A06. *More on Architecture Styles*

2014

Sungwon Kang

Table of Contents

Acknowledgment Much of the contents of 1, 2, and 3 is based on [Taylor 09] R. N. Taylor, N. Medvidovic, E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*, Wiley, 2009.

1. Comparison Table of Architectural Styles

Style Category & Name	Summary	Use It When . . .	Avoid It When . . .
Language-influenced styles LN			
Main program and subroutines	Main program controls program execution, calling multiple subroutines.	. . . application is small and simple.	. . . complex data structures needed (because of lack of encapsulation). . . . 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. . . . very high performance required.

Style Category & Name	Summary	Use It When . . .	Avoid It When . . .
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.
Dataflow 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, aka 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.

Style Category & Name	Summary	Use It When . . .	Avoid It When . . .
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. . . . 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 Category & Name	Summary	Use It When . . .	Avoid It When . . .
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.	. . . it is more efficient to move processing to a data set than the data set to processing. . . . it is desirous to dynamically customize a local processing node through inclusion of external code.	. . . security of mobile code cannot be assured, or sandboxed. . . . tight control of versions of deployed software is required.
Implicit Invocation			
Publish-subscribe	Publishers broadcast messages to subscribers.	. . . components are very loosely coupled. . . . subscription data is small and efficiently transported.	. . . 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.

Style Category & Name	Summary	Use It When . . .	Avoid It When . . .
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. . . . robustness in face of independent failures and high scalability required.	. . . trustworthiness of independent peers cannot be assured or managed. . . . designated nodes to support resource discovery unavailable.
More Complex Styles			
C2	Layered network of concurrent components communicating by events.	. . . independence from substrate technologies required. . . . heterogeneous applications. . . . support for product-lines desired.	. . . high-performance across many layers required. . . . multiple threads are inefficient.
Distributed objects	Objects instantiated on different hosts.	. . . objective is to preserve illusion of location-transparency.	. . . high overhead of supporting middleware is excessive. . . . network properties are unmaskable, in practical terms.

2. Simple Styles

2.1 Language-influenced

2.2 Layered

2.3 Dataflow

2.4 Shared Memory

2.5 Interpreter

2.6 Implicit Invocation

2.7 Peer-to-Peer

2.8 Other Styles

2.1 Traditional Language-Influenced Styles

- Languages like C, C++, Java and Pascal can be used to implement architecture in very different styles:
  - Main focuses
    - organizational relationships between components
    - Control-flow between components
- Traditional Language-Influenced Styles

  1) Main Program and Subroutines (MPS) Style

  2) Object-Oriented (OO) Style

1) MPS Style (1/3)

- C, Fortran, Pascal, Basic
- Components
  - The main program
  - The various subroutines(= procedures)
- Connectors
  - Procedure calls
- Design idea:
  - functional decomposition of the processing into repeated user interface steps (input and output)
    - all the processing that corresponds to simulating the environment and the actions of the spacecraft
- No shared memory

1) MPS Style (2/3)

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.
Qualities yielded: Modularity: Subroutines may be replaced with different implementations as 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.

1) MPS Style (3/3)

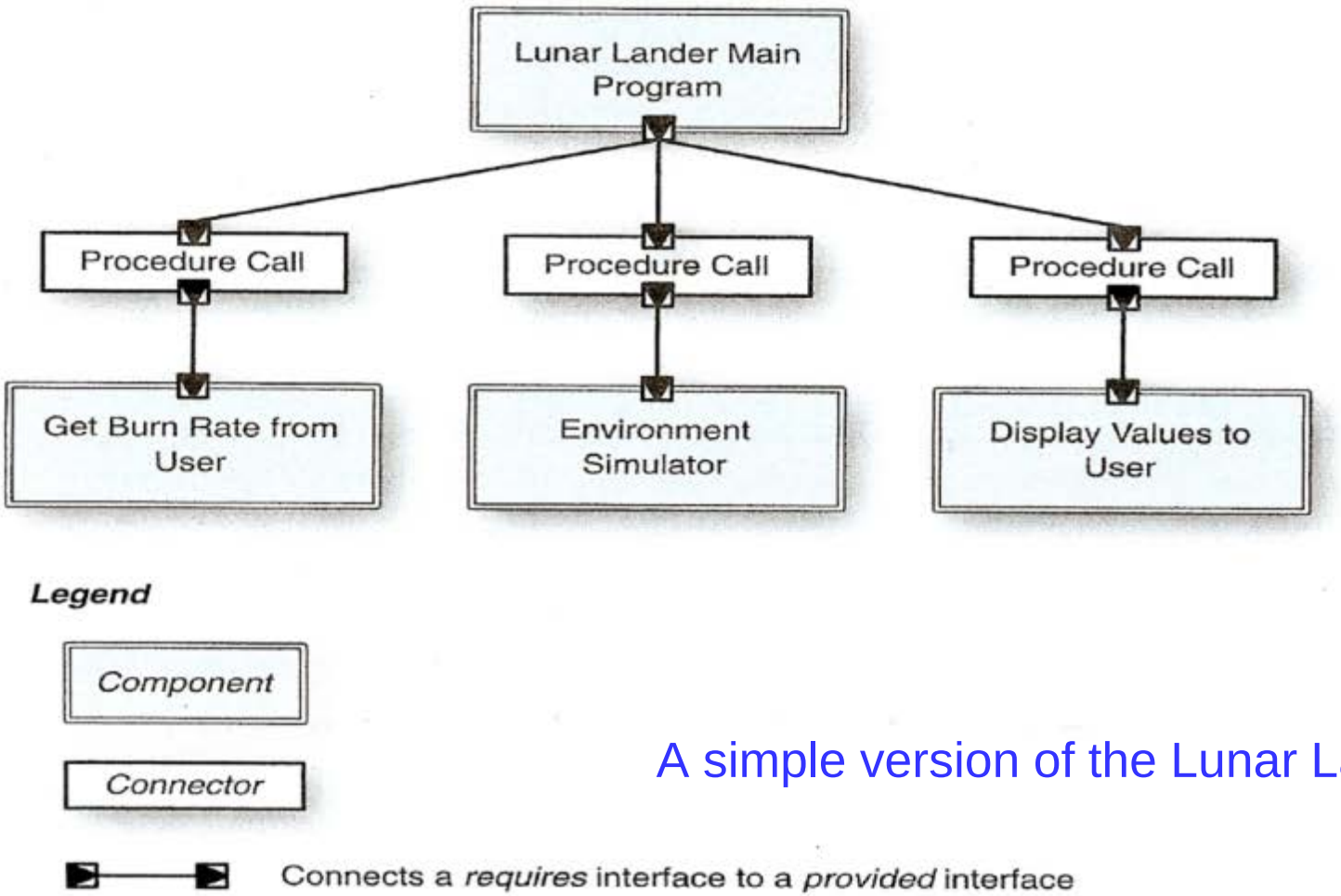


Figure 4-7. *Lunar Lander: Main program and subroutines.*

A simple version of the Lunar Lander program

2) OO Style (1/4)

- Similar to “Main Program and Subroutines”
- Difference
  - A world of objects whose lifetimes vary according to use
  - Need to understand the numerous static and dynamic relationships among the objects

Exercise: Design an OO style component and connector architecture for Lunar Lander program.

- Three encapsulations
  - Spacecraft
  - User’s interface
  - Environment: physics model that allows calculation of the descent rate

- Contrast with MPS
  - Interaction with the user (input and output) are handled by a single object

2) OO Style (2/4)

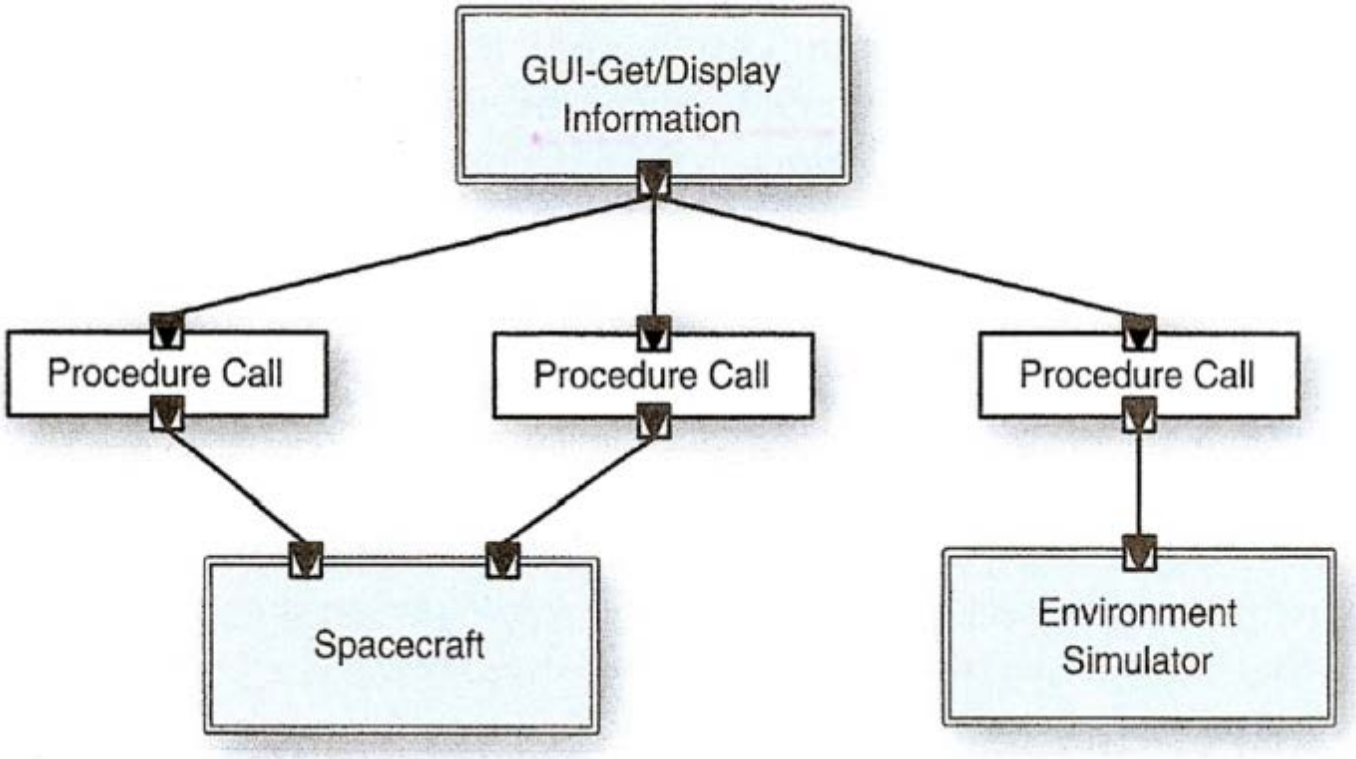
Summary: State strongly encapsulated with functions that operate on that state as objects. Objects must be instantiated before the objects' methods can be called.
Components: Objects (aka. instance of a class).
Connector: Method invocation (procedure calls to manipulate state).

Data elements: Arguments to methods.
Topology: Can vary arbitrarily; components may share data and interface functions through inheritance hierarchies.
Additional constraints imposed: Commonly: shared memory (to support use of pointers), single-threaded.
Qualities yielded: Integrity of data operations: data manipulated only by appropriate functions. Abstraction: implementation details hidden.

Typical uses: Applications where the designer wants a close correlation between entities in the physical world and entities in the program; pedagogy; applications involving complex, dynamic data structures.
Cautions: 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, such as in scientific computing. Lack of additional structuring principles can result in highly complex applications.
Relations to programming languages or environments: Java, C++.

2) OO Style (3/4)

Figure 4-8.
Lunar Lander in the object-oriented style.



2) OO Style (4/4)

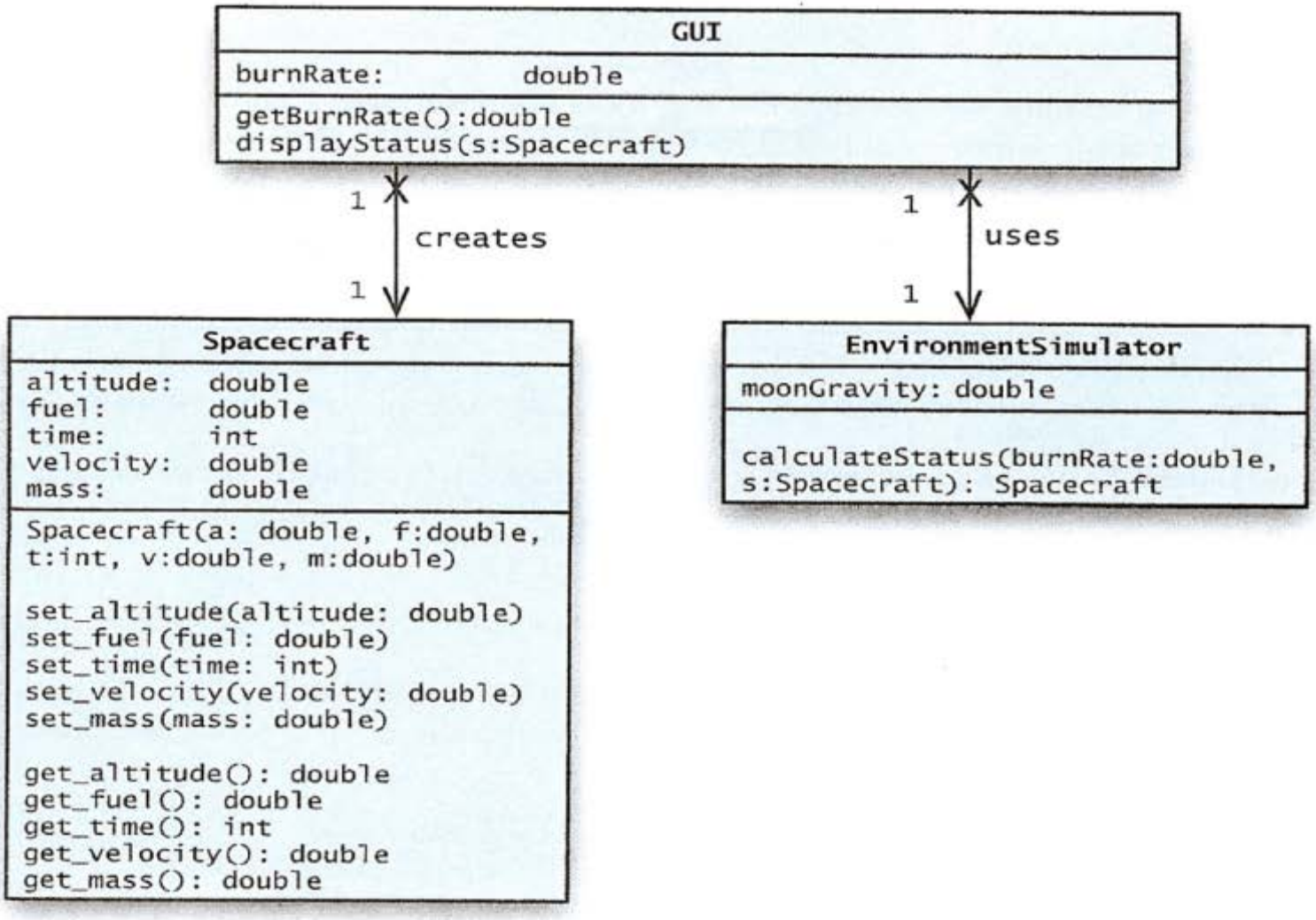


Figure 4-9. *UML representation of the classes used in Figure 4-8.*

2.2 Layered Styles

- Architecture is separated into ordered layers
  - A program within one layer may obtain services from layer below it

- Layered Styles
  1) Virtual Machines Style
  2) Client-server Style

1) Virtual Machines Style (1/4)

Summary: Consists of an ordered sequence of layers; each *layer*, or virtual machine, offers a set of services that may be accessed by programs (subcomponents) residing within the layer above it.

Components: Layers offering a set of services to other layers, typically comprising several programs (subcomponents).

Connectors: Typically procedure calls.

Data elements: Parameters passed between layers.

Topology: Linear, for strict virtual machines; a directed acyclic graph in looser interpretations.

Additional constraints imposed: None.

Qualities yielded: Clear dependence structure; 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.

Typical uses: Operating system design; network protocol stacks.

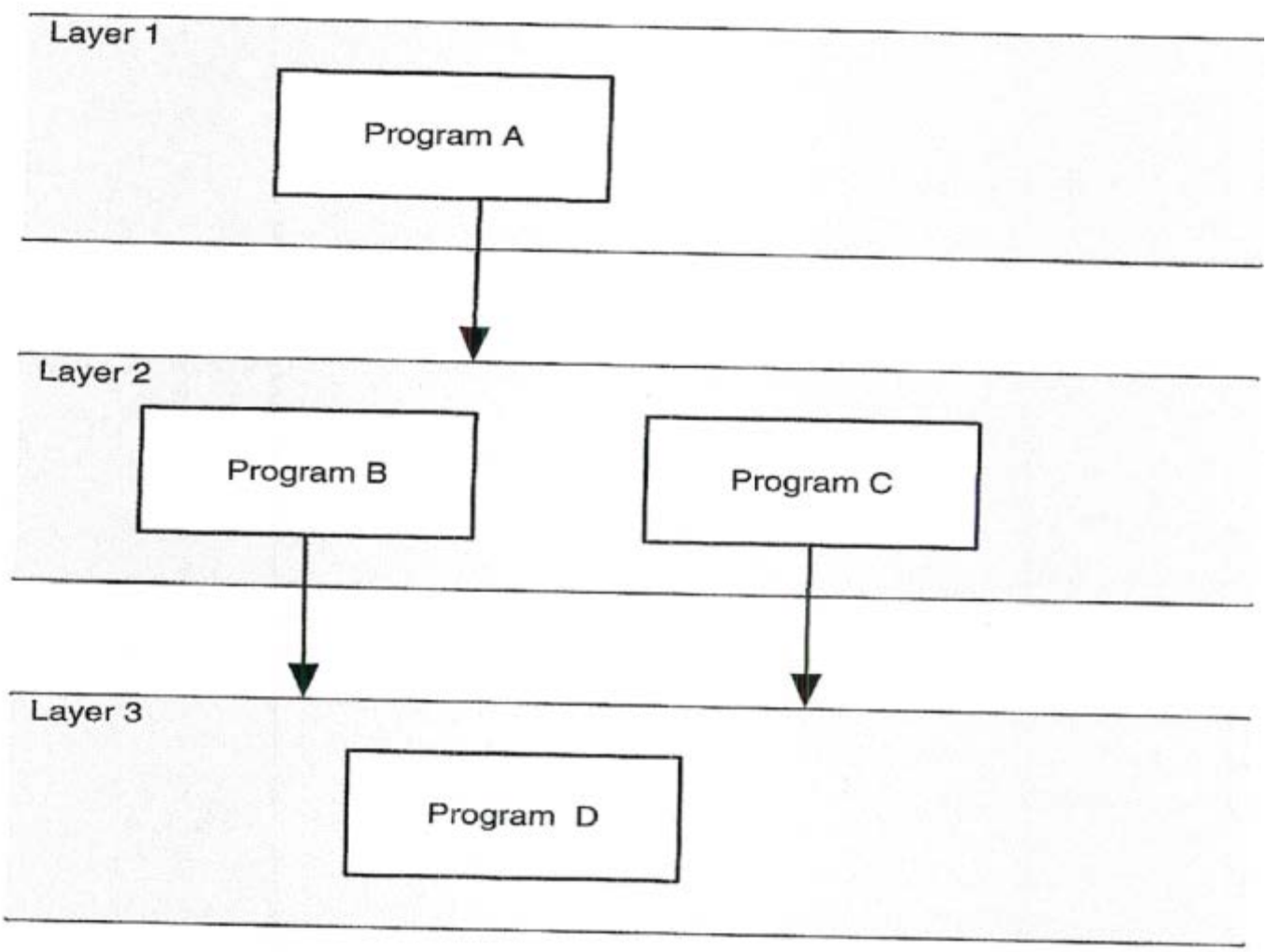
Cautions: Strict virtual machines with many levels can be relatively inefficient.

1) Virtual Machines Style (2/4)

- A layer offers a set of services (“a machine with a bunch of buttons and knobs”) that may be accessed by, at least, program residing within the layer above.
  - Provides interface

1) Virtual Machines Style (3/4)

Figure 4-10.
Notional virtual machines architecture.



1) Virtual Machines Style (4/4)

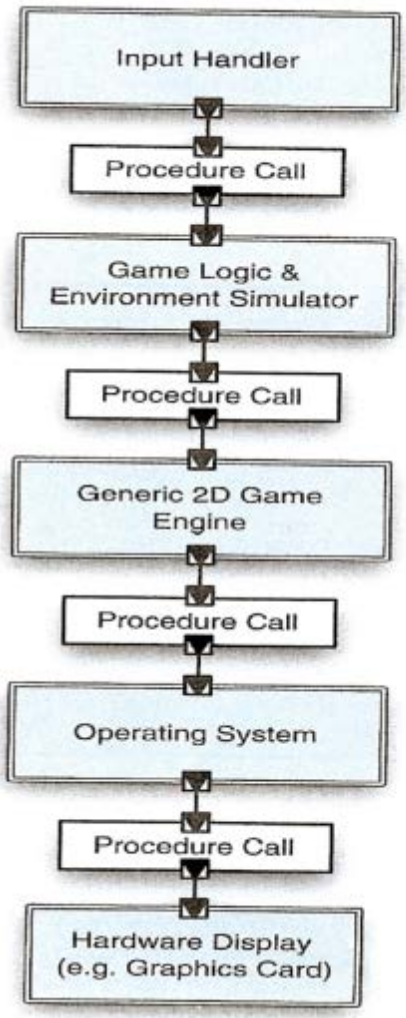


Figure 4-11. *Lunar Lander in the virtual machines style.*

2) Client-Server Style (1/2)

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 and server.
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.
Additional constraints imposed: Client-to-client communication prohibited.
Qualities yielded: Centralization of computation and data at the server, with the information made available to remote clients. A single powerful server can service many clients.

Typical uses: Applications where 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, such as many business applications.
Cautions: When the network bandwidth is limited and there are a large number of client requests.

2) Client-Server Style (2/2)

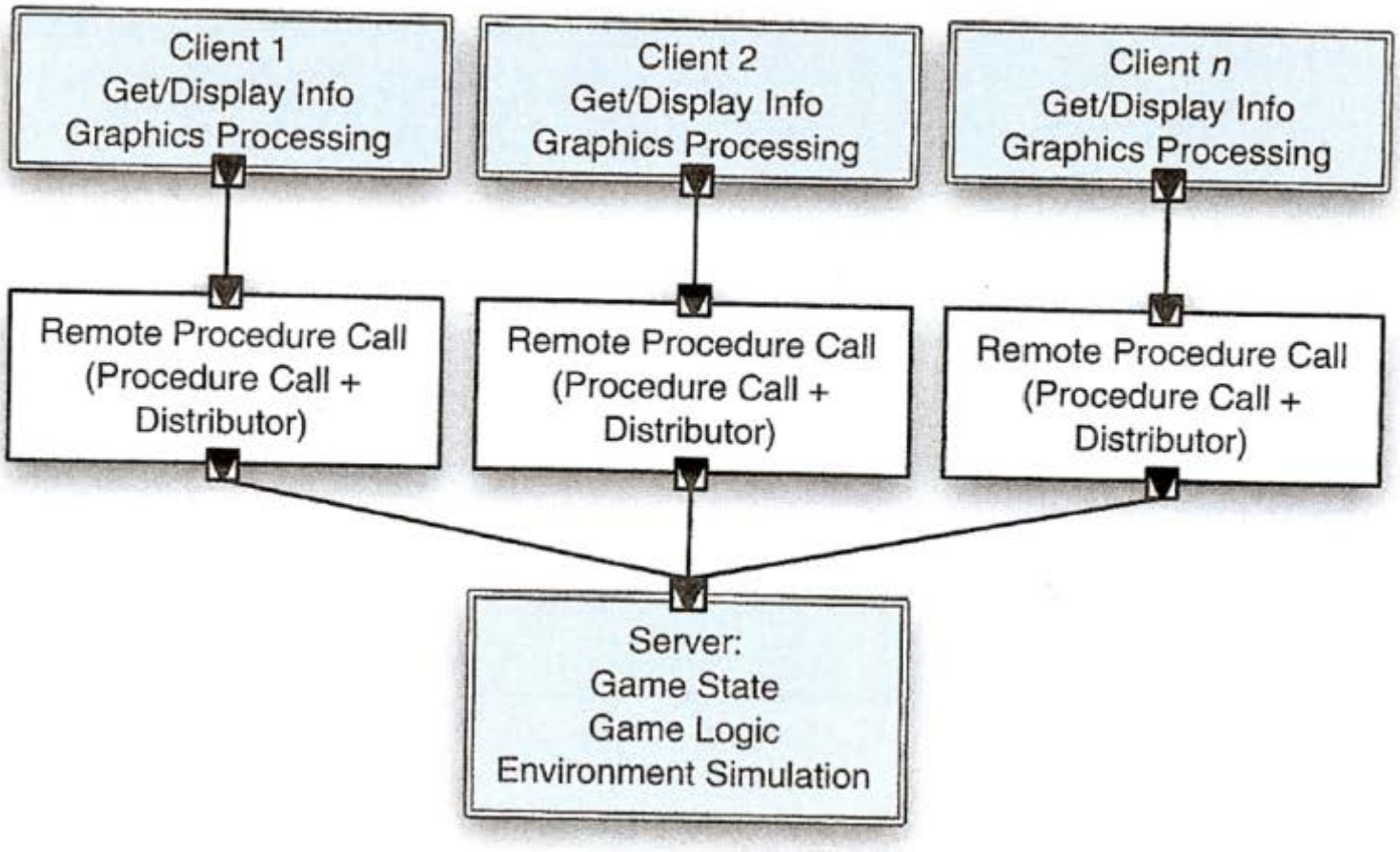


Figure 4-12. A *multiplayer version of Lunar Lander, in a client-server style.*

2.3 Dataflow Styles

- Dataflow Styles
  1) Batch-sequential Style
  2) Pipe-and-Filter Style

1) Batch-Sequential (1/3)

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, aka "sneaker-net."[5]
Data elements: Explicit, aggregate elements passed from one component to the next upon completion of the producing program's execution.
Topology: Linear.
Additional constraints imposed: One program runs at a time, to completion.
Qualities yielded: Severable execution; 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.

1) Batch-Sequential (2/3)

- Update a bank's record of all its accounts, based upon the overnight processing of the day's transactions.

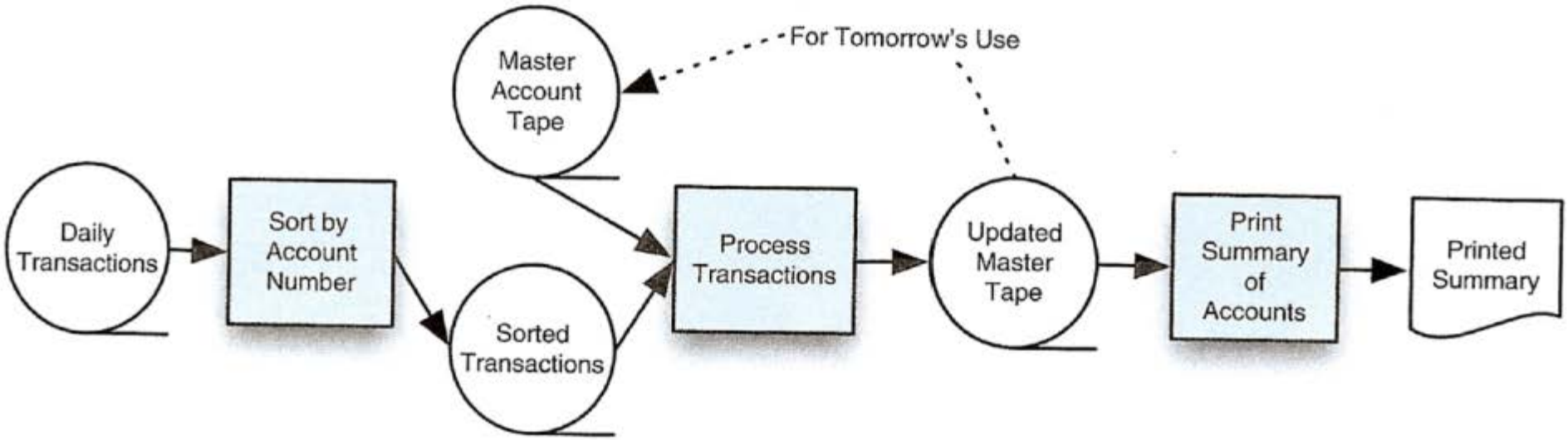


Figure 4-13. *Financial records processed in batch-sequential style.*

1) Batch-Sequential (3/3)

- Attempt to build a Lunar Lander game in the batch sequential style
- Is this a good approach? NO. Not interactive and not realtime !

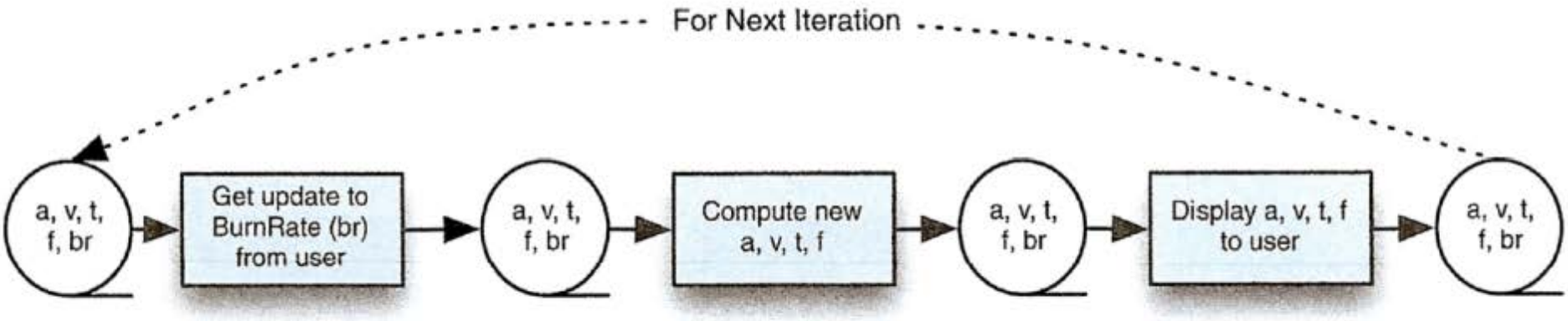
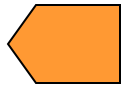


Figure 4-14. *Lunar Lander: Batch-sequential.*

2) Uniform Pipe-and-Filter (1/2)

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, though T fittings are possible.
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.



2) Uniform Pipe-and-Filter (2/2)

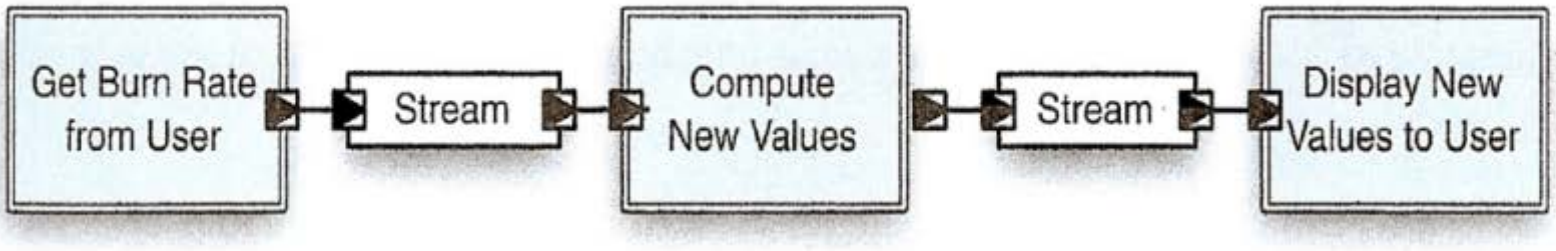


Figure 4-15. *Lunar Lander in pipe-and-filter style.*

2.4 Shared Memory Styles

- Shared Memory Styles
  1) Blackboard Style
  2) Rule-based/Expert System Style

1) Blackboard (1/2)

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 blackboard.
Topology: Star topology, with the blackboard at the center.
Variants: In one version of the style, programs poll the blackboard to determine if any values of interest have changed; in another version, a blackboard manager notifies interested components of an update to the blackboard.
Qualities yielded: Complete solution strategies to complex problems do not have to be preplanned. Evolving views of the 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 (requiring propagating changes to all the participating components).
Relations to programming languages or environments: Versions of the blackboard style that allow concurrency between the constituent programs require concurrency primitives for managing the shared blackboard.

1) Blackboard (2/2)

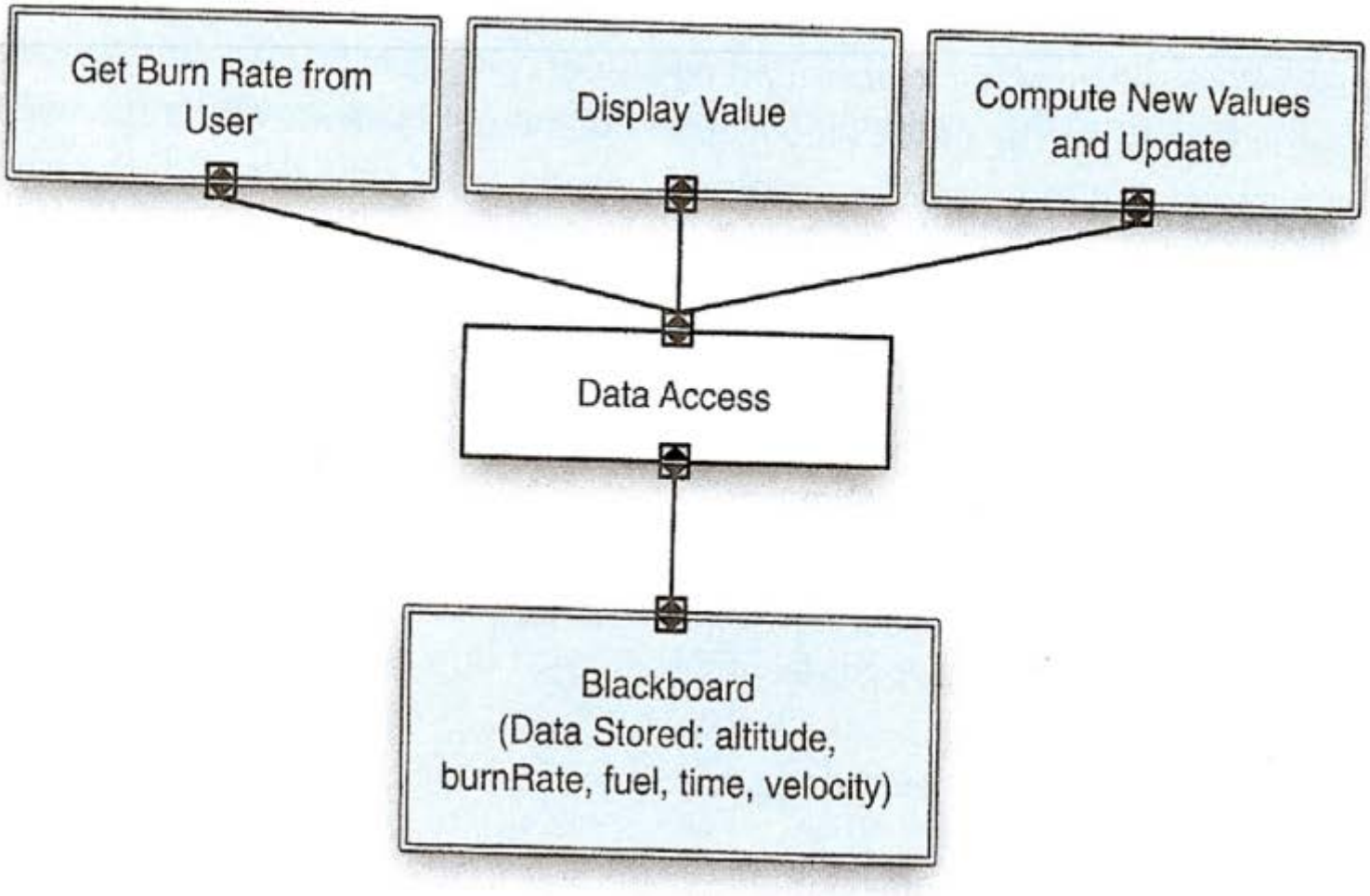


Figure 4-16. *Lunar Lander in blackboard style.*

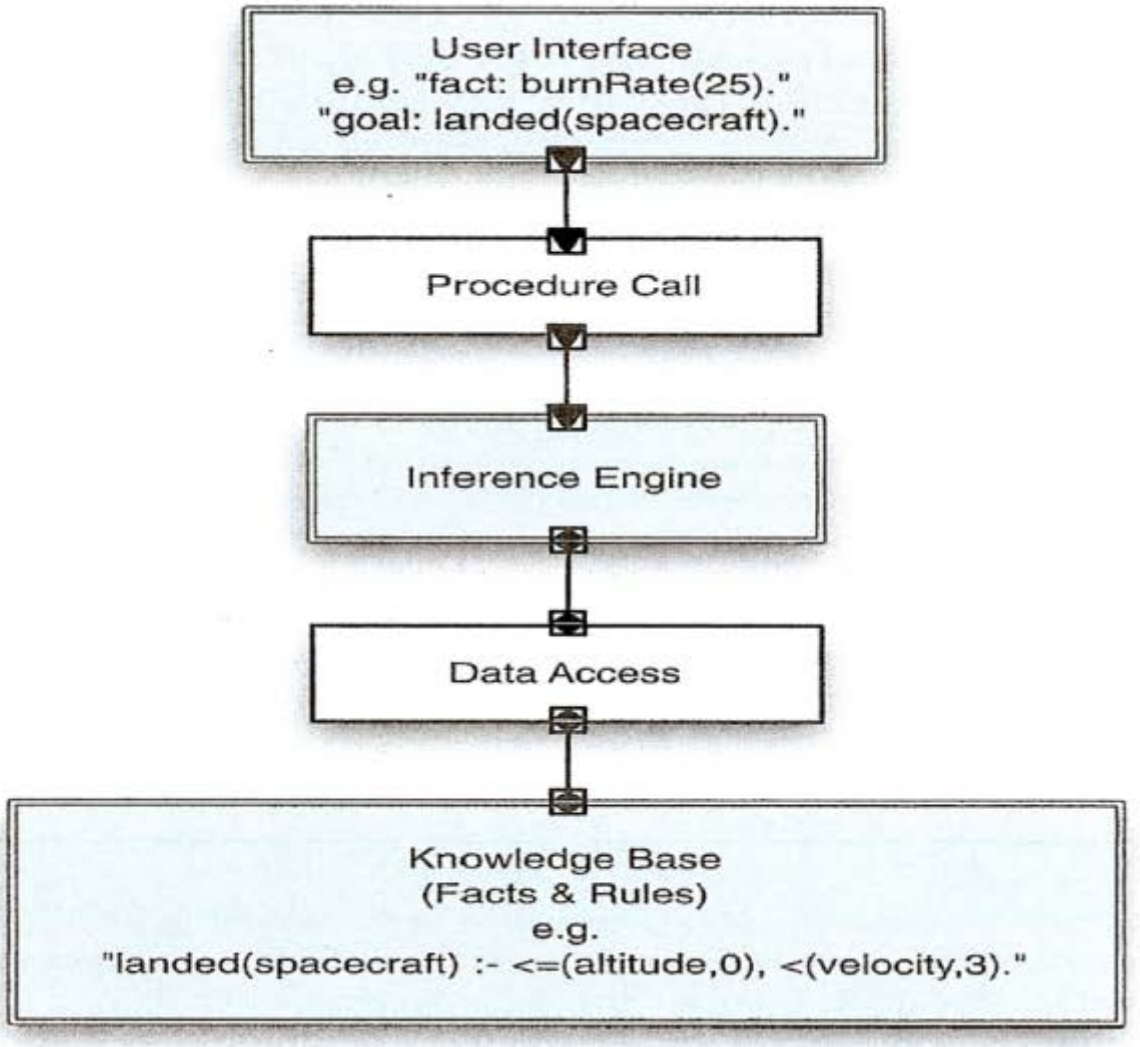
2) Rule-Based/Expert System (1/2)

Summary: 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.
Components: User interface, inference engine, knowledge base.
Connectors: Components are tightly interconnected, with direct procedure calls and/or shared data access.
Data Elements: Facts and queries.
Topology: Tightly coupled three-tier (direct connection of user interface, inference engine, and knowledge base).
Qualities yielded: Behavior of the application can be easily modified through dynamic addition or deletion of rules from the knowledge base. Small systems can be quickly prototyped. Thus useful for iteratively exploring problems whose general solution approach is unclear.

Typical uses: When the problem can be understood as matter of repeatedly resolving a set of predicates.
Cautions: When a large number of rules are involved, understanding the interactions between multiple rules affected by the same facts can become very difficult. Understanding the logical basis for a computed result can be as important as the result itself.
Relations to programming languages or environments: Prolog is a common language for building rule-based systems.

2) Rule-Based/Expert System (2/2)

Figure 4-17.
Lunar Lander in a rule-based style.



2.5 Interpreter Styles

- Interpreter Styles
  1) Basic Interpreter Style
  2) Mobile Code Style

1) Interpreter (1/2)

Summary: Interpreter parses and executes input commands, updating the state maintained by the interpreter.
Components: Command interpreter, program/interpreter state, user interface.
Connectors: Typically the command interpreter, user interface, and state are very closely bound with direct procedure calls and shared state.
Data elements: Commands.
Topology: Tightly coupled three-tier; state can be separated from the interpreter.
Qualities 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.

Typical uses: Superb for end-user programmability; supports dynamically changing set of capabilities.
Cautions: 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.
Relations to programming languages or environments: Lisp and Scheme are interpretive languages, and sometimes used when building other interpreters; Word/Excel macros.

1) Interpreter (2/2)

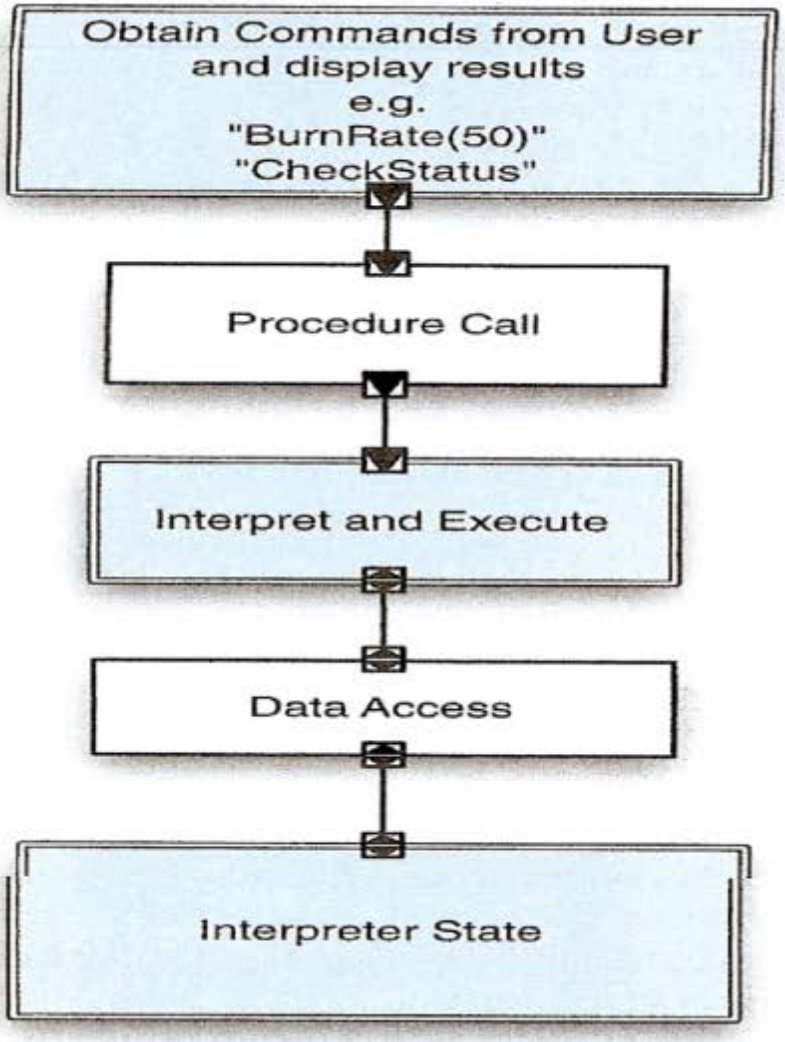


Figure 4-18.
Lunar Lander in interpreter style.

2) Mobile Code

Summary: Code moves to be interpreted on another host; depending on the variant, state does also.
Components: Execution dock, which handles receipt and deployment 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.
Topology: Network.
Variants: Code-on-demand, remote evaluation, and mobile agent.
Qualities yielded: Dynamic adaptability. Takes advantage of the aggregate computing power of available hosts; increased dependability through provision of migration to new hosts.

Typical uses: When processing large data sets in distributed locations, it becomes more efficient to have the code move to the location of these large data sets; when it is desirous to dynamically customize a local processing node through inclusion of external code.

Cautions: Security issues—execution of imported code may expose the host machine to malware. Other cautions—when it is necessary to tightly control the different software versions deployed; when costs of transmission exceed the cost of computation; when network connections are unreliable.
Relations to programming languages or environments: Scripting languages (such as JavaScript, VBScript), ActiveX controls. Grid computing.

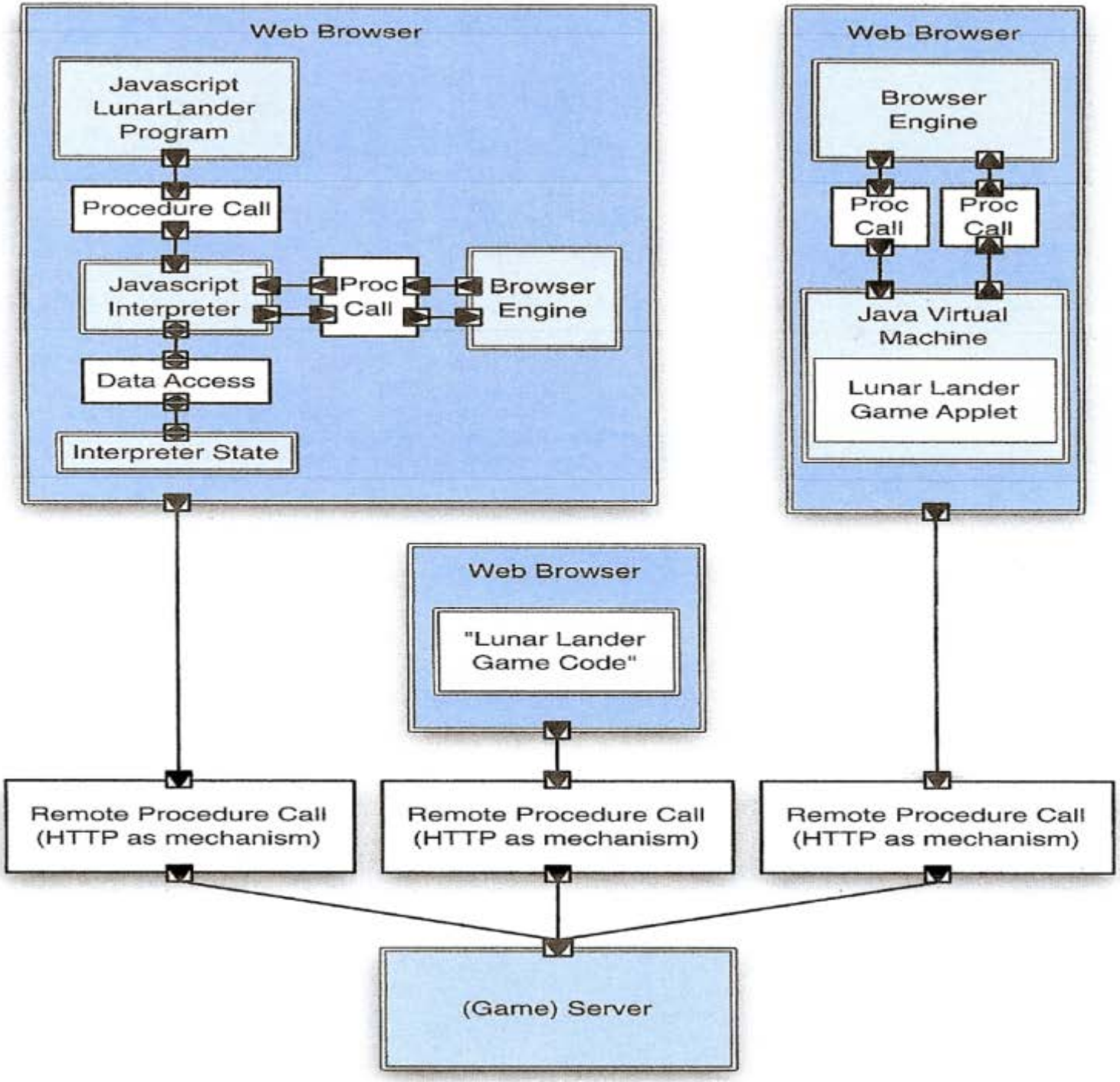


Figure 4-19.
Mobile code: Lunar Lander as code-on-demand.

Three different independent forms of mobile code

2.6 Implicit Invocation Styles

"Implicit invocation" describes the execution model of both the publish-subscribe and event-based styles. An action, such as invocation of a procedure, may be taken as the result of some activity occurring elsewhere in the system, but the two activities are distant from one another. 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.

- Implicit Invocation Styles
  1) Publish-Subscribe Style
  2) Event-Based Style

1) Publish-Subscribe (1/2)

Summary: Subscribers register/deregister to receive specific messages or specific content. Publishers maintain a subscription list and broadcast messages to subscribers either synchronously or asynchronously.
Components: Publishers, subscribers, proxies for managing distribution.

Connectors: Procedure calls may be used within programs, more 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.
Variants: Specific uses of the style may require particular steps for subscribing and unsubscribing. Support for complex matching of subscription interests and available information may be provided and be performed by intermediaries.
Qualities yielded: Highly efficient one-way dissemination of information with very low coupling of components.

Typical uses: News dissemination—whether in the real world or online events. Graphical user interface programming. Multiplayer-network–based games.
Cautions: When the number of subscribers for a single data item is very large a specialized broadcast protocol will likely be necessary.
Relations to programming languages or environments: In large-scale systems support for publish-subscribe is provided by commercial middleware technology.

1) Publish-Subscribe (2/2)

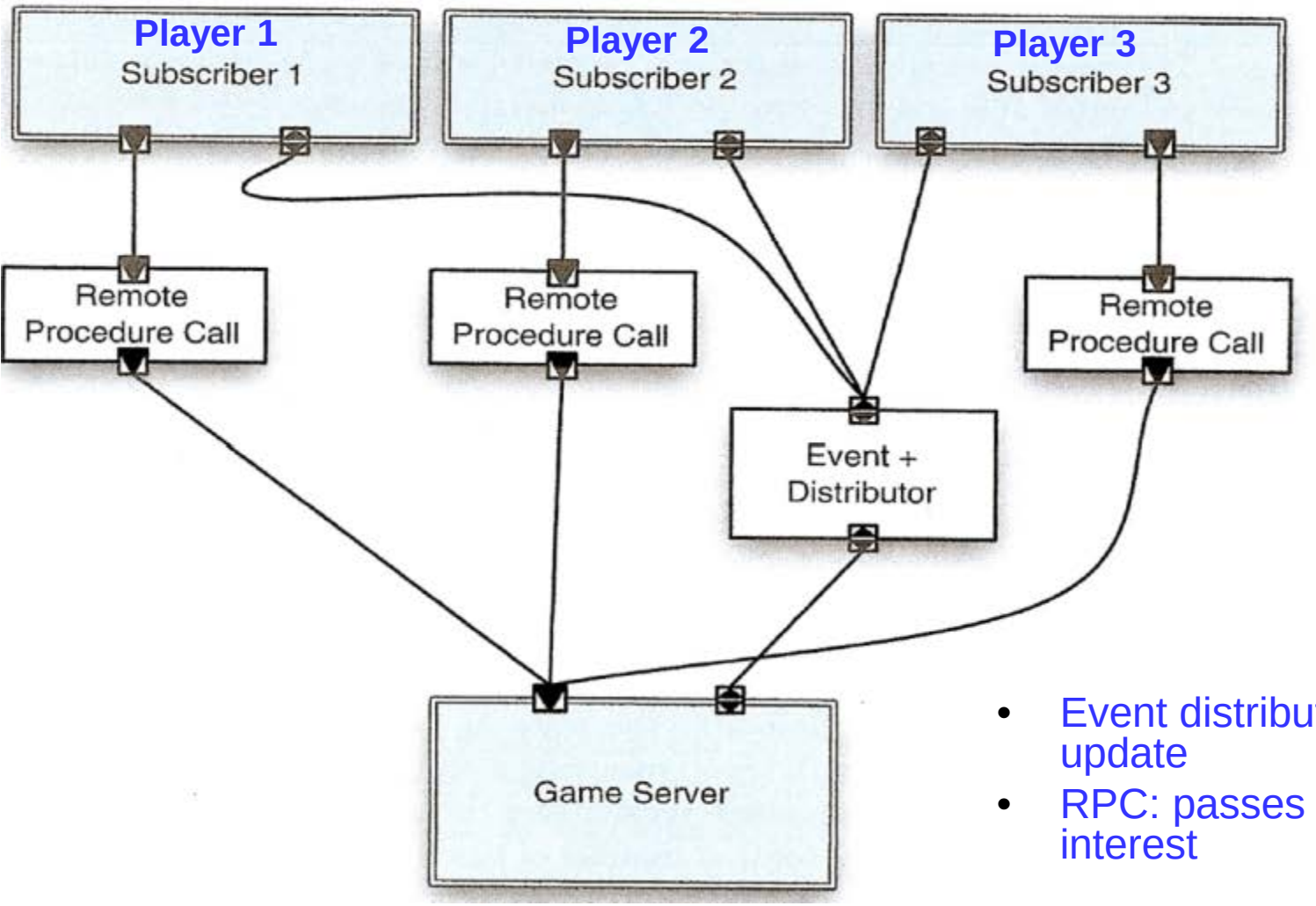


Figure 4-20. *Lunar Lander in publish-subscribe.*

- Event distributor: game software update
- RPC: passes registered info of interest

2) Event-Based (1/2)

Summary: Independent components asynchronously emit and receive events communicated over event buses.
Components: Independent, concurrent event generators and/or consumers.
Connectors: Event bus. In variants, more than one may be used.
Data elements: Events—data sent as a first-class entity over the event bus.

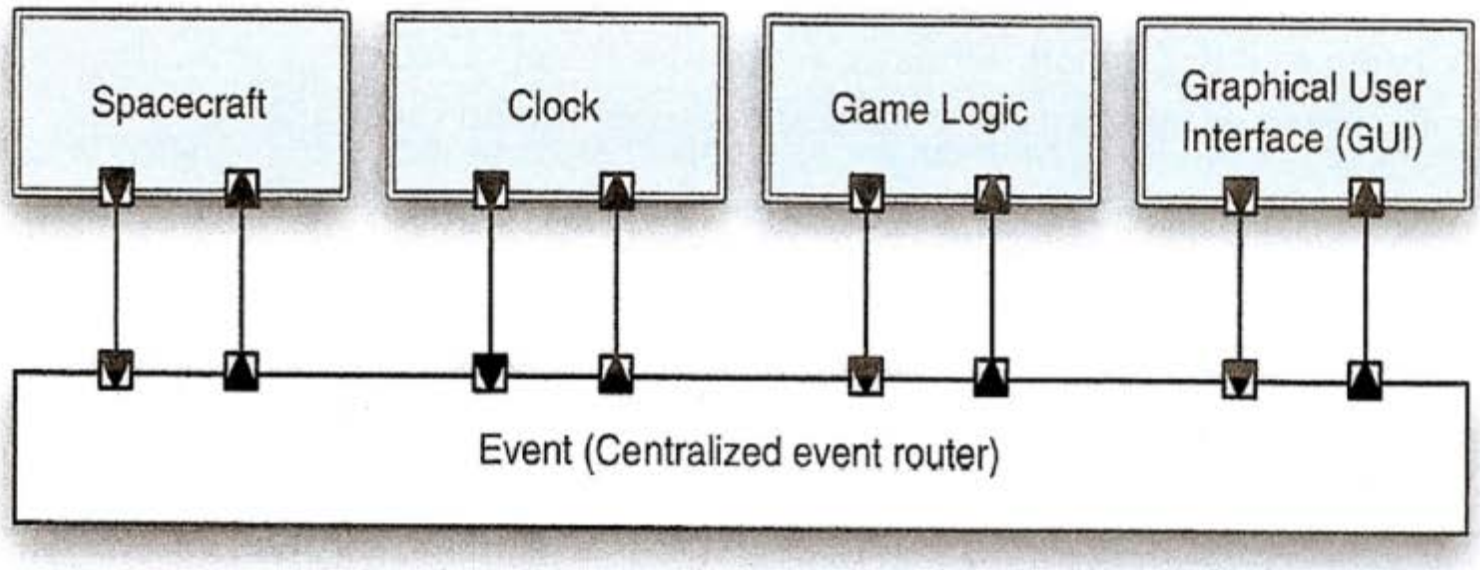
Topology: Components communicate with the event-buses, not directly to each other.
Variants: Component communication with the event-bus may either be push or pull based.
Qualities yielded: Highly scalable, easy to evolve, effective for highly distributed, heterogeneous applications.

Typical uses: User interface software, wide-area applications involving independent parties (such as financial markets, logistics, sensor networks).
Cautions: No guarantee if or when a particular event will be processed.
Relations to programming languages or environments: Commercial message–oriented middleware technologies support event-based architectures.

2) Event-Based (2/2)

- Every fraction of a second, the clock component sends out a tick notification to the event bus, which distributes that even to the other components.

Figure 4-21. *Lunar Lander in the event-based style.*



2.7 Peer-to-Peer Style

Summary: State and behavior are distributed among peers that can act as either clients or servers.

Components: Peers—independent components, having their own state and control thread.

Connectors: Network protocols, often custom.

Data elements: Network messages.

Topology: Network (may have redundant connections between peers); can vary arbitrarily and dynamically.

Qualities yielded: 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.

Typical uses: Where sources of information and operations are distributed and network is ad hoc.

Cautions: When information retrieval is time critical and cannot afford the latency imposed by the protocol. Security—P2P networks must make provision for detecting malicious peers and managing trust in an open environment.

Napster Architecture

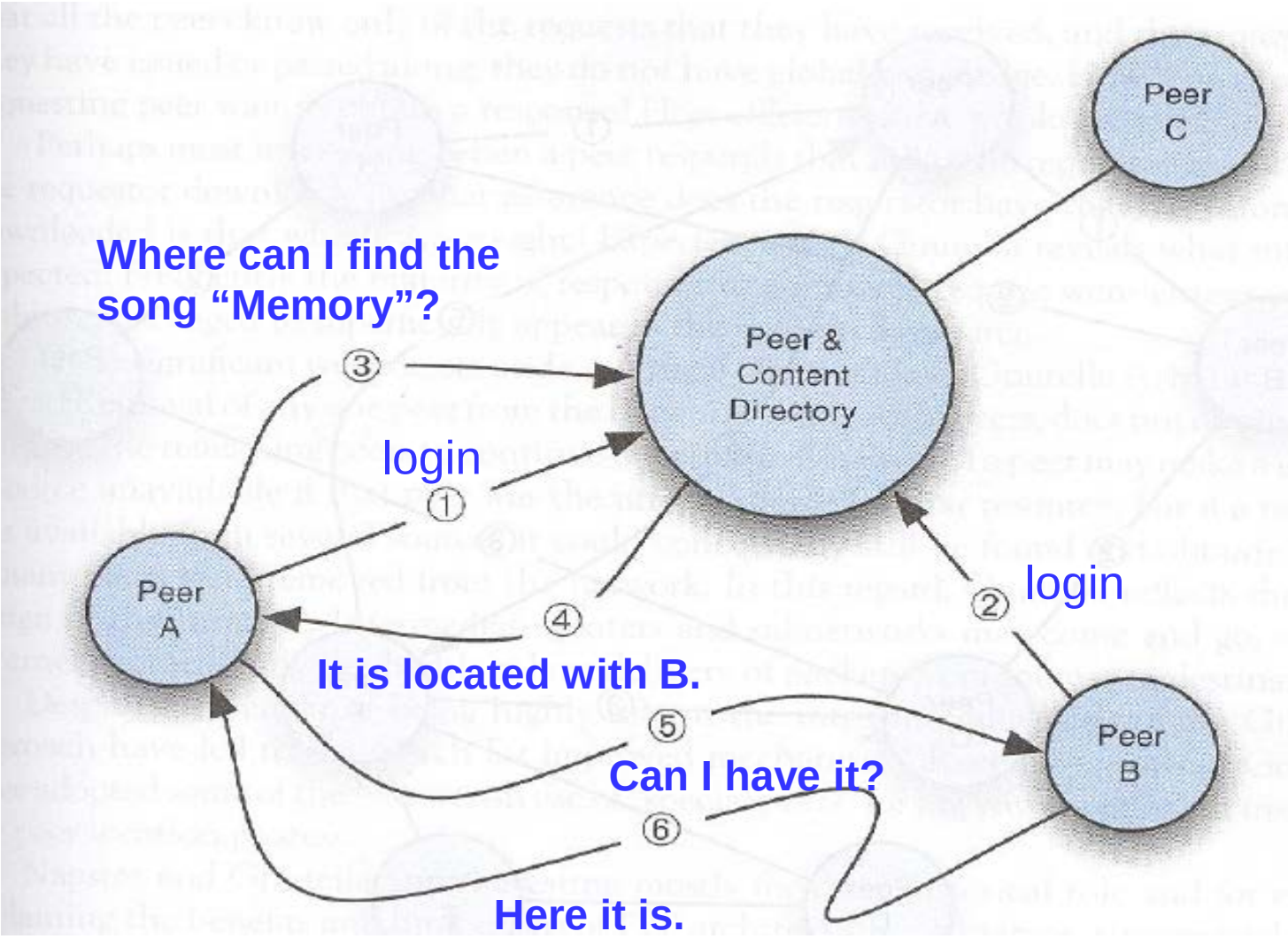


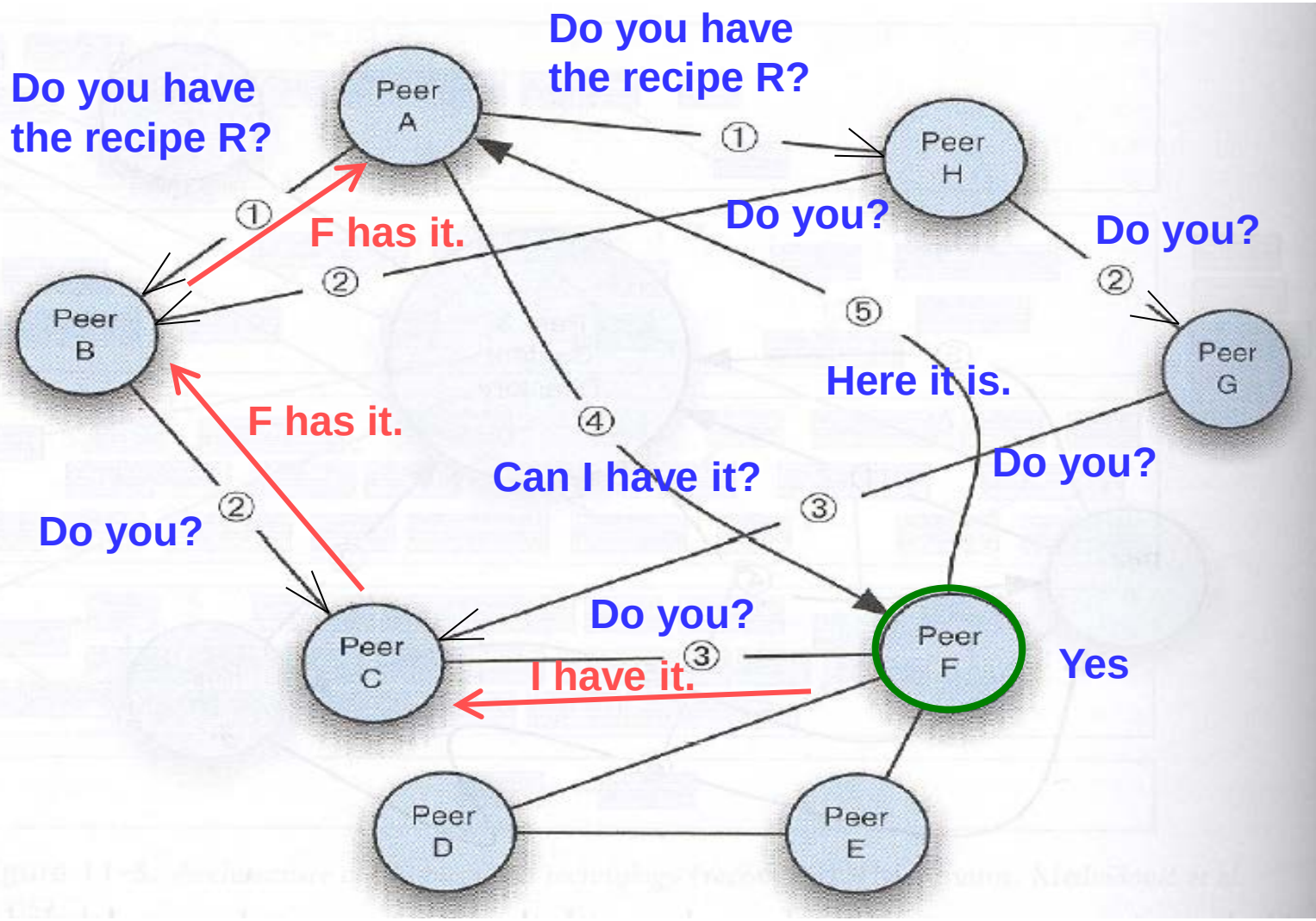
Figure 11-4. *Notional view of the operation of Napster. In steps 1 and 2, Peers A and B log in with the server. In step 3, Peer A queries the server where it can find Rondo Veneziano's "Masquerade." The location of Peer B is returned to A (step 4). In step 5, A asks B for the song, which is then transferred to A (step 6).*

Napster Architecture

- Architecturally
  - Hybrid of client-server and pure peer-to-peer
- Proprietary protocol for interaction between peers and the content directory
  - Only mp3 files shared
- Problem
  - If a song is highly desired or a server goes down

Peer-to-Peer Architecture in Gruntella Protocol

Figure 11-5.
Notional interactions between peers using the original Gnutella protocol.



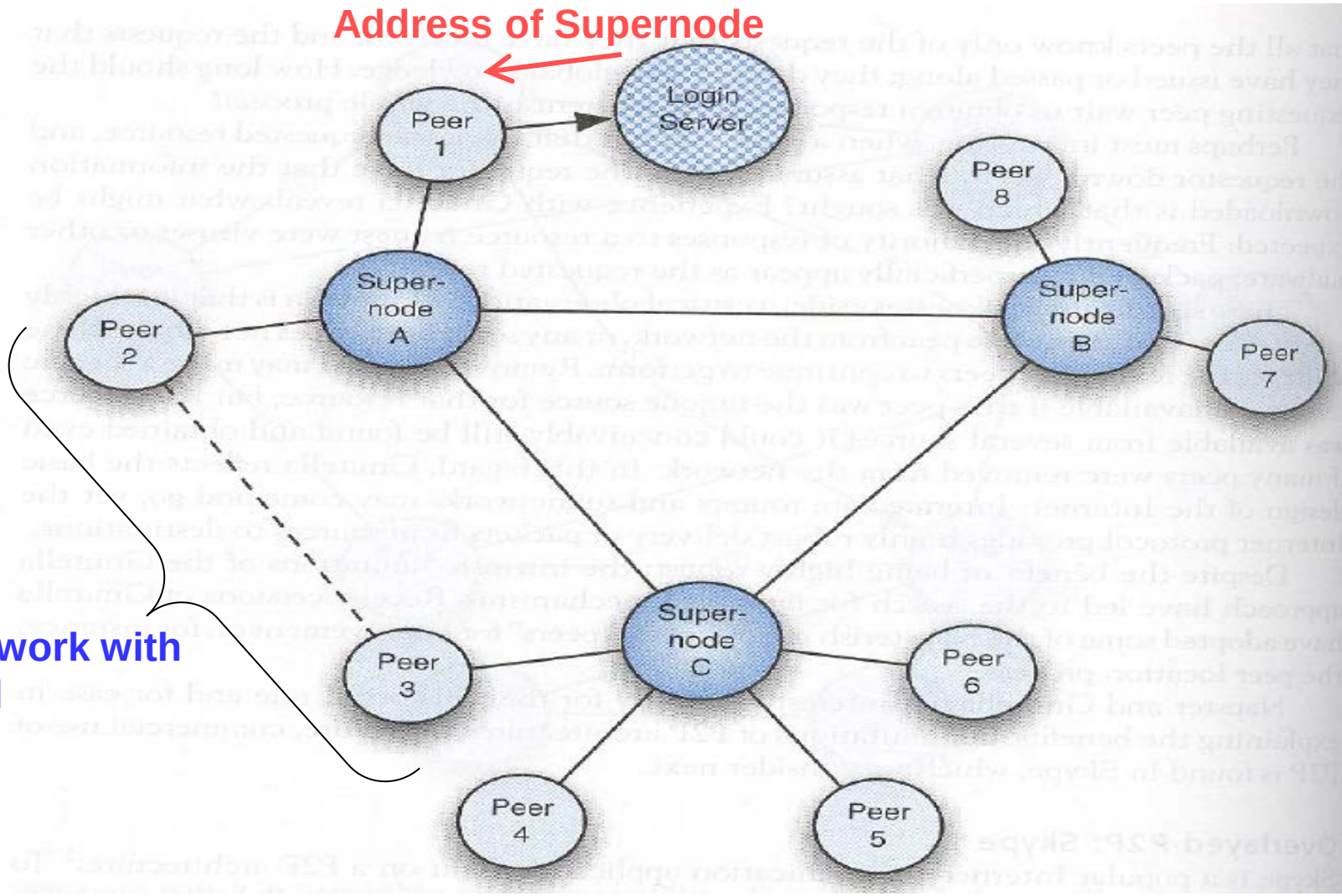
Peer-to-Peer Architecture in Gruntella Protocol

- Robust
  - Removal of any one per from the network or any set of peers , does not diminish the ability of the remaining peers to continue to perform

- Problem
  - No assurance that the requestor has downloaded the information that was sought.
    - The majority of responses to a resource request were viruses or other malware, packaged to superficially appear as the requested resource.

Skype Architecture

Figure 11-6.
Notional instance of the Skype architecture.



Skype Architecture

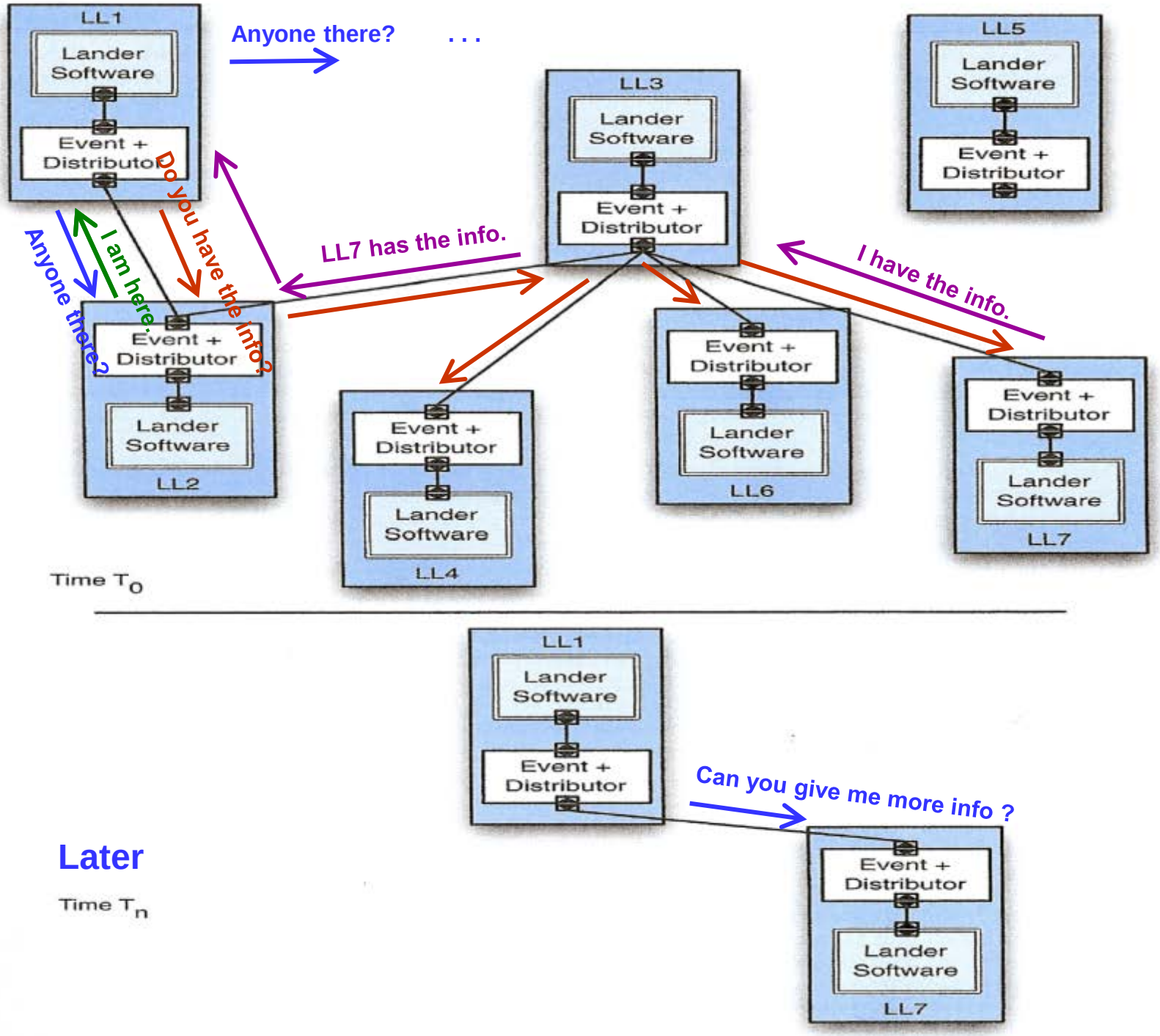
- Protocol
  - Proprietary
- Supernode
  - Provides directory service and call routing
  - Skype peers get promoted to supernode based on their history of network and machine performance.

    => What if a peer happened to “pay by usage amount” ?

- Architecture
  - Mixture of client-sever and peer-to-peer address discovery problem

    => network not flooded with request (in contrast with Gruntella)
  - Replication and distribution of the directories addresses the scalability and robustness problems of Napster
  - Restriction of participants to clients issued only by Skype, and making those clients highly resistant to inspection or modification, prevent malicious client from entering the network, avoiding the Gruntella problem.

Figure 4-22. *Lunar Lander a a P2P application.*



LL1 wants to find out if another spacecraft has already landed to avoid collision.

2.8 Other Architecture Styles

1) Pipes and Filters
2) Data Abstraction and Object-Oriented Organization
3) Event-based, Implicit Invocation
4) Layered Systems
5) Repositories

Acknowledgement.

Based on [Garlan 94] "An Introduction to Software Architecture."

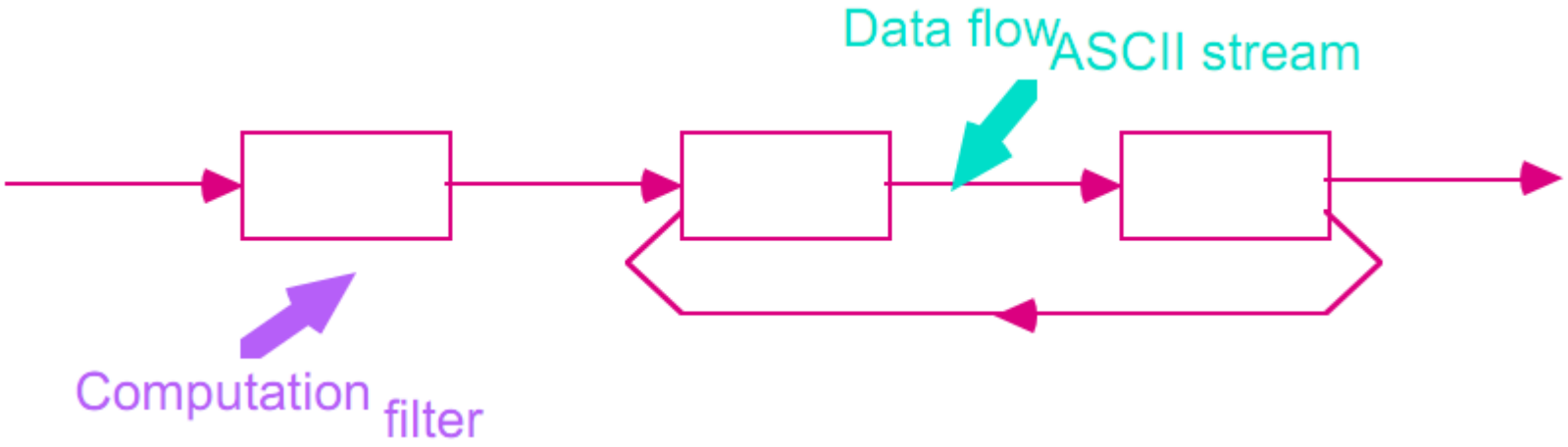
Prepared with Dr. Jinkyu Kim

Architecture Styles

- Architectural style:
  - determines the vocabulary of components and connectors that can be used in instances of that style,
  - together with a set of constraints on how they can be combined

- An architectural style must answer the following questions:
  - What is the structural pattern: the components, connectors, and constraints?
  - What is the underlying computational model?
  - What are the essential invariants of the style?
  - What are some common examples of its use?
  - What are the advantages and disadvantages of using that style?
  - What are some of the common specializations?

1) Pipes and Filters (1/2)



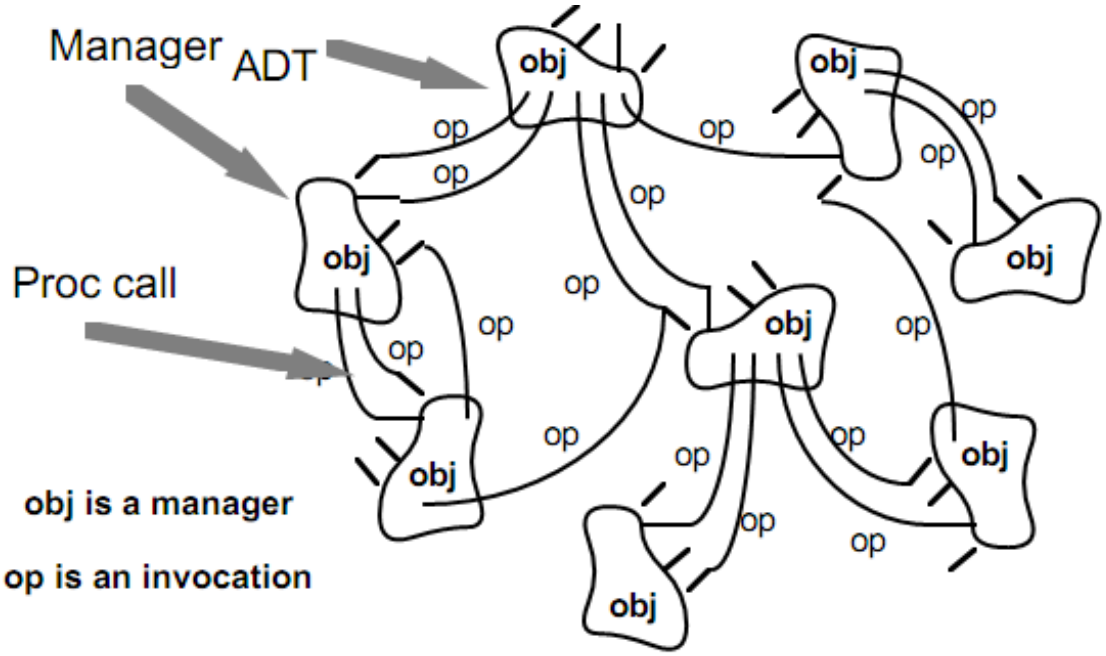
- Each component (Filter) has a set of inputs and a set of outputs
- A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order.
- Connectors (Pipe): conduits for the streams, transmitting outputs of one filter to inputs of another.
- Invariant:
  1. filters must be independent entities.
  2. filters do not know the identity of their upstream and downstream filters

1) Pipes and Filters (2/2)

- Advantages:
  1. Designer can understand the overall input/output behavior of a system.
  2. Support of reuse: any two filter can transmit data with communication agreements.
  3. Systems can be easily maintained and enhanced: new filters can be added to existing systems and old filters can be replaced by improved ones.
  4. Support of analysis: such as throughput and deadlock analysis
  5. Concurrent execution

- Disadvantages:
  1. Batch organization of processing: 1회성 자료처리.
  2. Typically not good at handling Interactive Applications.
  3. Having difficulties in maintaining correspondences between two separate.
  4. Force lowest common denominator (pipe) on data transmission: so that filters have to do additional works such as parsing or unparsing its data.

2) Data Abstraction and Object-Oriented Organization (1/2)



- Associated primitive operations are encapsulated in an abstract data type or object.
- The components in this style: objects
- Objects: a manager because it is responsible for preserving the integrity of a resource

2) Data Abstraction and Object-Oriented Organization (2/2)

- Advantages:
  1. Hide its representation from clients → Can change implementation without affecting clients.
  2. Designers can decompose problems into collections of interacting agents

- Disadvantages:
  - To interact with another object, it must identity that object.

    Cf) Filters do need not know what other filters are in the system in order to interact with them.
  - Side-effect problems:
    - if A uses object B and C also uses B, then C's effects on B look like unexpected side effects to A, and vice versa.

3) Event-based, Implicit Invocation (1/2)

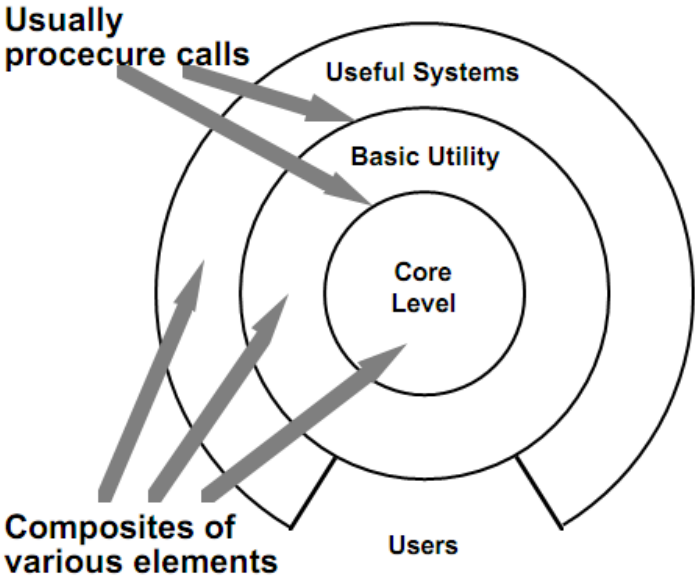
- Traditional way:
  - Component interfaces provide a collection of procedures and functions; Components interact with each other by explicitly invoking routines

- Alternative integration way:
  - Implicit invocation, reactive integration, and selective broadcast.
  - Component can announce (or broadcast) one or more events.
  - Other components can register an interest in an event by associating a procedure with the event.
  - Components in an implicit invocation style are modules whose interfaces provide both a collection of procedures and a set of events.

3) Event-based, Implicit Invocation (2/2)

- Invariant:
  - Announce component do not know which components will be affected by events
  - Components cannot make assumptions about order of processing, or even about what processing will occur as a result of their events.
- Advantages:
  - Strong support for reuse: Any component can be embedded into a system simply by registering it for the events.
  - Ease to system evolution: Components may be replaced by other components without affecting other components
- Disadvantages:
  - Components cannot control over the computation performed by the system.
  - Ex. When a component announces an event → no idea what other components will respond to it
  - Exchange of data: data can be passed with the event and depended on a shared repository for interaction
  - Reasoning about correctness of system.

4) Layered Systems (1/2)



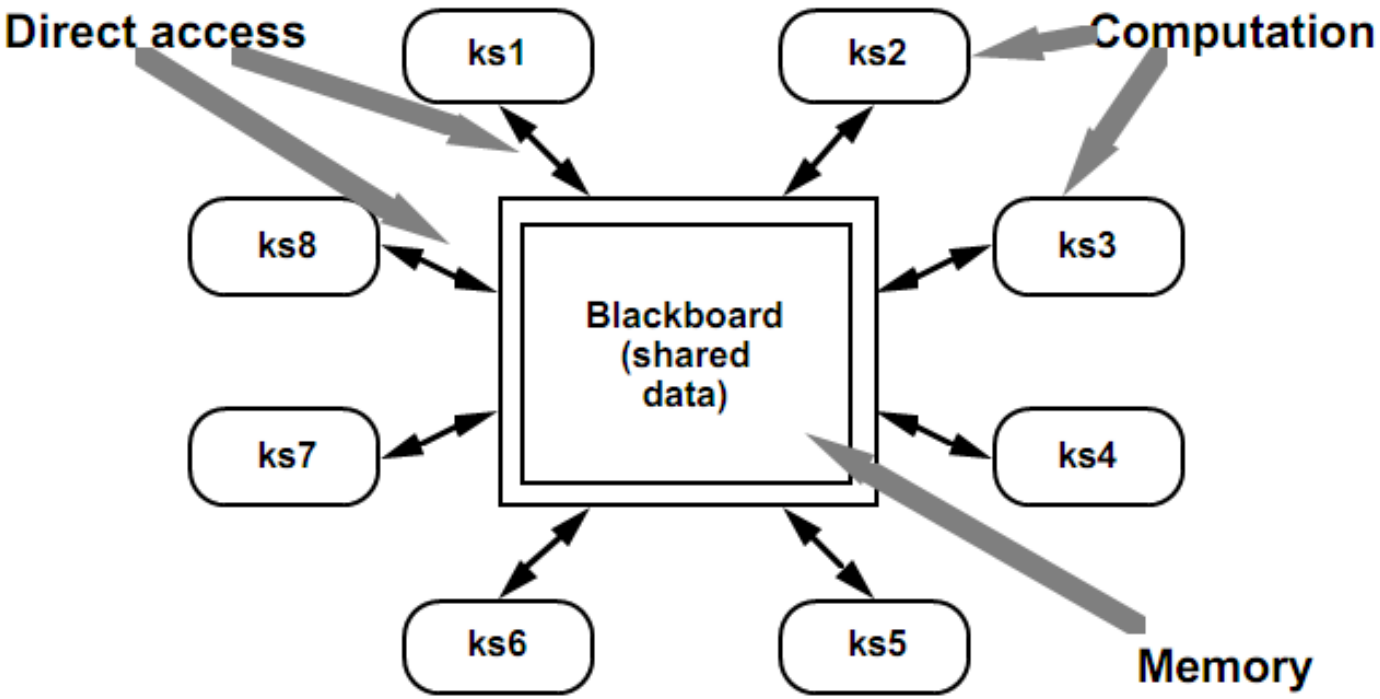
- A layered system is organized hierarchically
  - Each layer providing service to the layer above it and serving as a client to the layer below
- The most widely known examples of this style:
  - layered communication protocols: OSI 7 layer.

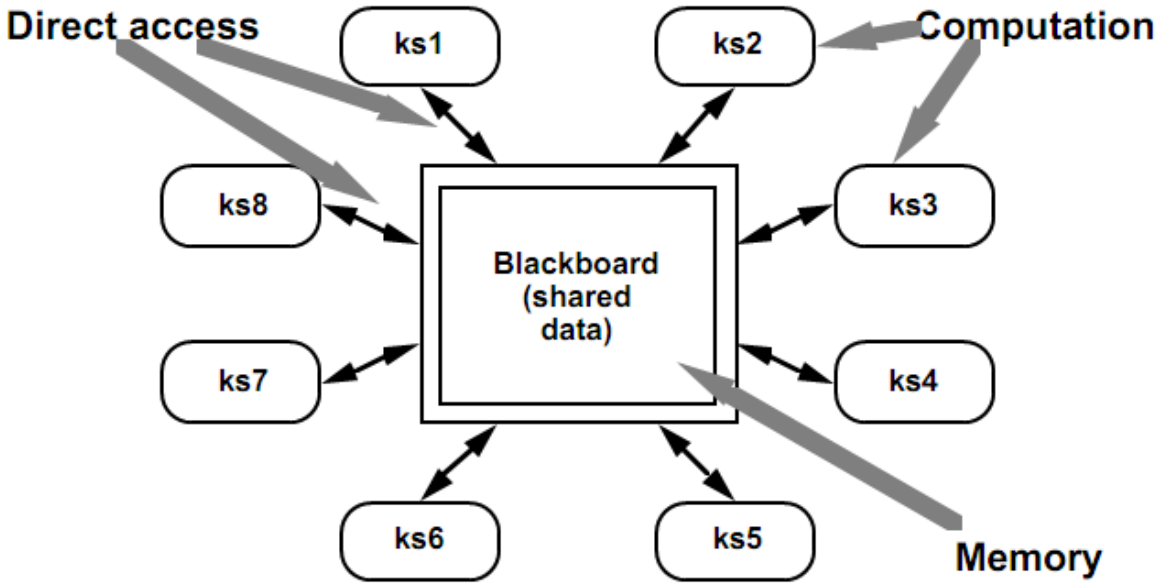
4) Layered Systems (2/2)

- Advantages:
  - Support of design based on increasing levels of abstraction.
  - Changes to the function of one layer affect at only above and below layers
  - Support reuse: different implementations of the same layer can be used interchangeably.

- Disadvantages:
  - Not all systems are easily structured in a layered fashion.
  - It can be quite difficult to find the right levels of abstraction

5) Repositories = Traditional database





- Two kinds of Component:
  - A central data structure represents the current state.
  - a collection of independent components operate on the central data store.

Example Blackboard Architecture

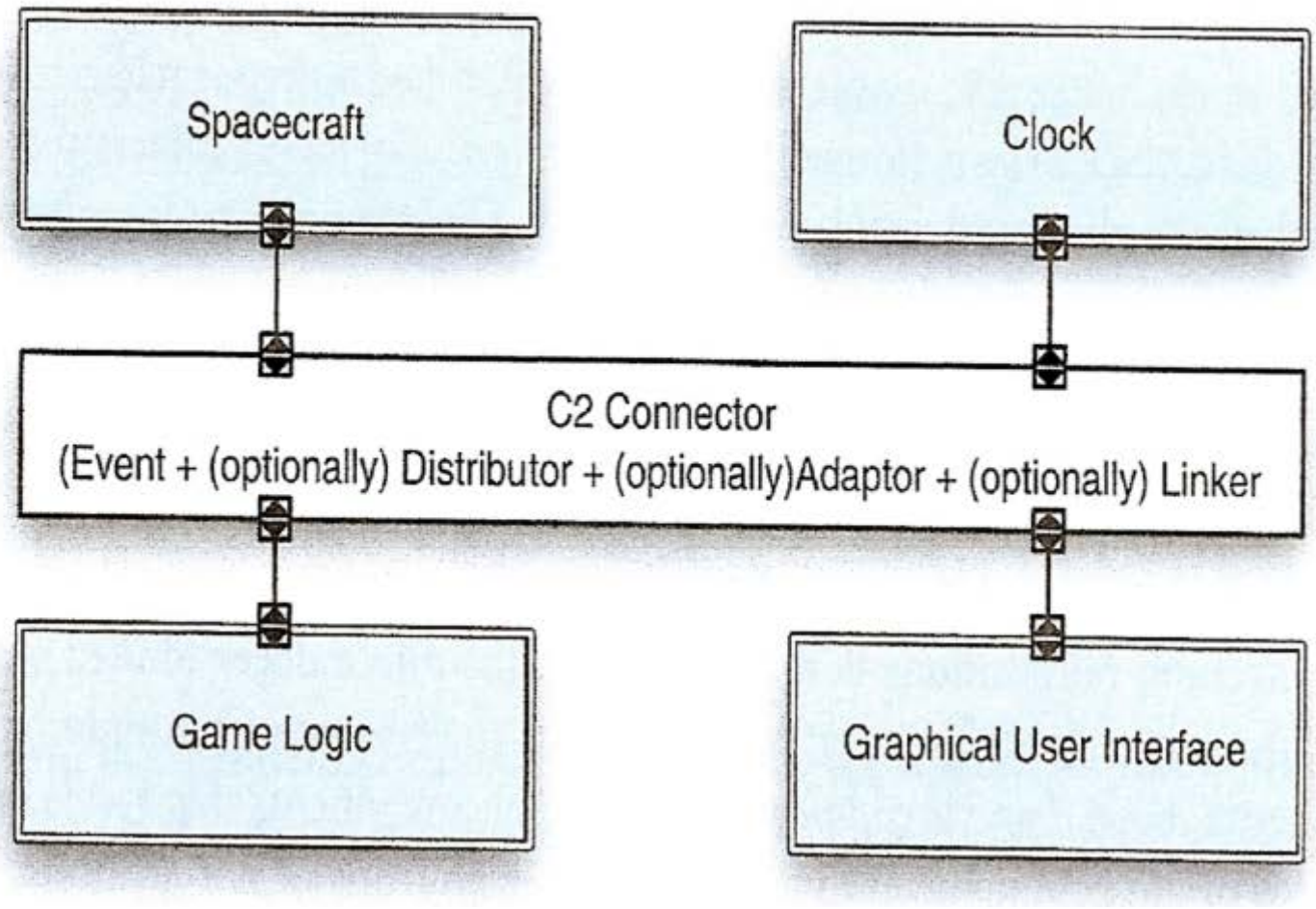
- Three Major parts:

1. Knowledge sources: separate, application-dependent knowledge,
2. Blackboard data structure: problem-solving state data, an application-dependent hierarchy
3. Control: Driven entirely by state of blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable = Query

3. Complex Styles

3.1 C2
3.2 Distributed Objects

Figure 4-23.
Lunar Lander in the C2 style.

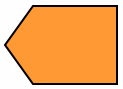


3.1 C2

Summary: An indirect invocation style in which independent components communicate exclusively through message routing connectors. Strict rules on connections between components and connectors induce layering.
Components: Independent, potentially concurrent message generators and/or consumers.
Connectors: Message routers that may filter, translate, and broadcast messages of two kinds—notifications and requests.
Data elements: Messages—data sent as first-class entities over the connectors. Notification messages announce changes of state. Request messages request performance of an action.
Topology: Layers of components and connectors, with a defined top and bottom, wherein notifications flow downward and requests upward.

Additional constraints imposed:

- All components and connectors have a defined top and bottom. The top of a component may be attached to the bottom of a single connector and the bottom of a component may be attached to the top of a single connector. No direct component-to-component links are allowed; there is, however, no bound on the number of components or connectors that may be attached to a single connector. When two connectors are attached to each other, it must be from the bottom of one to the top of the other.
- Each component has a top and bottom *domain*. The top domain specifies the set of notifications to which a component may react and the set of requests that the component emits up an architecture. The bottom domain specifies the set of notifications that this component emits down an architecture and the set of requests to which it responds.
- Components may be hierarchically composed, where an entire architecture becomes a single component in another, larger architecture.
- Each component may have its own thread(s) of control.
- There can be no assumption of a shared address space among components.



C2

Qualities yielded:

- Substrate independence: Ease in moving the application to new platforms.
- Applications composable from heterogeneous components running on diverse platforms.
  - Support for product lines.
  - Ability to program in the model-view-controller style, but with very strong separation between the model and the user interface elements.
  - Support for concurrent components.
  - Support for network-distributed applications.

Typical uses: Reactive, heterogeneous applications. Applications demanding low-cost adaptability.

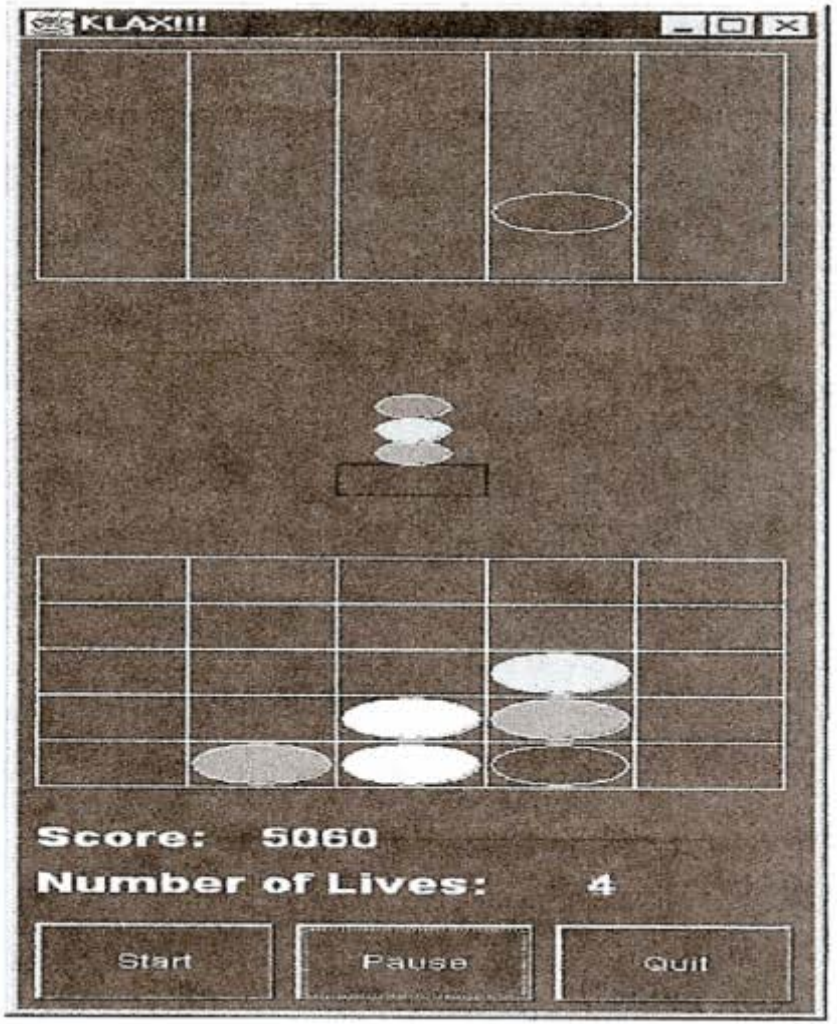
Cautions: Event-routing across multiple layers can be inefficient. Overhead high for some simple kinds of component interaction.

Relations to programming languages or environments: Programming frameworks are used to facilitate creation of implementations faithful to architectures in the style. Support for Java, C, Ada.

C2

Figure 4-24.
Screenshot of example KLAX-like game.

KLAX – game developed by Atari Games Co. in 1990



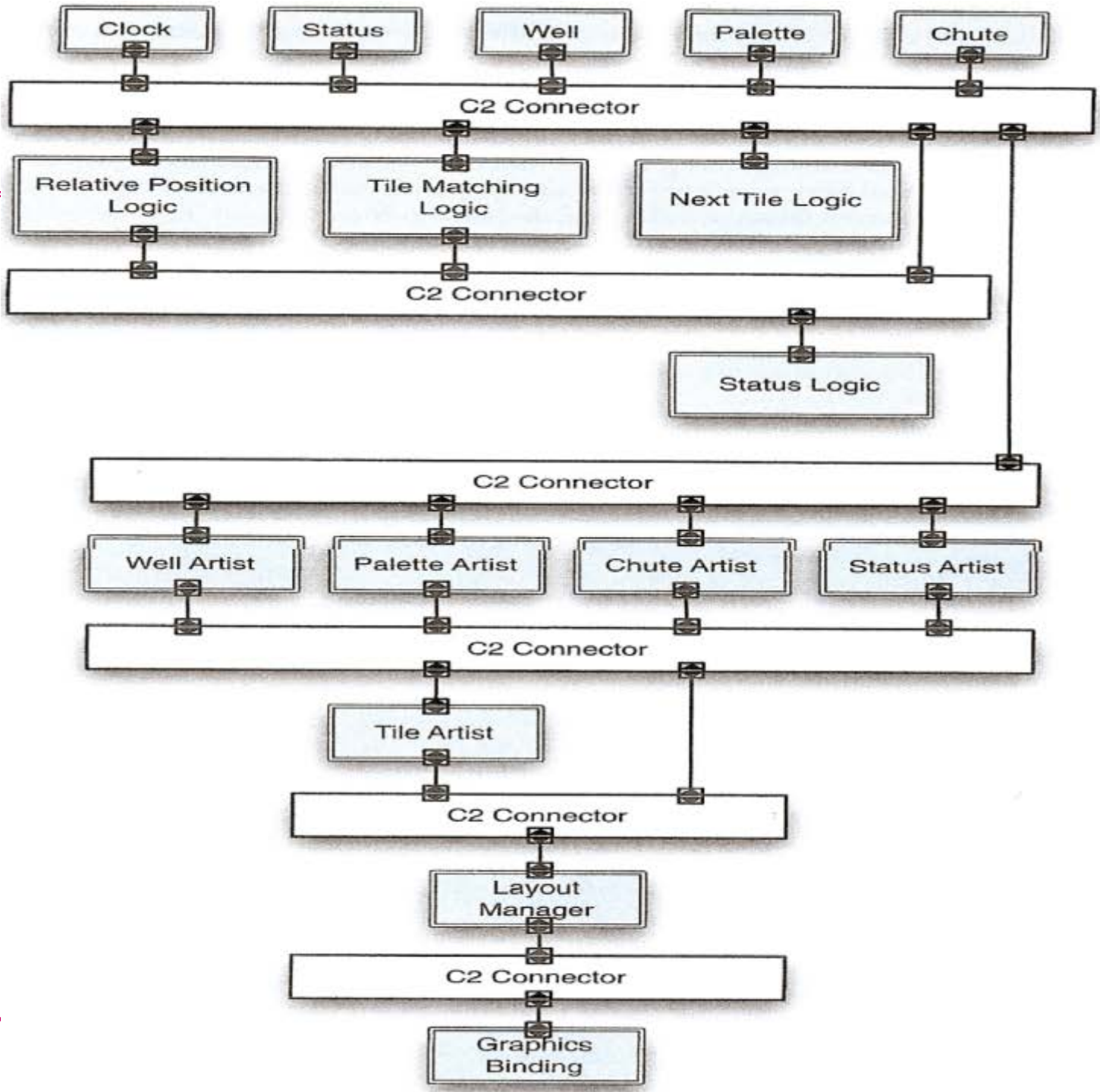


Figure 4-25. A KLAX-like video game in the C2 style.

If a tile is dropped, Relative Position Logic determines if the pallet is in a position to catch the tiles.

If so, a request is sent to Palette, adding the tile to the palette.

Otherwise, a notification is sent that a tile has been dropped, which is detected by Status logic, causing the number of lives to be decreased.

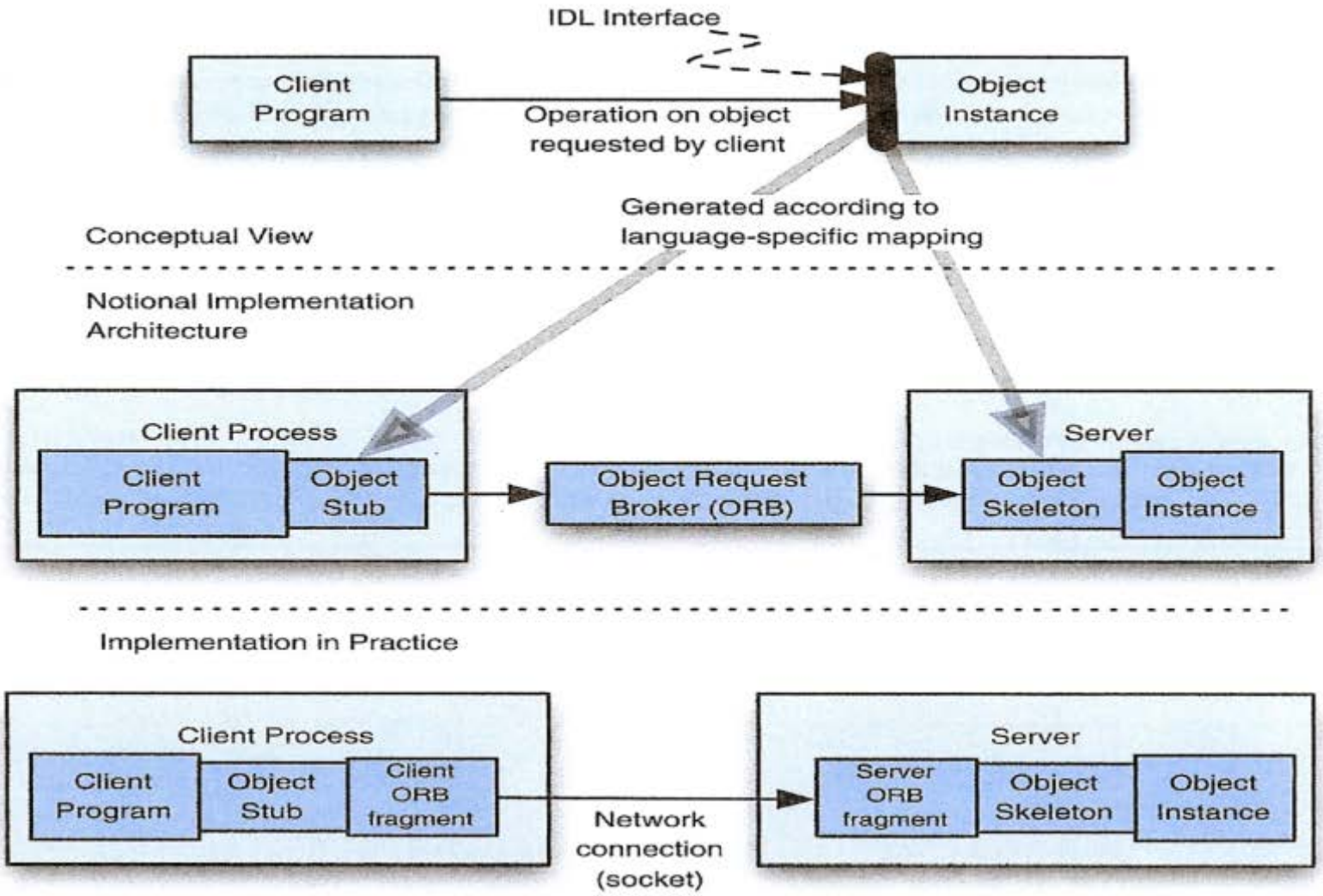
3.2 Distributed Objects

Summary: Application functionality broken up into objects (coarse- or fine-grained) that can run on heterogeneous hosts and can be written in heterogeneous programming languages. Objects provide services to other objects through well-defined provided interfaces. Objects invoke methods across host, process, and language boundaries via remote procedure calls (RPCs), generally facilitated by middleware.
Components: Objects (software components exposing services through well-defined provided interfaces).
Connector: Remote procedure calls (remote method invocations).
Data elements: Arguments to methods, return values, and exceptions.
Topology: General graph of objects from callers to callees; in general, required services are not explicitly represented.
Additional constraints imposed: Data passed in remote procedure calls must be serializable. Callers must deal with exceptions that can arise due to network or process faults.
Qualities yielded: Strict separation of interfaces from implementations as well as other qualities of object-oriented systems in general, plus mostly-transparent interoperability across location, platform, and language boundaries.

Typical uses: Creation of distributed software systems composed of components running on *different hosts. Integration of software components written in different programming languages* or for different platforms.
Cautions: Interactions tend to be mostly synchronous and do not take advantage of the concurrency present in distributed systems. Users wanting services provided by middleware (network, location, and/or language transparency) often find that the middleware induces the distributed objects style on their applications, whether or not it is the best style. Difficulty dealing with streams and high-volume data flows.
Relations to programming languages or environments: Implementations in almost every programming language and environment.

3.2 Distributed Objects

Figure 4-26. *Notional CORBA diagram.*



CORBA connects object across machine, language, and process boundaries.

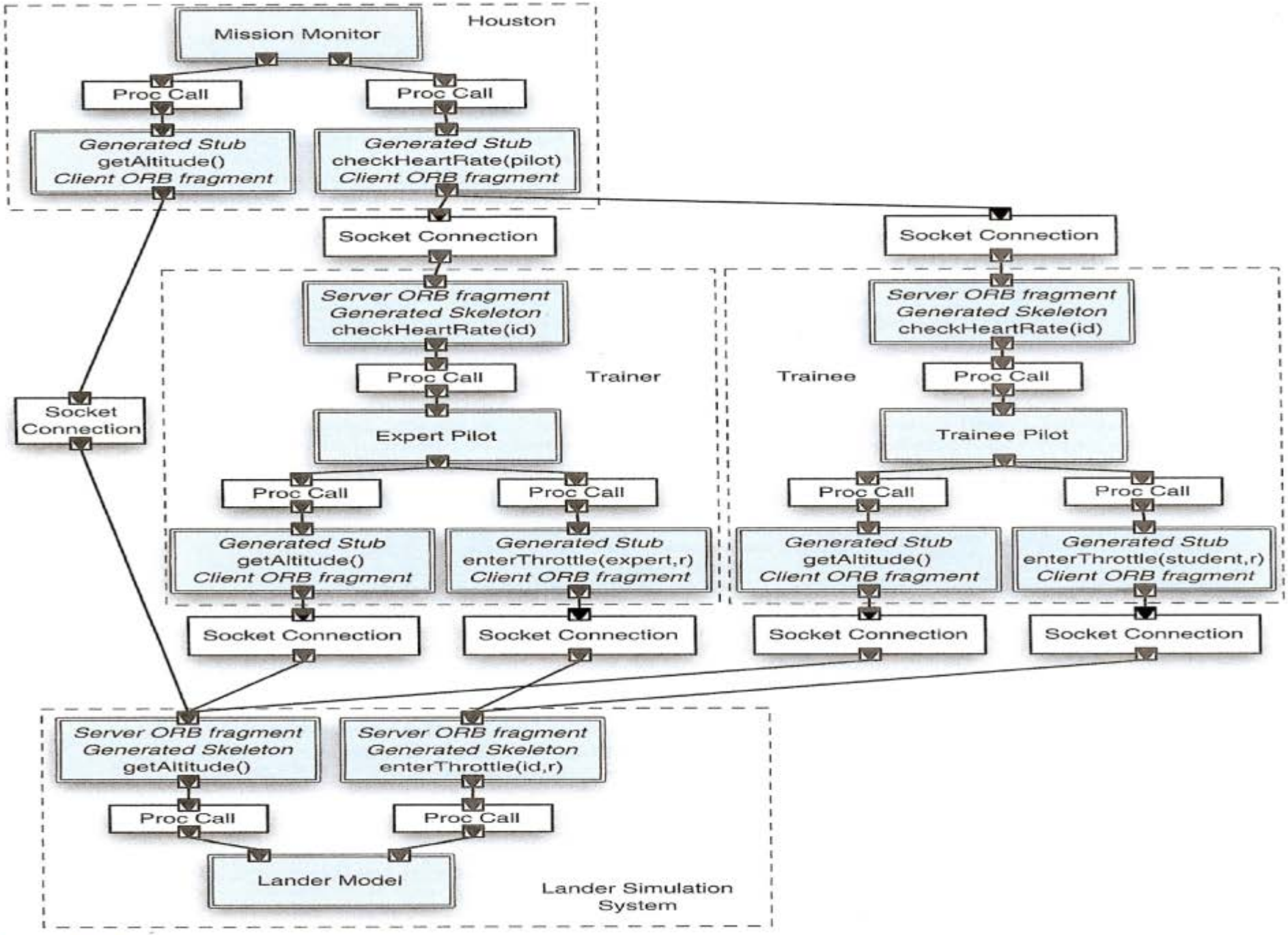


Figure 4-27. *Training version of Lunar Lander, in CORBA. Two pilots can have their hands on the controls.*

3.2 Distributed Objects

The Big Ball of Mud architectural style has been described by Brian Foote and Joseph Yoder as the most common architectural style of all (Foote and Yoder 1997). The style imposes no constraints, other than "get the job done." It offers the benefits of, "We didn't have to think too much", and "It'll work for now. I hope." Software in the BBoM style is haphazard and thrown together. Such systems often grow by accretion: New bits of code are stuck onto preexisting code to meet new demands, without discipline, plan, or care. The BBoM style is an important style for designers to keep in mind for one simple but critical reason: "If you can't articulate why your application is *not* a big ball of mud, then it is."

4. Case Studies

4.1 Key Words in Context

4.2 Cruise Control

Acknowledgement.

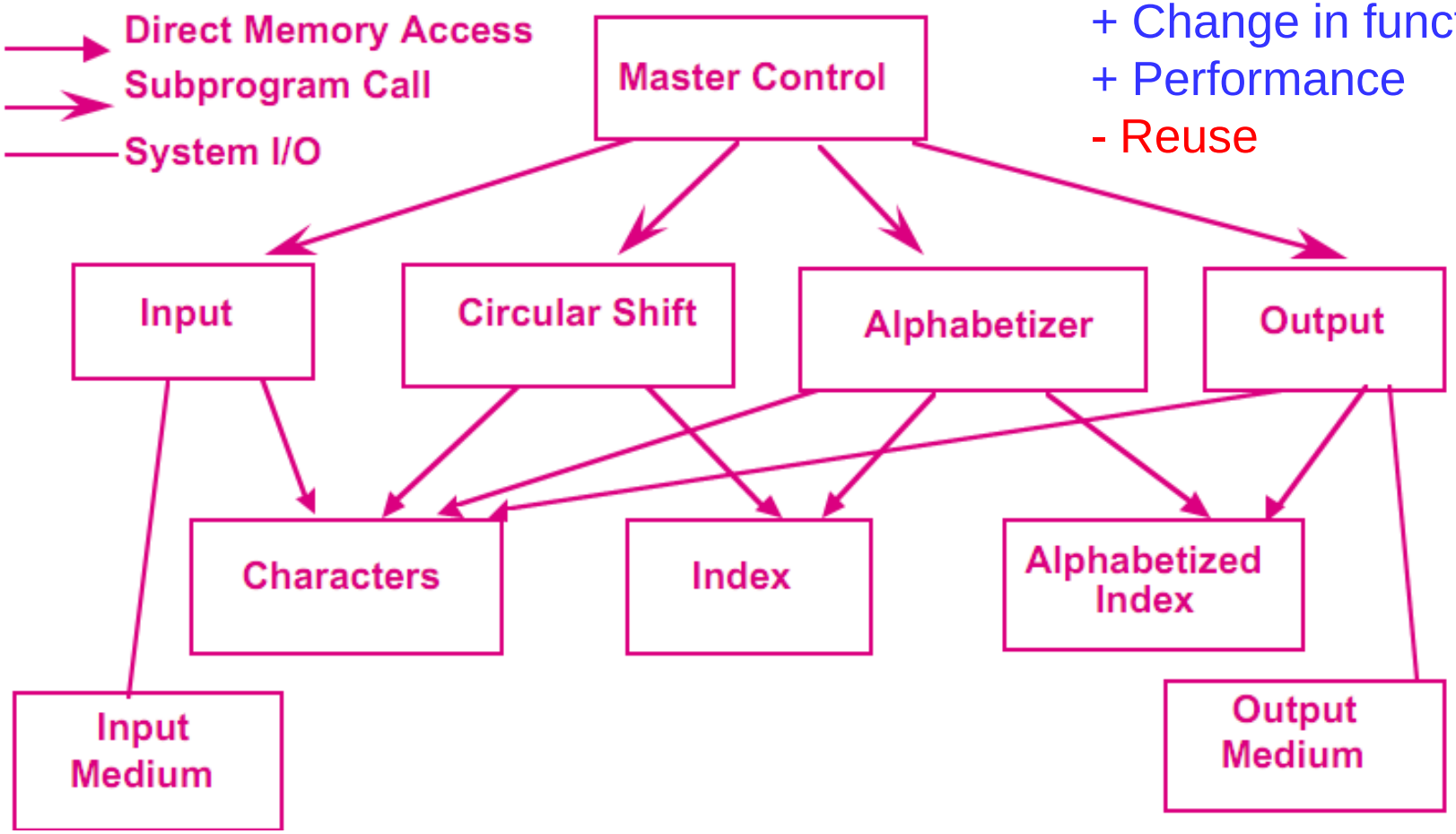
Based on [Shaw 96] "Software Architecture: Perspectives on an Emerging Discipline."

4.1 Key Word in Context

- The KWIC [Key Word in Context] index system:
  - Make full text search
  - Forms by sorting/aligning words within article title
    - Make them searchable
  - Any line may be 'circularly shifted'' by repeatedly removing the first word and appending it at the end of the line
- Considerations of software architecture:
  - Effect of changes on software design
    - Changes in processing algorithm
    - Changes in data representation
  - Analysis:
    - Enhancement to system function:
      - E.g. modify circular shifts; change system to be interactive
    - Performance: Both space and time
    - Component reuse

Solution 1: Main Program/Subroutine with Shared Data

- Change in algorithm
- Change in data rep
+ Change in function
+ Performance
- Reuse



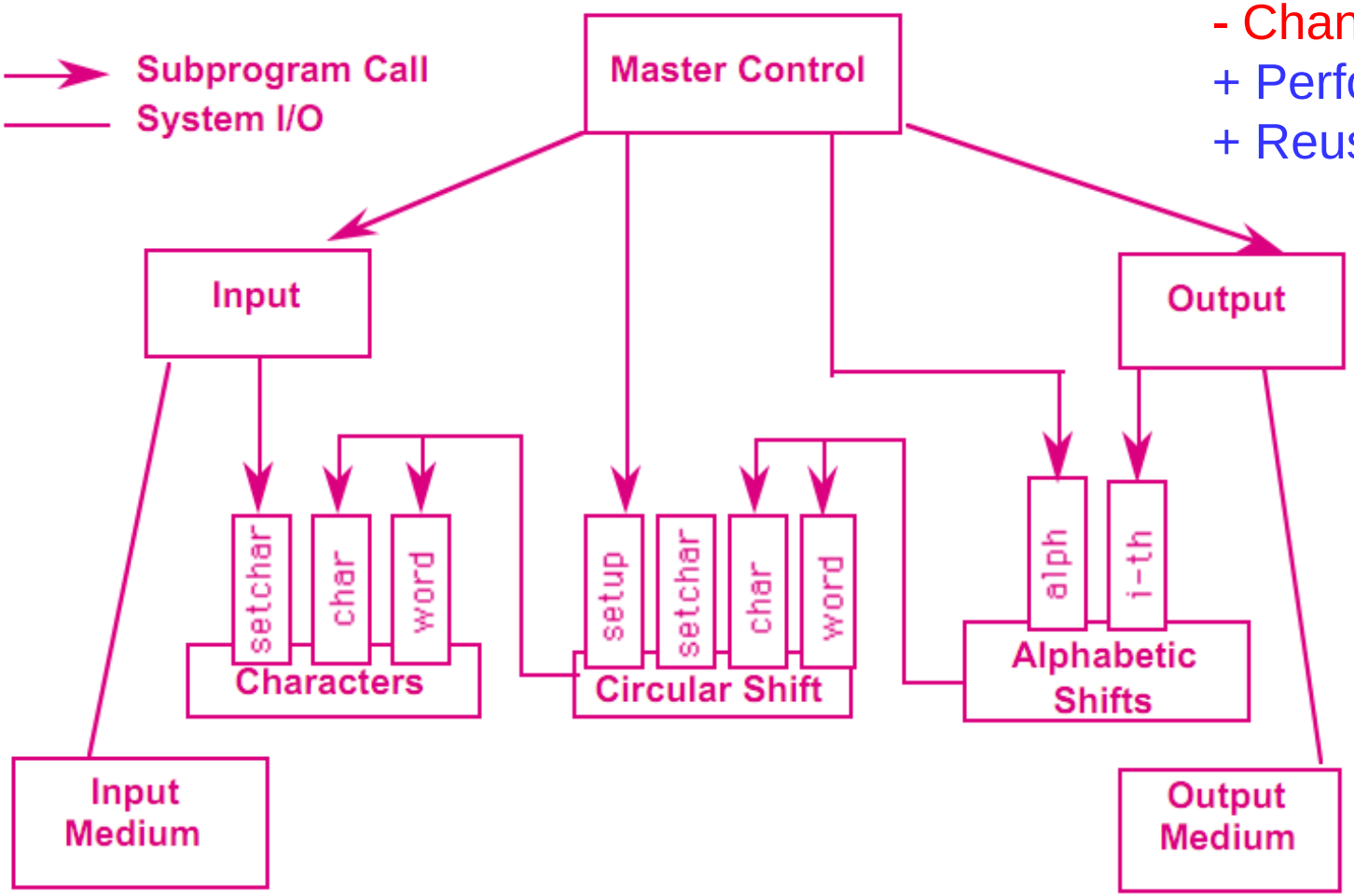
Solution 1: Main Program/Subroutine with Shared Data

- Decompose into the four basic functions: input, shift, alphabetize, and output.
- Data is communicated between the components through shared storage ("core storage").
- Communication between computational components and the shared data is an unconstrained read/write protocol.
- The solution → distinct computational aspects are isolated in different modules.
- Drawback:
  - No ability to handle changes.

    Example Data storage format affects all components.
  - Not support reuse

Solution 2: Abstract Data Types

- Change in algorithm
+ Change in data rep
- Change in function
+ Performance
+ Reuse

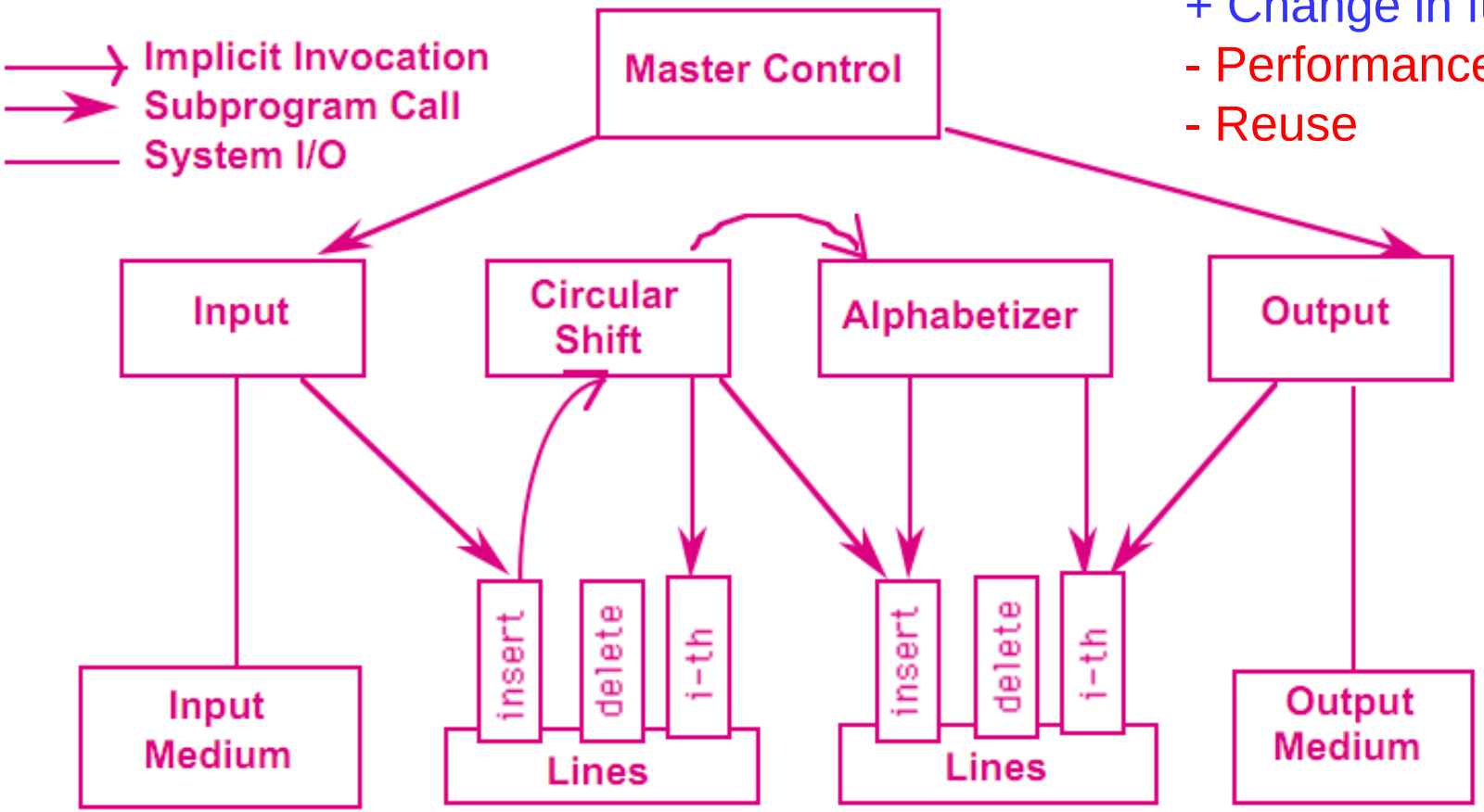


Solution 2: Abstract Data Types

- Decomposes the system into five modules
  - No directly shared data.
  - Module provides an interface that permits other components to access data only by invoking procedures in that interface.
- Advantages:
  - Concerns design changes such as both algorithms and data representations
  - Better support reuse than Solution 1
- Disadvantages:
  - No well-suited to function enhancements:
  - To add new functions to the system, the programmer must either
    - Modify the existing modules.
    - Add new modules that lead to performance penalties

Solution 3: Implicit Invocation

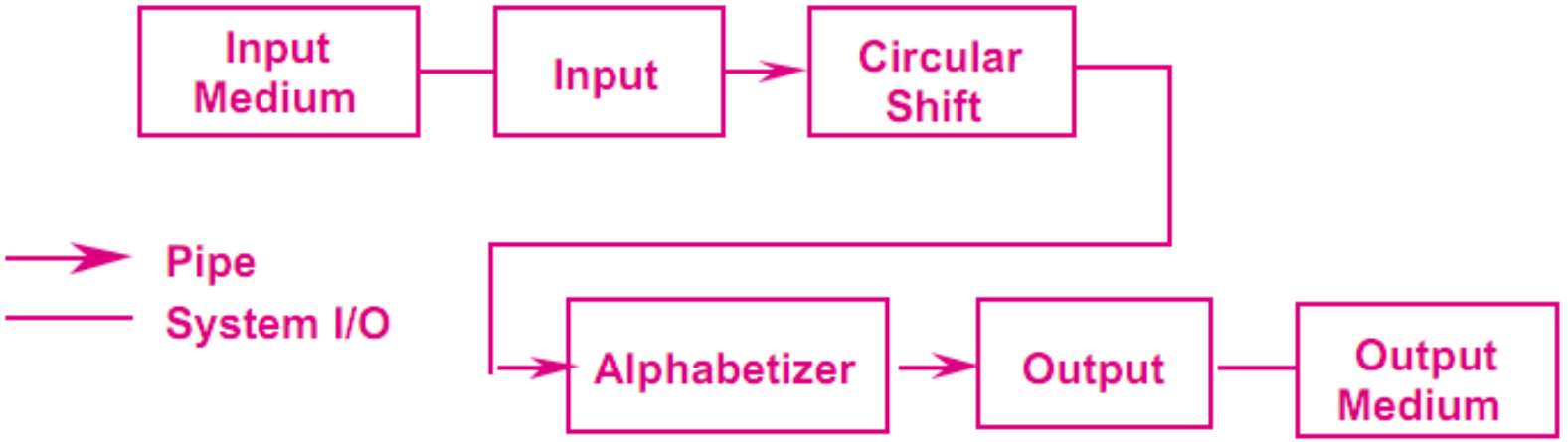
+ Change in algorithm
- Change in data rep
+ Change in function
- Performance
- Reuse



Solution 3: Implicit Invocation

- Design with a form of component integration based on shared data:
  - Similar to Solution 1.
  - Two important difference:
    1. Interface to the data is more abstract. (No direct access)
    2. Computations are invoked implicitly as data is modified
       (다른 component들은 모르게 수정)
- Advantages:
  - Ease to support functional enhancements.
  - Additional modules can be attached by registering and invoking on data-changing events.
  - Reuse is also supported
- Disadvantages:
  - Difficult to control the order of processing.
  - Invocations are data driven, use more space than the previous ones

Solution 4: Pipes and Filters



+ Change in algorithm
- Change in data rep
+ Change in function
- Performance
+ Reuse

Solution 4: Pipes and Filters

- A pipeline solution: 4 filters - input, shift, alphabetize, output
  - Each filter processes the data and sends it to the next filter.
  - Control is distributed.
- Advantages:
  - It maintains the intuitive flow of processing
  - It supports reuse, since each filter can function in isolation
  - New functions are easily added to the system by inserting filters at the appropriate point
  - It supports ease of modification, since filters are logically independent of other filters
- Disadvantages:
  - Impossible to modify the design to support an interactive system
  - Inefficient in use of space, since each filter must copy all of the data to its output ports.

Comparison

	1. Shared Data	2. ADT	3. Implicit Invocation	4. Pipe & Filter
Change in Algorithm	–	–	+	+
Change in Data Rep	–	+	–	–
Change in Function	+	–	+	+
Performance	+	+	–	–
Reuse	–	+	–	+

Comparison

- **Shared data solution**
  - Weak: support for changes in the overall processing algorithm, data representations, and reuse.
  - Strong: relatively good performance, adding a new processing component
- **Abstract data type solution**
  - Weak: changing the overall processing algorithm or adding new functions
  - Strong: changes to data representation, supports reuse
- **Implicit invocation solution**
  - Weak: Poor support for change in data representation and reuse, extra execution overhead.
  - Strong: adding new functionality
- **Pipe and filter solution**
  - Weak: decisions about data representation depends on the kind of data transmitted along the pipes.
  - Weak: the exchange format, there may be additional overhead involved in parsing and unparsing the data onto pipes
  - Strong: changes in processing algorithm, changes in function

4.2 Cruise Control

- Based on [Shaw 95]
  - Mary Shaw, "Beyond Objects: A Software Design Paradigm Based on Process Control", ACM Software Engineering Notes, Vol 20, No 1, January 1995, pp.27-38.

Process Control Definitions

- *Process variables:* properties of the process that can be measured; several specific kinds are often distinguished. **Do not** confuse process variables with program variables.
- *Controlled variable:* process variable whose value the system is intended to control
- *Input variable:* process variable that measures an input to the process
- *Manipulated variable:* process variable whose value can be changed by the controller
- *Set point:* the desired value for a controlled variable

- *Open loop system:* system in which information about process variables is not used to adjust the system.
- *Closed loop system:* system in which information about process variables is used to manipulate a process variable to compensate for variations in process variables and operating conditions.
- *Feedback control system:* the controlled variable is measured and the result is used to manipulate one or more of the process variables
- *Feedforward control system:* some of the process variables are measured and disturbances are compensated for without waiting for changes in the controlled variable to be visible.

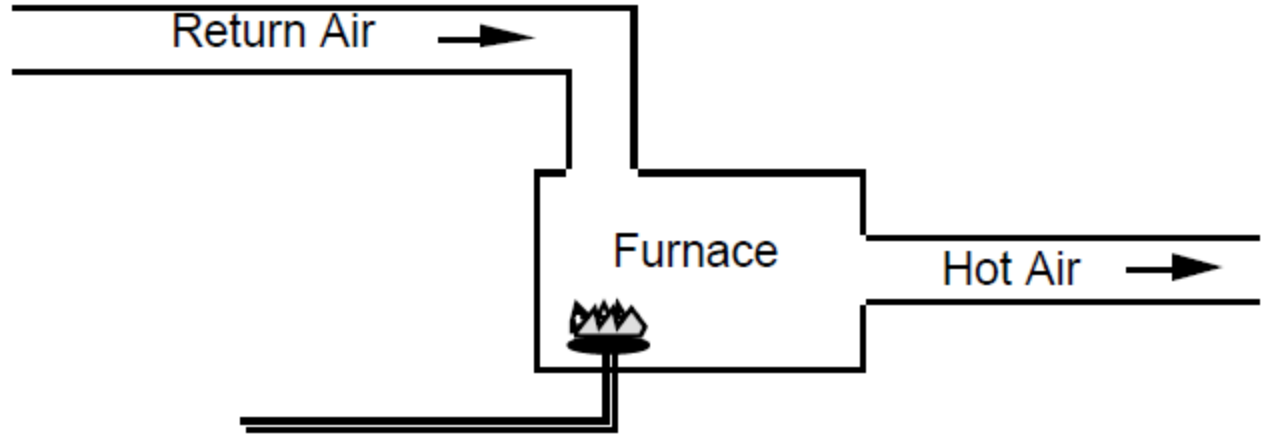


Figure 1: Open loop temperature control

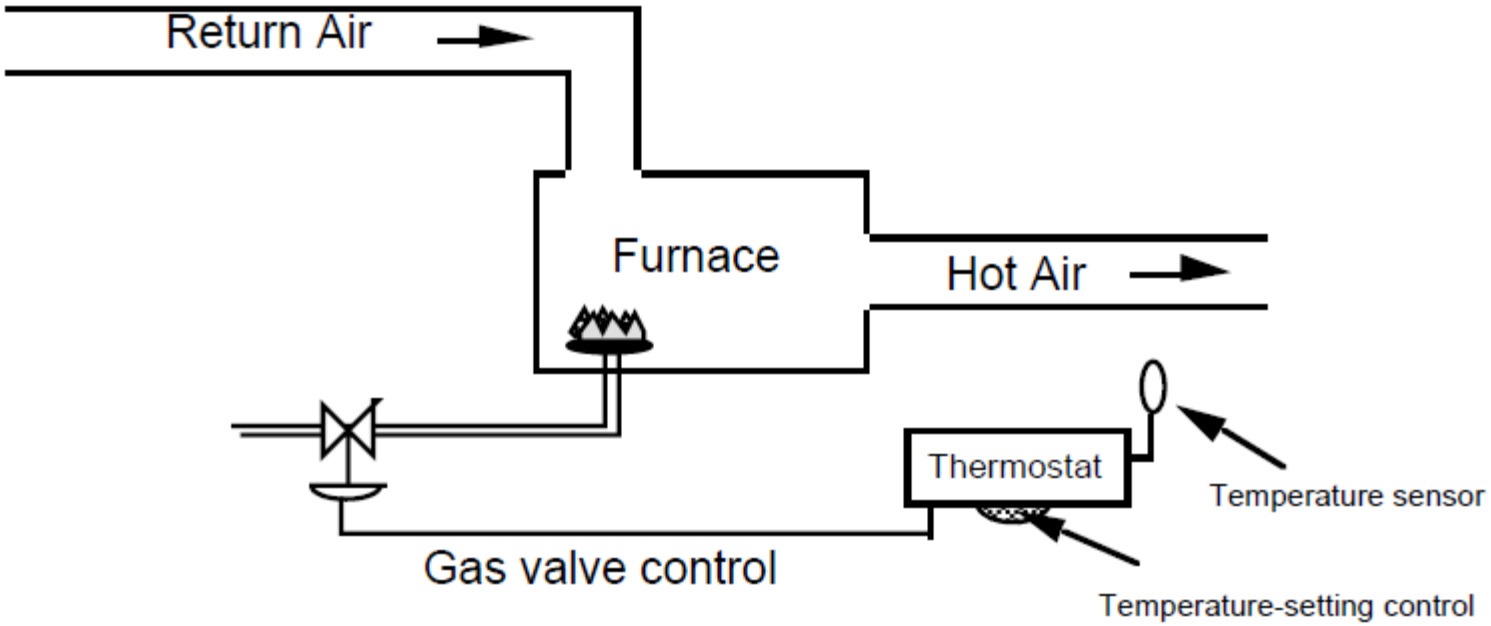


Figure 2: Closed loop temperature control

2 Types of Closed Control

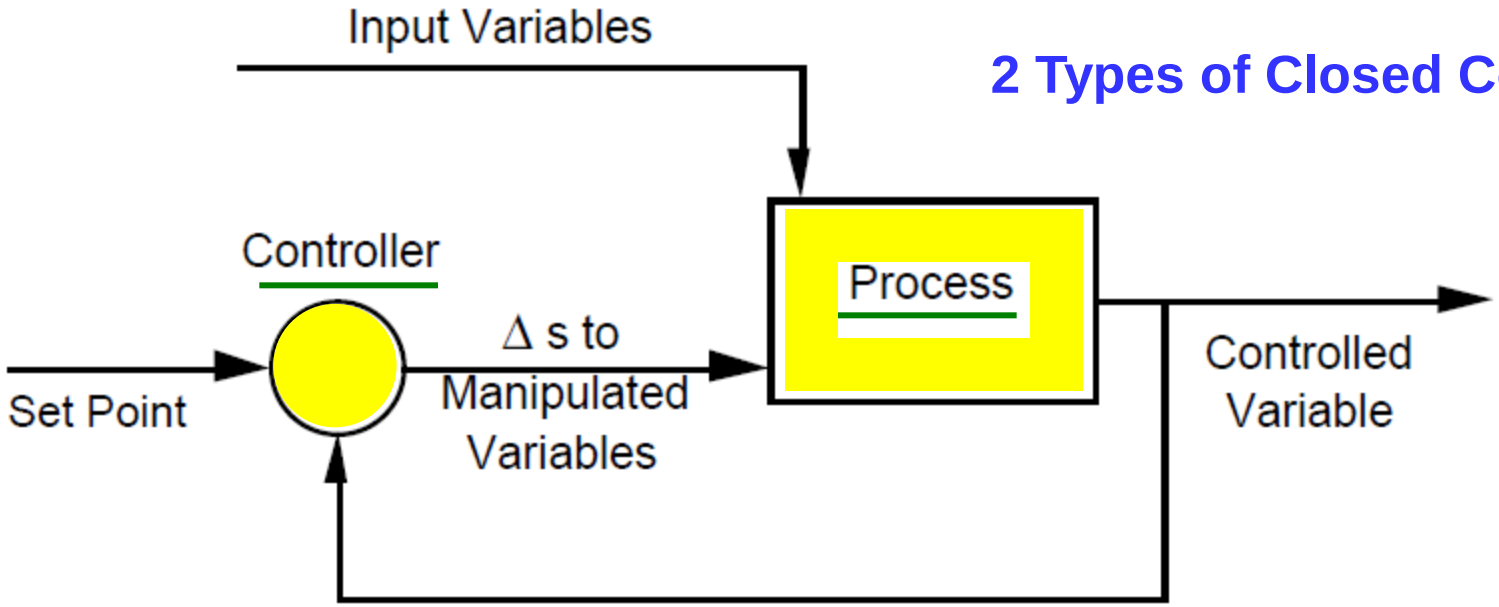


Figure 3: Feedback Control

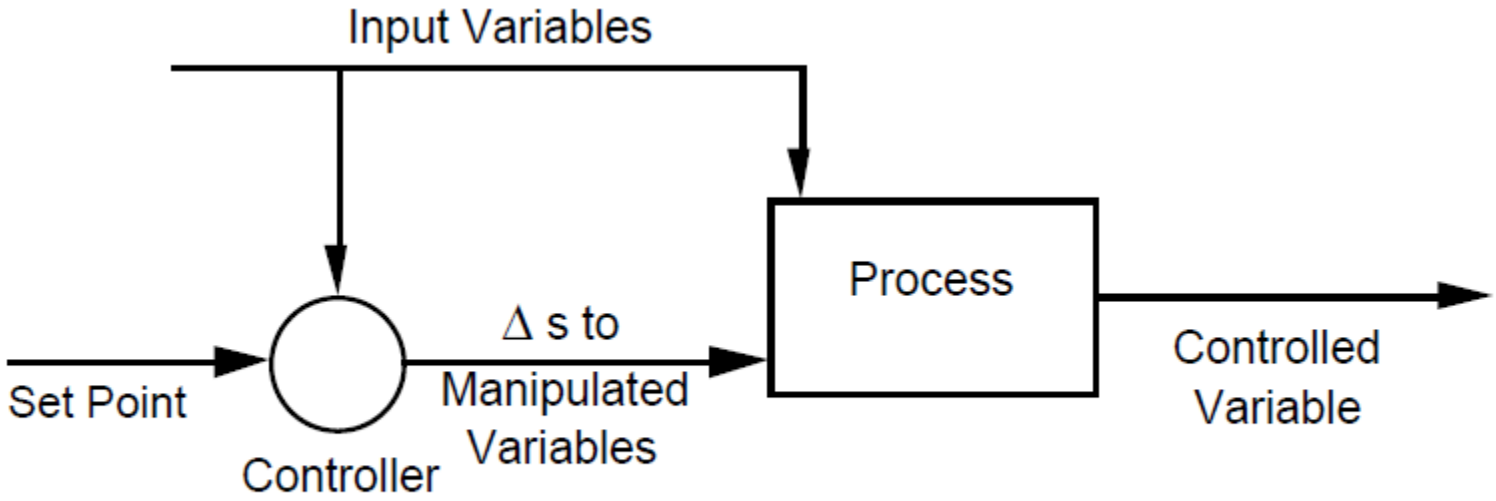


Figure 4: Feedforward Control

Booch Approach

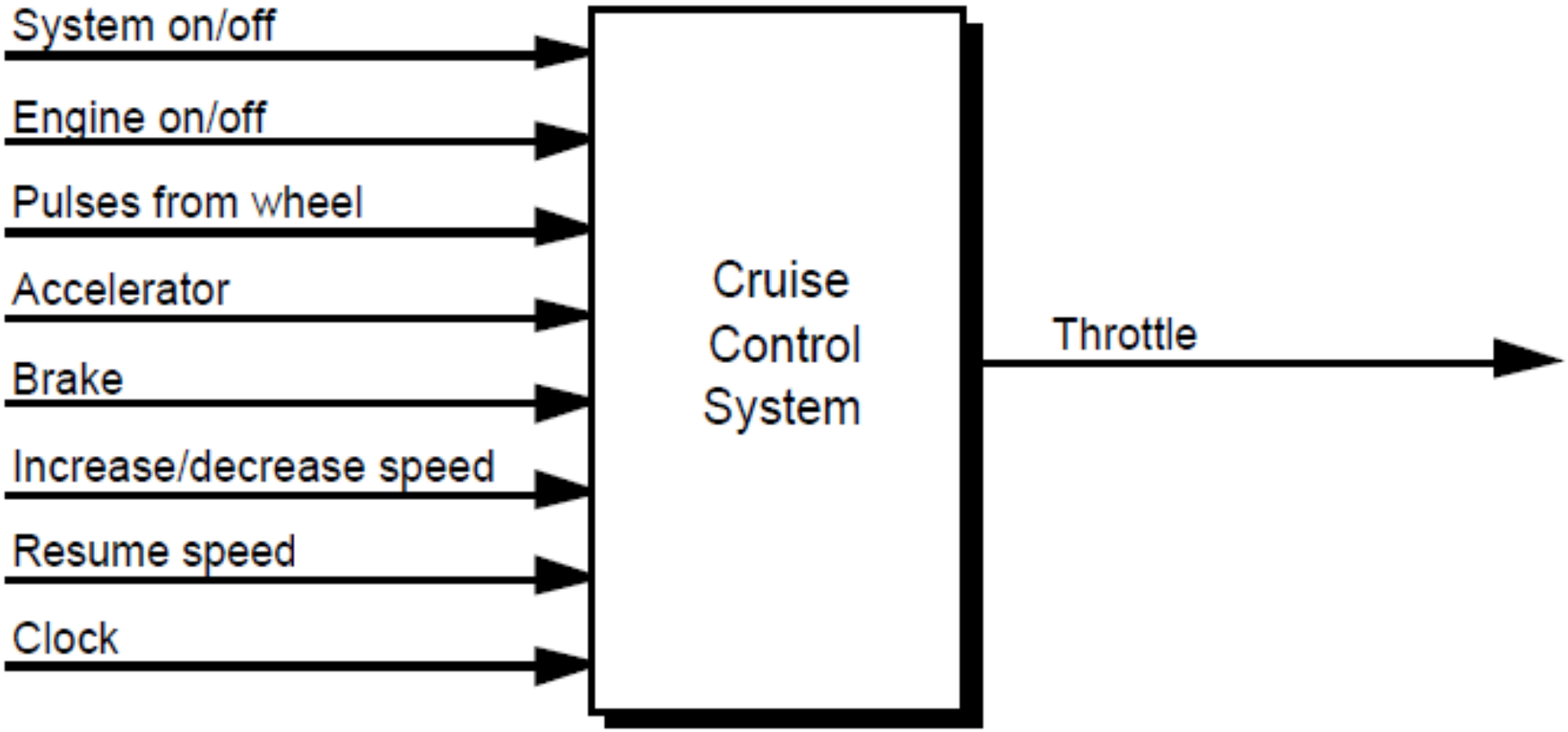


Figure 5: Booch block diagram for cruise control

Booch Approach - Input and Output

Input from the system:

- **System on/off** If on, denotes that the cruise-control system should maintain the car speed.
- **Engine on/off** If on, denotes that the car engine is turned on; the cruise-control system is only active if the engine is on.
- **Pulses from wheel** A pulse is sent for every revolution of the wheel.
- **Accelerator** Indication of how far the accelerator has been pressed.
- **Brake** On when the brake is pressed; the cruise-control system temporarily reverts to manual control if the brake is pressed.
- **Increase/Decrease Speed** Increase or decrease the maintained speed; only applicable if the cruise-control system is on.
- **Resume** Resume the last maintained speed; only applicable if the cruise control system is on.
- **Clock** Timing pulse every millisecond.

Output from the system:

- **Throttle** Digital value for the engine throttle setting.

Booch Approach - Design

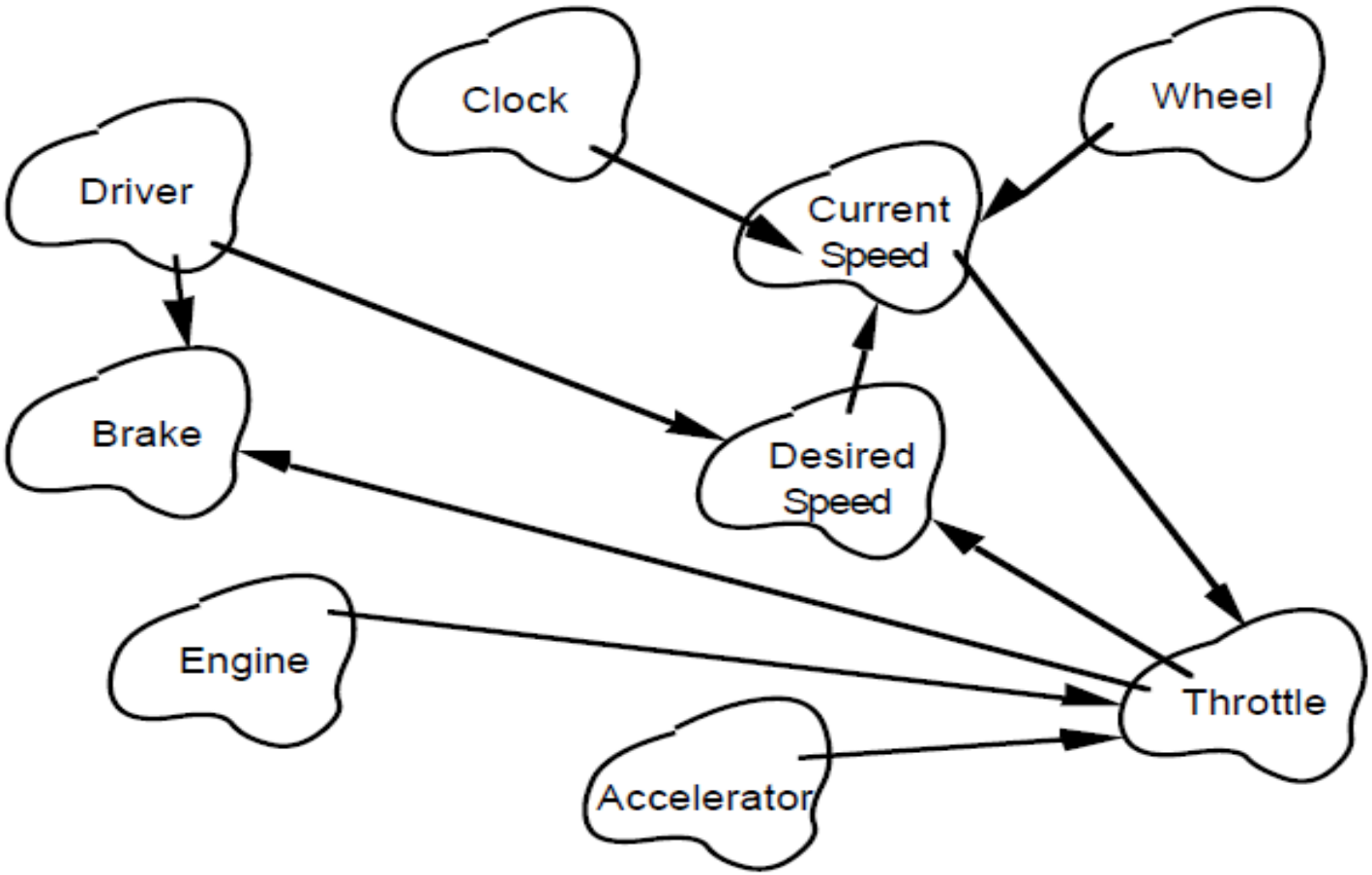


Figure 6: Booch's object-oriented design for cruise control

Architectural Approach

- Essential System Elements for Cruise Control - ***Computational elements***
  - *Process definition:*
    - The process takes a throttle setting as input and controls the speed of the vehicle.
  - *Control algorithm:*
    - Models the **current speed** based on the **wheel pulses**, compares it to the **desired speed**, and changes the throttle setting.
    - The **clock** input is needed to model **current speed** based on intervals between **wheel pulses**.
    - The policy decision about how much to change the **throttle** setting for a given discrepancy between **current speed** and **current speed** is localized in the control algorithm.

Architectural Approach

- Essential System Elements for Cruise Control - ***Data elements***
  - *Controlled variable:* The **current speed** of the vehicle.
  - *Manipulated variable:* The **throttle** setting.
  - *Set point:*
    - The **desired speed** is set and modified by the **accelerator** input and the **increase/decrease speed** input, respectively.
    - Several other inputs determine whether the cruise control is currently controlling the car: **System on/off**, **engine on/off**, **brake**, and **resume**.
      - **resume** restores automatic control, but only if the entire system is on.
    - These inputs are provided by the human driver (the operator, in process terms).
  - *Sensor for controlled variable:*
    - The current state is the **current speed**, which is modeled on data from a sensor that delivers **wheel pulses** using the **clock**.

Architectural Approach - Design

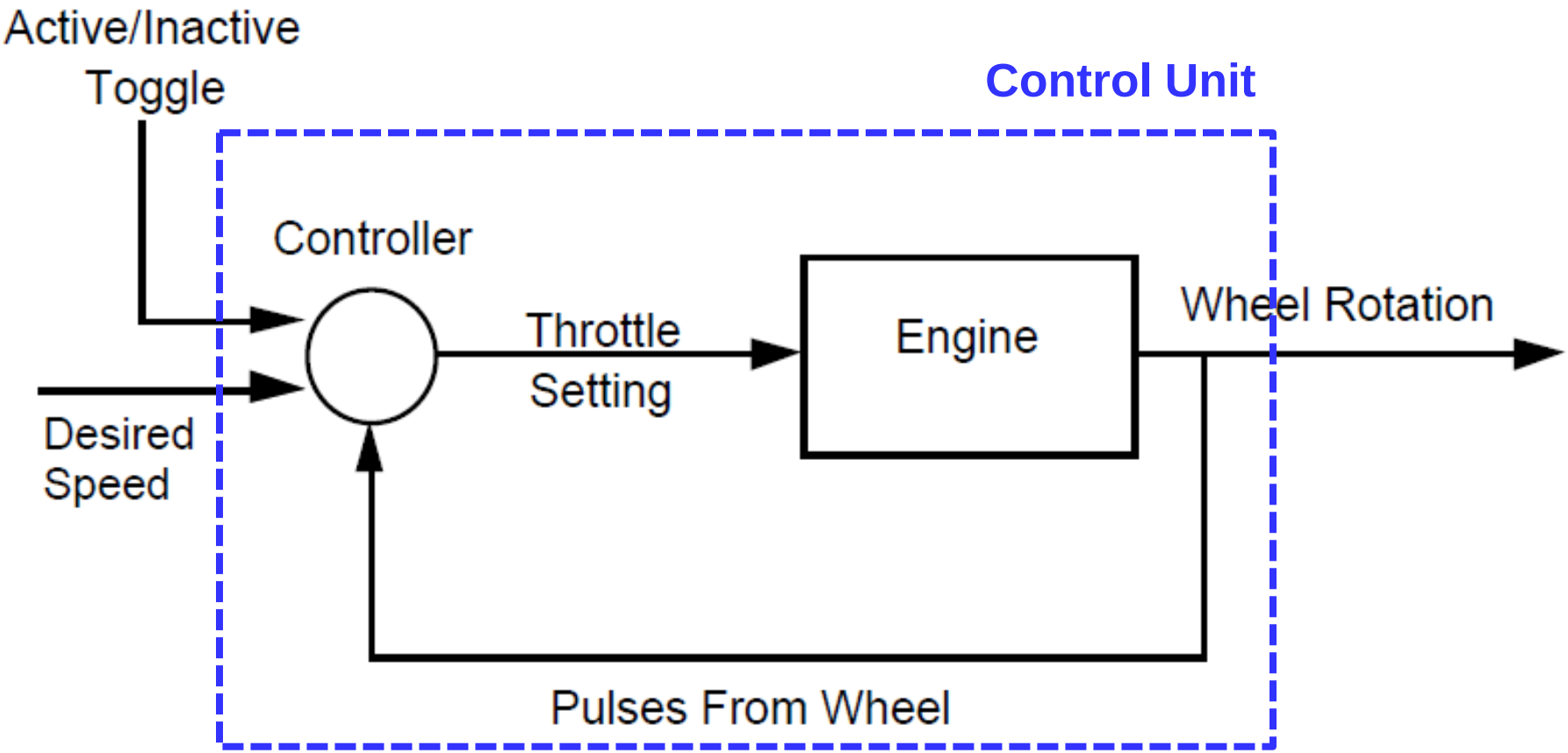


Figure 7: Control Architecture for Cruise Control

Architectural Approach - Design

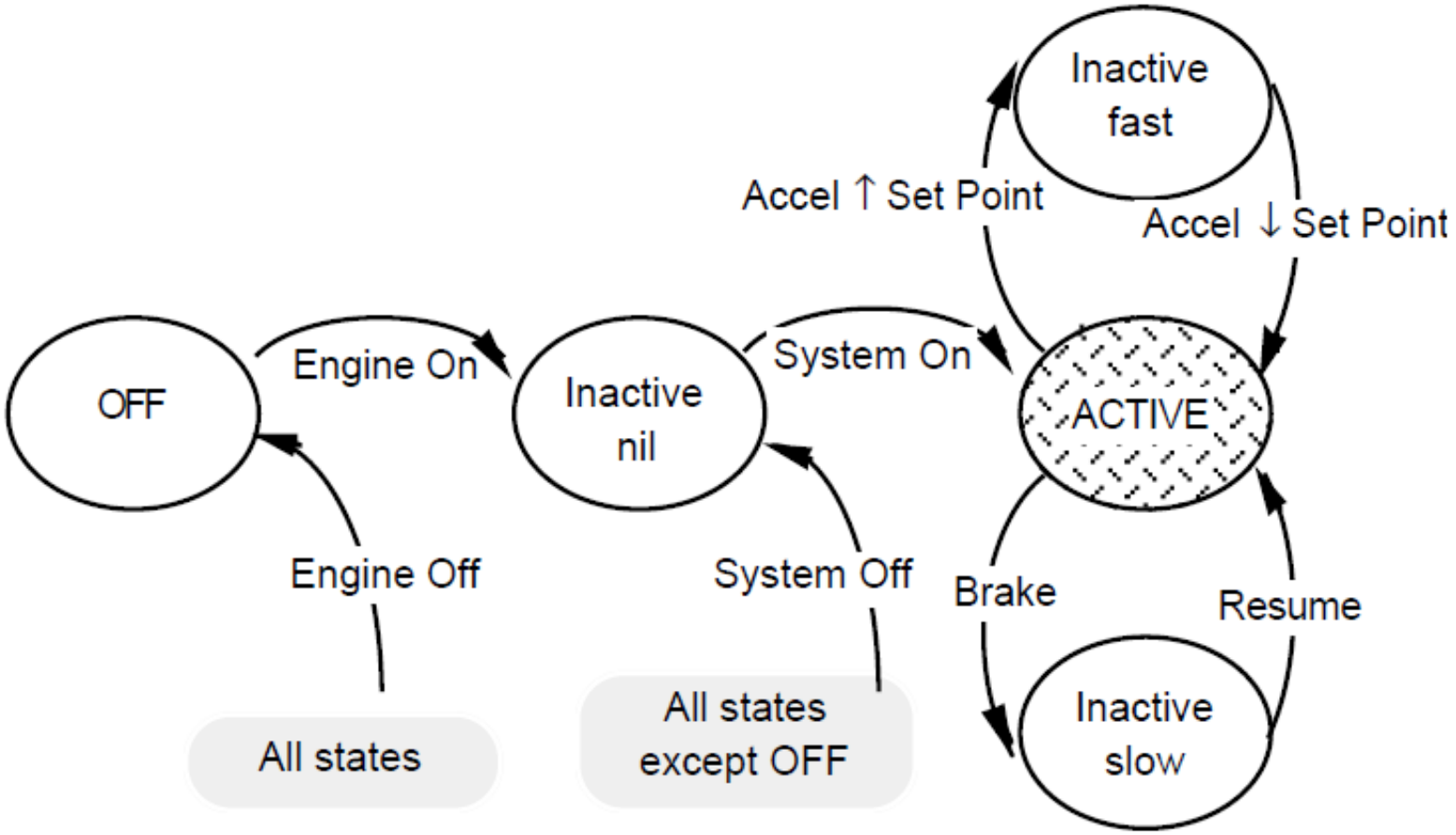


Figure 8: State Machine for Activation

Architectural Approach – Design

- Event Table for Determining Set Point
  - *Event: Effect on desired speed*
    - **Engine off**, **system off:** Set to “undefined”
    - **System on:** Set to current speed as estimated from wheel pulses
    - **Increase speed:** Increment desired speed by constant
    - **Decrease speed:** Decrement desired speed by constant

Architectural Approach – Design

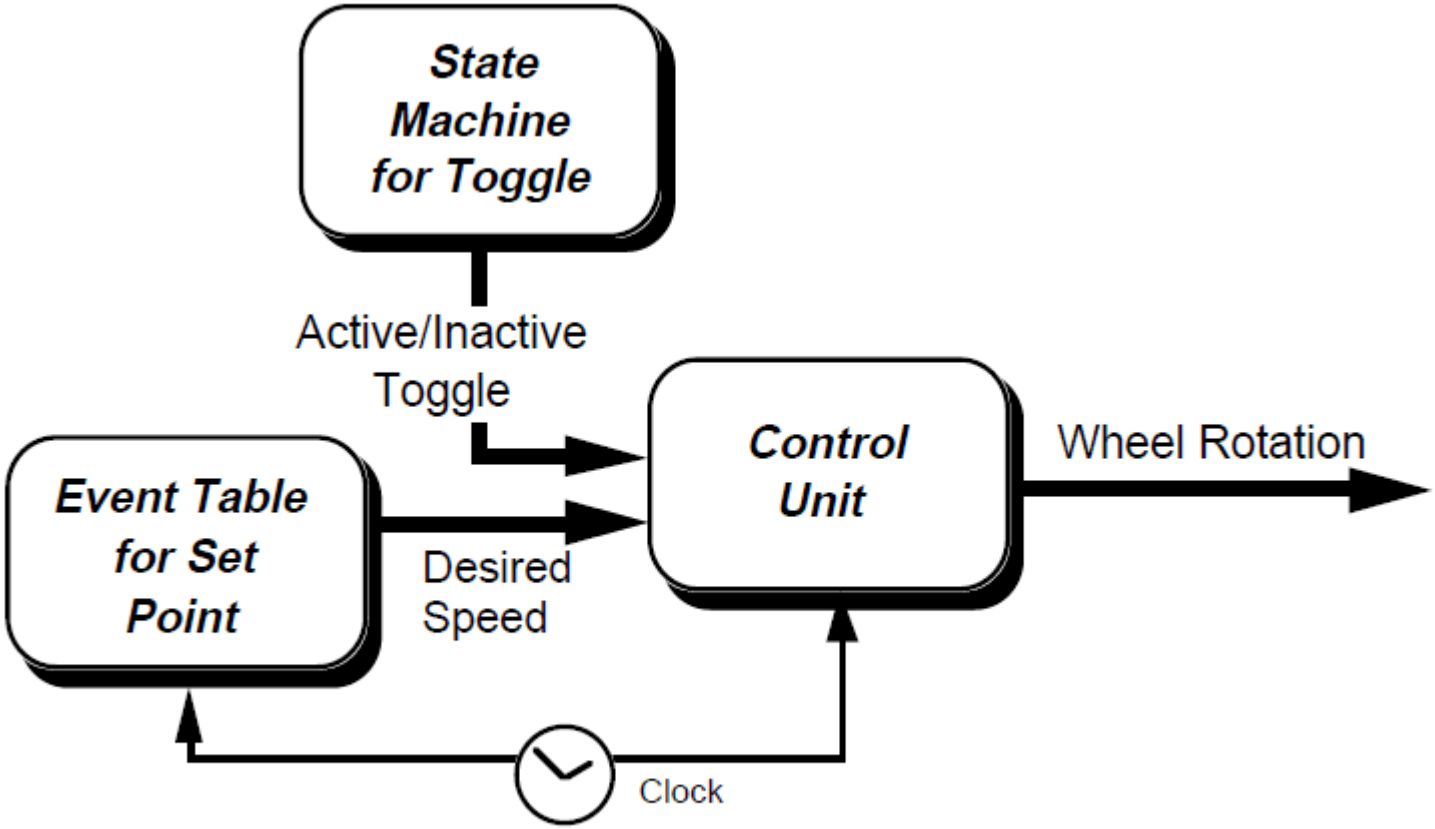


Figure 10: Complete cruise control system

Analysis and Discussion

- Explored an example in which the significant high-level decisions are better elicited by a *methodology based on process control* than on the OO methodology.
- Exemplifies a class of software system design problems in which a real-time process is controlled by embedded software.
  - Conceptually, such processes update the control status continuously.
- Thinking about these designs explicitly as process control problems leads the designer to a software organization that

  (1) separates ***process concerns*** from ***control concerns*** and

  (2) requires explicit attention to the appropriateness and correctness of the control strategy.

  => Early consideration of performance and correctness questions

Questions?