Designing Architectures

Software Architecture

Chapter 4

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

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

Method

Objectives

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

Engineering Design Process

- 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

Identifying a Viable Strategy

- Use fundamental design tools: abstraction and modularity
  - *But how?*
- Inspiration, where inspiration is needed; predictable techniques elsewhere
  - *But where is creativity required?*
- Applying own experience or experience of others

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

A Few Definitions… from the *OED Online*

- Abstraction: "The act or process of separating in thought, of considering a thing independently of its associations; or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs."
- Reification: "The mental conversion of … [an] abstract concept into a thing."
- Deduction: "The process of drawing a conclusion from a principle already known or assumed; spec. in Logic, inference by reasoning from generals to particulars; opposed to INDUCTION."
- Induction: "The process of inferring a general law or principle from the observation of particular instances (opposed to DEDUCTION, q.v.)."

Abstraction and the Simple Machines

- What concepts should be chosen at the outset of a design task?
  - One technique: Search for a "simple machine" that serves as an abstraction of a potential system that will perform the required task
  - For instance, what kind of simple machine makes a software system embedded in a fax machine?
    - At core, it is basically just a little state machine
- Simple machines provide a plausible first conception of how an application might be built
- Every application domain has its common simple machines

Simple Machines

Domain	**Simple Machines**
Graphics	Pixel arrays Transformation matrices Widgets Abstract depiction graphs
Word processing	Structured documents Layouts
Process control	Finite state machines
Income Tax Software	Hypertext Spreadsheets Form templates
Web pages	Hypertext Composite documents
Scientific computing	Matrices Mathematical functions
Banking	Spreadsheets Databases Transactions

Choosing the Level and Terms of Discourse

- Any attempt to use abstraction as a tool must choose a level of discourse, and once that is chosen, must choose the terms of discourse
- *Alternative 1*: initial level of discourse is one of the application as a whole (step-wise refinement)
- *Alternative 2:* work, initially, at a level lower than that of the whole application
    - Once several such sub-problems are solved, they can be composed together to form an overall solution
- *Alternative 3*: work, initially, at a level above that of the desired application
    - E.g., handling simple application input with a general parser

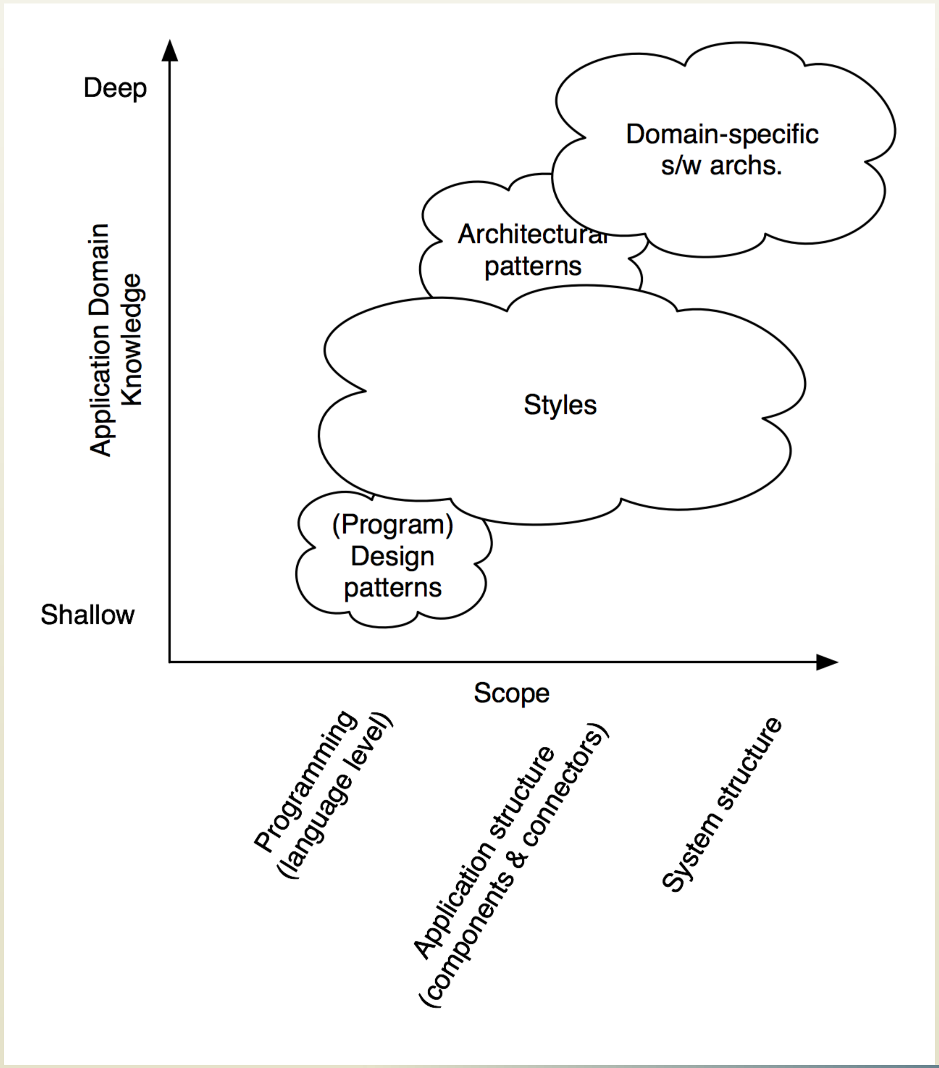
Separation of Concerns

- Separation of concerns is the subdivision of a problem into (hopefully) independent parts
  - The interface of an ATM is different from the logic of handling a bank account
- The difficulties arise when the issues are either actually or apparently intertwined
- Separation 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 (communication)

The Grand Tool: Refined Experience

- Experience must be reflected upon and refined
- The lessons from prior work include not only the lessons of successes, but also the lessons arising from failure
- Learn from success and failure of other engineers
  - Literature
  - Conferences
- Experience can provide that initial feasible set of "alternative arrangements for the design as a whole"
- Thousands of systems development experience has yielded many approaches and lessons, from DSSAs to simple programming languages design patterns

Patterns, Styles, and DSSAs


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

Patterns, Styles, and DSSAs (cont'd)

- The horizontal axis refers to the scope of applicability of the body of knowledge, ranging from design patterns focusing on object-oriented programming techniques to DSSAs that deal with complete applications
- The vertical axis reflects the amount of domain knowledge represented by (or encoded within) the body of knowledge, ranging from design patterns that encode small amounts of knowledge but are very generally applicable to a variety of applications, to DSSAs that encode substantial knowledge about the design of a specific domain's applications but cannot be used for other domains
- Note that these distinctions are blurry and the boundaries in the diagram are fuzzy

Domain-Specific Software Architectures

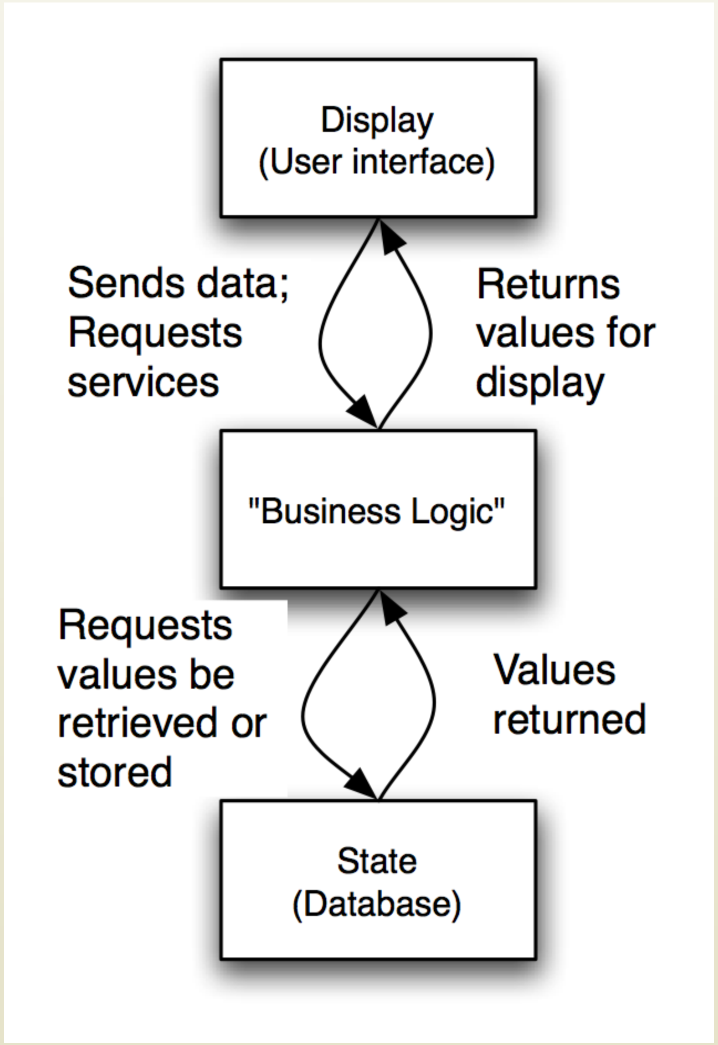
- A DSSA is an assemblage of software components
  - Specialized for a particular type of task (domain)
  - Generalized for effective use across that domain, and
  - Composed in a standardized structure (topology) effective for building successful applications
- Since DSSAs are specialized for a particular 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

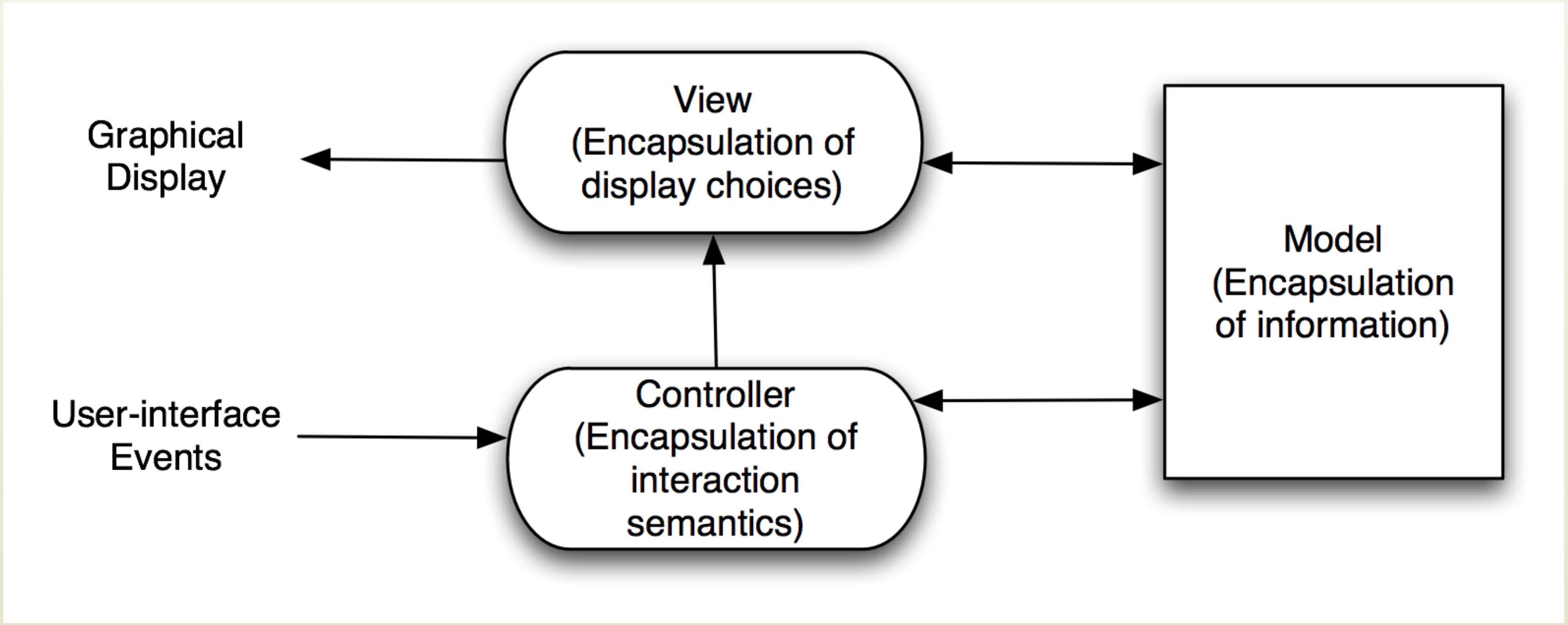
- Application Examples
  - Business applications (business logic can be banking transaction rules)
  - Multi-player games (business logic implements game rules, data store maintains the game state, each player has is own display)
  - Web-based applications (business logic is a Web server)


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

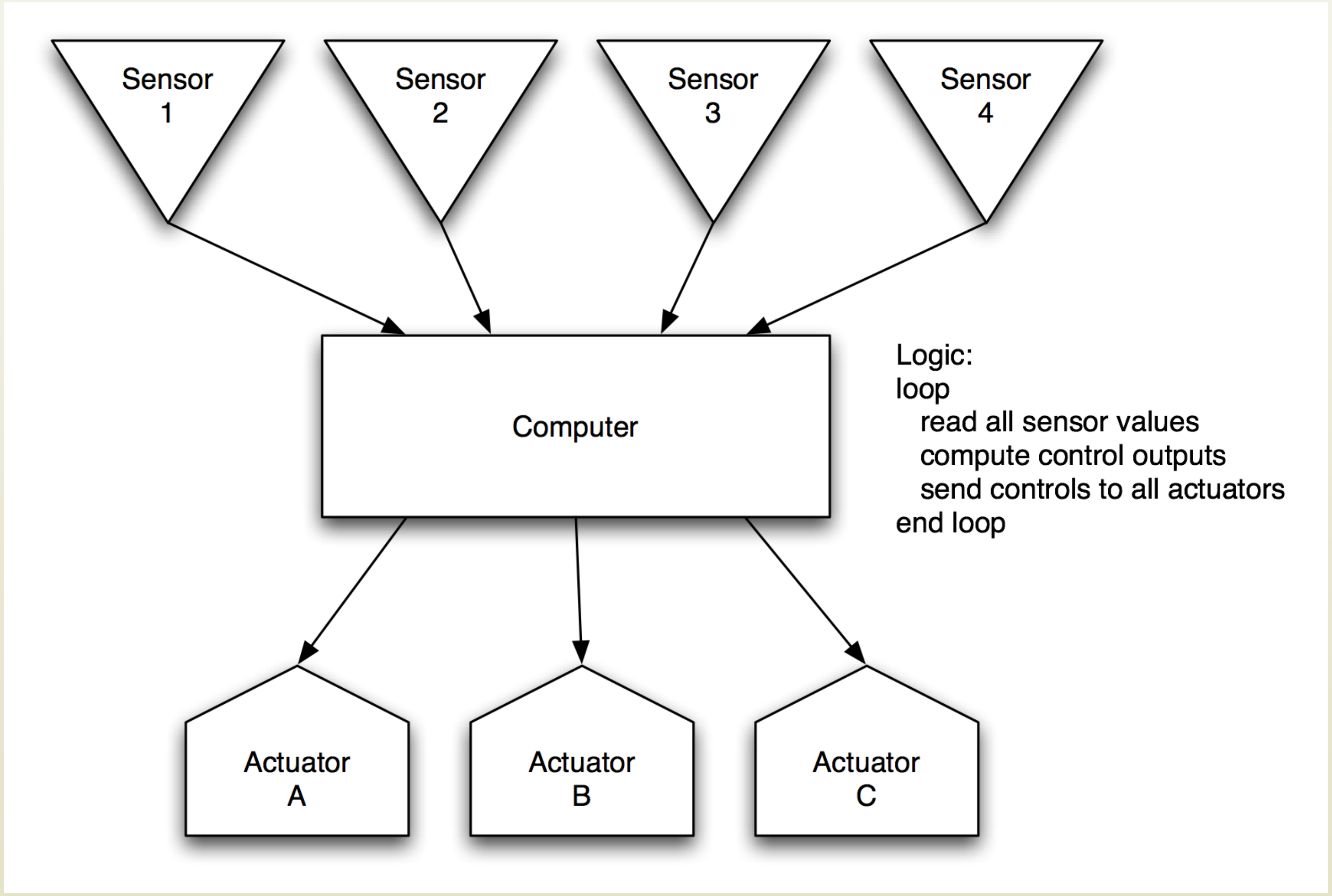
Model-View-Controller


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

Sense-Compute-Control (SCC)

- Typically, used in structuring embedded control applications (and, more generally, real-time software)
- A computer is embedded in some application; sensors from various devices are connected to the computer and may be sampled to determine their values
- Also, attached to the computer are hardware actuators
- The architectural pattern here is simply one of cycling through the steps of reading all the sensor values, executing a set of control laws or functions, and then sending outputs to the various actuators
- A clock is typically also used to control the timing of these activities

Sense-Compute-Control



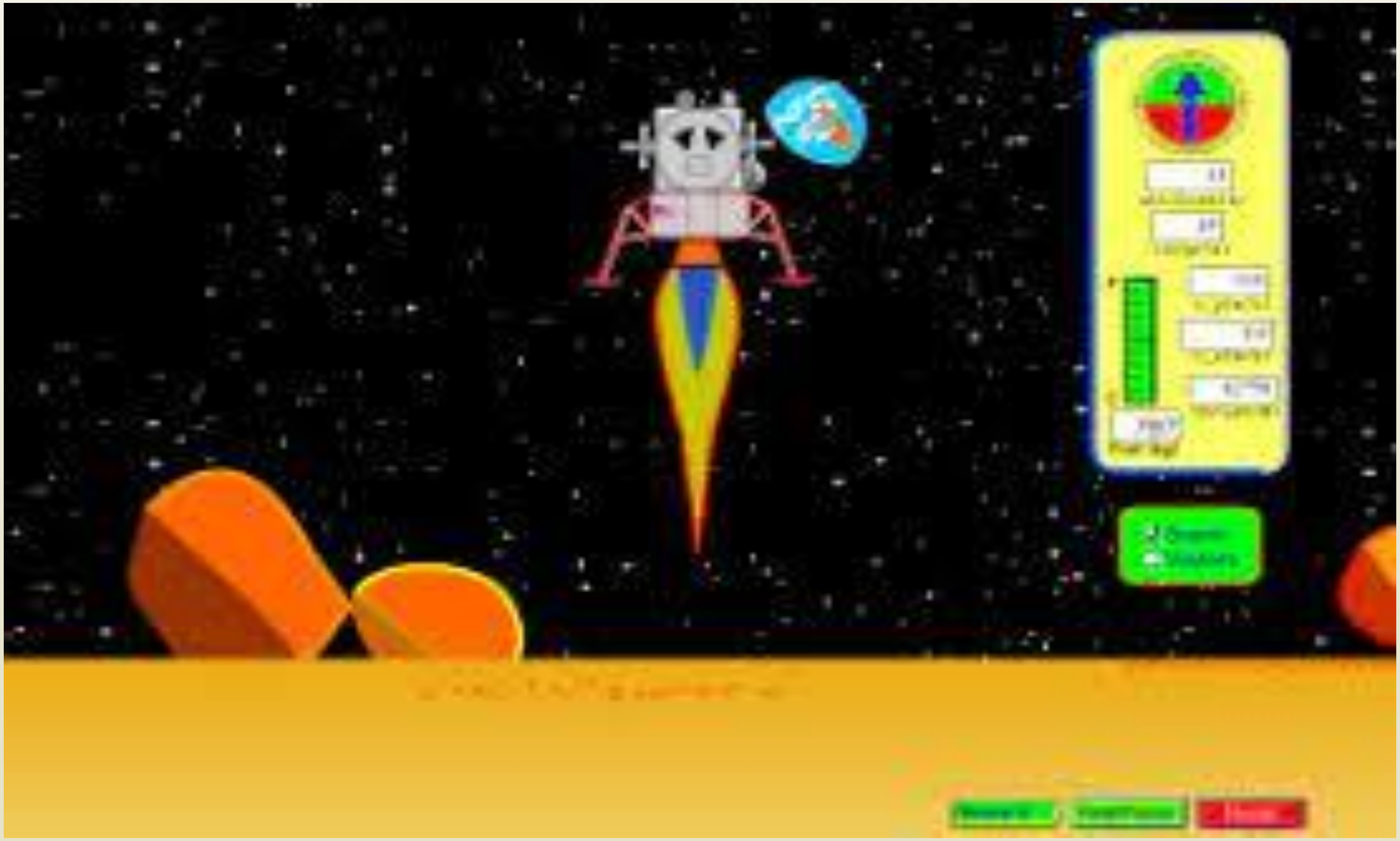
```
Logic:
loop
  read all sensor values
  compute control outputs
  send controls to all actuators
end loop
```

Objective: Structuring embedded control applications

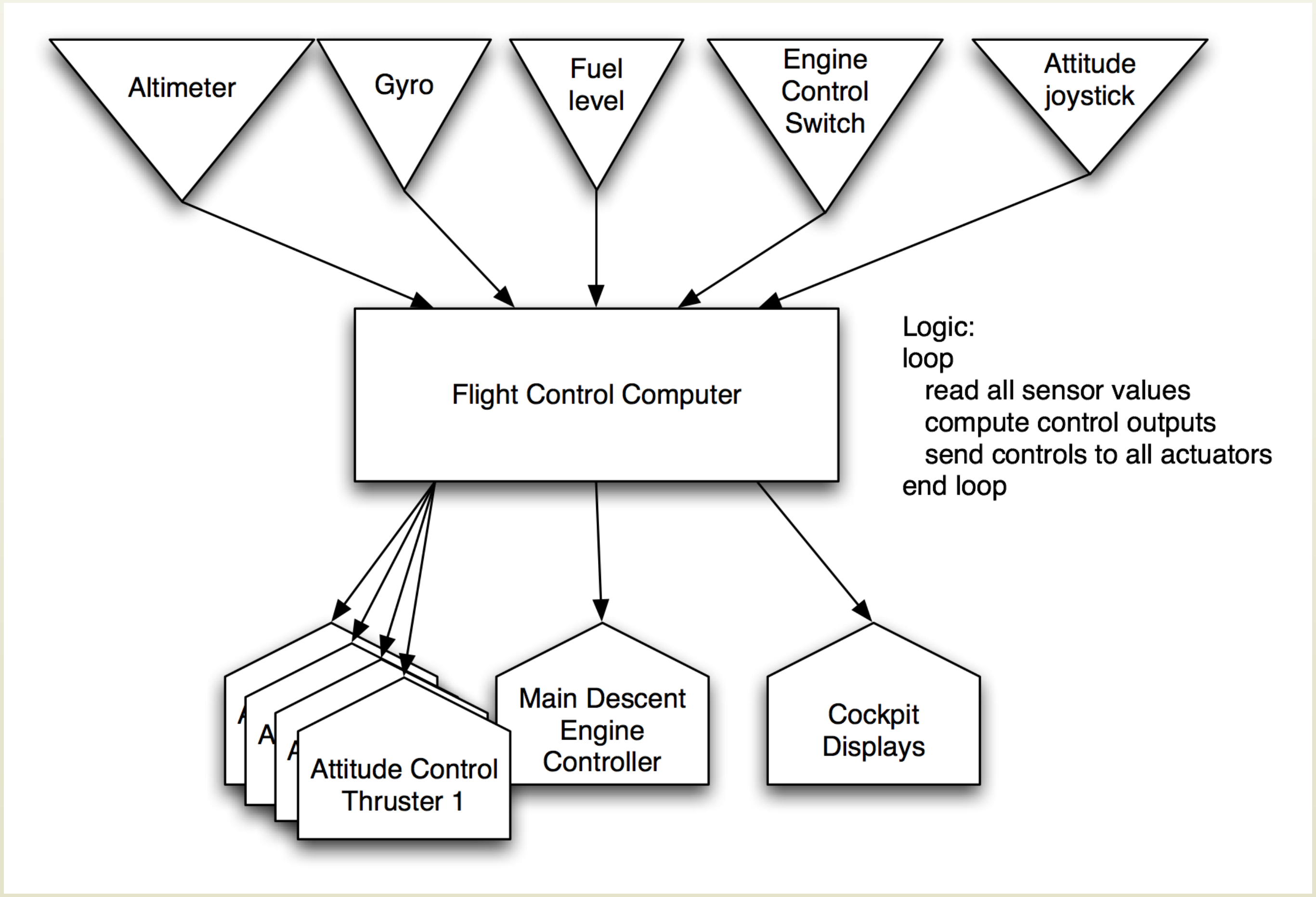
The Lunar Lander: A Long-Running Example

- 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 frames per second (fps): you win (whether there's fuel left or not)
    - "Advanced" version: joystick controls attitude & horizontal motion

The Lunar Lander Game



Sense-Compute-Control LL



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

Architectural Styles

- 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

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
- Recurring organizational patterns & idioms
  - Established, shared understanding of common design forms
  - Mark of mature engineering field
    - Shaw & Garlan
- Abstraction of recurring composition & interaction characteristics in a set of architectures
    - Taylor

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

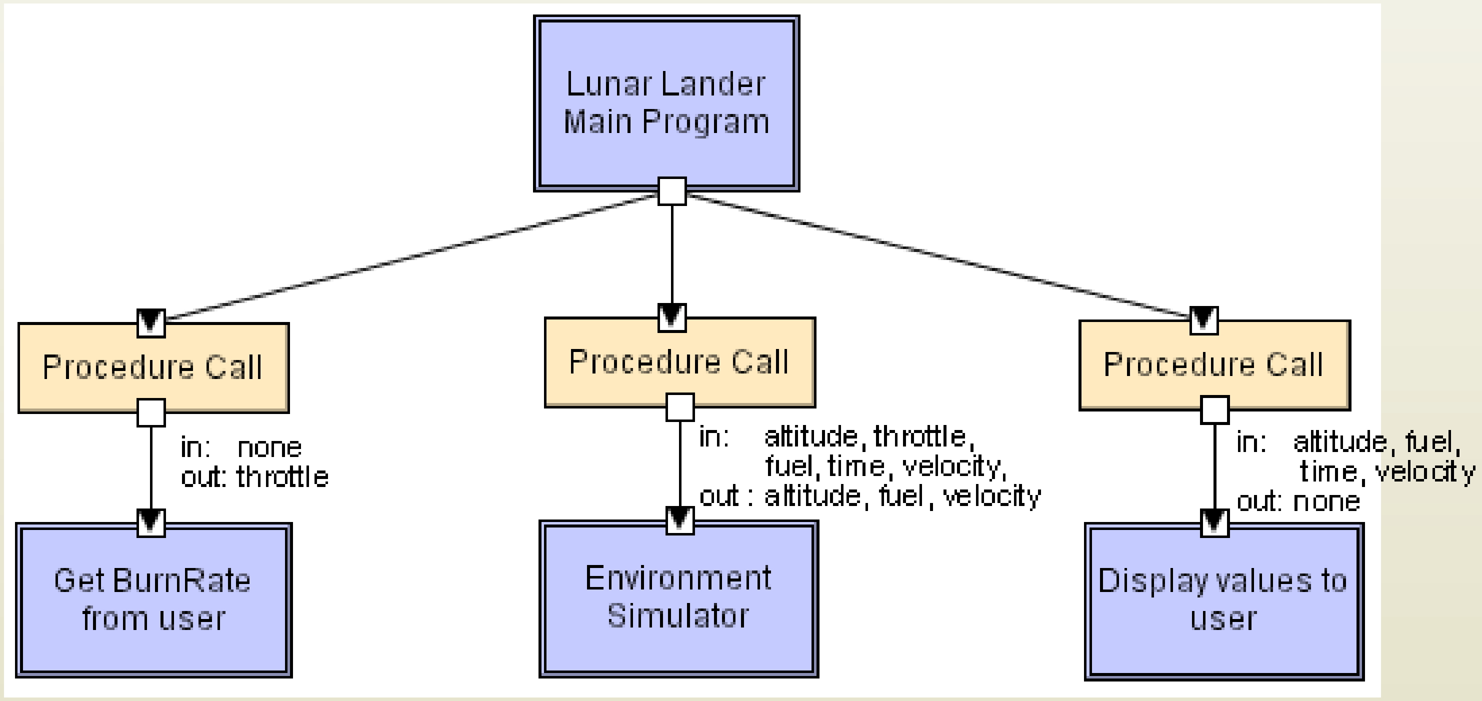
Style Analysis Dimensions

- 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?

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

Main Program and Subroutines LL



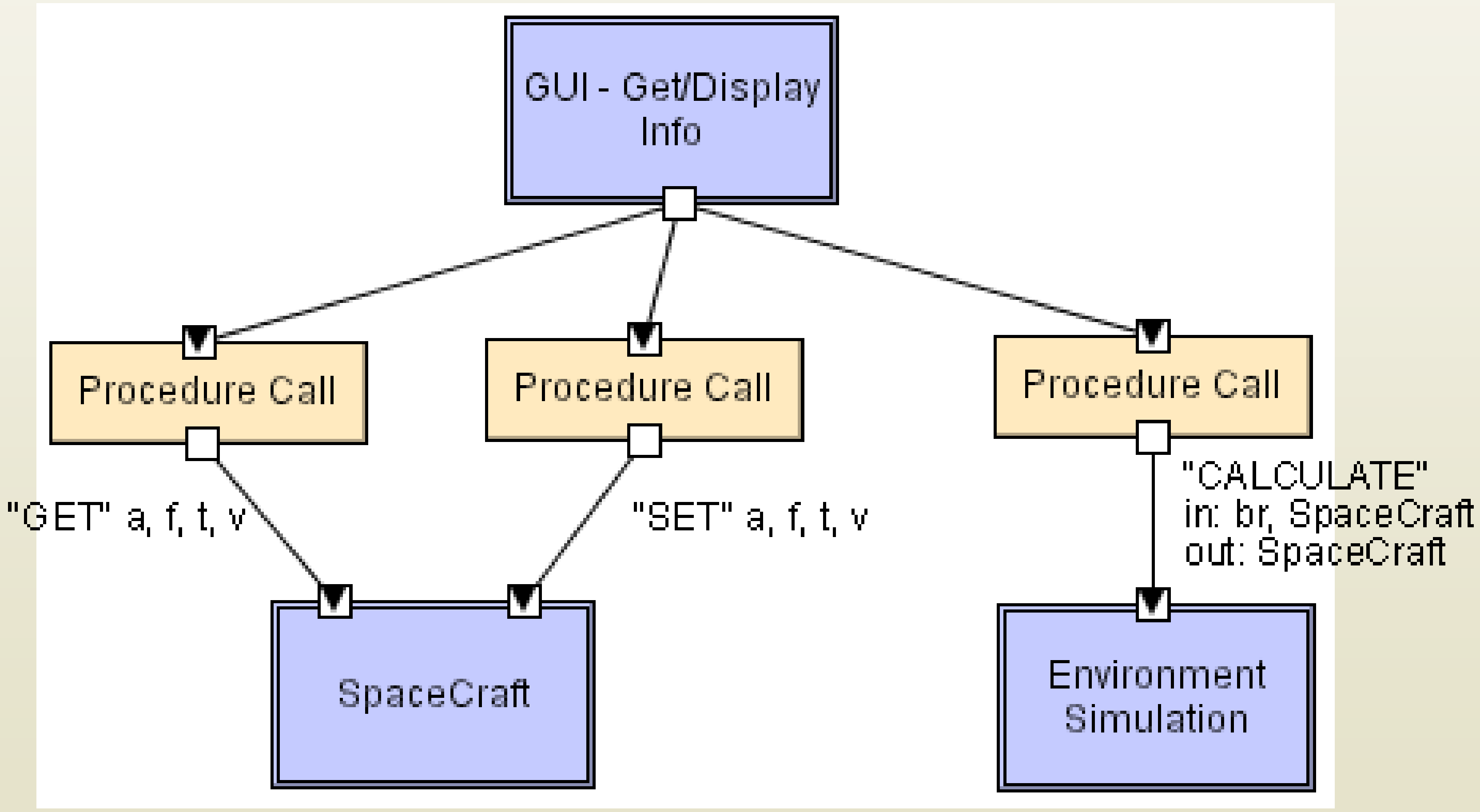
Main Program and Subroutines LL (cont’d)

- The main program displays greetings and instructions, then enters a loop and calls the three subroutines in turn
- The first component obtains the pilot’s throttle-setting input
- The second component serves as the environment simulator, determining how much fuel is left and what the altitude and descent rate are
  - The only flight control law here is the translation of a pilot-specified throttle percentage into the burn rate to control the descent engine
- The third component displays the updated state
- There is no clock, one cycle is a clock tick

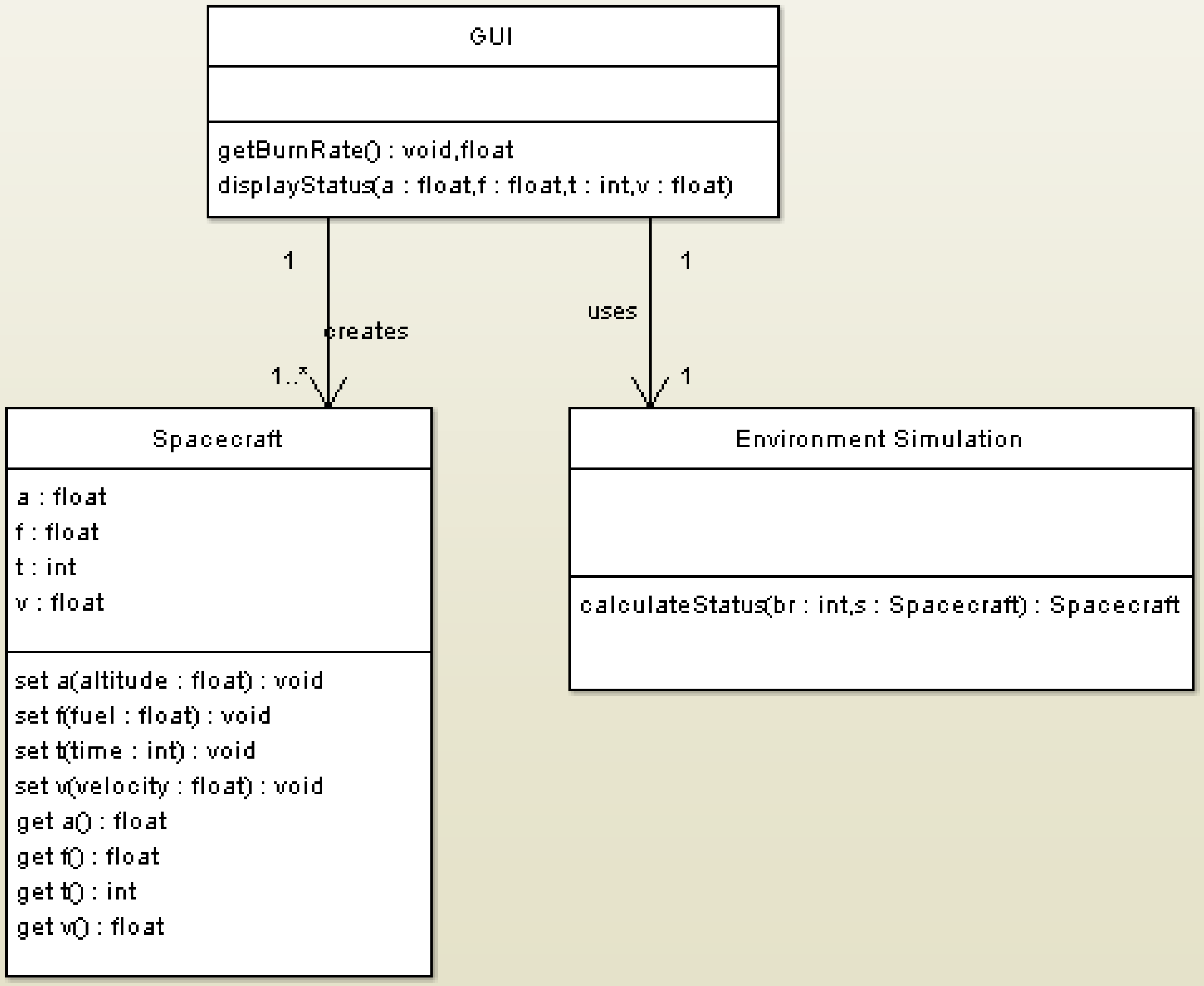
Object-Oriented Style

- 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

Object-Oriented LL (cont'd)

- Here we have three encapsulations:
  - Spacecraft
  - User interface
  - Environment (a physics model that allows calculation of the descent rate)
- In contrast to the MPS model before, a single object handles all (input and output) interactions with the user
  - The functional decomposition in the MPS model resulted in those interactions be performed by separate subroutines

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
- *Virtual machine* style results from fully opaque layers

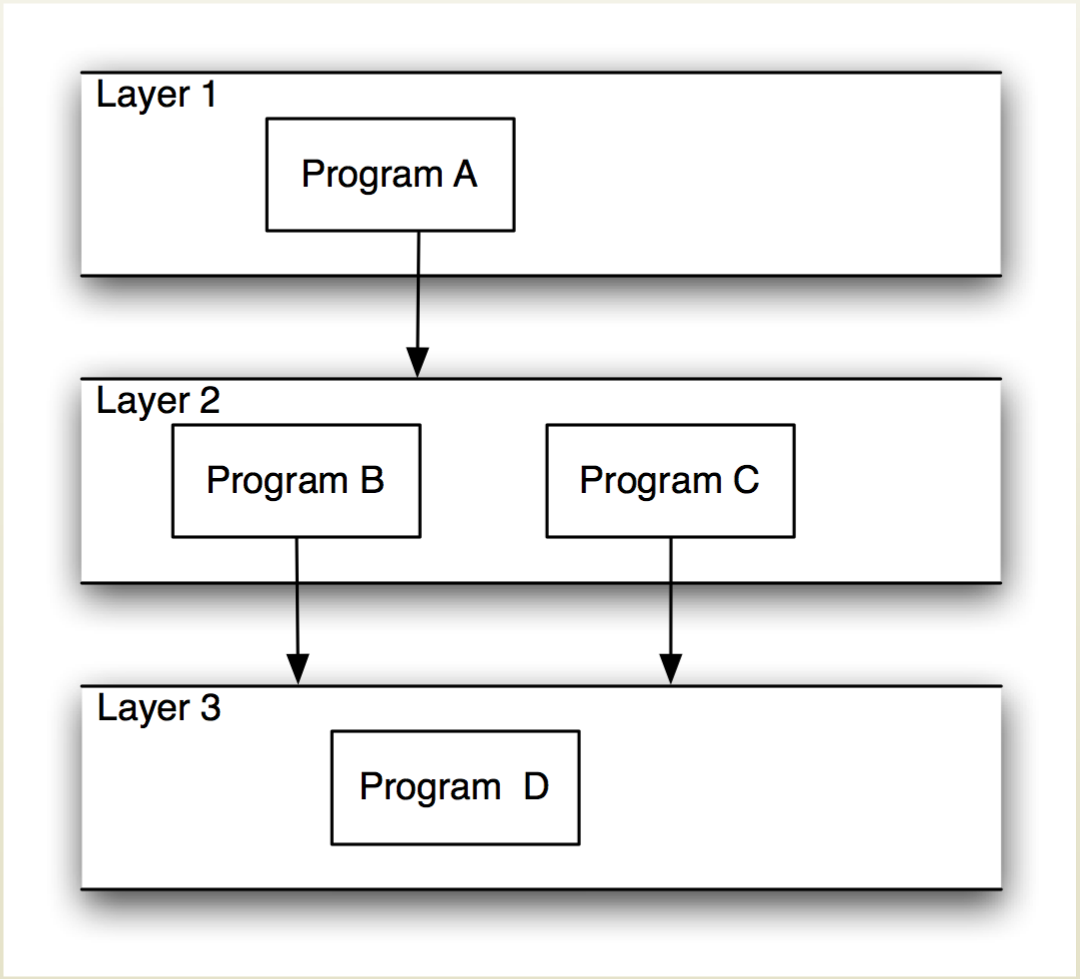
Layered Style (cont’d)

- 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

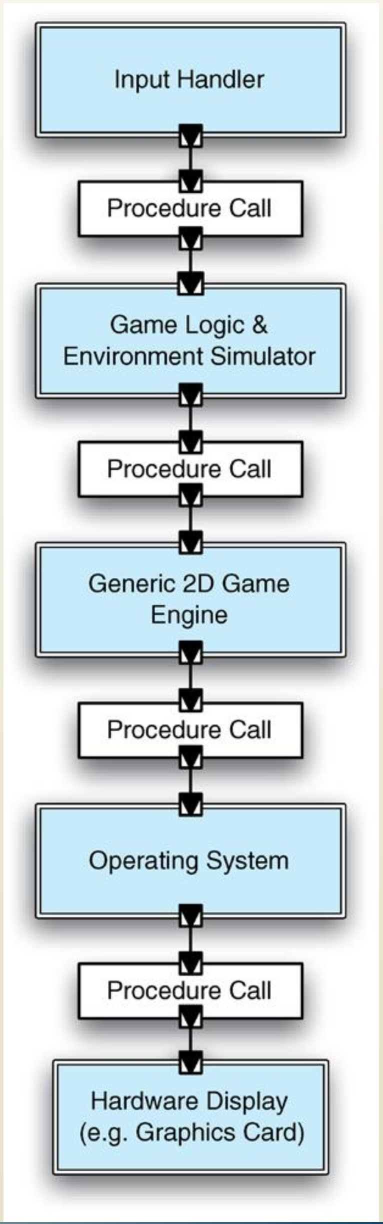
Layered Systems/Virtual Machines



Virtual Machines

- In the previous figure, program `A` in layer 1 can access the services offered by layer 2; but it does not need to know that these services may in fact be implemented by programs `B` and `C`
- At the same time, programs in one layer can access only those services provided by the layer immediately below their layer
  - So, `A` cannot access the services provided by layer 3
- Operating systems is a typical example of a layered style
  - Layer 1 has user applications
  - Layer 2 has directory and file management services
  - Layer 3 has device drivers

Layered / Virtual Machine LL



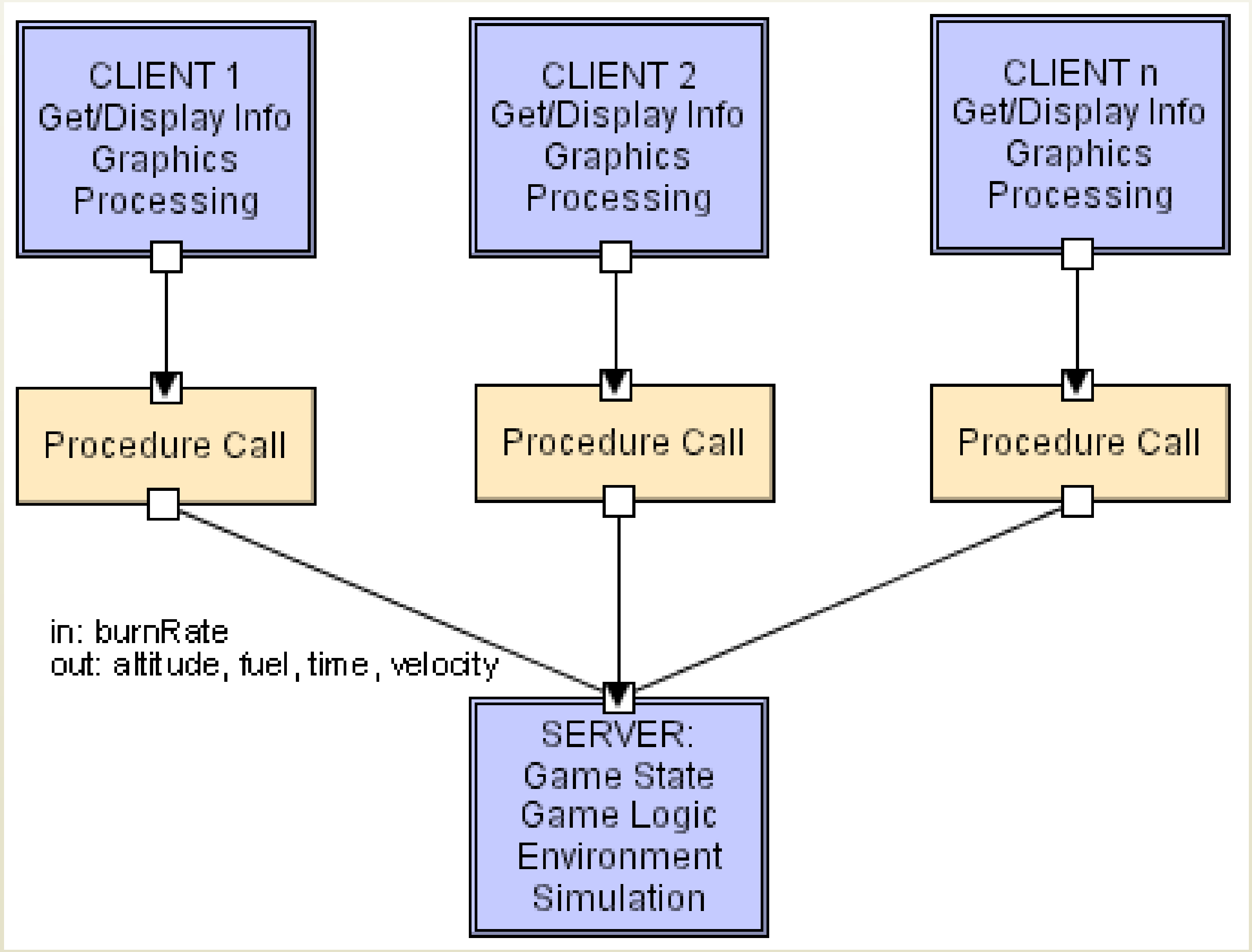
Layered / Virtual Machine LL (cont'd)

- The top layer handles input received from the user through the keyboard and calls the second layer to service those inputs
- The second layer includes the game logic and the environment simulator; it updates the game state (based on the user's input from the top layer) and calls the third layer to begin the process of displaying the updated game state to the user
- The third layer is a generic, two-dimensional game engine, that supports a wide variety of similar games
- The fourth level is the OS that supports platform specific UI, such as window management

Client-Server Style

- Effectively, a two-layer virtual machine with network connections
- The server is the virtual machine below the clients
- Components are clients and servers
- Servers do not know number or identities of clients
- Clients know server's identity
- Clients are mutually independent
- Clients can be "thin" or "thick", reflecting whether they include any significant processing beyond UI functions
- Connectors are RPC-based network interaction protocols, used by the clients to access the server

Client-Server LL



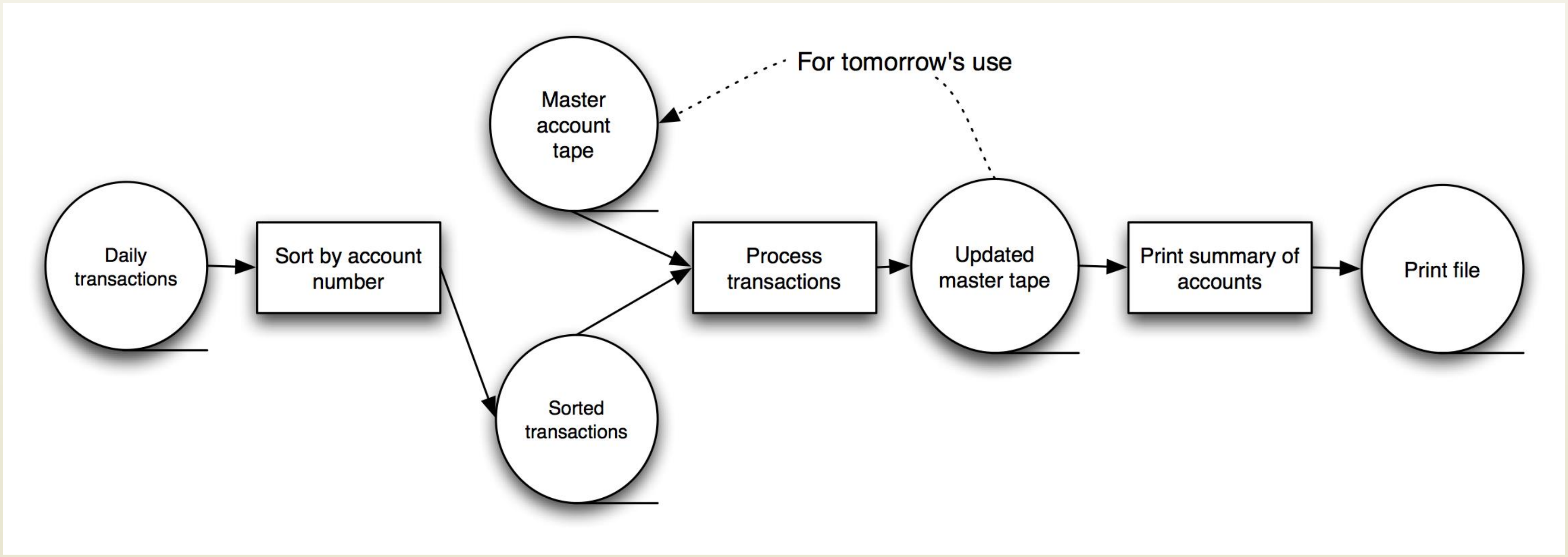
Client-Server LL (cont’d)

- In the shown system, three players simultaneously and independently play the game
- All game state, game logic, and environment simulation is performed on the server
- The clients perform the user interface functions
- The connectors are RPC with a ”distributor” which identifies network interaction paths and routes communication along those paths

Data-Flow Styles

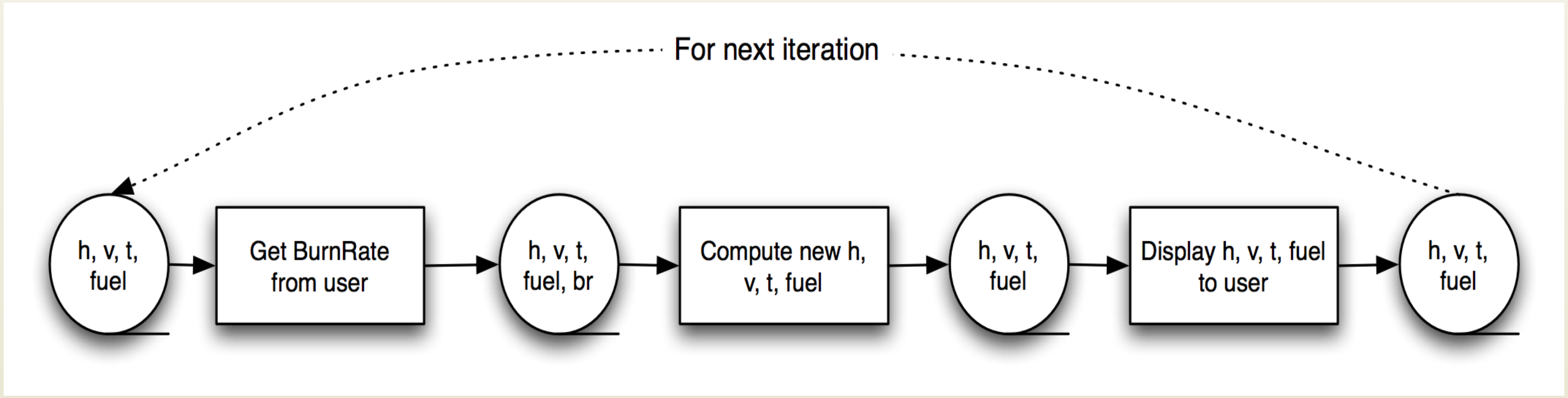
- Batch Sequential
  - Separate programs are executed in order; data is passed as an aggregate from one program to the next
  - Connectors: "The human hand" carrying tapes between the programs
  - Data Elements: Explicit, aggregate elements passed from one component to the next upon completion of the producing program's execution
- One of the oldest styles, when the limitations of computing equipment required the use of magnetic tapes

Batch-Sequential: A Financial Application



A master tape holding a bank’s accounts is updated with the day’s transactions held in a daily transactions tape

Batch-Sequential LL



- Each functional processing step is done by a separate program
- An updated version of the game is then handed off to the next program
- After the final step of displaying the game state is performed, the produced tape is carried back to the first program
- Not a highly interactive, real-time game!

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`

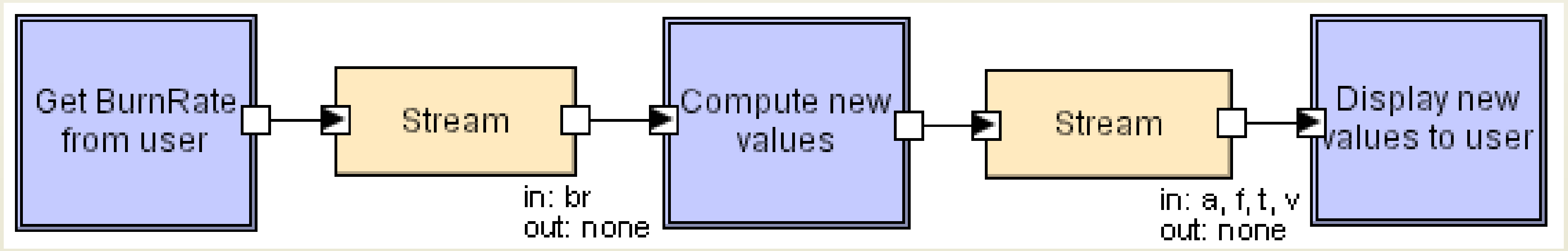
Pipe and Filter (cont'd)

- 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



Pipe and Filter LL (cont'd)

- `GetBurnRate` runs continuously on its own, prompting the user for a new burn rate
  - When a value is generated, it is sent off through the stream connector to the second filter
- The second filter also loops continuously on its own, updating time, serving as the environment simulator and calculating the descent rate of the spacecraft
- The third filter likewise loops, updating the display
- In this design the "compute new values" determines how much time has passed; could alternatively do that in the `GetBurnRate` filter, which would change semantics a bit

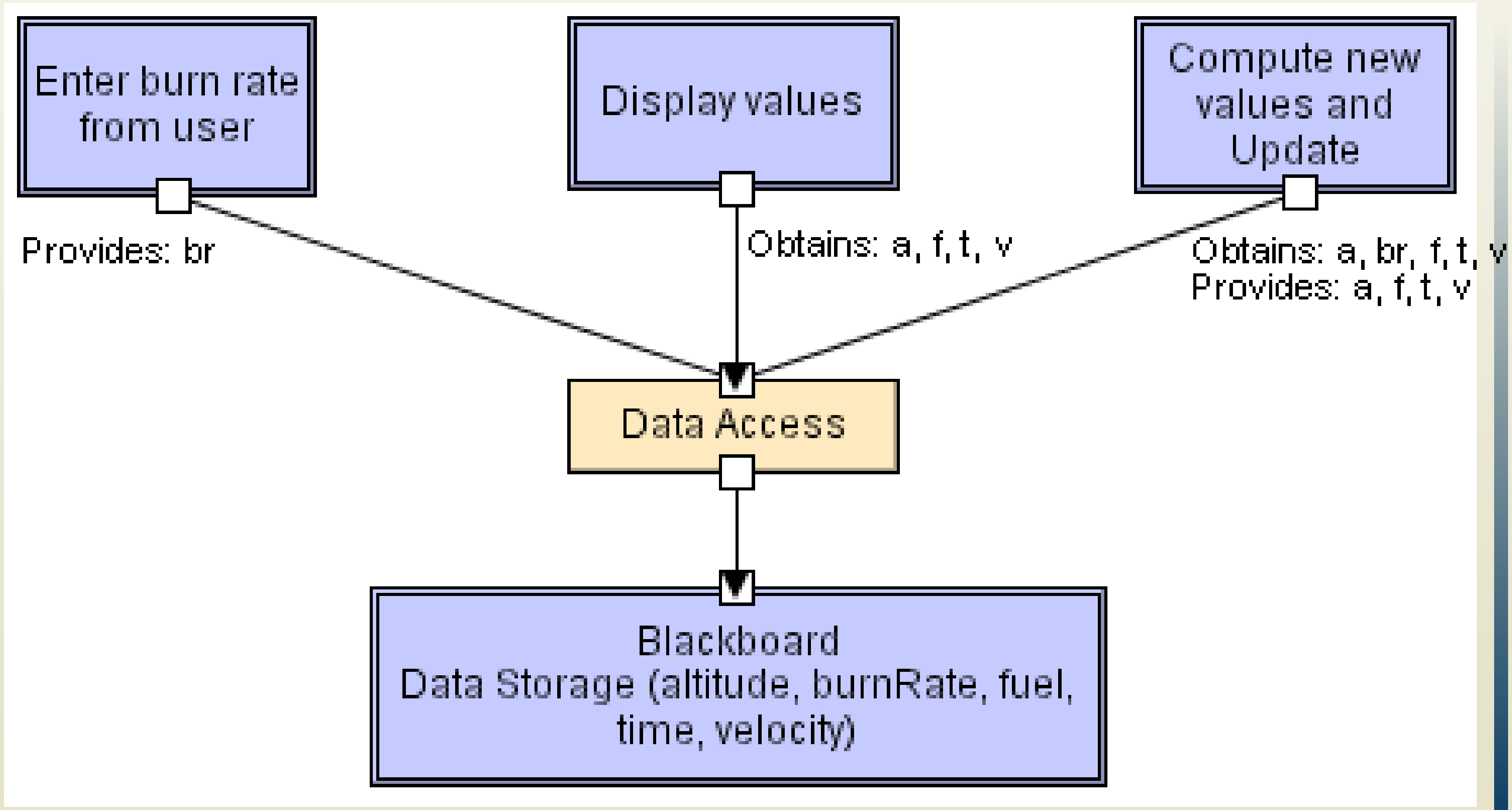
Shared Memory Styles

- The essence of shared memory (or shared state) styles is that multiple components have access to the same data store and communicate via that data store
- This corresponds to the (bad!) practice of using global data in procedural languages
- However, in the case of shared state styles, the center of the design attention is on these structured shared repositories, which are well-ordered and carefully managed

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)

Blackboard LL (cont'd)

- A single connector (`Data Access`) regulates access from the various components in manipulating the information on the blackboard
- The blackboard maintains the game state
- The first component updates the descent engine burn rate based on user input
- The second component displays to the user the current state of the spacecraft and any other aspect of the game state
- The third component updates the game state based upon a time and physics model

Rule-Based / Expert System Style

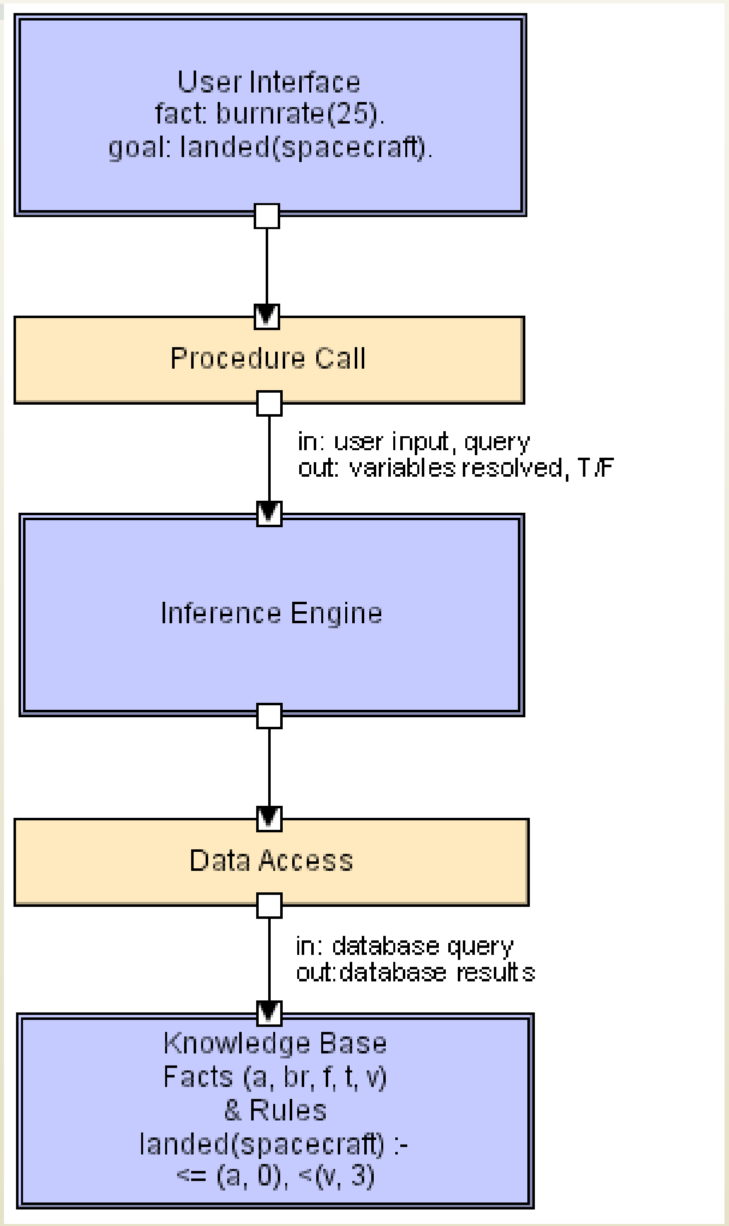
- Inference engine parses user input and determines whether it is a fact/rule or a query
  - If it is a fact/rule, it adds this entry to the knowledge base
  - Otherwise, it queries the knowledge base for applicable rules and attempts to resolve the query

Rule-Based / Expert System 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

- In this style, we maintain a set of consistent facts about the spacecraft
- The user enters the value of burn rate as a fact:
  - `burnrate(25)`
- To see what the status of the spacecraft is, the user switches to the goal mode and asks:
  - `landed(spacecraft)`
  - The inference engine queries the database and returns true or false

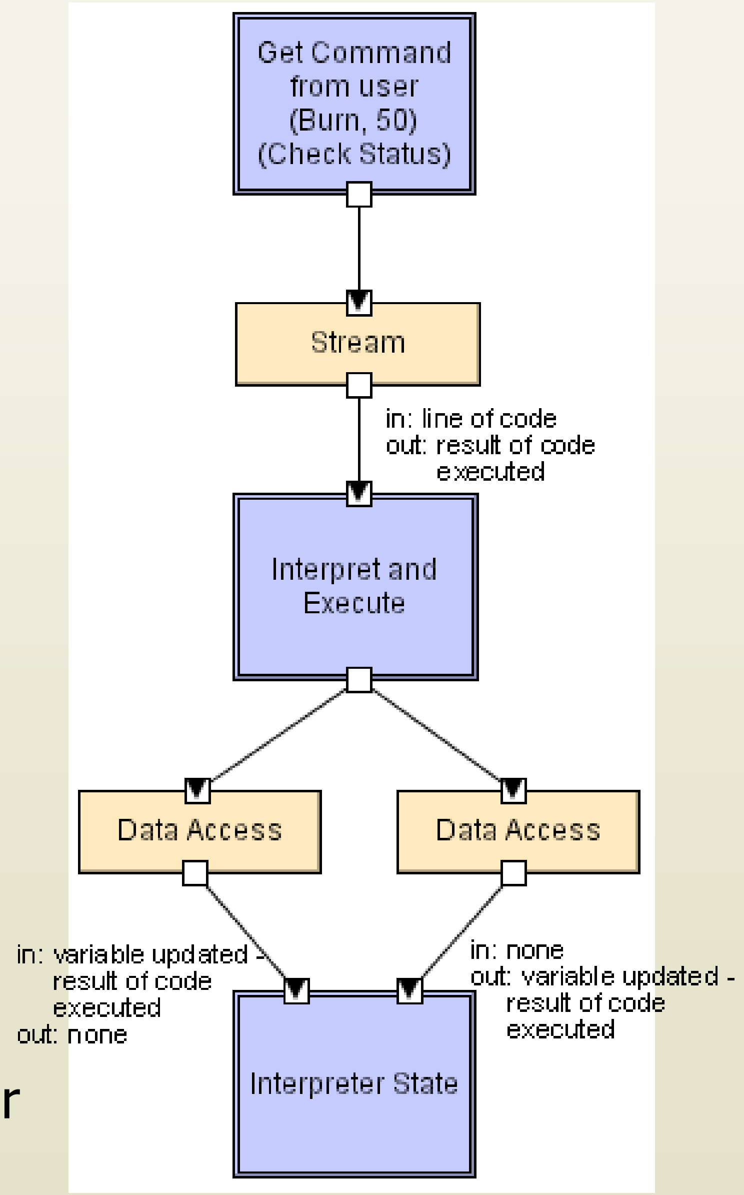


Interpreter Style

- 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
- Used in the programming languages Lisp and Scheme

Interpreter LL

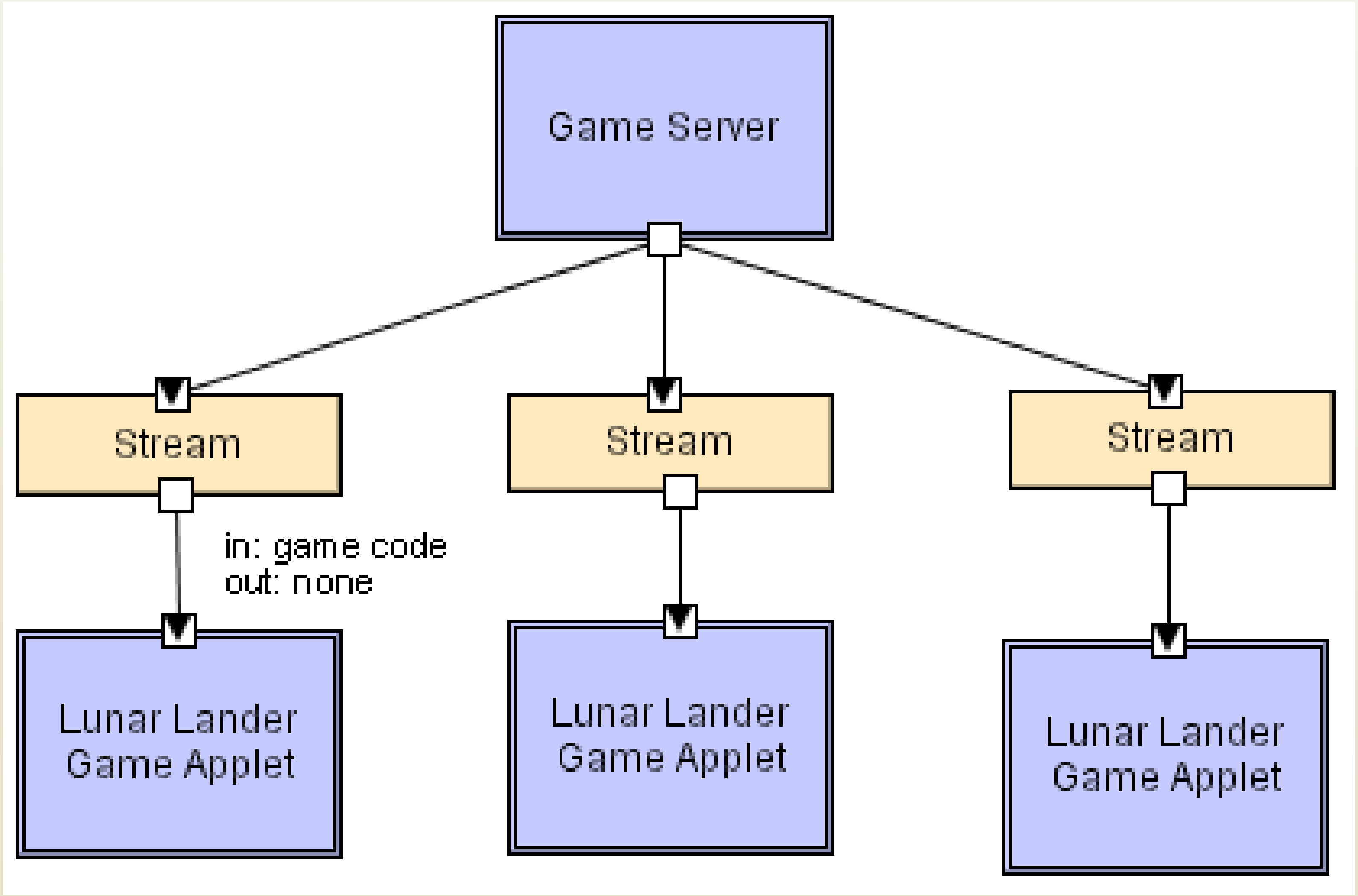
- For each command entered, the interpreter engine processes the code and updates the interpreter state as necessary
- The commands are specific directives on how the spacecraft should be manipulated
- The interpreter may return values to the user, depending on the command
- Each time the user enters a `BurnRate` command, the time is incremented
- When the user enters a `CheckStatus` command, the user receives info about altitude, fuel, etc.


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

- It enables code to be transmitted to a remote host for interpretation
- This may be due to a lack of local computing power or resources, or due to large data sets remotely located
- Mobile code can be classified as:
  - Code on demand, when the initiator has resources and state but downloads code from another site to be executed locally
  - Remote evaluation, when the initiator has the code but lacks the resources to execute the code
  - Mobile agent, when the initiator has the code and the state but resources are located elsewhere

Mobile Code LL



Mobile Code LL (cont'd)

- One client Web browser downloads code-on-demand in the form of a LL game applet via HTTP
- A second browser loads a JavaScript LL
- A third browser uses some other form that is not detailed
- All the game logic moves to the client machines, freeing the server's computing resources
- Each client machine maintains the game state independently of the other clients

Implicit Invocation Style

- Unlike the previously discussed styles, the implicit invocation styles are characterized by calls that are invoked indirectly and implicitly, as a response to a notification or an event
- This leads to ease of adaptation and enhanced scalability between very loosely coupled components

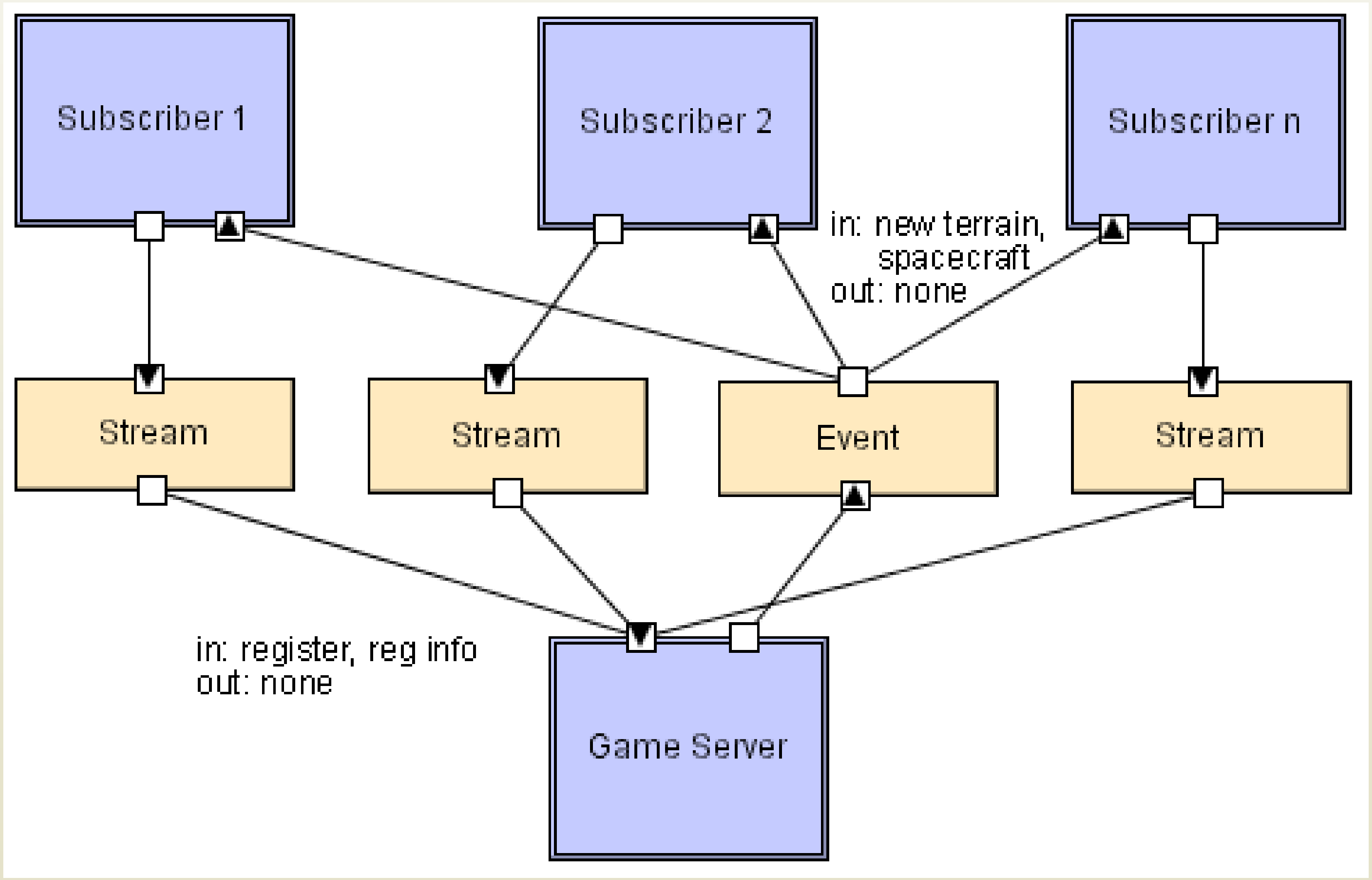
Publish-Subscribe

- The publisher periodically creates information and the subscriber obtains a copy of this information or at least is informed of its availability
- In a simple publish-subscribe style, the publisher maintains a list of subscribers
  - A procedure call is issued to the subscribers when new information is available
  - A subscriber may register his interest with a publisher by providing a procedure interface to be used by the latter (a “call-back”)

Publish-Subscribe (cont'd)

- For large-scale, network-based applications, subscriptions involve the use of network protocols and intermediate proxies and caches (in the same way that physical newspapers are distributed by carriers and not by a direct communication between the newspaper company and the readers)
- A typical example of this style is an online job posting service
  - Hiring managers post or publish job openings
  - Job seekers subscribe to receive notifications of new job postings

Publish-Subscribe LL



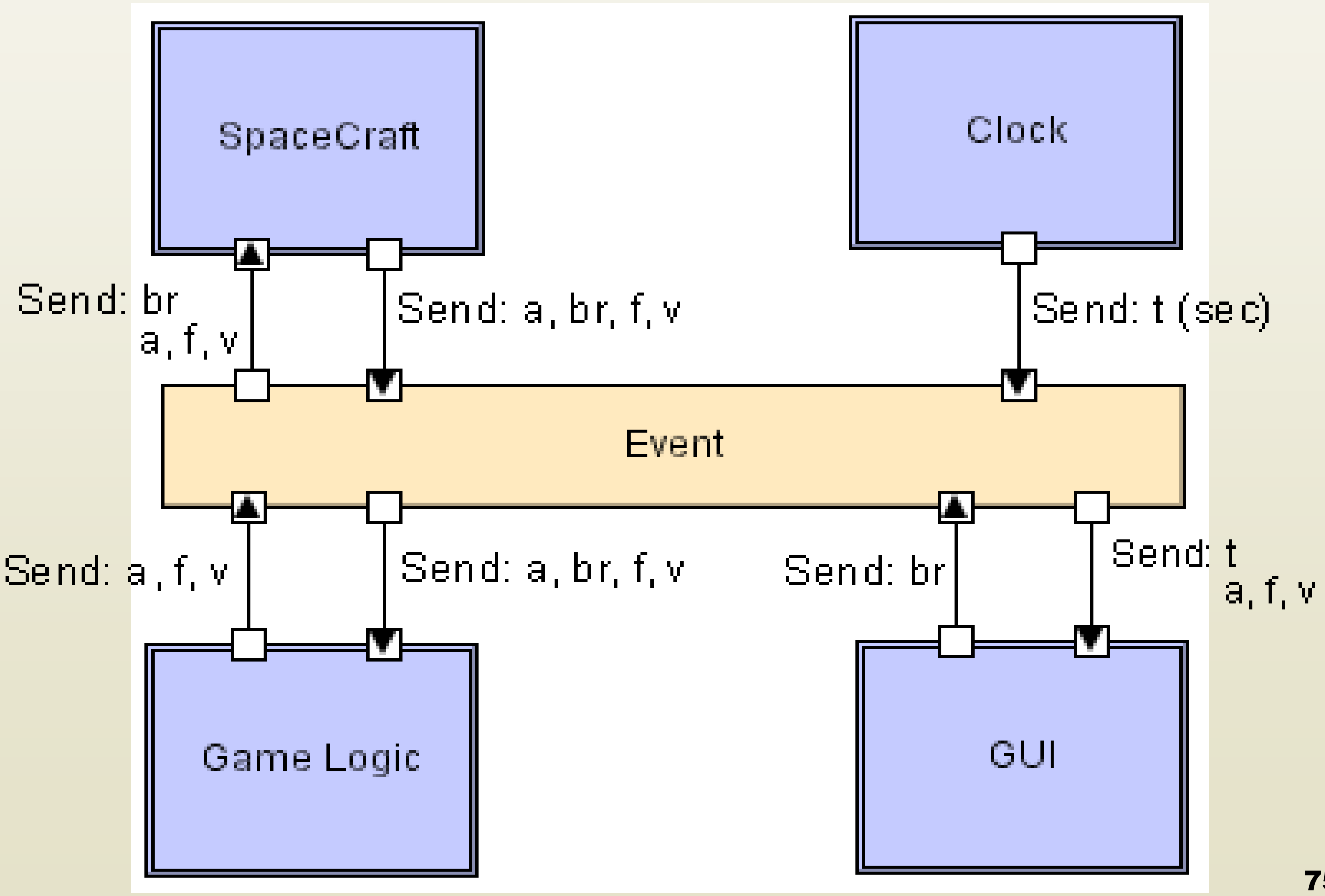
Publish-Subscribe LL (cont'd)

- The LL software is deployed to various network hosts
- Players (subscribers) register their hosts to a game server
  - The game server publishes information, such as a new Lunar terrain data, new spacecraft, and the locations of all the registered spacecraft playing
- Once registered, the players receive notifications for the information they have registered
  - The notification may contain the information (appropriate for a multi-player), or
  - A separate download link is provided for the information (e.g., for game updates)

Event-Based Style

- Independent components communicate solely by sending events through event-bus connectors
- Components may react in response to receipt of an event or may ignore it
- For efficiency, events are not distributed to all components but only to those that have expressed an interest in them
  - With this optimization the style becomes similar to publish-subscribe; however, in the the event-based style there is no distinction between publishers and subscribers and all components can both emit and receive events

Event-based LL



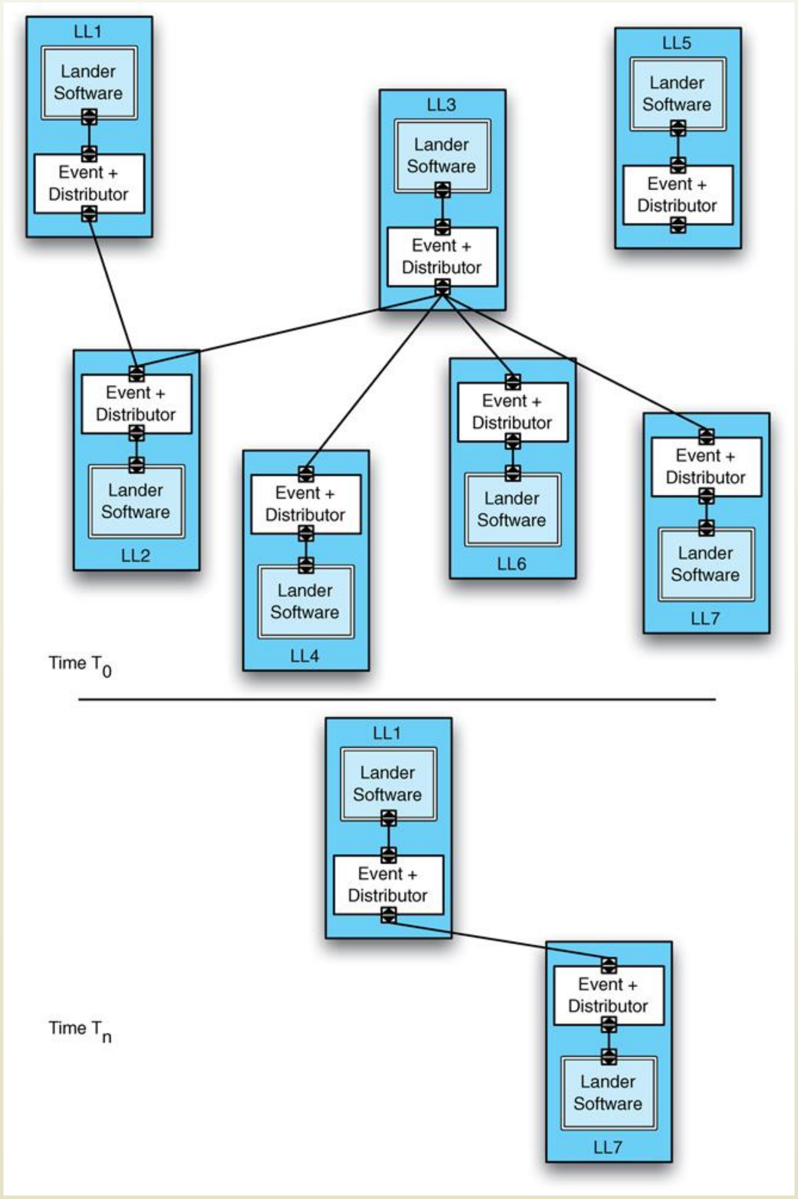
Event-based LL (cont'd)

- The clock component drives the game; every fraction of the second, it sends out a tick notification
- The spacecraft component, upon receiving a predefined number of notifications from the clock, recalculates the altitude, fuel level and velocity, and emits those values to the event bus
- The GUI component receives the events from the spacecraft component and updates the display; it also receives user settings which it emits to the event bus for the spacecraft component to receive them and update its internal state
- The game logic component, based on the events it receives from the spacecraft and the clock components, decides if the game is over

Peer-to-Peer Style

- It consists of a network of loosely coupled autonomous peers, each peer acting both as a client and a server
- Peers communicate using a network protocol, such as in the Napster and Gnutella file sharing applications
- Both information and control is fully decentralized
- For a peer to find another peer, it sends the requests to the peers it is connected with and these in return propagate it to other peers
- For efficiency, often some peers play special roles, either for locating other peers or for providing directories locating information

Peer-to-Peer LL



Peer-to-Peer LL (cont'd)

- A group of Lunar Lander spacecraft are on their way to land in different parts of the moon
- `LL1` wants to find out if another spacecraft has already landed at the landing spot it has chosen, and does the following to obtain this information:
  - It queries for available spacecraft within communication range
  - Only `LL2` responds but it doesn't have the information, so it passes the request to `LL3`
  - `LL3` doesn't also have the information, so it passes the request to `LL4`, `LL6` and `LL7`
  - `LL7` responds to `LL3` that it has the information, sends it, and then `LL3` passes it on to `LL1`
- At a later time (Tn), `LL7` comes within communication range and `LL1` can communicate directly with it

Heterogeneous Styles

- More complex styles created through composition of simpler styles
- REST (from the first lecture)
- 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
- Grew out of a desire to obtain the benefits of the MVC pattern in a distributed heterogeneous platform setting
- The intention was for C2 to be used for GUI applications but it found applicability in a much wider variety of applications

C2 Benefits

- **Substrate independence**
  - Ease in modifying the application to work with new platforms
- **Accommodating heterogeneity**
  - Components can be written in different languages and run on multiple, varying hardware platforms, communicating across a network
- **Support for product lines**
  - Ease of substituting one component with another to achieve similar but different applications

C2 Benefits (cont’d)

- **Ability to design in the MVC style**
  - But with very strong separation between the model and the UI elements
- **Natural support for concurrent components**
  - Whether running on a shared processor or multiple machines
- **Support for network-distributed applications**
  - Communication protocol details are kept out of the components and confined to connectors

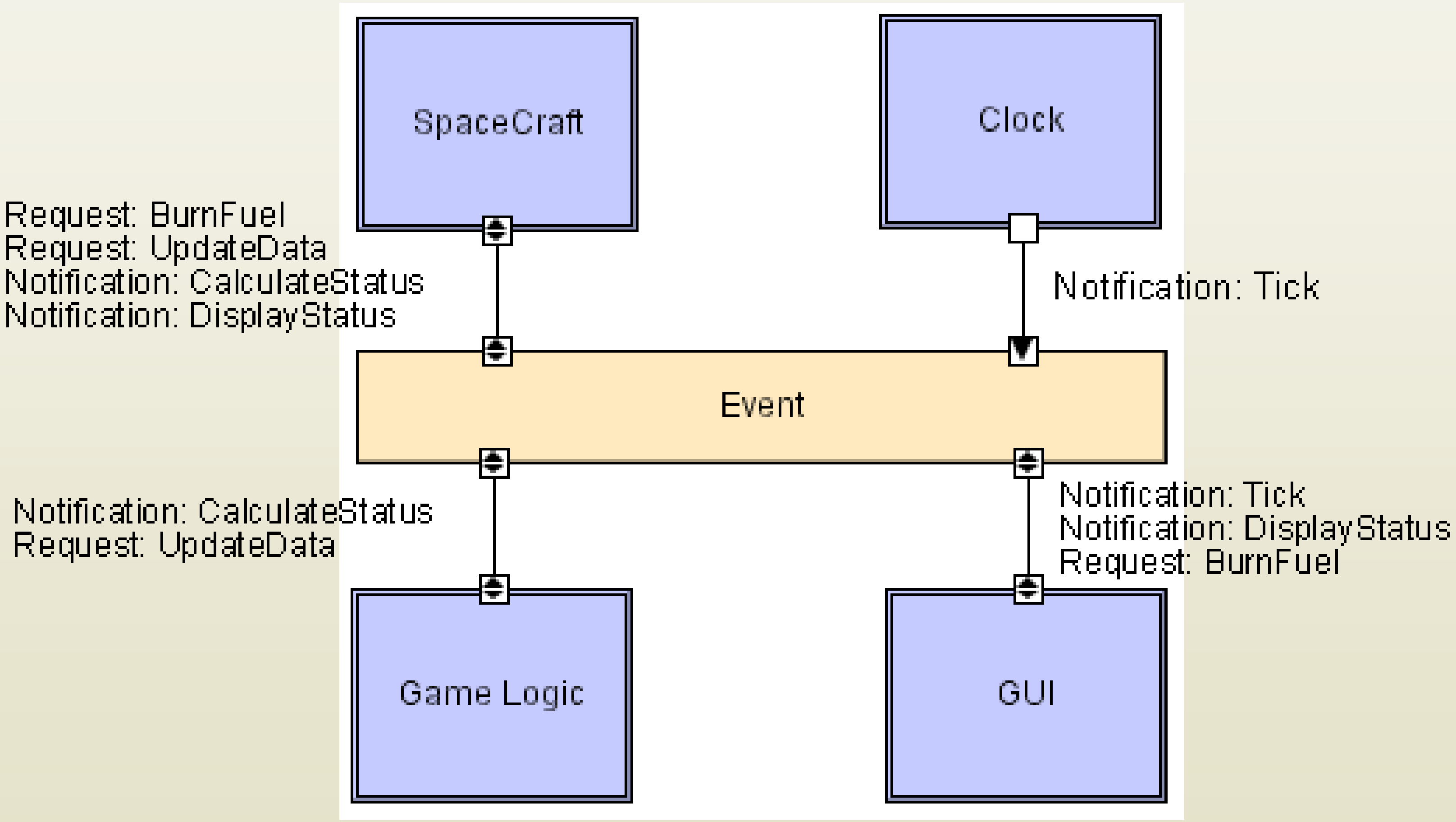
C2 Constraints

- **Topology**
  - Layers of components and connectors, with a defined “top” and “bottom”, wherein notifications flow downwards and requests upwards
  - The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector
- **Message-based communication**
  - All communication between components is achieved through exchanging messages, which are classified either as requests (for services) or notifications

C2 Constraints (cont’d)

- **Message flow and substrate independence**
  - Requests may only flow upwards and notifications may only flow downwards in an architecture; “substrate independence” means that a component can only be aware of components above it and is completely unaware of components below it
- **Interfaces**
  - Each component has a top and bottom domain; the top domain specifies the set of notifications to which a component may react and the bottom domain specifies the set of notifications that this component emits

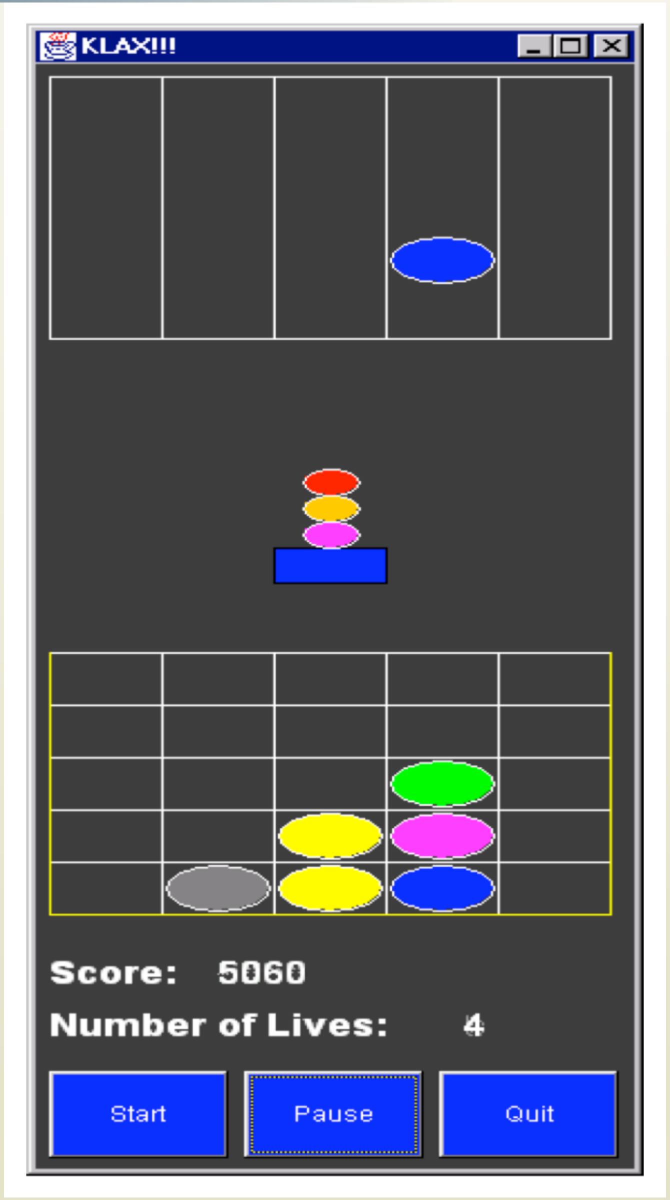
C2 LL


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

C2 LL (cont'd)

- With respect to the use of events, the C2 architecture operates largely like the event-based style one
- Here, however, both the spacecraft and clock components are guaranteed by the architecture to be completely unaware of the presence of the game logic and GUI components

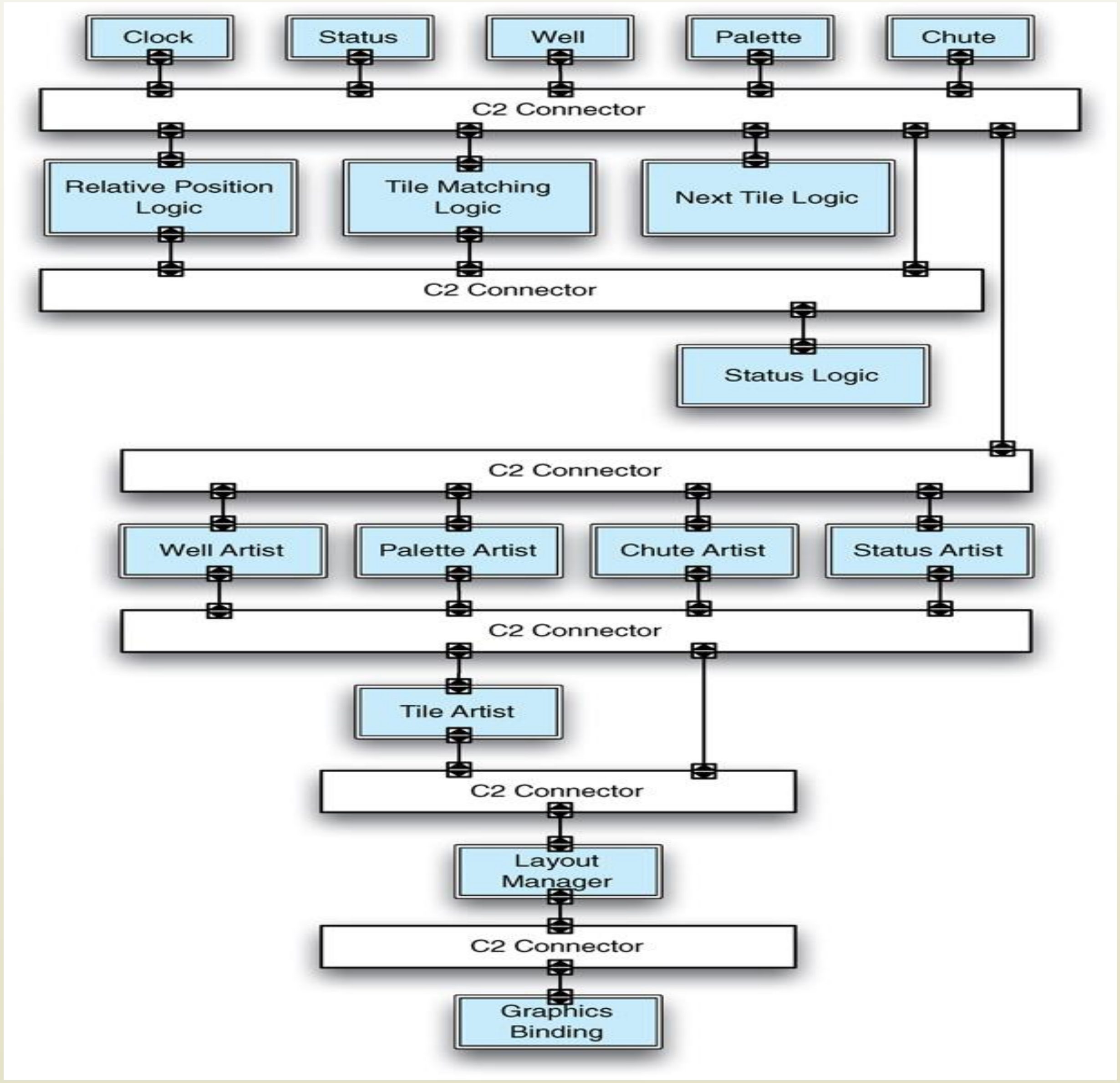
KLAX



KLAX(cont'd)

- Colored tiles fall from the chutes at the top of the user's screen
- A palette is used to move horizontally across the screen; it can catch the tiles as they fall from the chutes
- By inverting the palette, the tiles can be dropped into the wells below
- Matching three or more tiles of the same color, horizontally, vertically, or diagonally across the wells, points are scored and the tiles are removed
- Lives are lost if the tiles are not caught by the palette or the wells overflow

KLAX in C2


Clock
Status
Well
Palette
Chute
C2 Connector
Relative Position Logic
Tile Matching Logic
Next Tile Logic
C2 Connector
Status Logic
C2 Connector
Well Artist
Palette Artist
Chute Artist
Status Artist
C2 Connector
Tile Artist
C2 Connector
Layout Manager
C2 Connector
Graphics Binding

KLAX in C2(cont'd)

- Components are divided into three logical groups
  - At the top are the components that encapsulate the game's state and the clock; they are placed there because the game state is vital for the functioning of the other two groups of components
  - At the next level are the game logic components; they request changes of game state and interpret game state change notifications
  - Then are the artist components, which receive notifications for game state change and accordingly update their depictions

KLAX in C2(cont'd)

- The benefits of the C2 architecture are:
  - Easy creation of related but different games; by substituting three components, namely `TileArtist`, `TileMatchingLogic` and `NextTileLogic` with similar ones dealing with letters instead of tiles, the game becomes one where letters fall down the chutes and the game logic is based upon spelling words in the wells
  - Other additions or substitutions of components result in network-based multiplier games
  - Also, the application can run over network boundaries and take advantage of concurrency

Distributed Objects

- The simple object-oriented style is augmented with the client-server style to provide the notion of distributed objects
- Objects are instantiated on different hosts, each of which exposes a public interface
- The objects can be anything, from data structures to million-line legacy systems
- The interfaces are special in that all the parameters and return values must be *serializable* so they can go over the network
- The default mode of interaction between objects is synchronous procedure call, although asynchronous extensions are found, such as CORBA

CORBA

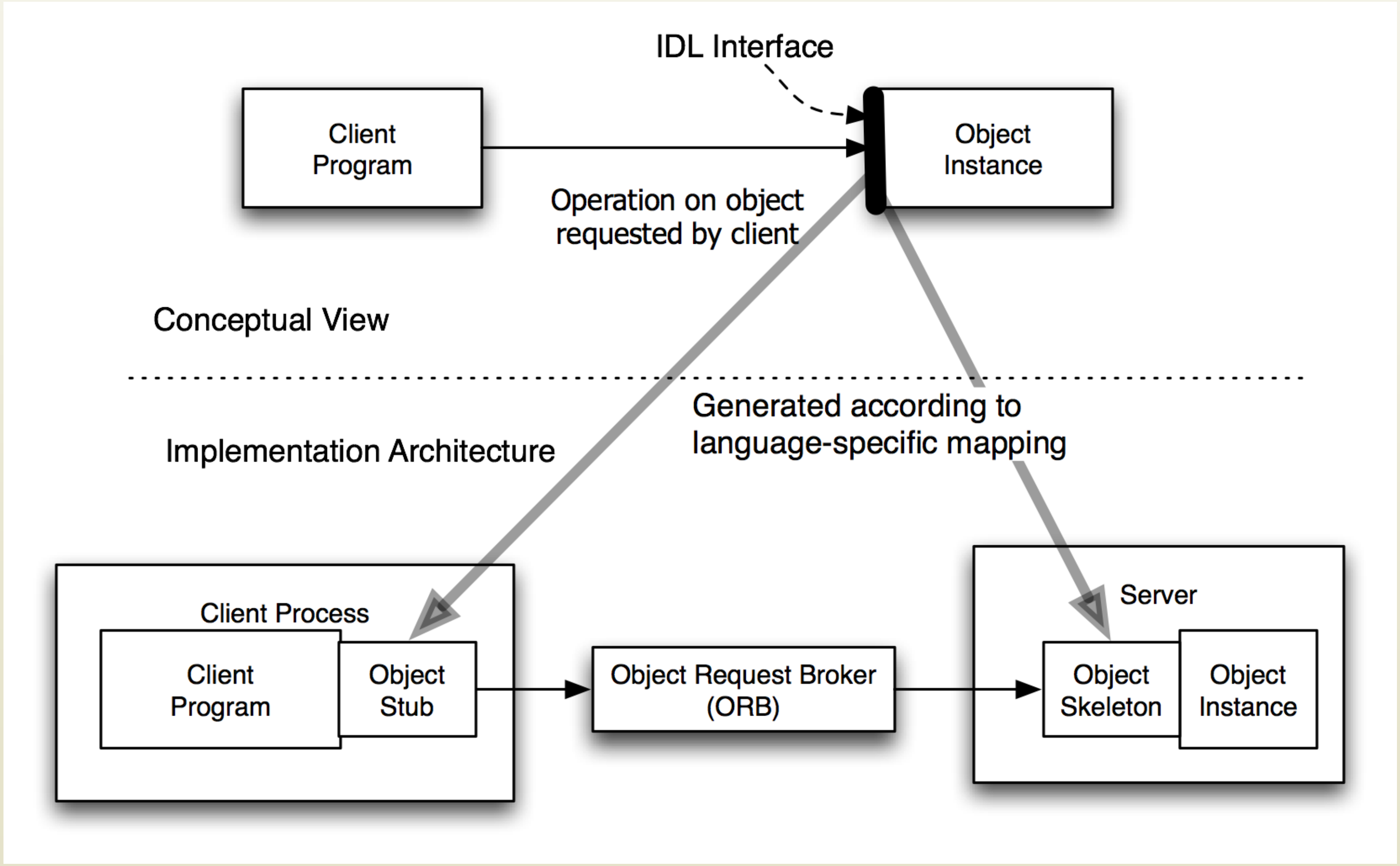
- A standard for implementing middleware that supports applications composed of distributed objects
- An application is broken up into objects, which are effectively software components that expose one or more provided interfaces
- The interfaces are specified in IDL (Interface Definition Language), a notation which is independent of any particular programming language or platform
- IDL closely resembles the way class interfaces are specified in an object-oriented programming language
- All access to an object occurs through calls to one of its IDL-specified interfaces, which are strongly typed (calls and parameters are type-checked at compile time)

CORBA (cont’d)

- An IDL description of the interface for the data store component of a LL system might look like this:

```
interface IDataStore
  double getAltitude()
  void setAltitude(in double newAltitude);
  double getBurRate();
  void setBurnRate(in double newBurnRate);
  void getStatus(out double altitude,
                 out double burnRate,
                 out double velocity,
                 out double fuel,
                 out double time);
```

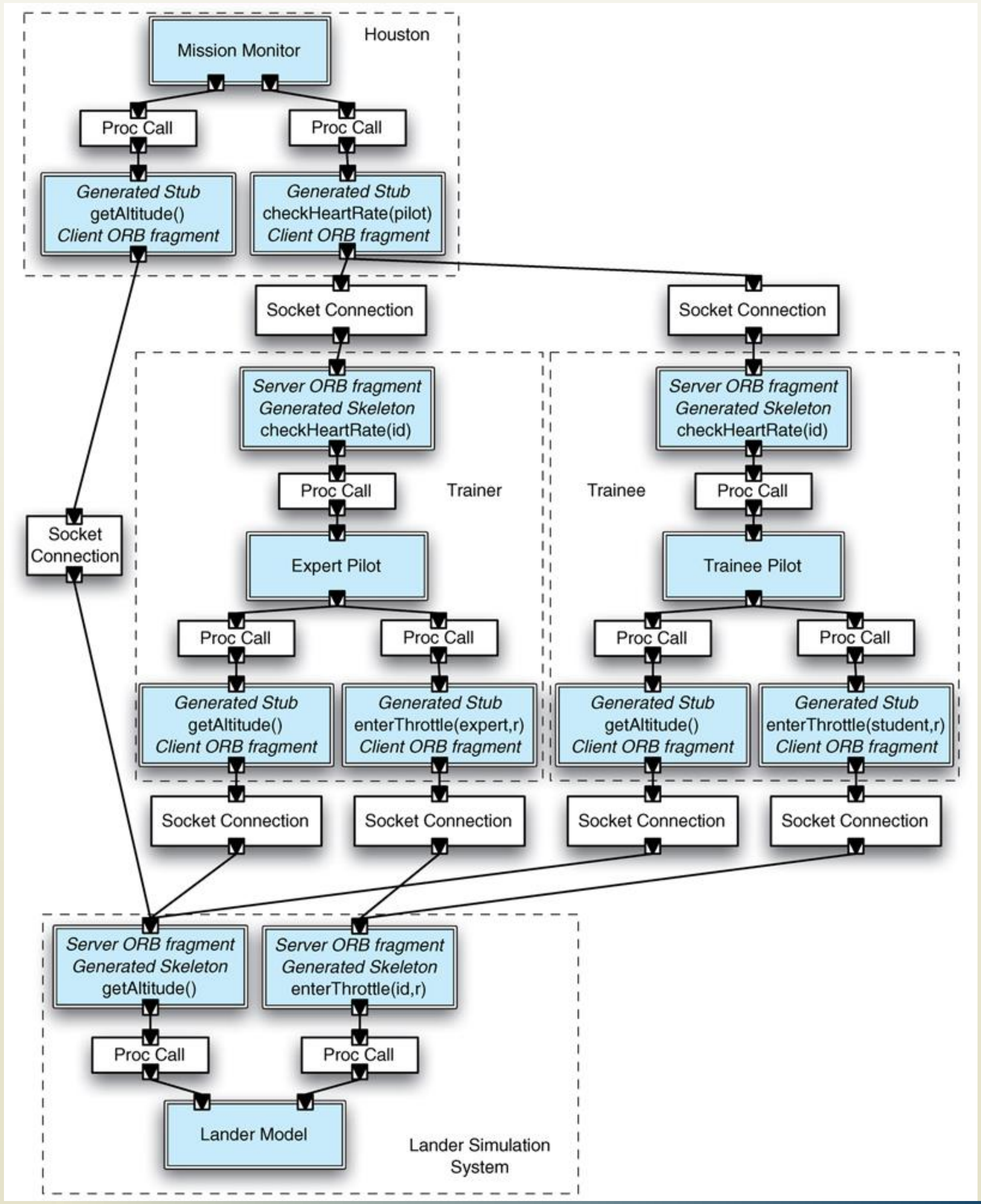
CORBA Concept and Implementation


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

CORBA Concept and Implementation (cont'd)

- The top portion shows the conceptual or logical view
  - A client program obtains a pointer to an object (`ObjectInstance`) that can perform a service for it
  - After obtaining this pointer, it makes calls to the object instance through one of its interface methods
- The bottom portion shows the implementation in a distributed environment
  - The system generates an object stub and an object skeleton, the former located at the machine of the client and the latter at the machine of the object
  - The client communicates with the object by calling the stub, which in turn calls the skeleton via the ORB

CORBA LL


Houston
Mission Monitor
Proc Call
Proc Call
Generated Stub getAltitude() Client ORB fragment
Generated Stub checkHeartRate(pilot) Client ORB fragment
Socket Connection
Socket Connection
Server ORB fragment Generated Skeleton checkHeartRate(id)
Server ORB fragment Generated Skeleton checkHeartRate(id)
Proc Call
Trainer
Trainee
Proc Call
Socket Connection
Expert Pilot
Trainee Pilot
Proc Call
Proc Call
Proc Call
Proc Call
Generated Stub getAltitude() Client ORB fragment
Generated Stub enterThrottle(expert,r) Client ORB fragment
Generated Stub getAltitude() Client ORB fragment
Generated Stub enterThrottle(student,r) Client ORB fragment
Socket Connection
Socket Connection
Socket Connection
Socket Connection
Server ORB fragment Generated Skeleton getAltitude()
Server ORB fragment Generated Skeleton enterThrottle(id,r)
Proc Call
Proc Call
Lander Model
Lander Simulation System

CORBA LL (cont'd)

- In this CORBA-based version, two independent user clients, `Trainer` and `Trainee`, can view the state of the Lander and can adjust the fuel consumption
- In addition, a third client, `Houston`, monitors the state of the Lander and the two pilots
- The state of the Lander is maintained on a remote server, the Lander Simulation System
- The clients manipulate the burn rate by making calls through the CORBA services
- Each pilot client also maintains the state of its pilot
- Houston can monitor both the state of the Lander as well as the states of the pilots

Observations about Styles

- 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; based upon the information from the divergence step, solution possibilities and new understandings of the problem are examined
  - 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
- A variety of transformation strategies are available, many of which can be seen and applied in software architecture
- Elements of a solution may be adapted, modified, substituted, reordered or combined

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 to assess the consequences of choices regarding these issues
  - Relate the costs of research and design to the penalty for taking wrong decisions; the penalty for not knowing must exceed the cost of finding out
  - Insulate uncertain decisions, where the design is unsure, or the related circumstances may change
  - Continually re-evaluate system “requirements” in light of what the design exploration yields

Putting it all Together

- The concepts and techniques for use in the design of software architectures have ranged from very basic ones (e.g., abstraction and separation of concerns) to extensive treatment of architectures, styles and patterns
- System architects must choose a feasible set of concepts on which they will build a satisfactory architecture
- The hardest part is to know whether to attempt to base the new design on some existing design or to start designing from scratch
- Significant insights can come from iteratively working with requirements but also from examining implementations

Insights from Requirements

- In many cases new architectures can be created based upon experience with and improvement to pre-existing architectures
- Architectures provide:
  - A vocabulary of basic concepts, means, approaches and possibilities
  - A framework to describe properties
  - A basis for analysis through knowledge of previous design decisions and related consequences

Insights from Requirements (cont'd)

- Experience in dealing with past design and new requirements can:
  - Show what the most critical issues are or what the most difficult problems are likely to be
  - Suggest what the key levels of discourse are and what vocabulary to use
  - Show successful patterns of product specialization
  - Suggest those architectural patterns and styles that have been effective in a specific domain
  - Show areas in which novel development is required

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

Summary

- Designing an architecture well is a skill to be learned and practiced
- The most effective approaches to design will be those that have been refined and seasoned within the domain of the new application
- In this chapter we introduced all the major conceptual elements of architecture-centric software engineering
- What remains is placing techniques and tools in the designer’s hands to enable productive exploitation of these conceptual elements