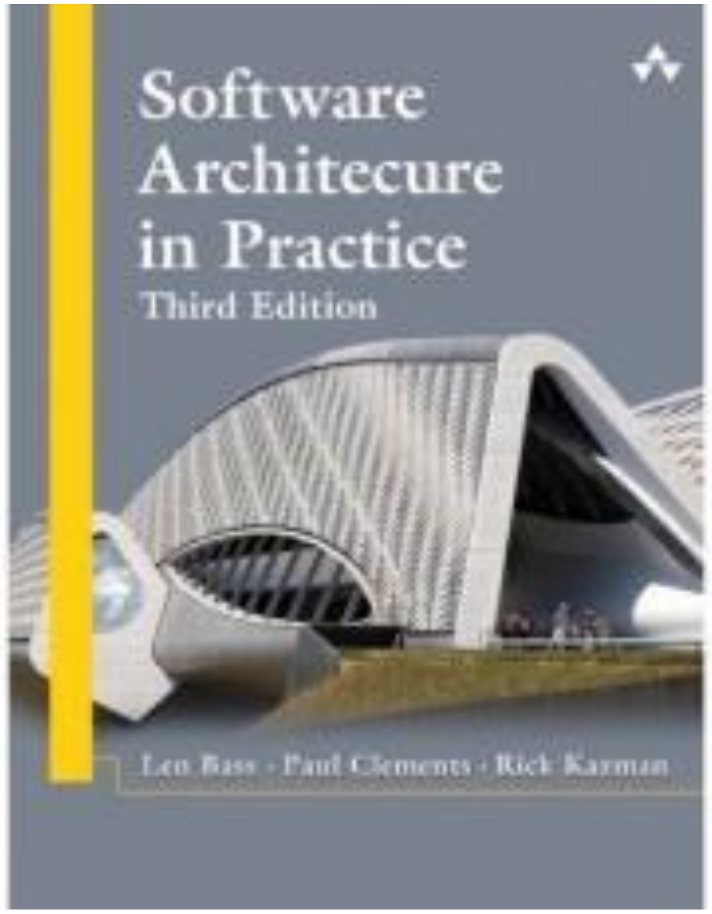
SE 4352
Software Architecture and Design

Fall 2018

Module 5

Chapter 13


Software
Architecure
in Practice
Third Edition
Len Bass · Paul Clements · Rick Kazman

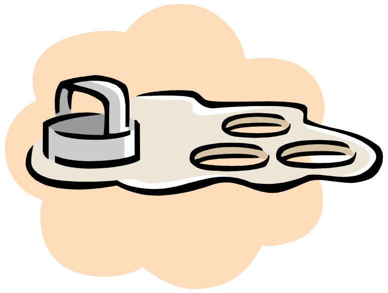
Recall…
Describing architectures involves…

- Components
- Connectors
  - relationships and interactions
- Constraints

Architectural Styles

- Hallmark of architectural design is use of idiomatic patterns or system organization
- Pattern=Architectural Style
- Developed over years
- Rich space
- Choices have tradeoffs
- Patterns are used in many engineering disciplines
- No comprehensive taxonomy

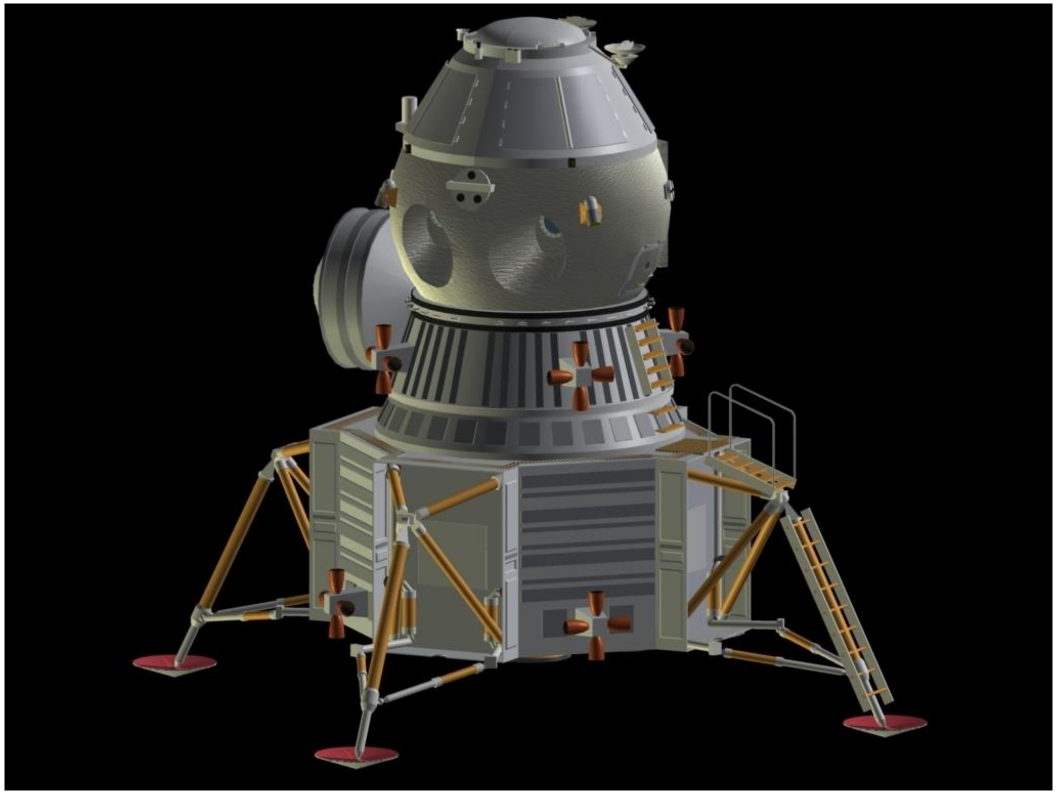
What is a Pattern?



An architectural pattern establishes a relationship between:

- *A context*. A recurring, common situation in the world that gives rise to a problem.
- *A problem.* The problem, appropriately generalized, that arises in the given context.
- *A solution.* A successful architectural resolution to the problem, appropriately abstracted. The solution for a pattern is determined and described by:
  - A set of element types (for example, data repositories, processes, and objects)
  - A set of interaction mechanisms or connectors (for example, method calls, events, or message bus)
  - A topological layout of the components
  - A set of semantic constraints covering topology, element behavior, and interaction mechanisms

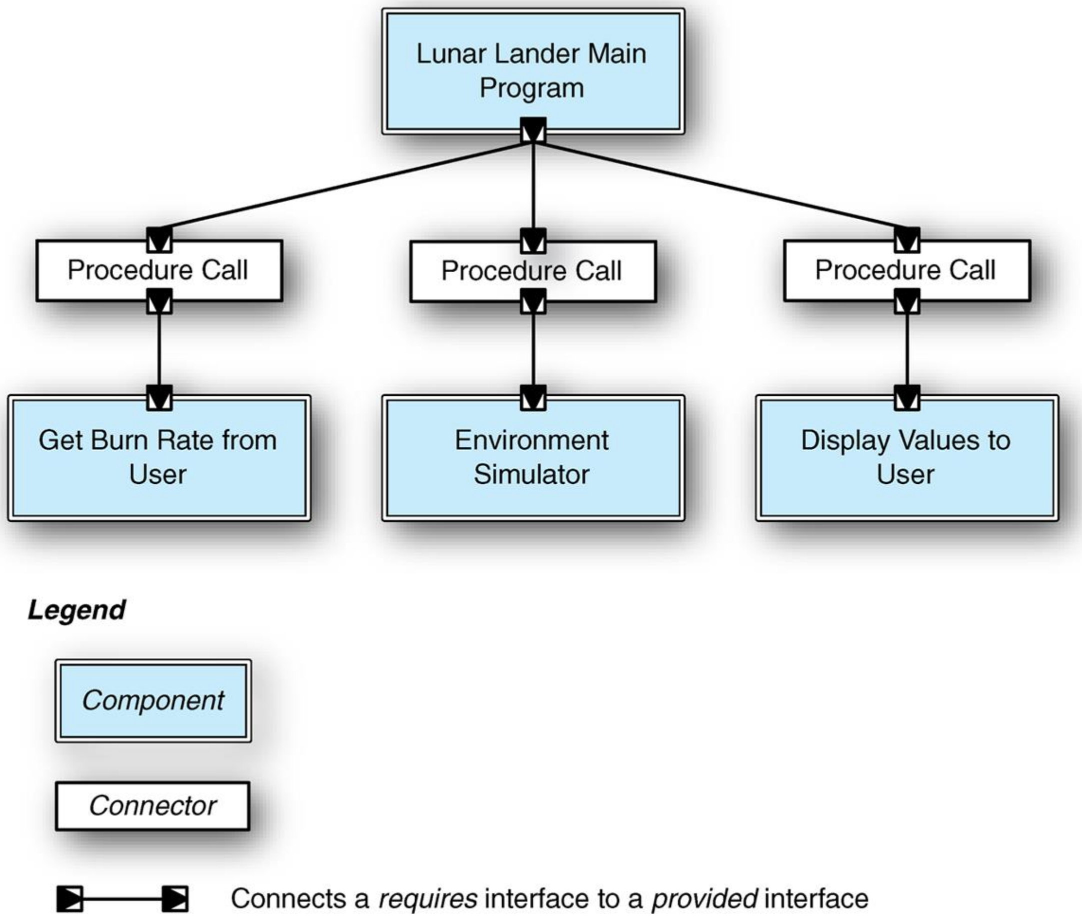
Example: Lunar Lander Game



Lunar Lander Game

- *Traditional Language-Influenced Styles* –
  Main program and subroutines, Object oriented
- *Layered* - Client server
- *Dataflow Styles* - Pipe and filter
- *Shared Memory* - Blackboard
- *Implicit Invocation* - Publish-subscribe
- *Peer to Peer*

Lunar Lander Game

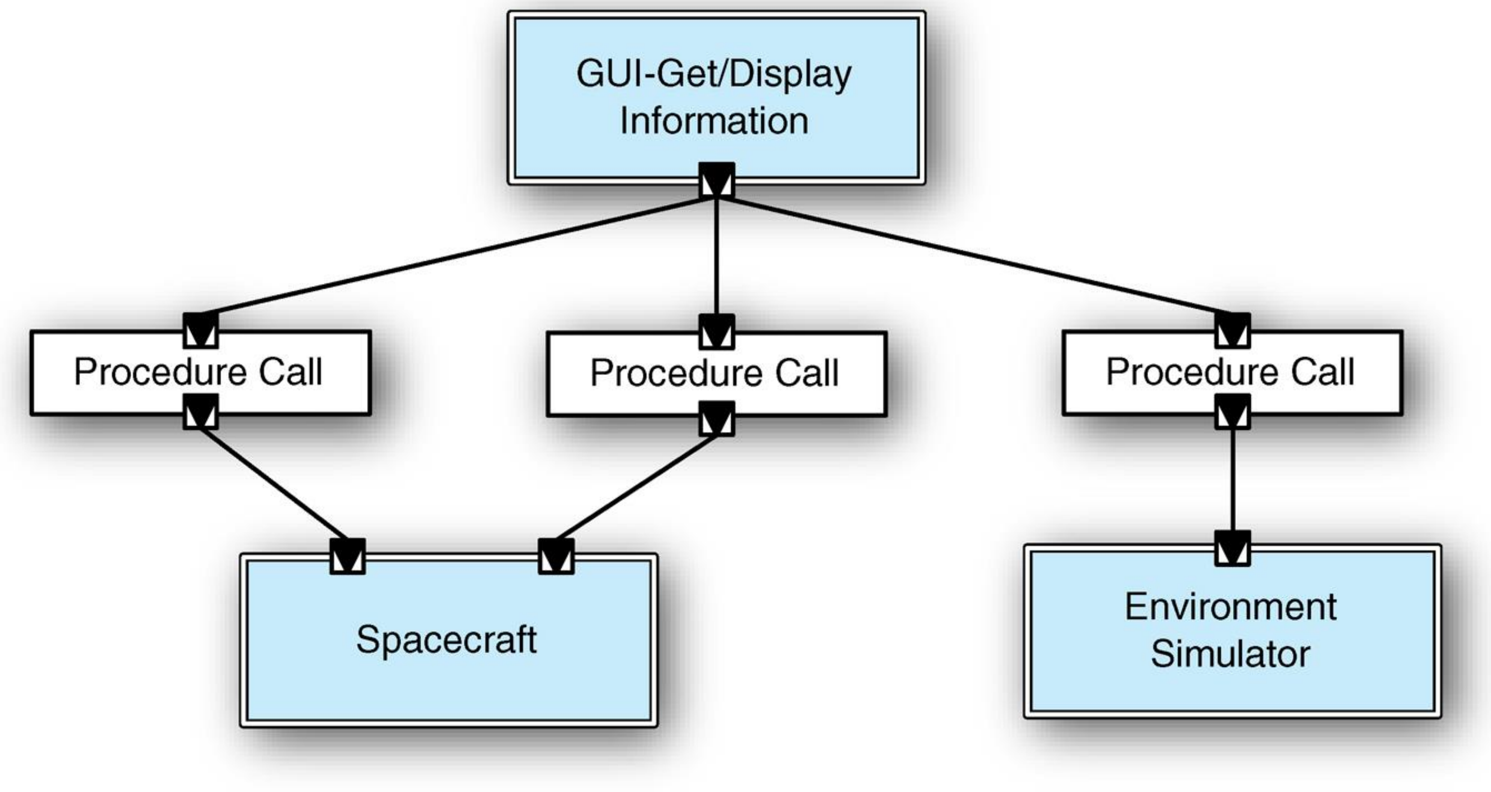


Main Program and Subroutines

- Components: Main and subroutines
- Connectors: Function/Procedure Calls
- Constraints: No shared memory (e.g. global variables)

- Data elements are values passed in/out of subroutines
- Traditional style programming
- Static organization of components is hierarchical
- Full structure becomes a directed graph
- Result is modularity
- Subroutines can be replaced if interface semantics are the same
- Typical use is for small programs
- Typically fails to scale to large applications

Example: Lunar Lander Game

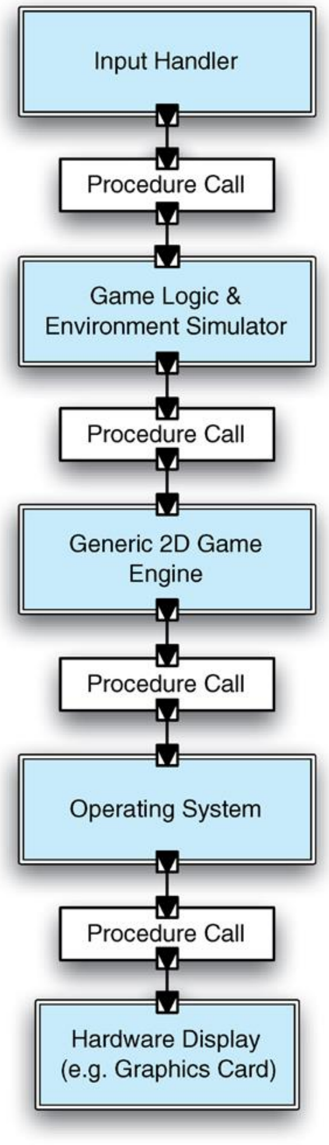


Object-Oriented

- Components: Objects (instances of a class)
- Connectors: Method invocation (procedure calls to manipulate state)
- Constraints: Objects must be instantiated before the objects' methods can be called; single-threaded; shared memory

- Data elements are arguments to methods
- State strongly encapsulated with functions that operate on that state as objects
- Provides for integrity of data operations; data manipulated only by appropriate functions; implementation details hidden
- Typically used when close correlation between entities in the physical world and entities in the program is desired
- Good for applications involving complex, dynamic data structures
- Use in distributed applications requires extensive middleware to provide access to remote objects
- Relatively inefficient for high-performance applications with large, regular numeric data structures (e.g. scientific computing)
- Lack of additional structuring principles can result in highly complex applications

Example: Lunar Lander Game



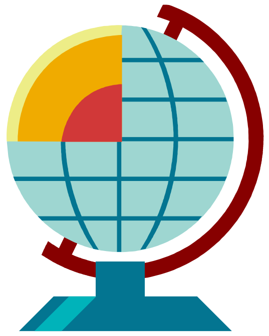
Layered Architecture

- Components: Layers offering a set of services to other layers, typically comprising several programs (subcomponents)
- Connectors: Typically procedure calls
- Constraints: programs at a given layer may only access the services provide by the layer immediately below it

- Data elements are parameters passed between layers
- Layered architecture consisting of an ordered sequence of layers
- Layer offers a set of services ("machine with a bunch of buttons and knobs") – termed the "provides interface" of the layer
- Services within a layer implemented by various programs
- Commonly used in operating systems and network protocol stacks
- Dependence Structure making software at upper levels immune to changes of implementation within lower levels as long as the service specifications are invariant; software at lower levels fully independent of upper levels
- Can be inefficient but efficiency improved if layers allowed to access all layers below it

Layer Pattern

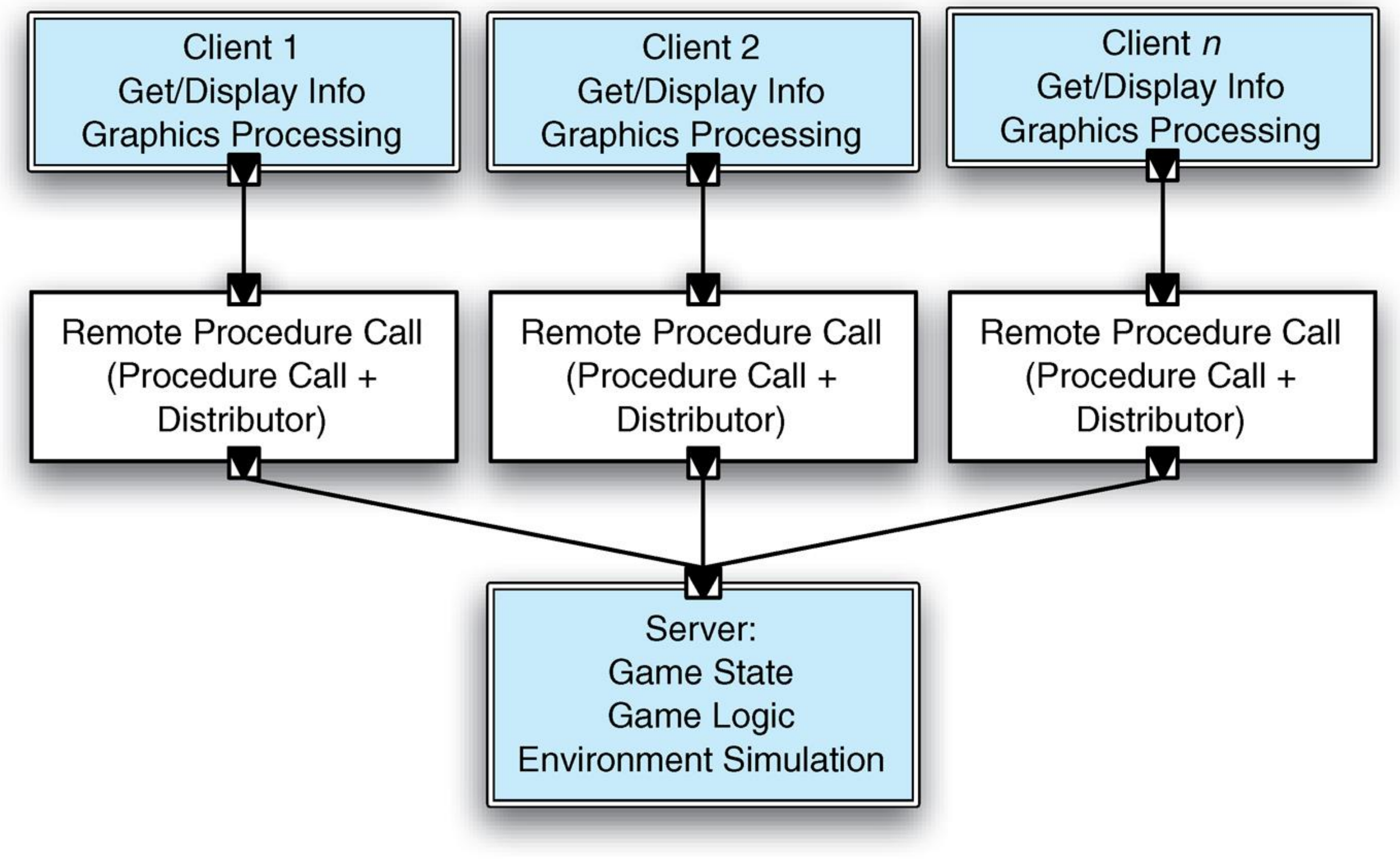
- **Context:** All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.
- **Problem:** The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.
- **Solution:** To achieve this separation of concerns, the layered pattern divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive set of services. The usage must be unidirectional. Layers completely partition a set of software, and each partition is exposed through a public interface.



Layer Pattern Solution

- Overview: The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers.
- Elements: *Layer*, a kind of module. The description of a layer should define what modules the layer contains.
- Relations: *Allowed to use.* The design should define what the layer usage rules are and any allowable exceptions.
- Constraints:
  - Every piece of software is allocated to exactly one layer.
  - There are at least two layers (but usually there are three or more).
  - The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).
- Weaknesses:
  - The addition of layers adds up-front cost and complexity to a system.
  - Layers contribute a performance penalty.

Example: Lunar Lander Game



Client-Server

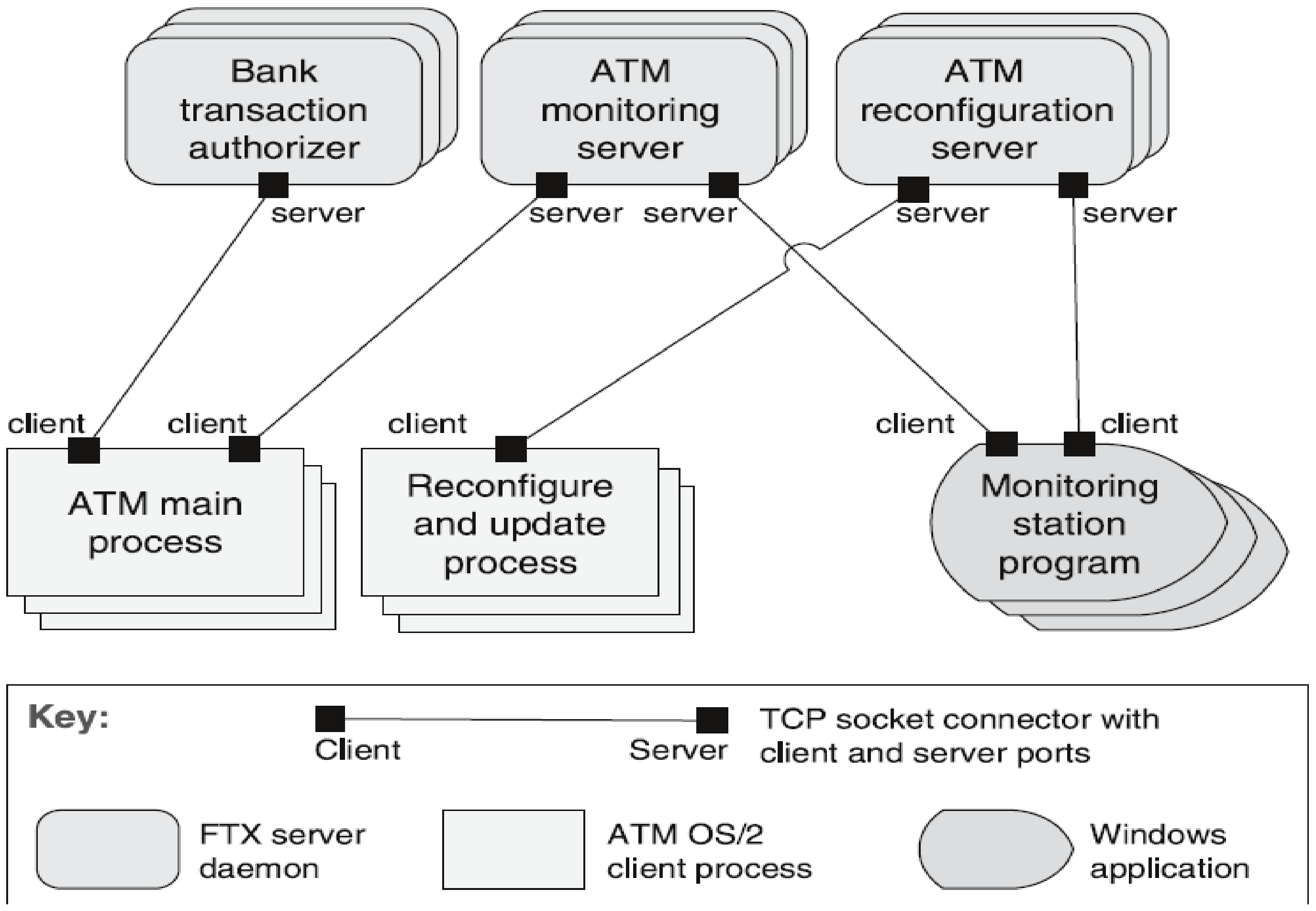
- <u>Components:</u> Clients and Servers
- <u>Connectors:</u> Remote Procedure Calls, network protocols
- <u>Constraints:</u> Client to client communication prohibited

- Data elements are parameters and return values sent by the connections
- 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.
- Two level with multiple clients making requests
- Centralization of computation and data at server, with information made available to remote clients
- Typically used when centralization of data is required or where processing and data storage benefit from a high-capacity machine, and where clients primarily perform simple user interface tasks
- When network bandwidth is limited and there are a large number of client requests, performance quickly degrades

Client-Server Pattern

- **Context:** There are shared resources and services that large numbers of distributed clients wish to access, and for which we wish to control access or quality of service.
- **Problem:** By managing a set of shared resources and services, we can promote modifiability and reuse, by factoring out common services and having to modify these in a single location, or a small number of locations. We want to improve scalability and availability by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers.
- **Solution:** Clients interact by requesting services of servers, which provide a set of services. Some components may act as both clients and servers. There may be one central server or multiple distributed ones.

Client-Server Example



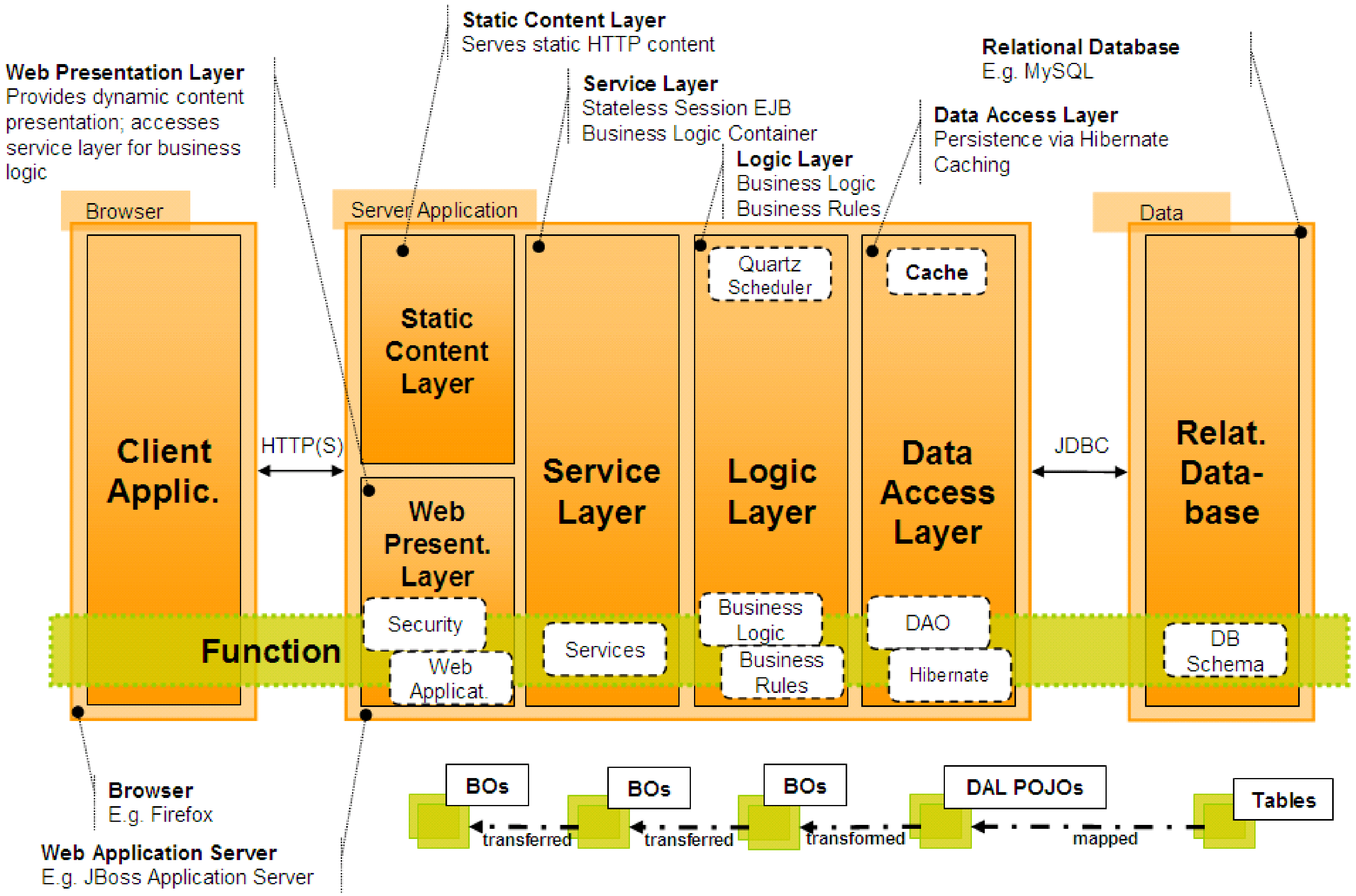
Client-Server Solution - 1

- Overview: Clients initiate interactions with servers, invoking services as needed from those servers and waiting for the results of those requests.
- Elements:
  - *Client,* a component that invokes services of a server component. Clients have ports that describe the services they require.
  - *Server:* a component that provides services to clients. Servers have ports that describe the services they provide.
- *Request/reply connector:* a data connector employing a request/reply protocol, used by a client to invoke services on a server. Important characteristics include whether the calls are local or remote, and whether data is encrypted.

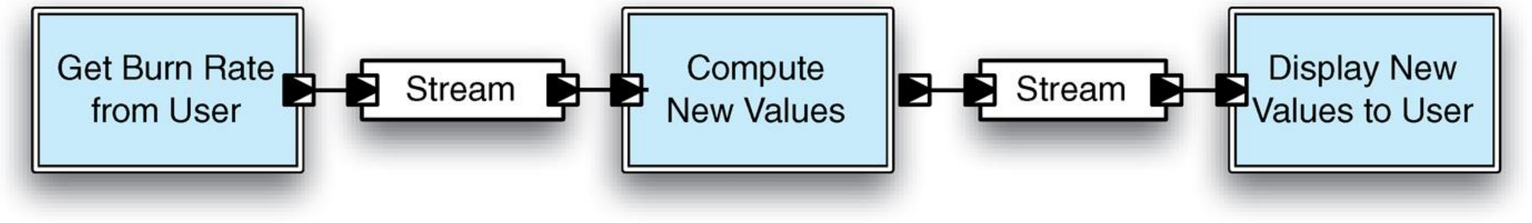
Client-Server Solution- 2

- Relations: The *attachment* relation associates clients with servers.
- Constraints:
  - Clients are connected to servers through request/reply connectors.
  - Server components can be clients to other servers.
- Weaknesses:
  - Server can be a performance bottleneck.
  - Server can be a single point of failure.
  - Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.

Example of a more complex



Example: Lunar Lander Game



Pipe-and-Filter

- Components: Independent programs, known as filters
- Connectors: Explicit routers of data streams; service provided by operating system
- Constraints: data must be linear data streams

- Data elements are not explicit
- Separate programs are executed, potentially concurrently; data is passed as a stream from one program to another
- Filters are mutually independent
- Simple structure of incoming and outgoing data streams facilitates novel combinations of filters for new, composed applications
- Typically ubiquitous in operating system application programming
- No interaction between programs
- Not possible to exchange complex data structures between filters
- Prevalent in Unix shells

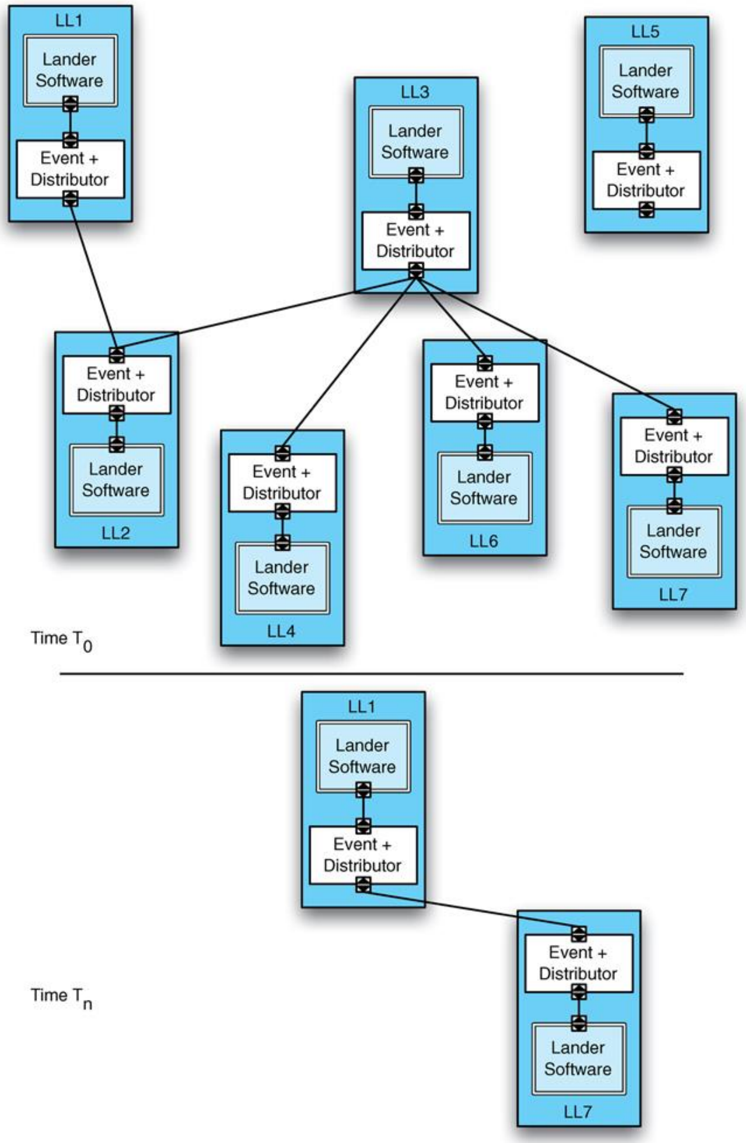
Pipe and Filter Pattern

- **Context:** Many systems are required to transform streams of discrete data items, from input to output. Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.
- **Problem:** Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other. The components, being generic and loosely coupled, are easily reused. The components, being independent, can execute in parallel.
- **Solution:** The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data. Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter. A single filter can consume data from, or produce data to, one or more ports.

Pipe and Filter Solution

- Overview: Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.
- Elements:
  - *Filter,* which is a component that transforms data read on its input port(s) to data written on its output port(s).
  - *Pipe,* which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through.
- Relations: The *attachment* relation associates the output of filters with the input of pipes and vice versa.
- Constraints:
  - Pipes connect filter output ports to filter input ports.
  - Connected filters must agree on the type of data being passed along the connecting pipe.

Example: Lunar Lander Game



Peer-to-Peer

- Components: Peers—independent components, having their own state and control thread
- Connectors: Network protocols
- Constraints:

- Data elements are network messages
- State and behavior are distributed among peers that can act as either clients or servers
- Network topology that might include redundancy
- 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 computer power
- Typically used where sources of information and operations are distributed and network is ad hoc
- Not recommended when information retrieval is time critical as protocol imposes latency
- Security considerations must be addressed

Peer-to-Peer Pattern

- **Context:** Distributed computational entities—each of which is considered equally important in terms of initiating an interaction and each of which provides its own resources—need to cooperate and collaborate to provide a service to a distributed community of users.
- **Problem:** How can a set of “equal” distributed computational entities be connected to each other via a common protocol so that they can organize and share their services with high availability and scalability?
- **Solution:** In the peer-to-peer (P2P) pattern, components directly interact as peers. All peers are “equal” and no peer or group of peers can be critical for the health of the system. Peer-to-peer communication is typically a request/reply interaction without the asymmetry found in the client-server pattern.

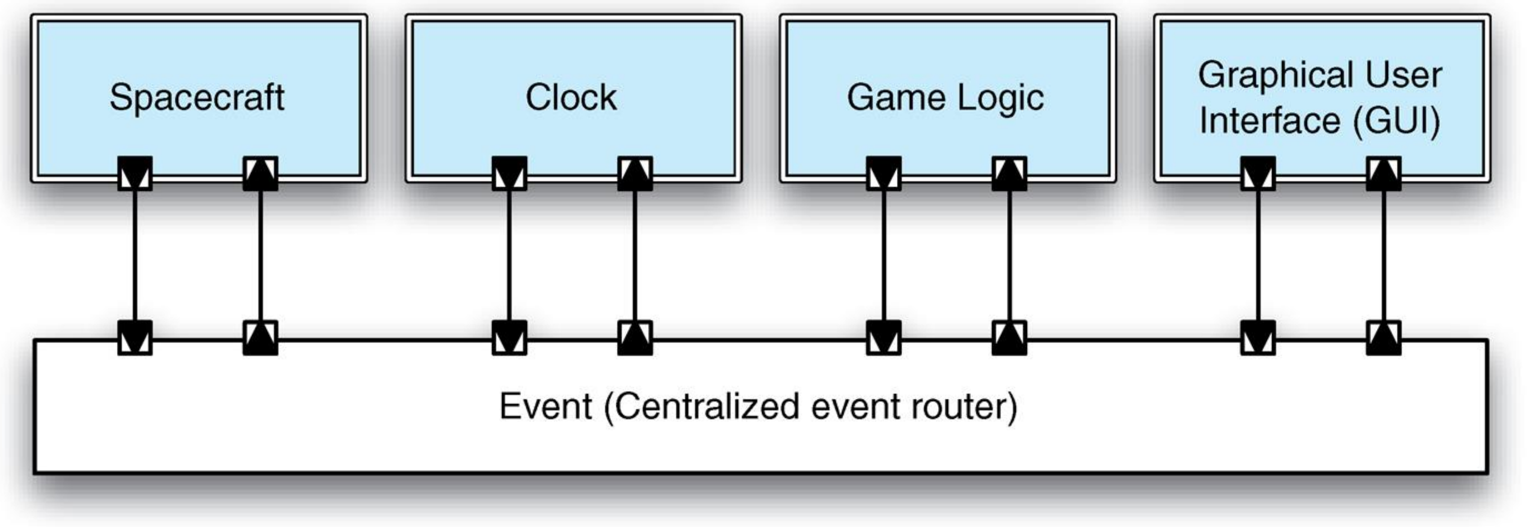
Peer-to-Peer Solution - 1

- Overview: Computation is achieved by cooperating peers that request service from and provide services to one another across a network.
- Elements:
  - *Peer,* which is an independent component running on a network node. Special peer components can provide routing, indexing, and peer search capability.
  - *Request/reply connector,* which is used to connect to the peer network, search for other peers, and invoke services from other peers. In some cases, the need for a reply is done away with.
- Relations: The relation associates peers with their connectors. Attachments may change at runtime.

Peer-to-Peer Solution - 2

- Constraints: Restrictions may be placed on the following:
  - The number of allowable attachments to any given peer
  - The number of hops used for searching for a peer
  - Which peers know about which other peers
  - Some P2P networks are organized with star topologies, in which peers only connect to supernodes.
- Weaknesses:
  - Managing security, data consistency, data/service availability, backup, and recovery are all more complex.
  - Small peer-to-peer systems may not be able to consistently achieve quality goals such as performance and availability.

Example: Lunar Lander Game


Spacecraft
Clock
Game Logic
Graphical User Interface (GUI)
Event (Centralized event router)

Publish-Subscribe

- <u>Components:</u> Publishers, subscribers, proxies for managing distribution
- <u>Connectors:</u> Procedure Calls, network protocols
- <u>Constraints:</u>

- Data elements are subscriptions, notifications, published information
- Subscribers register/deregister to receive specific message or specific content
- Publishers maintain a subscription list and broadcast messages to subscribers either synchronously or asynchronously
- Subscribers connect to publishers either directly or may receive notification via a network protocol from intermediaries
- Highly efficient one-way dissemination
- Often used for multiplayer network based games and graphical information display
- If large number of subscribers for single data item is very large, specialzed protocol might be necessary

Publish-Subscribe Pattern

- **Context:** There are a number of independent producers and consumers of data that must interact. The precise number and nature of the data producers and consumers are not predetermined or fixed, nor is the data that they share.
- **Problem:** How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers so they are unaware of each other's identity, or potentially even their existence?
- **Solution:** In the publish-subscribe pattern, components interact via announced messages, or events. Components may subscribe to a set of events. Publisher components place events on the bus by announcing them; the connector then delivers those events to the subscriber components that have registered an interest in those events.

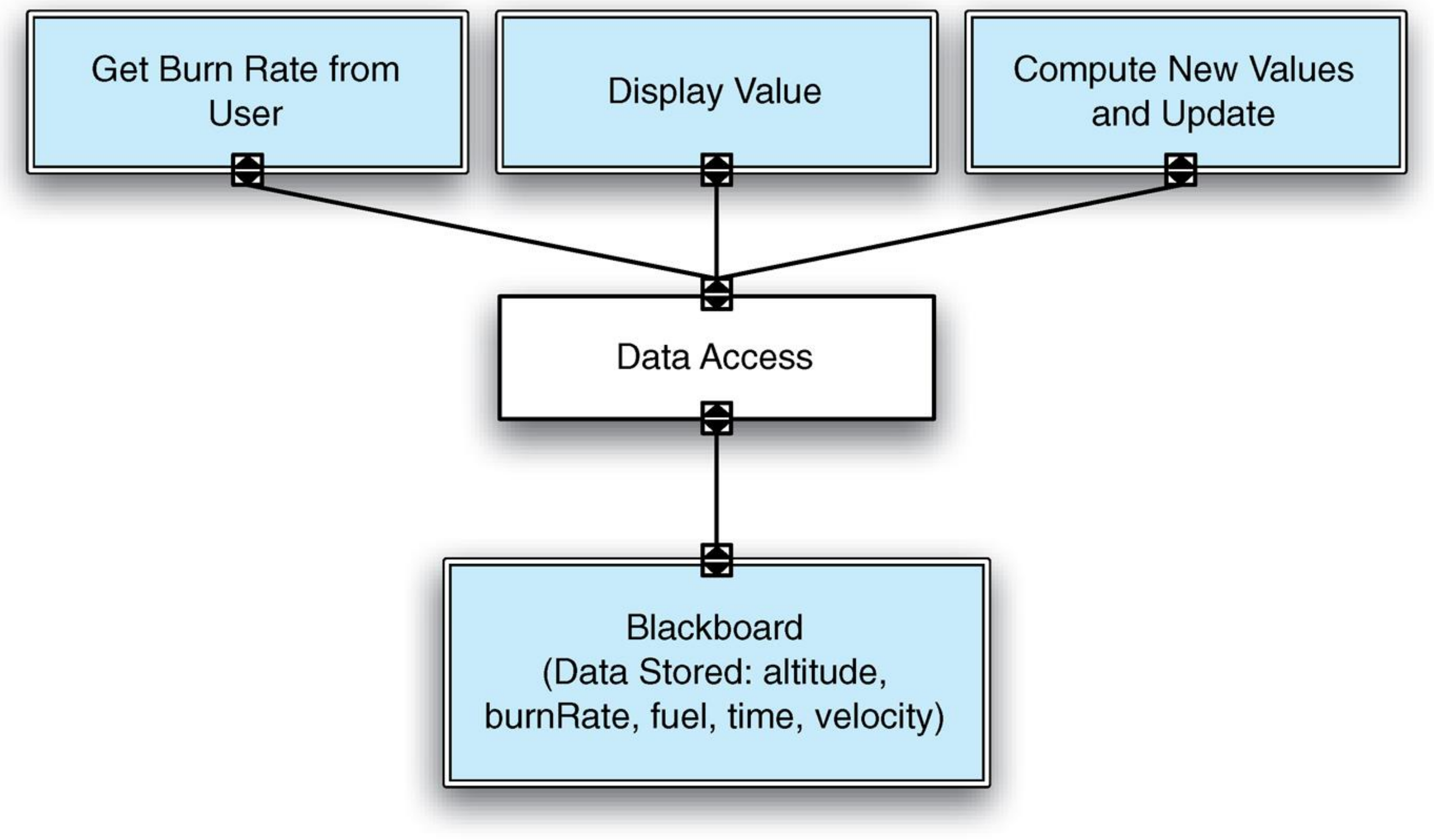
Publish-Subscribe Solution – 1

- Overview: Components publish and subscribe to events. When an event is announced by a component, the connector infrastructure dispatches the event to all registered subscribers.
- Elements:
  - *Any C&C component* with at least one publish or subscribe port.
  - *The publish-subscribe connector*, which will have *announce* and *listen* roles for components that wish to publish and subscribe to events.
- Relations: The *attachment* relation associates components with the publish-subscribe connector by prescribing which components announce events and which components are registered to receive events.

Publish-Subscribe Solution - 2

- Constraints: All components are connected to an event distributor that may be viewed as either a bus—connector—or a component. Publish ports are attached to announce roles and subscribe ports are attached to listen roles.
- Weaknesses:
  - Typically increases latency and has a negative effect on scalability and predictability of message delivery time.
  - Less control over ordering of messages, and delivery of messages is not guaranteed.

Example: Lunar Lander Game


Get Burn Rate from User
Display Value
Compute New Values and Update
Data Access
Blackboard
(Data Stored: altitude, burnRate, fuel, time, velocity)

Blackboard

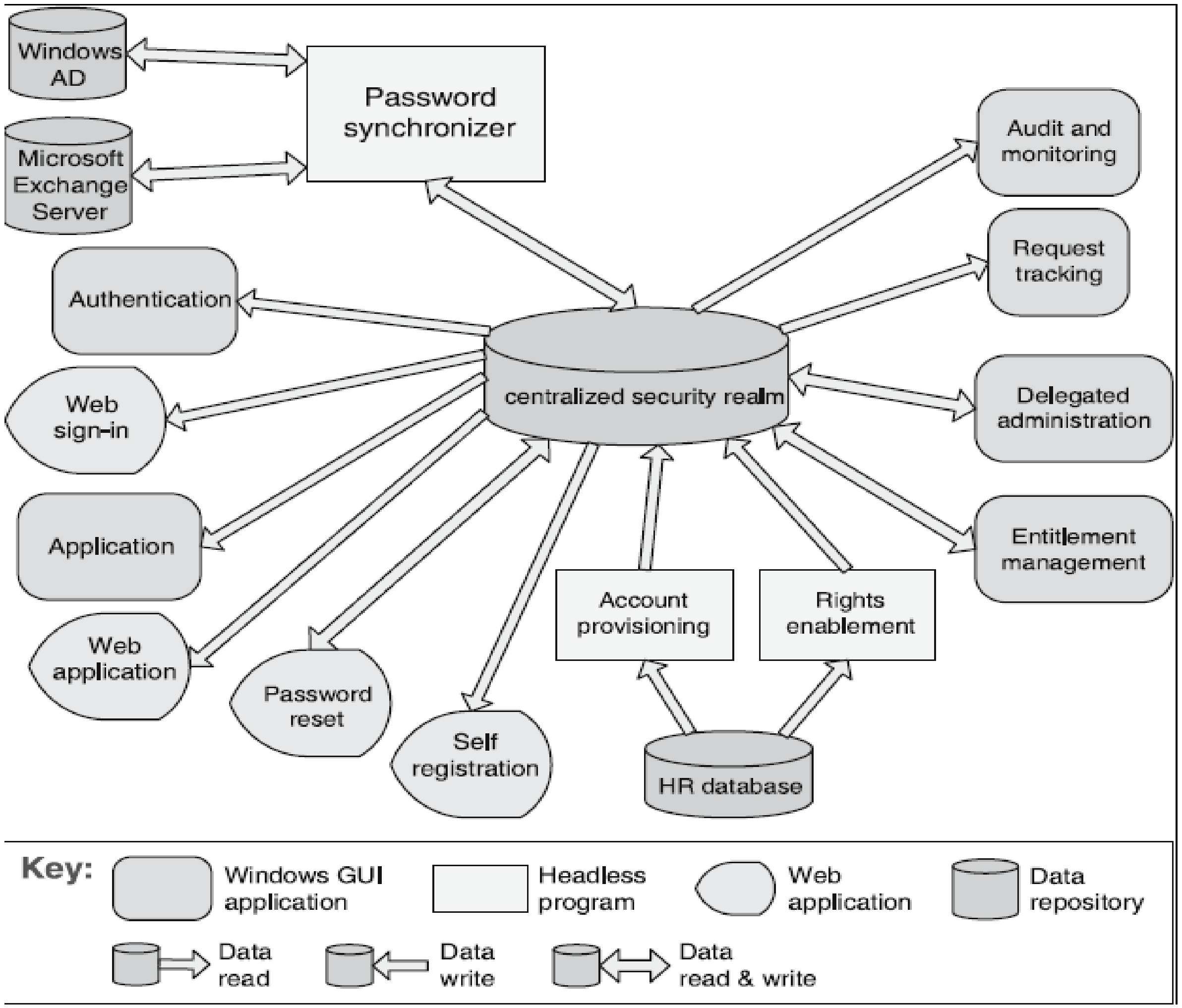
- Components: Independent programs, sometimes referred to as “knowledge sources”
- Connectors: Access to the blackboard may be by direct memory reference or can be through procedure call or database query
- Constraints: Regulation between independent programs not automatic

- Data elements are stored in the blackboard
- Independent programs access and communicate exclusively through a global data repository, known as a blackboard
- Star topology with blackboard at the center
- Some versions may poll the blackboard to determine if any values of interest have changed while others notify interested components to updates to the blackboard
- Complete solution strategies to complex problems do not have to be preplanned
- Evolving views of the data/problem determine the strategies that are adopted
- Typically used in heuristic problem solving in AI applications
- Not applicable when a well-structured solution strategy is available

Shared-Data Pattern

- **Context:** Various computational components need to share and manipulate large amounts of data. This data does not belong solely to any one of those components.
- **Problem:** How can systems store and manipulate persistent data that is accessed by multiple independent components?
- **Solution:** In the shared-data pattern, interaction is dominated by the exchange of persistent data between multiple *data accessors* and at least one *shared-data store*. Exchange may be initiated by the accessors or the data store. The connector type is *data reading and writing*.

Shared Data Example



Shared Data Solution - 1

- Overview: Communication between data accessors is mediated by a shared data store. Control may be initiated by the data accessors or the data store. Data is made persistent by the data store.
- Elements:
  - *Shared-data store.* Concerns include types of data stored, data performance-oriented properties, data distribution, and number of accessors permitted.
  - *Data accessor component*.
  - *Data reading and writing connector*.

Shared Data Solution - 2

- Relations: *Attachment* relation determines which data accessors are connected to which data stores.
- Constraints: Data accessors interact only with the data store(s).
- Weaknesses:
  - The shared-data store may be a performance bottleneck.
  - The shared-data store may be a single point of failure.
  - Producers and consumers of data may be tightly coupled.

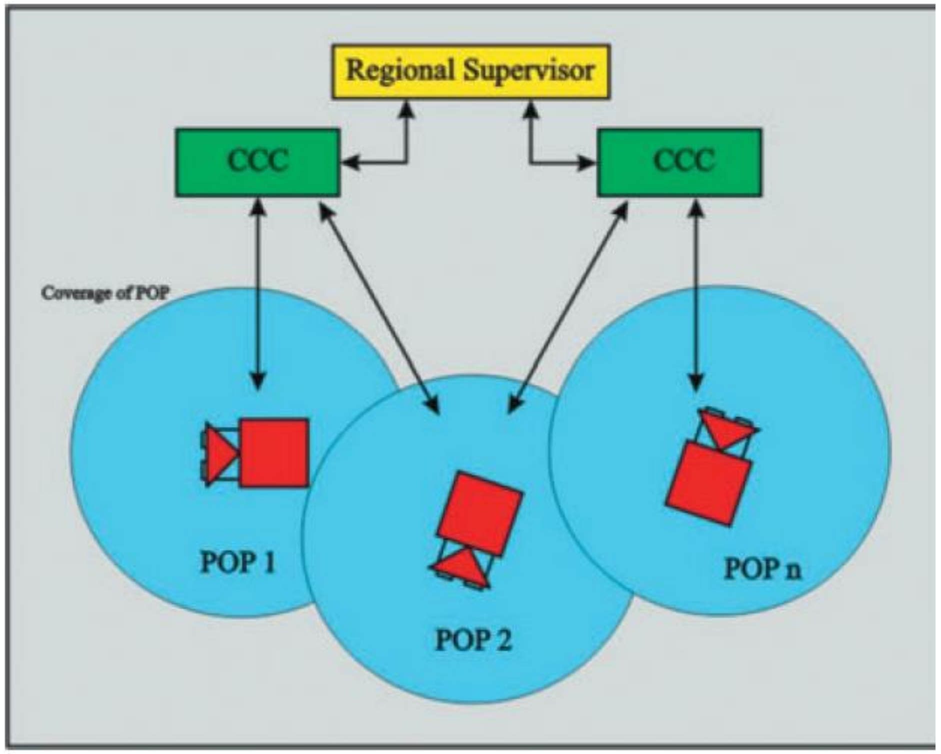
Example

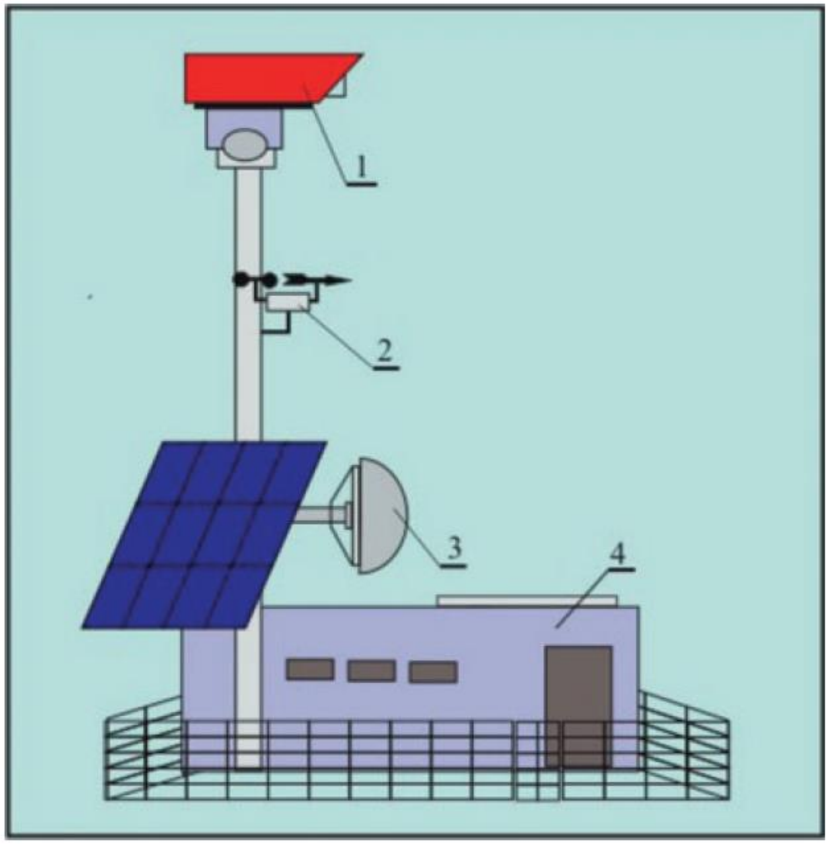
- Consider an automated teller machine (ATM) system. Brainstorm any one architectural style for the system.

Example

Consider a forest fire detection system. Brainstorm any one architectural style for the system.

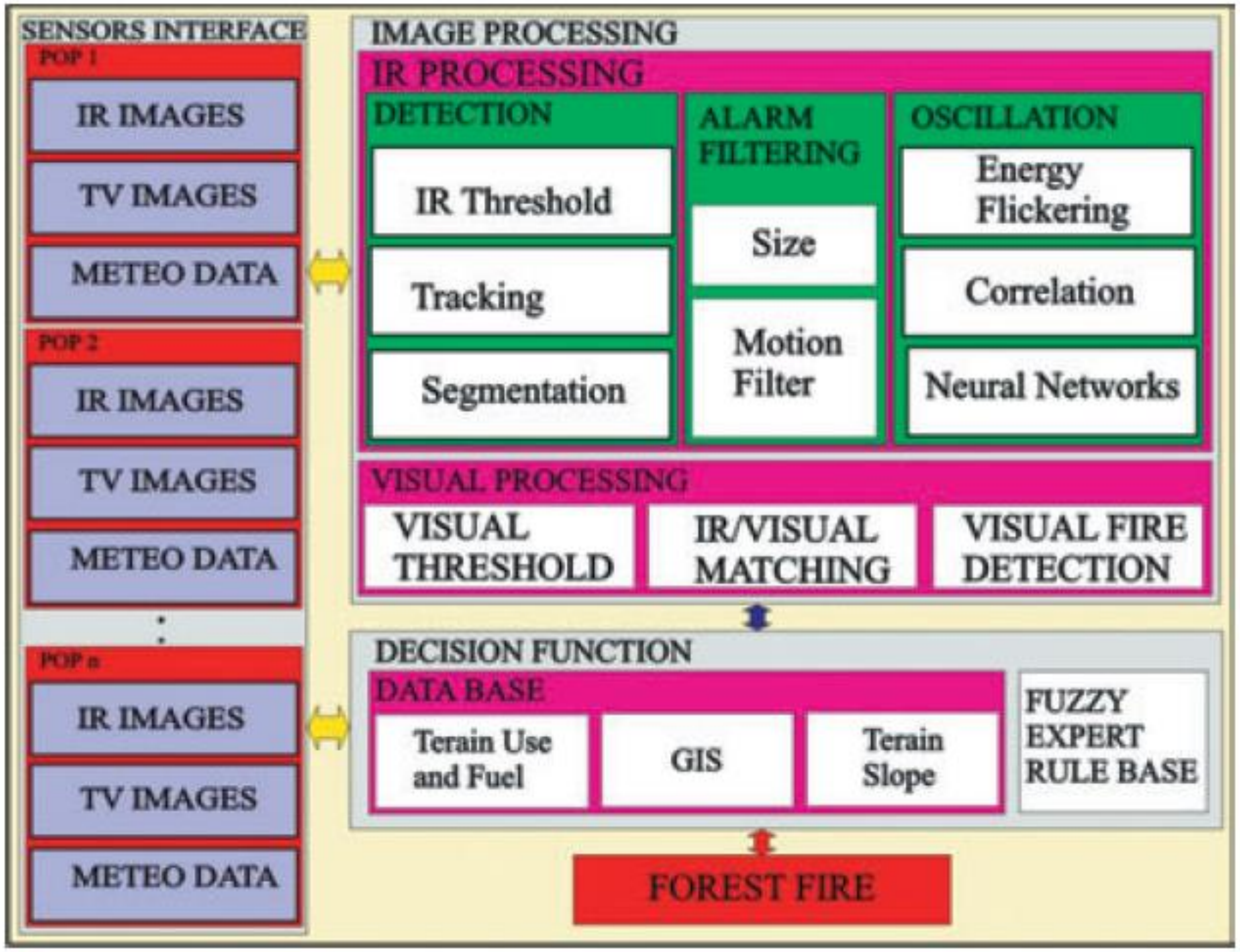
Example





Peripheral observation point.

Example



The block scheme of the detection system.