# Software Architecture Paradigms

*"We have found that understanding software architecture is the key to developing many important software solutions."*

B. Stroustrup, 1991.

## Objectives

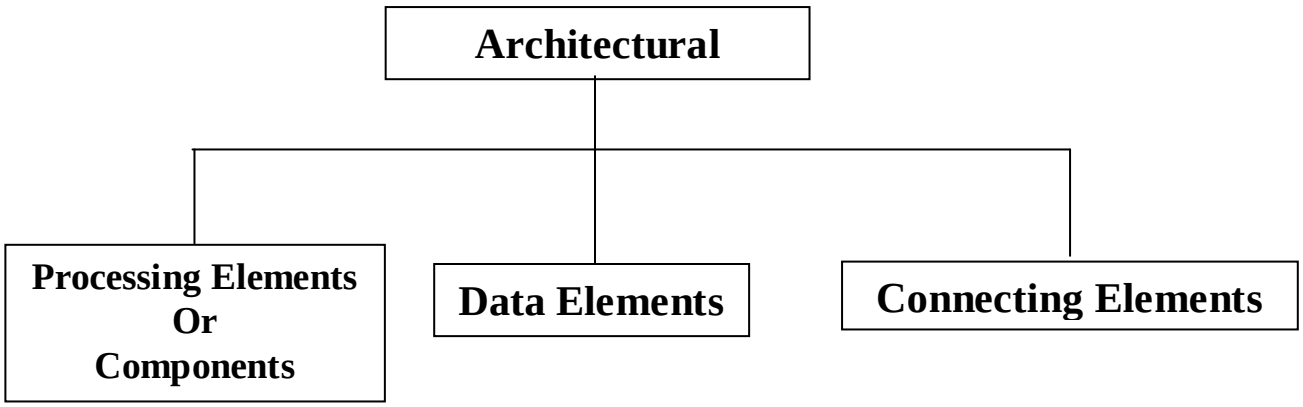
- An architecture design document is the key technical document used to determine whether the critical system requirements are met.
- Software design determines how requirements are realized as software structures.
- This is the immediate step after the requirements engineering phase in the software life-cycle.
- In software development, it is useful to organize architectures into families and associate families with typical applications: This will help reduce the overall development time
- Where are we in the software life cycle?

<table>
<tr><th>Requirements</th><th>Design</th><th>Implementation</th><th>….</th></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>Software Architectures</td><td></td><td></td></tr>
<tr><td></td><td></td><td>Components</td><td></td></tr>
<tr><td></td><td colspan="2">Software Component Architecture</td><td></td></tr>
<tr><td colspan="2">DSSA: Domain-Specific Software Architectures</td><td></td><td></td></tr>
<tr><td></td><td colspan="2">Frameworks</td><td></td></tr>
<tr><td></td><td>Design Patterns</td><td></td><td></td></tr>
<tr><td></td><td></td><td>Which Programming language?</td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>

# 1. Definitions

- A model for describing software architectures was introduced by Perry and Wolf in 1992. A description of a software architecture consists of three basic elements:

  - A processing element or component is a software structure that transforms its inputs into required outputs.
  - A data element consists of information needed for processing or information to be processed by a processing element.
  - Connecting elements are the "glue" that holds different pieces of an architecture together.




# 2. Architecture Style [J. Peters and W. Pedrycz 2000]:

- An **architecture style** is a pattern of structural organization in an architecture. It appears as a toolbox containing tools (architecture) useful in constructing different kinds of software modules.

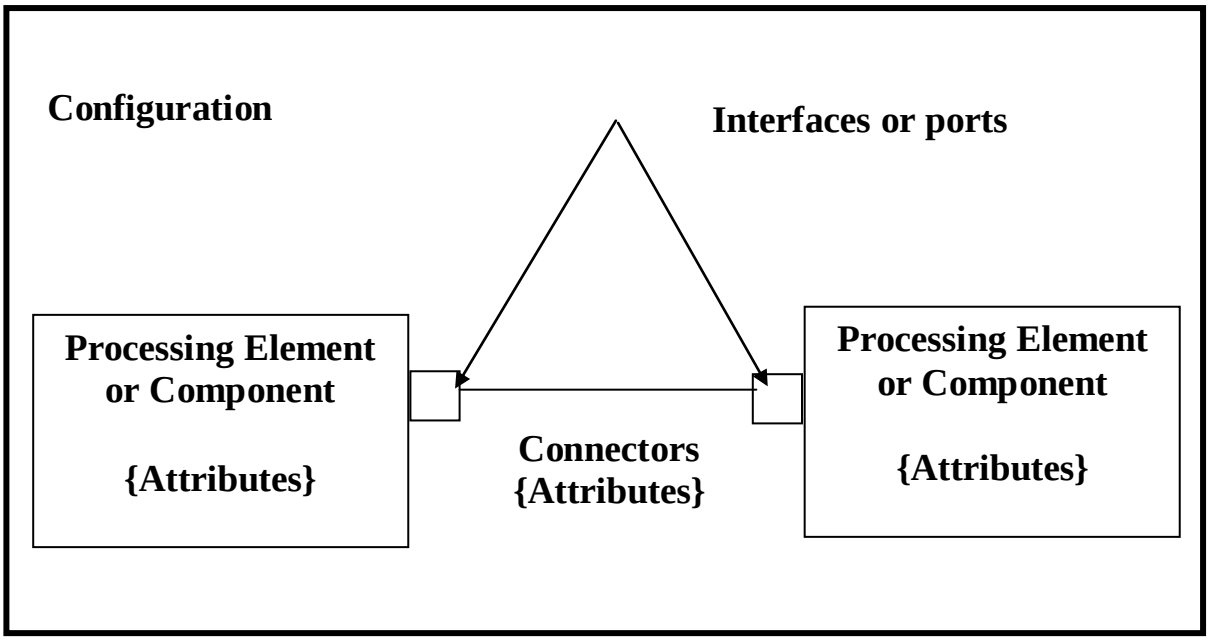
- Characteristics:
  - ✓ **Types of components**: Processing elements used to transform data (i.e., formatting routines in text formatting package).
  - ✓ **Types of connectors**: Control and data paths between components.

- ✓ **Constraints**: Restrictions on processing, data, and allowable ways to "wire" components together.

- Architectural styles provide four things:

  - ✓ **Vocabulary**: A set of design elements such as pipes, filters, client, severs, parsers, databases, etc.
  - ✓ **Design rules**: These are the set of constraints that dictates how the processing elements should be connected.
  - ✓ **Semantic interpretation:** An architectural style provides a well-defined meaning of the connected design elements.
  - ✓ **Analyses:** Many styles provide analyses that can be performed on systems build in that style, i.e., deadlock detection, scheduling, etc.

- Detailed Description:




  - ○ **Components** communicate through one or more ports or interfaces.
  - ○ **Connectors**: can be implemented using shared variables, remote procedure call, and message passing.
  - ○ **An interface**: user interface, variables, a method that can be called by another component, events, etc.
  - ○ **Attributes** define the behavior of the element:
    - ▪ **Components**: run-time constraints, protocols, etc.

        - **Connectors**: rate, capacity, latency, asynchronous vs. asynchronous, etc.
        - **Interfaces**: direction of communication, buffering capacity, etc.
    - **Configuration**:
        - Also called topology
        - Can be modeled as a graph where nodes are the components of the architecture and links are different connectors between components.
        - Help analyze performance of the architecture style:
            - Bottleneck
            - Longest path (System response)
            - Etc.

- Example of software architecture:

    ✓ **Architecture element**: Internet Uniform Resource Locator (URL):

        ♦ **Processing elements**: Access method: ftp, http, telnet
        ♦ **Data**: Machine name, Directory name, File name.

        ♦ **Connectors**: // precedes machine name and / precedes directory name or file name.

    ✓ **Architecture element**: Application package: MS Word

        ♦ **Processing elements or components**: Spelling, grammar, Thesaurus, Word count, etc.
        ♦ **Data element**: File name
        ♦ **Connectors**: Pull-down menu and item buttons

- **Categories:**

    ✓ **Dataflow Systems**:
    Pipelines (pipes & filters)

Batch processing,
Process control

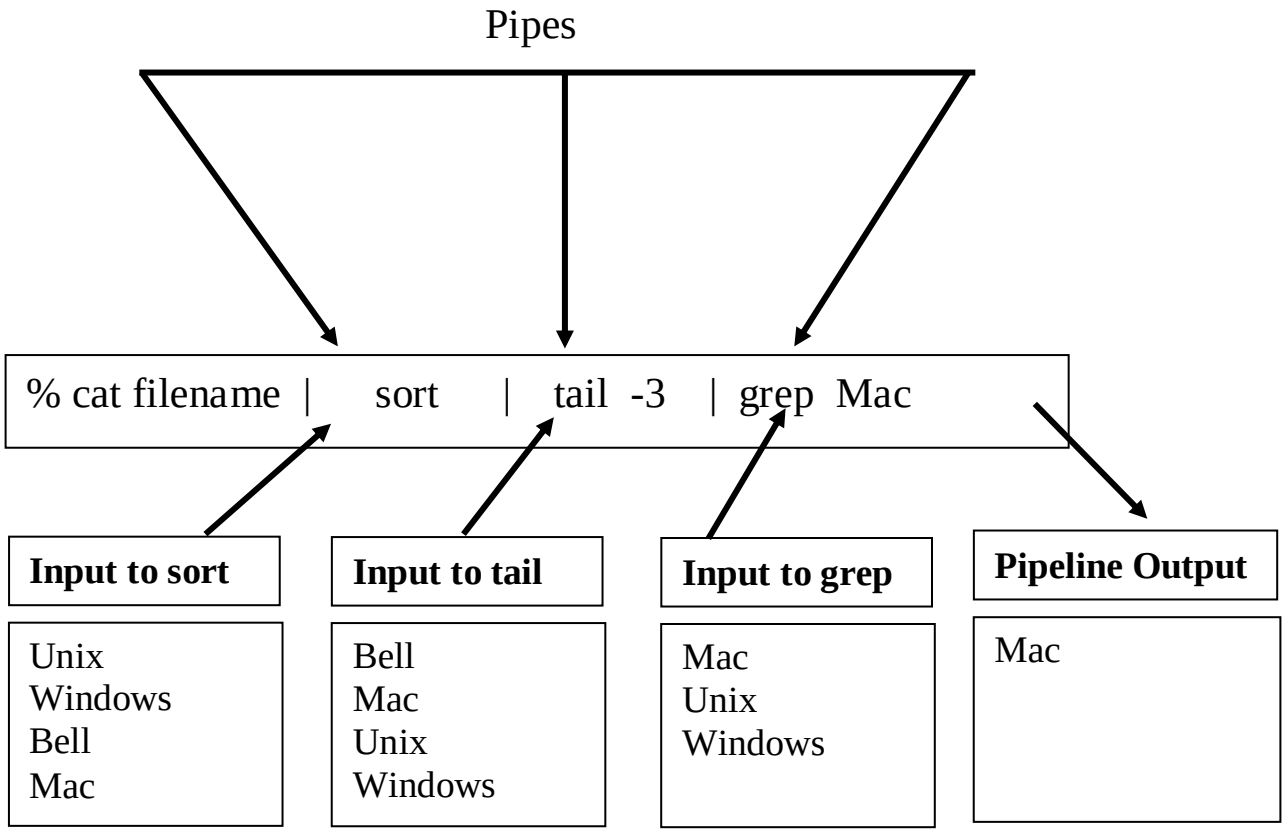
- ✓ **Call-&Return**:
  - Object-oriented (OO)
  - Layered
  - Procedure oriented
- ✓ **Independent-Process**:
  - Event-action system
  - Distributed system
  - Communicating processes
  - Parallel processes
  - Agent
- ✓ **Repository**:
  - Database
  - Hypertext
  - Archival
  - Blackboard
- ✓ **Virtual Machine**:
  - Intelligent system
  - Rule-based system
  - Interpreters

# 3. Data Flow Systems

## 3.1 Pipelines

- This architecture style is modeled after assembly lines in manufacturing plants.
- The pipe consists of stages. When one stage completes its processing its outputs become the inputs of the following stage.
- In software engineering, pipeline architecture consists of a set of connected processing elements called **filters**.
- A **filter** transforms its inputs and its results become inputs to the pipe connected to the next filter.
- The **data element** in a pipe is usually called a **stream**.
- The **connecting elements** between filters are called pipes

- Example: A Unix shell programming command



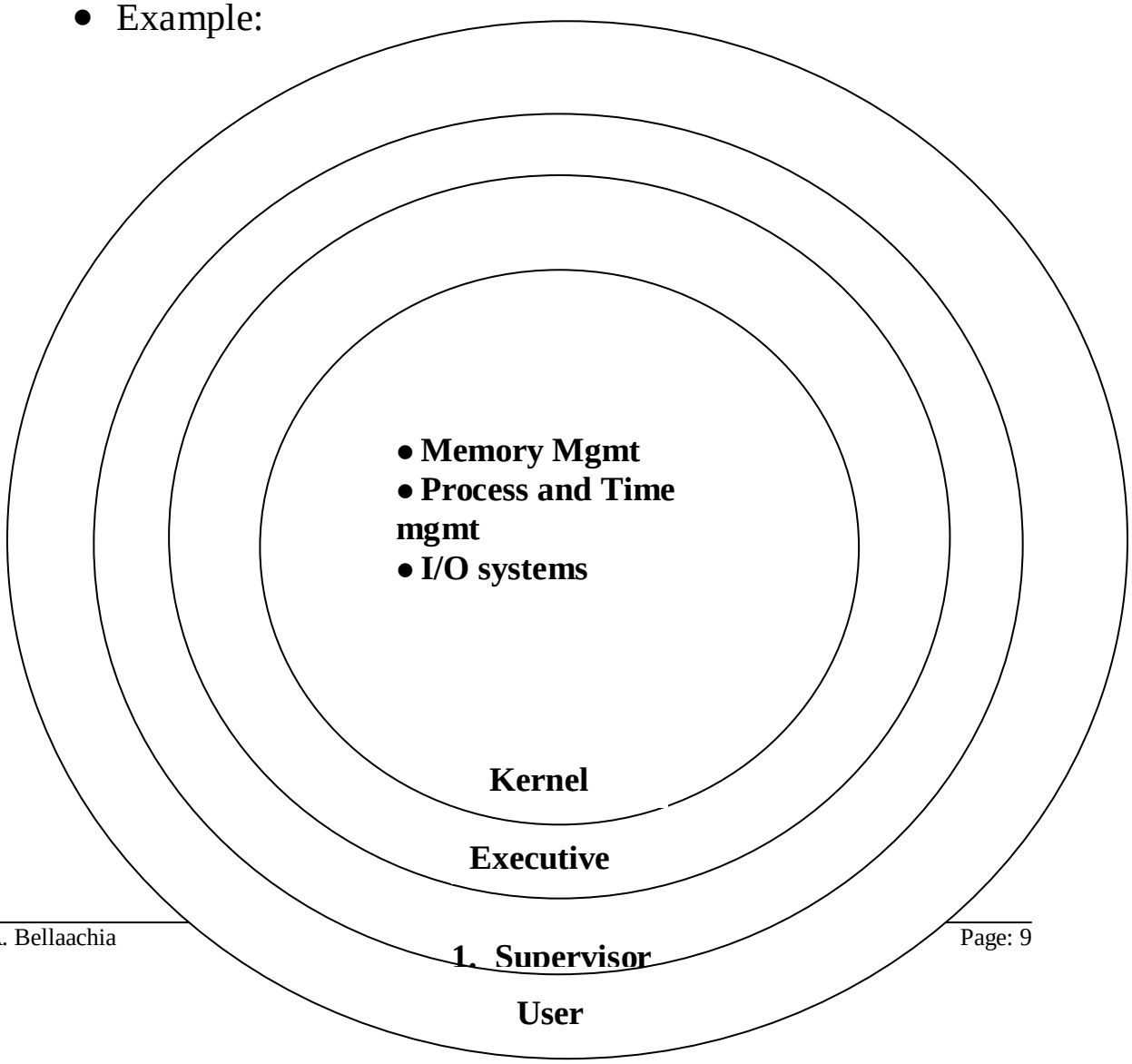
# 4. Call & return Systems

## 4.1 Layered Architecture:

- Layered architecture style works well whenever system requirements call independent tasks organized hierarchically.

- Architecture layers have been used in database, operating, computer-to-computer communications, etc.

- Definitions:
  - ✓ A layered architecture is designed as a hierarchy of client-server processes that minimizes interaction between layers.
  - ✓ Each layer acts as a client for the module above it and acts as a server for the module below it in an architecture layer.
- Example:



- ✓ **User** layer: provides utilities, application libraries, tools, and programming languages.

- ✓ **Supervisor** layer: provides command language interpreter (interface between users and inner layers of VMS).

- ✓ **Executive** layer: provides record managements.

- ✓ **Kernel** layer: Handles I/O (device drivers), schedules and control processes, manages memory.

- Advantages:

  - ✓ Layered architecture provides an incremental approach to designing a complex system. Maintainability: Layered can be easily maintained and even replaced.

  - ✓ Extendibility: new services (layers) can be easily added.

## 4.2 N-Tier Architecture : (www.n-tier.com)

### 4.2.1 Motivations

- **N-Tier** applications mean using whatever mix of **Computer Hardware** and/or **Software Layers** you need, in order to build a modular information system.
- What does N-tier mean?
  - N-Tier means "Any Number of Tiers"
  - ~ No Limits ~
  - Levels/Layers/Tiers
  - Clients & Customers
  - Objects & Components
  - Servers & Services

- Programs partitioned into **Tiers** allow each layer or component part to be developed, managed, deployed and enhanced **independently**.
- The N-Tier model of computing enables the overall performance and maintainability of Client/Server systems to be substantially improved.
- This layered environment also simplifies code distribution, since most of the business logic has been moved from the client to the server.
- Other advantages of Multi-Tier Client/Server architectures include:

    - Changes to the user interface or to the application logic are largely independent from one another, allowing the application to evolve easily to meet new requirements.

    - The client is insulated from database and network operations. The client can access data easily and quickly without having to know where data is or how many servers are on the system.

### 4.2.2 Architecture:

- In a multi-tier environment, the client implements the presentation logic (thin client). The business logic is implemented on an application server(s) and the data resides on database server(s).
- A Multi-tier architecture is thus defined by the following three component layers:

1. A front-end component, which is responsible for providing portable presentation logic;
2. A back-end component, which provides access to dedicated services, such as a database server.
3. A middle-tier component, which allows users to share and control business logic by isolating it from the actual application;

**1. Client/Server (C/S)**

- Client/Server is simply an architectural method of providing information to an end user; but that's where the simplicity ends.

- Client/Server is a general description of a networked system where a client program initiates contact with a separate server program (usually on a different machine) for a specific function or purpose. The client exists in the position of the requester for the service provided by the server.

**2. Different Types of Architectures**

- The term Client/Server has traditionally been associated with a desktop PC connected over a network to some sort of SQL-database server.

- One-Tier ~ Monolithic (C/S) Architectures

  The Information Technology (IT) industry, have been practicing a simple form of Client/Server computing since the initial inception of the mainframe. That configuration, a mainframe host and a directly connected, (unintelligent) terminal constitutes a one-tier C/S system.

- Two-Tier Client/Server Architectures
  - In two-tier client/server architecture, the client communicates directly with the database server. The application or business logic either resides on the client or on the database server in the form of stored procedures.
  - A two-tier (C/S) model first began to emerge with the applications developed for local area networks in the late eighties, and was primarily based upon simple file sharing techniques implemented by X-base style products (dBase, FoxPro, Clipper, Paradox, etc.).
- Three-Tier Client/Server Architectures
  - A newer generation of Client/Server implementations takes this segmented model a step further and adds a middle tier to achieve a '3-tier' architecture.

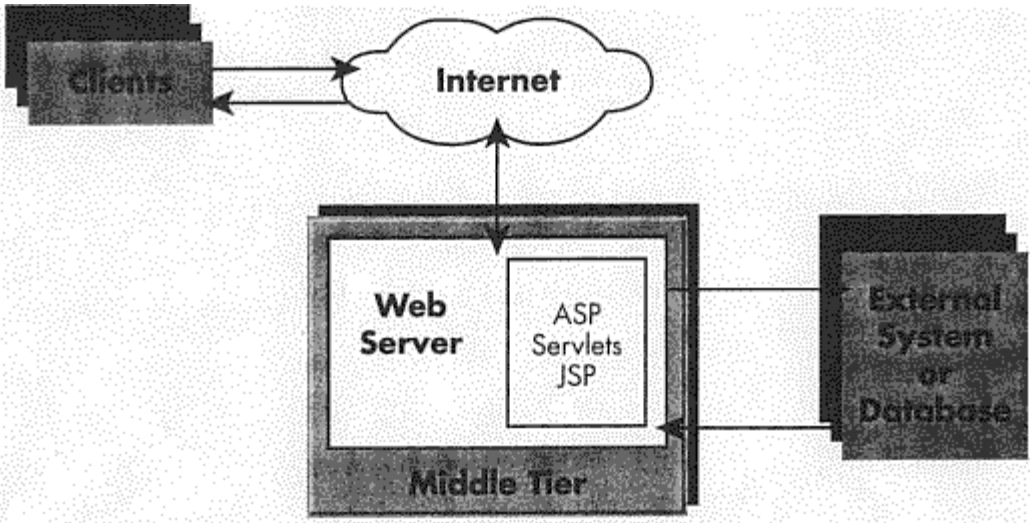



FIGURE 3.2. A web server application.

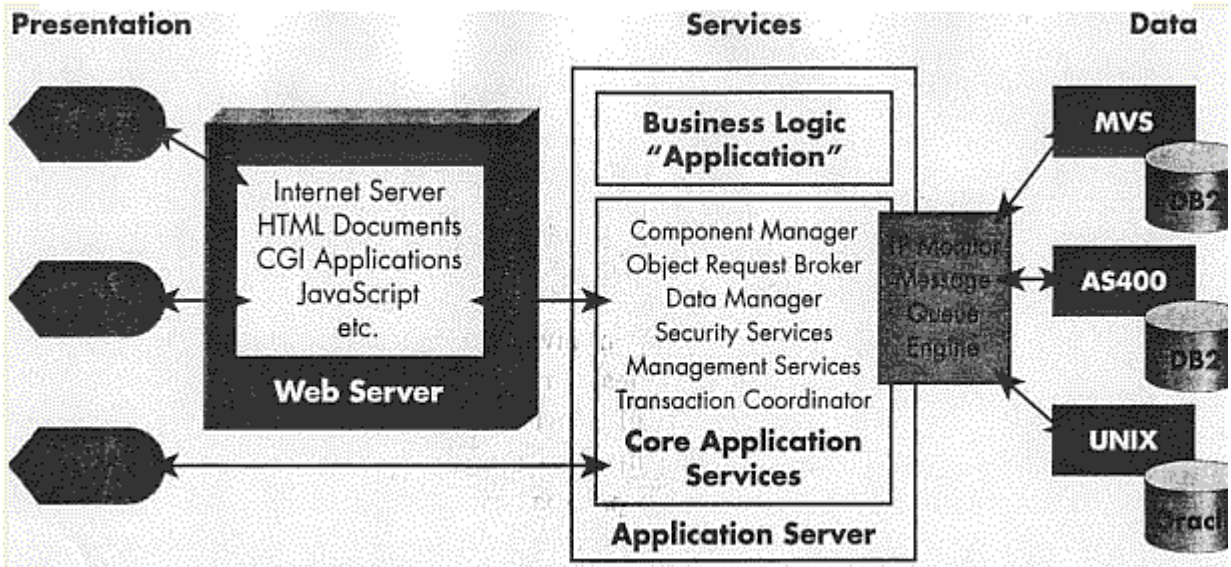



FIGURE 3.4. A logical view of an application server.

- Fat Clients
  - The two-tier model initially involved a non-mainframe host, (a network file server) and an intelligent "fat" client where most of the processing occurs.
  - This configuration did not scale well however, to facilitate large or even mid-size information systems (greater than 50 or so connected clients).

- Fat Servers
  - An alternative 'Thin' Client < --- > 'Fat' Server configuration, where the user invokes procedures stored at the database server.
  - The 'Fat' Server model, is more effective in gaining performance

- **Fat Middle**
  - A multi-tier architecture augments traditional client/server and two-tier computing by introducing (one or more) middle-tier components.
  - The client system interacts with the middle-tier via a standard protocol such as HTTP or RPC. The middle-tier interacts with the backend server via standard database protocols such as SQL, ODBC and JDBC.
  - This middle-tier contains most of the application logic, translating client calls into database queries and other actions, and translating data from the database into client data in return.

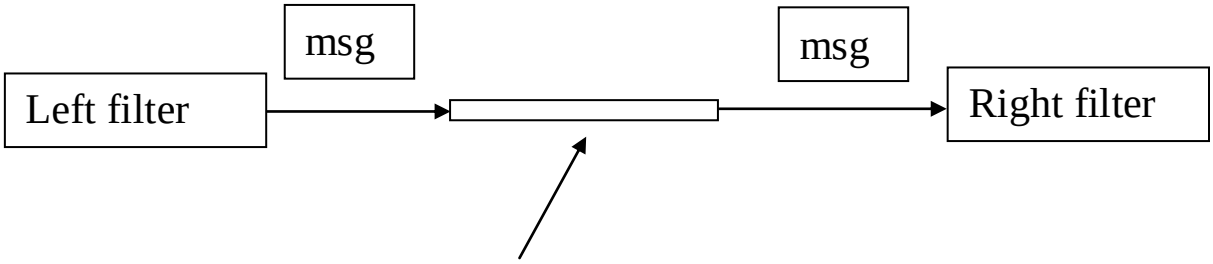
# 5. Independent Processes

## 5.1 Communicating Processes:

- A communicating process is an object with input and output ports.
- A port is an identifiable means of "wiring" process together. Ports are connected by input and output channels.
- Processes are connected together to form different topology, i.e., mesh, tree, etc.
- A communicating processes form of architecture includes: Processes and channels to communicate these processes.
- Examples: Pipeline, Mesh, Tree
- Process-based architecture can be described using Communicating Sequential Processes (CSP) specification language. CSP was introduced by Hoare (1978, 1985).
- The following table gives the notation used by CSP:

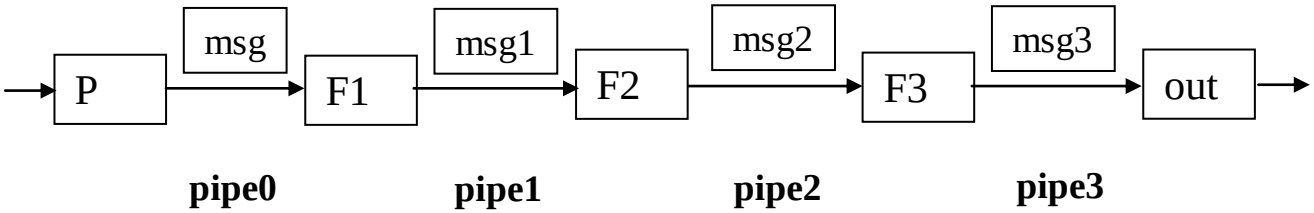
| | |
|---|---|
| A → P | Event a then process P |
| P \|\| Q | P in parallel with Q |
| P ; Q | P successfully followed by Q |
| B * P | While b repeat P |
| (a → P \| b → Q) | Choose a → P or b → Q, depending on evaluation of a, b (assume a b) |
| *P | Iterate P |
| VM = P | Process P named VM |
| X:= e | Assign value e to x |
| B ! e | Output value of message e on channel b |
| B ? e | Input message e from channel b |

- Example 1:




pipe = (left?msg ->  (right!msg ->pipe))

Input message from Left filter & output message to Right filter




- ✓ Let P be the name of a process that supplies the input to the pipeline, and out is the process that output the final result of the pipeline.
- ✓ Fi be filters of the pipe.
- ✓ Message msg is received by the left channel, output by the right channel, and then the process reverts back to pipe (waiting for the next message).

**Steps:**

1. Label all the pipelines.
2. Describe the overall topology of the architecture.
3. Give the details of individual pipelines.

**1. Labels the pipelines:**

Pipe0, pipe1, etc.

**2. Topology:**

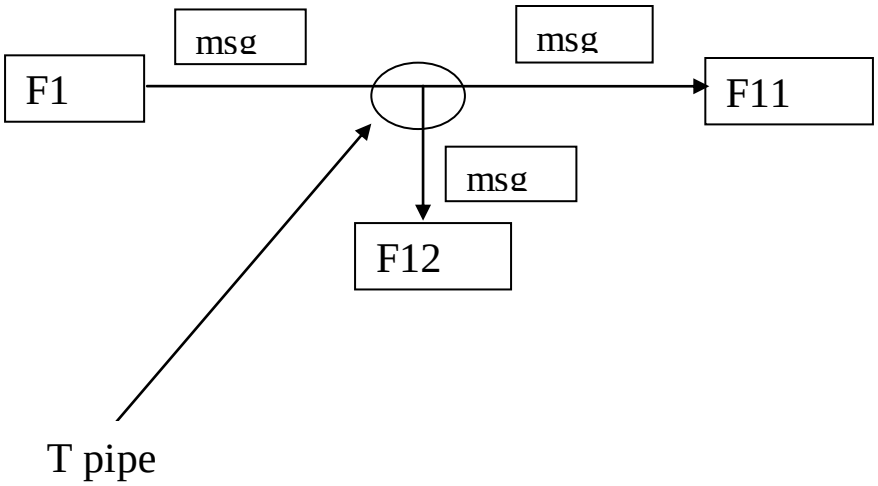
Pipeline = *(P;pipe0;F1;pipe1;F2;pipe2;F3;pipe3;out)

**3. Details:**

Each pipei is described as follows:

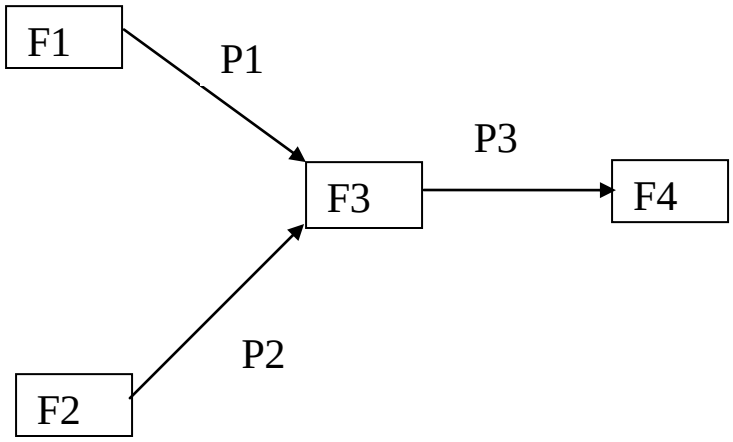
Pipe0 = (P ? msg -> (F1 ! msg -> pipe0) // input from P sent to filter F1
Pipe1 = (F1 ? msg1 -> (F2 ! msg1 -> pipe1) // input from F1 sent to filter F2
Pipe2 = (F2 ? msg2 -> (F3 ! msg2 -> pipe2) // input from F2 sent to filter F3
Pipe3 = (F3 ? msg3 -> (out ! msg3 -> pipe3) // input from F3 sent to filter out

- Example 3: a T pipe




tpipe = (F1 ? msg -> ( F11 ! msg ->tpipe
|| F12 ! msg -> tpipe)
)

Example 4: Choice "|"

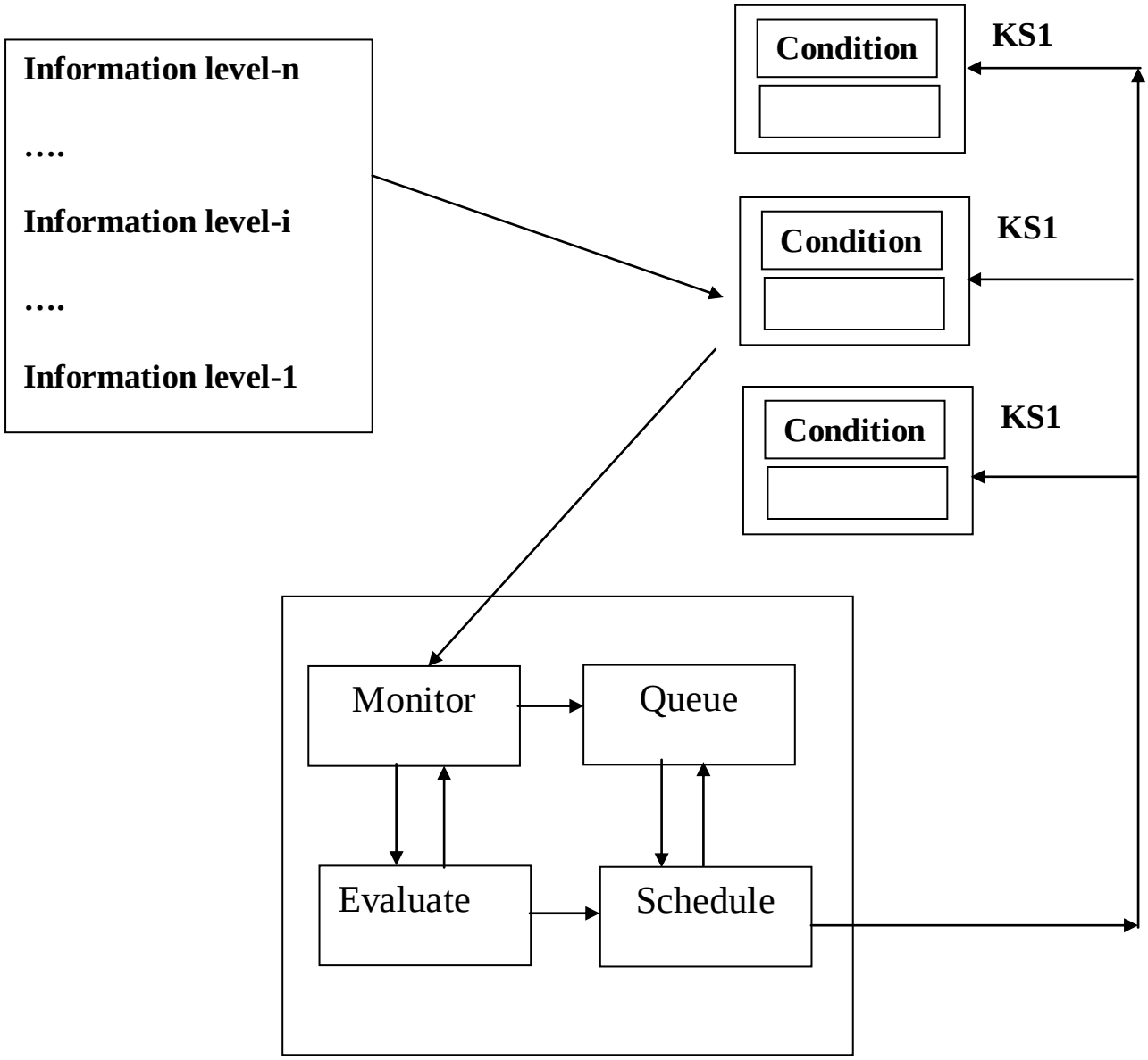



P = *((F1;P1) | (F2;P2));F3;P3;F4)
P1 = F1?m → (F3!m → P1)
P2 = F2?m → (F3!m → P2)
P3 = F3?m → (F4!m → P3)

# 6. Repository

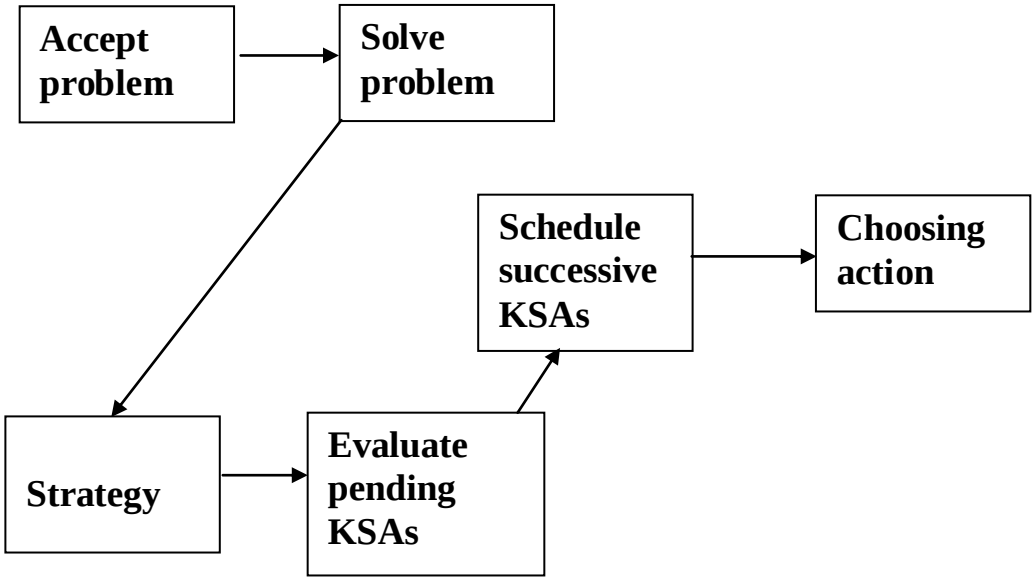
## 6.1 Blackboard Architecture

- The blackboard architecture is used as a central repository for all shared information
- It is a knowledge-based form of repository appropriate in applications requiring cooperative problem solving by virtual minds, human minds, or both (Hayes-Roth 1985).
- There are three basic components of a blackboard architecture:
  - ✓ Knowledge source: are independent expert panel members (processes) for particular problem parts. Their actions are triggered by satisfaction of particular conditions.
  - ✓ Blackboard: Repository of problem-solving state data, organized in an application-dependent hierarchy: level-n (highest) to level-1 (lowest).
  - ✓ Control: (1) monitors information in the blackboard, (2) maintains permissible combinations of knowledge source activations, (3) schedules pending knowledge-source activations (KSA), (4) evaluates local problem specific to the blackboard.




1. An event on the blackboard panel can simultaneously trigger diverse knowledge. But only one knowledge source can run a task at a time.
2. Once a condition in a knowledge source is satisfied, this adds a knowledge activation resource record (KSAR) to a queue.
3. The *scheduler* selects the appropriate KSARs and calls the responsible knowledge sources to run the tasks. Tasks are run after all preconditions have written their KSARs.

- **Backboard control**



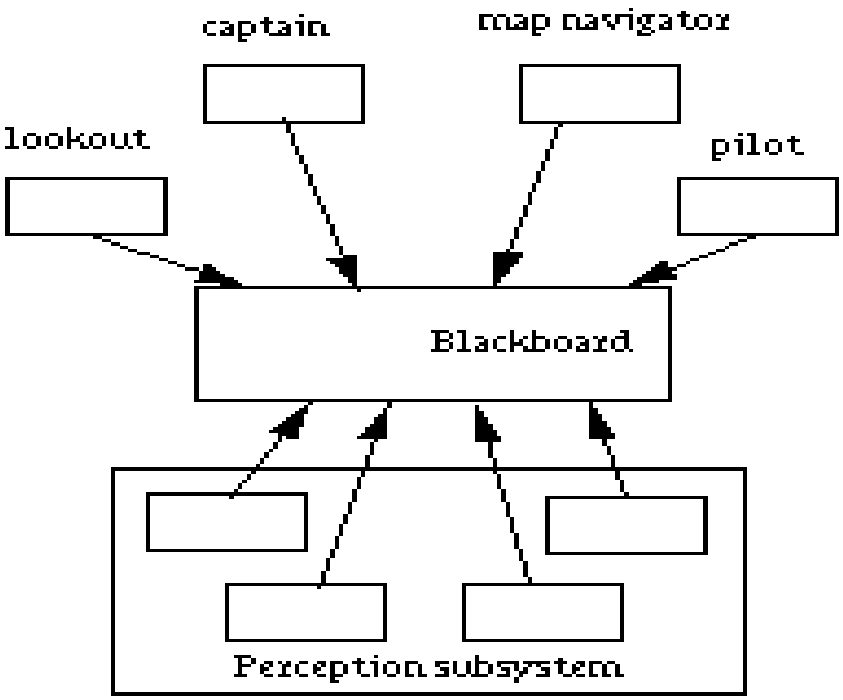

  - Benefits:

    - ✓ **Integration of knowledge sources** is managed directly by the control system.
    - ✓ **Modularity** -- each knowledge source is independent which eases development and maintenance.
    - ✓ **Flexibility** -- the **Blackboard** architecture allows blackboard applications to adapt to changing requirements much more flexibly than the rigid traditional procedural software applications.
    - ✓ **Software reuse** -- accrues in three ways:

      - The independence and modularity of knowledge sources means that new applications can easily be constructed using existing knowledge sources.
      - Legacy (traditional procedural) software investments can be preserved because they can be incorporated as knowledge sources.
      - The Blackboard itself is application independent, and is easily applied to new problem domains.

    - ✓ **Extensibility** -- new knowledge sources may be developed and added without impacting the existing system

- Examples
    - Hearsay II Implementation
- Mobile Robot: Solution 4: Blackboard Architecture
    - http://www.cs.cmu.edu/~ModProb/MRsol4.html
    - It was used in the NAVLAB project, as part of the CODGER system [Shafer86].




    - The components of CODGER are:
        - The "captain", the overall supervisor.
        - The "map navigator", the high level path planner.
        - The "lookout", a module that monitors the environment for landmarks.
        - The "pilot", the low level path planner and motor controller.

    - The perception subsystem, the modules that accept the raw input from multiple sensors and integrate it into a coherent interpretation.
- (R1) The components (including the modules inside the perception subsystem) communicate via the characteristic central database of the blackboard systems. Modules indicate their interest in certain types of information. The database returns them such data either immediately or when some other module inserts them into the database. For instance, the lookout may watch for certain geographic features; the database informs it when the perception subsystem stores images matching the description.
- (R2) The blackboard is also the means for resolving conflicts or uncertainties in the robot's world view. For instance, the lookout's landmark detections provide a reality check for the distance estimation by dead-reckoning, both stored in the database. The modules responsible for the uncertainty resolution register with the database to obtain the necessary data.

# 7. Architecture Evaluation: Good Structure

## 7.1 Goals

- **Maximize** interaction within each (**cohesion**) and **minimize** interaction between components (**coupling**)

## 7.2 Cohesion

- The degree to which the internals of a component are related
- A component has high cohesion if all of its elements are strongly related: elements are grouped together for a logical reason, not just by chance. They cooperate to achieve the common goal of the component.
- High cohesion ➔ well-designed reusable component
- There are seven cohesion levels:

1. Coincidental cohesion ➔ **BAD**
2. Logical cohesion
3. Temporal cohesion
4. Procedural cohesion
5. Communication cohesion
6. Informational cohesion
7. Functional cohesion ➔ **GOOD**

**3. Coincidental cohesion**

- A component has coincidental cohesion if it performs completely unrelated actions.

- Ex.

```
Component (p1,p2,p3)
Begin
    Update_item_record (p1);
    Delete_orders(p2);
    Insert_new_customer_info(p3);
End
```

- Problem: Not reusable.

**4. Logical cohesion**

- A component has logical cohesion when it performs a series of related actions, one of which is selected by the calling component.

- Ex.

  ➔ New_operation(function_code,p1,p2,p3)
  /* p1 and p3 are not used when this function is called with a function_code > 7 and less than 20 */

  ➔ A component performing all I/O operations: disk, tape, printers, etc.

- Problems:

  ➔ Component interface is difficult to understand
  ➔Difficult to maintain

**5. Temporal cohesion**

- A component has temporal cohesion when it performs a series of related actions in time.

- Ex.

  ➔ An initialization component:
  Initial several unrelated objects: customer_table, item_table, etc.

  ➔ Components to manipulate these objects are located in other components

- Problems: Maintenance and Reusability

**6. Procedural cohesion**

- A component has procedural cohesion if it performs a series of actions related by the sequence of steps to be followed by the product.

- Ex.

  Read(part_number, part_table);
  Update(repair_record)
  maintain(cutomer_table)

- Better than temporal cohesion

- Problem: Reusability: actions are weakly related.

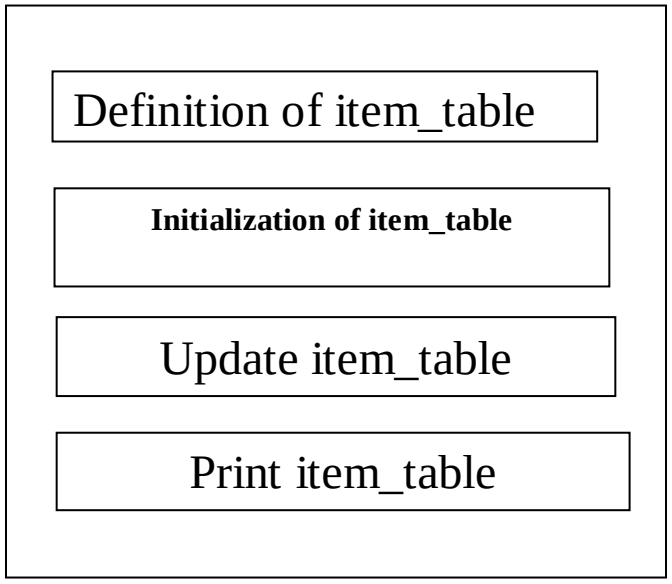
**7. Communicational cohesion**

- A component has communicational cohesion if it performs a series of actions related by the sequence of steps to be followed by the product and if all actions are performed on the same data.
- Better than procedural cohesion: actions of the component are closely related.
- Ex.

```
Update_record(R,table);
Write_record(R,audit_table);
```

**8. Informational cohesion**

- A component has informational cohesion if it performs a number of actions each with its own entry point, with independent code for each action, all performed on the same data structure.
- All actions in an informational cohesion level are closely related.
- Informational cohesion is optimal for OO.
- Ex: ADT



**9. Functional cohesion**

- A component that performs one action and achieves one single goal has functional cohesion.

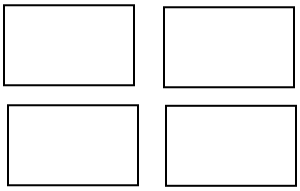
- Ex.

  Compute(sales_commission);

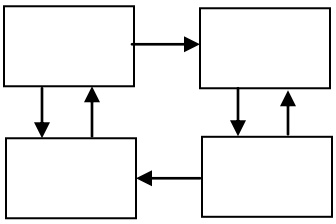
- Communicational cohesion: reusable, easy to maintain and to understand.

## 7.3 Coupling

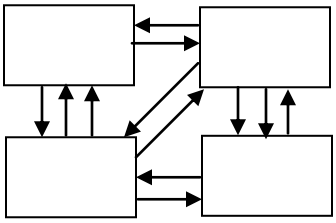
- Originally designed to support programming in the large
  - Components
  - Encapsulation
  - Interfaces



Uncoupled
no dependencies



Loosely coupled
some dependencies



Highly coupled
many dependencies

- Coupling measures the degree to which the components of a design are related.

- Ex. Component A calls a routine provided by B or access an object of component B.

  Component A and B have high coupling if they depend on each other heavily.

- low coupling is a desirable feature. Components are:

  ➔ Components are Reusable

➔ Components are easily tested and modified
➔ Reduction of the maintenance cost

- There are five coupling levels:

1. Data coupling ➔ GOOD
2. Stamp coupling
3. Control coupling
4. Common coupling
5. Content coupling ➔ BAD

**1. Content Coupling**

- Two components are content-coupled if one directly references the contents of the other.
  - Ex.
    - Component p modifies a statement of component q
    - Component p refers to local data of component q
    - Component p refers to a local label of component q

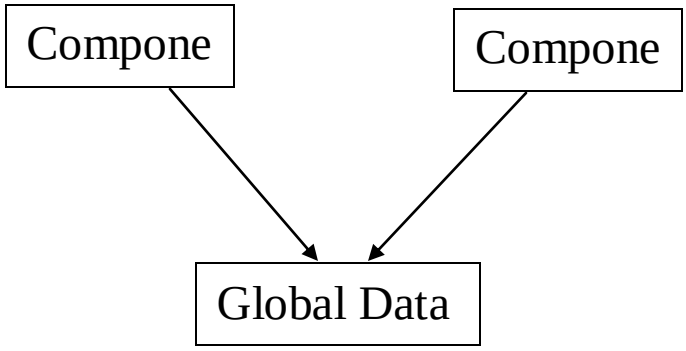
- Problems:

  - Any change to q may require a change to p.

  - It is impossible to reuse component p without reusing q

**2. Common Coupling**

- Two components are common-coupled if they both have access to the same global data.
- Ex.




  - Both component p and component q read and write a record of a database. Read-only is not a common coupling.

- Problems:
  - Unreadable code:
    - ➔

```
While(global_variable==0)
{
    if(local_variable>val)
        function_1();
    else
        function_2();
}
```

    - ➔ global_variable may be changed by function_1() or function_2().
    - ➔ Modification of the declaration of the global_variable may yield a modification of every component that accesses the global variable.
  - Common-coupled are not reusable
    - ➔ They depend on components where global variables are declared.

### 3. Control Coupling

- Two components are control-coupled if one passes an element of control to the other component; one component explicitly the logic of the other.

- Ex.

  If p calls q and q passes back a flag to p that says, "I am unable to perform the requested action Ai, please write the message number 1020.

➔ q informs p as what to do ➔ Control-coupling

### 4. Stamp Coupling

- Two components are stamp-coupled if a whole data structure is passed as an argument but the called component operates on only some of the individual components of that data structure.

- Problems:
  - Data access cannot be controlled: More data is passed than needed.

- Optimization:

  - Passing different variables ➔ slower
  - Passing one single record ➔ faster

    ❖ Knuth's first law: Don't!!

➢ Required optimization ➔ Leave it to the experts

❖ Knuth's second law: Not yet!!

**5. Data Coupling**

- Two components are data-coupled if all arguments are homogeneous data items.
- It is a desirable goal.

- Data-coupled components are easier to maintain.

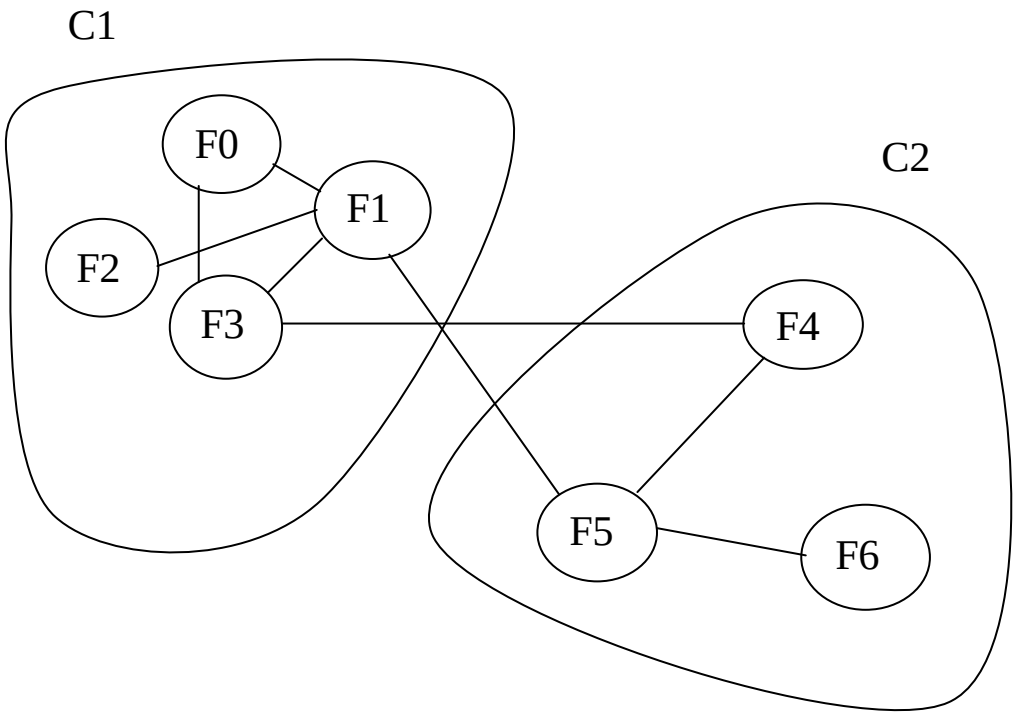
# 8. Formal Definitions

- A software system can be modeled as a graph. Each node (Fi) in the graph corresponds to a method (function or procedure) call of a component and each edge corresponds to an interaction (dependency) between methods in the system:

- Cohesion:
  - The cohesion of a component is the extent to which its individual methods are needed to perform some tasks. Cohesion of a component is defined in terms of the ratio of internal relationships to the total number of relationships. The cohesion of a component mi is:

$$CH(mi) = Ri / (Ri+Re)$$

Where Ri is the number of internal links and Re is the number of external links.




  - Example:

The cohesion of C1 and C2 in Figure 1 are:

CH(C1) = 2/3
CH(C2) = 1/2

- The cohesion of a software system is the average cohesion of all its n components:

  $CH = (1/n) * \sum CH(Ci)$ for all Ci of the system.

- Example: The cohesion of the system in Figure 1 is:

  CH = 7/12

- Coupling

  - The coupling of a component is the ratio of the number of external links to the total number of links:

    $CP(Ci) = Re /(Ri+Re)$

  - The coupling of a system is the average coupling of all its n components:

    $CP = (1/n)* \sum CP(Ci)$ for all mi of the system.

# 9. Summary

- Architectural Elements
  - Processing elements
  - Connecting elements
  - Data Elements
  - Configuration file

- Architectural Styles (major ones)
  - **Dataflow Systems**
    - **Pipelines**
      - Each layer is client for layer below it
      - output of one stage = input to next
      - Example: Compilers
  - **Call & Return Systems**
    - **Layered**
      - Each layer is client for layer below it
      - advantages: incremental, extendable
      - N-tier / Client-Server
      - layers can be developed independently
      - Examples: Operating Systems, Web-based applications
  - **Independent-Process**
    - **Communicating Processes:**
      - Using CSP to describe different process topologies
  - **Repository**
    - **Blackboard**
      - central repository for shared info
      - 3 components – knowledge source, controller, repository (blackboard)
      - Example: no one single answer – fingerprints