
Architectural Styles

Software Architecture

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 styles are a primary way of characterizing lessons from experience in software system design.

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

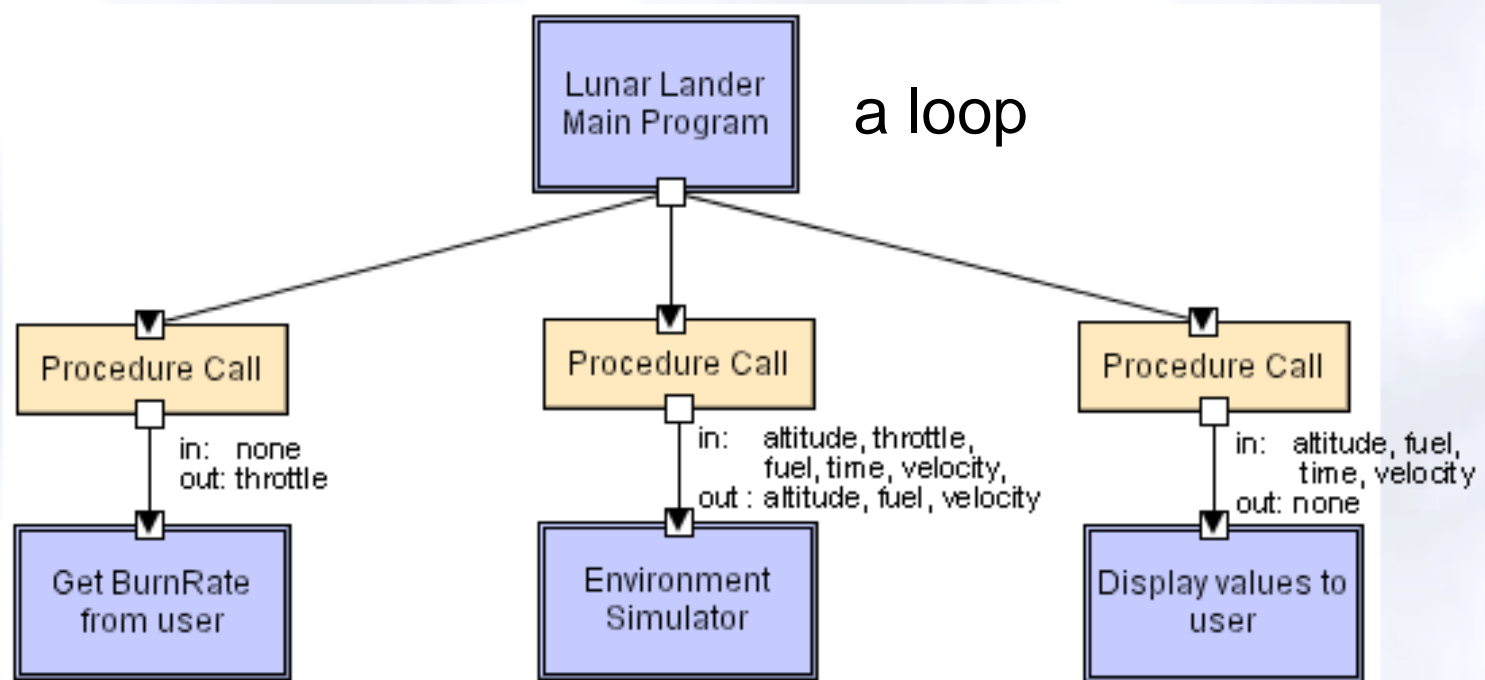
Main Program and Subroutines

- Summary: Decomposition based upon separation of functional processing steps.
- Components: Main program and subroutines.
- Connectors: Function/procedure calls.
- Data elements: Values passed in/out of subroutines.
- Topology: Static organization of components is hierarchical; full structure is a directed graph.
- Additional constraints imposed: None.

Main Program and Subroutines

- Qualities yielded : modularity. Subroutines may be replaced with different implementations long as interface semantics are unaffected.
- Typical uses: Small programs, pedagogical purposes.
- Cautions: Typically fails to scale to large applications; inadequate attention to data structures;
Unpredictable effort required to accommodate new requirements.
- Relations to programming languages or environments: Traditional imperative programming languages, such as BASIC, Pascal, or C.

Main Program and Subroutines (LL)



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

Object-Oriented Style

- Summary: State encapsulated with functions that operate on that state as objects; object must be instantiated before objects' methods can be called
- Components: objects
- Connectors: method invocations (procedure calls to manipulate states)
- Data elements: arguments to methods
- Qualities yielded:
integrity of data operations: data manipulated only by appropriate functions;
Abstraction

Object-Oriented Style

- Typical use:
 - ✓ Applications where the designer wants a close correlation between entities in the physical world and entities in the program;
 - ✓ applications involving complex, dynamic data structures.

- Cautions:

Distributed applications requires extensive middleware to **provide access to remote objects**.

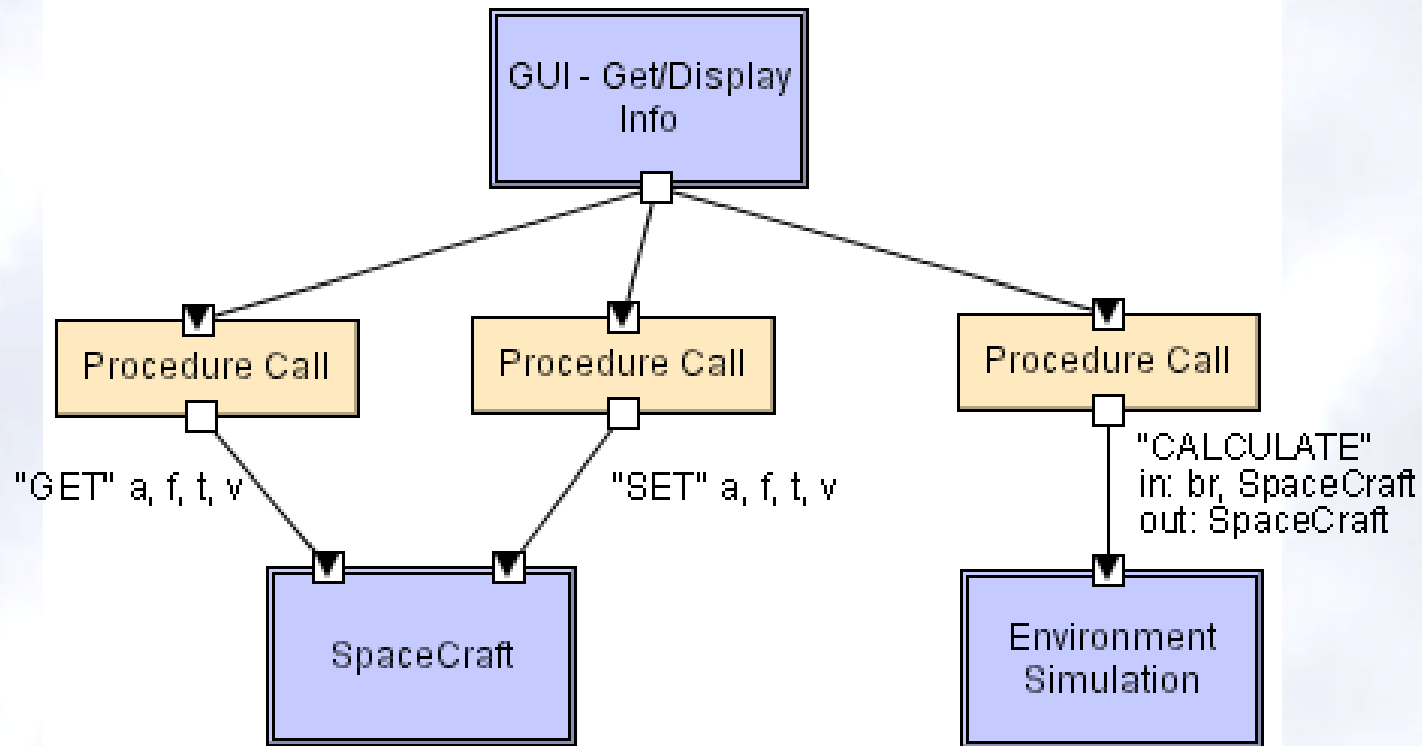
Inefficient for high-performance applications with large, numeric data structures, such as scientific computing.

- Relations to PL: Java, C++

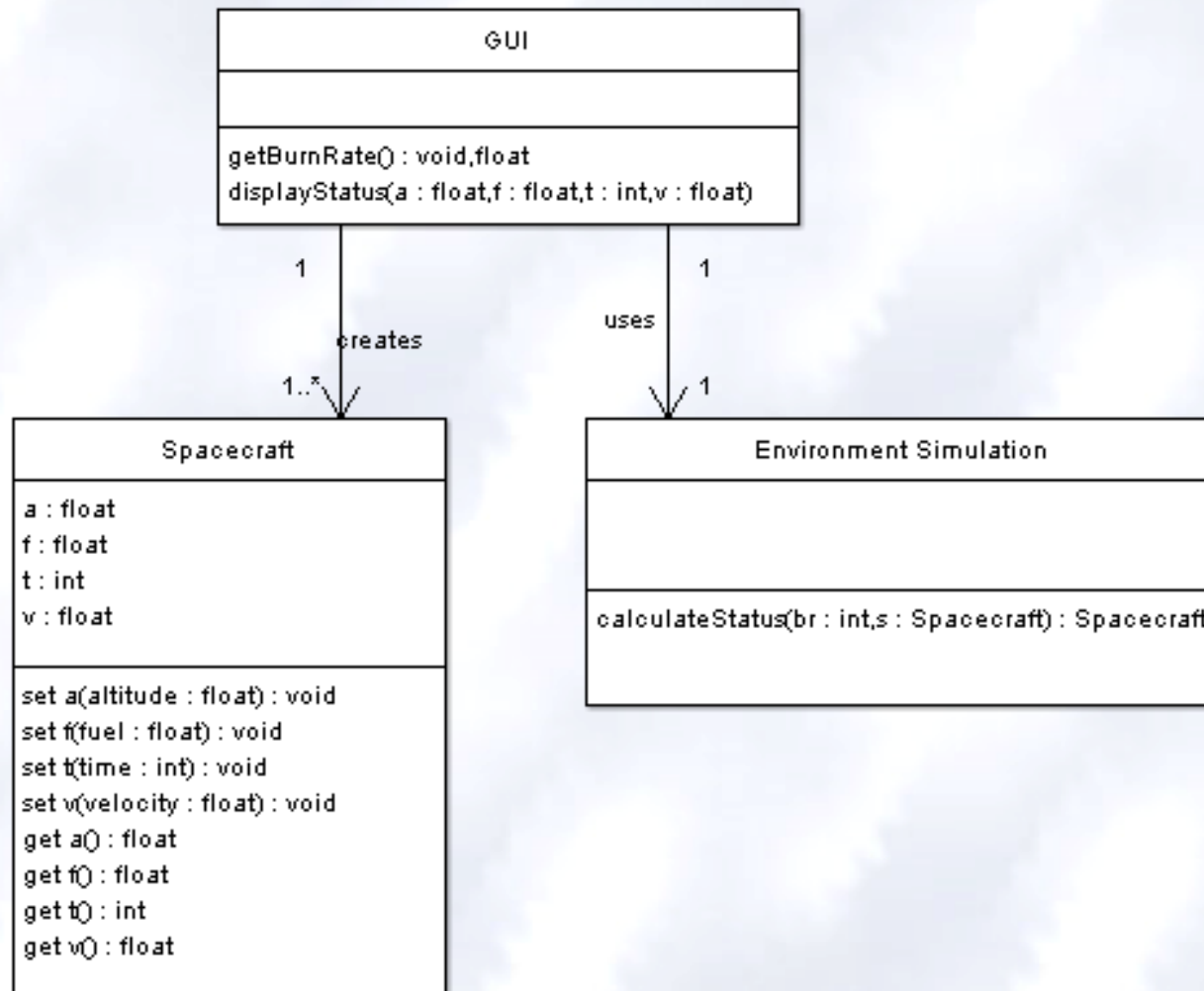
Object-Oriented Style

- 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



Some Common Styles

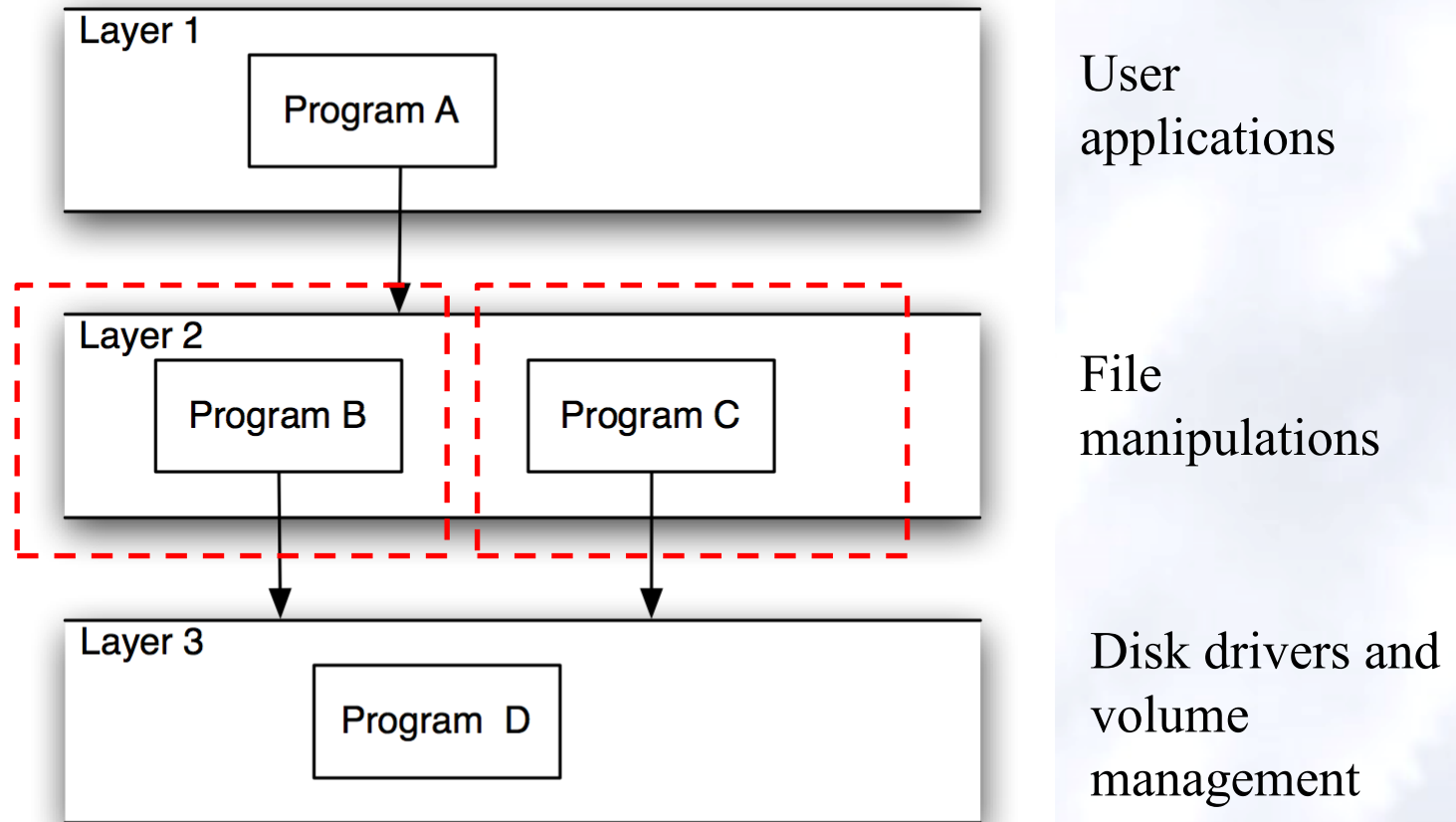
- Traditional, language-influenced styles
 - Main program and subroutines
 - Object-oriented
- **Layered**
 - Virtual machines**
 - Client-server
- Data-flow styles
 - Batch sequential
 - Pipe and filter
- Shared memory
 - Blackboard
 - Rule based
- Interpreter
 - Interpreter
 - Mobile code
- Implicit invocation
 - Event-based
 - Publish-subscribe
- Peer-to-peer
- “Derived” styles
 - C2
 - CORBA

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

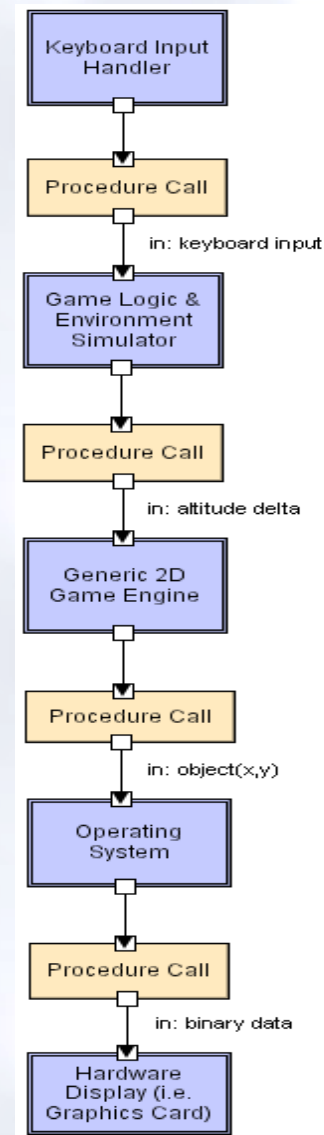
The essence of a layered style is that an architecture is separated into ordered layers, wherein a program within one layer may obtain services from a layer below it.

Layered Systems/Virtual Machines



Operating systems designs

Layered LL



Layered Style (cont'd)

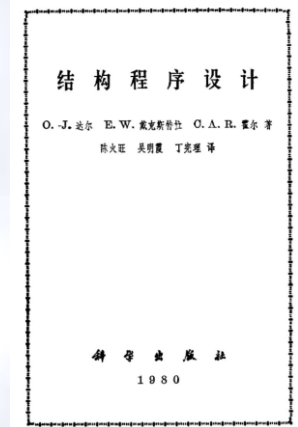
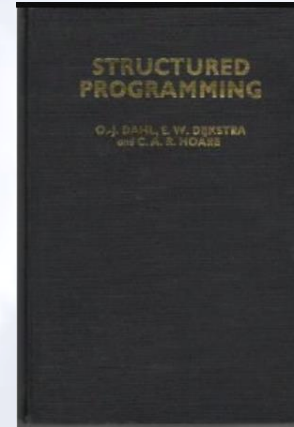
Edsger Wybe Dijkstra(1930-2002), 荷兰计算机科学家, 1972年图灵奖获得者。

最短路径算法, 开创性贡献如编译器、操作系统、分布式系统、程序设计、编程语言、程序验证、软件工程、图论等。

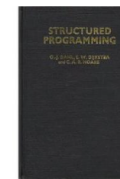
独立发明了逆波兰式并用栈实现表达式的求值;
银行家算法; 哲学家就餐问题、睡眠理发师问题;
临界区、死锁、信号量;

提出GOTO有害论, 开启**结构化编程运动**, 此前对人们来说程序只是写给计算机的最重要的是能run起来, 此后人们意识到除了能run还需要别人也能看懂;

1972年与C.A.R. Hoare 和Ole-Johan Dahl合著<<**结构化编程**>>一书; 参与了第一个ALGOL60编译器的设计和实现, 这是最早支持递归的编译器; 设计实现了具有多任务支持的批处理系统THE, 并**首次提出操作系统的分层结构**。从70年代开始, Dijkstra的研究兴趣开始转向形式化验证。

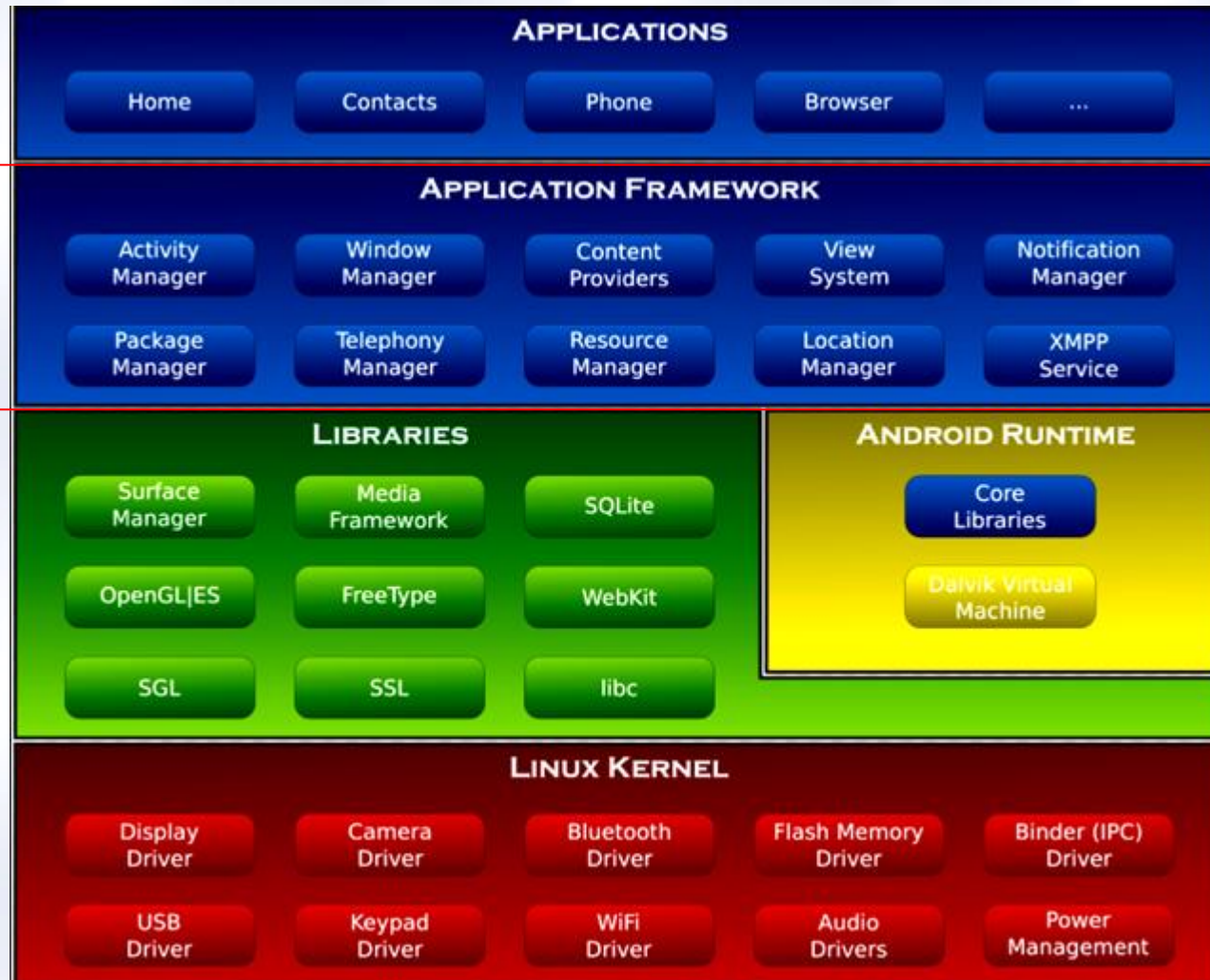


Structured Programming



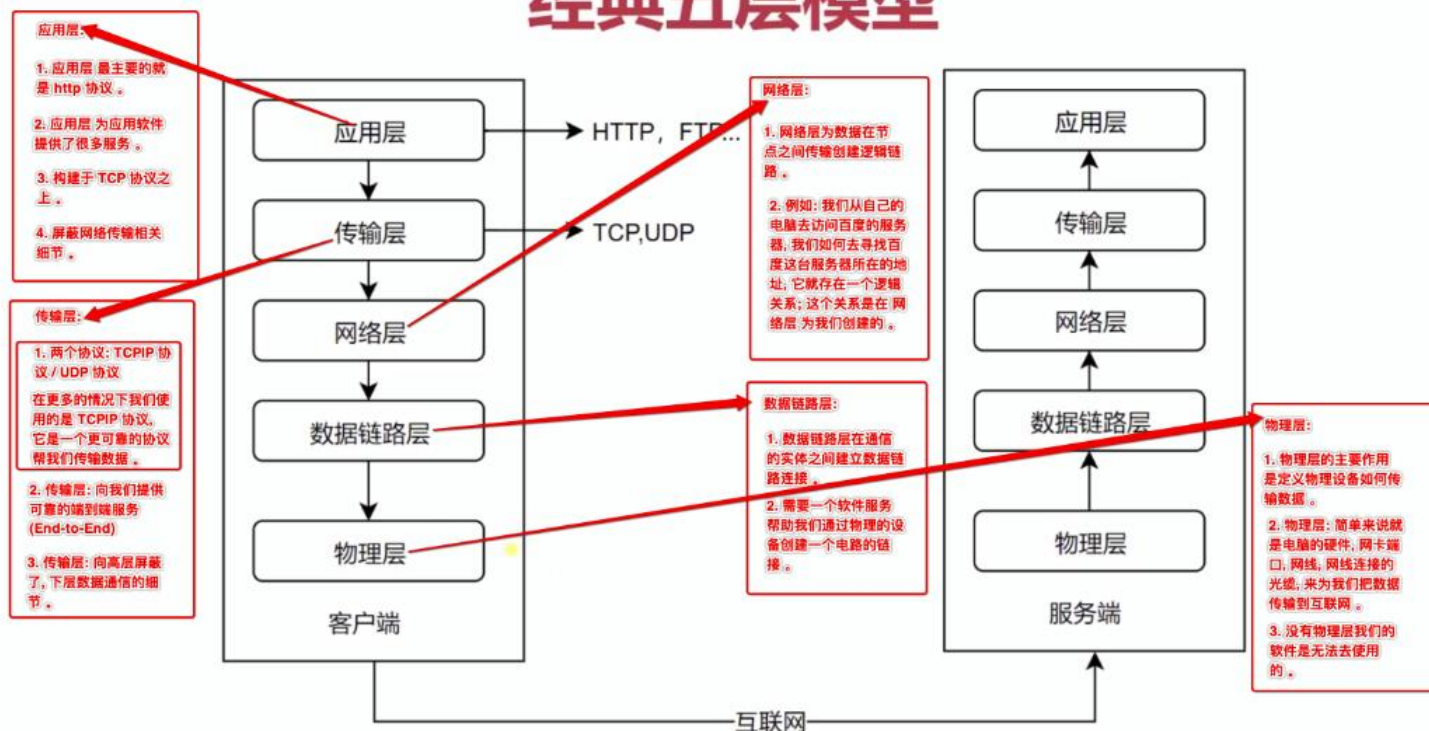
作者: Edsger Wybe Dijkstra / C. A. R. Hoare, / Ole-Johan Dahl
出版社: Academic Press
出版年: 1972-2-11
页数: 418
定价: USD 18.00
装帧: Hardcover
ISBN: 9780122005503

Layered Style (cont'd)



Layered Style (cont'd)

经典五层模型



Layered Style (cont'd)

- Summary

Consists of an ordered sequence of layers;

Each layer, or virtual machine, offers a set of services that may be accessed by programs residing within the layer above it.

- Components: layers, typically comprising several programs
- Connectors: typically procedure calls
- Data elements: parameters passed between layers
- Topology: linear, for strict vms; a directed acyclic graph
- Qualities: clear dependency structure; sw upper immune to the changes of the lower; the lower is independent of upper levels

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

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

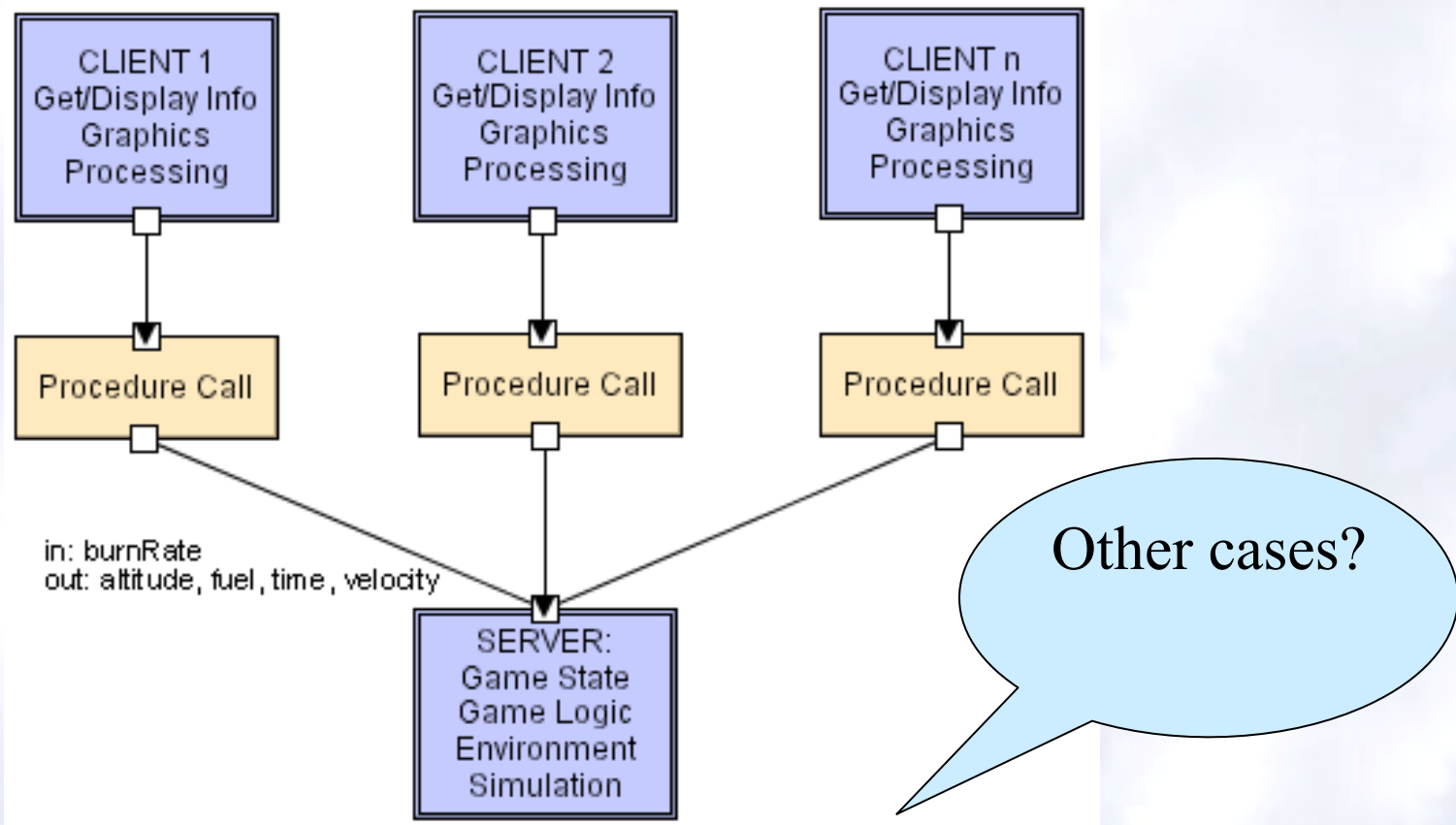
Client-Server Style

- Simply, is understood as a two-layer virtual machine with network connections.
- The server is the virtual machine below the clients, each of which accesses the virtual machine's interfaces via remote procedure calls or equivalent network access methods.
- Typically, there are multiple clients that access the same server;
- The clients are mutually independent.
- The obligation of the server is to provide the specific services requested by each client.

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



Client-Server LL

- **Summary:** Clients send service requests to the server, which performs the required functions and replies as needed with the requested information. Communication is initiated by the clients.
- **Components:** Clients/servers;
- **Connectors:** Remote procedure call, network protocols.
- **Data elements:** Parameters and return values as sent by the connectors.
- **Topology:** Two level, with multiple clients making requests to the server.

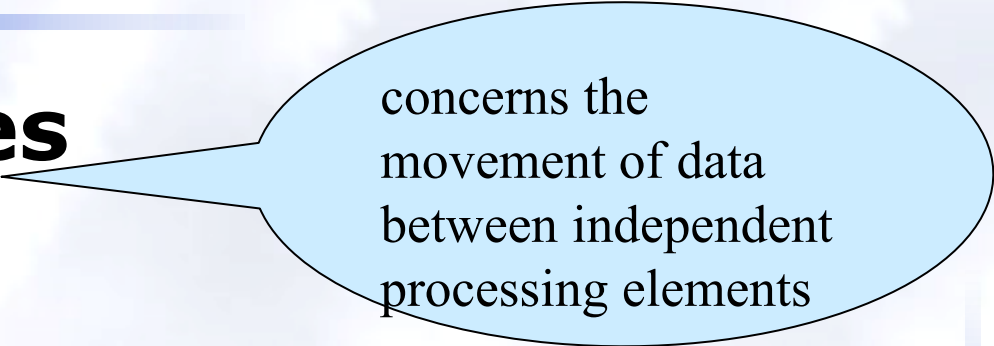
Client-Server LL

- Additional constraints imposed: Client-to-client communication prohibited.
- **Qualities yielded:** Centralization of computation and data at the server; A single powerful server can service many clients.
- **Typical uses:** Centralization of computation and data are required
- **Cautions:** a large number of client requests.

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

Data-Flow Styles



concerns the
movement of data
between independent
processing elements

Batch Sequential

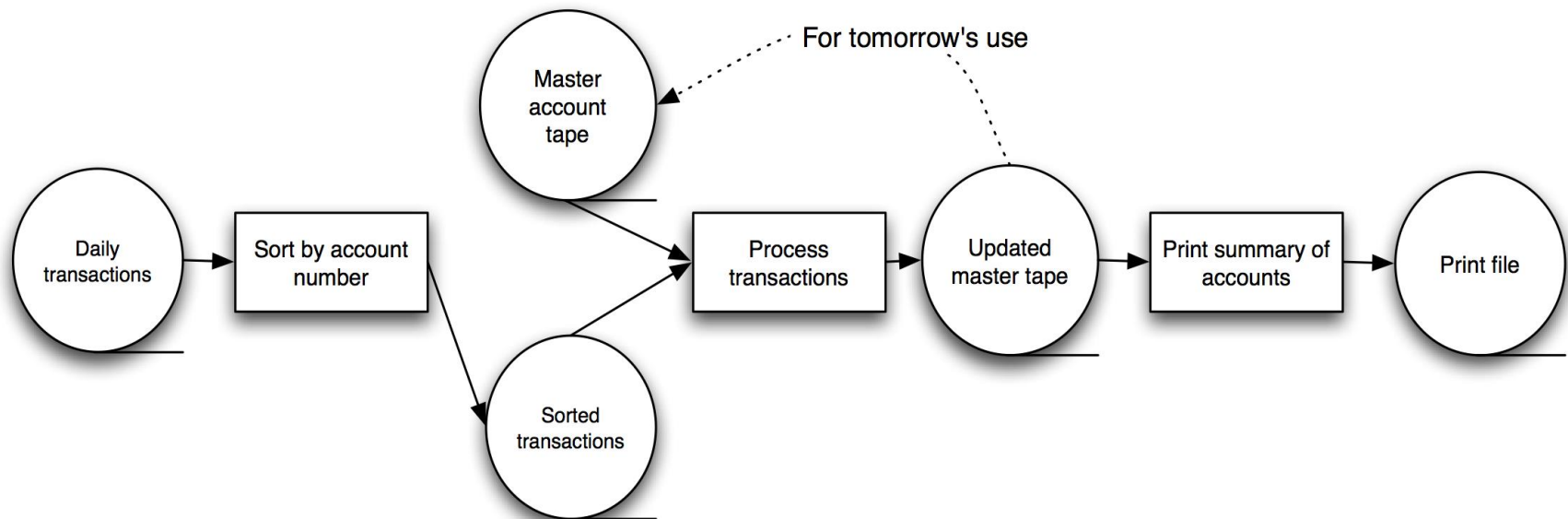
Separate programs are executed in order; data is passed as an aggregate from one program to the next.

Connectors: "The human hand" carrying tapes between the programs, a.k.a. "sneaker-net "

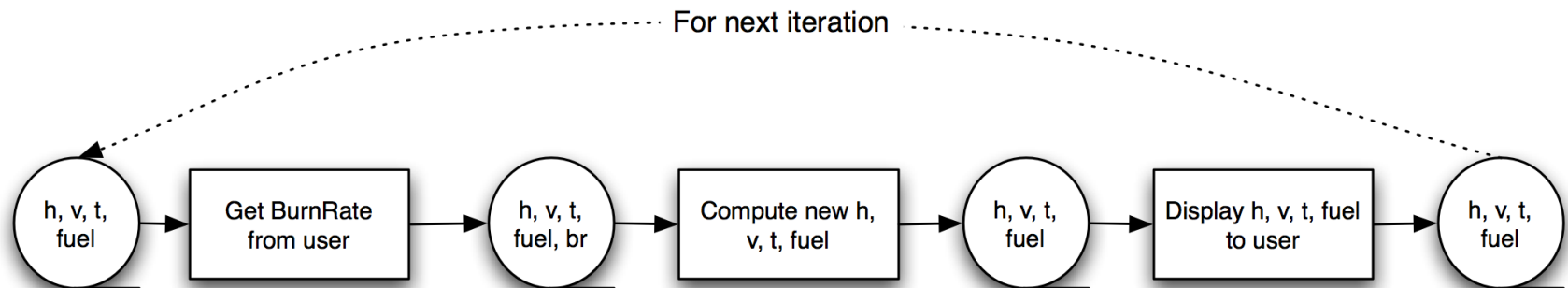
Data Elements: Explicit, aggregate elements passed from one component to the next upon completion of the producing program's execution.

- 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!

Not a highly interactive, real-time game!

Data-Flow Styles

Batch Sequential

- **Summary:** Separate programs are executed in order; data is passed as an aggregate from one program to the next.
- **Components:** Independent programs.
- **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.
- **Topology:** Linear.

Data-Flow Styles

Batch Sequential

- Additional constraints imposed: **One program runs a time, to completion.**
- Qualities yielded: simplicity.
- Typical uses: Transaction processing in financial systems.
- Cautions: When interaction between the components is required; when concurrency between components is possible or required.
- Relations to programming languages or environments: None

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

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 vs. Batch Sequential

The filters can operate concurrently, with no requirement for a producing component to finish before a component that consumes the producer's output begins

Pipe and Filter (cont'd)

Summary: Separate programs are executed, potentially concurrently; data is passed as a stream from one program to the next.

Components: Independent programs, known as filters.

Connectors: Explicit routers of data streams; service provided by operating system.

Data elements: Not explicit; must be (linear) data streams. In the typical Unix/Linux/DOS implementation the streams must be text.

Topology: Pipeline.

Pipe and Filter (cont'd)

Qualities yielded: Filters are mutually independent. Simple structure of incoming and outgoing data streams facilitates novel combinations of filters for new, composed applications.

Typical uses: Ubiquitous in operating system application programming.

Cautions: When complex data structures must be exchanged between filters; when interactivity between the programs is required.

Relations to programming languages or environments: Prevalent in Unix shells:

Pipe and Filter LL



Other cases?

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

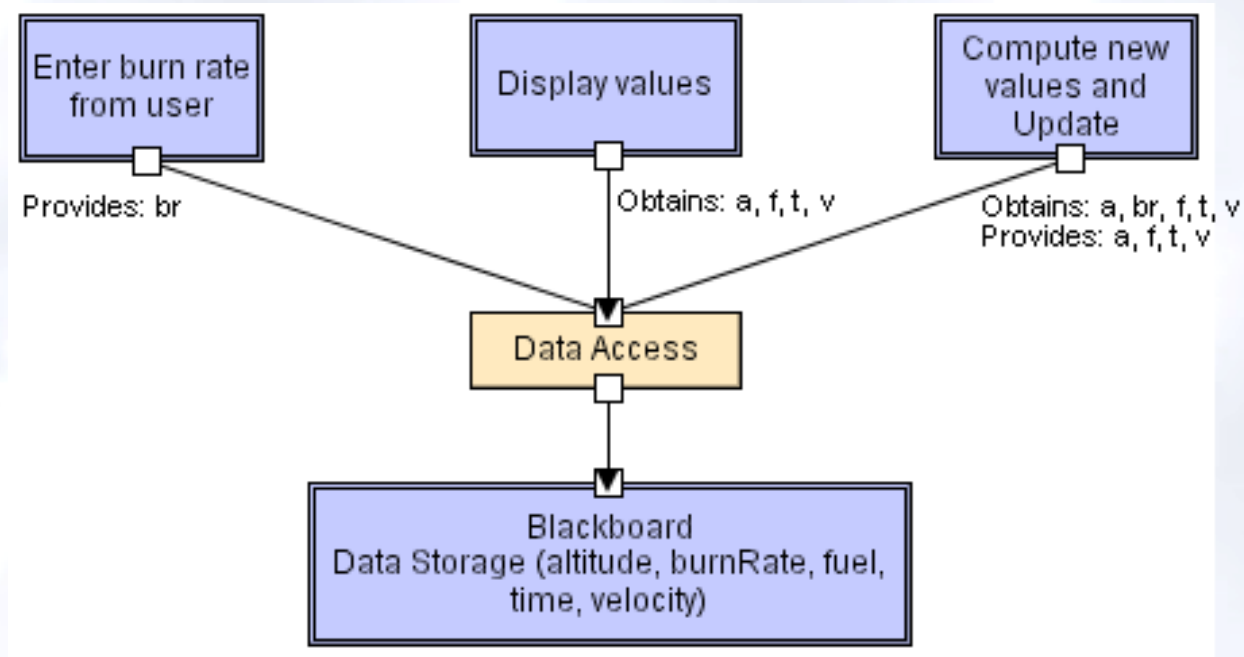
Shared memory Style

- The essence of **shared state styles** (sometimes colloquially referred to as **shared memory styles**) is that multiple components have access to the **same data store**, and communicate through that data store.
- This corresponds roughly to the ill-advised practice of **using global data** in C or Pascal programming.
- The difference is that with shared state styles **the center of design attention is explicitly on these structured, shared repositories**, and that consequently they 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



Blackboard Style

- **Summary:** Independent programs access and communicate exclusively through a global data repository, known as a blackboard.
- **Components:** Independent programs, sometimes referred to as "knowledge sources," blackboard.
- **Connectors:** Access to the blackboard may be by **direct memory reference**, or can be through a **procedure call** or a **database query**.
- **Data elements:** Data stored in the center.
- **Topology:** **Star** topology, with the blackboard at the center.

Blackboard Style

- **Qualities yielded:** Complete solution strategies to complex problems **do not have to be preplanned.** **Evolving views** of data/problem determine the strategies that are adopted.
- **Typical uses:** Heuristic problem solving in artificial intelligence applications.
- **Cautions:** When a well-structured solution strategy is available; when interactions between the independent programs require complex regulation; when representation of the data on the blackboard is subject to frequent change.

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

Rule-Based/Expert 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.

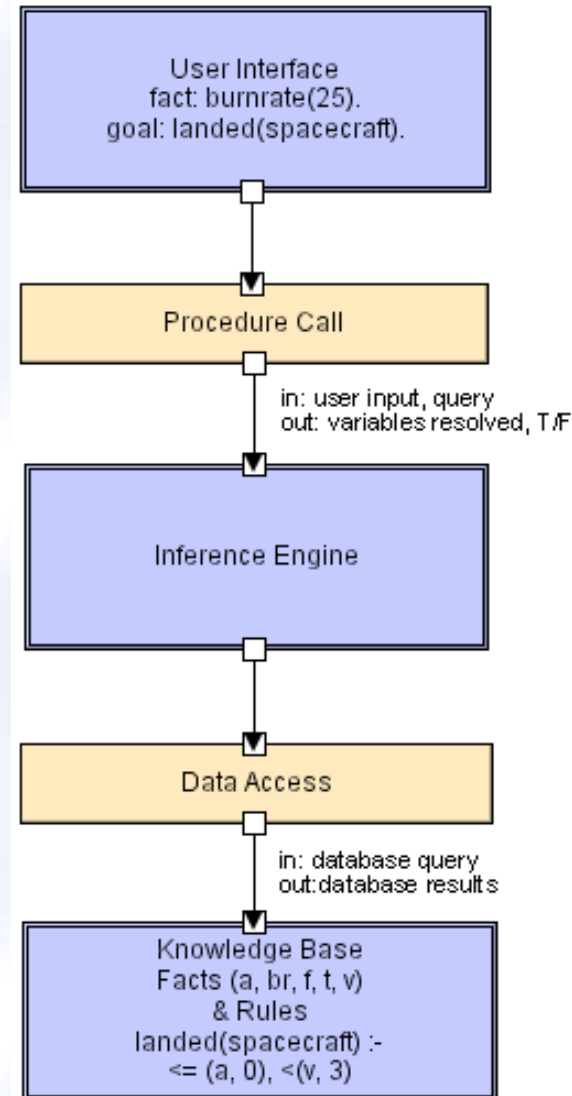
The shared memory is a so-called **knowledge base**.

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

IF fact1 & fact2
Then results



Some Common Styles

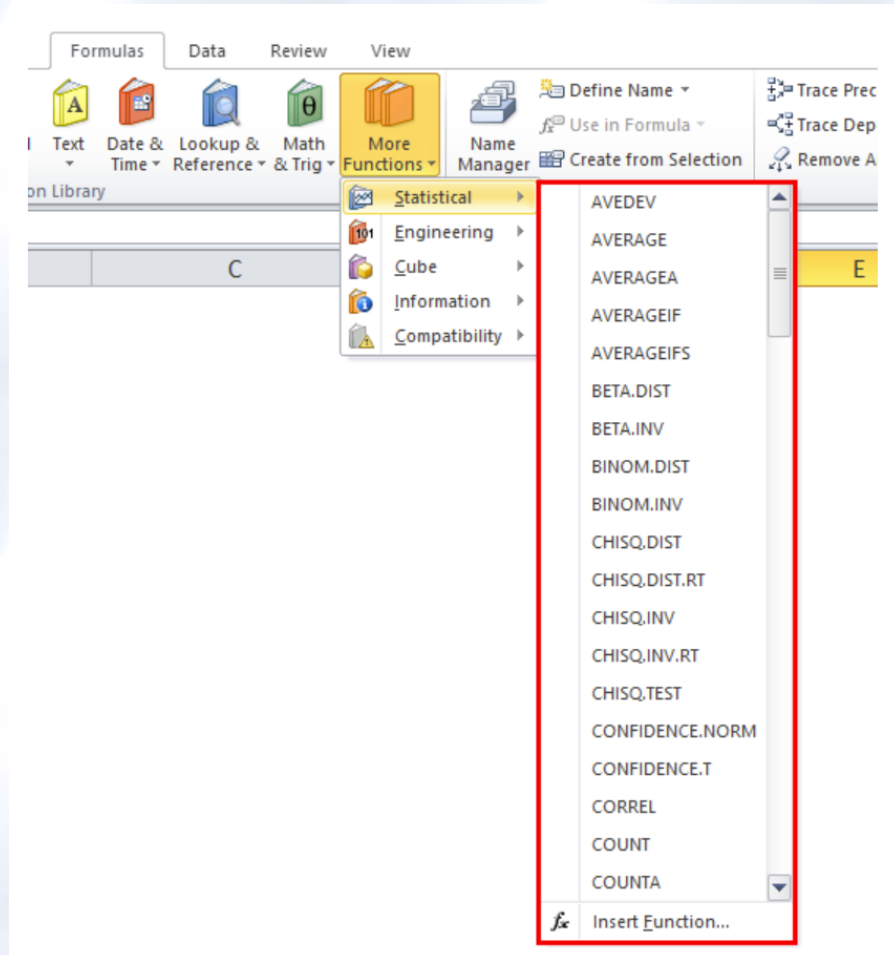
- Traditional, language-influenced styles
 - Main program and subroutines
 - Object-oriented
- Layered
 - Virtual machines
 - Client-server
- Data-flow styles
 - Batch sequential
 - Pipe and filter
- Shared memory
 - Blackboard
 - Rule based
- **Interpreter**
 - Interpreter**
 - Mobile code
- Implicit invocation
 - Event-based
 - Publish-subscribe
- Peer-to-peer
- “Derived” styles
 - C2
 - CORBA

Interpreter Style

- The distinctive characteristic of interpreter styles is **dynamic, on-the-fly interpretation of commands.**
- Commands are **explicit statements**, possibly created moments before they are executed, possibly encoded in human-readable and editable text.
- Interpretation proceeds by starting with an initial execution state, obtaining the first command to execute, executing the command over the current execution state, thereby modifying that state, then proceeding to execute the next command.

Interpreter Style

- Excel's formulas are in fact commands interpreted by the Excel execution engine-the interpreter.
- Excel's macros are interpreted by the Visual Basic interpreter.



Interpreter Style

- **Summary:** 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.
- **Quality yielded:** 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.
- **Uses:** Superb for end-user programmability; supports dynamically changing set of capabilities
- Lisp and Scheme

Interpreter Style

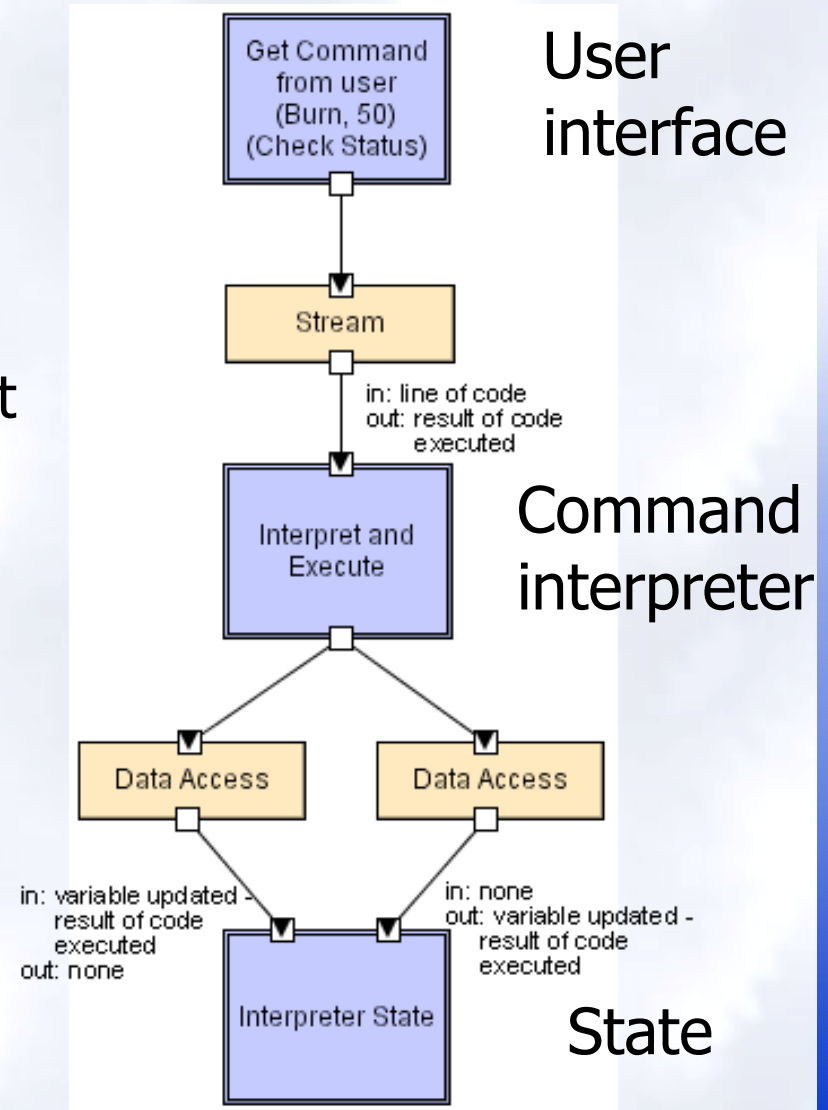
- **Caution:**

When fast processing is needed;
it takes longer to execute interpreted code than executable code;
memory management may be an issue, especially when multiple interpreters are invoked simultaneously.

Interpreter LL

"BurnRate(50)": the interpreter takes BurnRate as the command and the parameter as the amount of fuel to burn. It then calculates the necessary updates to the altitude, fuel level, and velocity.

"CheckStatus": the user will receive the current state of altitude, fuel, time, and velocity.



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

Mobile-Code Style

- Mobile code styles **enable code to be transmitted to a remote host for interpretation.**
- This may be due to a lack of local computing power, lack of resources, or due to large data sets remotely located.

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 Style

- **Code-on-demand**

Resource (local); code (remote); execute code locally

- **Remote evaluation**

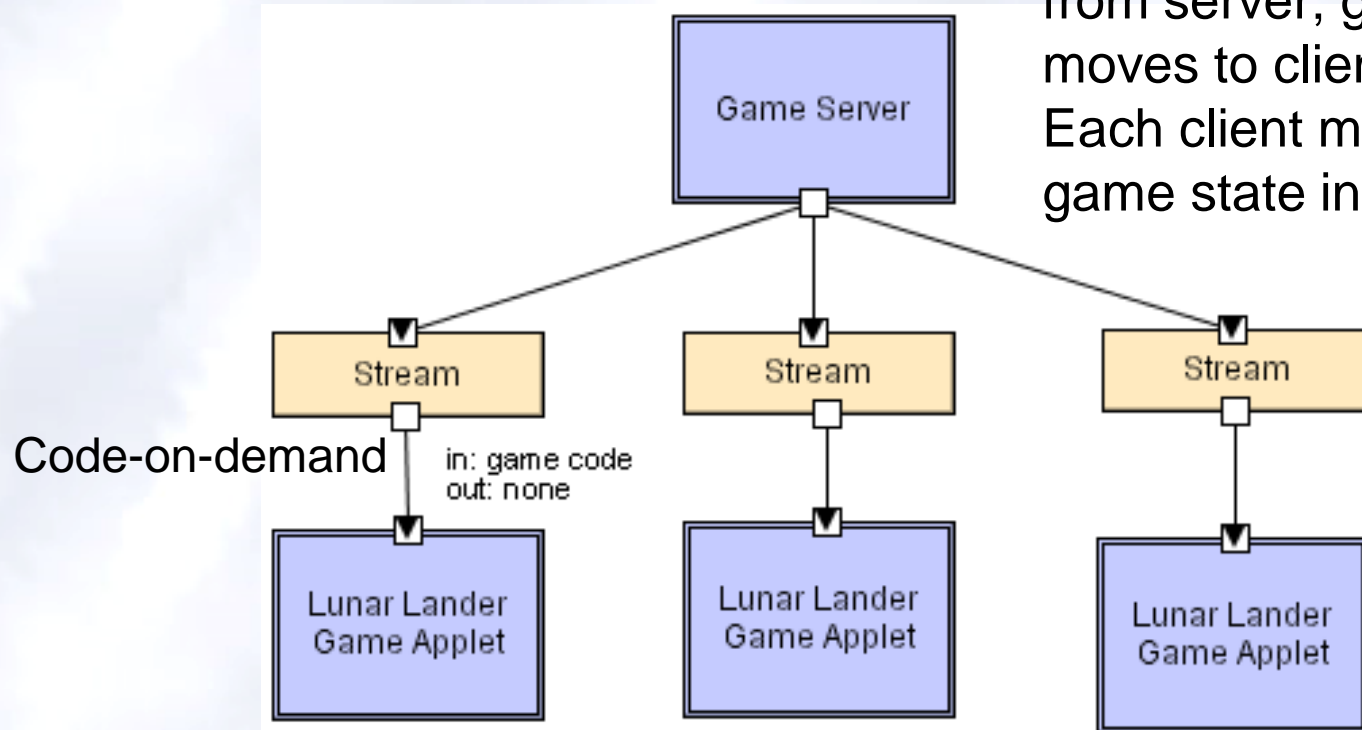
Resource (remote); code (local); execute remotely

- **Mobile agent**

Some resources, state, code (local); move to the agent to execute code with states.

Mobile Code LL

Download scripting code from server; game logic moves to clients.
Each client maintains the game state independently.



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

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**
 - Publish-subscribe**
 - Event-based**
- Peer-to-peer
- “Derived” styles
 - C2
 - CORBA

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; loosely coupled components

- 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

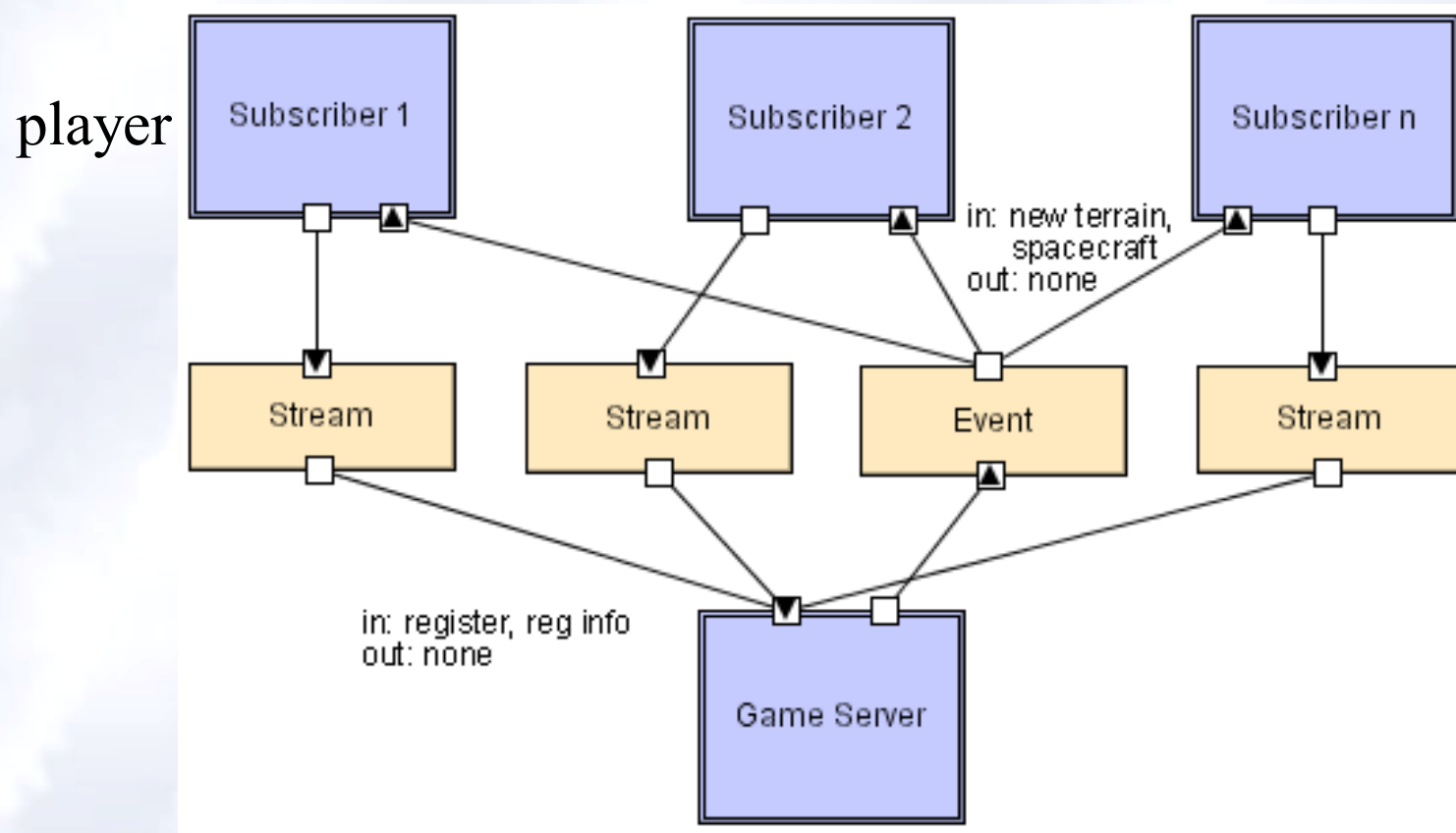
Publish-Subscribe

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



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**
 - Publish-subscribe**
 - Event-based**
- Peer-to-peer
- “Derived” styles
 - C2
 - CORBA

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 Style

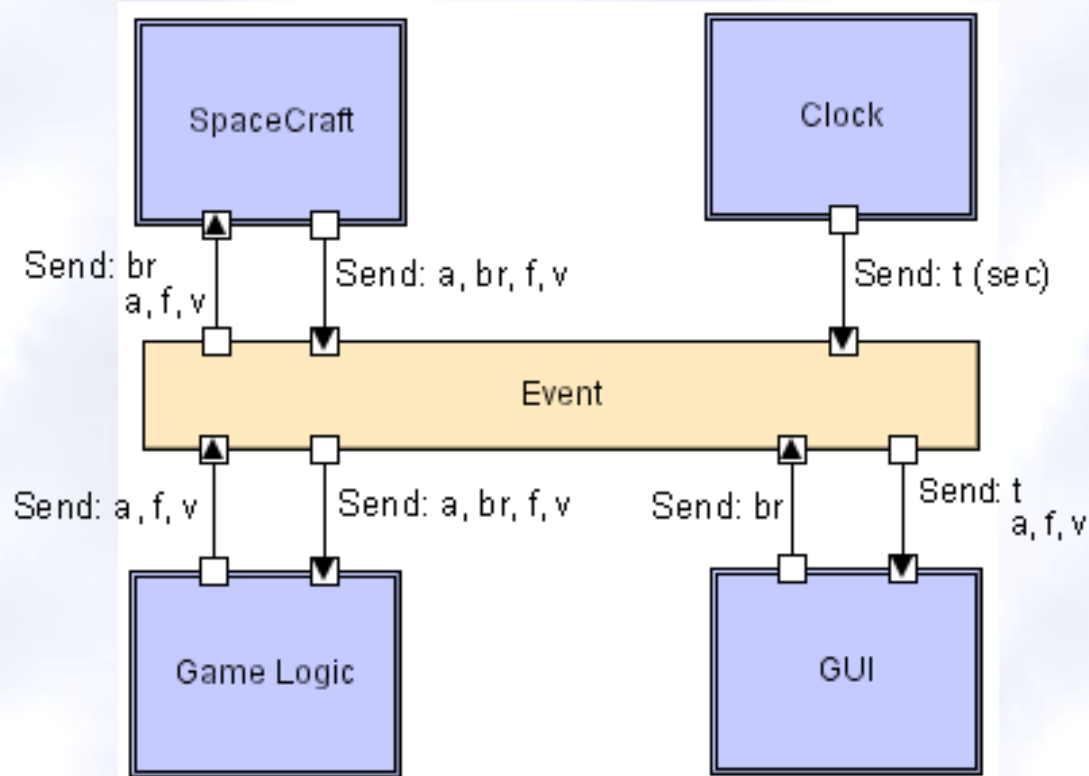
- Event-based vs. publish-subscribe

There is no classification of components into pub. and sub.

All component potentially both emit and receive events.

Event-based style is suited to strongly **decoupled concurrent** components, where at any given time a component either may be creating information of potential interest to others or may be consuming information.

Event-based LL



Summary of implicit invocation style

- Though there is a causal relationship, the invocation is **indirect**, since there **is no direct coupling between the components involved**.
- The invocation is **implicit**, since the causing component has no awareness that its action ultimately causes an invocation elsewhere in the system.

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
 - Publish-subscribe
 - Event-based
- **Peer-to-peer**
- “Derived” styles
 - C2
 - CORBA

Peer-to-Peer Style

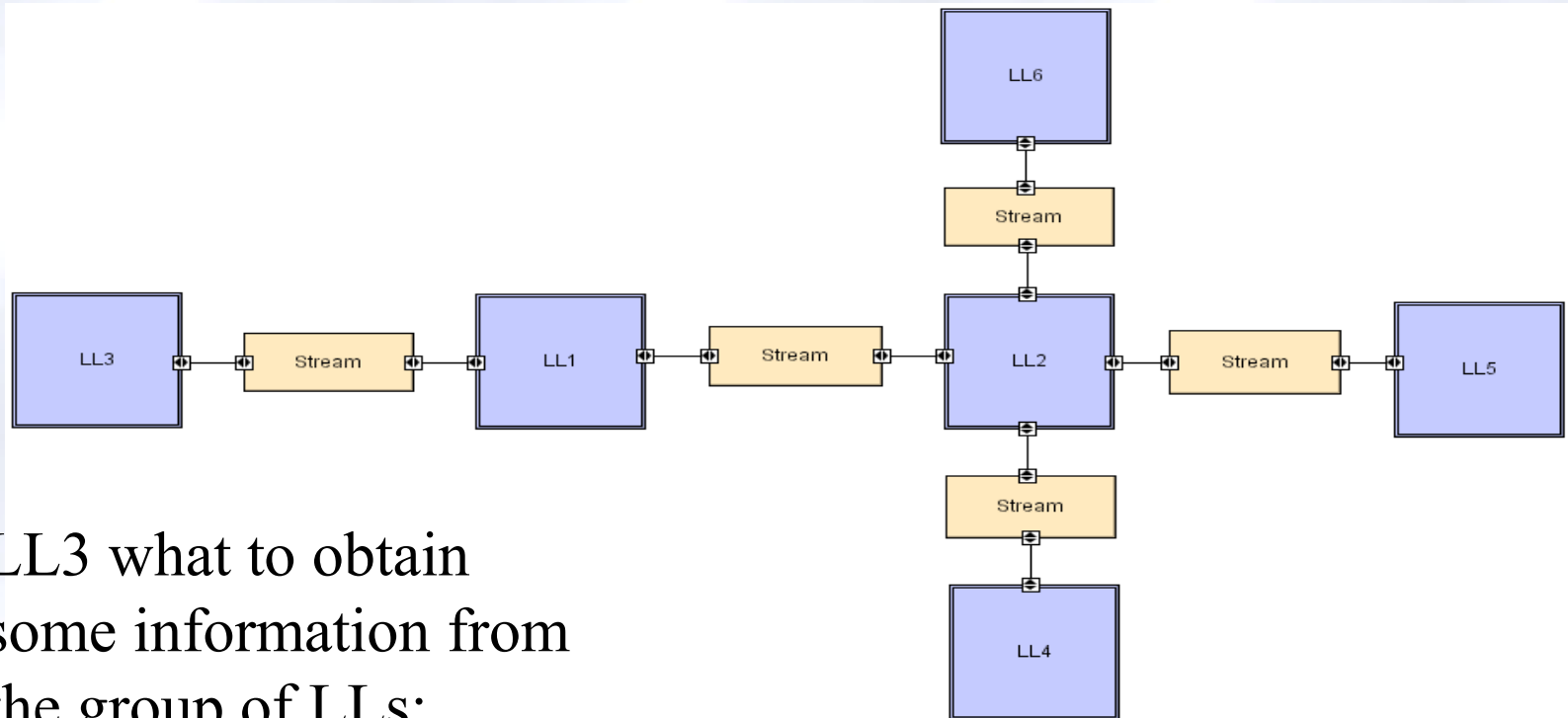
- State and behavior are **distributed** among peers which can **act as either clients or servers**.
 - Peers: **independent (autonomous)** components, having their own state and control thread.
 - Connectors: **Network** protocols, often custom.
 - Data Elements: Network messages
-
- State and logic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

Peer-to-Peer Style (cont'd)

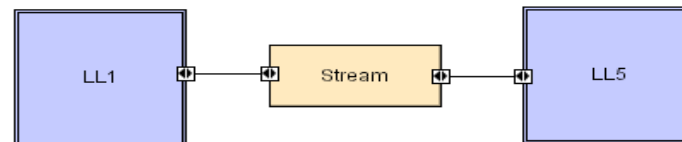
- Topology: Network (may have redundant connections between peers); can vary arbitrarily and dynamically
- Supports **decentralized computing** with flow of control and resources distributed among peers.
- Highly **robust** in the face of failure of any given node.
- **Scalable** in terms of access to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

But caution on the protocol!

Peer-to-Peer LL



LL3 what to obtain
some information from
the group of LLs:
LL3->LL1->LL2->LL5-
>LL2->LL1->LL3



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
 - Publish-subscribe
 - Event-based
- Peer-to-peer
- **"Derived" styles**
 - C2**
 - CORBA**

The C2 Style

- A general **multi-tier** architectural style
Asynchronous message-based integration of **independent** components
- Based on past experience with the ***model-view-controller*** design pattern
Generalizes architectures for user interface management systems
- Notable feature: ***flexible connectors***
Accommodate arbitrary numbers of components
Facilitate runtime architectural change

The C2 Style

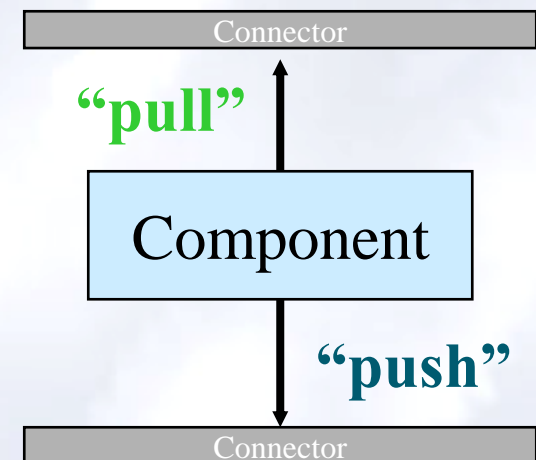
- Principle of *substrate independence*
A component need not know anything about components *beneath* it
We'll define the layering rules and the notion of "beneath" shortly ...
- Rich design environment for composition & analysis
- **Autonomous** components
own thread and non-shared address space
- All communication is by **asynchronous events which must pass through a connector**
connectors may be complex and powerful
- Some additional rules to promote **reuse**

The C2 Style

- **Asynchronous, event-based communication among autonomous components, mediated by active connectors**
- No component-component links
- Event-flow rules
- Hierarchical application

Notifications fall

Requests rise



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
 - Publish-subscribe
 - Event-based
- Peer-to-peer
- **“Derived” styles**
 - C2**
 - CORBA**

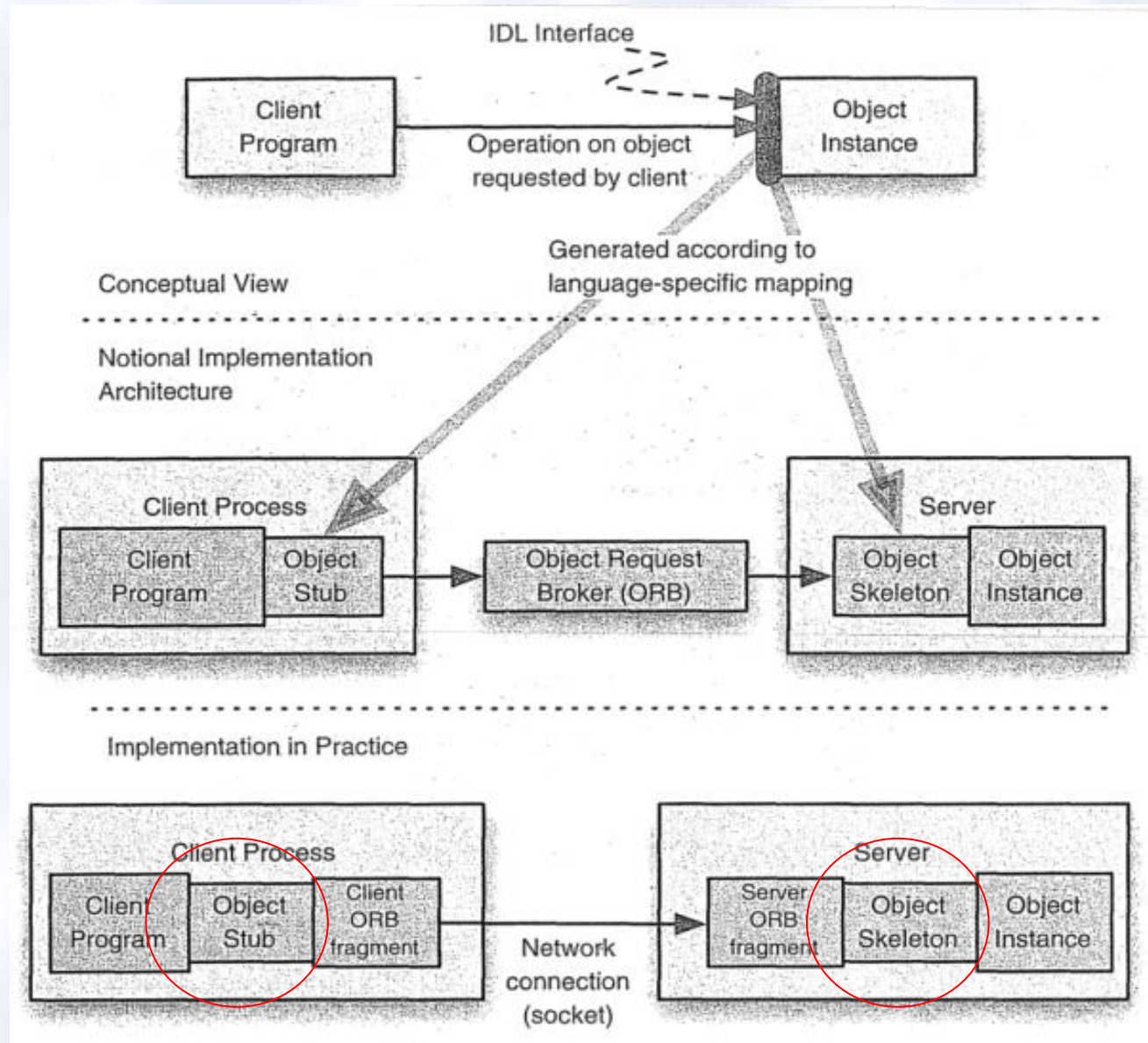
Distributed Objects

- The fundamental vocabulary comes, of course, from the simple **object-oriented style**.
- OO style is augmented with the **client-server style** to provide the notion of distributed objects, **with access to those objects from, potentially, different processes executing on different computers**.
- Objects are instantiated on different hosts, each exposing a public interface.
- The interface : all parameters and return values must be **serializable** so they can go over the network.
- Interaction between objects: **synchronous** procedure call; **asynchronous** forms such as in CORBA.

CORBA

- CORBA is a **standard** for implementing middleware that supports development of applications composed of distributed objects.
- The basic idea behind CORBA is that an application is broken up into objects, which are effectively software components that expose one or more provided **interfaces**.
- These provided interfaces are specified in terms of a programming language-and platform-neutral notation called the **Interface Definition Language (IDL)**.

CORBA



CORBA

- Remote vs. local procedure call

Remote call: all data must be serializable.

Remote call suffers from a much wider variety of potential failures than local calls.

Microservice

<https://www.martinfowler.com/microservices/>