Software Architecture and Design Thinking 116U01C701

Module 1

Module 1: Software Architecture& Design Thinking (10)

- Basic Concepts
- Designing Architectures
- Conventional architectural styles
- Concepts of Software Architecture, Models, Processes, Stakeholders
- Styles and Architectural patterns, Pipes and Filters, Event based
- Implicit invocation, Layered systems, Repositories, Interpreters

Basic Concepts

A software architecture is defined as: “A software system architecture is the set of principal design decisions made about the system.”

SA is the blue print for system’s construction.

As the size and complexity of software systems increase, the design and specifications of overall system structure become more significant issues than the choice of algorithms and data structures of computation.

Basic Concepts

Design decision encompass every aspect of system under development related to:

- System structure: how elements should be organized and composed
- Functional behavior: how data processing, storage and visualization will be sequenced
- Interaction: communication between all elements using event
- Non-functional properties: system dependability will be ensured
- Implementation: how the elements will be implemented

Basic Concepts

Software Architecture (SA) Level of Design

Structural issues:

- Organization of a system as a composition of components
- Global control structures
- The protocols of communication
- Synchronization and data access
- The assignments of functionality to design elements
- The composition of design elements
- Physical distribution
- Scaling and performance
- Dimensions of evolution
- Selection of design Alternatives

Basic Concepts

SA involves:

- the description of elements from which system are built
- interactions among those elements
- patterns that guide their compositions
- constraints on these patterns

A system will be defined in terms of collection of components and interactions among them, which may become part of another bigger system

Basic Concepts

SA will be :

- Abstractly represented as box- and –line diagrams accompanying details explaining the meaning of each symbol specifying choice of each component

Basic Concepts

SA involves:

- the description of elements from which system are built
- interactions among those elements
- patterns that guide their compositions
- constraints on these patterns

A system will be defined in terms of collection of components and interactions among them, which may become part of another bigger system

Temporal Aspect

- Design decisions are and made over a system’s lifetime
  → Architecture has a temporal aspect
- At any given point in time the system has only one architecture
- A system’s architecture will change over time

Basic Concepts

Prescriptive architecture v/s Descriptive architecture:

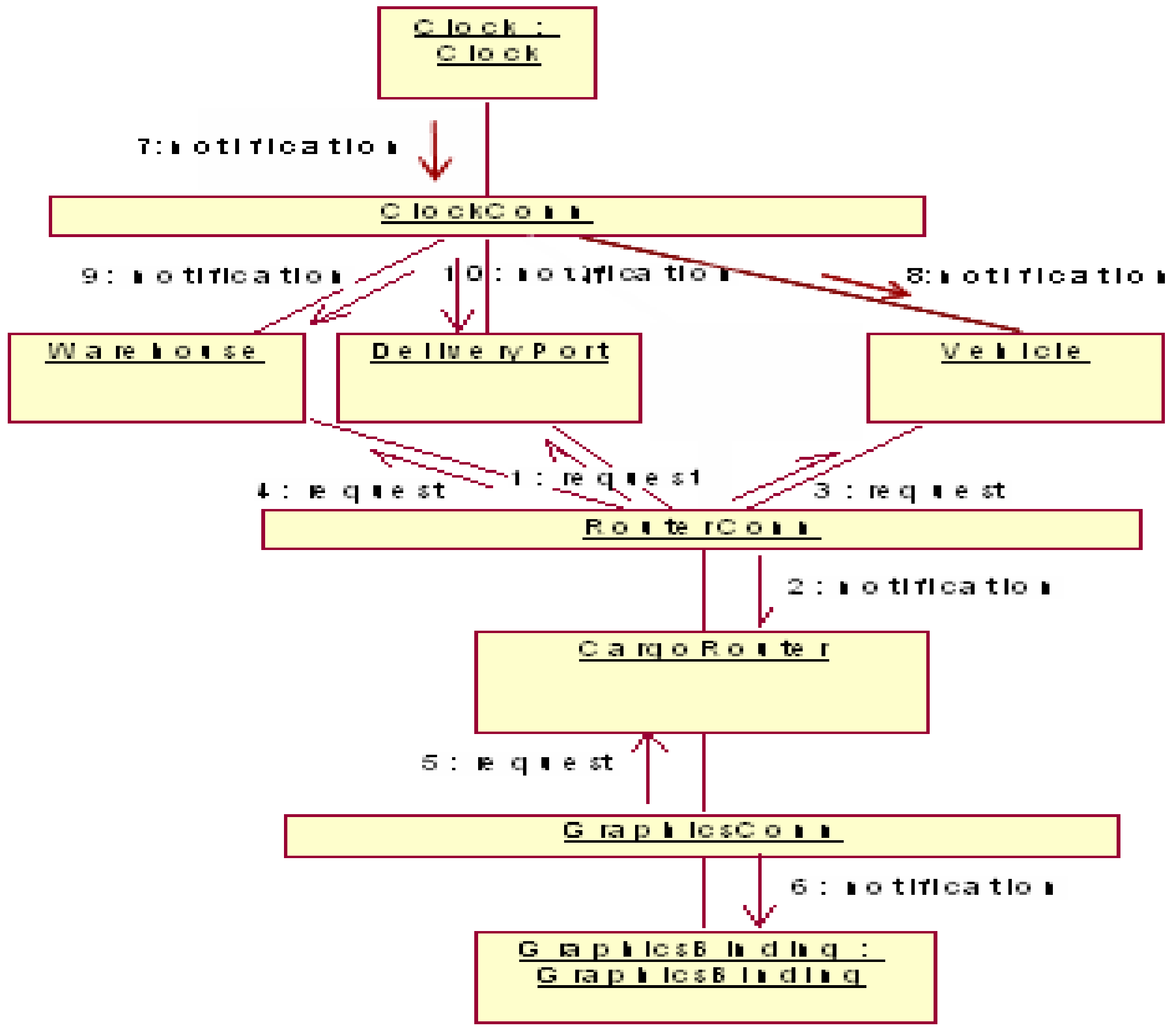
- As intended – as conceived architecture
- Perspective architecture **need not** necessarily exist in any tangible form **P**
- **P will be refined and realized with a set of artifacts A**
- Descriptive architecture is representing it in any form (UML etc.)

Basic Concepts

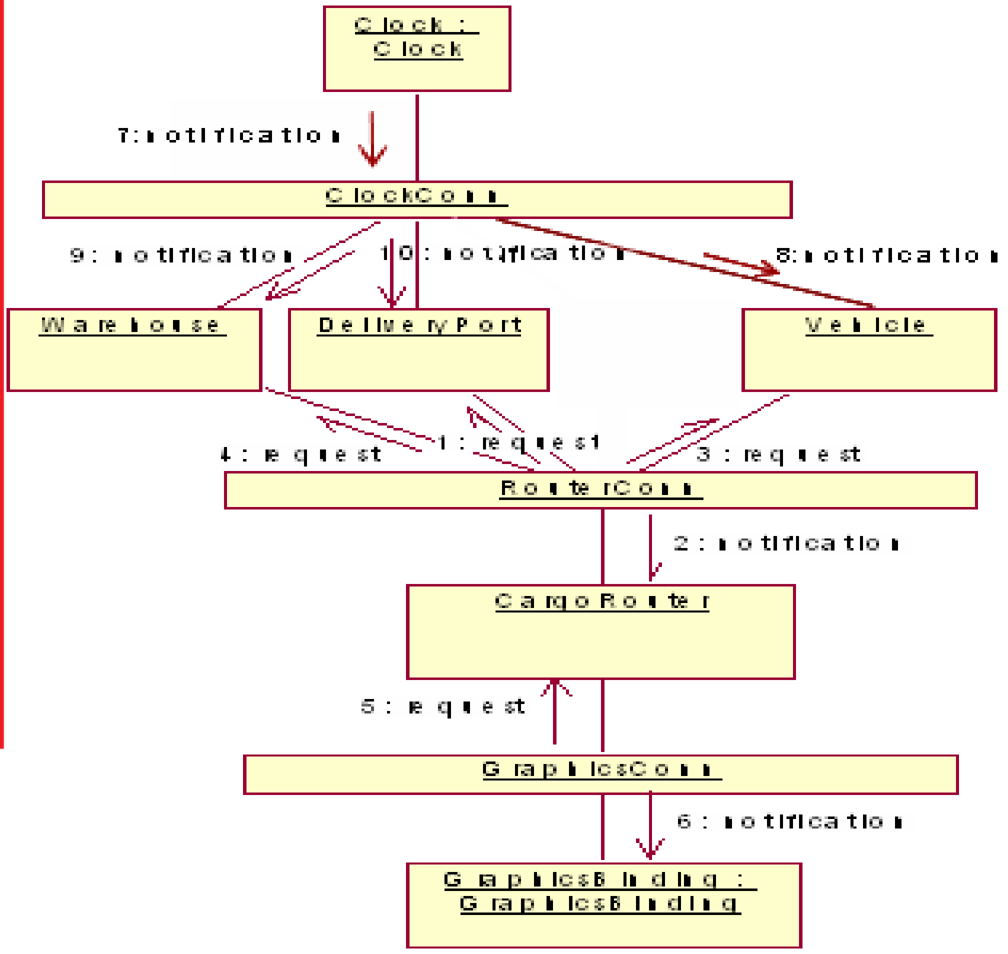
Prescripective architecture v/s Descriptive architecture:

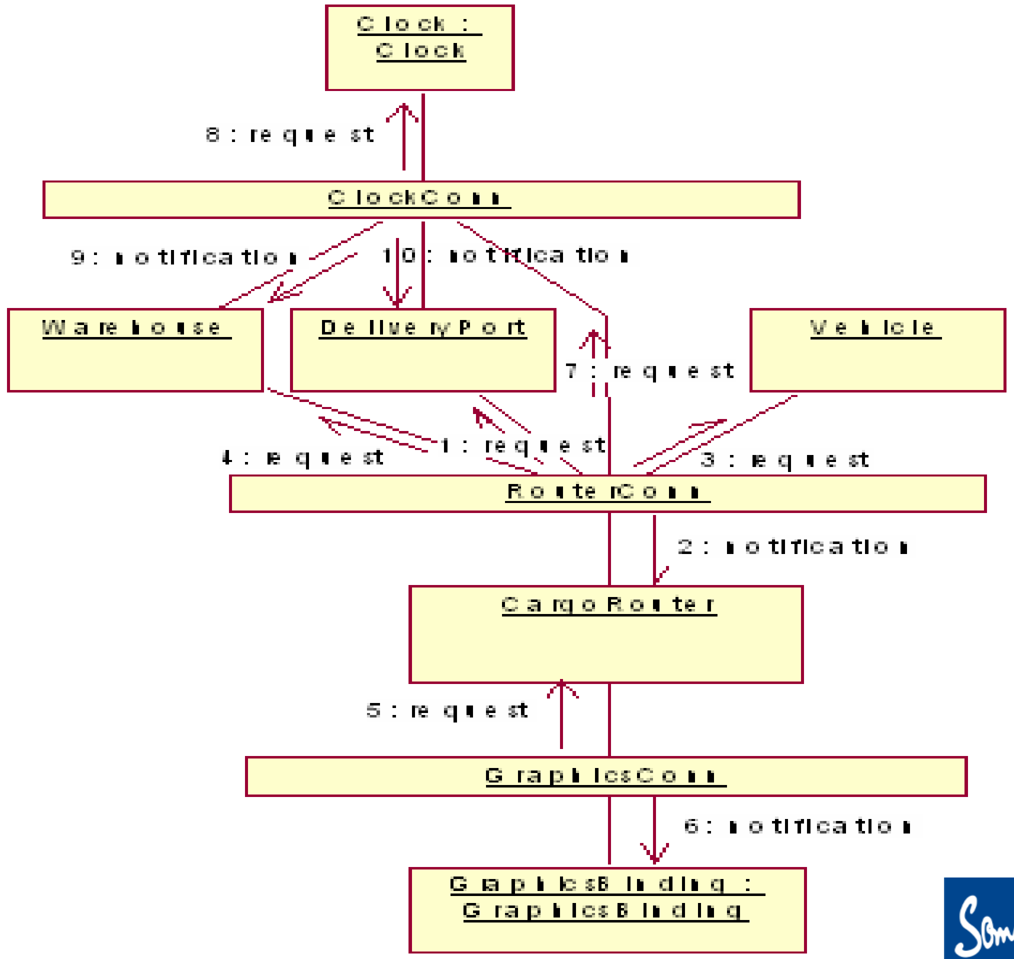
- Greenfield development: inception of intial set P1, A1 and D1 is empty
- Brownfield development: D1 is non empty, P1 may be empty

As-Designed vs. As-Implemented Architecture



As-Designed vs. As-Implemented Architecture





As-Designed vs. As-Implemented Architecture

- Which architecture is “correct”?
- Are the two architectures consistent with one another?
- What criteria are used to establish the consistency between the two architectures?
- On what information is the answer to the preceding questions based?

Architectural Evolution

- When a system evolves, ideally its prescriptive architecture is modified first
- In practice, the system – and thus its descriptive architecture – is often directly modified
- This happens because of
  - Developer sloppiness
  - Perception of short deadlines which prevent thinking in detail and documenting
  - Lack of documented prescriptive architecture
  - Need or desire for code optimizations
  - Inadequate techniques or tool support

Basic Concepts

Architectural Degradation:

- **P** and **D** would be identical at the start but over the period there could be changes while realizing the architecture
- When a system is initially developed or when already implemented is evolved , P is modified

Architectural Degradation

- Two related concepts
  - Architectural drift
  - Architectural erosion
- *Architectural drift* is introduction of principal design decisions into a system's descriptive architecture that
  - are not included in, encompassed by, or implied by the prescriptive architecture
  - but which do not violate any of the prescriptive architecture's design decisions
- *Architectural erosion* is the introduction of architectural design decisions into a system's descriptive architecture that violate its prescriptive architecture

Architectural Recovery

- If architectural degradation is allowed to occur, one will be forced to *recover* the system's architecture sooner or later
- *Architectural recovery* is the process of determining a software system's architecture from its implementation-level artifacts
- Implementation-level artifacts can be
  - Source code
  - Executable files
  - Java .class files

Deployment

- A software system cannot fulfill its purpose until it is *deployed*
    - Executable modules are physically placed on the hardware devices on which they are supposed to run
- The deployment view of an architecture can be critical in assessing whether the system will be able to satisfy its requirements
- Possible assessment dimensions
    - Available memory
    - Power consumption
    - Required network bandwidth

Software Architecture’s Elements

- A software system’s architecture typically is not (and should not be) a uniform monolith
- A software system’s architecture should be a composition and interplay of different elements
  - Processing
  - Data, also referred as information or state
  - Interaction

Components

- Elements that encapsulate processing and data in a system’s architecture are referred to as *software components*
- Address key system concerns such as:
    - Processing / functionality/ behavior
    - State: information or data
    - Interaction: interconnection, communication, coordination
- Components typically provide application-specific services

Components

Definition

- A *software component* is an architectural entity that
  - encapsulates a subset of the system’s functionality and/or data
  - restricts access to that subset via an explicitly defined interface
  - has explicitly defined dependencies on its required execution context

- Component is a locus computation and state in a system
- Can be as a simple as an operation or as a complex as an entire system depending on perspective
- Can be “seen” by the users (software or human) from outside through interface or its like a “black box”
- Components implements software engineering principles such as encapsulation, abstraction and modularity

Components

- Usable and reusable across applications
- Extent of the context by components include:
    - Component’s required interface to services provided by other components in a system on which component depends for its ability to perform its operation
    - Availability of resources, such as data files or directories on which the components relies
    - Required system software , such as programming language run time environments, middleware platforms, operating systems, network protocols and device drivers
    - Hardware configuration needed to execute the components

Reusability- math libraries, GUI toolkit, word processor

Connectors

- In complex systems *interaction* may become more important and challenging than the functionality of the individual components
- **Definition**
    - A *software connector* is an architectural building block tasked with effecting and regulating interactions among components
- In many software systems connectors are usually simple procedure calls or shared data accesses
    - Much more sophisticated and complex connectors are possible!
- **Connectors typically provide application-independent interaction facilities**

Examples of Connectors

- Shared data access:
  - nonlocal variables or shared memory
  - Allow multiple software components to interact by reading from and writing to the shared facilities
  - Interaction is distributed in time
- Distribution connectors:
  - Encapsulate network library APIs
  - Usually coupled with more basic connectors
  - E.g. remote procedure call (RPC)
- Adaptor connectors –stdio.h

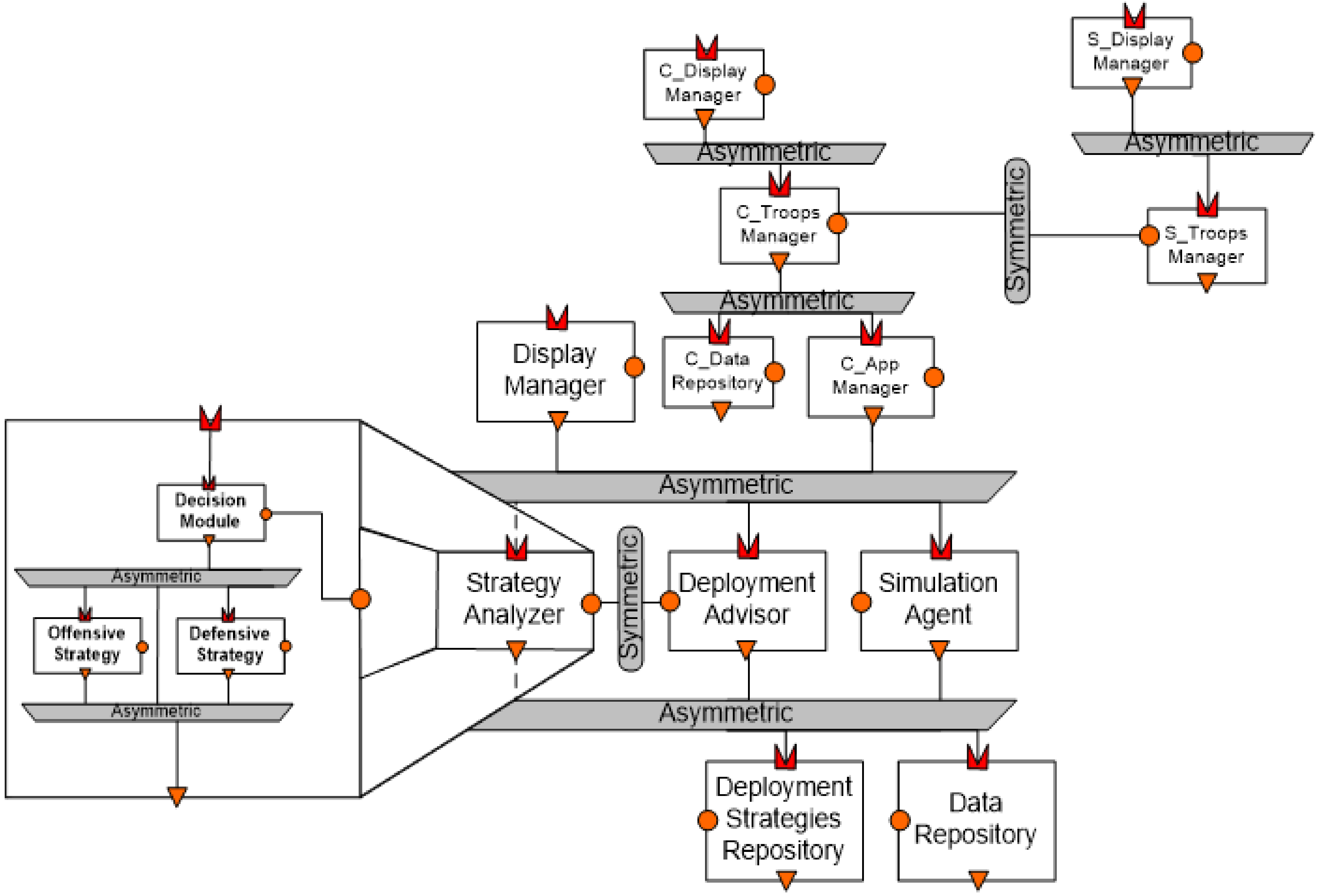
Examples of Connectors

- Procedure call connectors - mod(a,b)
- Message passing connectors
- Streaming connectors – printf()
- Distribution connectors
- Wrapper/adaptor connectors –stdio.h

Configurations

- Components and connectors are composed in a specific way in a given system’s architecture to accomplish that system’s objective
- **Definition**
  - An *architectural configuration*, or topology, is a set of specific associations between the components and connectors of a software system’s architecture
  - Called topology also

An Example Configuration



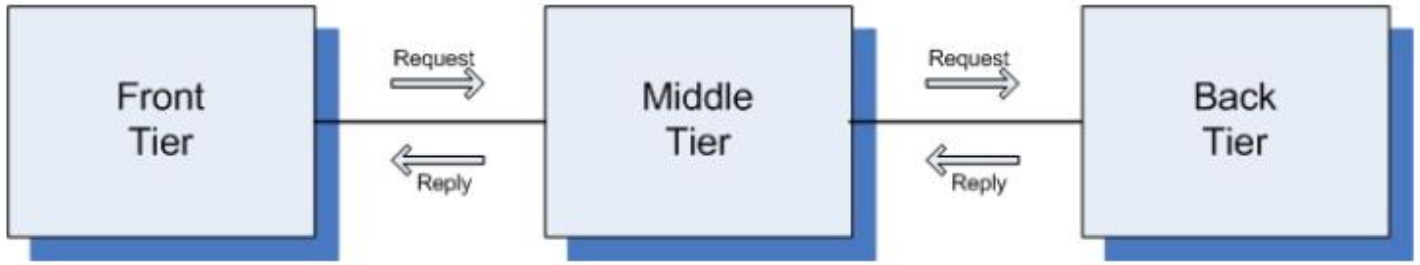
Architectural Styles

- Certain design choices regularly result in solutions with superior properties
  - Compared to other possible alternatives, solutions such as this are more elegant, effective, efficient, dependable, evolvable, scalable, and so on
- **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

Architectural Patterns

- **Definition**
  - 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
- A widely used pattern in modern distributed systems is the *three-tiered system* pattern
  - Science
  - Banking
  - E-commerce
  - Reservation systems

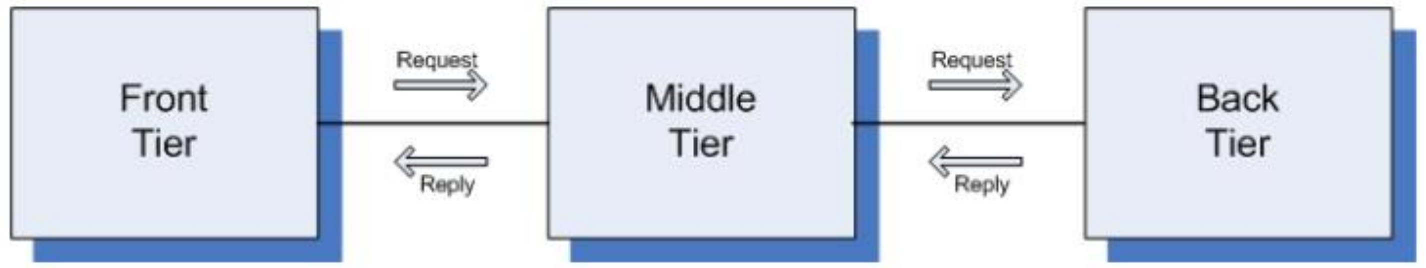
Three-Tiered Pattern



- Front/client Tier
  - Contains the user interface functionality to access the system's services
  - GUI
- Middle / application / business logic Tier
  - Contains the application's major functionality
  - Incharge of significant process
  - Service provider to front tier
  - Deployed at Server host
- Back / back end / data Tier
  - Contains the application's data access and storage functionality

  - Interaction is based on request-reply paradigm
  - Strictly adhere to synchronous, request triggered

Three-Tiered Pattern



Order processing GUI

Order Processing server

Order Database

Architectural Models, Views, and Visualizations

- Architecture Model
  - An artifact documenting some or all of the architectural design decisions about a system
- Architecture Visualization
  - A way of depicting some or all of the architectural design decisions about a system to a stakeholder
- Architecture View
  - A subset of related architectural design decisions

Architectural Processes

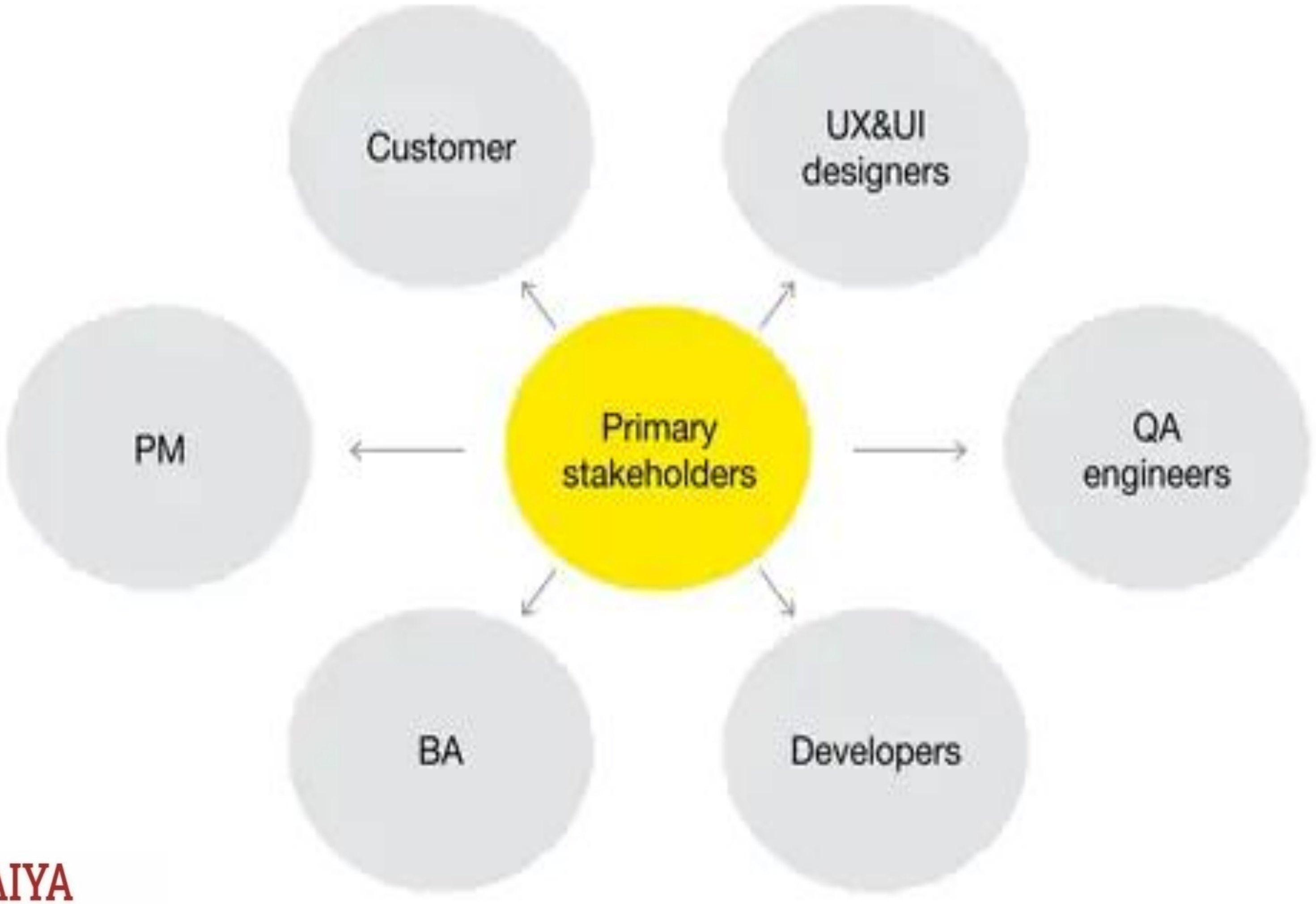
- Architectural design
- Architecture modeling and visualization
- Architecture-driven system analysis
- Architecture-driven system implementation
- Architecture-driven system deployment, runtime redeployment, and mobility
- Architecture-based design for non-functional properties, including security and trust
- architectural adaptation

Stakeholders in a System’s Architecture

- Stakeholders are individuals, groups, or organizations that are actively involved in a software project, can influence it due to their position, and whose interests may be affected by the success or failure of the project.

- Architects
- Developers
- Testers
- Managers
- Customers
- Users
- Vendors

Stakeholders in a Svstem’s Architecture



Engineering Design Process/ Designing Architectures

- Feasibility stage: identifying a set of feasible concepts for the design as a whole
- Preliminary design stage: selection and development of the best concept.
- Detailed design stage: development of engineering descriptions of the concept.
- Planning stage: evaluating and altering the concept to suit the requirements of production, distribution, consumption and product retirement.

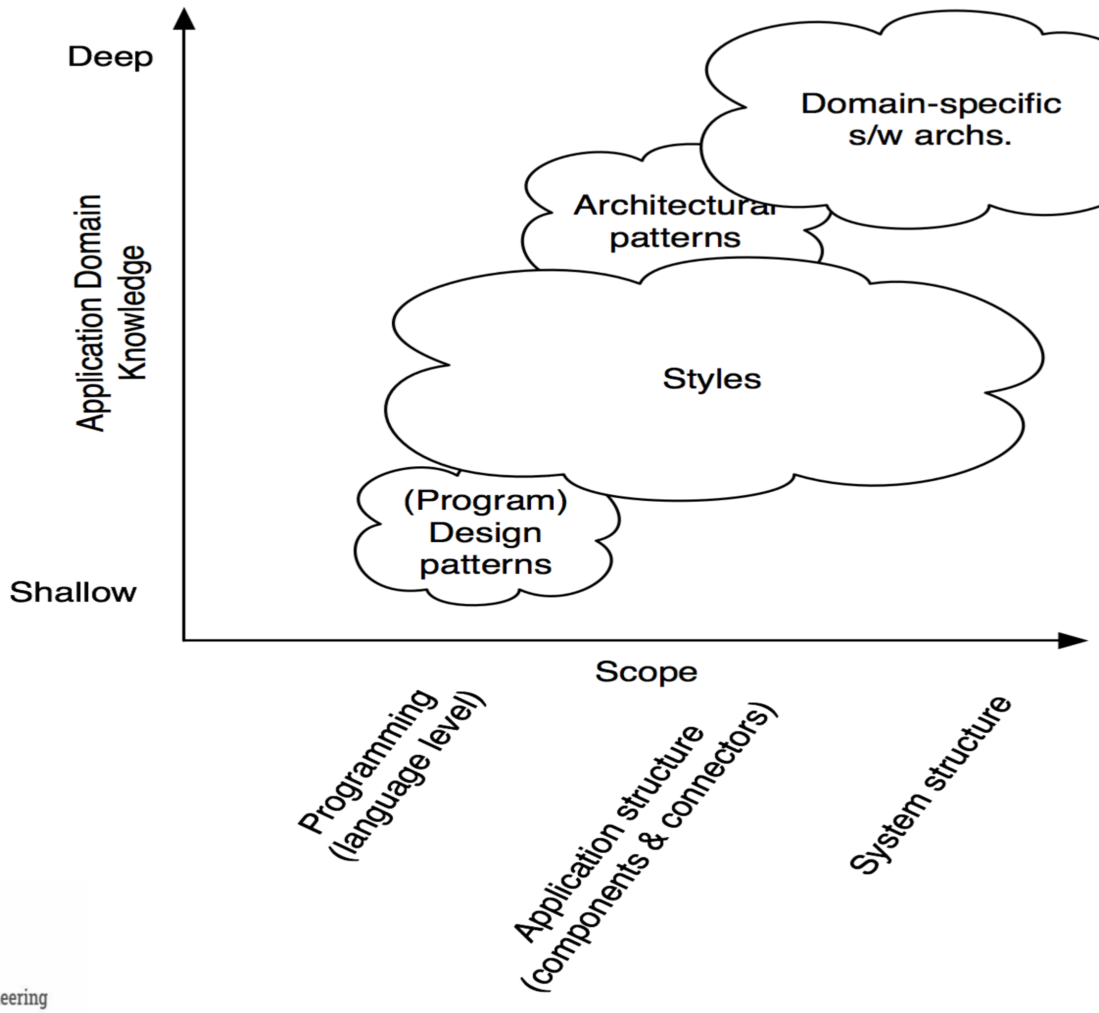
Potential Problems

- If the designer is unable to produce a set of feasible concepts, progress stops.
- As problems and products increase in size and complexity, the probability that any one individual can successfully perform the first steps decreases.
- The standard approach does not directly address the situation where system design is at stake, i.e. when relationship between a set of products is at issue.
- As complexity increases or the experience of the designer is not sufficient, alternative approaches to the design process must be adopted.

Alternative Design Strategies

- Standard
  - Linear model described above
- Cyclic
  - Process can revert to an earlier stage
- Parallel
  - Independent alternatives are explored in parallel
- Adaptive (“lay tracks as you go”)
  - The next design strategy of the design activity is decided at the end of a given stage
- Incremental
  - Each stage of development is treated as a task of incrementally improving the existing design

Patterns, Styles, and DSSAs(Domain-Specific Software Architectures)



Deep
Shallow
Application Domain Knowledge
Domain-specific s/w archs.
Architectural patterns
Styles
(Program) Design patterns
Scope
Programming (language level)
Application structure (components & connectors)
System structure

Domain-Specific Software Architectures

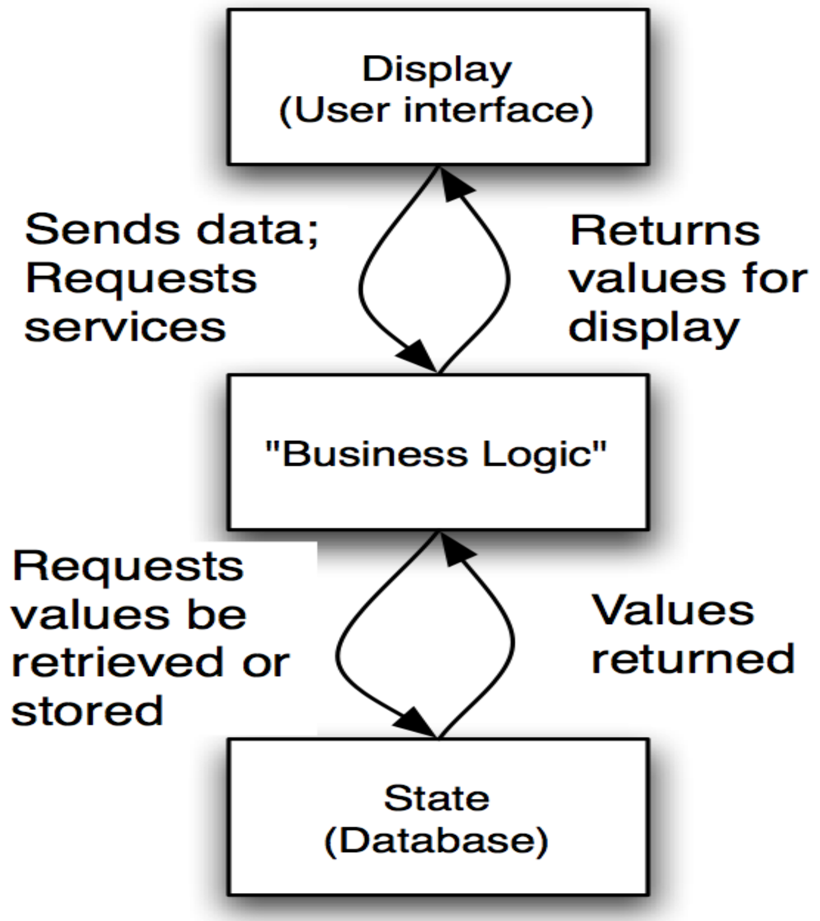
- A DSSA is an assemblage of software components
  - specialized for a particular type of task (domain),
  - generalized for effective use across that domain, and
  - composed in a standardized structure (topology) effective for building successful applications.
- Since DSSAs are specialized for a particular domain they are only of value if one exists for the domain wherein the engineer is tasked with building a new application.
- DSSAs are the pre-eminent means for maximal reuse of knowledge and prior development and hence for developing a new architectural design.

Architectural Patterns

- 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.
- Architectural patterns are similar to DSSAs but applied “at a lower level” and within a much narrower scope.

State-Logic-Display: Three-Tiered Pattern

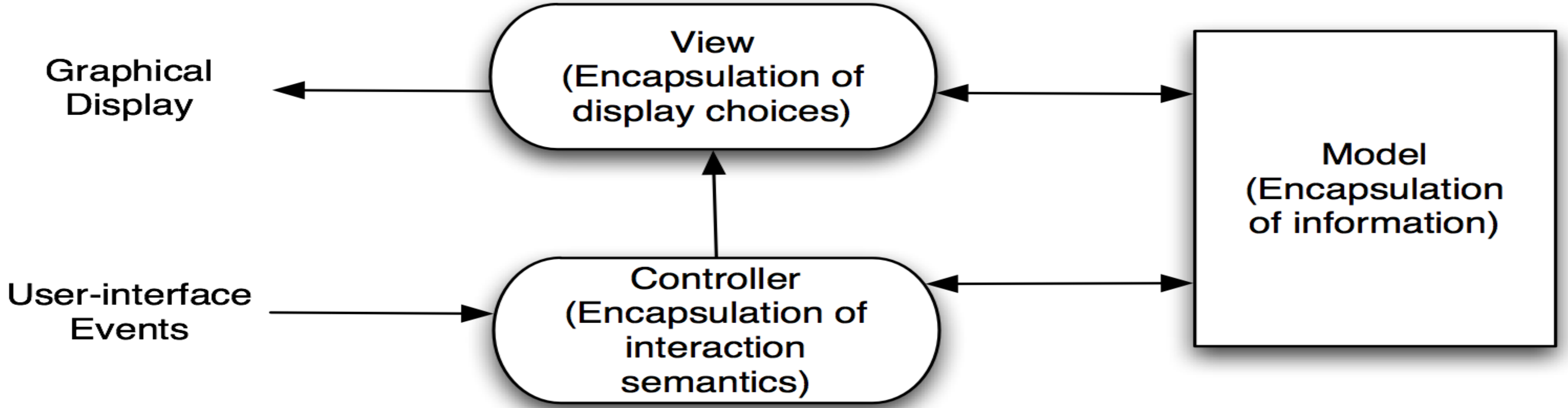
- Application Examples
  - Business applications
  - Multi-player games
  - Web-based applications


Display
(User interface)
Sends data; Requests services
Returns values for display
"Business Logic"
Requests values be retrieved or stored
Values returned
State
(Database)

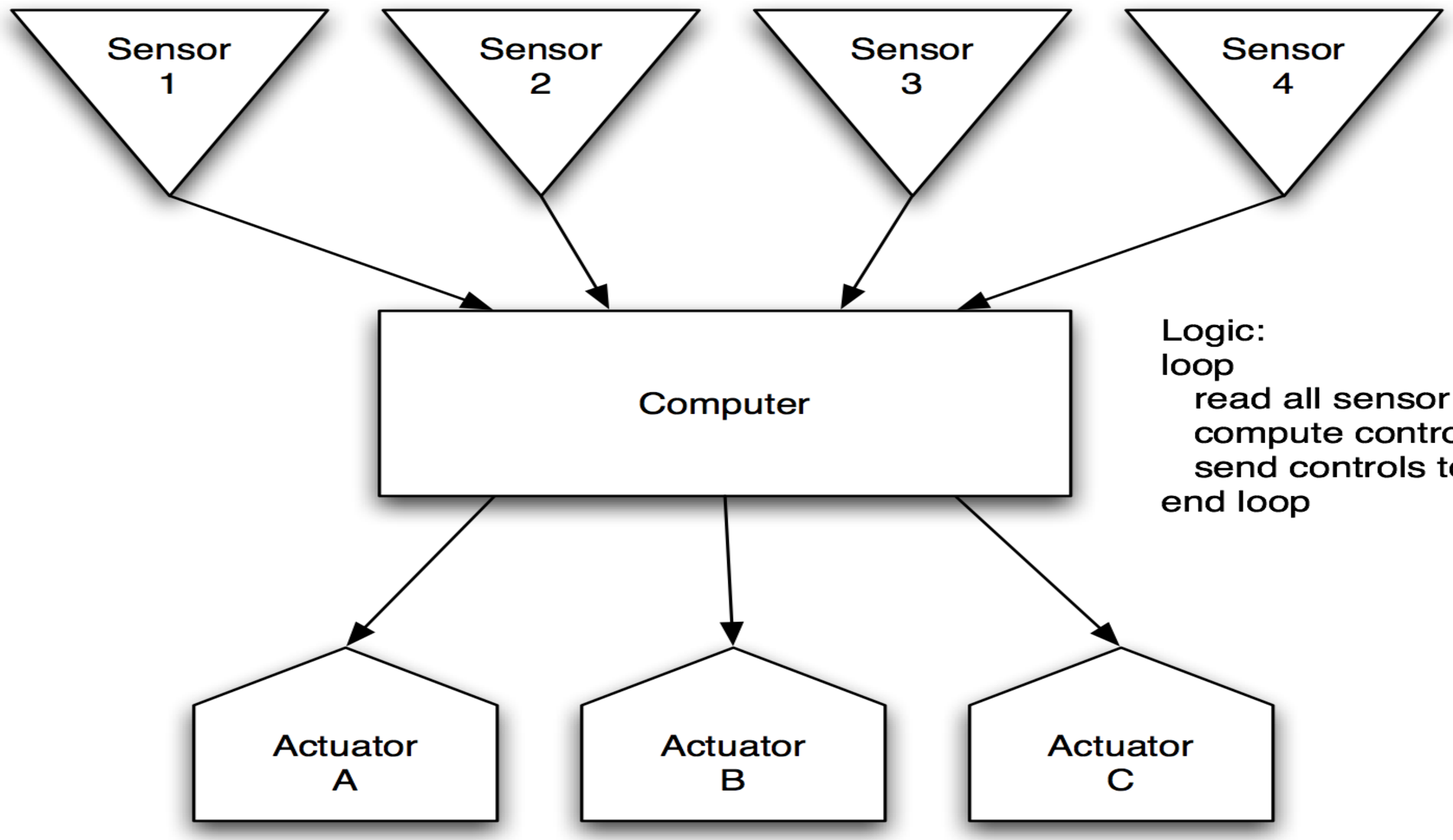
Model-View-Controller (MVC)

- Objective: Separation between information, presentation and user interaction.
- When a model object value changes, a notification is sent to the view and to the controller. So that the view can update itself and the controller can modify the view if its logic so requires.
- When handling input from the user the windowing system sends the user event to the controller; If a change is required, the controller updates the model object.

Model-View-Controller


Graphical Display
View (Encapsulation of display choices)
Model (Encapsulation of information)
User-interface Events
Controller (Encapsulation of interaction semantics)

Sense-Compute-Control



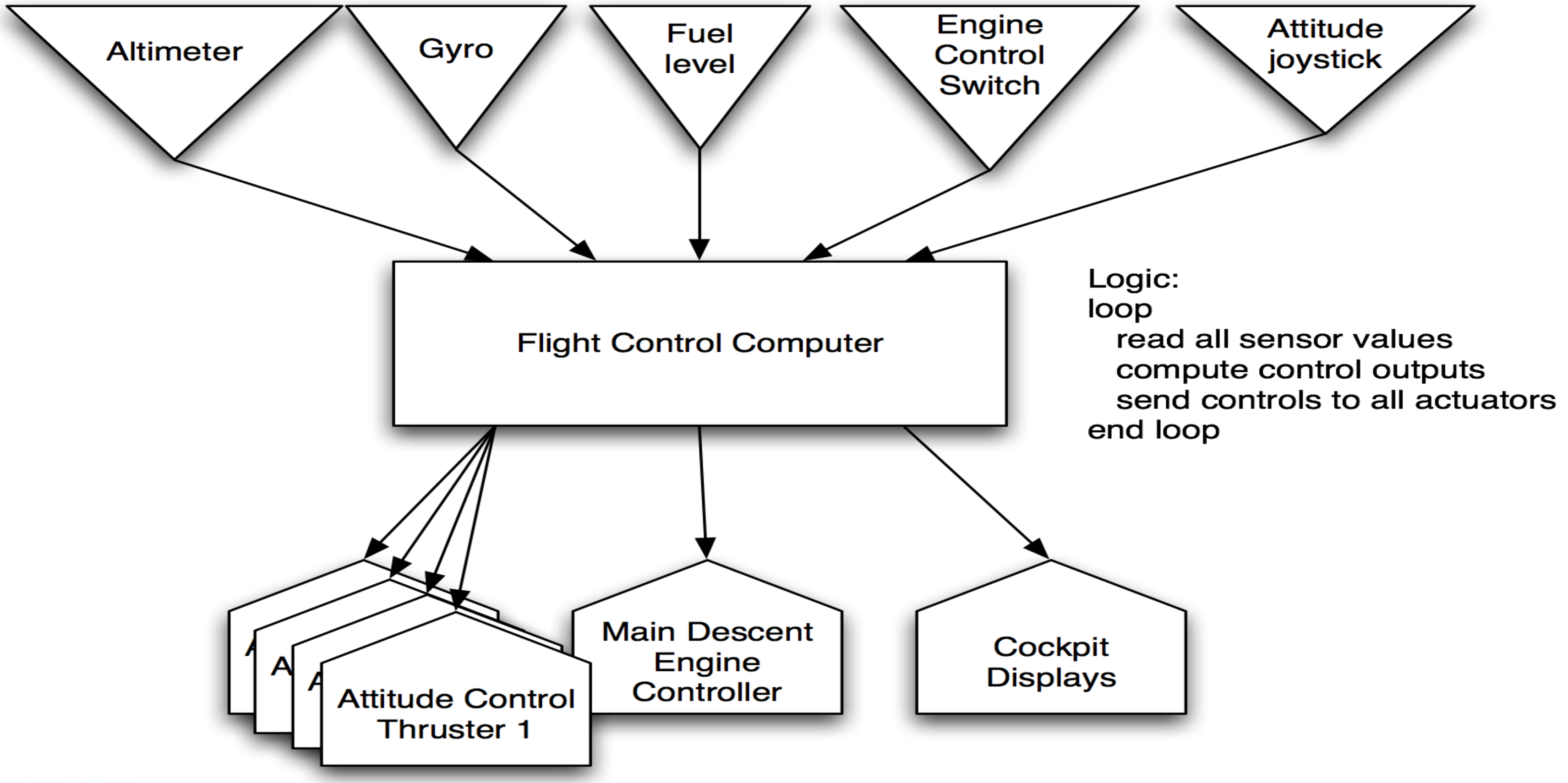
Logic:
```
loop
   read all sensor values
   compute control outputs
   send controls to all actuators
end loop
```

Objective: Structuring embedded control applications

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

Sense-Compute-Control LL



```
Logic:
loop
   read all sensor values
   compute control outputs
   send controls to all actuators
end loop
```

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
- 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
- Many styles exist

Definitions of Architectural Style

- 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.

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

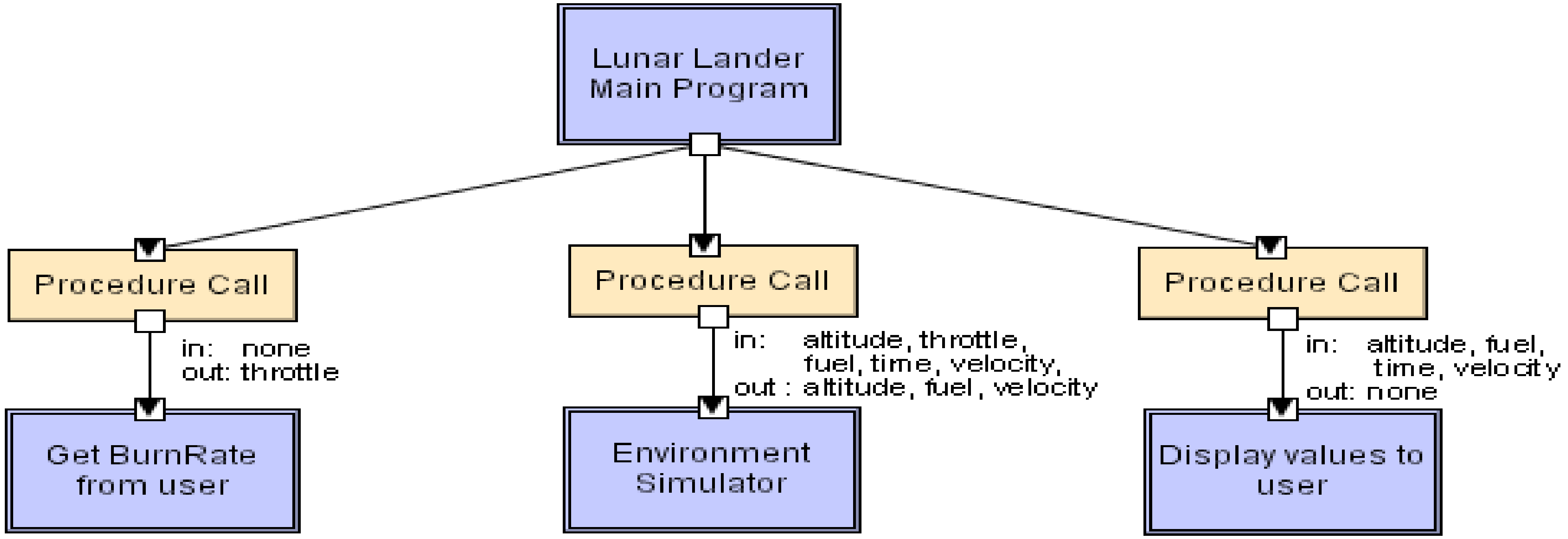
Some Common Styles

- 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
- “Derived” styles
  - C2
  - CORBA

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

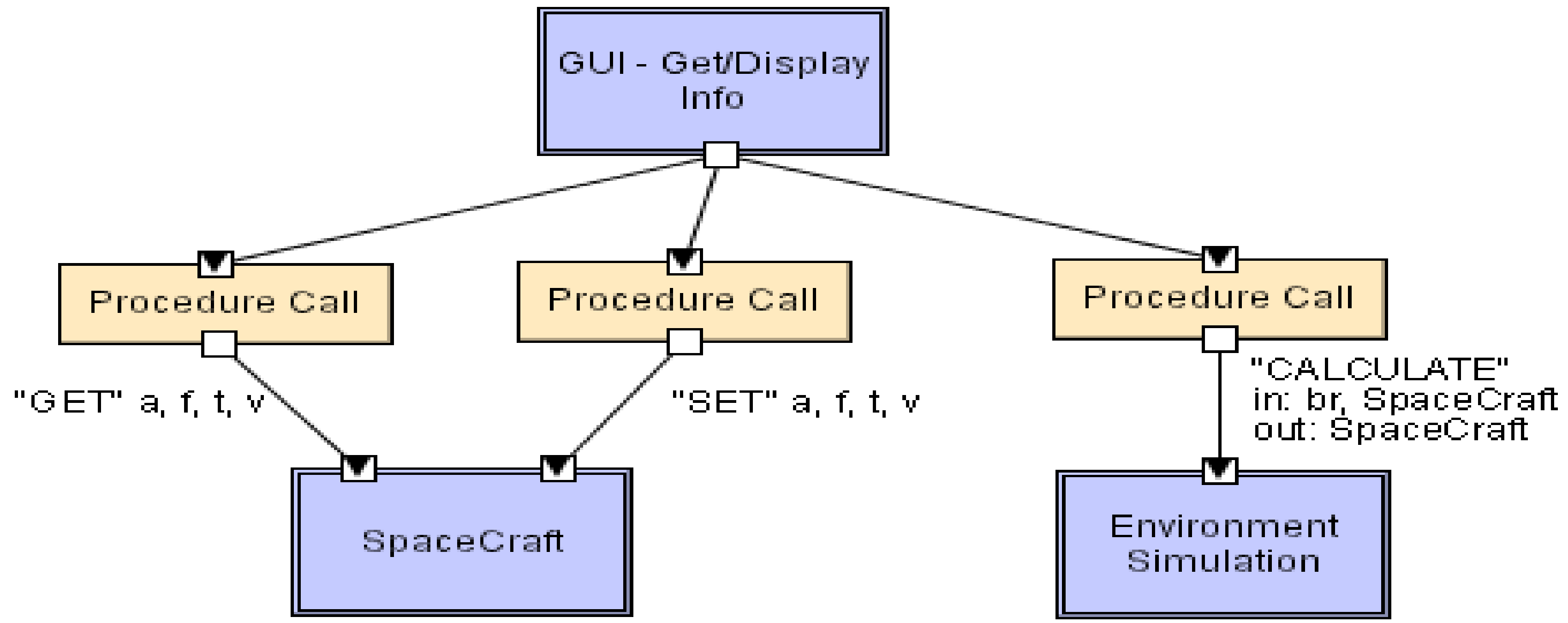
Main Program and Subroutines LL



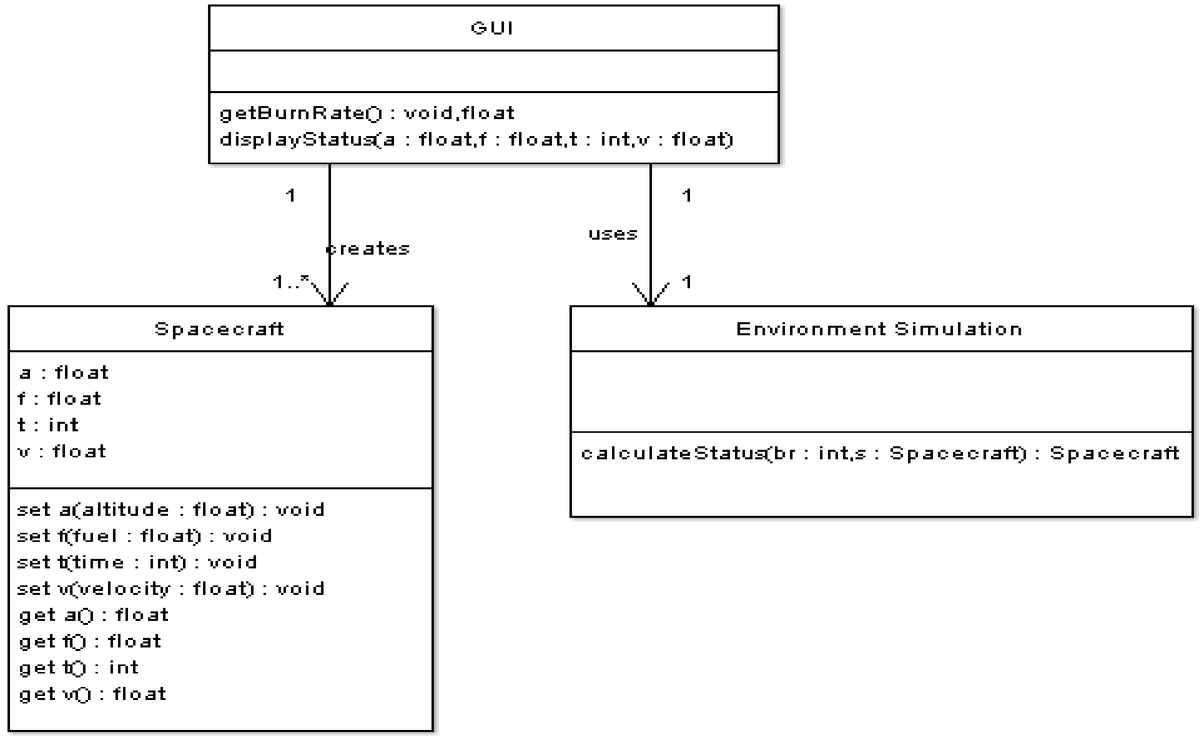
Object-Oriented Style

- 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


GUI
getBurnRate() : void,float
displayStatus(a : float,f : float,t : int,v : float)
1
creates
1..*
1
uses
1
Spacecraft
a : float
f : float
t : int
v : float
set a(altitude : float) : void
set f(fuel : float) : void
set t(time : int) : void
set v(velocity : float) : void
get a() : float
get f() : float
get t() : int
get v() : float
Environment Simulation
calculateStatus(br : int,s : Spacecraft) : Spacecraft

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

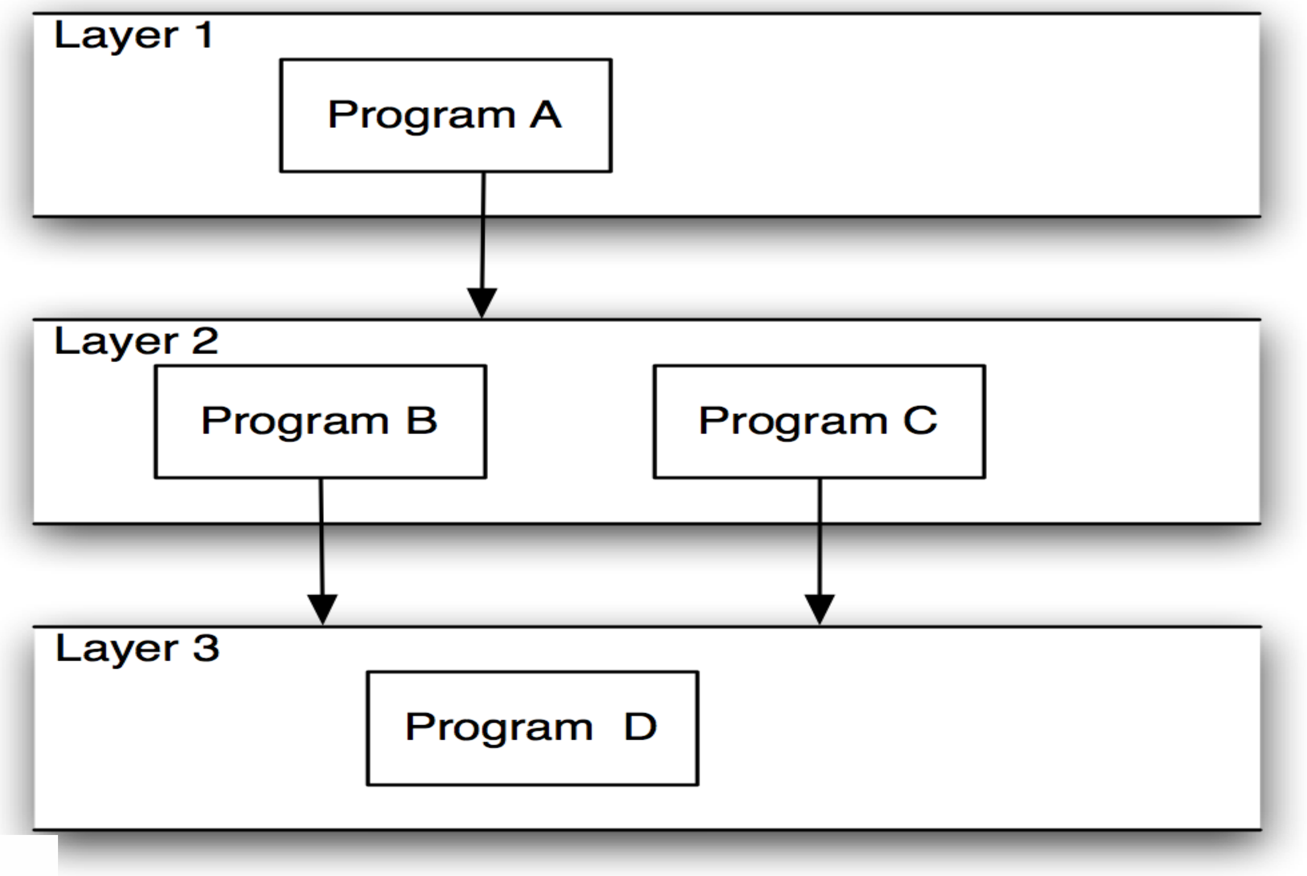
Layered Style (cont'd)

- 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

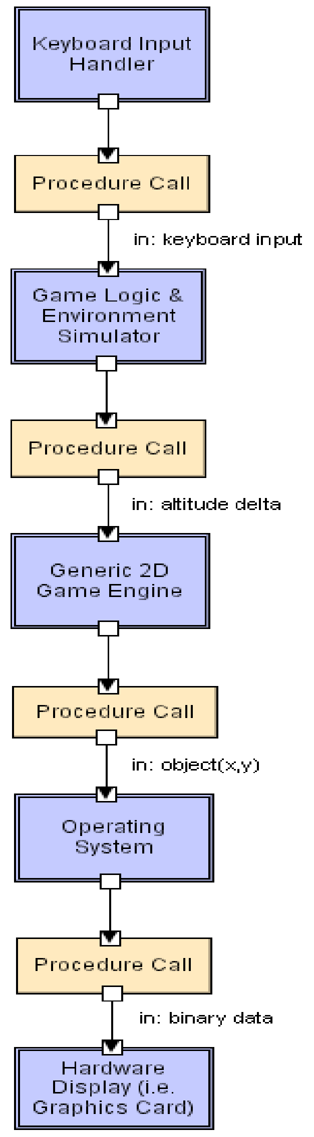
Layered Style (cont’d)

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

Layered Systems/Virtual Machines



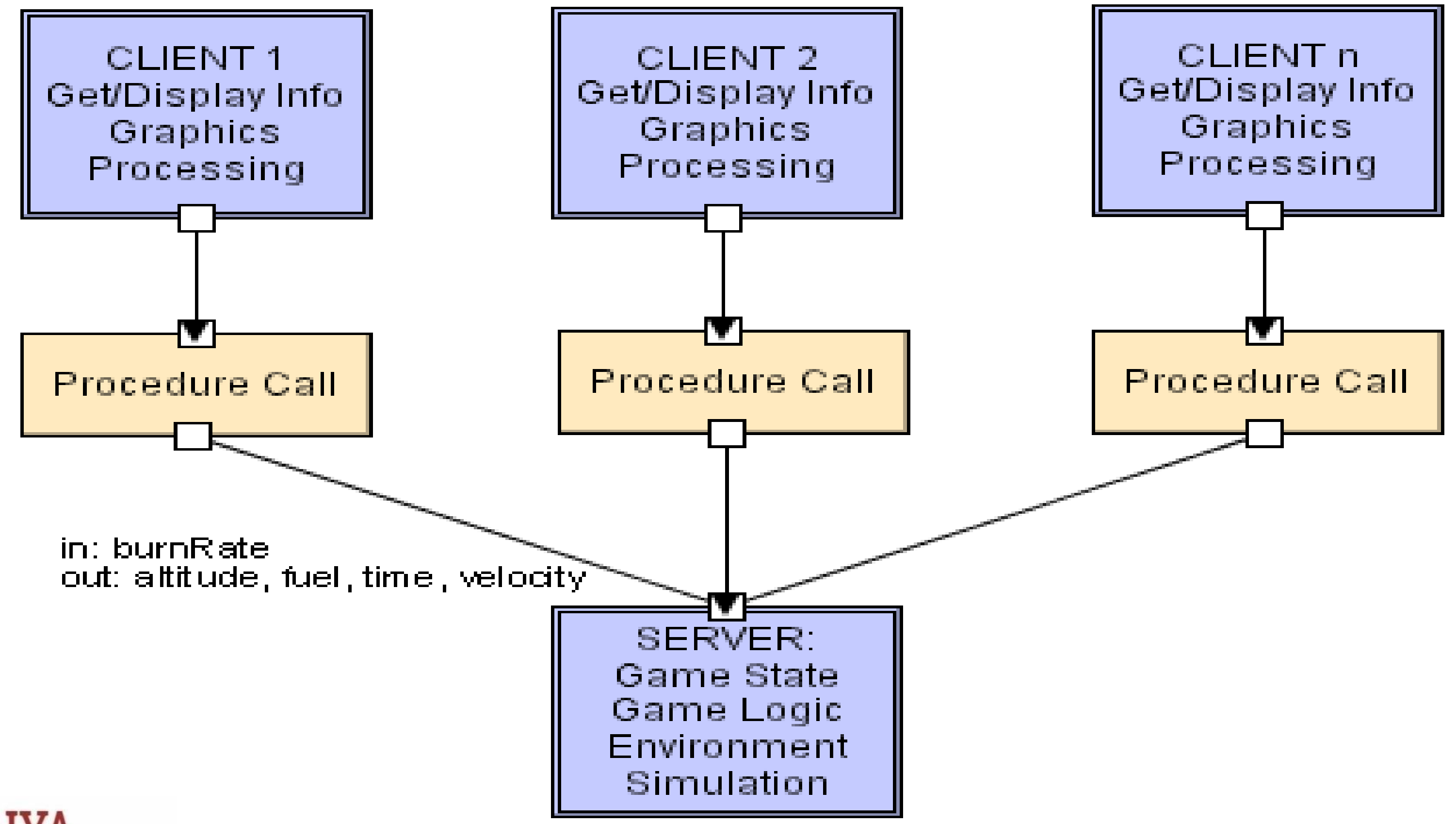
Layered LL


Keyboard Input Handler
Procedure Call
in: keyboard input
Game Logic & Environment Simulator
Procedure Call
in: altitude delta
Generic 2D Game Engine
Procedure Call
in: object(x,y)
Operating System
Procedure Call
in: binary data
Hardware Display (i.e. Graphics Card)

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



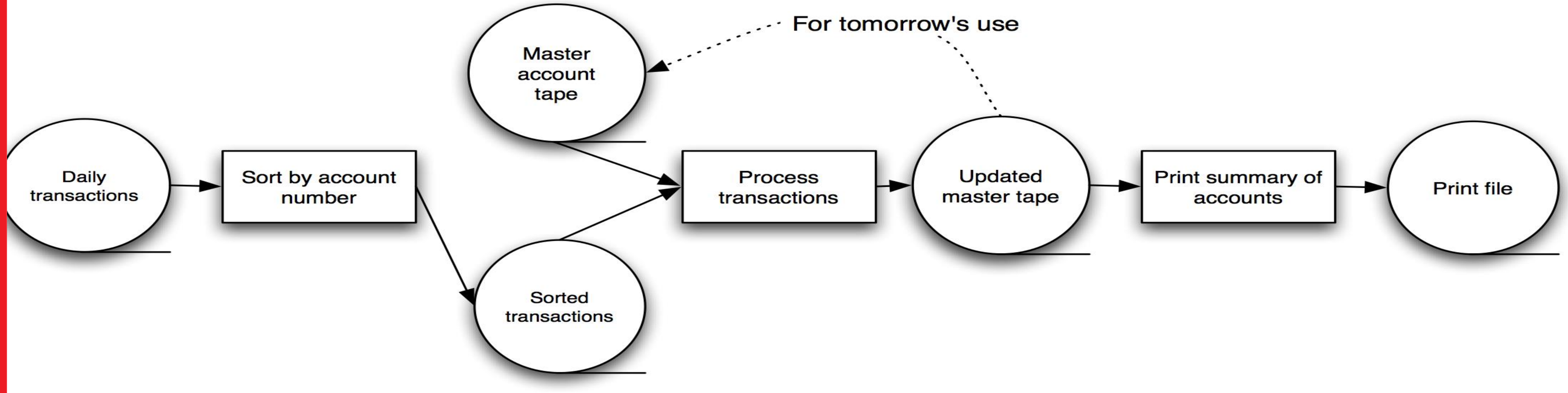
Data-Flow Styles

Batch Sequential

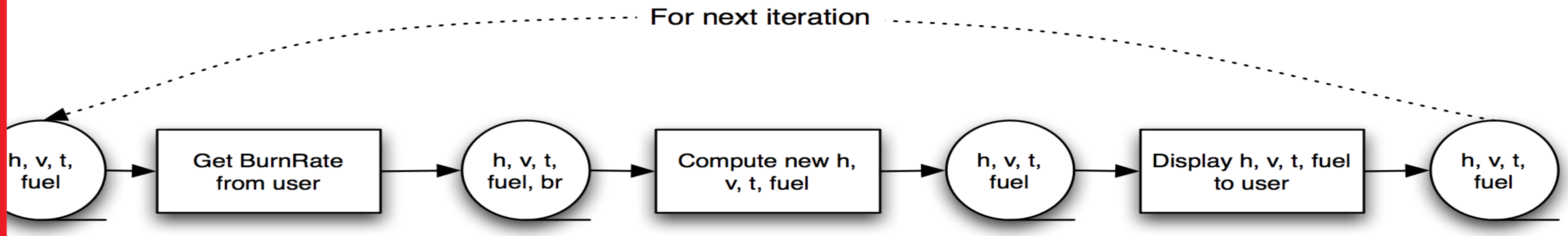
- Separate programs are executed in order; data is passed as an aggregate from one program to the next.
- Connectors: "The human hand" carrying tapes between the programs, a.k.a. "sneaker-net "
- Data Elements: Explicit, aggregate elements passed from one component to the next upon completion of the 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 signal processing
  - Distributed systems parallel programming
- Example: `ls invoices | grep -e August | sort`

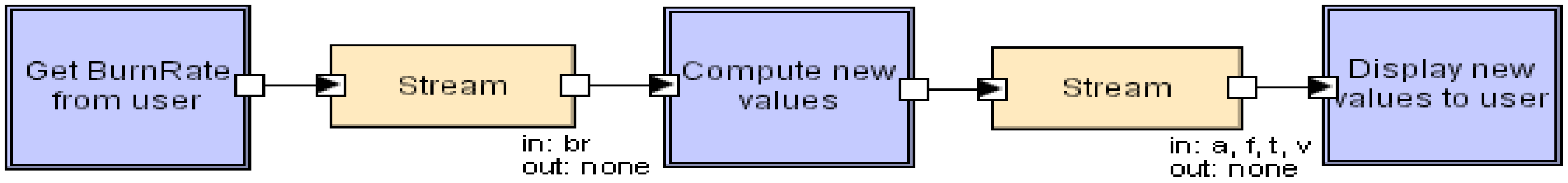
Pipe and Filter (cont’d)

- 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

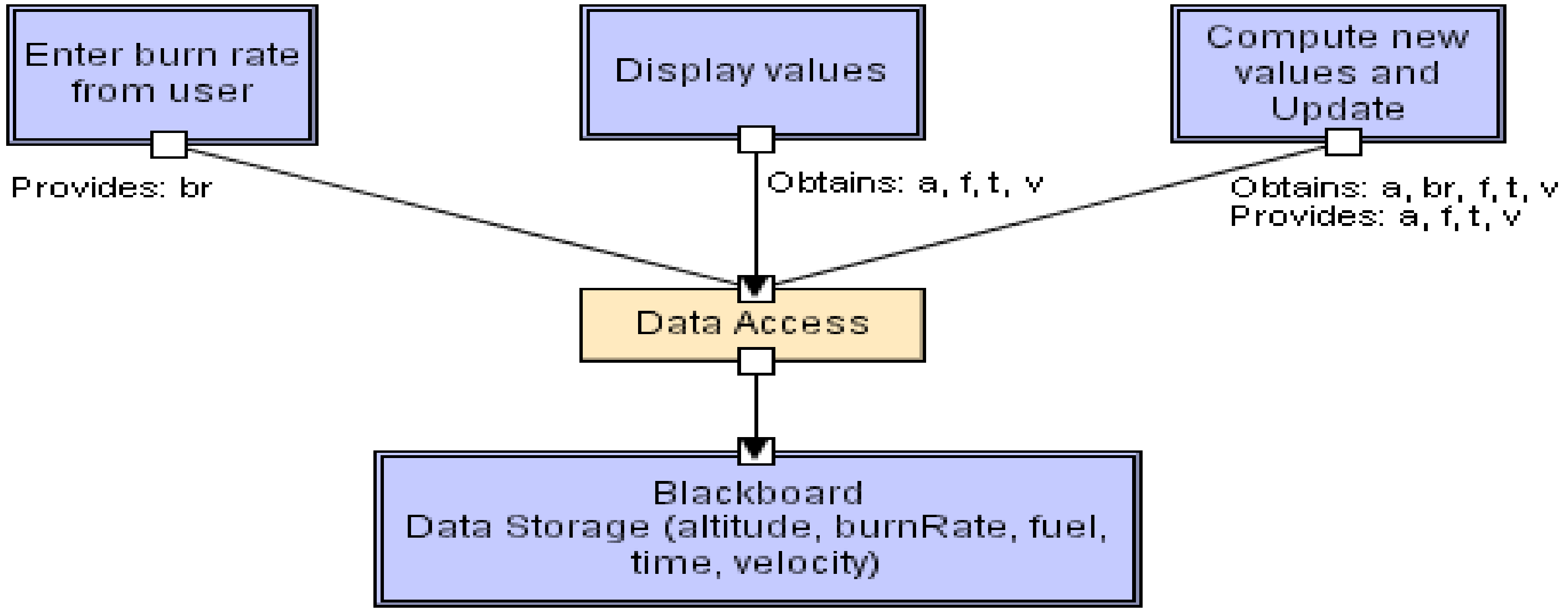
Pipe and Filter LL



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


Enter burn rate from user
Display values
Compute new values and Update
Provides: br
Obtains: a, f, t, v
Obtains: a, br, f, t, v
Provides: a, f, t, v
Data Access
Blackboard
Data Storage (altitude, burnRate, fuel, time, velocity)

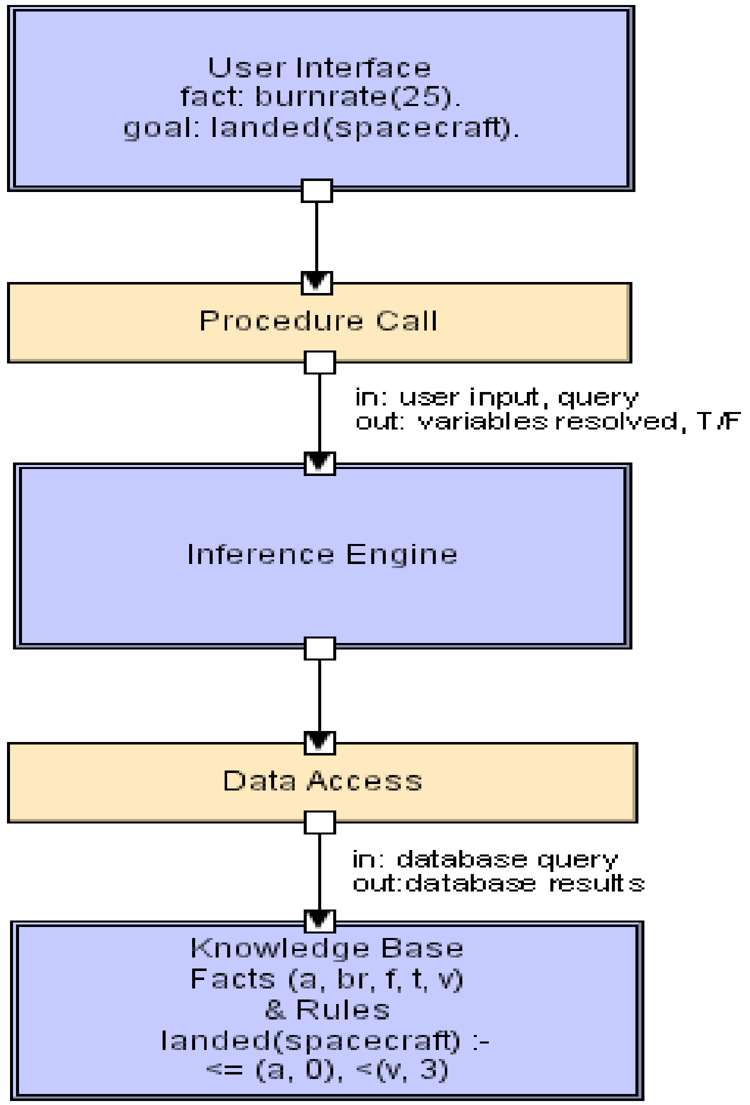
Rule-Based Style

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

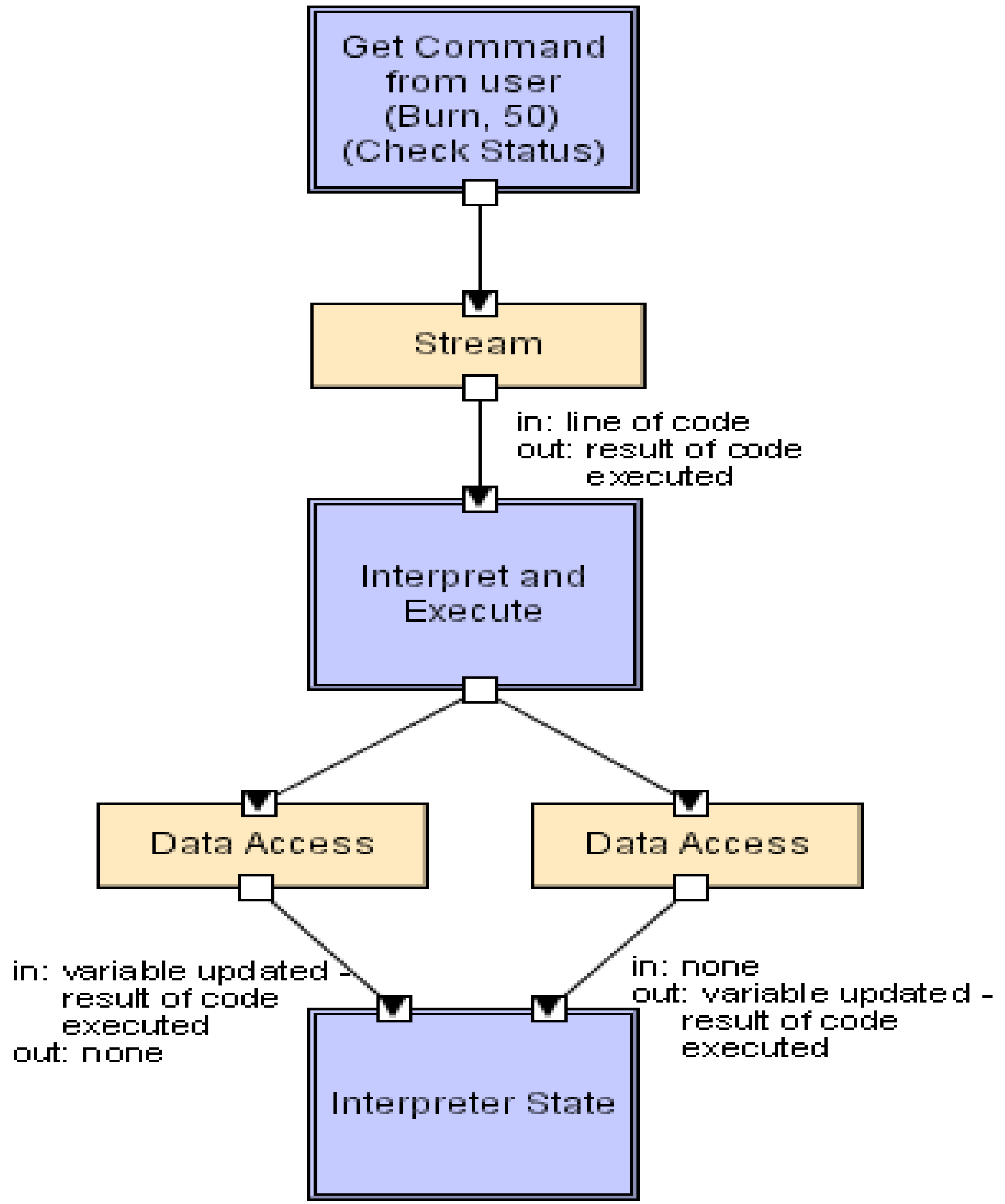


Interpreter Style

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

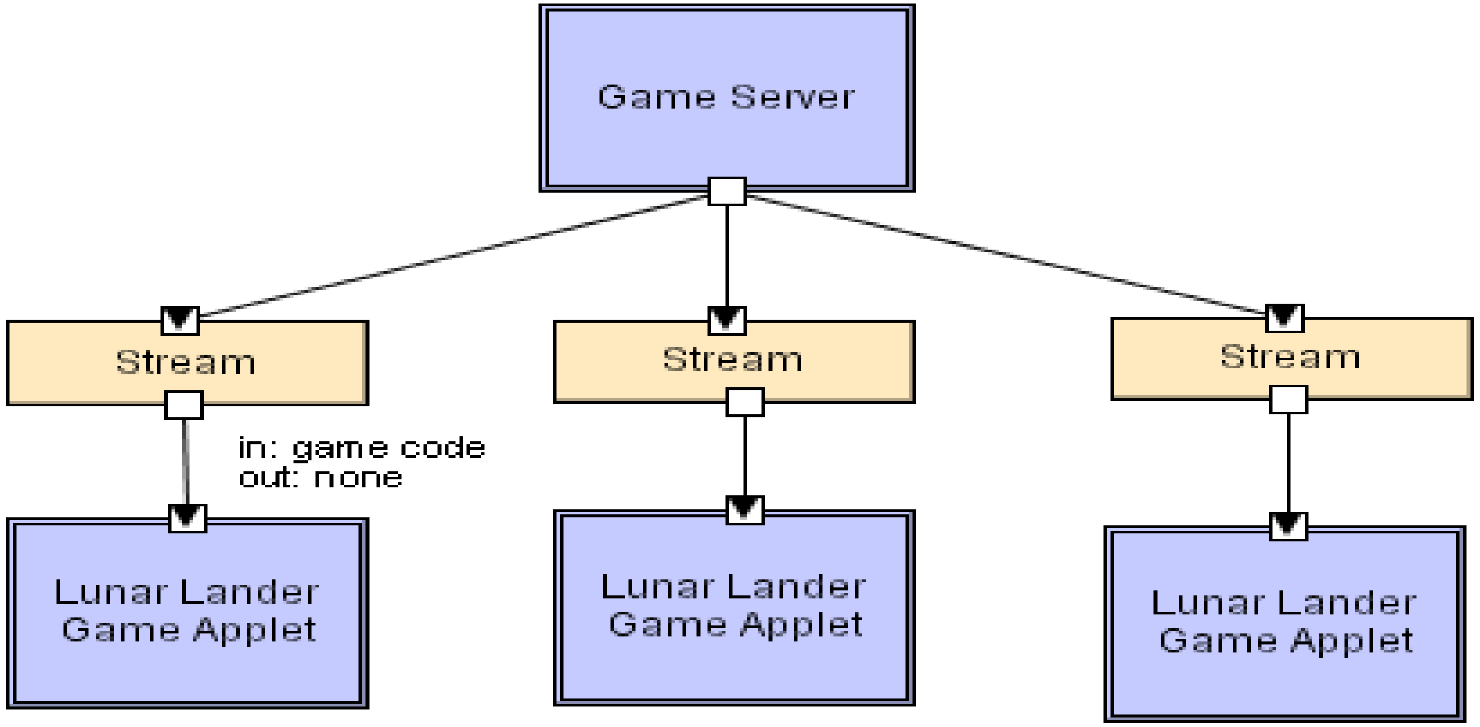
Interpreter LL


Get Command from user (Burn, 50) (Check Status)
Stream
in: line of code
out: result of code executed
Interpret and Execute
Data Access
Data Access
in: variable updated - result of code executed
out: none
in: none
out: variable updated - result of code executed
Interpreter State

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, VBScr

embec

cros.

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

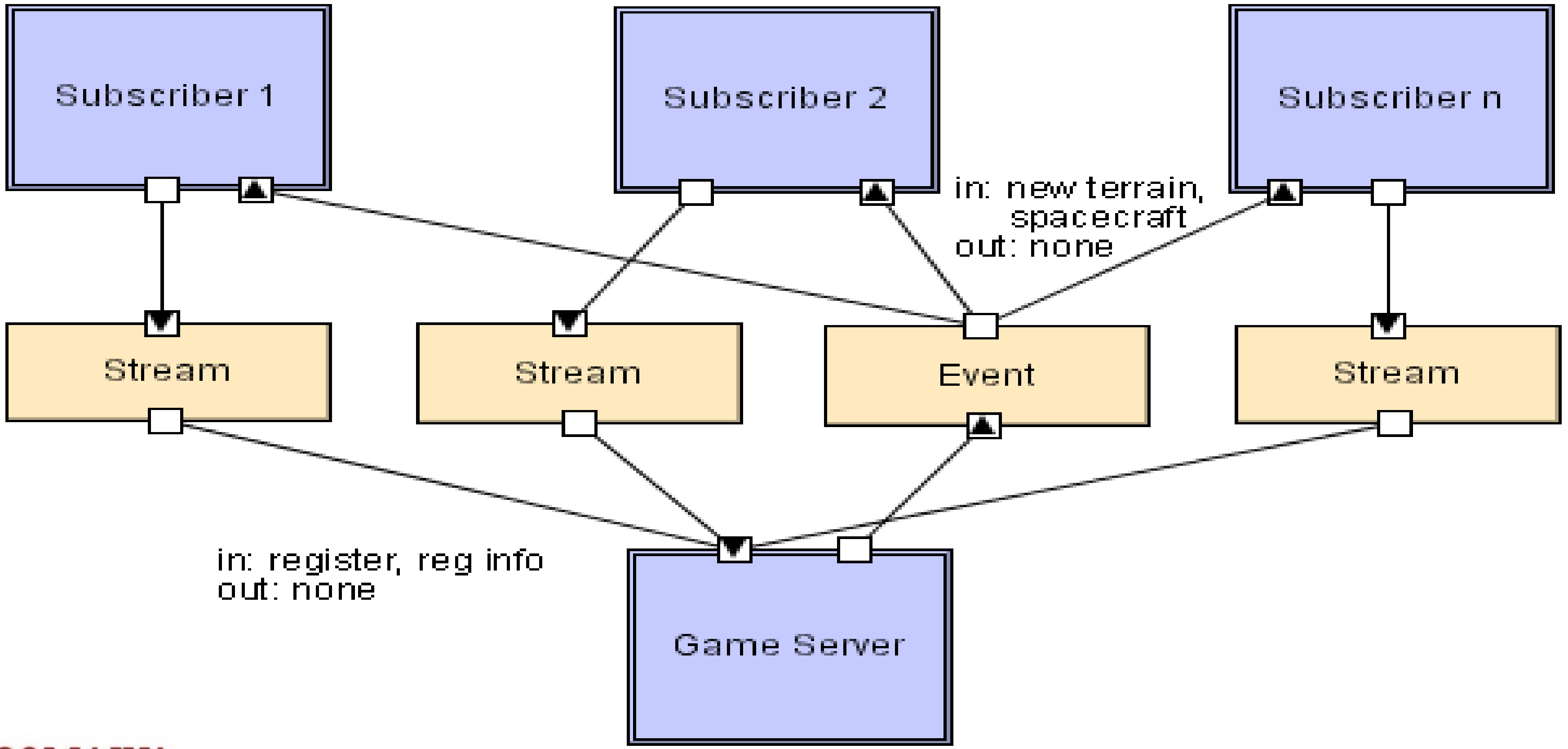
Publish-Subscribe

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

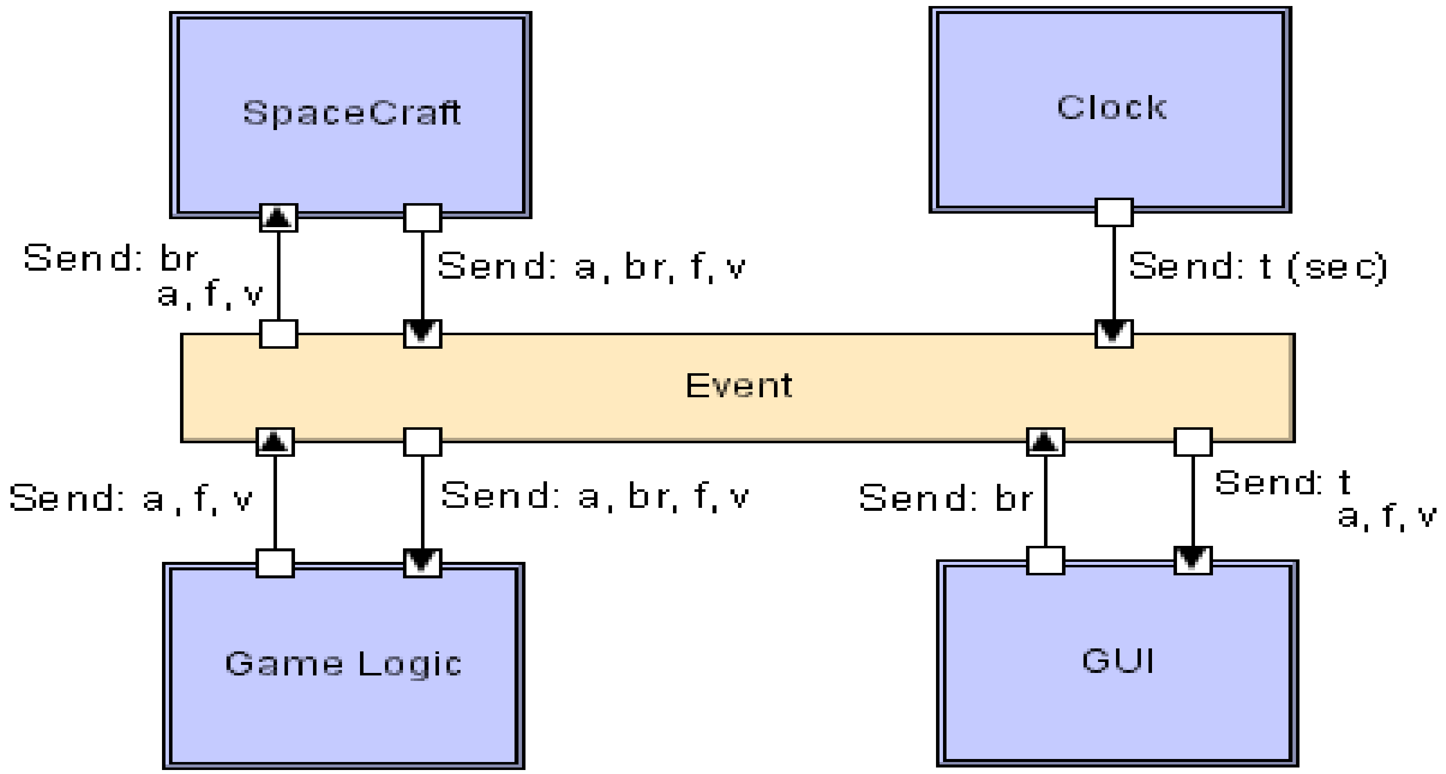
Pub-Sub LL


Subscriber 1
Subscriber 2
Subscriber n
in: new terrain, spacecraft
out: none
Stream
Stream
Event
Stream
in: register, reg info
out: none
Game Server

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



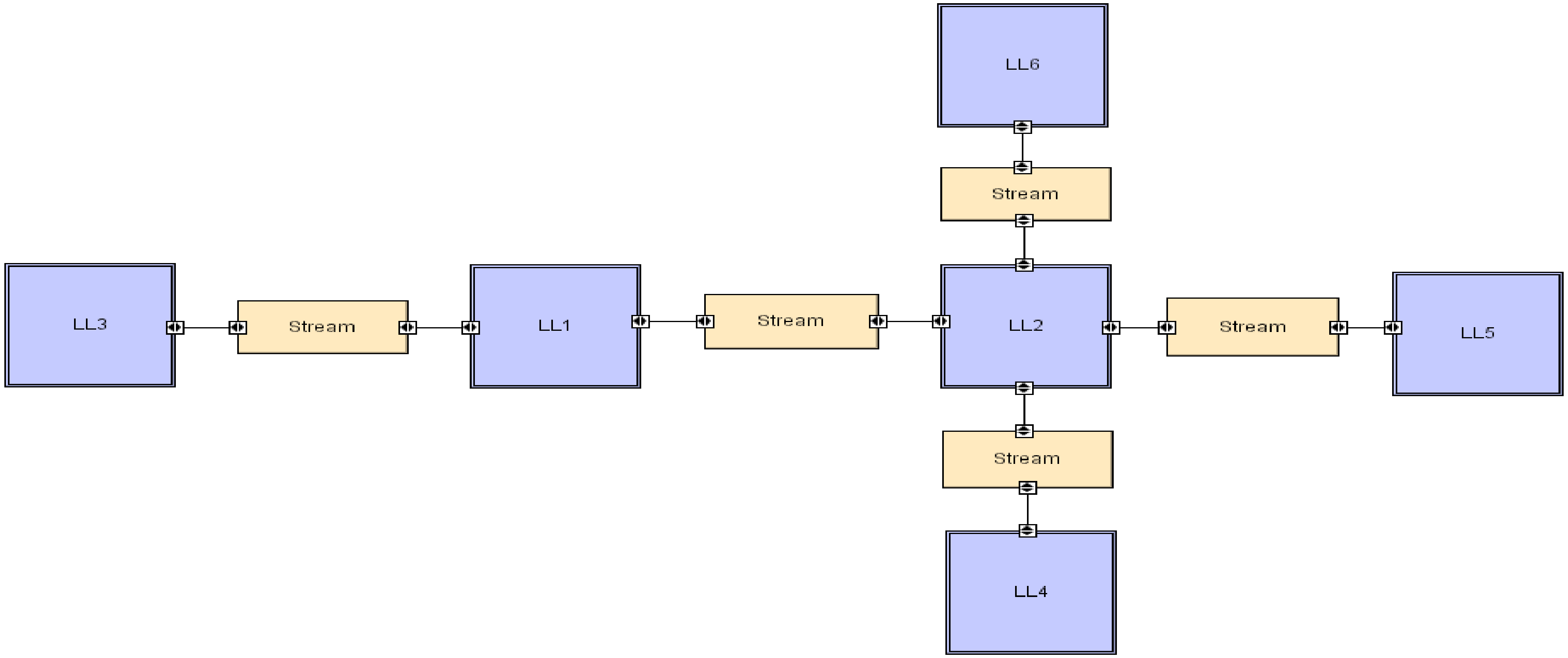
Peer-to-Peer Style

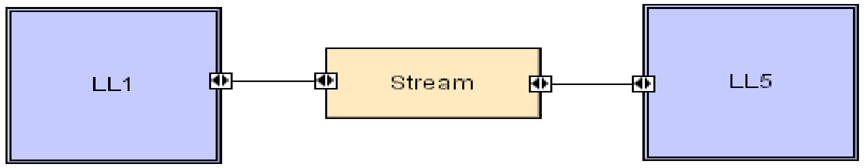
- State and behavior are distributed among peers which can act as either clients or servers.
- Peers: independent components, having their own state and control thread.
- Connectors: Network protocols, often custom.
- Data Elements: Network messages

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





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
  - CORBA

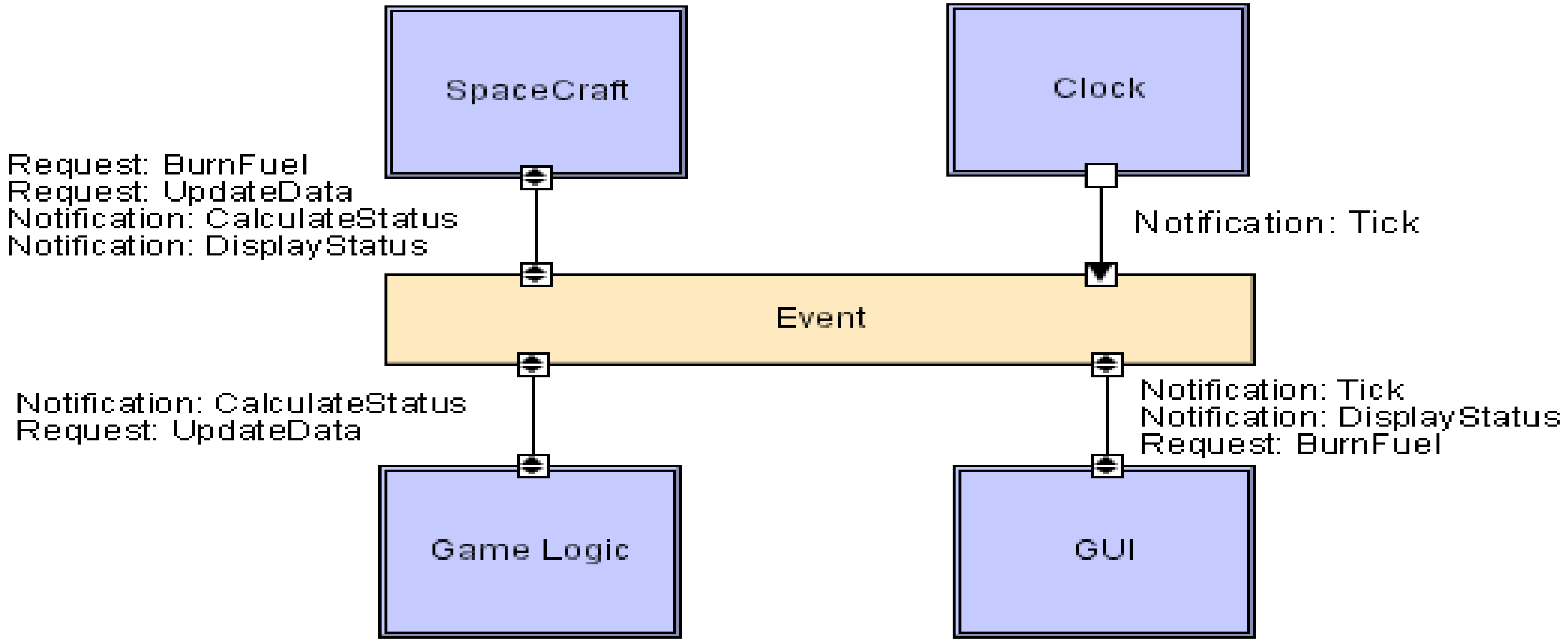
C2 Style

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

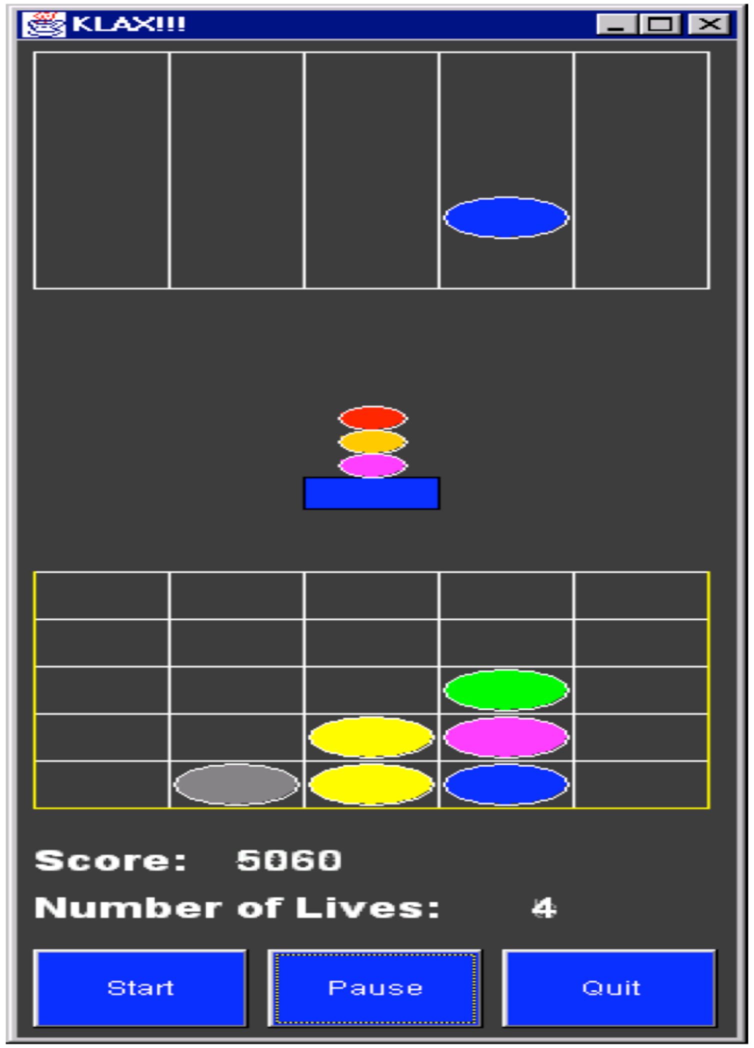
C2 Style (cont’d)

- 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.

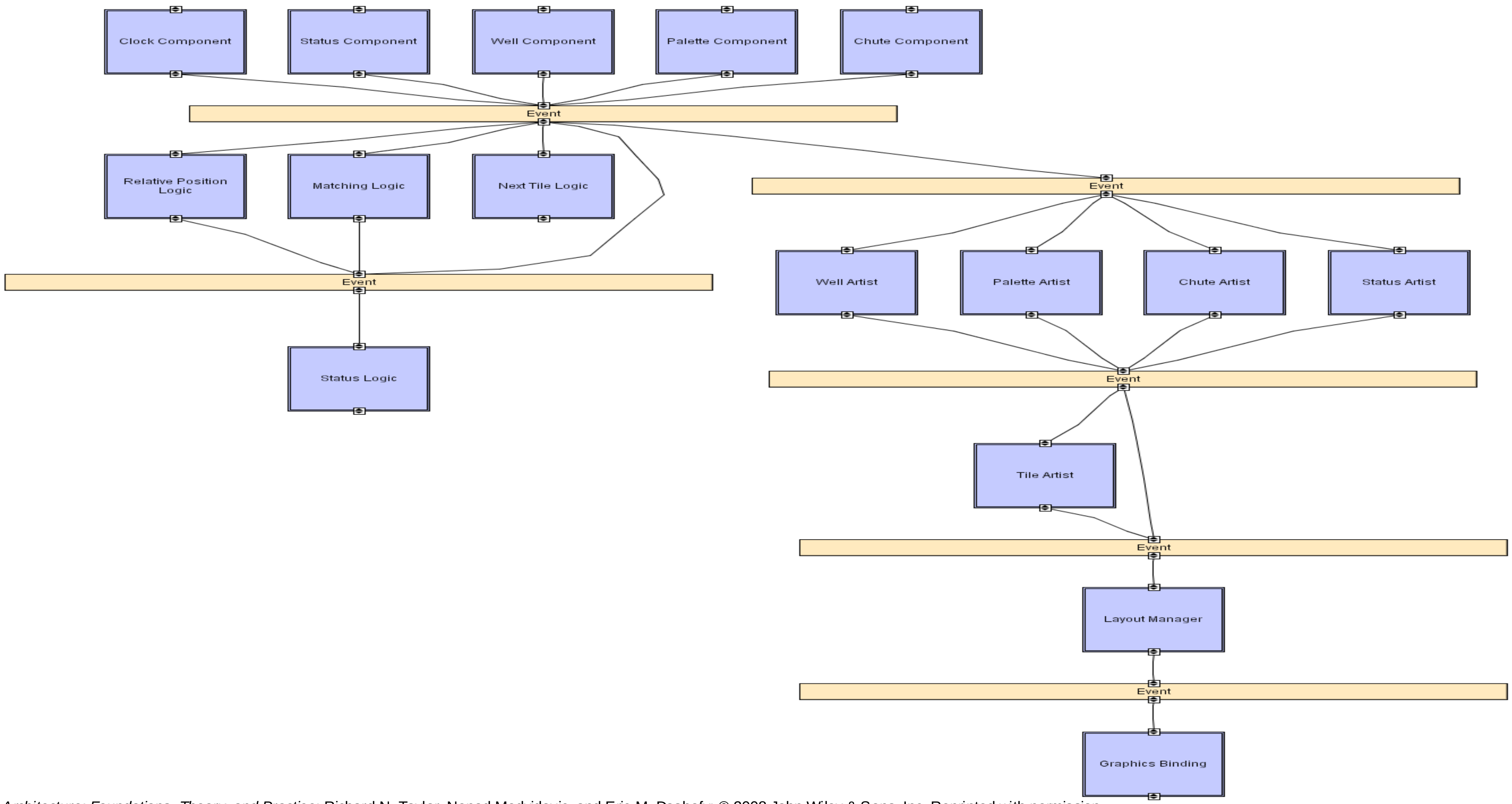
C2 LL


SpaceCraft
Clock
Request: BurnFuel
Request: UpdateData
Notification: CalculateStatus
Notification: DisplayStatus
Notification: Tick
Event
Notification: CalculateStatus
Request: UpdateData
Notification: Tick
Notification: DisplayStatus
Request: BurnFuel
Game Logic
GUI

KLAX



KLAX in C2

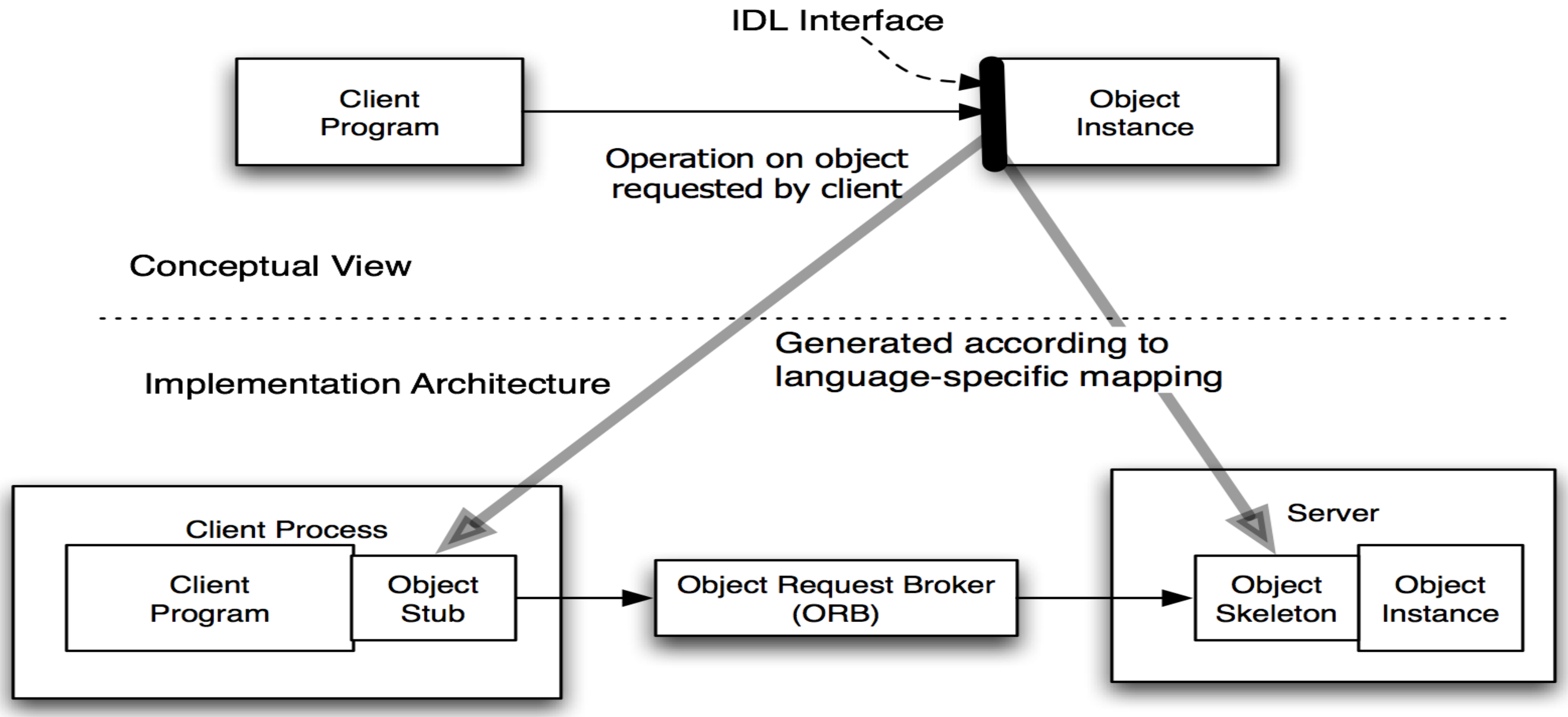


Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

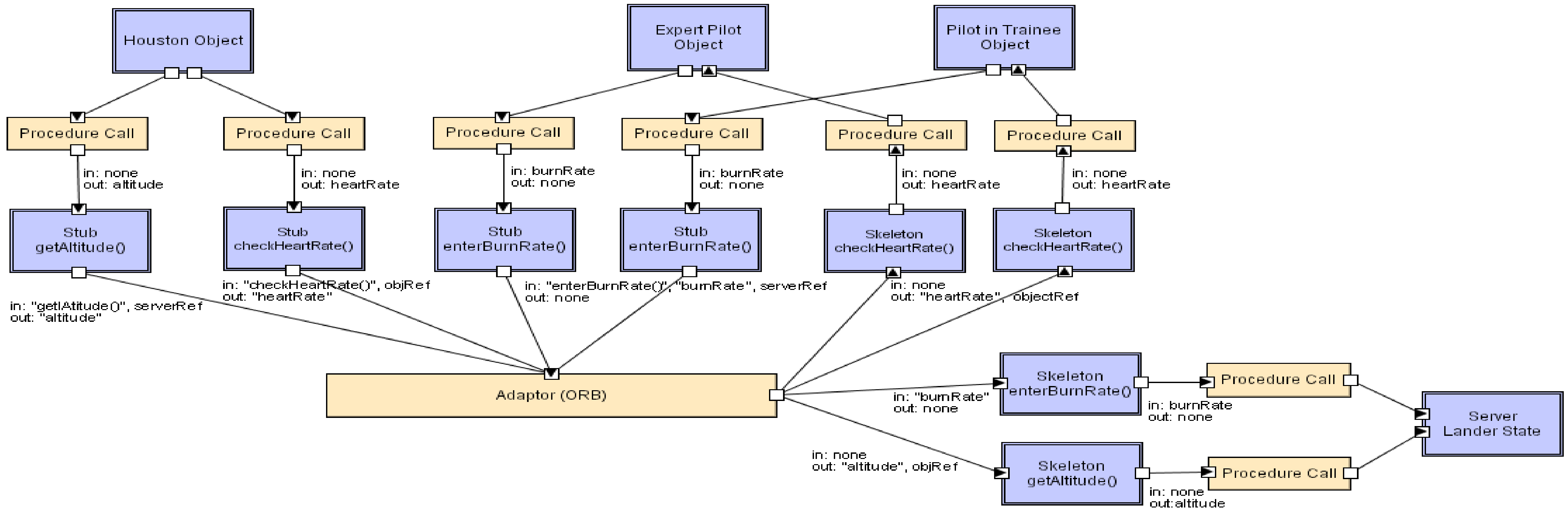
Distributed Objects: CORBA Common Object Request Broker Architecture

- “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

CORBA Concept and Implementation


IDL Interface
Client Program
Object Instance
Operation on object requested by client
Conceptual View
Implementation Architecture
Generated according to language-specific mapping
Client Process
Client Program
Object Stub
Object Request Broker (ORB)
Server
Object Skeleton
Object Instance

CORBA LL



Houston Object
Expert Pilot Object
Pilot in Trainee Object
Procedure Call
in: none
out: altitude
in: none
out: heartRate
in: burnRate
out: none
Stub getAltitude()
Stub checkHeartRate()
Stub enterBurnRate()
Skeleton checkHeartRate()
in: "getlAtitude()", serverRef
out: "altitude"
in: "checkHeartRate()", objRef
out: "heartRate"
in: "enterBurnRate()", "burnRate", serverRef
out: none
in: none
out: "heartRate", objectRef
Adaptor (ORB)
Skeleton enterBurnRate()
in: "burnRate"
out: none
Skeleton getAltitude()
in: none
out: "altitude", objRef
in: none
out:altitude
Server Lander State

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 desirous 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.

Design Recovery

- What happens if a system is already implemented but has no recorded architecture?
- The task of design recovery is
  - examining the existing code base
  - determining what the system’s components, connectors, and overall topology are.
- A common approach to architectural recovery is clustering of the implementation-level entities into architectural elements.
  - Syntactic clustering
  - Semantic clustering

Syntactic Clustering

- Focuses exclusively on the static relationships among code-level entities
- Can be performed without executing the system
- Embodies inter-component (a.k.a. coupling) and intra-component (a.k.a. cohesion) connectivity
- May ignore or misinterpret many subtle relationships, because dynamic information is missing

Semantic Clustering

- Includes all aspects of a system’s domain knowledge and information about the behavioral similarity of its entities.
- Requires interpreting the system entities’ meaning, and possibly executing the system on a representative set of inputs.
- Difficult to automate
- May also be difficult to avail oneself of it

When There’s No Experience to Go On…

- The first effort a designer should make in addressing a novel design challenge is to attempt to determine that it is genuinely a novel problem.
- Basic Strategy
  - Divergence – shake off inadequate prior approaches and discover or admit a variety of new ideas
  - Transformation – combination of analysis and selection
  - Convergence – selecting and further refining ideas
- Repeatedly cycling through the basic steps until a feasible solution emerges.

Analogy Searching

- Examine other fields and disciplines unrelated to the target problem for approaches and ideas that are analogous to the problem.
- Formulate a solution strategy based upon that analogy.
- A common “unrelated domain” that has yielded a variety of solutions is nature, especially the biological sciences.
    - E.g., Neural Networks

Brainstorming

- Technique of rapidly generating a wide set of ideas and thoughts pertaining to a design problem
    - without (initially) devoting effort to assessing the feasibility.
- Brainstorming can be done by an individual or, more commonly, by a group.
- Problem: A brainstorming session can generate a large number of ideas… all of which might be low-quality.
- The chief value of brainstorming is in identifying categories of possible designs, not any specific design solution suggested during a session.
- After brainstorm the design process may proceed to the Transformation and Convergence steps.

“Literature” Searching

- Examining published information to identify material that can be used to guide or inspire designers
- Many historically useful ways of searching “literature” are available
- Digital library collections make searching extraordinarily faster and more effective
  - IEEE Xplore
  - ACM Digital Library
  - Google Scholar
- The availability of free and open-source software adds special value to this technique.

Morphological Charts

- The essential idea:
    - identify all the primary functions to be performed by the desired system
    - for each function identify a means of performing that function
    - attempt to choose one means for each function such that the collection of means performs all the required functions in a compatible manner.
- The technique does not demand that the functions be shown to be independent when starting out.
- Sub-solutions to a given problem do not need to be compatible with all the sub-solutions to other functions in the beginning.

Removing Mental Blocks

- If you can’t solve the problem, change the problem to one you can solve.
  - If the new problem is “close enough” to what is needed, then closure is reached.
  - If it is not close enough, the solution to the revised problem may suggest new venues for attacking the original.

Controlling the Design Strategy

- The potentially chaotic nature of exploring diverse approaches to the problem demands that some care be used in managing the activity
- Identify and review critical decisions
- Relate the costs of research and design to the penalty for taking wrong decisions
- Insulate uncertain decisions
- Continually re-evaluate system “requirements” in light of what the design exploration yields

Insights from Requirements

- In many cases new architectures can be created based upon experience with and improvement to pre-existing architectures.
- Requirements can use a vocabulary of known architectural choices and therefore reflect experience.
- The interaction between past design and new requirements means that many critical decisions for a new design can be identified or made as a requirement

Insights from Implementation

- Constraints on the implementation activity may help shape the design.
- Externally motivated constraints might dictate
  - Use of a middleware
  - Use of a particular programming language
  - Software reuse
- Design and implementation may proceed cooperatively and contemporaneously
  - Initial partial implementation activities may yield critical performance or feasibility information

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.