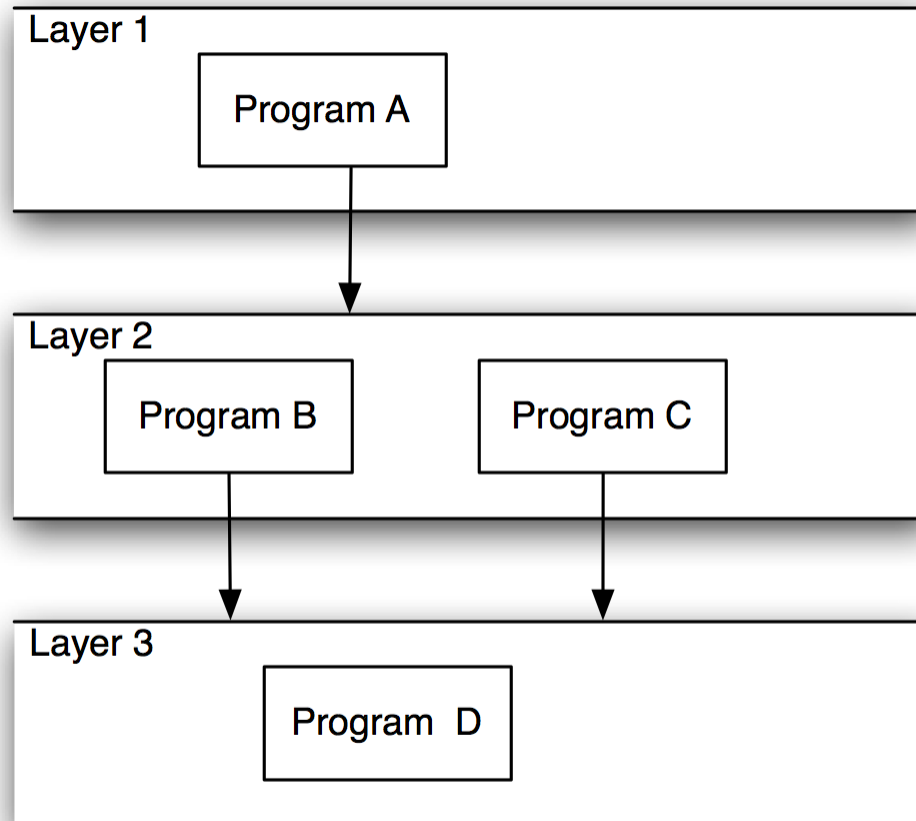
- Hierarchical system organization
- Each layer exposes an interface (API) to be used by above layers
- Each layer acts as a
  - *Server*: service provider to layers “above”
  - *Client*: service consumer of layer(s) “below”
- Connectors are protocols of layer interaction
- Example: operating systems
- *Virtual machine* style results from fully opaque layers

# Layered Style (cont'd)

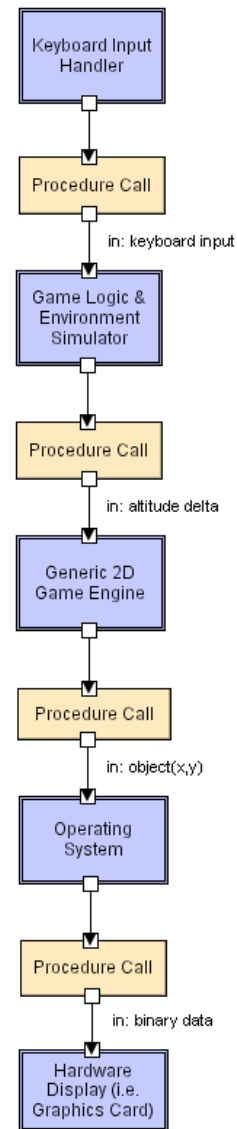
- Advantages
  - Increasing abstraction levels
  - Evolvability
  - Changes in a layer affect at most two layers
  - Different implementations are allowed as long as interface is preserved
  - Standardized layer interfaces for libraries/frameworks
- Disadvantages
  - Not universally applicable
  - Performance
  - Layers may have to be skipped
  - Determining the correct abstraction level



# Layered Systems/Virtual Machines



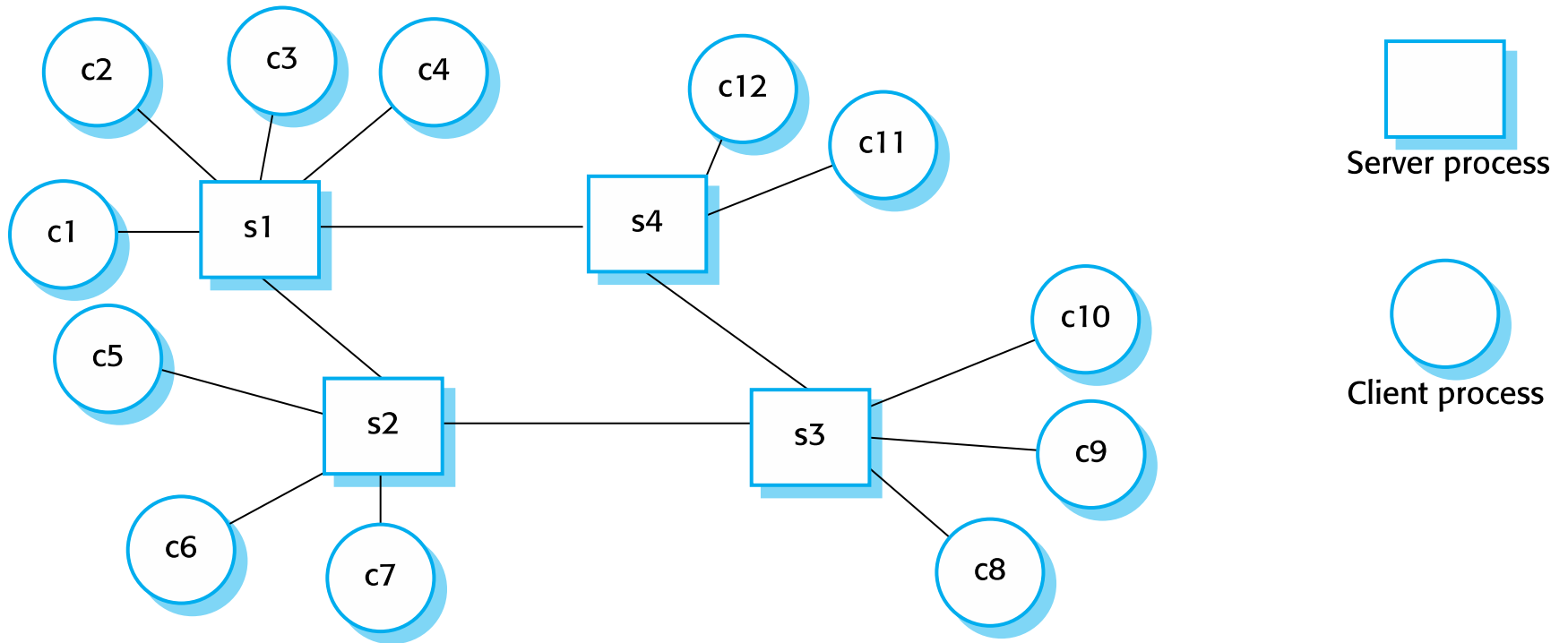
# Layered LL



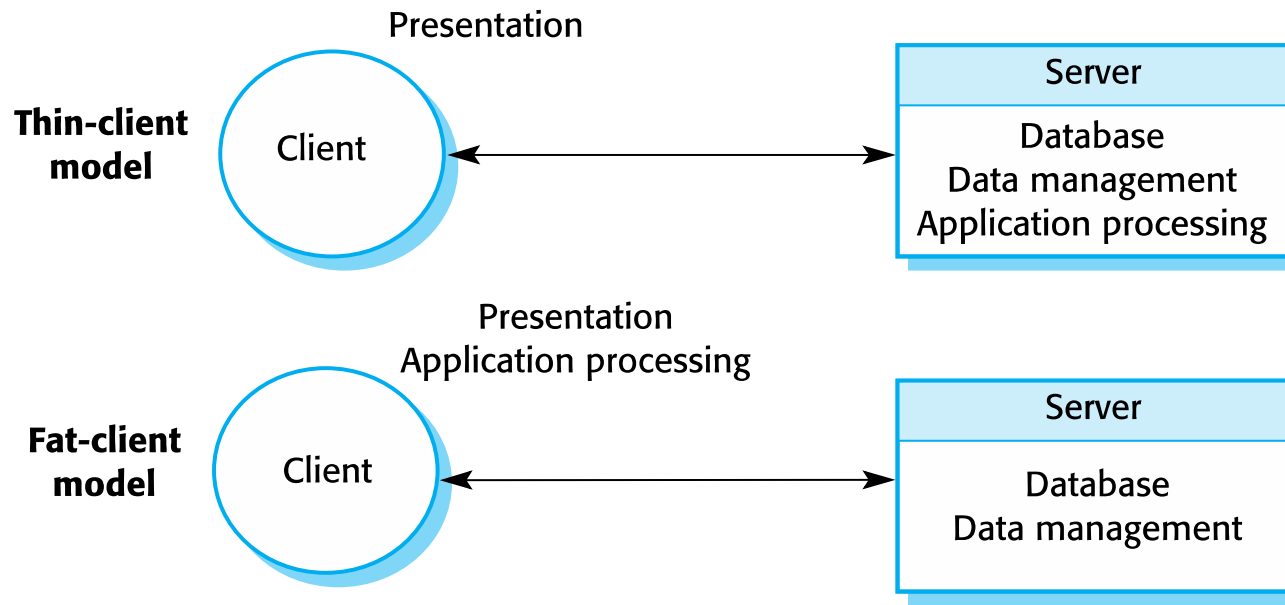
# Client-Server Style

- Components are clients and servers
- Servers do not know number or identities of clients
- Clients know server's identity
- Connectors are RPC-based network interaction protocols

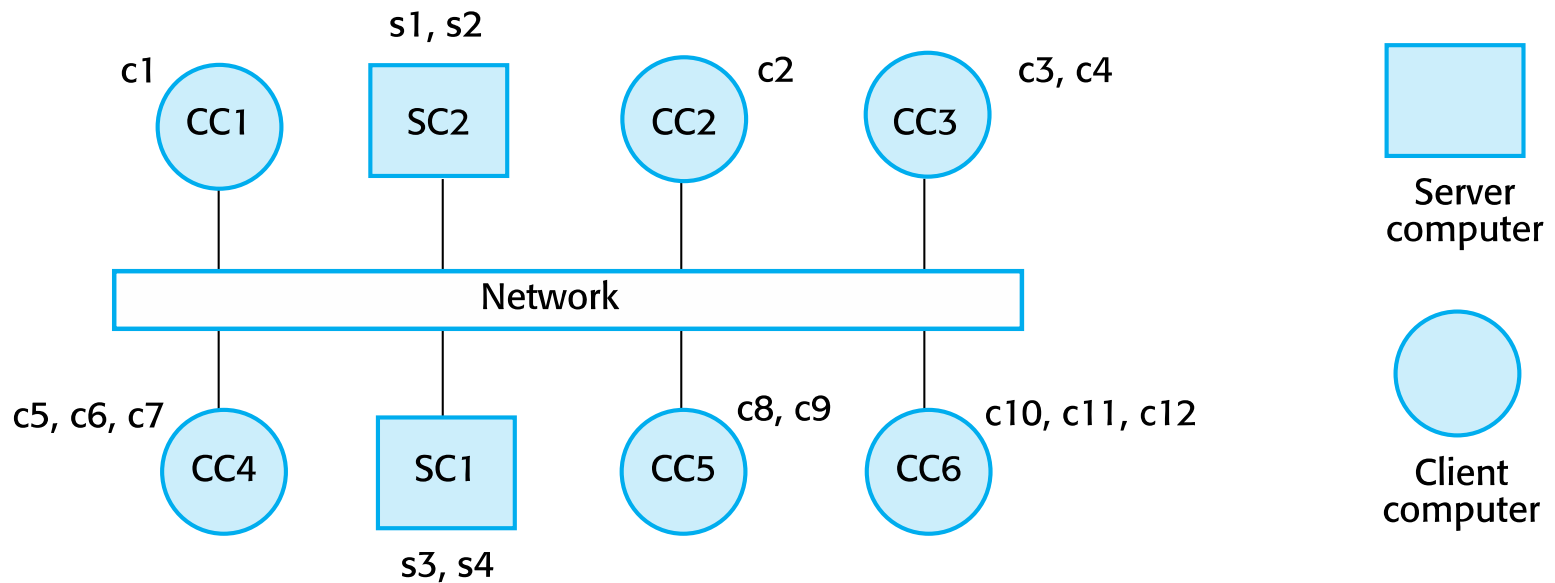
# Client-server interaction



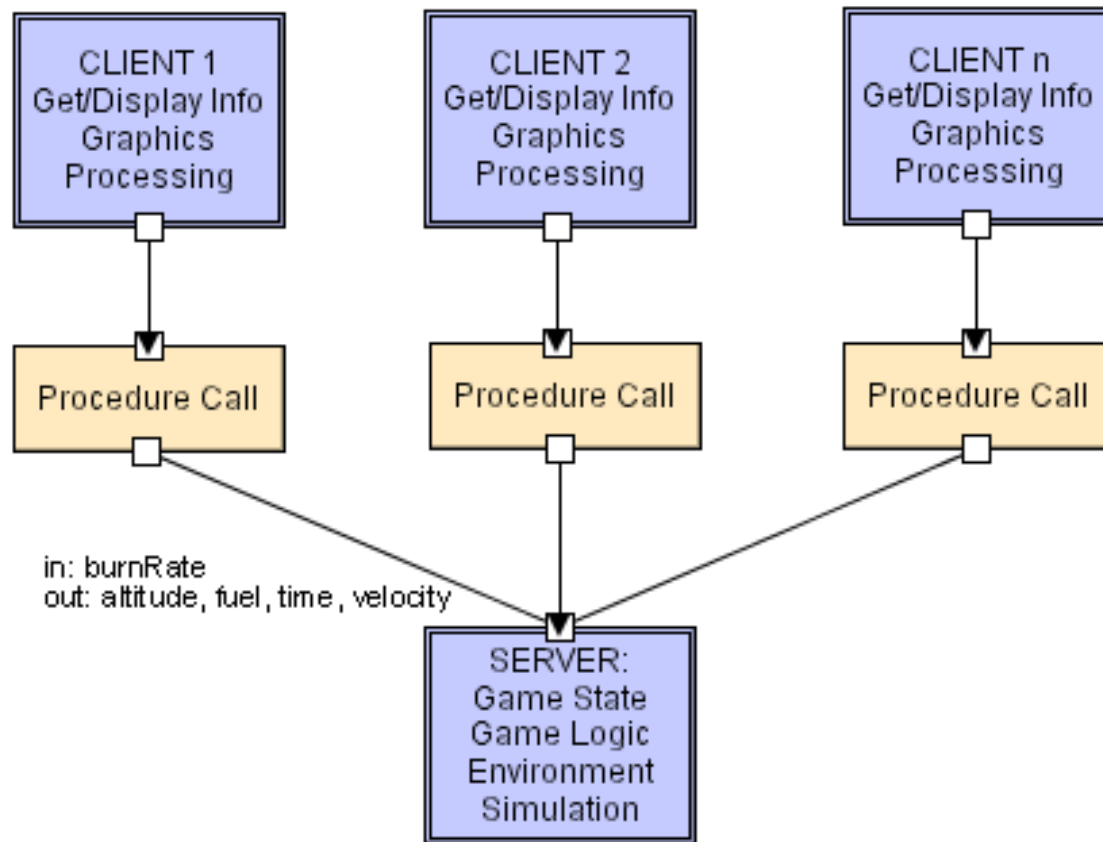
# Client-Server Variations



# Mapping of clients and servers to networked computers



# Client-Server LL



# Dataflow: Pipe and Filter Style

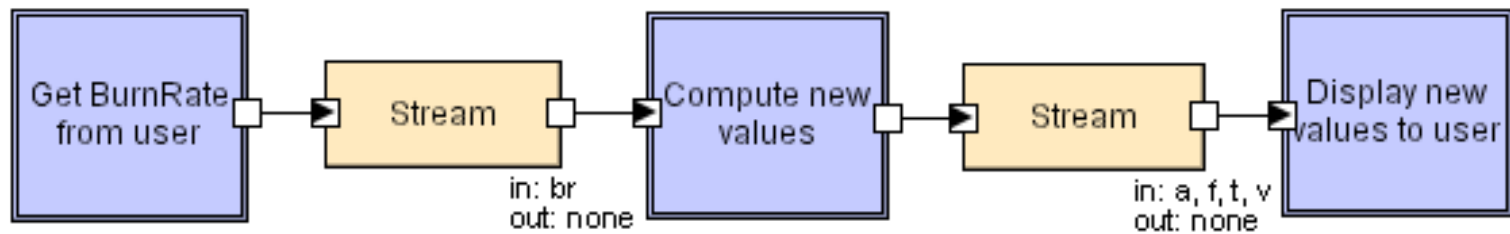
- Components are filters
  - Transform input data streams into output data streams
  - Possibly incremental production of output
- Connectors are pipes
  - Conduits for data streams
- Style invariants
  - Filters are independent (no shared state)
  - Filter has no knowledge of up- or down-stream filters
- Examples
  - UNIX shell                      signal processing
  - Distributed systems              parallel programming
- Example: `ls invoices | grep -e August | sort`



# Pipe and Filter (cont'd)

- Advantages
  - System behavior is a succession of component behaviors
  - Filter addition, replacement, and reuse
    - Possible to hook any two filters together
  - Certain analyses
    - Throughput, latency, deadlock
  - Concurrent execution
- Disadvantages
  - Batch organization of processing
  - Interactive applications
  - Lowest common denominator on data transmission

# Pipe and Filter LL



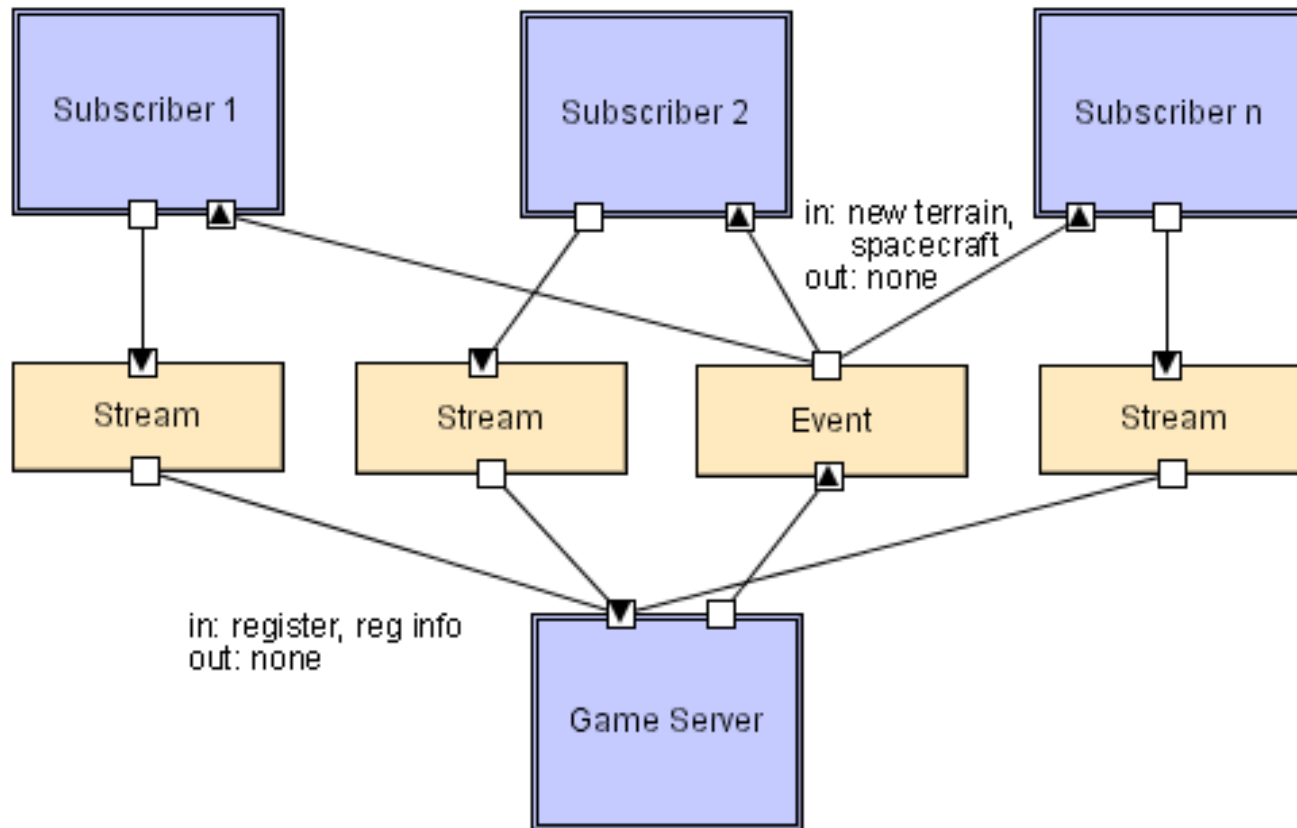
# Publish-Subscribe

Subscribers register/deregister to receive specific messages or specific content. Publishers broadcast messages to subscribers either synchronously or asynchronously.

## Publish-Subscribe (cont'd)

- Components: Publishers, subscribers, proxies for managing distribution
- Connectors: Typically a network protocol is required. Content-based subscription requires sophisticated connectors.
- Data Elements: Subscriptions, notifications, published information
- Topology: Subscribers connect to publishers either directly or may receive notifications via a network protocol from intermediaries
- Qualities yielded: Highly efficient one-way dissemination of information with very low-coupling of components

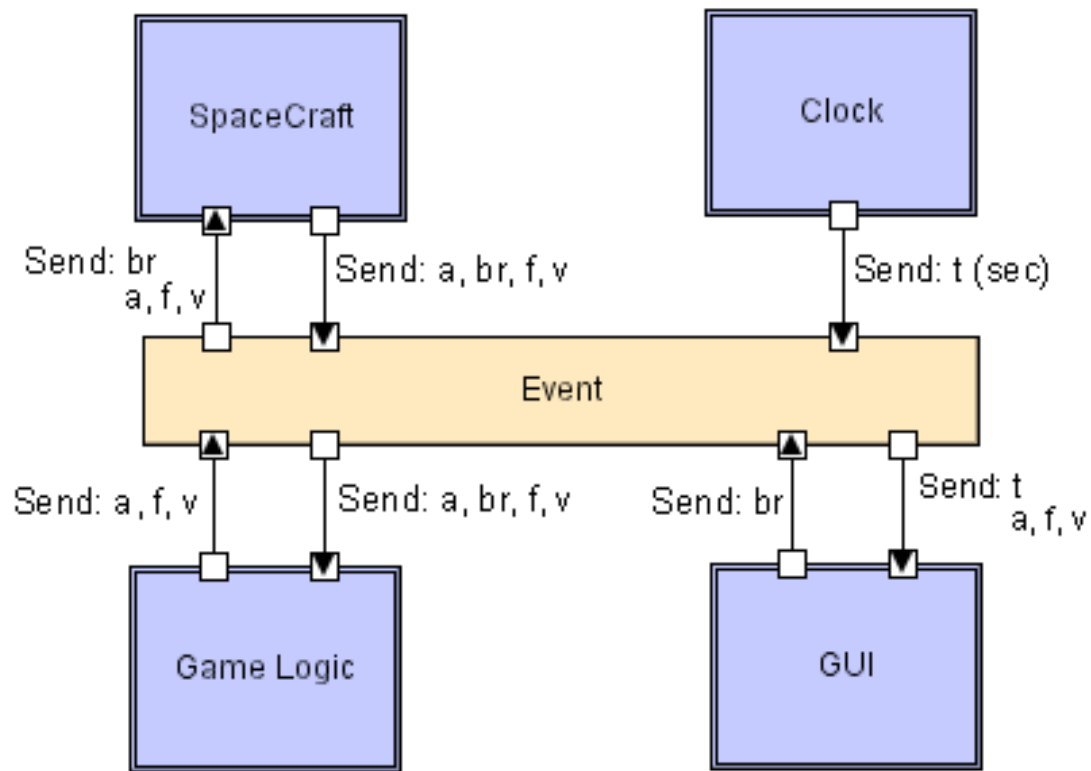
# Pub-Sub LL



# Event-Based Style

- Independent components asynchronously emit and receive events communicated over event buses
- Components: Independent, concurrent event generators and/or consumers
- Connectors: Event buses (at least one)
- Data Elements: Events – data sent as a first-class entity over the event bus
- Topology: Components communicate with the event buses, not directly to each other.
- Variants: Component communication with the event bus may either be push or pull based.
- Highly scalable, easy to evolve, effective for highly distributed applications.

# Event-based LL



# Peer-to-Peer Style

- State and behavior are distributed among peers which can act as either clients or servers.
- Peers: independent components, having their own state and control thread.
- Connectors: Network protocols, often custom.
- Data Elements: Network messages



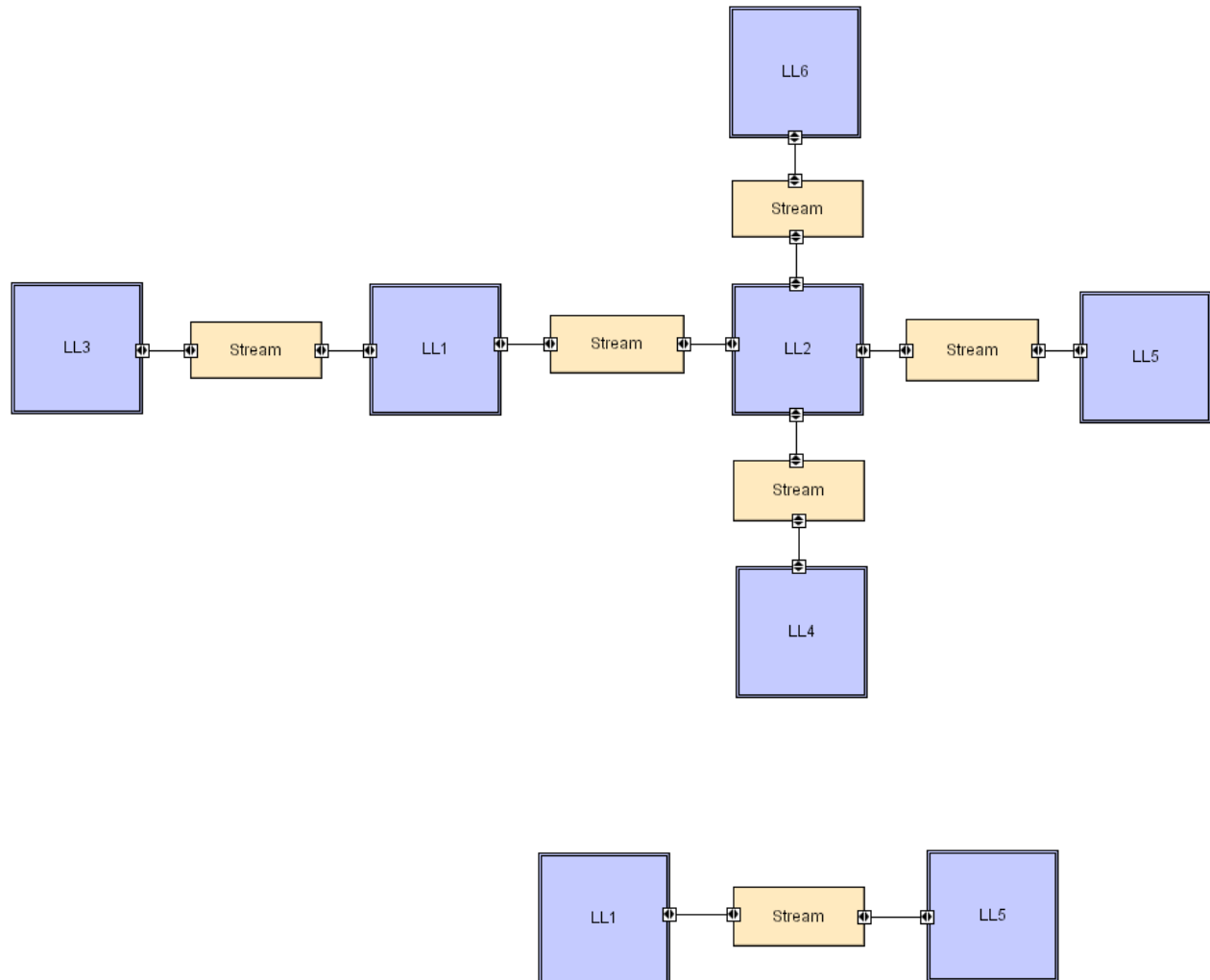
## Peer-to-Peer Style (cont'd)

- Topology: Network (may have redundant connections between peers); can vary arbitrarily and dynamically
- Supports decentralized computing with flow of control and resources distributed among peers.
- Highly robust in the face of failure of any given node.
- Scalable in terms of access to resources and computing power.
  - But caution on the protocol!

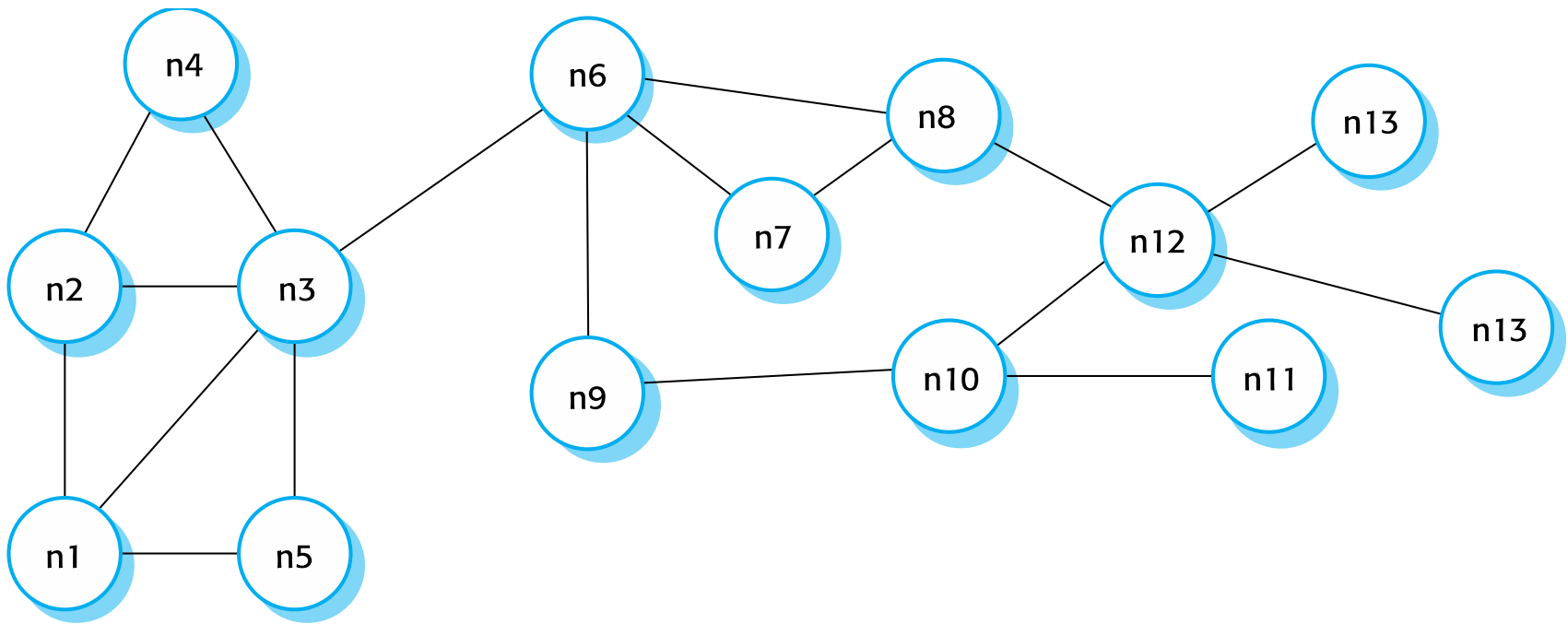
# Peer-to-peer systems

- File sharing systems based on the BitTorrent protocol
- Messaging systems such as Jabber
- Payments systems – Bitcoin
- Databases – Freenet is a decentralized database
- Phone systems – Viber
- Computation systems - SETI@home

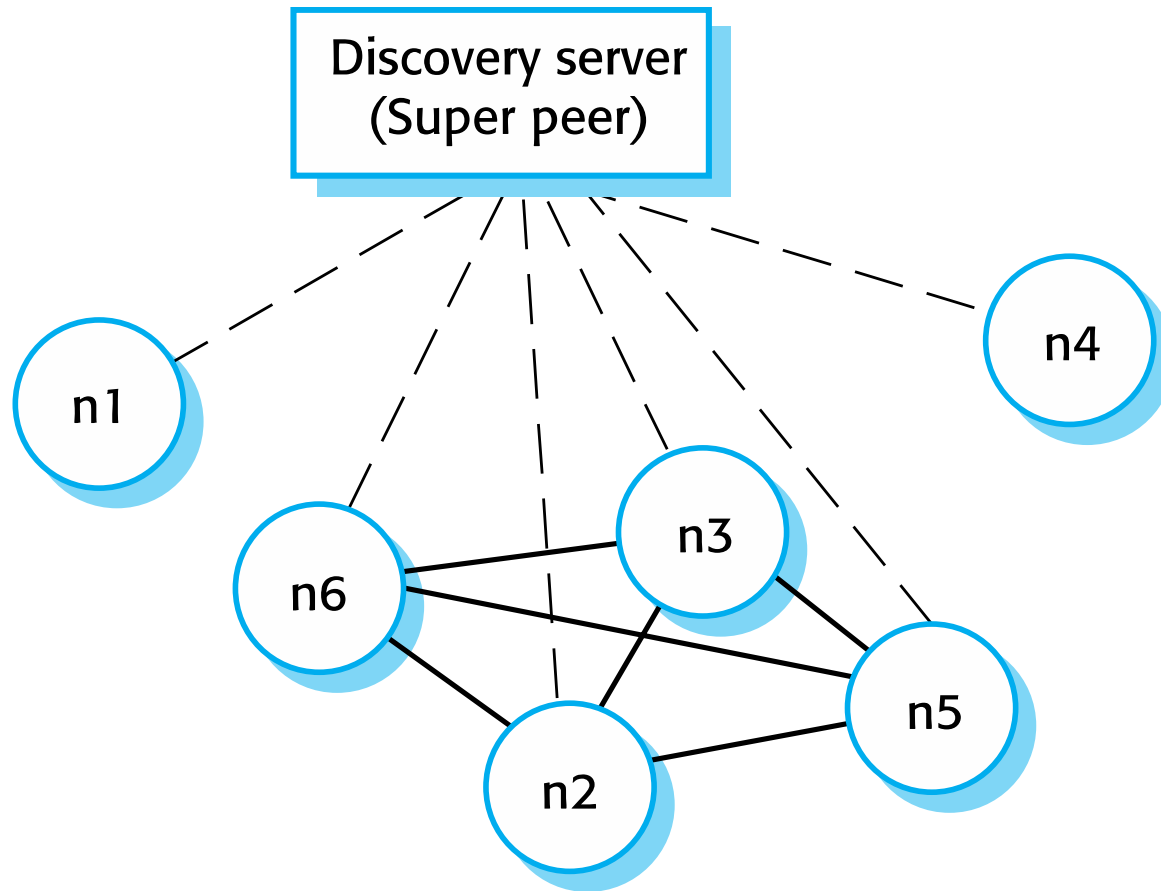
# Peer-to-Peer LL



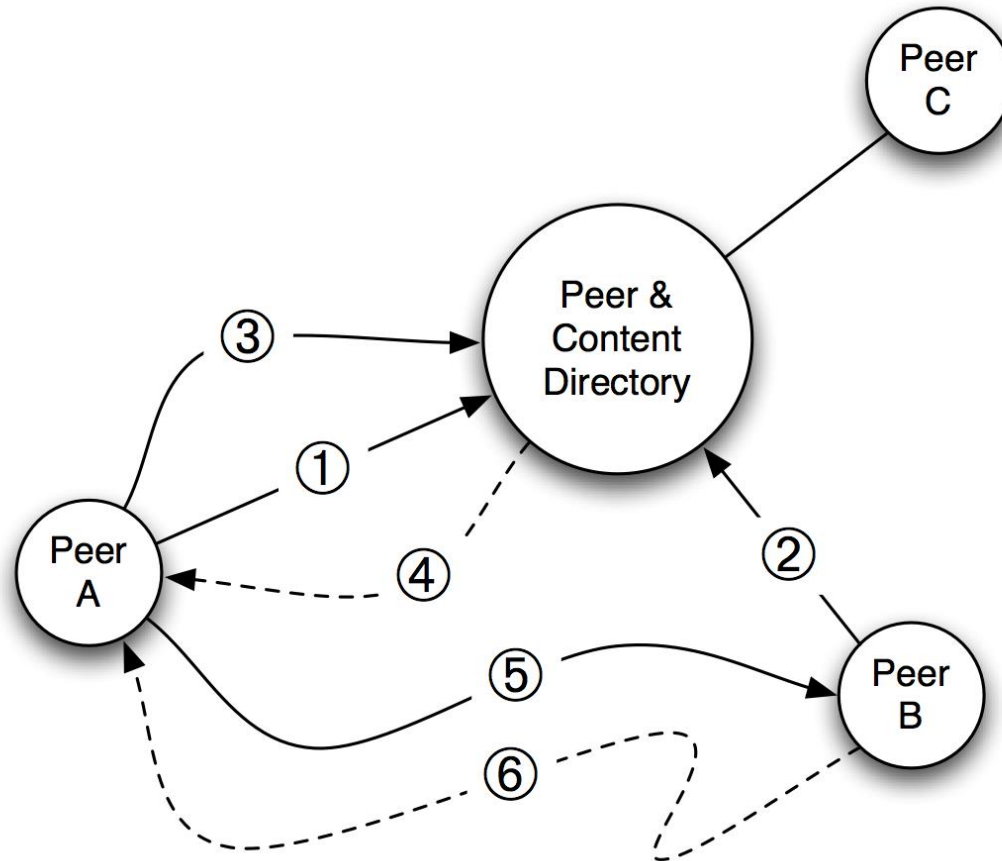
# A decentralized p2p architecture



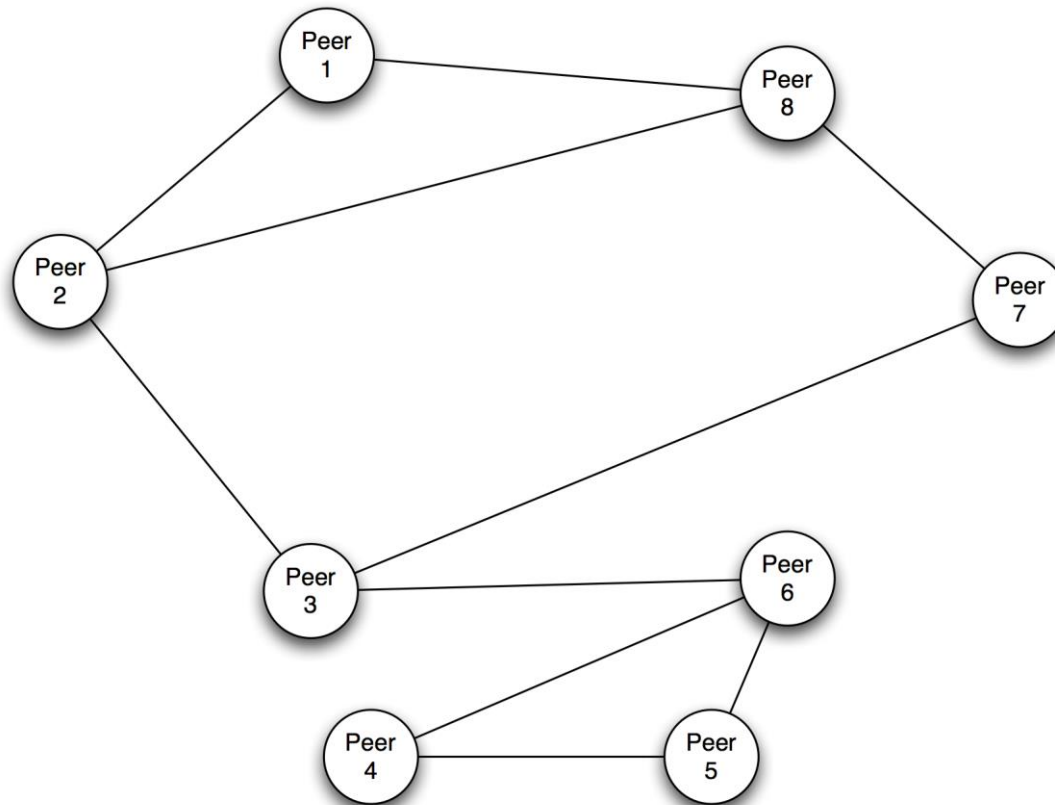
# A semicentralized p2p architecture



# Example P2P – Napster



# Example P2P – Gnutella (original)



# Example P2P – Skype

