



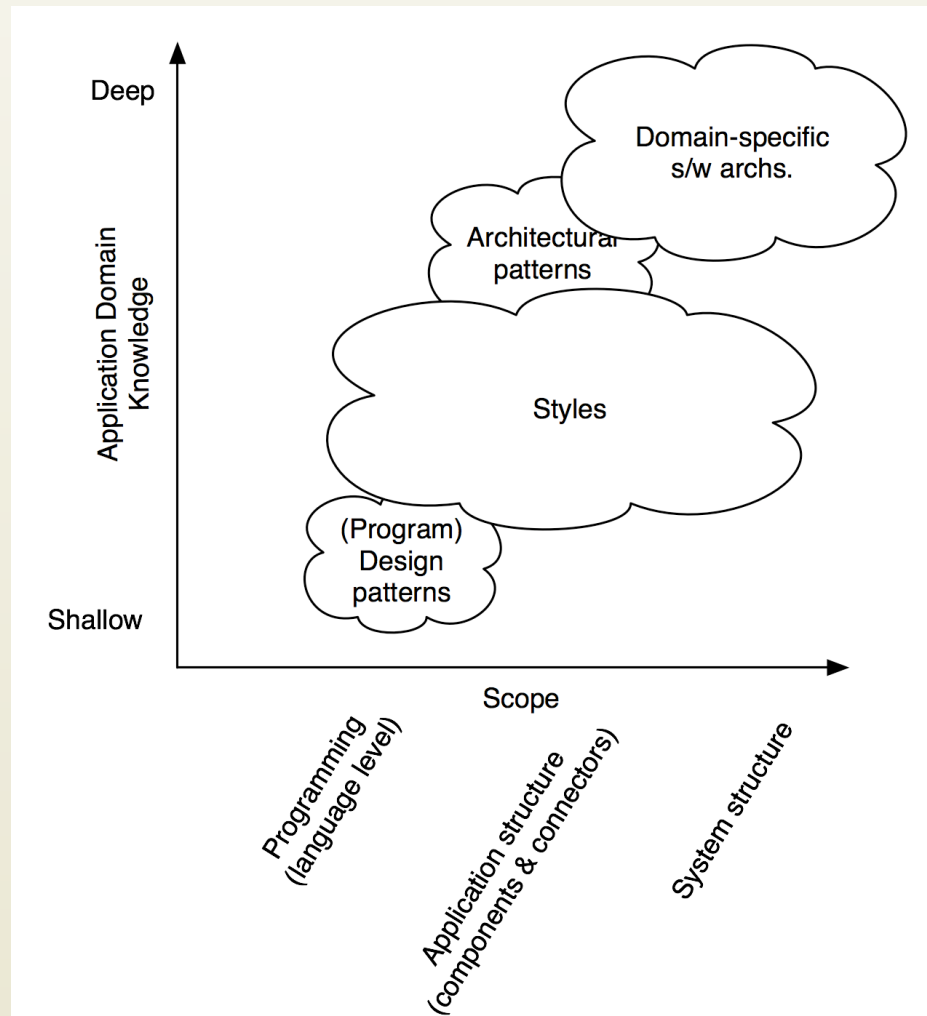
Architectural Patterns and Styles

Software Architecture
Lecture 4

Learning Objectives

- Delineate the role of Styles and Patterns in Software architecture
- Understand the role and benefits of architectural styles
- Understand and apply common styles in your designs
- Construct complex styles from simpler styles

Patterns, Styles, and DSSAs



The Lunar Lander: A Long-Running Example

- 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

Architectural Styles: Definition

- 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
- A primary way of characterizing lessons from experience in software system design
- Reflect less domain specificity than architectural patterns
- Useful in determining everything from subroutine structure to top-level application structure

Basic Properties of Styles

- A vocabulary of design elements
 - ◆ Component and connector types; data elements
 - ◆ e.g., pipes, filters, objects, servers
- A set of configuration rules
 - ◆ Topological constraints that determine allowed compositions of elements
 - ◆ e.g., a component may be connected to at most two other components
- A semantic interpretation
 - ◆ Compositions of design elements have well-defined meanings
- Possible analyses of systems built in a style

Benefits of Using Styles

- Design reuse
 - ◆ Well-understood solutions applied to new problems
- Code reuse
 - ◆ Shared implementations of invariant aspects of a style
- Understandability of system organization
 - ◆ A phrase such as “client-server” conveys a lot of information
- Interoperability
 - ◆ Supported by style standardization
- Style-specific analyses
 - ◆ Enabled by the constrained design space
- Visualizations
 - ◆ Style-specific depictions matching engineers’ mental models

Style Analysis Dimensions

- What is the design vocabulary?
 - ◆ Component and connector types
- What are the allowable structural patterns?
- What is the underlying computational model?
- What are the essential invariants of the style?
- What are common examples of its use?
- What are the (dis)advantages of using the style?
- What are the style's specializations?

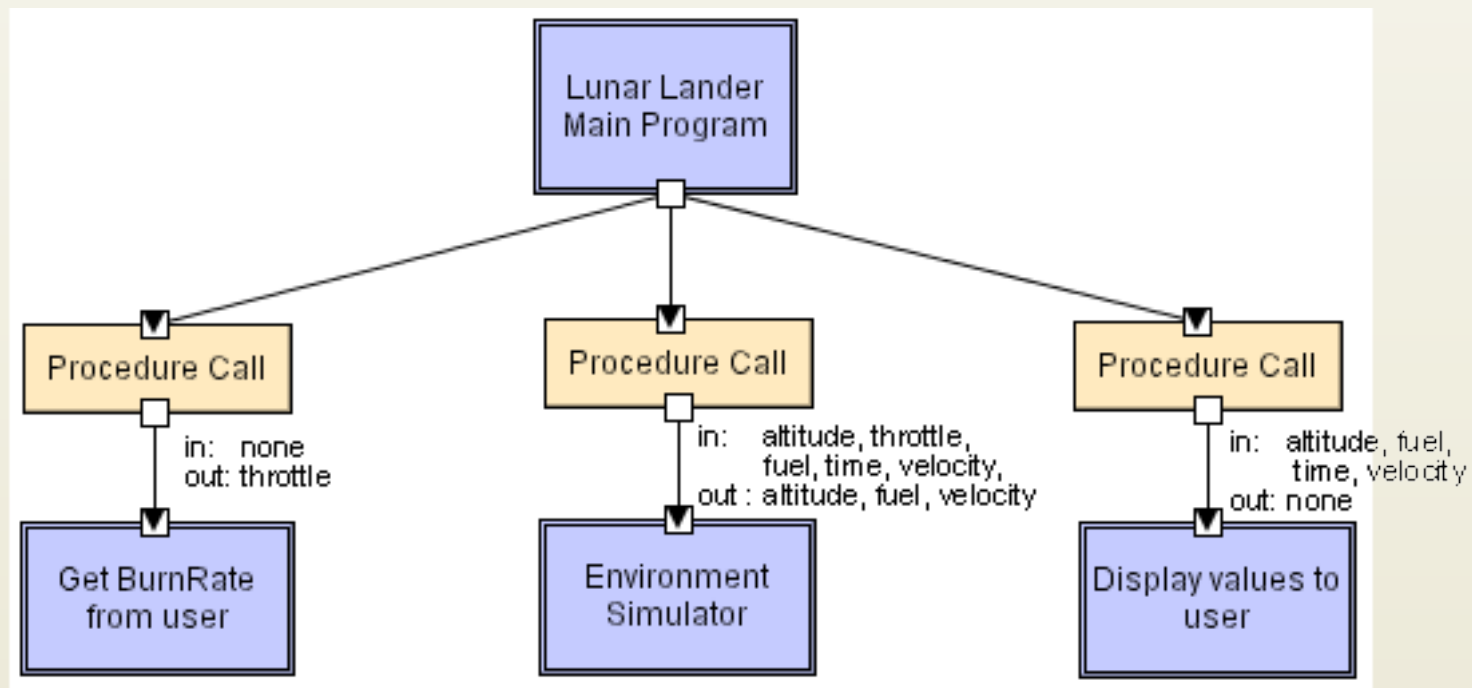
Learning Objectives

- Delineate the role of Styles and Patterns in Software architecture
- Understand the role and benefits of architectural styles
- Understand and apply common styles in your designs
- Construct complex styles from simpler styles

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
- Distributed Objects
 - ◆ C2
 - ◆ CORBA

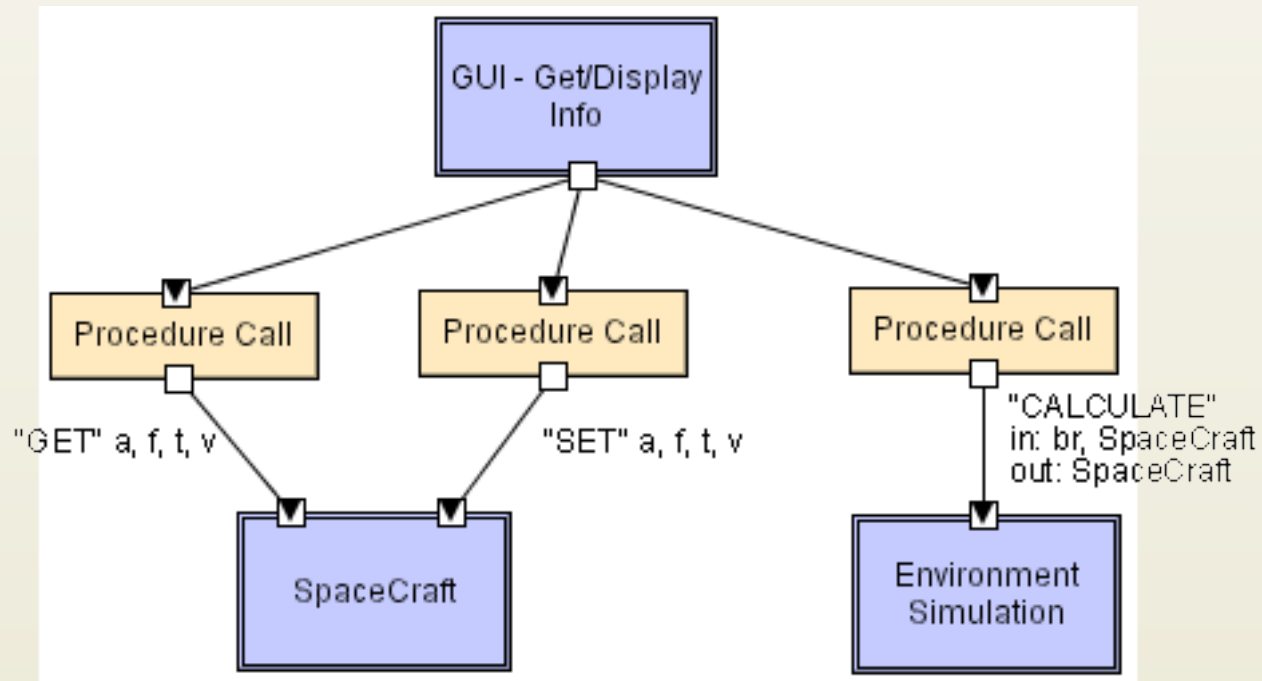
Main Program and Subroutines LL



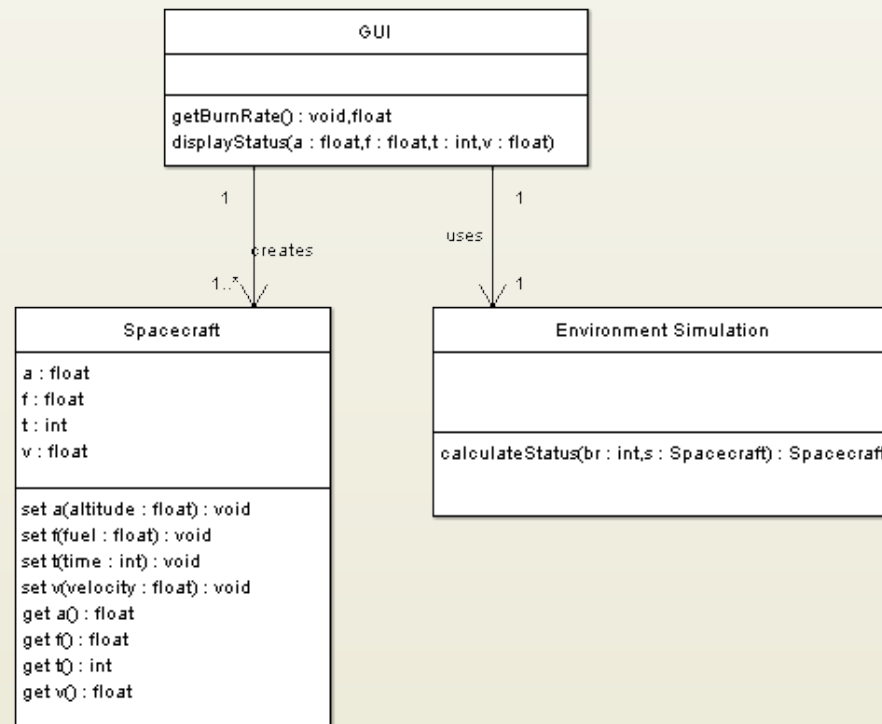
Object-Oriented Style

- 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



Layered Style

- Hierarchical system organization
 - ◆ “Multi-level client-server”
 - ◆ 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

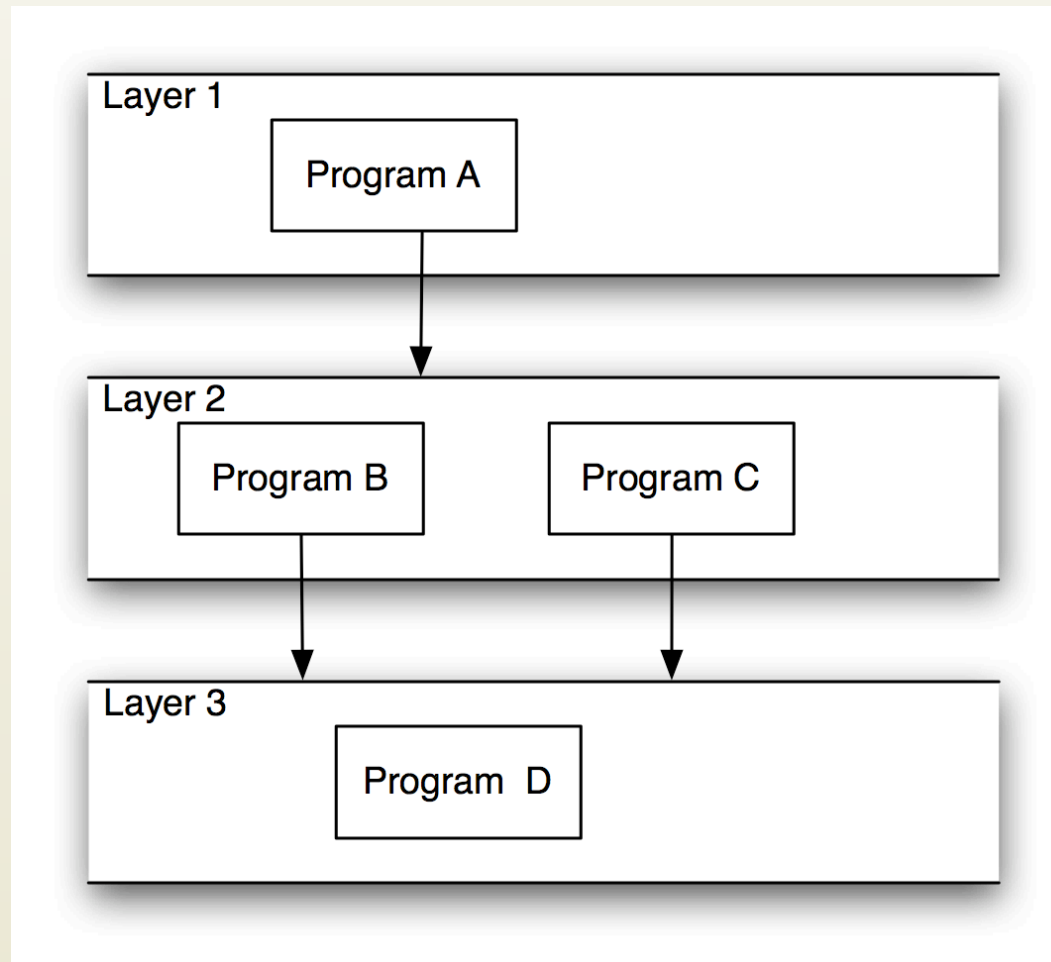
Layered Style (cont'd)

- Advantages
 - ◆ Increasing abstraction levels
 - ◆ Evolvability
 - ◆ Changes in a layer affect at most the adjacent two layers
 - Reuse
 - ◆ Different implementations of layer are allowed as long as interface is preserved
 - ◆ Standardized layer interfaces for libraries and frameworks

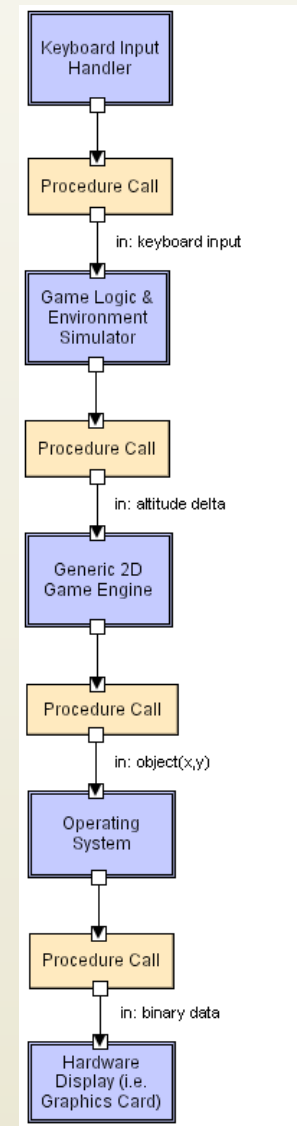
Layered Style (cont'd)

- Disadvantages
 - ◆ Not universally applicable
 - ◆ Performance
- Layers may have to be skipped
 - ◆ Determining the correct abstraction level

Layered Systems/Virtual Machines



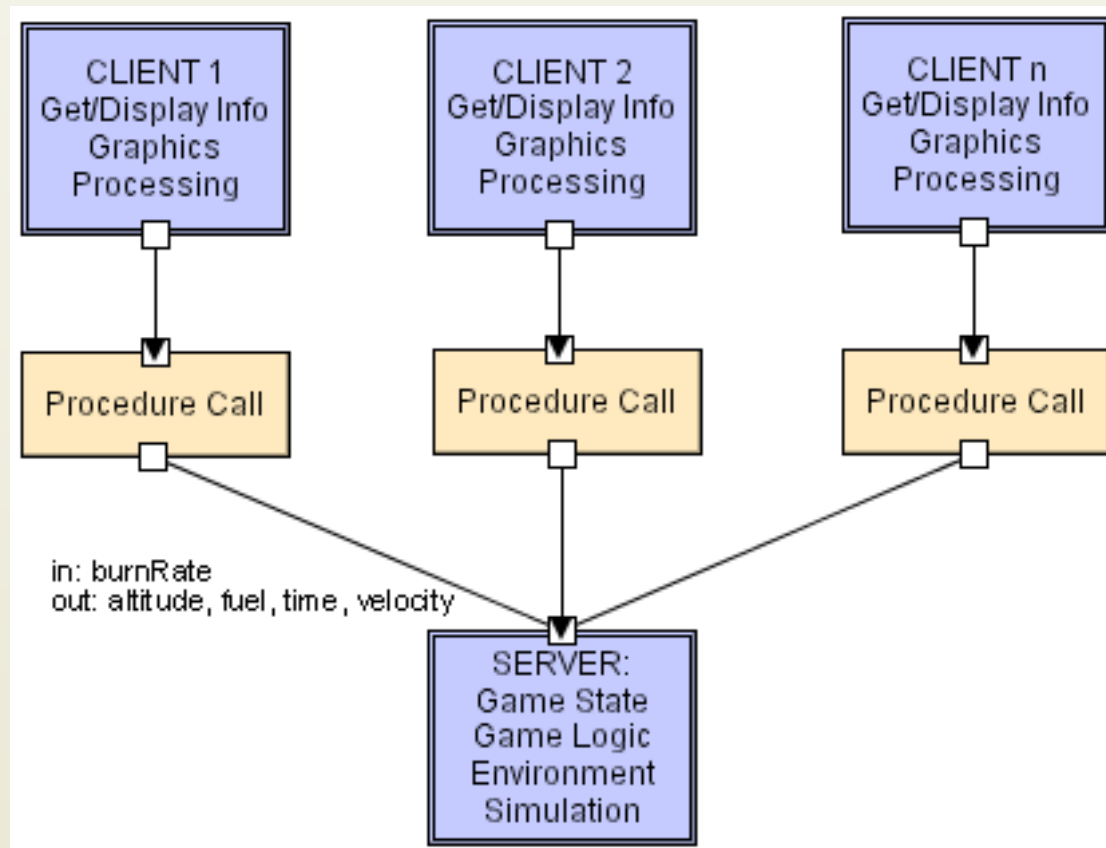
Layered LL



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 LL

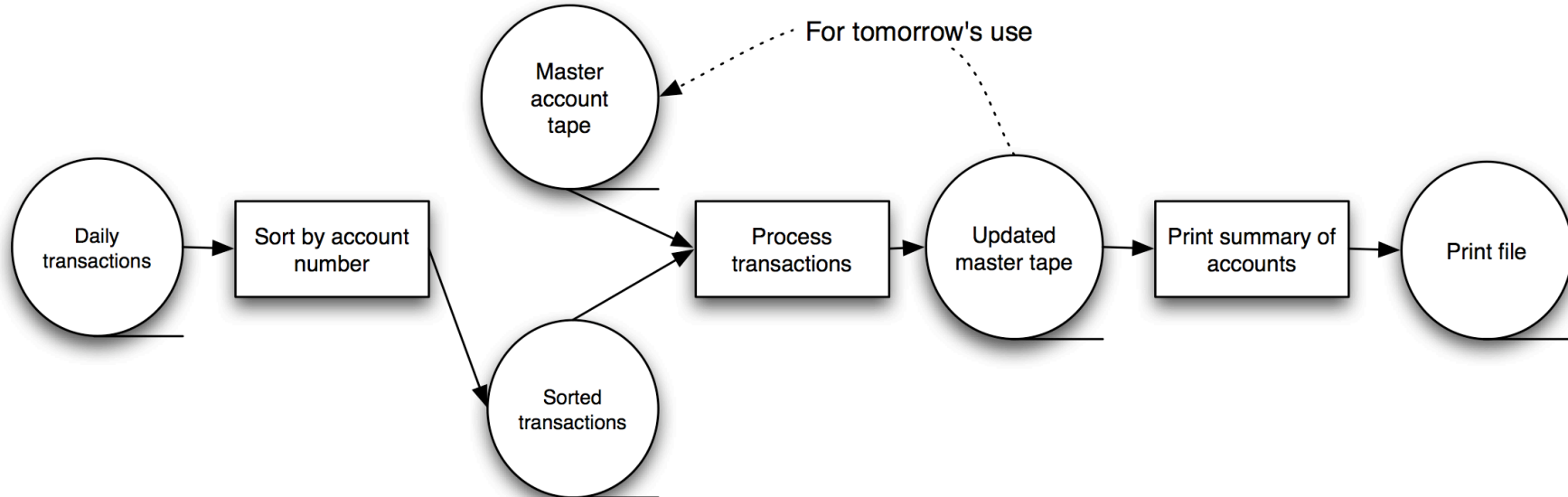


Data-Flow Styles

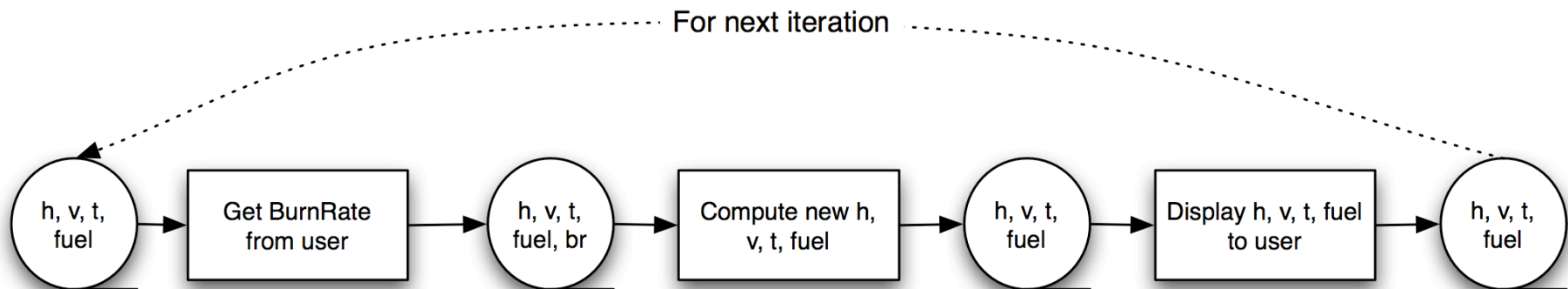
Batch Sequential

- ◆ Separate programs are executed in order; data is passed as an aggregate from one program to the next.
- ◆ Connectors: “The human hand” carrying tapes between the programs, a.k.a. “sneaker-net ”
- ◆ Data Elements: Explicit, aggregate elements passed from one component to the next upon completion of the producing program’s execution.
- Typical uses: Transaction processing in financial systems. “The Granddaddy of Styles”

Batch-Sequential: A Financial Application



Batch-Sequential LL



Not a recipe for a successful lunar mission!

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
 - ◆ Distributed systems
 - ◆ signal processing
 - ◆ parallel programming
- Example: `ls invoices | grep -e August | sort`

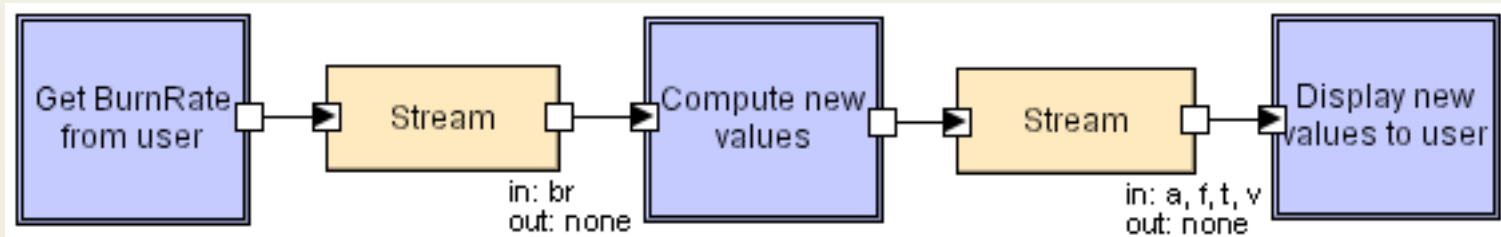
Pipe and Filter (cont'd)

- Variations
 - ◆ Pipelines — linear sequences of filters
 - ◆ Bounded pipes — limited amount of data on a pipe
 - ◆ Typed pipes — data strongly typed
- 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

Pipe and Filter (cont'd)

- Disadvantages
 - ◆ Batch organization of processing
 - ◆ Interactive applications
 - ◆ Lowest common denominator on data transmission

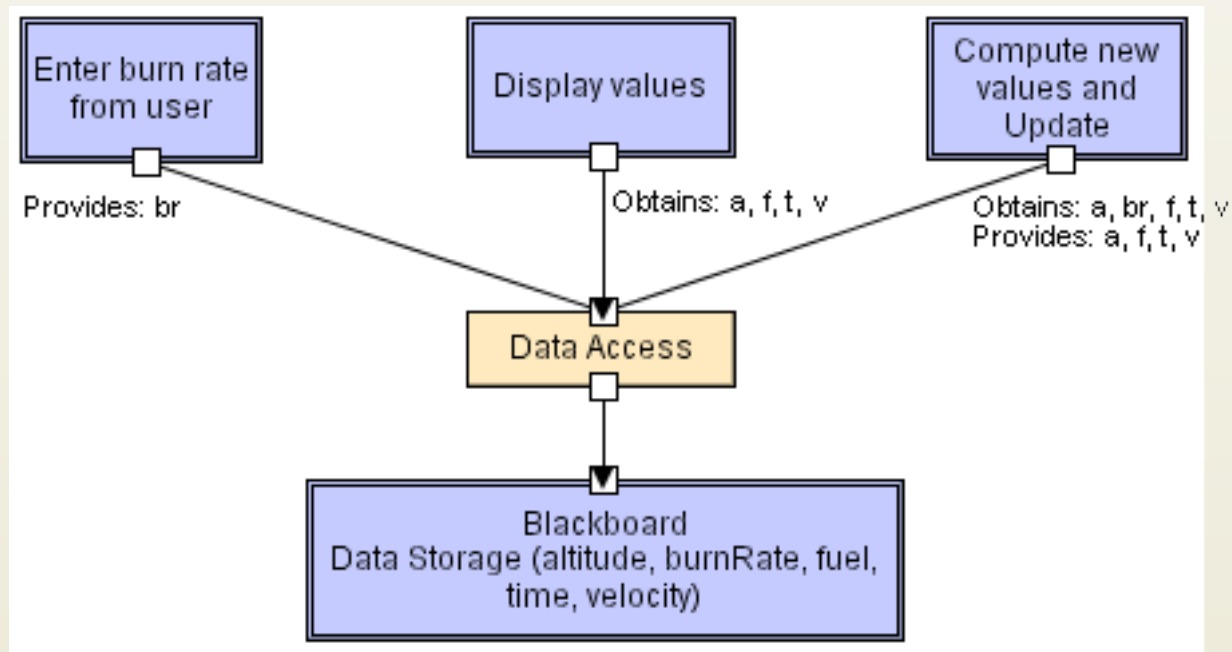
Pipe and Filter LL



Blackboard Style

- Two kinds of components
 - ◆ Central data structure — blackboard
 - ◆ Components operating on the blackboard
- System control is entirely driven by the blackboard state
- Examples
 - ◆ Typically used for AI systems
 - ◆ Integrated software environments (e.g., Interlisp)
 - ◆ Compiler architecture

Blackboard LL



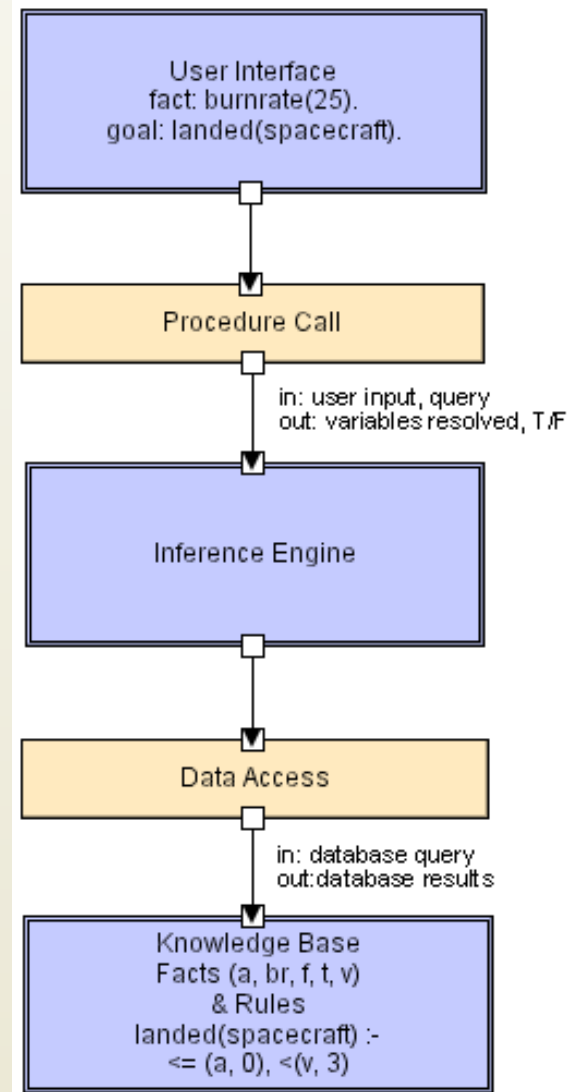
Rule-Based Style

Inference engine parses user input and determines whether it is a fact/rule or a query. If it is a fact/rule, it adds this entry to the knowledge base. Otherwise, it queries the knowledge base for applicable rules and attempts to resolve the query.

Rule-Based Style (cont'd)

- Components: User interface, inference engine, knowledge base
- Connectors: Components are tightly interconnected, with direct procedure calls and/or shared memory.
- Data Elements: Facts and queries
- Behavior of the application can be very easily modified through addition or deletion of rules from the knowledge base.
- Caution: When a large number of rules are involved understanding the interactions between multiple rules affected by the same facts can become *very* difficult.

Rule Based LL

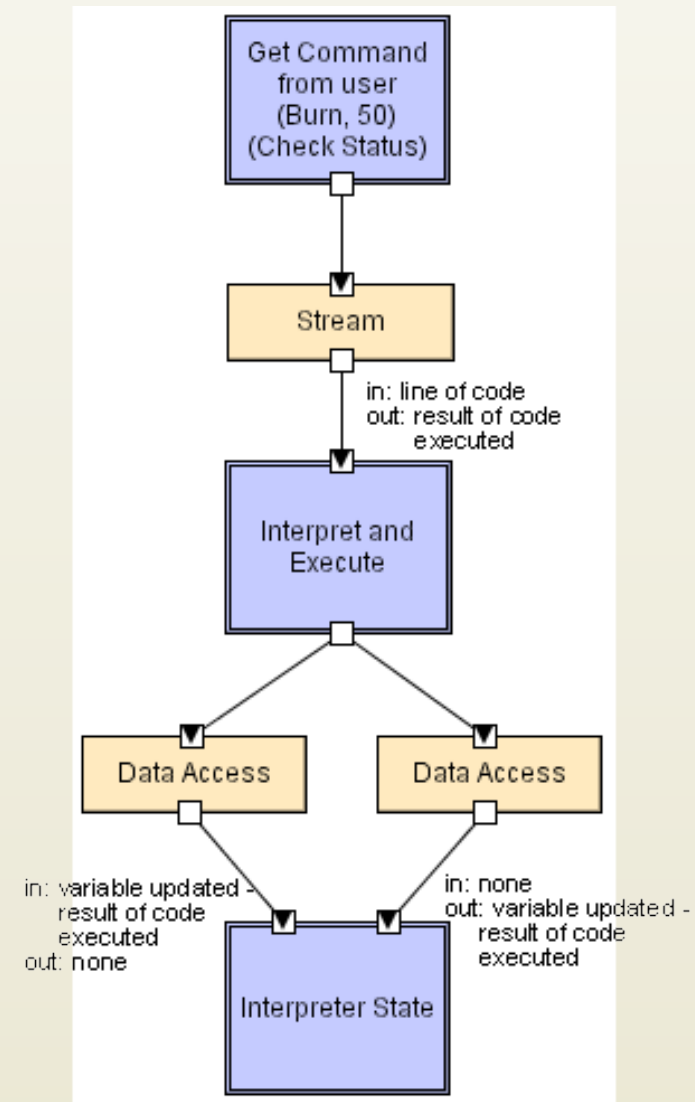


Interpreter Style

Interpreter parses and executes input commands, updating the state maintained by the interpreter

- Components: Command interpreter, program/interpreter state, user interface.
- Connectors: Typically very closely bound with direct procedure calls and shared state.
- Highly dynamic behavior possible, where the set of commands is dynamically modified. System architecture may remain constant while new capabilities are created based upon existing primitives.
- Superb for end-user programmability; supports dynamically changing set of capabilities
- Lisp and Scheme

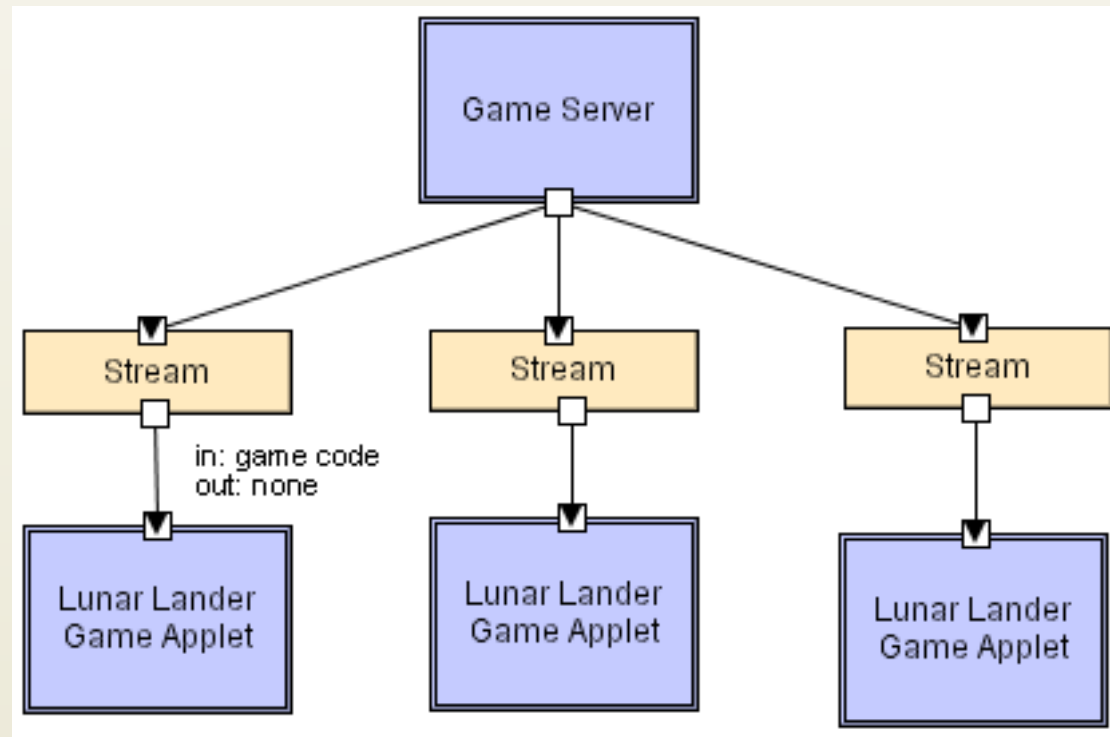
Interpreter LL



Mobile-Code Style

- Summary: a data element (some representation of a program) is dynamically transformed into a data processing component.
- Components: “Execution dock”, which handles receipt of code and state; code compiler/interpreter
- Connectors: Network protocols and elements for packaging code and data for transmission.
- Data Elements: Representations of code as data; program state; data
- Variants: Code-on-demand, remote evaluation, and mobile agent.

Mobile Code LL



Scripting languages (i.e. JavaScript, VBScript), ActiveX control, embedded Word/Excel macros.

Implicit Invocation Style

- Event announcement instead of method invocation
 - ◆ “Listeners” register interest in and associate methods with events
 - ◆ System invokes all registered methods implicitly
- Component interfaces are methods and events
- Two types of connectors
 - ◆ Invocation is either explicit or implicit in response to events
- Style invariants
 - ◆ “Announcers” are unaware of their events’ effects
 - ◆ No assumption about processing in response to events

Implicit Invocation (cont'd)

- Advantages
 - ◆ Component reuse
 - ◆ System evolution
 - Both at system construction-time & run-time
- Disadvantages
 - ◆ Counter-intuitive system structure
 - ◆ Components relinquish computation control to the system
 - ◆ No knowledge of what components will respond to event
 - ◆ No knowledge of order of responses

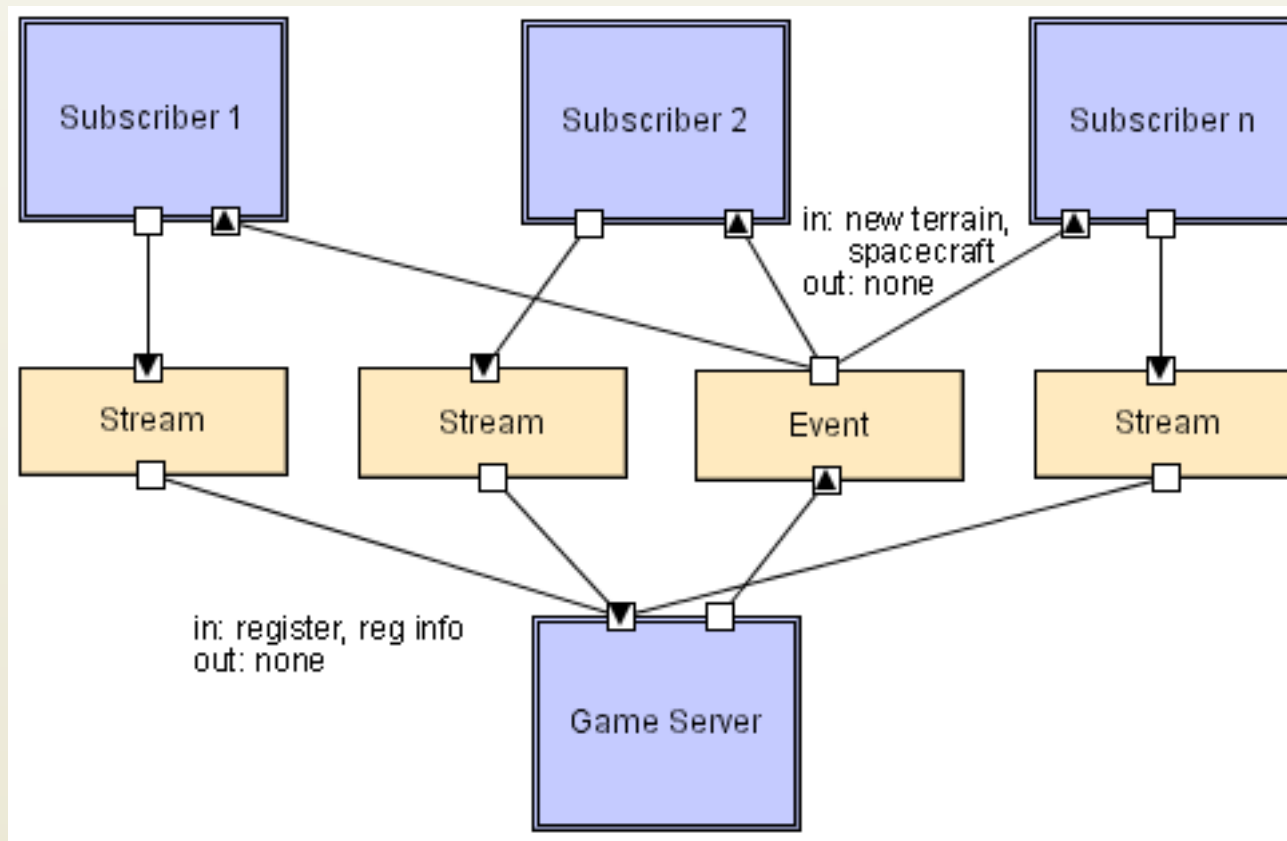
Publish-Subscribe

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

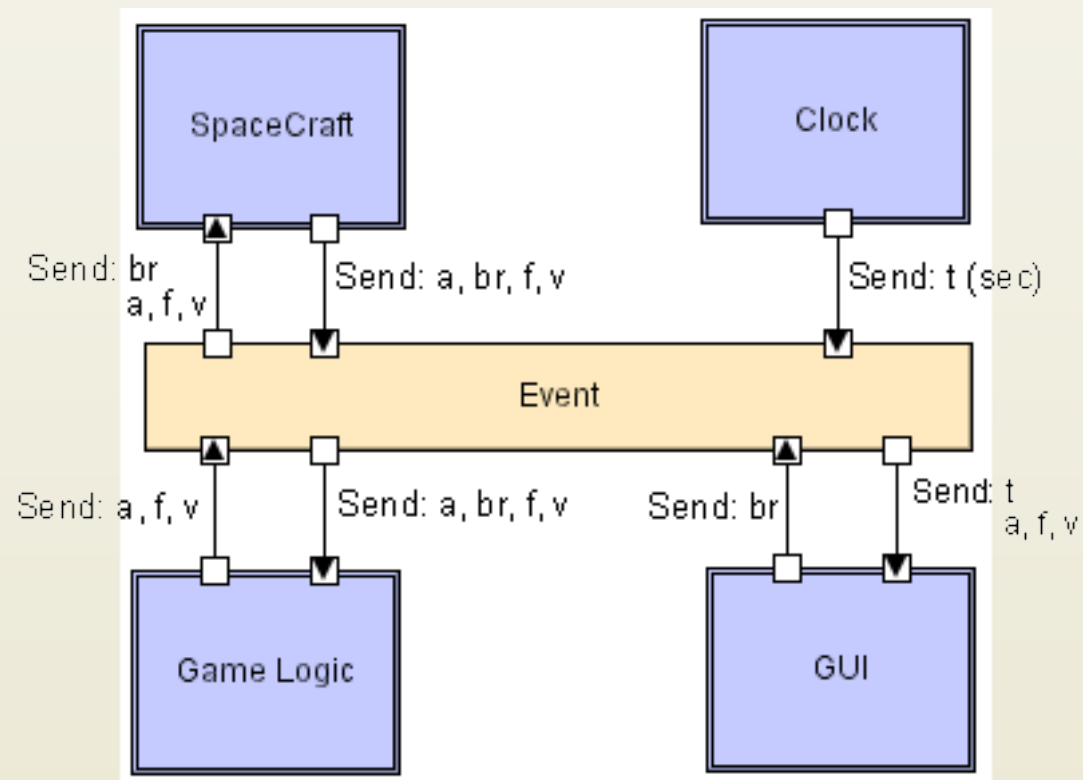
Pub-Sub LL



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



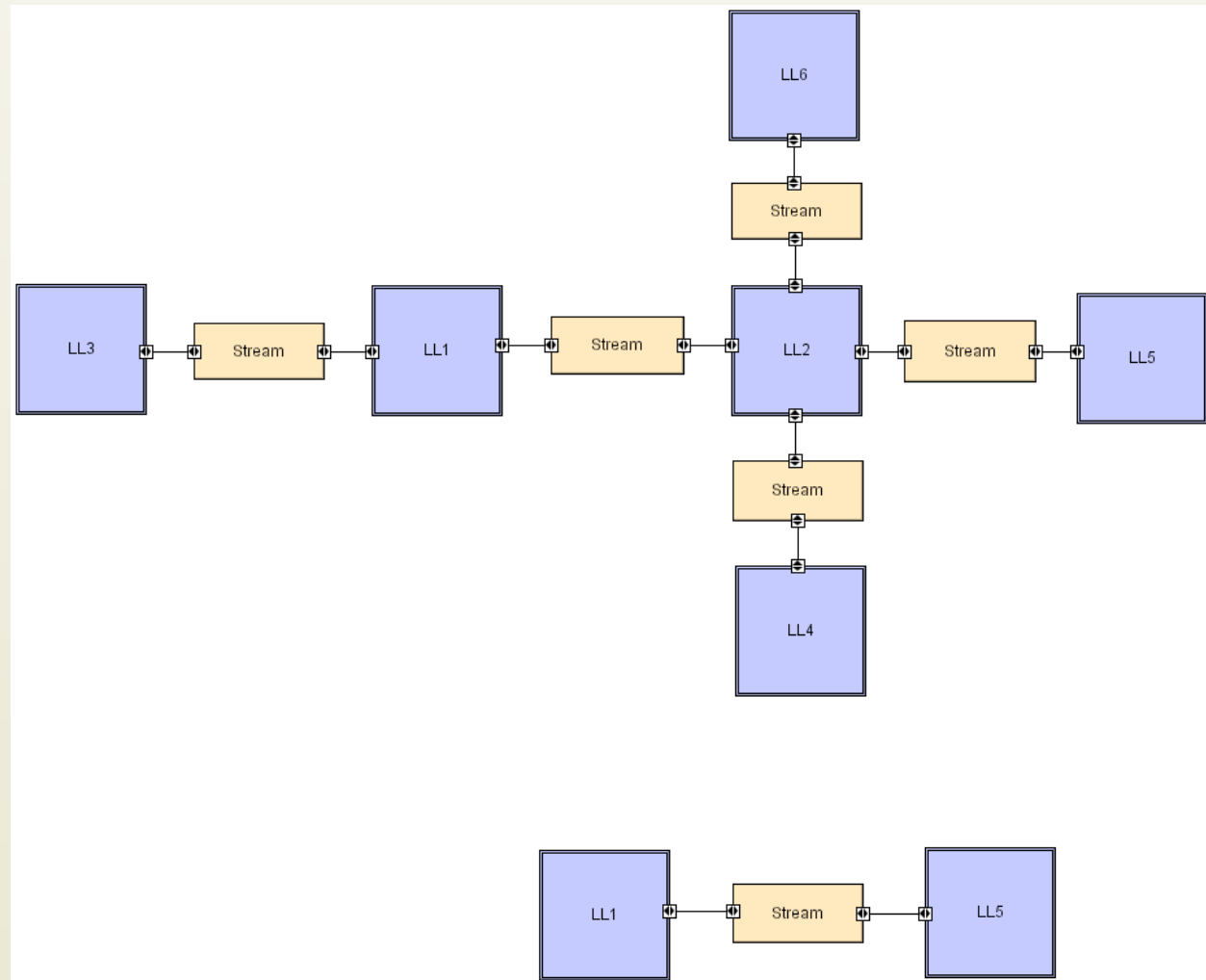
Peer-to-Peer Style

- 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 LL



Learning Objectives

- Delineate the role of Styles and Patterns in Software architecture
- Understand the role and benefits of architectural styles
- Understand and apply common styles in your designs
- Construct complex styles from simpler styles

Heterogeneous Styles

- More complex styles created through composition of simpler styles
- REST (from the first lecture)
 - ◆ Complex history presented later in course
- C2
 - ◆ Implicit invocation + Layering + other constraints
- Distributed objects
 - ◆ OO + client-server network style

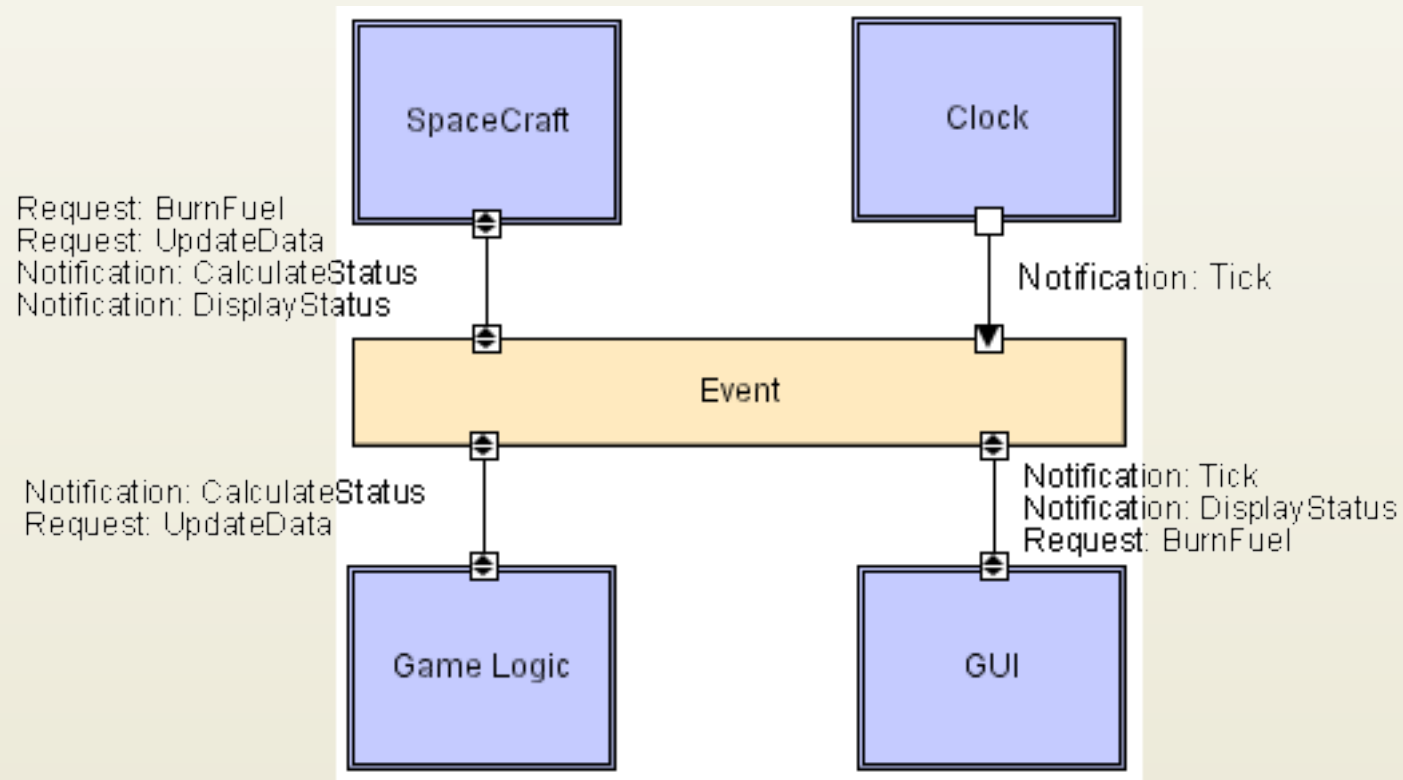
C2 Style

An indirect invocation style in which independent components communicate exclusively through message routing connectors. Strict rules on connections between components and connectors induce layering.

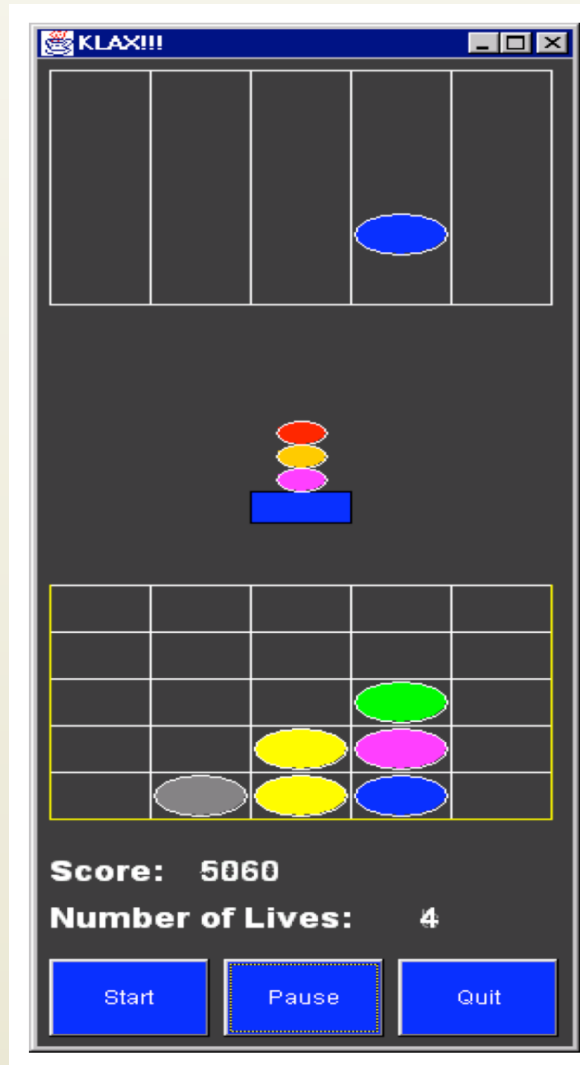
C2 Style (cont'd)

- Components: Independent, potentially concurrent message generators and/or consumers
- Connectors: Message routers that may filter, translate, and broadcast messages of two kinds: notifications and requests.
- Data Elements: Messages – data sent as first-class entities over the connectors. Notification messages announce changes of state. Request messages request performance of an action.
- Topology: Layers of components and connectors, with a defined “top” and “bottom”, wherein notifications flow downwards and requests upwards.

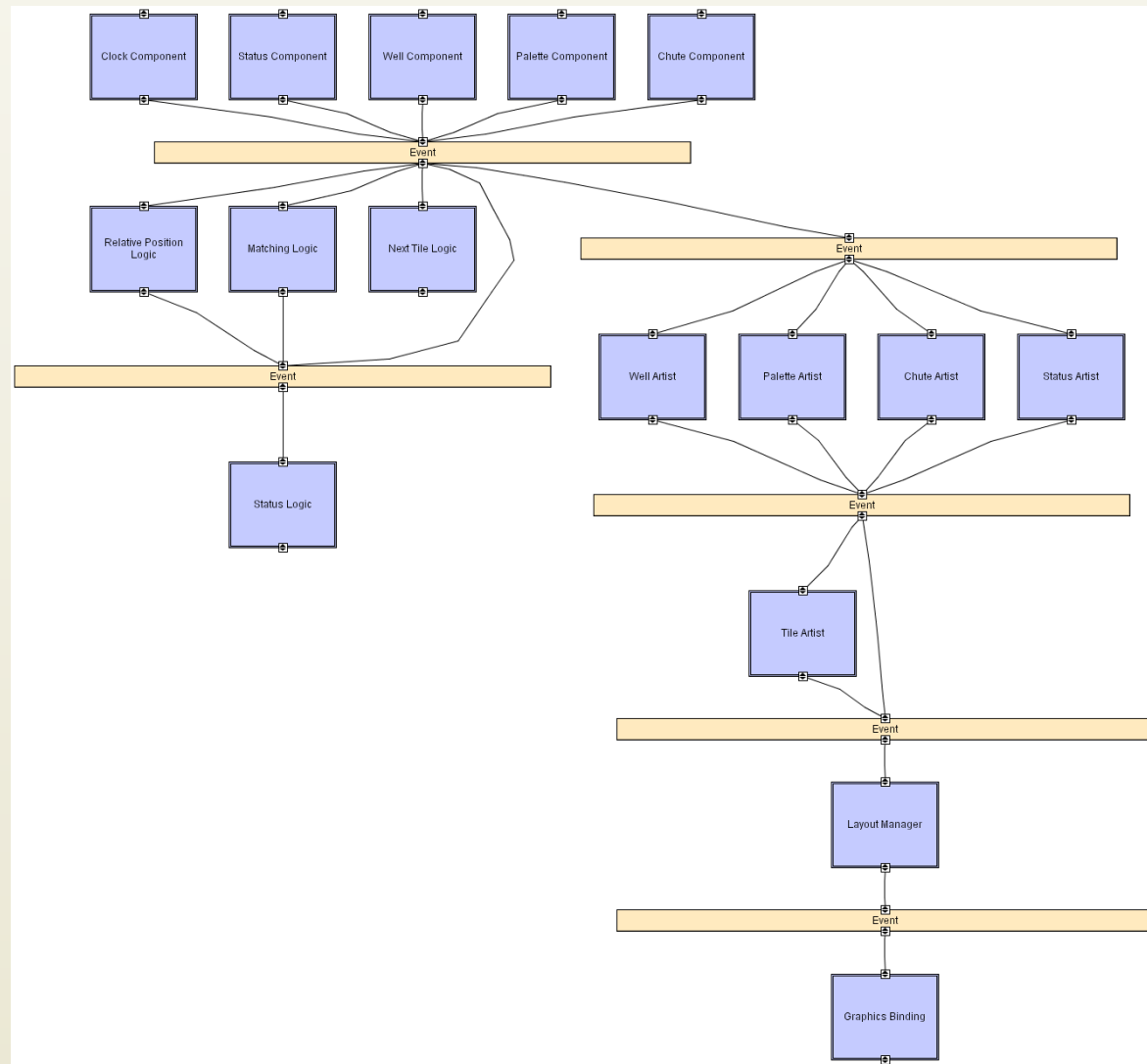
C2 LL



KLAX



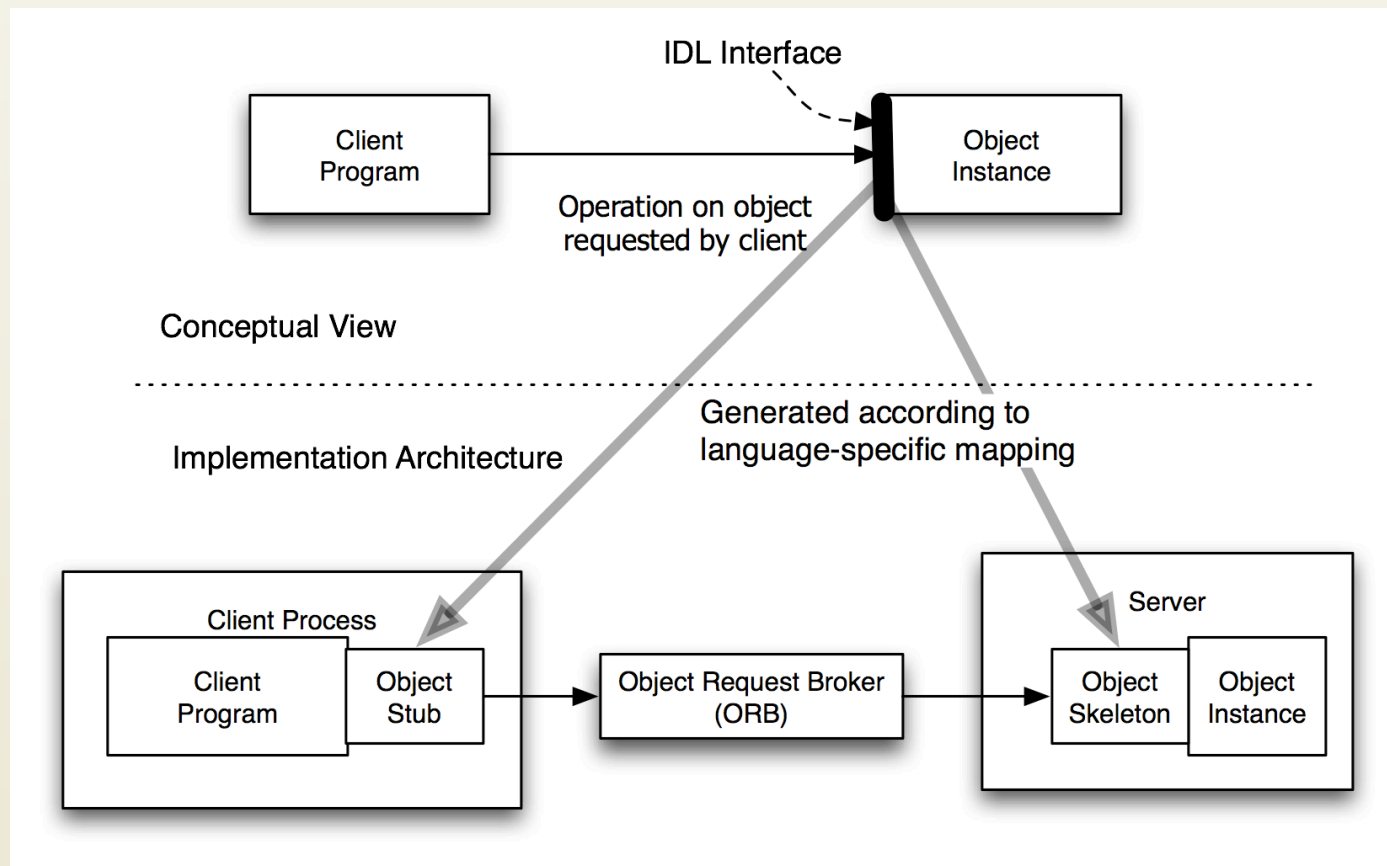
KLAX in C2



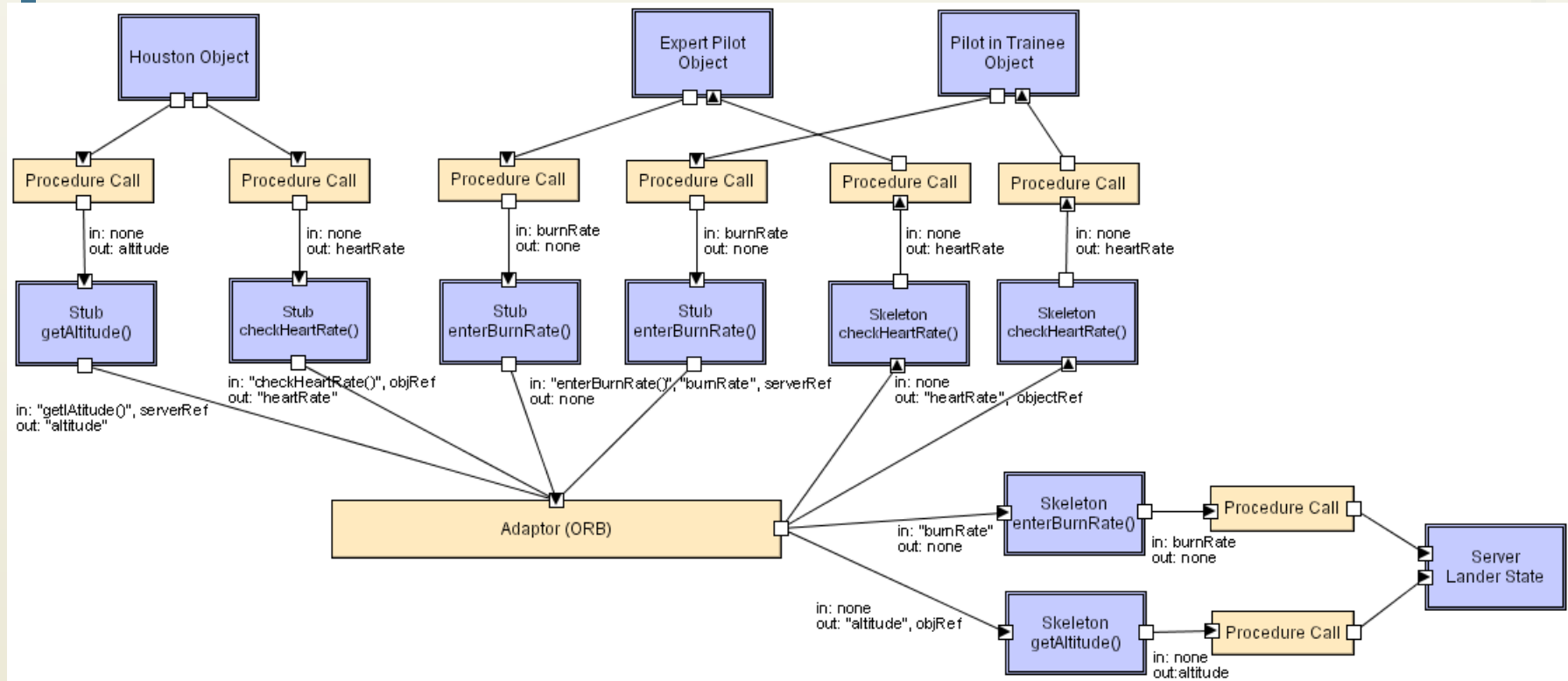
Distributed Objects: CORBA

- “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.

CORBA Concept and Implementation



CORBA LL



Learning Objectives

- Delineate the role of Styles and Patterns in Software architecture
- Understand the role and benefits of architectural styles
- Understand and apply common styles in your designs
- Construct complex styles from simpler styles

Observations

- Different styles result in
 - ◆ Different architectures
 - ◆ Architectures with greatly differing properties
- A style does not fully determine resulting architecture
 - ◆ A single style can result in different architectures
 - ◆ Considerable room for
 - Individual judgment
 - Variations among architects
- A style defines domain of discourse
 - ◆ About problem (domain)
 - ◆ About resulting system

Style Summary (1/4)

Style Category & Name	Summary	Use It When	Avoid It When
Language-influenced styles			
Main Program and Subroutines	Main program controls program execution, calling multiple subroutines.	Application is small and simple.	Complex data structures needed. Future modifications likely.
Object-oriented	Objects encapsulate state and accessing functions	Close mapping between external entities and internal objects is sensible. Many complex and interrelated data structures.	Application is distributed in a heterogeneous network. Strong independence between components necessary. High performance required.
Layered			
Virtual Machines	Virtual machine, or a layer, offers services to layers above it	Many applications can be based upon a single, common layer of services. Interface service specification resilient when implementation of a layer must change.	Many levels are required (causes inefficiency). Data structures must be accessed from multiple layers.
Client-server	Clients request service from a server	Centralization of computation and data at a single location (the server) promotes manageability and scalability; end-user processing limited to data entry and presentation.	Centrality presents a single-point-of-failure risk; Network bandwidth limited; Client machine capabilities rival or exceed the server's.

Style Summary, continued (2/4)

Data-flow styles

Batch sequential	Separate programs executed sequentially, with batched input	Problem easily formulated as a set of sequential, severable steps.	Interactivity or concurrency between components necessary or desirable. Random-access to data required.
Pipe-and-filter	Separate programs, a.k.a. filters, executed, potentially concurrently. Pipes route data streams between filters	[As with batch-sequential] Filters are useful in more than one application. Data structures easily serializable.	Interaction between components required. Exchange of complex data structures between components required.

Shared memory

Blackboard	Independent programs, access and communicate exclusively through a global repository known as blackboard	All calculation centers on a common, changing data structure; Order of processing dynamically determined and data-driven.	Programs deal with independent parts of the common data. Interface to common data susceptible to change. When interactions between the independent programs require complex regulation.
Rule-based	Use facts or rules entered into the knowledge base to resolve a query	Problem data and queries expressible as simple rules over which inference may be performed.	Number of rules is large. Interaction between rules present. High-performance required.

Style Summary, continued (3/4)

Interpreter

Interpreter

Interpreter parses and executes the input stream, updating the state maintained by the interpreter

Highly dynamic behavior required. High degree of end-user customizability.

High performance required.

Mobile Code

Code is mobile, that is, it is executed in a remote host

When it is more efficient to move processing to a data set than the data set to processing.
When it is desirable to dynamically customize a local processing node through inclusion of external code

Security of mobile code cannot be assured, or sandboxed.
When tight control of versions of deployed software is required.

Style Summary, continued (4/4)

Implicit Invocation

Publish-subscribe	Publishers broadcast messages to subscribers	Components are very loosely coupled. Subscription data is small and efficiently transported.	When middleware to support high-volume data is unavailable.
Event-based	Independent components asynchronously emit and receive events communicated over event buses	Components are concurrent and independent. Components heterogeneous and network-distributed.	Guarantees on real-time processing of events is required.

Peer-to-peer

Peers hold state and behavior and can act as both clients and servers	Peers are distributed in a network, can be heterogeneous, and mutually independent. Robust in face of independent failures. Highly scalable.	Trustworthiness of independent peers cannot be assured or managed. Resource discovery inefficient without designated nodes.
---	--	---

More complex styles

C2	Layered network of concurrent components communicating by events	When independence from substrate technologies required. Heterogeneous applications. When support for product-lines desired.	When high-performance across many layers required. When multiple threads are inefficient.
Distributed Objects	Objects instantiated on different hosts	Objective is to preserve illusion of location-transparency	When high overhead of supporting middleware is excessive. When network properties are unmaskable, in practical terms.