Software Design and Architecture

Lecture 07

Syed Salman Ahmed

Architectural Styles

Designing Architectures

- Who can design architectures?
- Can it be learnt?

- Design Process:
  - Feasibility stage
  - Preliminary design stage
  - Detailed design stage
  - Planning stage

Architectural Conception

- Experienced:
  - Abstraction & Simple machines
  - Choosing the levels & terms of discourse
  - Separation of concerns

- Non-Experienced:
  - Basic:
    - Divergence
    - Transformation
    - Convergence
  - Detailed:
    - Analogy
    - Brainstorming
    - Literature Searching
    - Morphological Charts
    - Removing mental blocks
  - Keep an eye on *requirements* and *implementation*

Design Styles

- Architectural styles are a primary way of characterizing lessons from experience in software system design.
- 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 and
  - elicit beneficial qualities in each resulting system

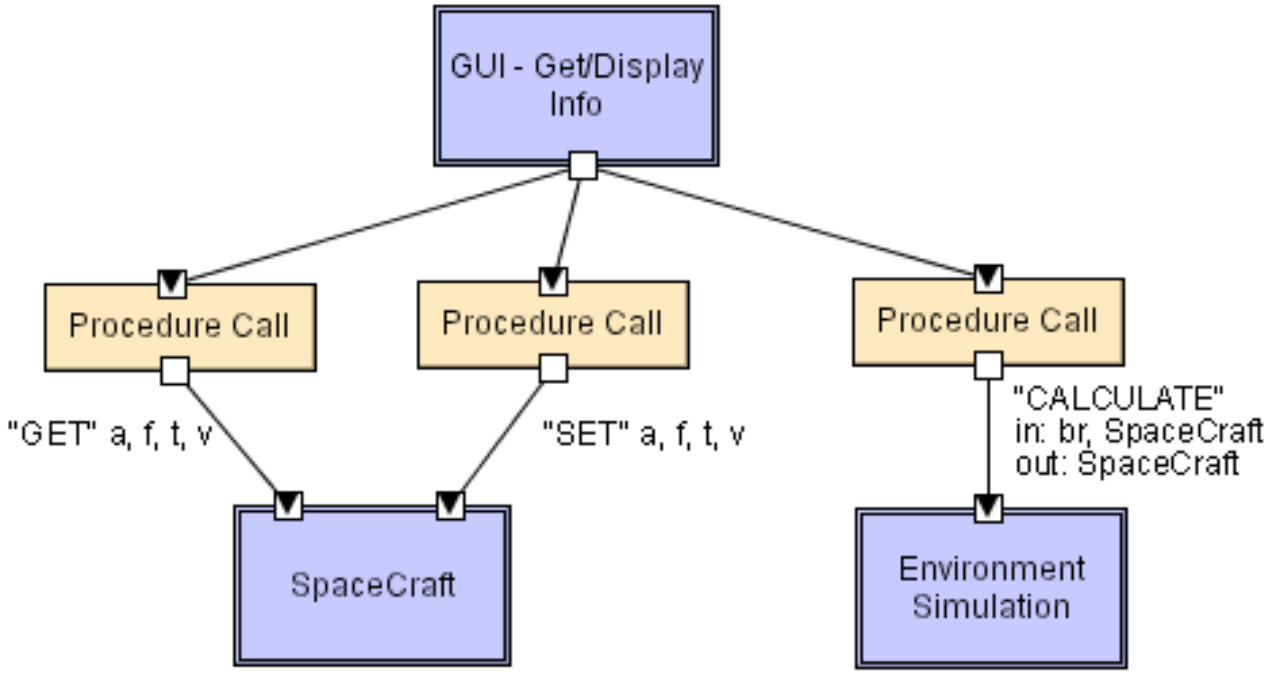
Design Styles - Classification

- 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
  - Publish subscribe
  - Event based
- Peer to peer

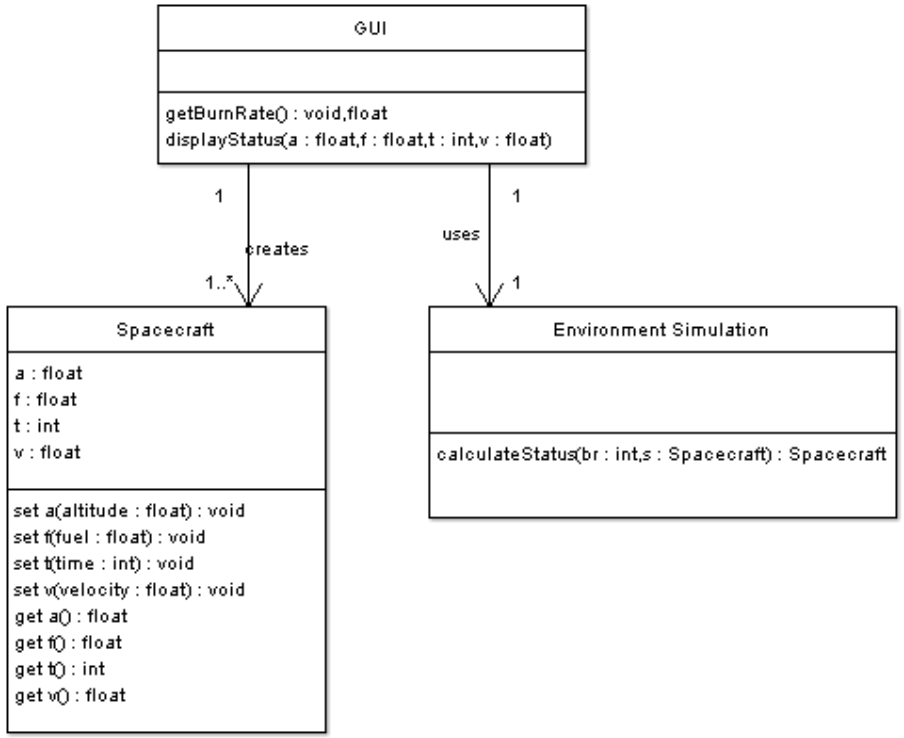
Object-Oriented Style

- 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



OO 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

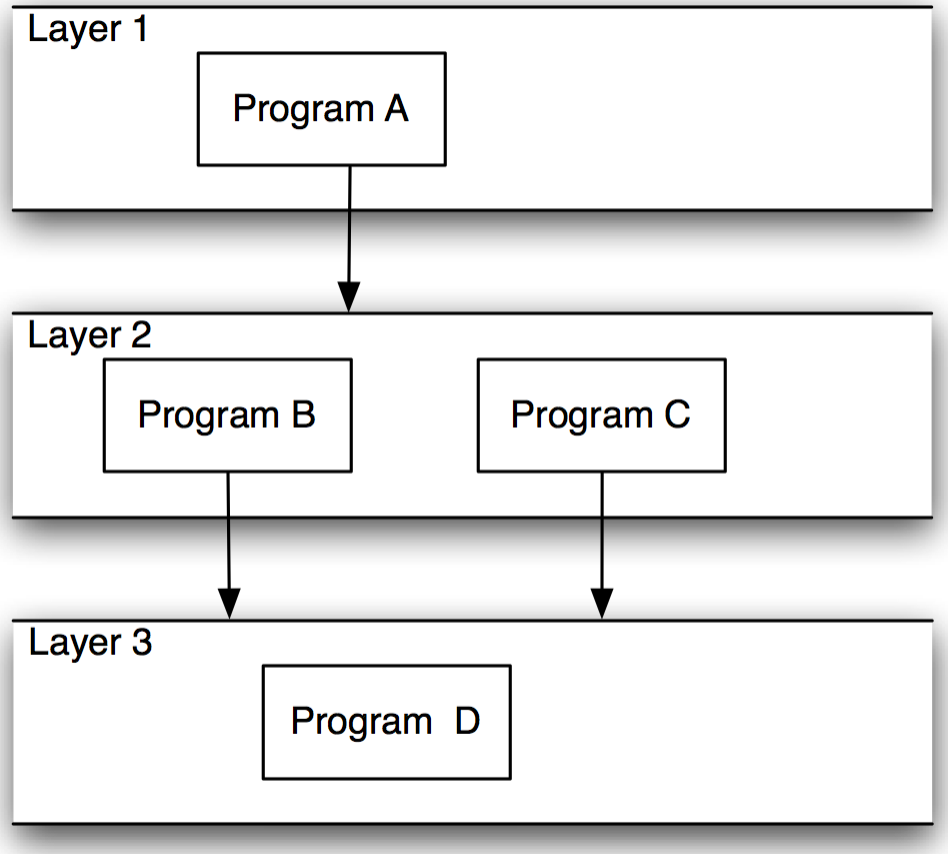
Layered Style (cont'd)

- 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

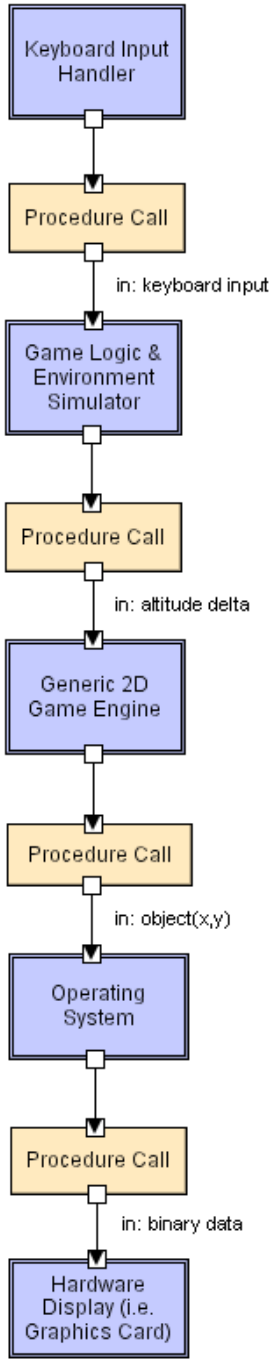
Layered Style (cont'd)

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

Layered Systems/Virtual Machines



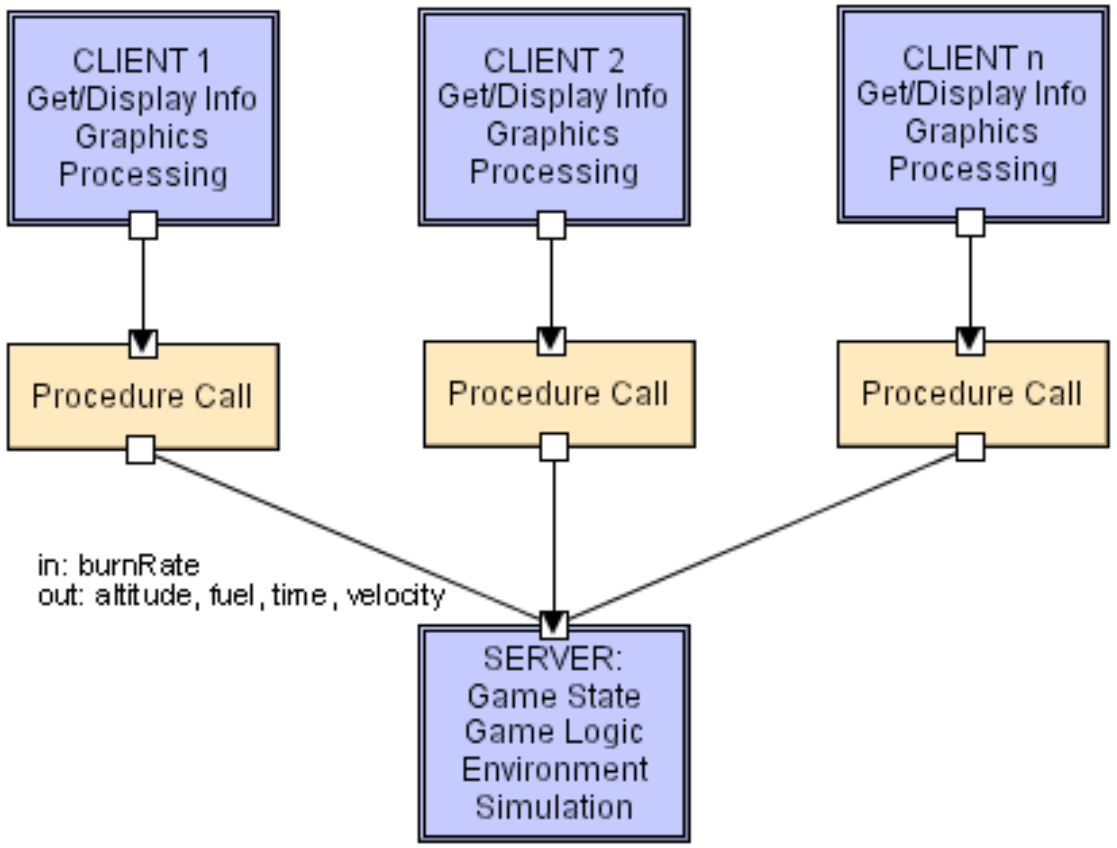
Layered


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



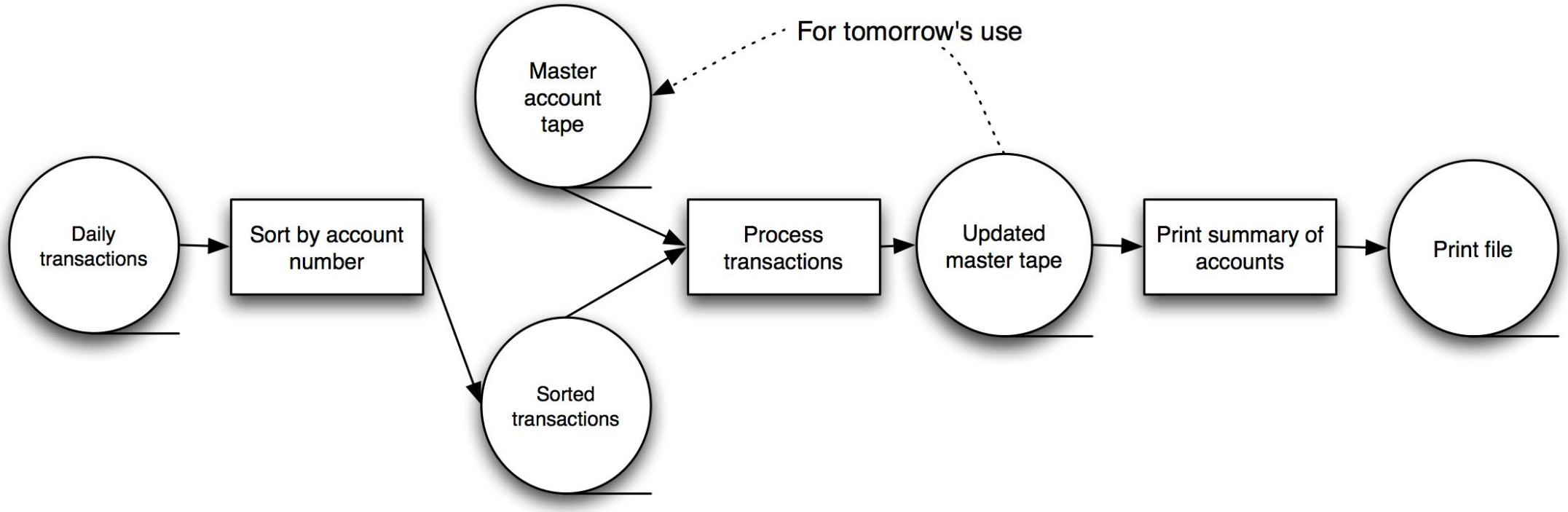
Data-Flow Styles

Batch Sequential

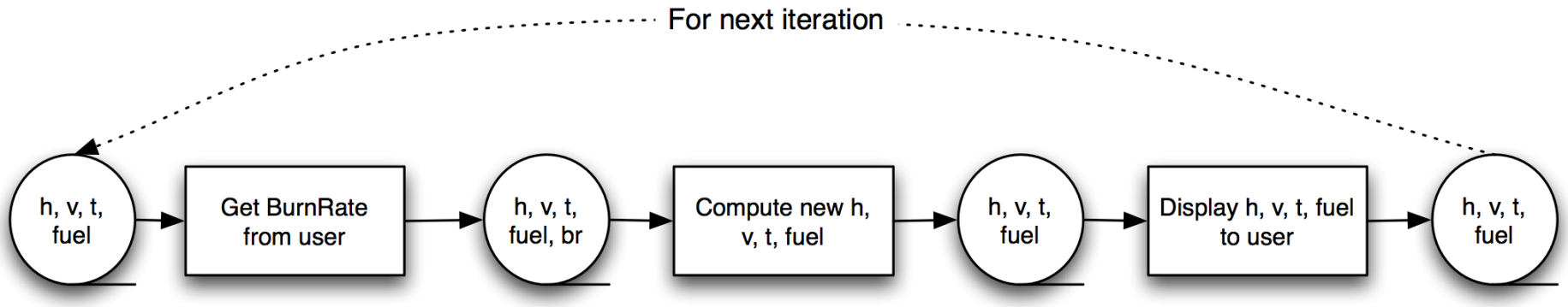
- Separate programs are executed in order; data is passed as an aggregate from one program to the next.
- Connectors: “The human hand” carrying tapes between the programs, a.k.a. “sneaker-net ”
- Data Elements: Explicit, aggregate elements passed from one component to the next upon completion of the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

- Typical uses: Transaction processing in financial systems. “The Granddaddy of Styles”

Batch-Sequential: A Financial Application



Batch-Sequential



Not a recipe for a successful lunar mission!

Pipe and Filter Style

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

Pipe and Filter (cont'd)

- Variations
  - Pipelines — linear sequences of filters
  - Bounded pipes — limited amount of data on a pipe
  - Typed pipes — data strongly typed
- Advantages
  - System behavior is a succession of component behaviors
  - Filter addition, replacement, and reuse
    - Possible to hook any two filters together
  - Certain analyses
    - Throughput, latency, deadlock
  - Concurrent execution

Pipe and Filter (cont'd)

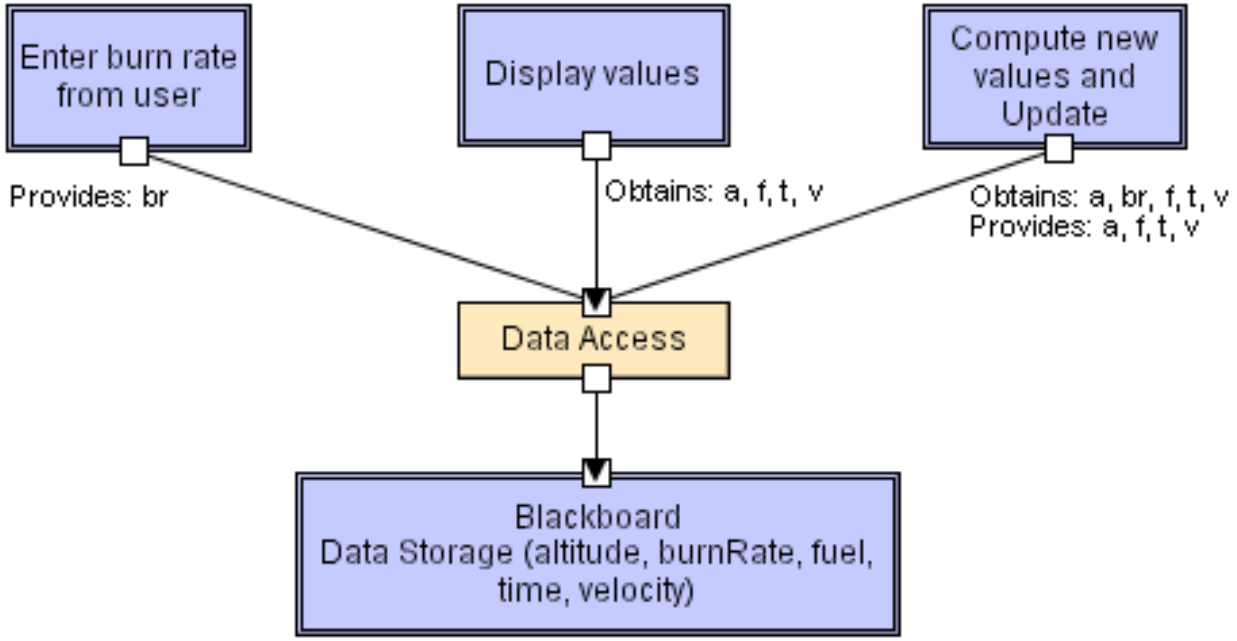
- Disadvantages
  - Batch organization of processing
  - Interactive applications
  - Lowest common denominator on data transmission

Pipe and Filter

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


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)

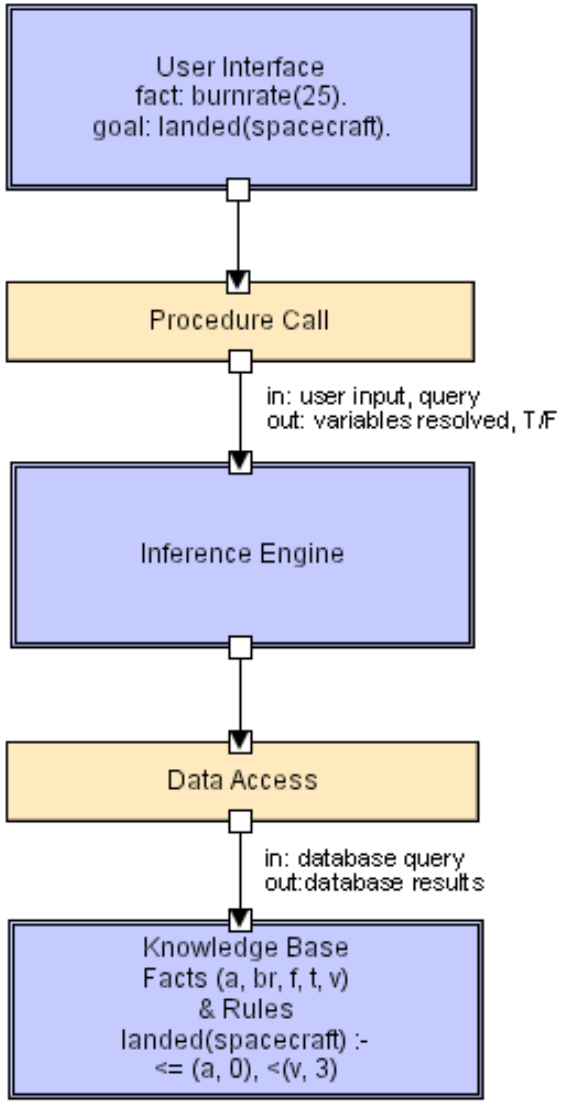
Rule-Based Style

- 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

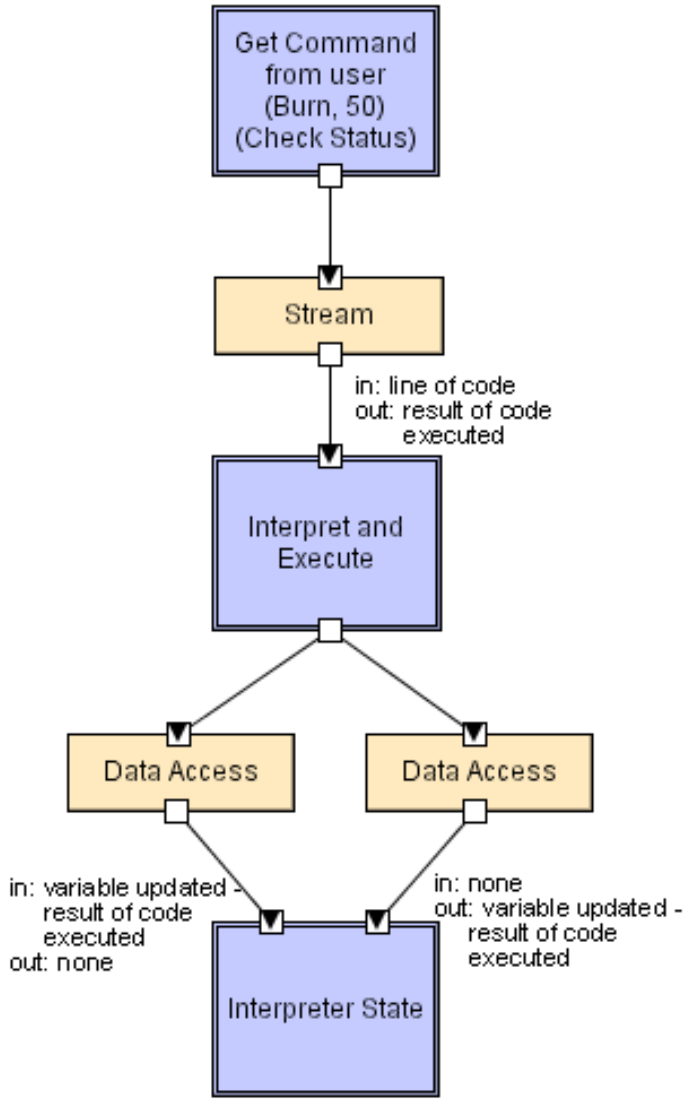

User Interface
fact: burnrate(25).
goal: landed(spacecraft).
Procedure Call
in: user input, query
out: variables resolved, T/F
Inference Engine
Data Access
in: database query
out:database results
Knowledge Base
Facts (a, br, f, t, v)
& Rules
landed(spacecraft) :-
<= (a, 0), <(v, 3)

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

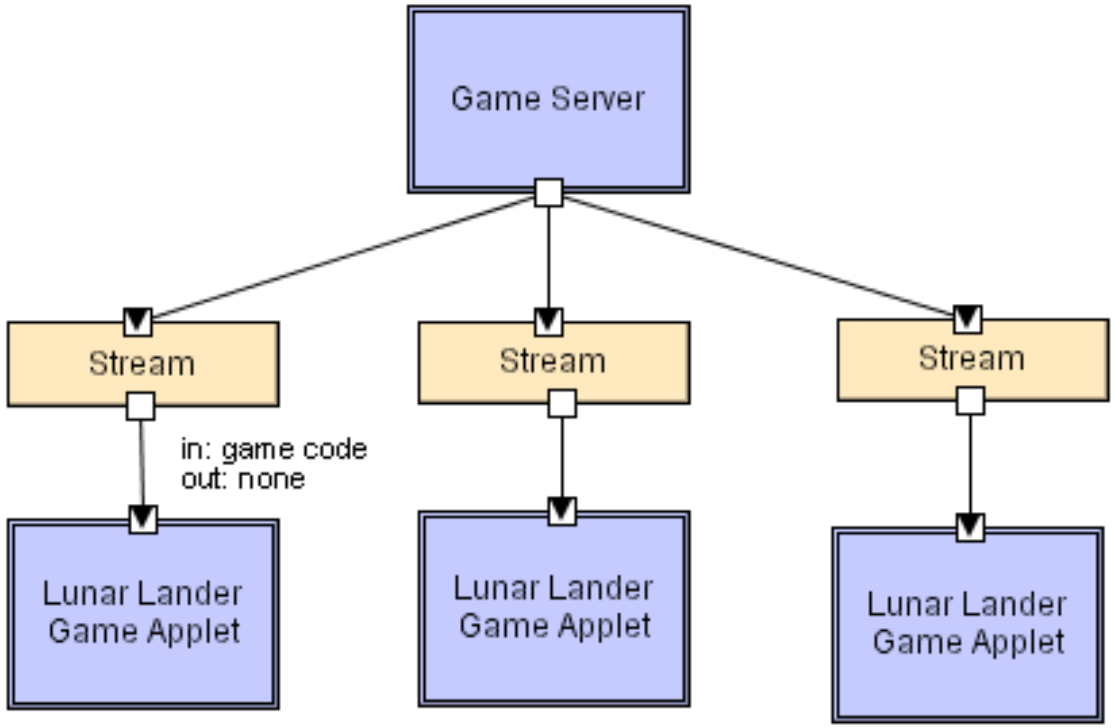
Interpreter



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



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

Implicit Invocation Style

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

Implicit Invocation (cont'd)

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

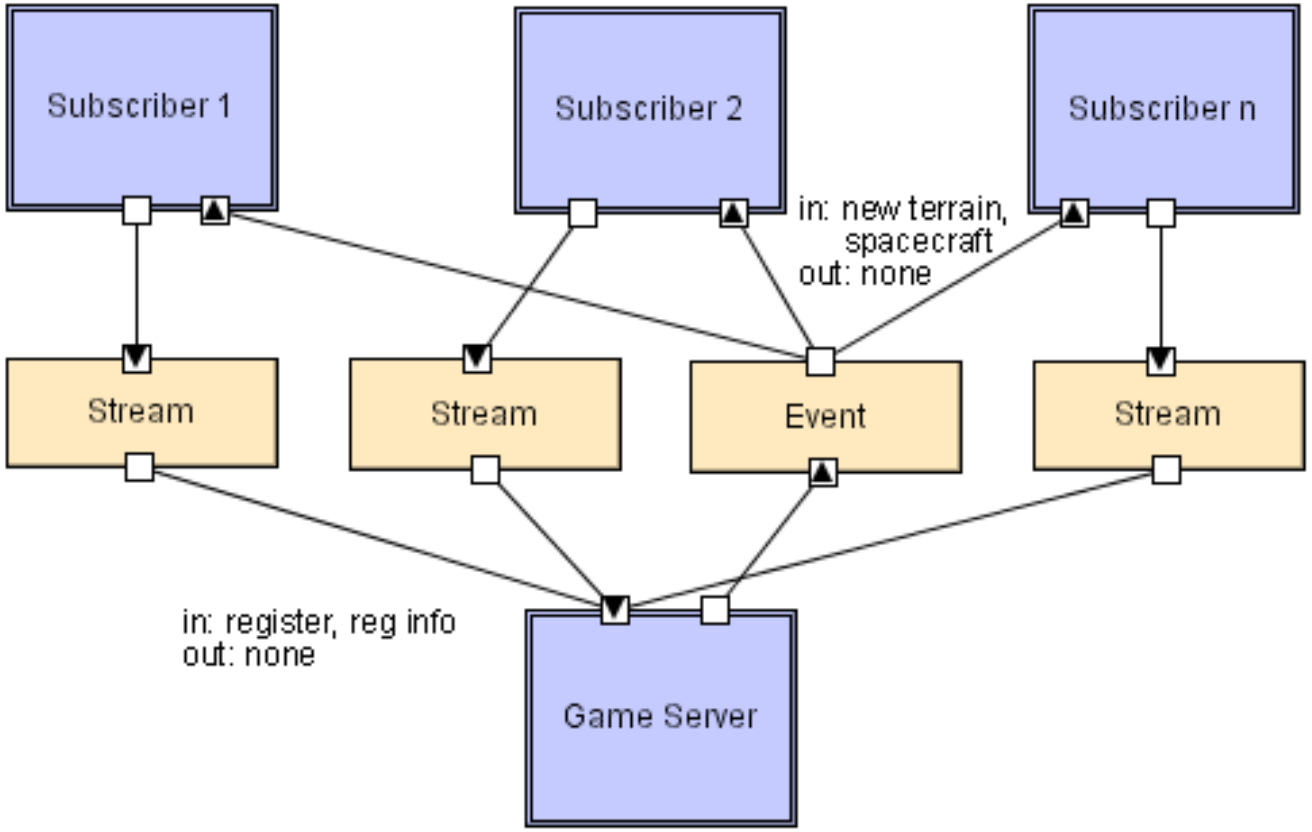
Publish-Subscribe

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

Publish-Subscribe (cont'd)

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

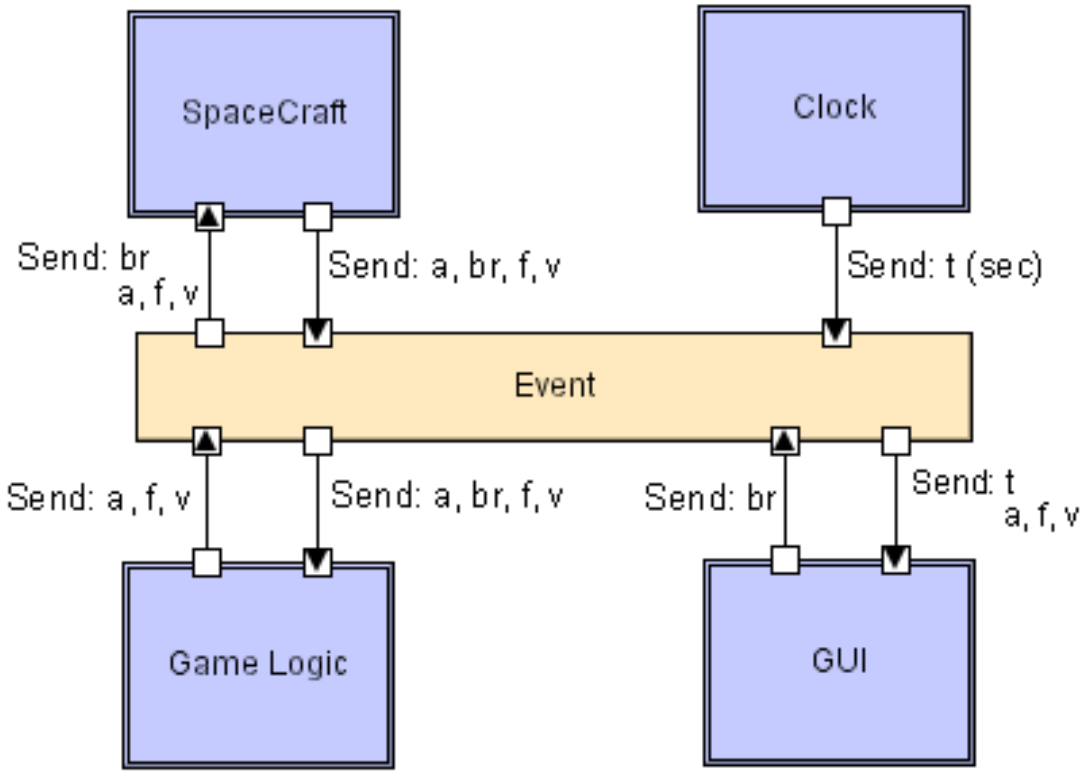
Pub-Sub


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


SpaceCraft
Clock
Send: br
a, f, v
Send: a, br, f, v
Send: t (sec)
Event
Send: a, f, v
Send: a, br, f, v
Send: br
Send: t
a, f, v
Game Logic
GUI

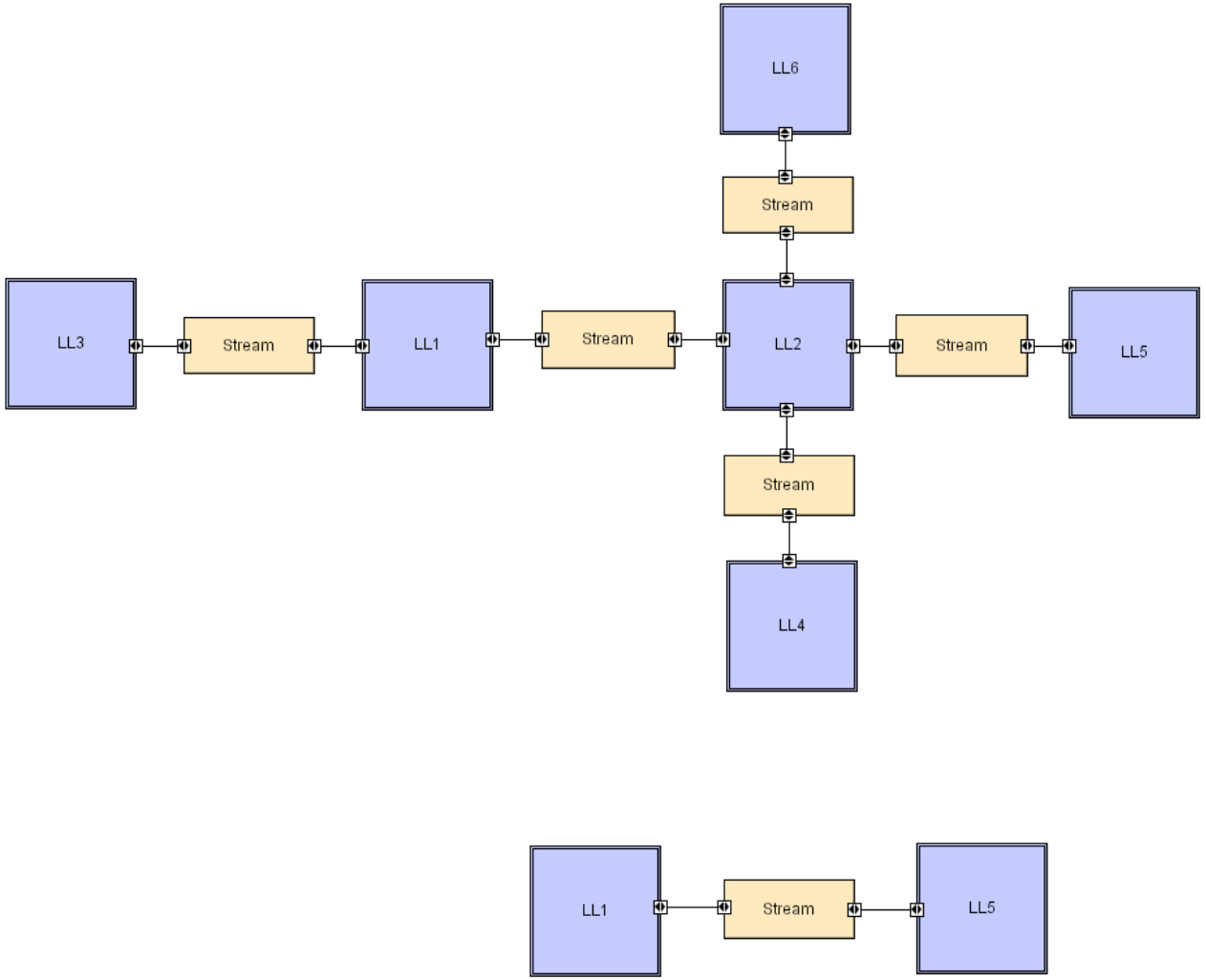
Peer-to-Peer Style

- 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


LL6
Stream
LL3
Stream
LL1
Stream
LL2
Stream
LL5
Stream
LL4
LL1
Stream
LL5

References

- *Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; Chapter 5