Software design

Architectural Styles[†]

[†]Material from textbook by Shaw and Garlen; slides by Spiros Mancoridis (modified by Grant Weddell)

Definition

An **architectural style** defines a family of systems in terms of a pattern of structural organization. It determines:

- The **vocabulary** of components and connectors that can be used in instances of that style; and
- A set of **constraints** on how they can be combined. For example, one might constrain
  - the topology of the descriptions (*e.g.*, no cycles), or
  - execution semantics (*e.g.*, processes execute in parallel).

Determining an architectural style

We can understand what a style is by answering the following questions.

- What is the **structural pattern** (*i.e.*, components, connectors, 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** of that style?

Describing an architectural style

The architecture of a specific system is a collection of

- computational components, and
- descriptions of the interactions between these components (connectors).

Describing an architectural style (cont'd)

Software architectures are represented as graphs where **nodes** represent components,

- *procedures*
- *modules*
- *processes*
- *tools*
- *databases*

and **edges** represent connectors.

- *procedure calls*
- *event broadcasts*
- *database queries*
- *pipes*

Styles reviewed (from text)

- Pipes and Filters
- Object-Oriented
- Implicit Invocation
- Layered
- Repositories
- Interpreters
- Process-Control

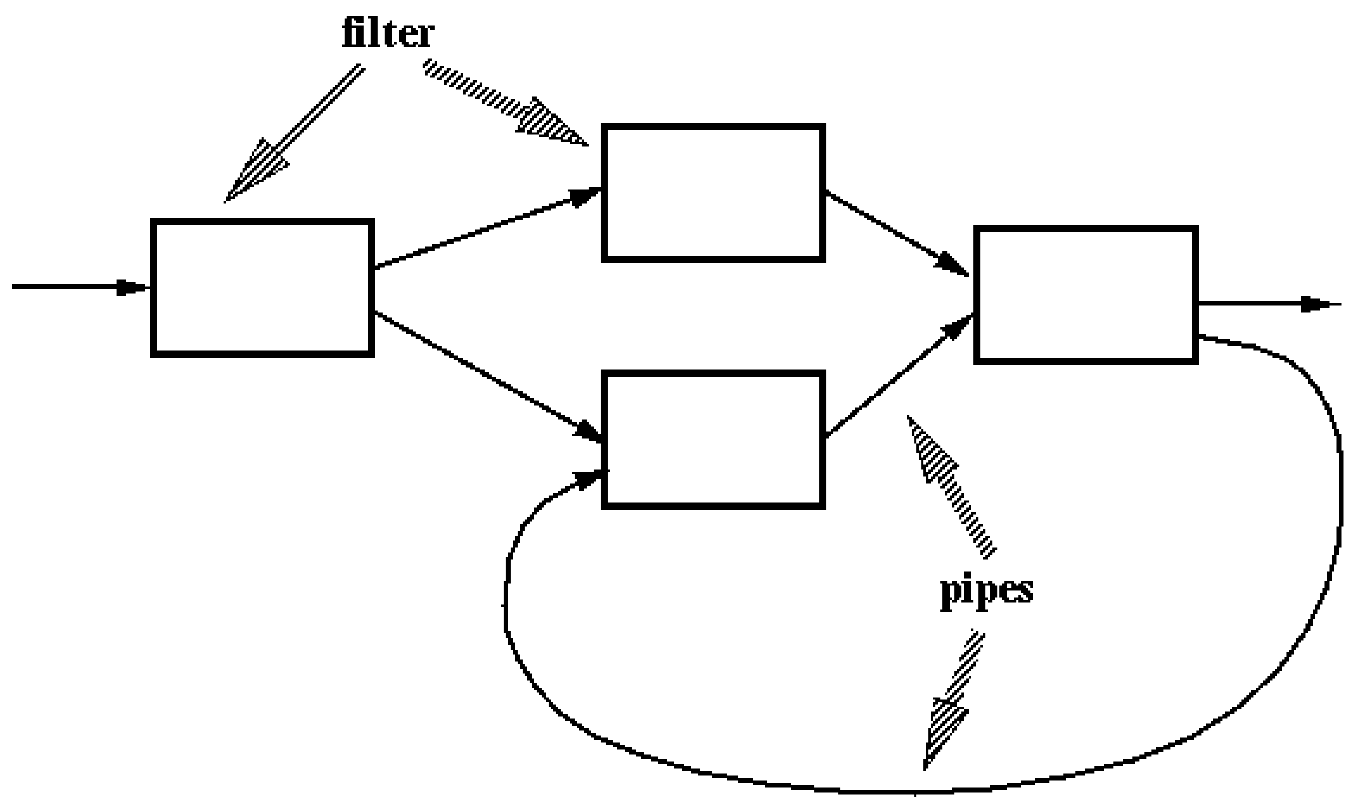
Pipe and Filter

- Suitable for applications that require a defined series of independent computations to be performed on ordered data.
- A component reads streams of data on its inputs and produces streams of data on its outputs.
- Very little feedback available from later operations applied to data.

Pipe and Filter (cont'd)

- **<u>Components:</u>** called **filters**, apply local transformations to their input streams and often do their computing incrementally so that output begins before all input is consumed.
- **<u>Connectors:</u>** called **pipes**, serve as conduits for the streams, transmitting outputs of one filter to inputs of another.

Pipe and Filter (cont'd)



Pipe and Filter invariants

- Filters **do not share state** with other filters.
- Filters **do not know the identity** of their upstream or downstream filters.
- The **correctness** of the output of a pipe and filter network should not depend on the order in which their filters perform their incremental processing.

Pipe and Filter specializations

- **Pipelines:** Restricts topologies to linear sequences of filters.
- **Batch Sequential:** A degenerate case of a pipeline architecture where each filter processes all of its input data before producing any output.

Pipe and Filter examples

Unix Shell Scripts: Provides a notation for connecting Unix processes via pipes.

- `cat file | grep Erroll | wc -l`

Traditional Compilers: Compilation phases are pipelined, though the phases are not always incremental. The phases in the pipeline include:

- *lexical analysis* + *parsing* + *semantic analysis* + *code generation*

Pipe and Filter advantages

- **Easy to understand** the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters.
- They **support reuse**, since any two filters can be hooked together, provided they agree on the data that is being transmitted between them.

Pipe and Filter advantages (cont'd)

- Systems can be **easily maintained and enhanced**, since new filters can be added to existing systems and old filters can be replaced by improved ones.
- They permit certain kinds of **specialized analysis**, such as throughput and deadlock analysis.
- They naturally **support concurrent execution**.

Pipe and Filter disadvantages

- Not good for handling **interactive systems**, because of their transformational character.
- Excessive parsing and unparsing leads to **loss of performance** and **increased complexity** in writing the filters themselves.

Case study: architecture of a compiler

Hybrid compiler architectures

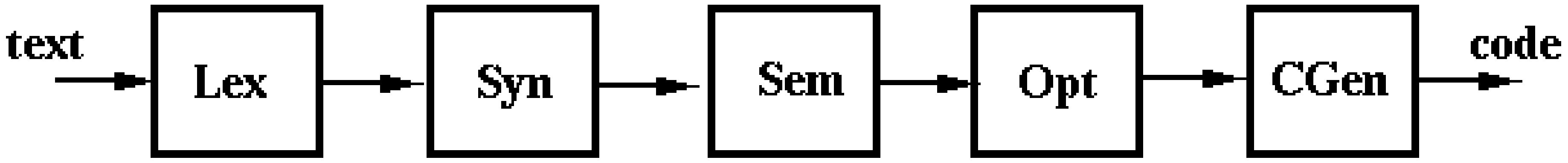
- The new view accommodates various tools (*e.g.*, syntax-directed editors) that operate on the internal representation rather than the textual form of a program.
- Architectural shift to a **repository** style, with elements of the **pipeline** style, since the order of execution of the processes is still predetermined.

Architecture of a compiler

The architecture of a system can change in response to improvements in technology. This can be seen in the way we think about compilers.

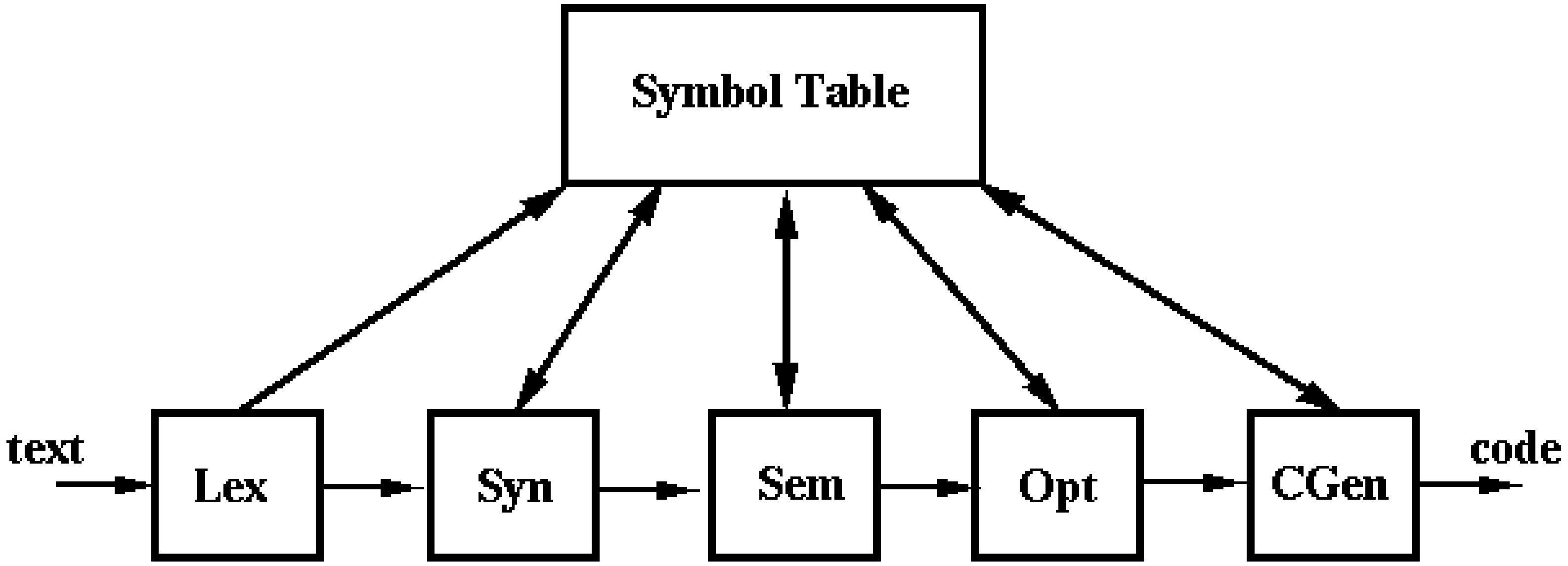
Early compiler architectures

In the 1970s, compilation was regarded as a sequential (batch sequential or pipeline) process:



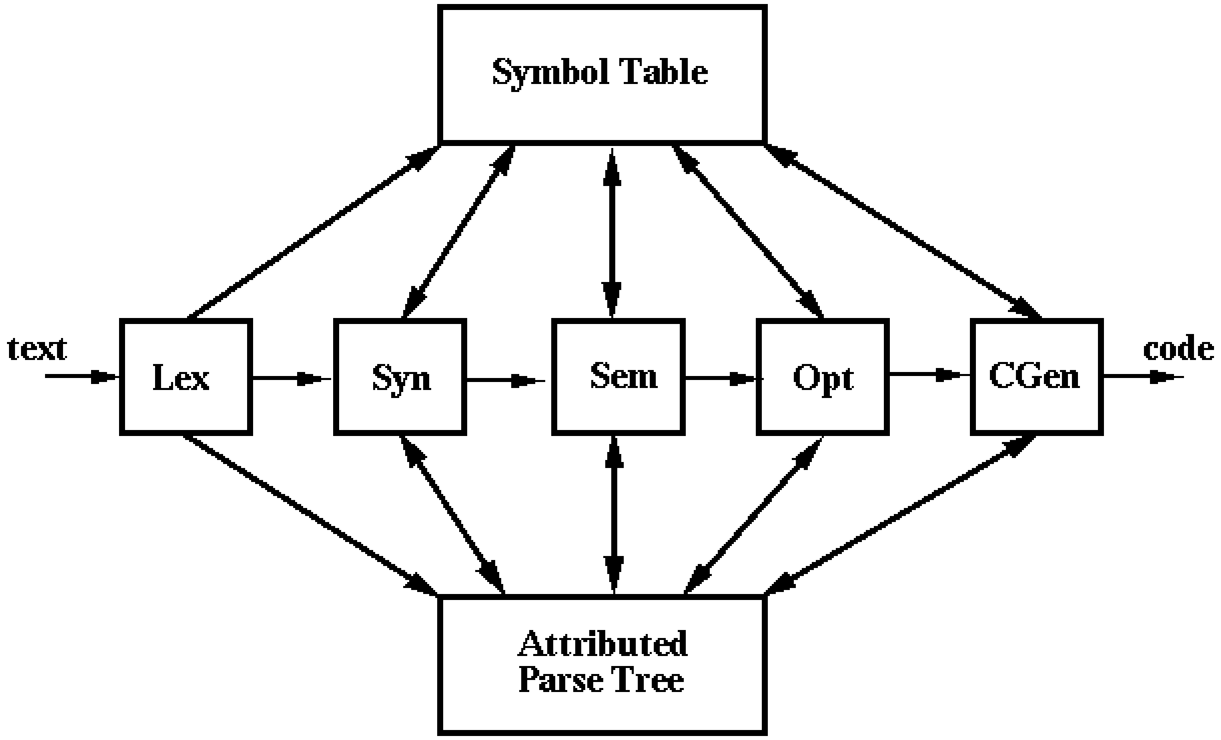
Early compiler architectures (cont'd)

Most compilers create a separate symbol table during lexical analysis and used or updated it during subsequent passes.

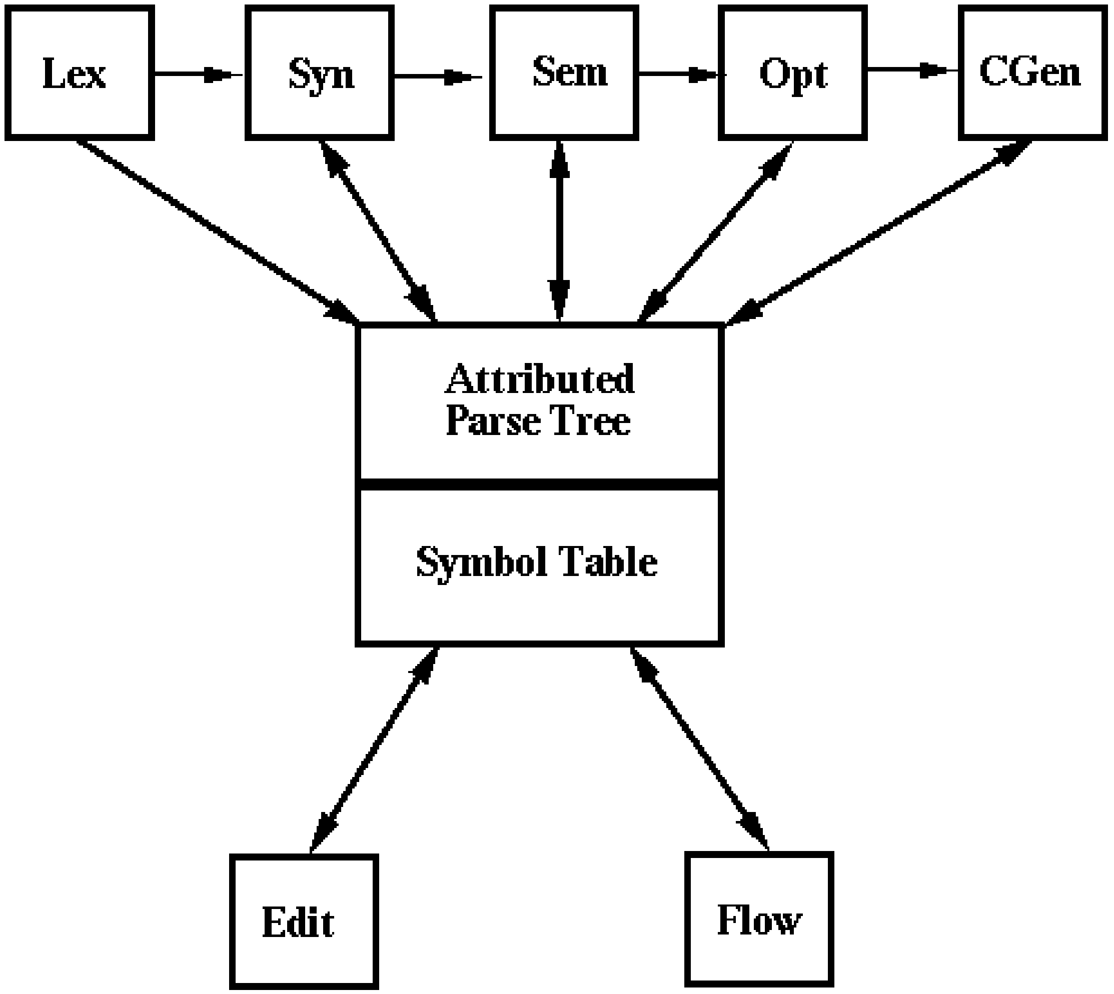


Modern compiler architectures (cont'd)

Later, in the mid 1980s, increasing attention turned to the intermediate representation of the program during compilation.



Hybrid compiler architectures



Object-Oriented

- Suitable for applications in which a central issue is identifying and protecting related bodies of information (data).
- Data representations and their associated operations are encapsulated in an **abstract data type**.

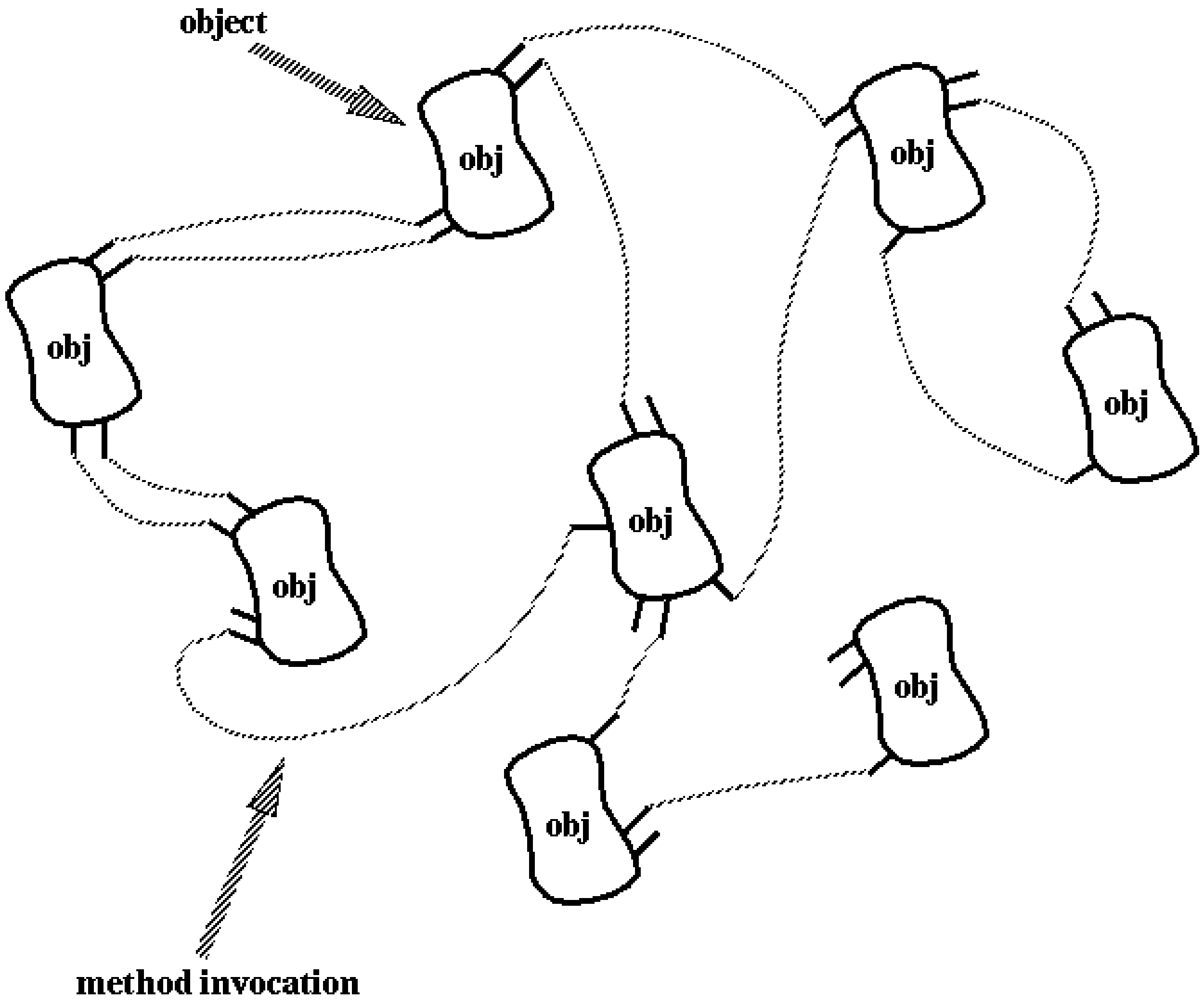
Components: are **objects**.

Connectors: are function and procedure invocations (**methods**).

Object-Oriented (cont'd)

- Subtle shift in programming style.
- Not all parameters are equally significant.
- Procedure invocation is determined by object, rather than by case statements.
- Restrictions on how information within objects can be used (encapsulation).
- Usefulness determines lifetime of data.
- Reuse achieved through inheritance.

Object-Oriented (cont'd)



Object-Oriented invariants

- Objects are responsible for preserving the **integrity** (*e.g.*, some invariant) of the data representation.
- The data representation is **hidden** from other objects.

Object-Oriented specializations

- Distributed objects.
- Objects with multiple interfaces.
- COM.

Object-Oriented advantages

- Because an object hides its data representation from its clients, it is possible to **change the implementation without affecting those clients**.
- Can **design** systems as collections of autonomous interacting agents.
- Intuitive mapping from real world objects to implementation in software.

Object-Oriented advantages (cont'd)

- Good ability to manage construction and destruction of objects, centralizing these operations in one place.
- Relatively easy to distribute objects.
- Shifts focus towards object interfaces and away from arbitrary procedure interfaces.
- Moves away from the "any procedure can call any procedure" paradigm.

Object-Oriented disadvantages

- In order for one object to interact with another object (via a method invocation) the first object must know the **identity** of the second object.
    - Contrast with Pipe and Filter Style.
    - When the identity of an object changes it is necessary to modify all objects that invoke it.
- Partially resolved by using conventions such as COM.

Object-Oriented disadvantages (cont’d)

- The distinction between an object changing its content, and becoming a new object are too harsh.
- Objects cause **side effect problems**:
    - E.g., *A* and *B* both use object *C*, then *B*'s affect on *C* look like unexpected side effects to *A*.
    - Essentially the problem here is that objects have persistent state.
- Managing object destruction is hard.

Microsoft’s COM architecture

- “Component Object Model”.
- A collection of rules and restrictions on how objects may be both used and managed.
- An interface is a “named” collection of methods having a predefined purpose and call/return protocol.
- An object has one or more interfaces. It can be manipulated only through it’s interfaces.

COM (cont'd)

- Each interface name is identified by a globally unique id (GUID).
- Every COM object has the `Iunknown` interface.
    - `QueryInterface(REFIID riid, void **vPP)`
    - `AddRef(void)`
    - `Release(void)`
- Other interfaces are accessed via a pointer to the interface returned by `QueryInterface` if the object supports the specified interface.

COM advantages

- Methods supported by an object can be determined at runtime.
- Reduces risk of performing illegal operations on an object as a result of recasting it incorrectly.
- Reference counting simplifies management of object’s lifetime.
- Objects may incrementally be assigned new interfaces cleanly.

COM advantages (cont’d)

- Makes distribution of objects easier.
- Interfaces can be implemented in C++ by merely arranging for an object class to multiply inherit from each of its supported interfaces.
- Avoids the problem of trying to change an existing interface definition. You don’t.

COM advantages (cont'd)

- Objects can be registered in the NT registry.
- The software needed to support objects can be dynamically loaded from DLL's upon demand.
- The software needed to support an object need only be loaded once, not once per program.
- Common interfaces are shared by all objects that support them.

COM disadvantages

- Tedium and overhead associated with obtaining and freeing interface pointers.
- All methods are called indirectly via the interface pointer, which is inefficient.
- Defined interfaces may not be changed.
- Need to dynamically create all COM objects. Can’t delete static objects.

COM disadvantages (cont'd)

- Some difficulty aggregating COM objects into larger objects because of the shared `IUnknown` interface.
- Risk of loading a "Trojan horse".

Implicit Invocation

- Suitable for applications that involve loosely-coupled collection of components, each of which carries out some operation and may in the process enable other operations.
- A generalization of event driven code in which relevant state is included with the event, and multiple processes can "see" events.
- Particularly useful for applications that must be reconfigured on the fly.
    - Changing a service provider.
    - Enabling or disabling capabilities.

Implicit Invocation (cont’d)

- Suitable for applications that involve loosely-coupled collection of components, each of which carries out some operation and may in the process enable other operations.
- A generalization of event driven code in which relevant state is included with the event, and multiple processes can “see” events.

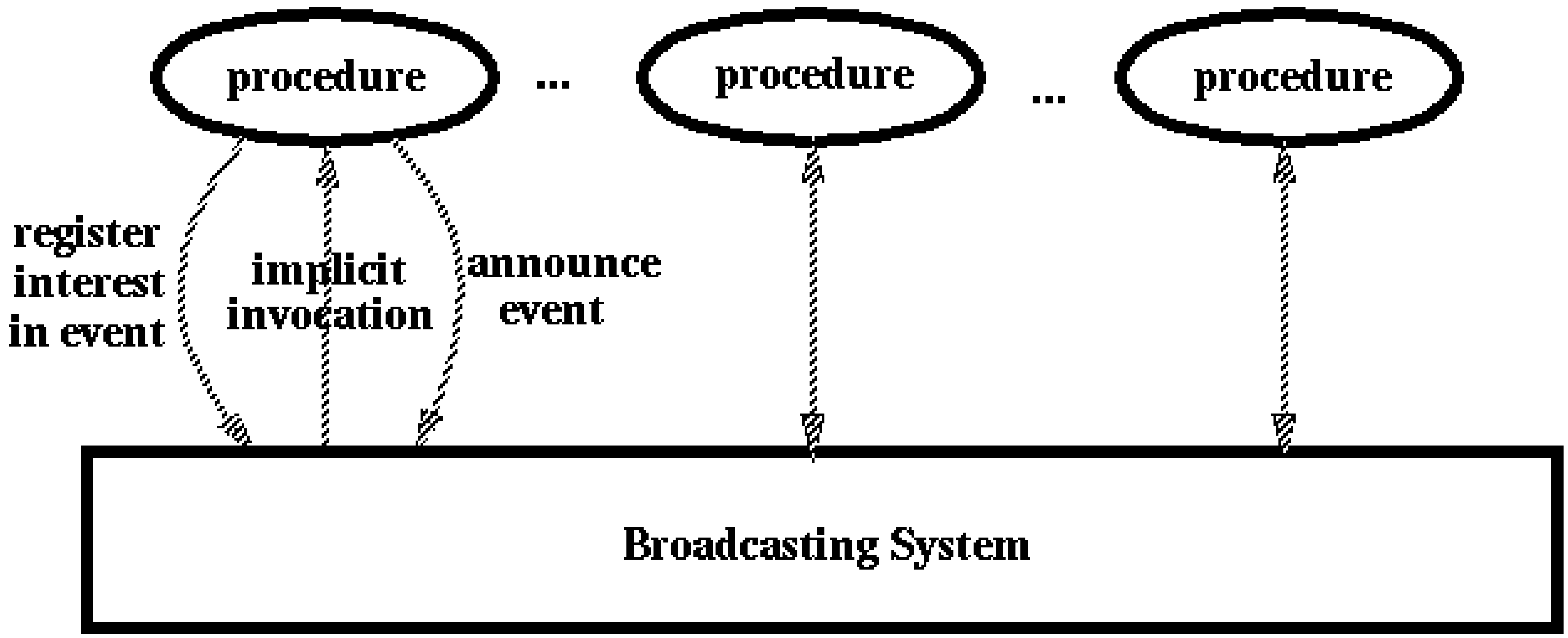
Implicit Invocation (cont'd)

Instead of invoking a procedure directly …

- A **component** can announce (or broadcast) one or more events.
- Other **components** in the system can register an interest in an event by associating a procedure with the event.
- When an event is announced, the broadcasting system (**connector**) itself invokes all of the procedures that have been registered for the event.

Implicit Invocation (cont'd)

An event announcement "implicitly" causes the invocation of procedures in other modules.



Implicit Invocation invariants

- Announcers of events do not know which components will be affected by those events.
- Components cannot make assumptions about the order of processing.
- Components cannot make assumptions about what processing will occur as a result of their events (perhaps no component will respond).

Implicit Invocation specializations

- Often connectors in an implicit invocation system also include the **traditional procedure call** in addition to the bindings between event announcements and procedure calls.

Implicit Invocation examples

Used in **programming environments** to integrate tools:

- Debugger stops at a breakpoint and makes that announcement.
- Editor responds to the announcement by scrolling to the appropriate source line of the program and highlighting that line.

Implicit Invocation examples (cont'd)

- Used to enforce integrity constraints in **database management systems** (called triggers).
- Used in **user interfaces** to separate the presentation of data (views) from the applications that manage that data.
- Used in **user interfaces** to allow correct routine of function keys etc. to logic.
- Used in **forms** to allow generic logic.

Implicit Invocation advantages

- Provides strong support for **reuse** since any component can be introduced into a system simply by registering it for the events of that system.
- **Eases system evolution** since components may be replaced by other components without affecting the interfaces of other components in the system.

Implicit Invocation advantages (cont'd)

- **Eases system development** since one has to only map the events which occur to the software that manages them, and these events are often predefined.
- **Case tools** hide the complexities of managing the flow of events.
- **Asynchronous** interface improves performance, response times etc.

Implicit Invocation disadvantages

- When a component announces an event:
    - It has no idea what other components will respond to it.
    - It cannot rely on the order in which the responses are invoked.
    - It cannot know when responses are finished.
- Feedback involves generating additional events that are routed to callback routines.

Implicit Invocation disadvantages (cont')

- There is no single defined flow of logic within such systems.
- It can be hard to consider all possible events that may occur, and their interactions.
- Such systems can be very hard to both maintain and debug.
- There is the risk that you end up communicating with "Trojan horses".

Layered

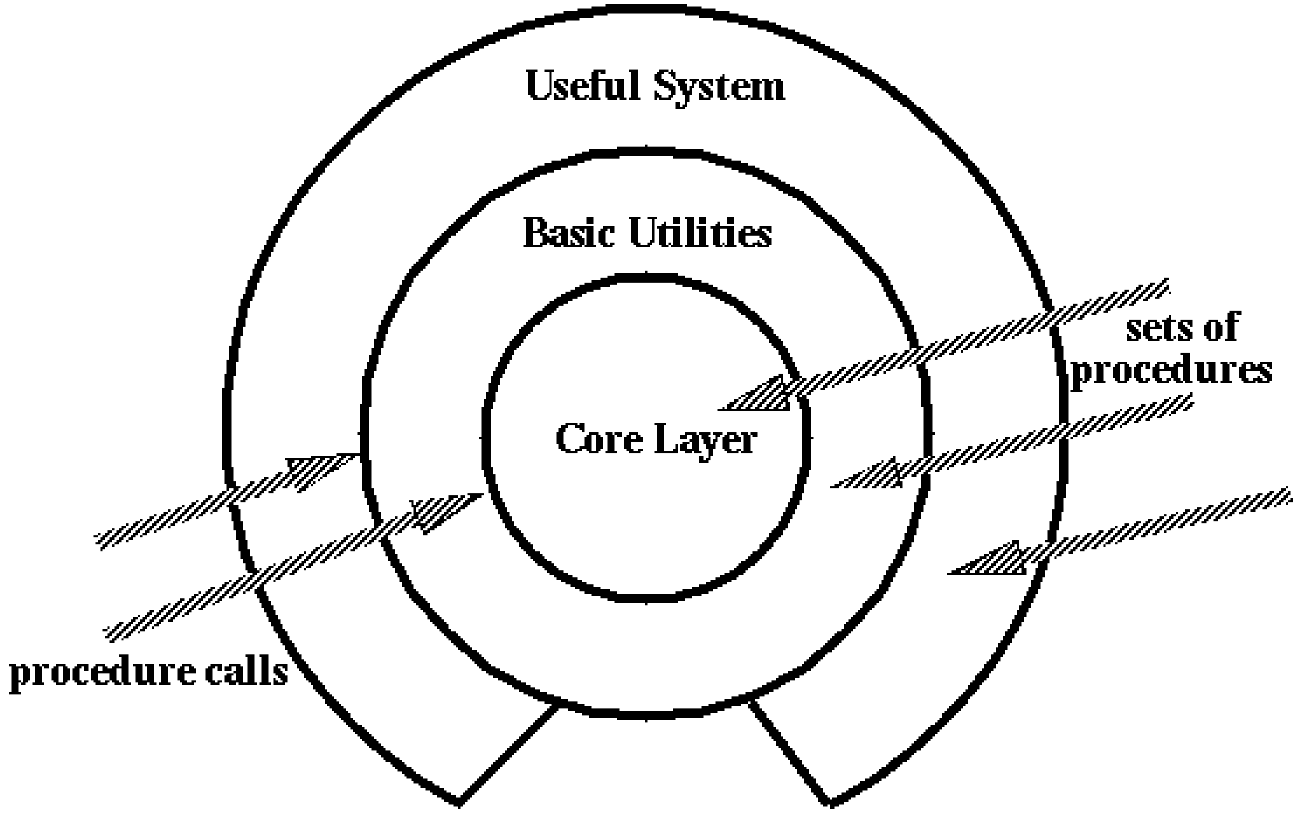
- Suitable for applications that involve distinct classes of services that can be **organized hierarchically**.
- Each layer provides service to the layer above it and serves as a client to the layer below it.
- Only carefully selected procedures from the inner layers are made available (exported) to their adjacent outer layer.

Layered (cont’d)

Components: are typically collections of procedures.

Connectors: are typically procedure calls under restricted visibility.

Layered (cont'd)



Layered specializations

- Often exceptions are be made to **permit non-adjacent layers to communicate directly**.
- This is usually done for efficiency reasons.

Layered examples

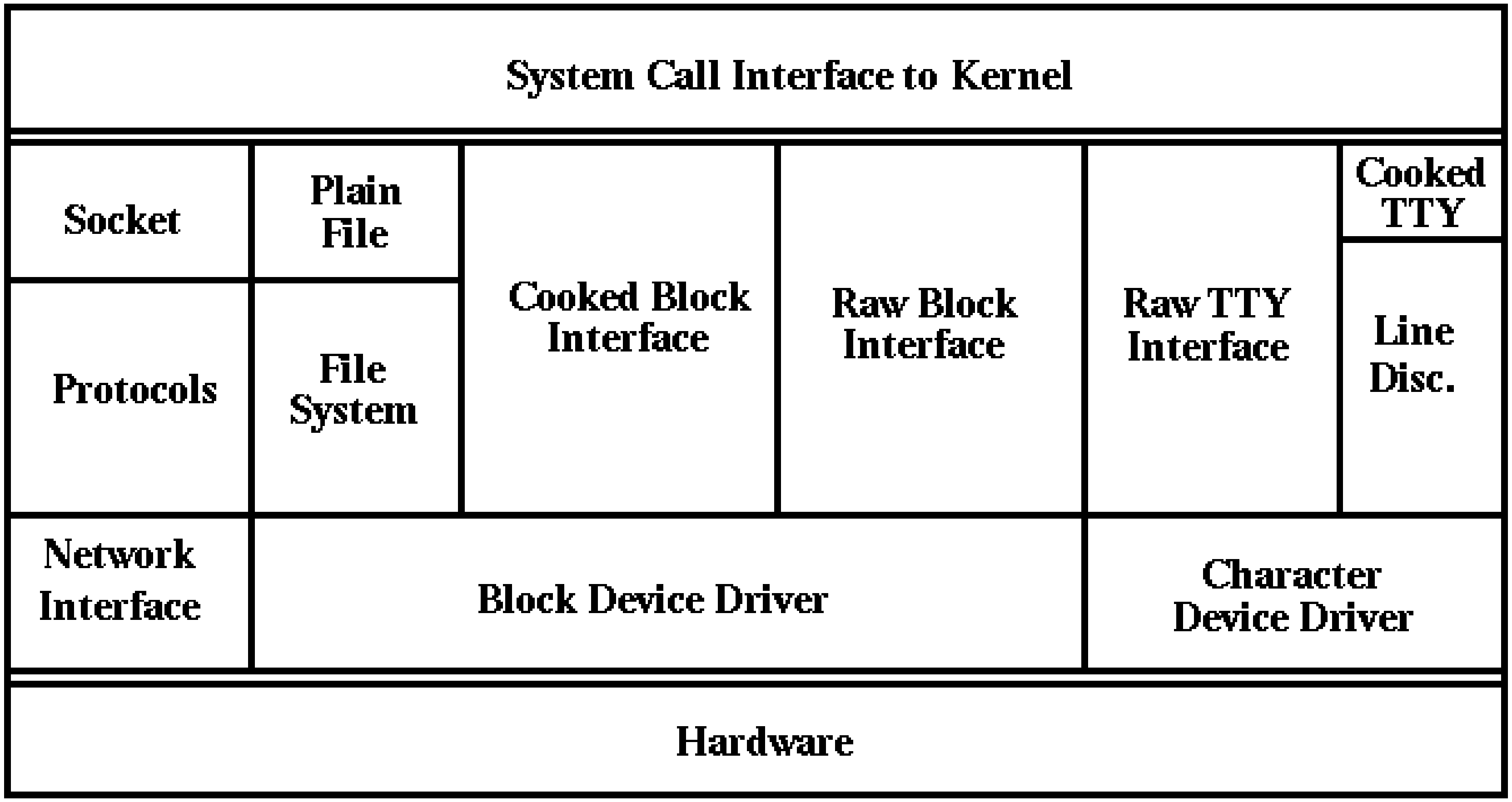
Layered Communication Protocols:

- Each layer provides a substrate for communication at some level of abstraction.
- Lower levels define lower levels of interaction, the lowest level being hardware connections (physical layer).

Operating Systems

- Unix

The Unix layered architecture



Layered advantages

- **Design:** based on increasing levels of abstraction.
- **Enhancement:** since changes to the function of one layer affects at most two other layers.
- **Reuse:** since different implementations (with identical interfaces) of the same layer can be used interchangeably.

Layered disadvantages

- Not all systems are easily structured in a layered fashion.
- Performance requirements may force the coupling of high-level functions to their lower-level implementations.
- Adding layers increase the risk of error.

Eg. `getchar()` doesn't work correctly on Linux if the code is interrupted, but `read()` does.

Layered by language

- Different languages address very different needs.
- It is sometimes useful to layer software according to language layers.
- The languages employed define the interfaces between layers.

Layered language example

The web.

- HTML
- ASP
- VBScript
- OLE
- OLE/DB
- C++

Repository

- Suitable for applications in which the central issue is establishing, augmenting, and maintaining a complex central body of information.
- The information must typically be manipulated in a variety of ways. Often long-term *persistence* is required.

Repository (cont'd)

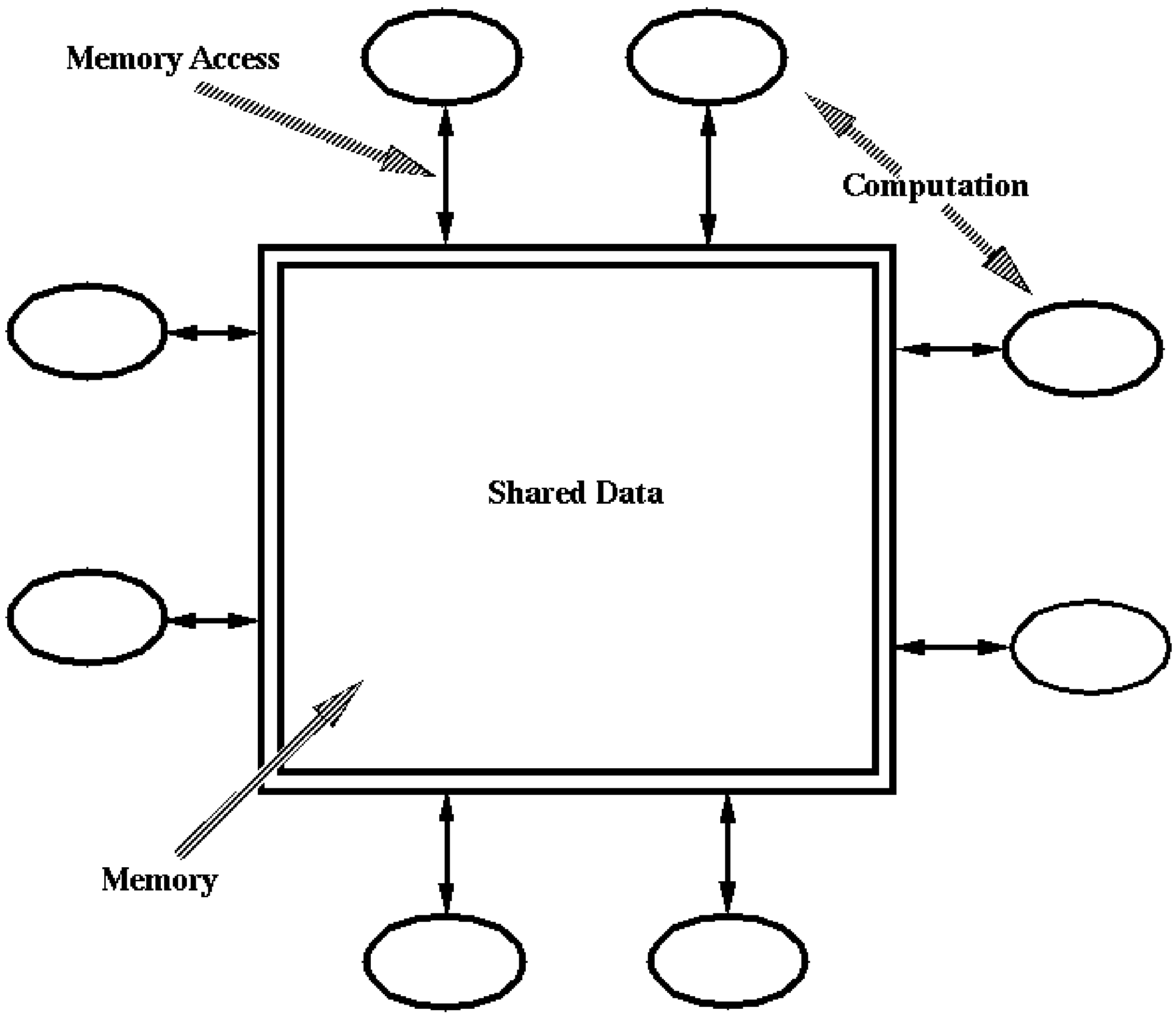
Components:

- A central data structure representing the correct state of the system.
- A collection of independent components that operate on the central data structure.

Connectors:

- Typically procedure calls or direct memory accesses.

Repository (cont'd)



Repository specializations

- Changes to the data structure trigger computations.
- Data structure in memory (persistent option).
- Data structure on disk.
- Concurrent computations and data accesses.

Repository examples

- Information systems.
- Programming environments.
- Graphical editors.
- AI knowledge bases.
- Reverse engineering systems.
- SQL:1999 is computationally complete.

Repository advantages

- **Efficient** way to store large amounts of data.
- **Sharing** model is published as the repository schema.
- Centralized **management** for
    - backup,
    - security and
    - concurrency control.

Repository disadvantages

- Must agree on a data model a priori.
- Difficult to distribute data.
- Data evolution is expensive.

Interpreter

Suitable for applications in which the most appropriate language or machine for executing the solution is not directly available.

Interpreter (cont'd)

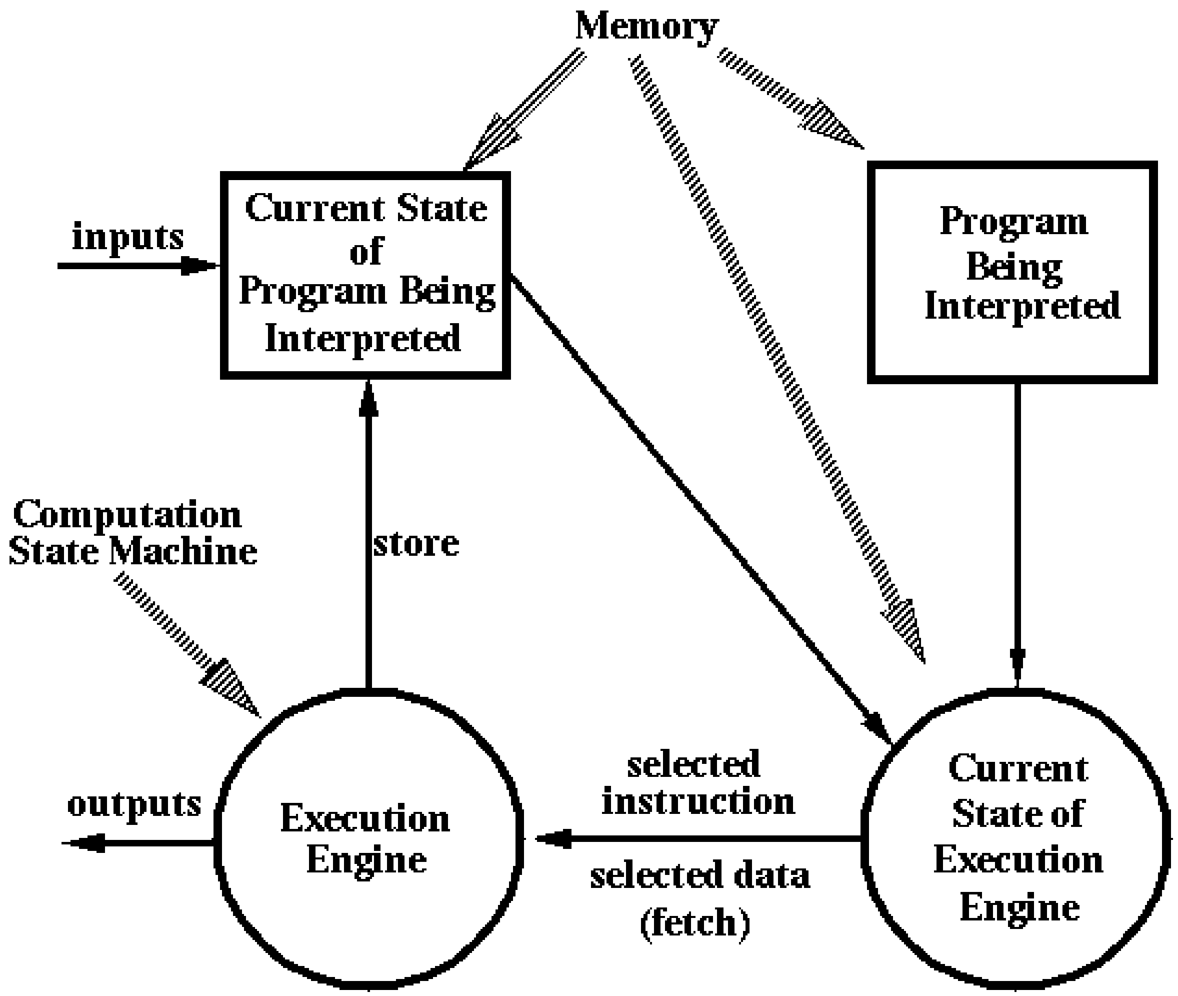
Components: include one state machine for the execution engine and three memories.

- Current state of the execution engine.
- Program being interpreted.
- Current state of the program being interpreted.

Connectors:

- Procedure calls.
- Direct memory accesses.

Interpreter (cont’d)


Memory
Current State of Program Being Interpreted
inputs
Program Being Interpreted
Computation State Machine
store
outputs
Execution Engine
selected instruction
selected data (fetch)
Current State of Execution Engine

Interpreter examples

Programming Language Compilers:

- Java
- Smalltalk

Rule Based Systems:

- Prolog
- Coral

Scripting Languages:

- Awk
- Perl

Interpreter examples (cont’d)

Micro coded machine

- Implement machine code in software.

Cash register / calculator

- Emulate a clever chip using a cheap one.

Database plan

- The database engine interprets the plan.

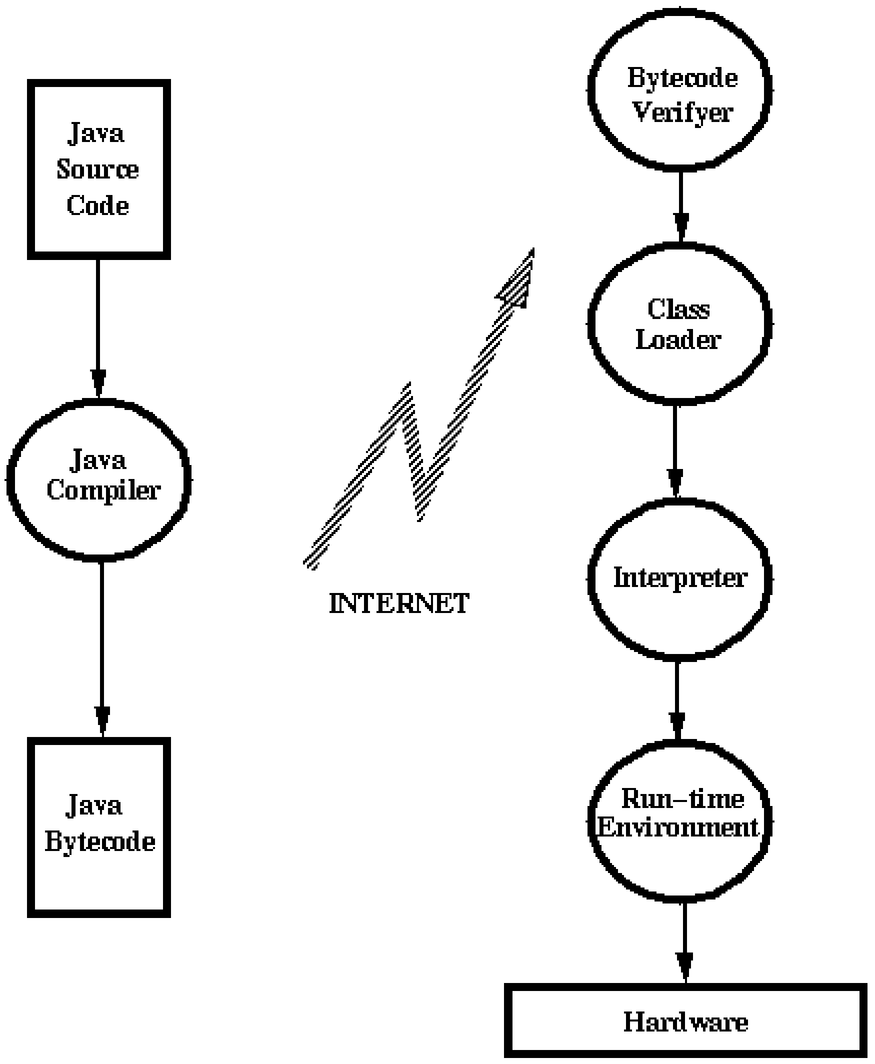
Presentation package

- Display a graph, by operating on the graph.

Interpreter advantages

- Simulation of non-implemented hardware; keeps cost of hardware affordable.
- Facilitates portability of application or languages across a variety of platforms.
- Behaviour of system defined by a custom language or data structure; makes software easier to develop and understand.
- Separates the *how do we do this*, from the *how do we say what it is we want to do*.

Java architecture



Interpreter disadvantages

- Extra level of indirection **slows** down execution.
- Java has an option to compile code:
    - JIT (Just In Time) compiler.

Process-Control

Suitable for applications whose purpose is to maintain specified properties of the outputs of the process at (sufficiently near) given reference values.

<u>Components:</u>

- **Process definition** includes mechanisms for manipulating some process variables.
- **Control algorithm** for deciding how to manipulate process variables.

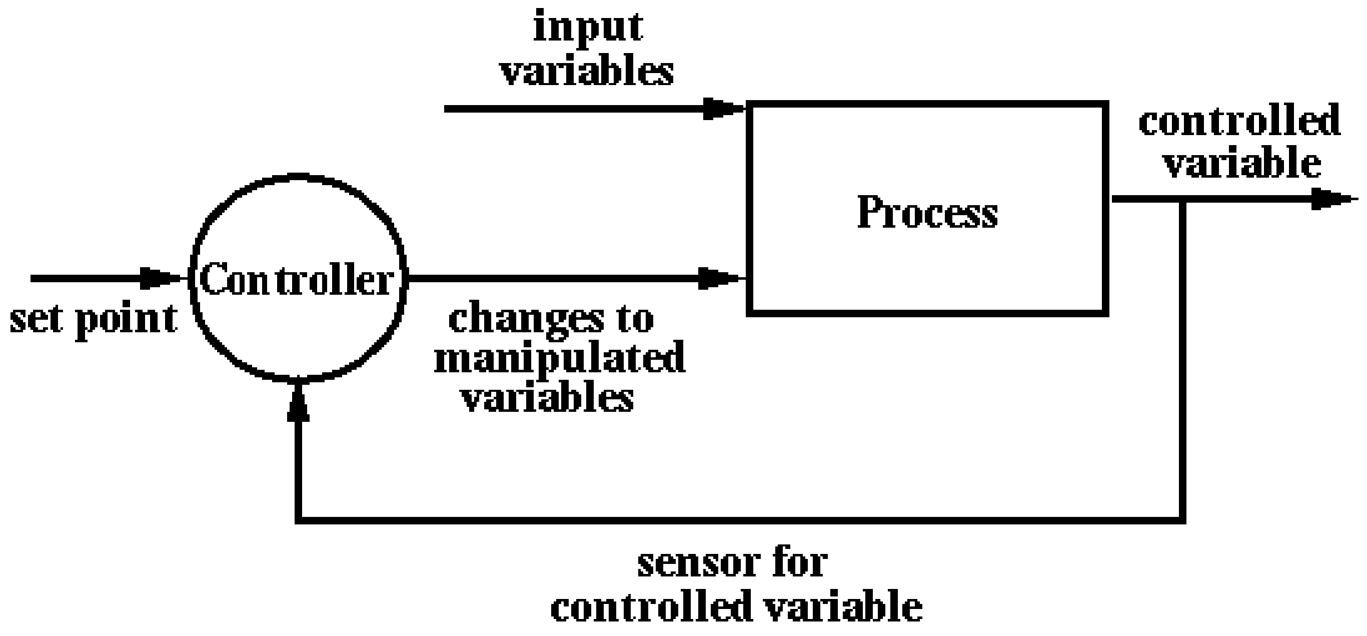
Process-Control (cont’d)

Connectors: are the data flow relations for:

- **Process Variables:**
  - *Controlled variable* whose value the system is intended to control.
  - *Input variable* that measures an input to the process.
  - *Manipulated variable* whose value can be changed by the controller.
- **Set Point** is the desired value for a controlled variable.
- **Sensors** to obtain values of process variables pertinent to control.

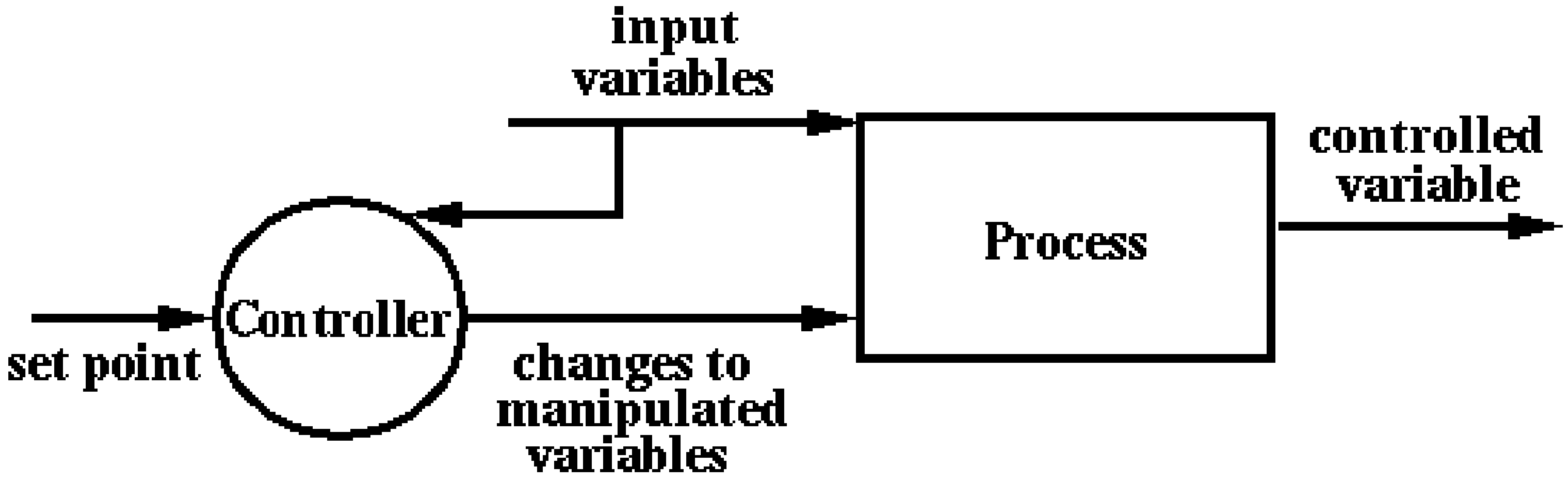
Feed-Back Control System

The controlled variable is measured and the result is used to manipulate one or more of the process variables.


input variables
Process
controlled variable
set point
Controller
changes to manipulated variables
sensor for controlled variable

Open-Loop Control System

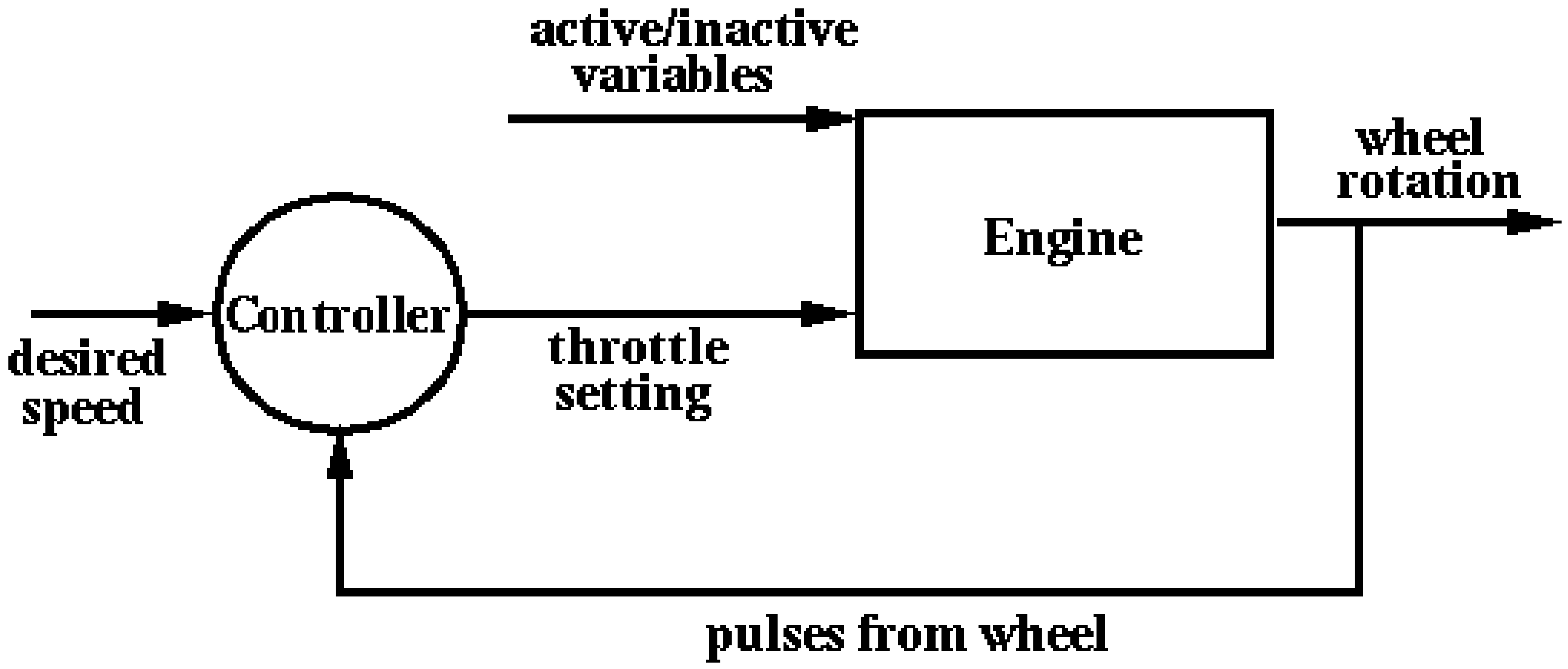
Information about process variables is not used to adjust the system.



Process Control examples

Real-time system software to control:

- Automobile anti-lock brakes.
- Nuclear power plants
- Automobile cruise-control



Process Control examples (cont’d)

- Hardware circuits that implement clocks, count, add etc.

Additional styles

- Client-Server
- Event driven
- Table driven
- Co-routines
- Bootstrapped
- Iterative enhancement

Client-Server

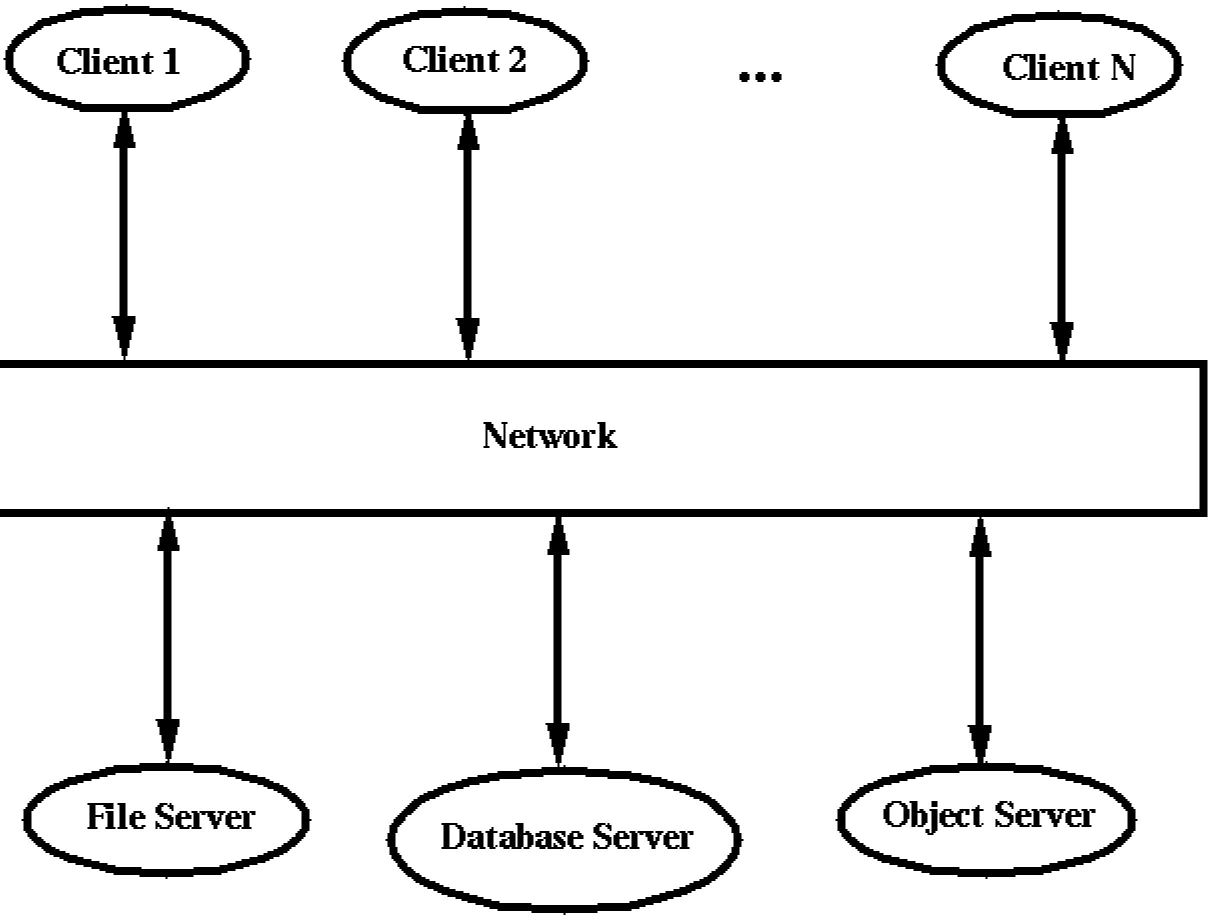
Suitable for applications that involve distributed data and processing across a range of components.

<u>**Components:**</u>

- **Servers:** Stand-alone components that provide specific services such as printing, data management, etc.
- **Clients:** Components that call on the services provided by servers.

<u>**Connector:**</u> The network, which allows clients to access remote servers.

Client-Server



Client-Server examples

File Servers:

- Primitive form of data service.
- Useful for sharing files across a network.
- The client passes request for files over the network to the file server.

Client-Server examples (cont'd)

Database Servers:

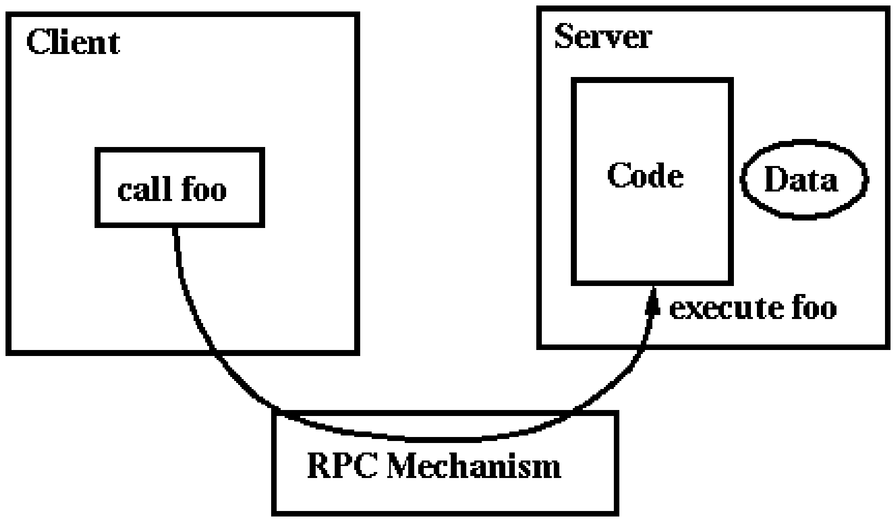
- More efficient use of distributing power than file servers.
- Client passes SQL requests as messages to the DB server; results are returned over the network to the client.
- Query processing done by the server.
- No need for large data transfers.
- Transaction DB servers also available.

Client-Server examples (cont’d)

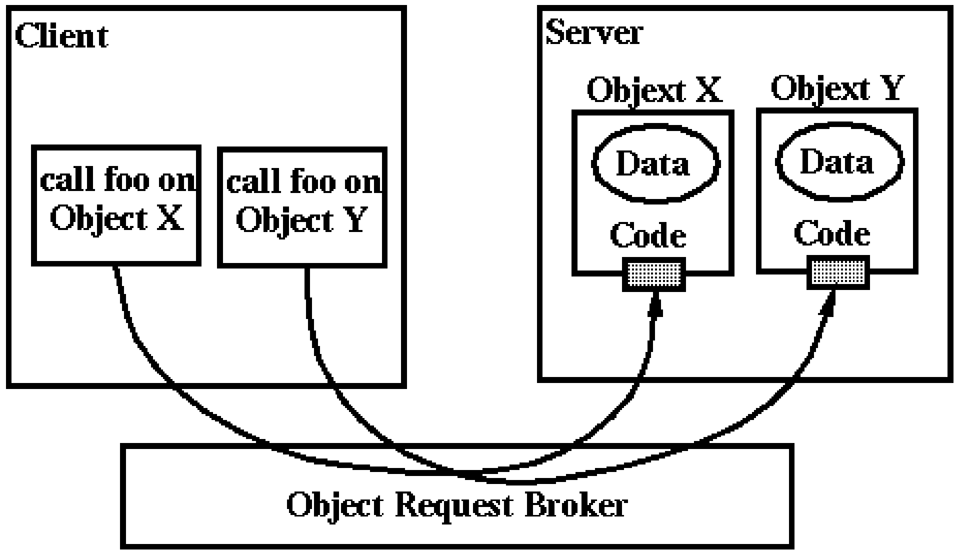
Object Servers:

- Objects work together across machine and network boundaries.
- ORBs allow objects to communicate with each other across the network.
- IDLs define interfaces of objects that communicate via the ORB.
- ORBs are an evolution of the RPC.

RPCs vs. ORBs



1) Remote Procedure Call (RPC)



2) Object Request Broker (ORB)

Client-Server advantages

- Distribution of data is straightforward.
- Transparency of location.
- Mix and match heterogeneous platforms.
- Easy to add new servers or upgrade existing servers.
- Functional client server interface.
- Simplifies distant levels of recursion.
- One server can support multiple clients.

Client-Server disadvantages

- No central register of names and services; it may be hard to determine what services are available.
- Hard to asynchronously communicate with server. E.g.: *cancel database query*.
- Server can’t initiate communication with clients. The best it can do is provide complex responses when its services are requested.

Client-Server disadvantages (cont’d)

- Overhead of packing and unpacking data encoded in messages, particularly when client and server are on the same machine. (In good client-server implementations this problem is avoided.)
- Potential restrictions on the data types and structures that can be used. E.g.: `int64`, *unicode*, etc.

Event driven

- The logic is essentially reversed so that the detail is performed at the highest level.
- The decision as to what detail must be performed next is kept in a static or dynamic table.
    - Communication is via global or object state.
- Useful when we are required to “return” from code, but don’t wish to leave it.

Event driven examples

Barber shop simulation.

- After creating an initial future event, events themselves introduce additional future events.

Servers that “chit chat” with clients.

Table driven

- The logic is essentially governed by tables or data structures which are *precomputed* and then compiled into the code.
- Useful when we wish to reduce the run time complexity of the code by precomputing its appropriate behaviour in data inserted into the code at compile time.
- Improves performance of system.

Table driven examples

Yacc

- Tables are generated which determine how parsing is to be performed.

Cribbage game

- Value of cribbage hands precomputed.

Event/table driven pros&cons

- Can produce clean solutions to seemingly difficult problems
- Can be hard to grasp what is going on as different events occur in unclear orders.
- Some programmers have difficulty making the transition from conventional code to event driven code.

Co-routines

- The whole is bigger than the parts, but the parts cannot easily be decomposed into sequential operations.
- Separate parts must communicate with each other without loosing stack state.
- Parts run in separate threads and the overall operation is tightly coupled to produce the desired computation.

Co-routine examples

- Concurrent error detection and correction.
- Buffer management.
- Disk update by any must cause all to be notified, so that caches can be reloaded.

Bootstrapped

A quick but inefficient way of creating a tool can lead to a tool which allows creation of the same tool in better ways.

Bootstrap examples

- Parsers accept as input a document whose syntax conforms to the parser's meta language. Therefore parsers must themselves contain parsers.
- Java engine written in C++, and then in interpreted Java, and then in compiled Java.

Bootstrap pros/cons

- Avoids need to design good solution when a bad solution leads directly to a better one.
- Can’t recreate the tool if you loose the tool. Need to be very careful when changing bootstrapped code not to destroy ability to produce tool from source code.

Iterative enhancement

- Want appearance of intelligent behaviour.
- Impossible to quantify what intelligence is.
- Start by writing a very dumb program.
- Keep adding logic which makes it less dumb.
- Terminate when no longer able to improve behaviour of resulting logic.

Iterative enhancement pro/cons

- Allows concurrent design and development.
- Can lead to surprising intelligence.
- Displays the same characteristics as human intelligence.. Rather unpredictable and not always right.
- Very hard to predict apriori how successful exercise will be.

Iterative enhancement example

Bridge program …

- Deal hand.
- Enforce basic rules of play.
- Add sensible rules for how to play well.
- Consider making finesses, etc.
- Logic identifies the least worse card to play based on huge number of empirical rules drawn from observation of codes prior behaviour.
- Release code when changes do not improve play.

Orthogonal issues

- Detect and eliminate memory leakage's
- Code should be re-entrant
    - Don’t condition logic based on static data
- Code should be thread safe
    - Avoid global state
    - Protect object state against concurrent update
- Code interrupt safe
    - Anticipate unexpected throws, interrupts etc.
    - E.g.: “out of memory” or `Cntl-C`