Software Design and Architecture

[Architecture Patterns] – Chapter 04, L02

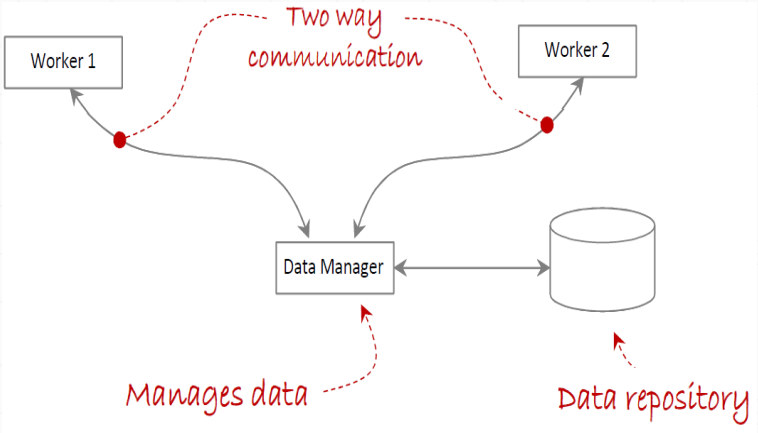
Lecture Outlines- Architecture Patterns – Chapter 04

- Data-Centered Systems
  - Overview
  - Patterns:
    - Blackboard
- Data Flow Systems
  - Overview
  - Patterns:
    - Pipes-and-Filters
- Distributed Systems
  - Overview
  - Patterns:
    - Client Server
    - Broker

Data-Centered Systems – Overview

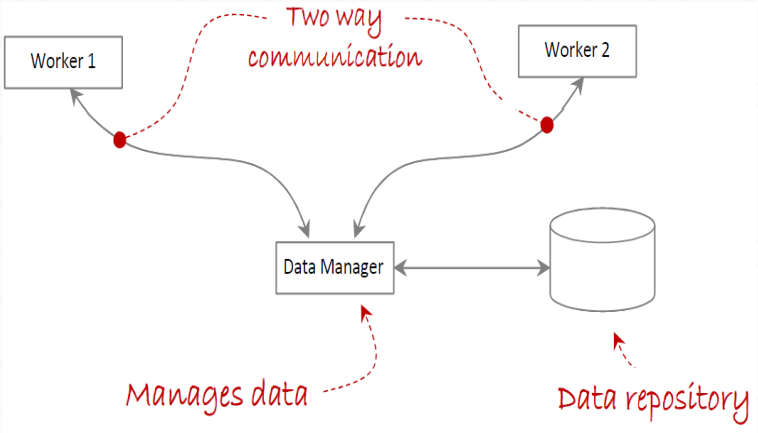
➢ Data-centered systems are systems primarily decomposed around a main central repository of data. These include:

- **Data management component**: the data management component controls, provides, and manages access to the system's data.
- **Worker components**: worker components execute operations and perform work based on the data.



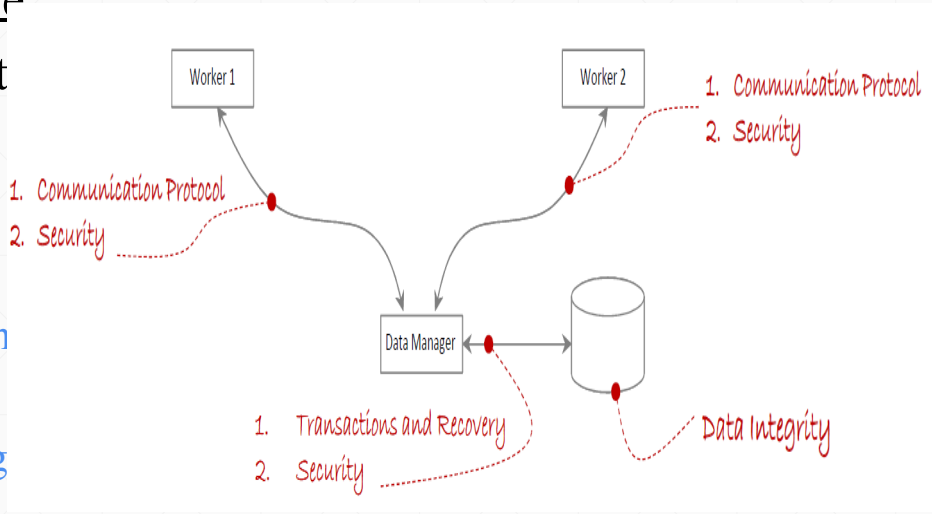
Data-Centered Systems – Overview

- Communication in data-centered systems is characterized by a one-to-one bidirectional communication between a worker component and the data management component.
- Worker components do not interact with each other directly; all communication goes through the data management component.



Data-Centered Systems – Overview

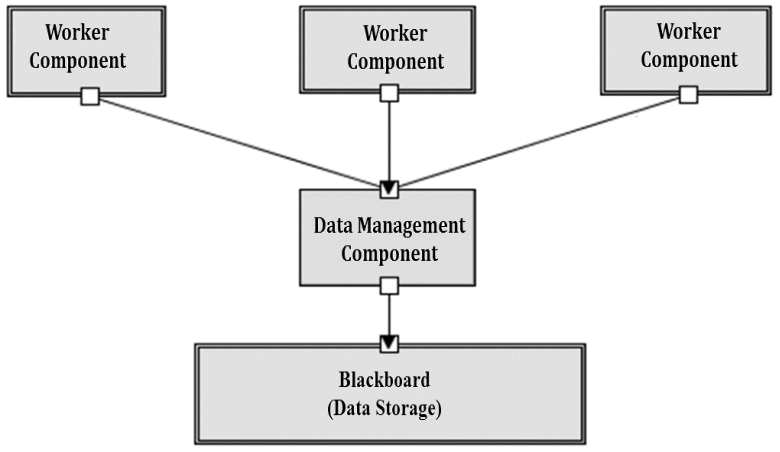
- Because of the architecture of these systems, they must consider issues with:
  - **Data integrity**
    - **Extra: Integrity** means that data is complete, accurate, an consistent across various databases, systems, or during any operation (like transfer, storage, or retrieval).
  - **Communication protocols** between worker and data management



Extra: A **DBMS** can be viewed in **two ways**:
•As a **Repository** (Data-Centered Style) → all apps use one central shared database.
•As part of a **Client-Server system** (Distributed Style) → clients send SQL queries + server processes them+ Database → stores data.

Blackboard Architectural Pattern

- Blackboard decomposes systems into components that work around a central data component to provide solutions to complex problems.
  - These components work independently from each other to provide partial solutions to problems using an opportunistic problem-solving approach.
  - That is, there are no predetermined, or correct, sequences of operations for reaching the problem's solution.



Blackboard Architectural Pattern

➢ **The Blackboard architectural pattern resembles the approach a group of scientists would employ to solve a complex problem.**

- **Consider a group of scientists at one location using a blackboard (chalkboard, whiteboard, or electronic blackboard) to solve a complex problem.**
- **Assume that to manage the problem-solving process, a mediator controls access to the blackboard.**
- **Once the mediator (or controller) assigns control to the blackboard, a scientist evaluates the current state of the problem and if possible, advances its solution before releasing control of the blackboard.**
- **With new knowledge obtained from the previous solution attempt, control is assigned to the next scientist who can further improve the problems' state.**
- **This process continues until no more progress can be made, at which point the blackboard system reaches a solution.**
- **Extra note:**
  - **There is no strict order in which team members must contribute. They jump in as needed, making decisions on the spot about the best way to add to the solution based on the current information available.**

➢ **This behavior is famous in expert systems; therefore, the Blackboard architectural pattern is a good choice for depicting the logical architecture of expert systems.**

- **Extra Note:**

- **An expert system is a type of AI that uses knowledge and inference rules to solve complex problems typically handled by human experts. It mimics human decision-making to provide solutions in fields like medicine, finance, and customer service.**

➢ Extra explanation for the previous slide:

➢ The blackboard architecture is like a group project where each person works on their part independently but uses a shared whiteboard to write down their ideas and see others' contributions. This way, everyone can add, modify, or build upon the information on the whiteboard at any time, moving the project forward based on what is currently known or needed.

Blackboard: A central hub like a communal whiteboard where all information is displayed and updated, allowing everyone involved to see and contribute.

Agents: Independent contributors, each with unique skills, who interact with the blackboard to add their expertise and update information as needed.

Controller: The organizer who manages how and when the agents access the blackboard, ensuring the project progresses smoothly and efficiently.

Extra: Real-World Example: Emergency Room (ER)

Imagine an emergency room in a hospital as a blackboard system:

•**Blackboard**:
The patient's chart, visible and accessible by all medical staff, where information about the patient's condition and treatment is continuously updated.

•**Agents**:
Various medical professionals (doctors, nurses, specialists) who view the patient's chart (blackboard) and contribute based on their expertise.

- •For instance, a nurse might update vital signs, a doctor orders blood tests or some medication, and a specialist provides their opinion on a complex condition.

•**Controller**:
it will manage when and which agents have access to the blackboard (or patient chart) and may prioritize or schedule their contributions based on the patient's changing condition and the updates on the chart.

Blackboard Architectural Pattern

Extra Explanation:

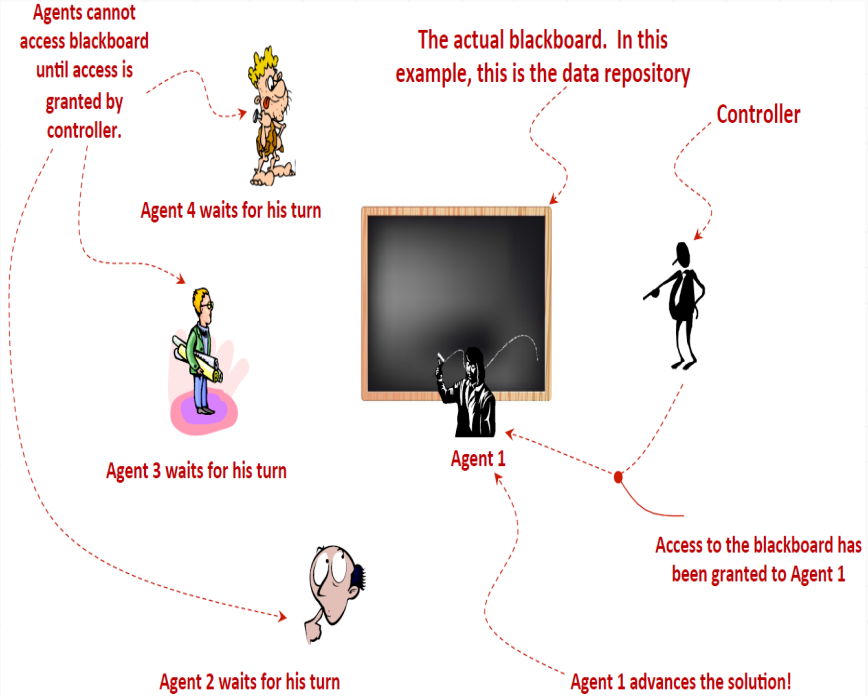
Blackboard: The central data repository where all the current information is displayed and accessible.

Agents: There are multiple agents (Agent 1, Agent 2, Agent 3, and Agent 4), each waiting for their turn to access the blackboard. Each agent brings unique expertise and contributions to solve parts of a larger problem.

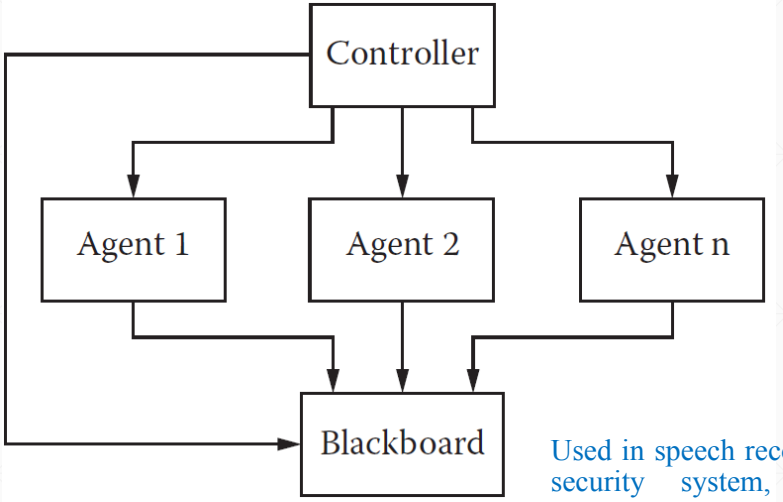
Controller: Manages access to the blackboard. The controller decides which agent should access the blackboard at any given time, ensuring an orderly and productive problem-solving process.

Process:

Agent 1 is currently accessing the blackboard and making contributions or updates based on the existing information. Other agents (Agent 2, Agent 3, Agent 4) are waiting for their turn to access the blackboard as controlled by the controller.



Blackboard Architectural Pattern



Used in speech recognition, image recognition, security system, and business resource management systems

Blackboard Architectural Pattern

Extra

Consider a Students' Scheduling System where the students need to enroll in courses and create a schedule based on the **history of previous completed courses**, his/her **working schedule**, and the **courses offered** by the university.

1. **Blackboard**
   - A *shared knowledge base*.
   - Everyone (Controller + Agents) **reads/writes here**.
   - Think of it like a "big whiteboard in the room" where all info is posted.
2. **Agents**
   - Specialized problem solvers (e.g., StudentHistory, WorkSchedule, CourseOfferings).
   - They **watch the Blackboard** to see if they can contribute.
3. **Controller**
   - Coordinates *when* and *which* agent should act.
   - Decides the order of problem-solving.

Why Blackboard Has Provider Interfaces?

- Blackboard is the **central hub**.
- Both **Controller** and **Agents** need to **access shared data**.
- That's why Blackboard provides interfaces → so anyone can **read/write/update info**.

If Blackboard didn't provide this:

- Agents would need to talk directly to the Controller for data, which breaks the Blackboard idea (central knowledge store).
- The Controller would become too "heavy," holding all data instead of just managing flow.

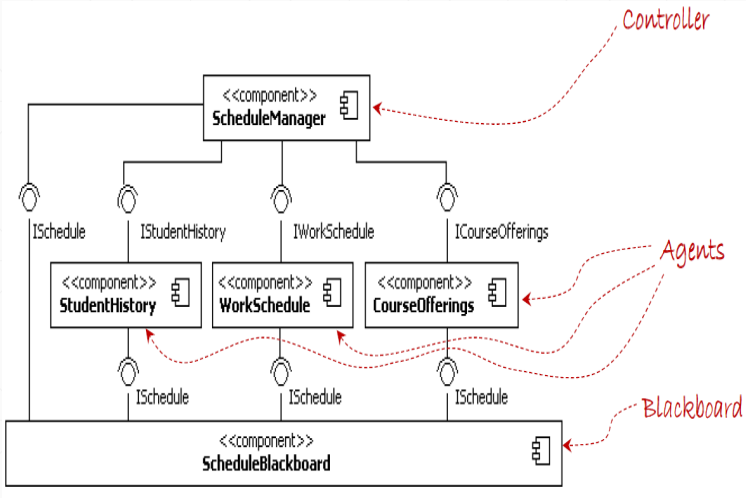
Why Controller Doesn't Provide Interfaces to Agents?

- The Controller's job is *not* to store data → its job is to **orchestrate**.
- Agents don't ask the Controller for data.
- Instead, the Controller just tells:
  "Hey Agent, now check the Blackboard."
- Then the Agent goes to Blackboard (through its interface) and does its work.

Why Agents Provide Interfaces to Controller?

The Controller needs to invoke Agents when it decides they should act. For this, Agents expose a provider interface (like "services" they can perform). Example: The Controller says: "CourseOfferings Agent, run your algorithm on the Blackboard. " It can only do this if the Agent provides an interface (like run(), update(), etc.).

(component diagram)



Extra: Continue

- **Who does what (one line each)**
  - **Blackboard**: shared data store; exposes an API to read/write/update facts.
  - **Agents**: specialized solvers; when invoked, they **use the Blackboard API** to read inputs and write results.
  - **Controller (Scheduler)**: orchestrates **when/which** agent runs; it doesn't pass data around.
- **Why those interfaces in the component diagram?**
  - **Blackboard → provided interface (lollipop)**: "Here's my data API (read/write/update)."
    - **Agents & Controller** have **required** interfaces to the Blackboard because they need that API.
  - **Agents → provided interface**: "Here's my service (e.g., run(), update())."
    - **Controller** has a **required** interface to each Agent so it can trigger them.
  - **Controller** generally **does not** provide data services to Agents; it just **invokes** them.

Step-by-step flow (typical cycle)

1. **Controller** decides "Agent A should run now." → calls AgentA.run().
2. **Agent A** reads what it needs **from the Blackboard**.
3. **Agent A** computes and **writes** new/updated knowledge **to the Blackboard**.
4. **Controller** checks the Blackboard (or gets notified), then picks the next Agent.

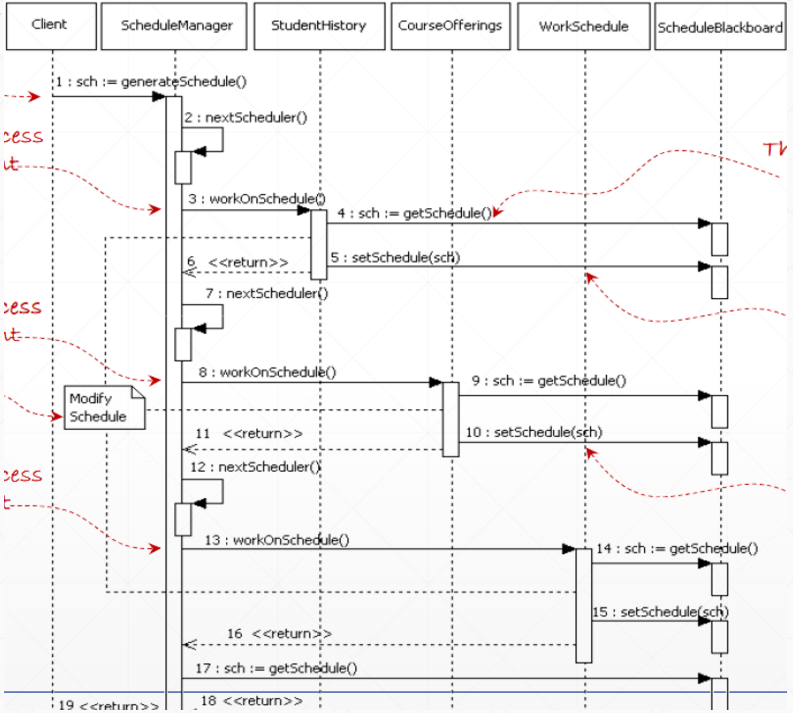
General guideline

- **Control flow**: Controller → Agents (invocation).
- **Data flow**: Agents ↔ **Blackboard** (read/write).
- **Agents do *not*** talk to each other directly; they communicate **indirectly via the Blackboard**.

So: your "agents can't access the blackboard" interpretation is the only incorrect piece—agents **should** access it, but **only through the Blackboard's interface**, and **only when the Controller schedules them**.

Blackboard Architectural Pattern

- **Extra:**
- **Client**: Student portal UI → clicks **"Generate Schedule"**.
- **ScheduleManager**: Controls the process → decides which scheduler runs next.
- **StudentHistory**: Student record → passed courses, prerequisites, restrictions.
- **CourseOfferings**: Course data → available sections, times, instructors.
- **WorkSchedule**: Scheduling modules → each one improves the schedule (check eligibility, remove conflicts, optimize preferences).
- **ScheduleBlackboard**: Shared workspace → stores the current draft schedule using getSchedule() / setSchedule().



- **Flow:**

Extra Explanation:

How Your Schedule is Built: Step-by-Step

1-**The Client** asks the **ScheduleManager** to create a schedule by calling generateSchedule().

2- The **ScheduleManager** (the controller) gets the next agent (nextScheduler()) to coordinate with three specialist agents, who all use the shared **ScheduleBlackboard** to build the schedule.

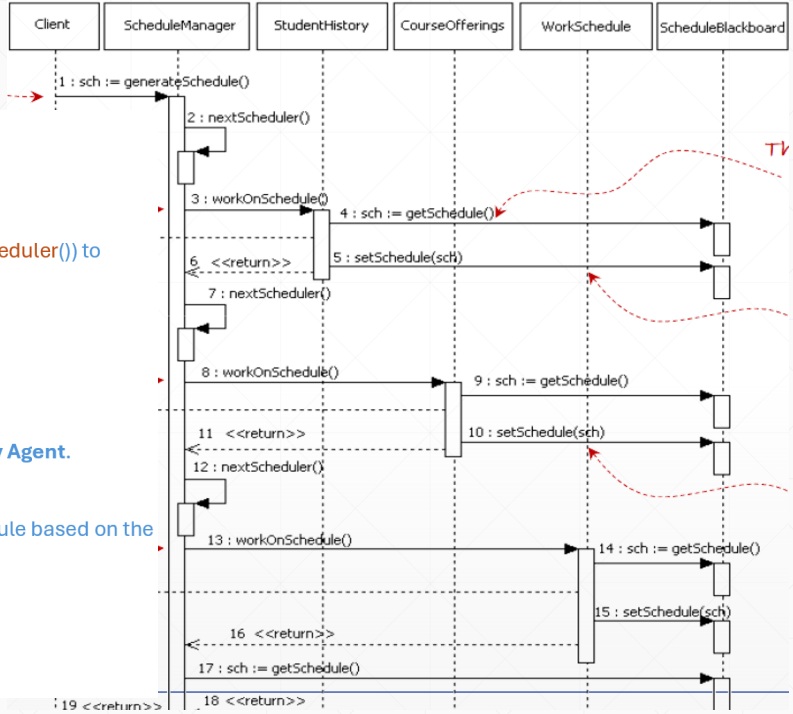
Phase 1: StudentHistory Agent Works

3- **ScheduleManager** calls workOnScheduler() on the **StudentHistory Agent**.

4- **StudentHistory Agent** reads the current schedule data from the **ScheduleBlackboard** using getScheduled(). It modifies the schedule based on the student's past courses and degree requirements.

5- It saves this updated version back to the **ScheduleBlackboard** using setScheduled(sch).

6- It returns control to the **ScheduleManager**.



Phase 2: CourseOfferings Agent Works

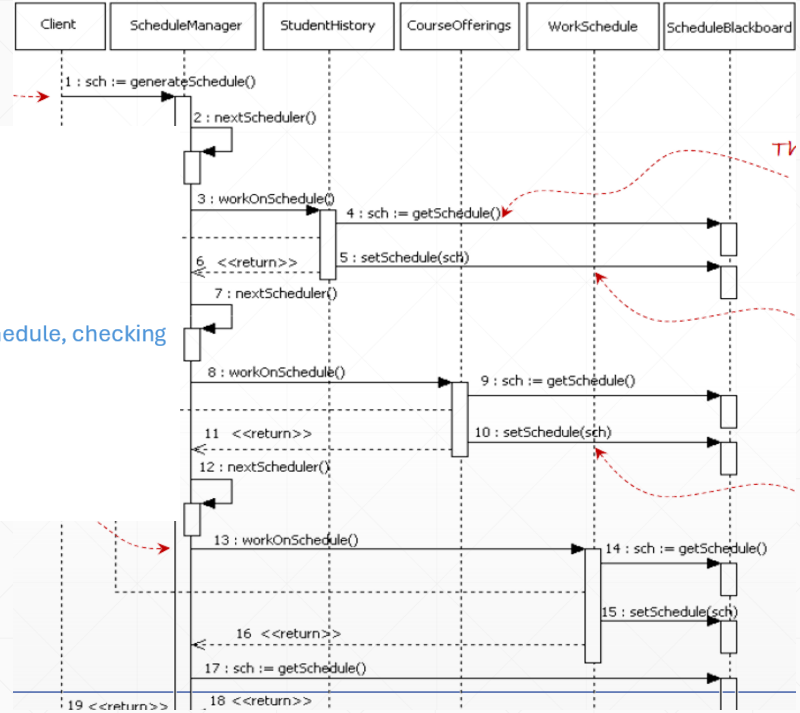
7-**ScheduleManager** gets the next agent (nextScheduler())

8-and calls workOnScheduler() on the **CourseOfferings Agent**.

9-**CourseOfferings Agent** reads the current schedule from the **ScheduleBlackboard** using getScheduled(). It modifies the schedule, checking which proposed courses are actually available this semester.

10-It saves this realistic version back to the **ScheduleBlackboard** using setScheduled(sdn).

11-It returns control to the **ScheduleManager**.



Phase 3: WorkSchedule Agent Works

12-**ScheduleManager** gets the next agent (nextScheduler())

13- calls workOnScheduler() on the **WorkSchedule Agent**.

14-**WorkSchedule Agent** reads the current schedule from the **ScheduleBlackboard** using getScheduled().It modifies the schedule to avoid conflicts with the student's work commitments.

15-It saves this final, conflict-free version back to the **ScheduleBlackboard** using setScheduled(sdn).

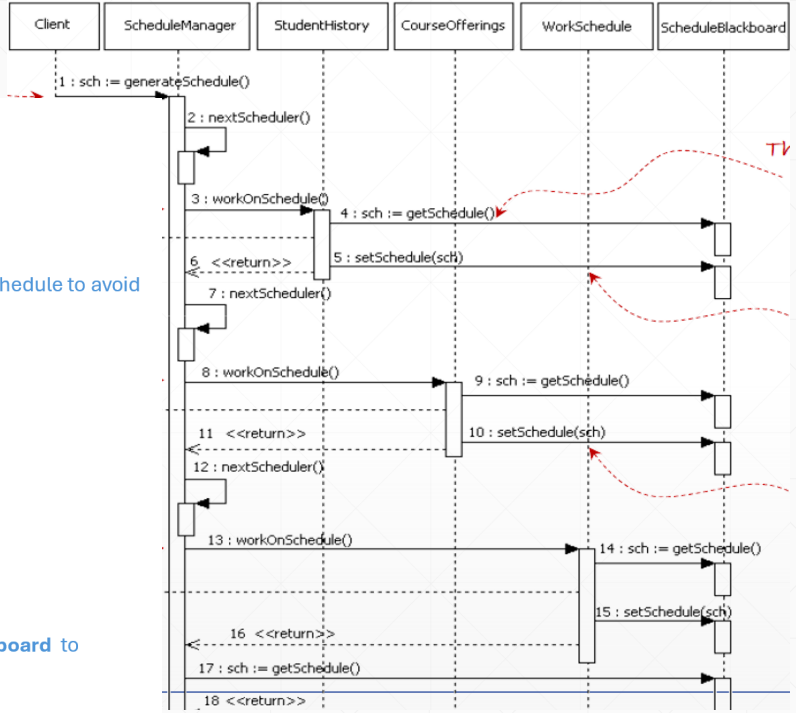
16-It returns control to the **ScheduleManager**.

Phase 4: Schedule is Delivered

17-**ScheduleManager** retrieves the completed schedule from the **ScheduleBlackboard** one final time using getScheduled().

18 -The finalized schedule is delivered back from **ScheduleBlackboard** to the **ScheduleManager**.

19 -The finalized schedule is delivered back to the **Client**.



Blackboard Architectural Pattern

Extra:

Quality	Description
Modifiability	Agents are compartmentalized and independent from each other; therefore, it is easy to add or remove agents to fit new systems.
Reusability	Specialized components can be reused easily in other applications.
Maintainability	Allows for separation of concerns and independence of the knowledge based agents; therefore, maintaining existing components becomes easier.

1. **Modifiability**: Easy to add a new **SportsSchedule** agent without changing others.
2. **Reusability**: The **StudentHistory** agent can be used in a different system, like a graduation checker.
3. **Maintainability**: A bug in the **CourseOfferings** agent can be fixed without touching the **WorkSchedule** agent.

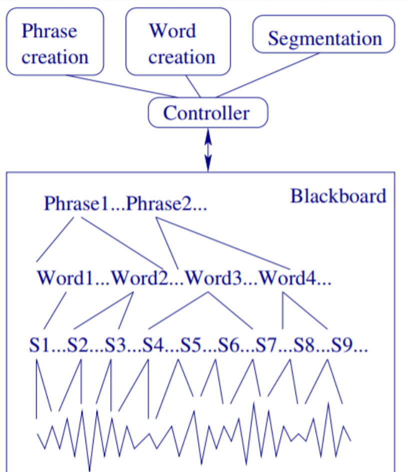
Extra: the difference between the Deployment Diagram and Component Diagram

Aspect	Deployment Diagram	Component Diagram
Purpose	**Shows physical deployment of artifacts (software, libraries, etc.) on hardware (servers, computers, devices).**	**Shows organization and dependencies among software components.**
Focus	**Hardware, runtime nodes, physical distribution, and communication links.**	**Software components, their relationships, interfaces, and dependencies.**
Example	**Web app: how web server, app server, and DB server are configured, connected, and accessed.**	**Web app: user authentication library, data access objects, business logic processors, and how they interact.**

Blackboard Architectural Pattern - Exercise

Extra :

- ➢ Building a software system for speech recognition
- ➢ Input is speech recorded in waveform in single words as well as whole sentences
- ➢ Output is a text of English phrases printed on the screen
- ➢ Use the … as a guide.



Use the bl… e **componen…** or the system ba… t.

[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

Speech Recognition System

Extra:Continue of Example

➢ **Input Text:**
"Artificial intelligence is transforming education. Students can now learn with personalized tutors. Teachers use AI to design better lessons.“

➢ **Step 1: Segmentation Agent**

- Splits into sentences:
- ***"Artificial intelligence is transforming education."***
- *"Students can now learn with personalized tutors."*
- *"Teachers use AI to design better*

➢ **Step 3: Phrase Creation Agent**

➢ Groups words into grammatically and semantically correct phrases:

➢ Sentence 1:

- Noun Phrase(NP): *Artificial intelligence*
- Verb Phrase(VP): *is transforming*
- Noun Phrase(NP): NP: *education*

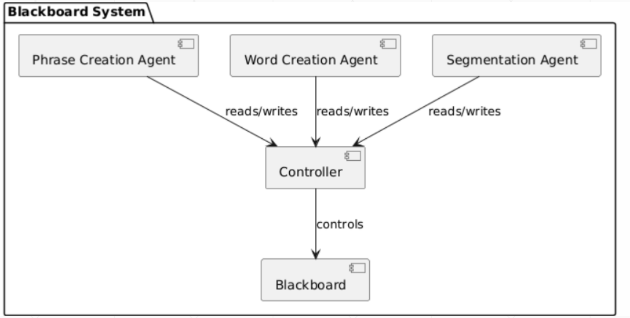
➢ **Final Structured Output**

➢ **Sentence:** *"Artificial intelligence is transforming education."*

➢ **Words:** [Artificial, intelligence, is,

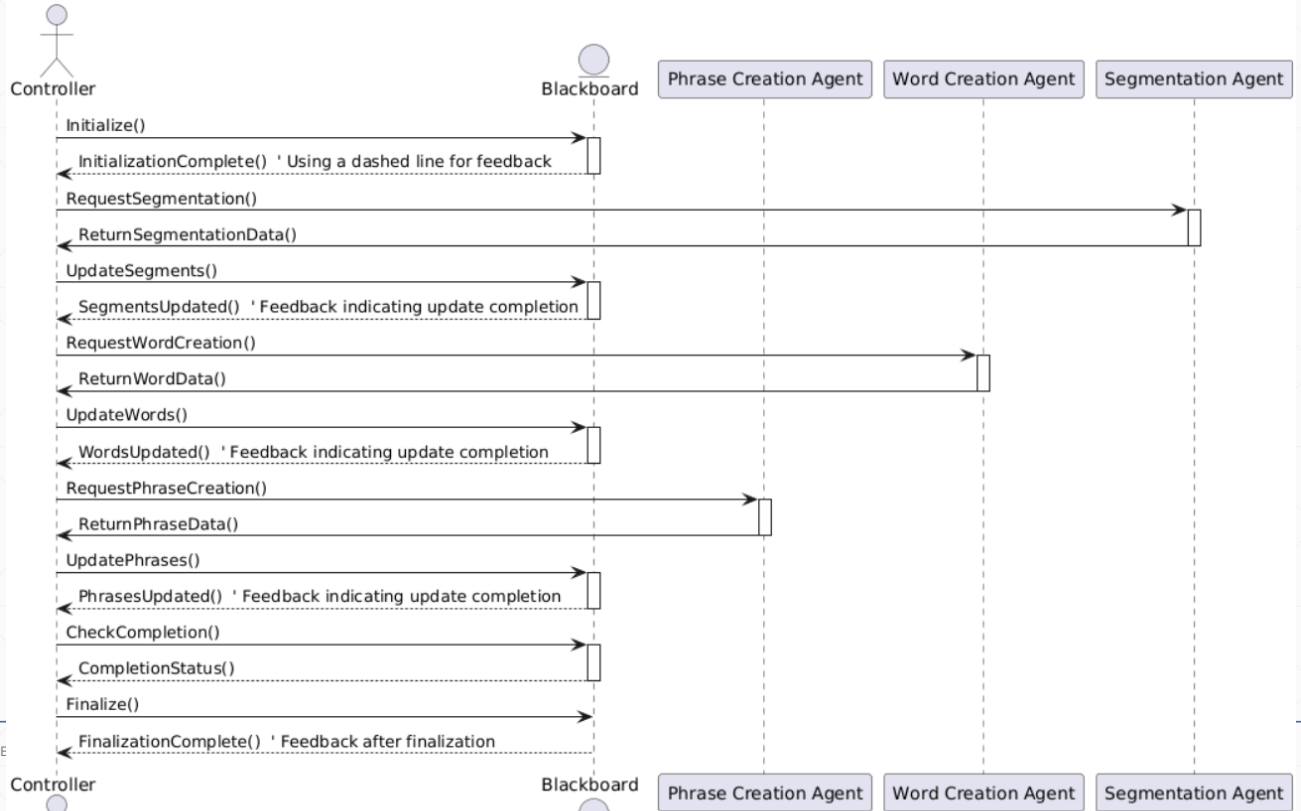
Extra: Solution – Simple Component diagram without Interfaces

[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]



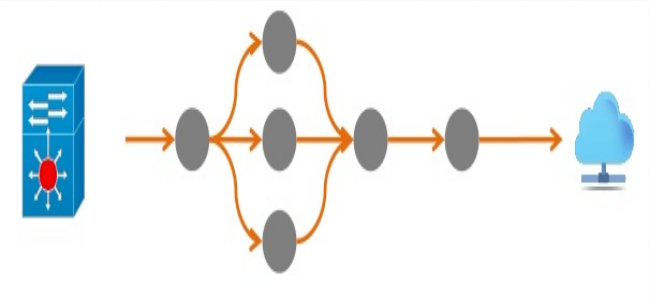
Note
•If the **controller is active** (offering explicit services like scheduling, prioritizing, or enforcing policies), then **give it a provided interface**
•If the **controller is passive/hidden** (just internally managing communication between agents and the blackboard), then **no provided interface.**

Extra: Solution –Sequence diagram



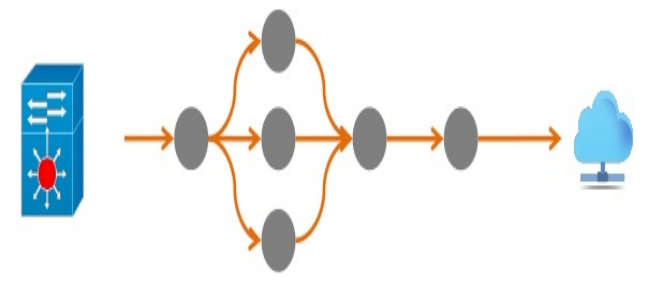
Data Flow Systems

- Data flow systems are decomposed around the central theme of transporting data (or data streams) and transforming the data along the way to meet application-specific requirements.
- Typical responsibilities found in components of data-flow systems include:
  - **Worker components**, those that perform work on data
  - **Transport components**, those that transporting data



Data Flow Systems

- **Worker components** abstract data transformations and processing that need to take place before forwarding data streams in the system, e.g.,
  - Encryption and decryption
  - Compression and decompression
  - Changing data format, e.g. ,from binary to XML, from raw data to information, etc.
  - Enhancing, modifying, storing, etc. of the data
- **Transport components** abstract the management and control of the data transport mechanisms.

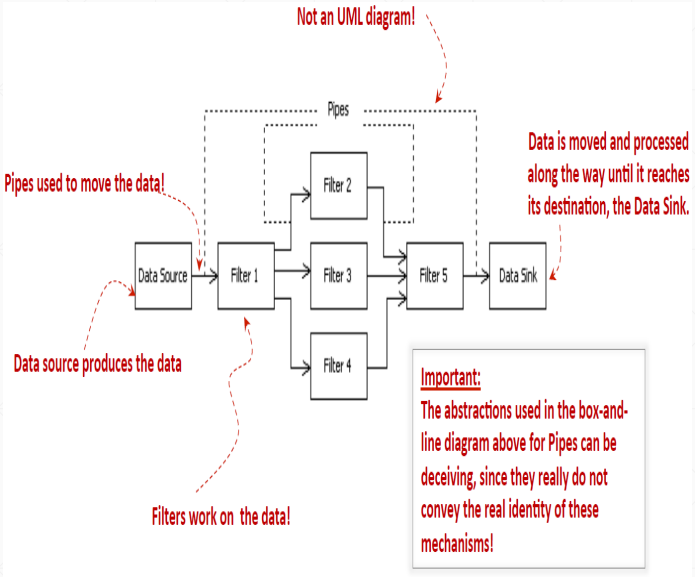


➢ An example of an architectural pattern for data flow systems is the "**Pipes-and-Filters"** pattern

Pipes-and-Filters Architectural Pattern

Extra:

- **Data Source** → Produces the input data.
- **Pipes** → Carry data from one step to another.
- **Filters** → Process the data (transform, clean, analyze, etc.) before passing it on.
- **Data Sink** → Final destination where processed data is stored or used.
- **Processing Flow** → Data moves step by step through filters until it reaches the sink.
- **Important Note** → The boxes and arrows are only abstractions; in real systems, pipes and filters are more complex than shown.



Pipes-and-Filters Architectural Pattern

➢ Pipes-and-Filters is composed of the following components:

- **Data source** - produces the data
- **Filter** - processes, enhances, modifies, etc. the data
- **Pipes** - Provide connections between data source and filter, filter to filter, and filter to data sink.
- **Data Sink** - data consumer

Pipes-and-Filters Architectural Pattern – Example1

- A common example for the Pipes-and-Filters pattern: the architecture of a Language Processor system (e.g., compiler, interpreter)
- The example is illustrated in the next slide

Pipes-and-Filters Architectural Pattern – Example1

Extra:

Components:

1. **Lexical Analyzer**:
   - **Function**: Breaks down the source code into meaningful elements called tokens.
   - **Example**: In the code snippet `int x = 5;`, the lexical analyzer identifies `int`, `x`, `=`, `5`, and `;` as tokens.
2. **Parser**:
   - **Function**: Takes tokens and forms a parse tree which represents the grammatical structure of the code.
   - **Example**: From `int x = 5;`, it creates a tree where `int x` forms a declaration node and `= 5` forms an initialization node.

3. Code Generator:

- **Function**: The code generator translates the parse tree (which organizes elements of code into a structural hierarchy) into machine code or an intermediate representation.
- **Example**: After parsing `int x = 5;`, the code generator would create assembly or bytecode instructions that allocate memory for an integer variable x and then store the value 5 in that memory location. In machine code, this could be represented as two instructions:
  - One to allocate memory for x.
  - Another to move the number 5 into the memory allocated for x.

4. Optimizer:

- **Function**: Makes the code run faster and use fewer resources.

Example: For the code `int x = 5;`, the optimizer might simplify memory operations so more efficient instruction that does both at once, reducing the overall steps the machine must perform.

instead of:

Step 1: Reserve memory for x

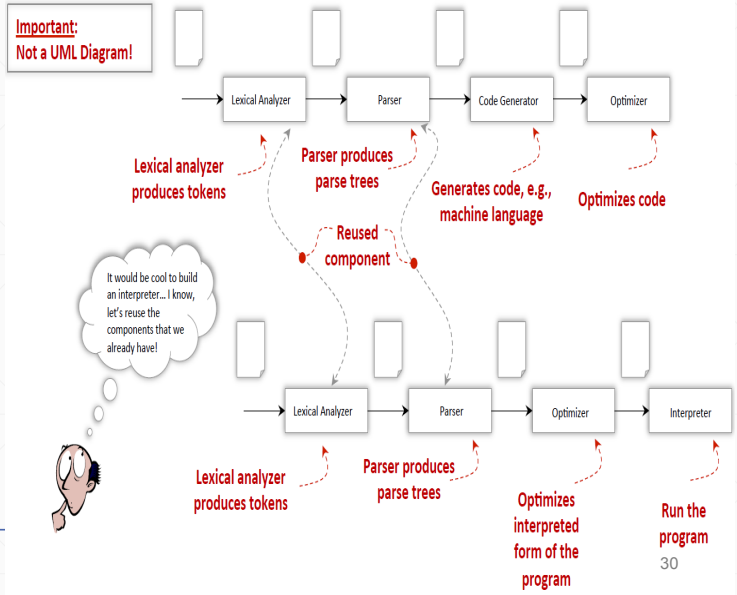
Step 2: Move the number 5 into that memory

It might combine them into:

One machine instruction that says: "create space for x and put 5 there right away."

5. Interpreter:

- **Function**: If the system uses an interpreter instead of (or in addition to) a compiler, the interpreter directly executes the code from a high-level or intermediate form without transforming it into machine code.
- **Example**: In the case of `int x = 5;`, an interpreter would directly execute these instructions at runtime:
  - Allocate memory for x.
  - Set that memory to 5.



Pipes-and-Filters Architectural Pattern – Example1

Extra explanation:

1.Lexical Analyzer (1:call())

- **Function**: The Lexical Analyzer processes the initial input data.
- **Operation**: It performs the **1:call()** function which likely involves analyzing the input data to identify tokens.

2. Parser (2:process() , 3:Call())

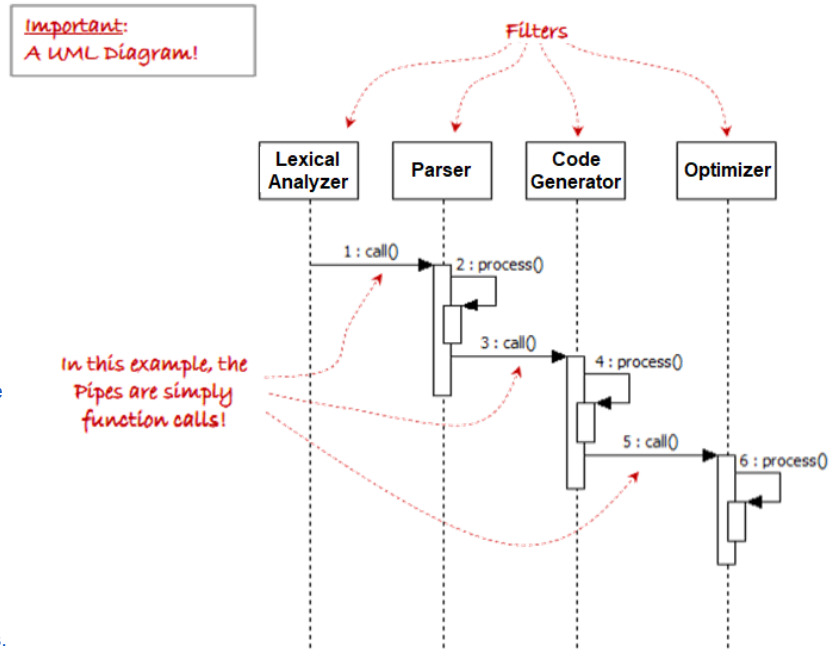
- **Function**: The Parser takes tokens from the Lexical Analyzer.
- **Operation**:
- The **2:process()** function parses these tokens to construct a parse tree or some form of syntactic representation of the input data.
  3:call() → Passes the parse tree or parsed structure to the Code Generator.

3. Code Generator (4:process(), 5:call())

- **Function**: The Code Generator takes the output from the Parser.
- **Operations**:
  - **4:process():** Translates the parse tree into intermediate code or directly into machine code.
  - **5:call():** Might finalize the code generation process, performing any final adjustments or optimizations on the code itself before it's passed to the Optimizer.

4. Optimizer (6:process())

- **Function**: The Optimizer enhances the efficiency of the code produced by the Code Generator.
- **Operation**: The **6:process()** function optimizes the code to improve performance, reduce memory usage, or streamline execution paths.



Pipes-and-Filters Architectural Pattern – Example2

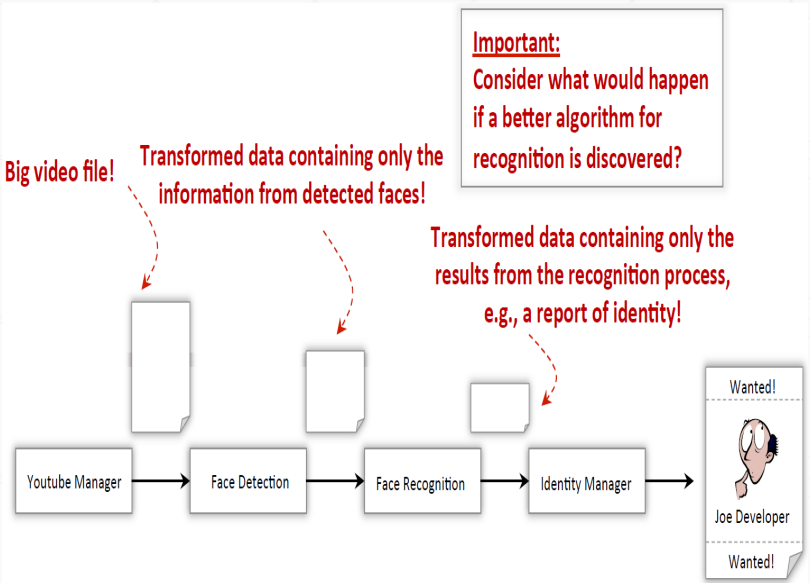
- Consider software that houses algorithms for automatically determining the identity of an individual:
  - The software access videos (with audio) from YouTube
  - The software detects faces of individuals in the video (Face detection is used to determine if a face is in the video)
  - The software recognizes faces from the video (Face recognition is used to determine the identity of the person from the detected face).
  - Based on detection and recognition, the software predicts the identity of individuals in the video
- Using the pipes and filters architecture, the logical structure of the system can be modeled as shown in the next slide

Pipes-and-Filters Architectural Pattern – Example2

•**Extra Explanation:**

•**Big Video File**: A large video file serves as the input source.
•**YouTube Manager**: This component processes the video file to prepare it for further analysis.
•**Face Detection**: This stage identifies and isolates faces within the video data. It transforms the video data into specific information related only to the detected faces.

•**Face Recognition**: This process analyzes the detected faces to identify individuals by matching features with known faces.
•**Identity Manager**: Produces a final output based on the face recognition results, such as identity reports, which could be used for verification or security purposes.



Pipes-and-Filters Architectural Pattern – Example2

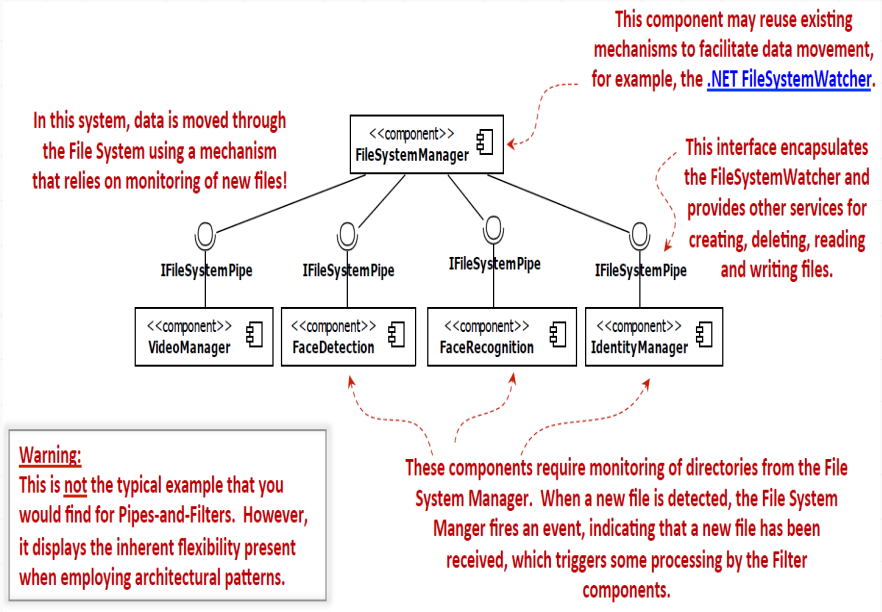
- In the previous example, the box-and-line diagram was useful for visualizing the components in the system, however, it conveyed nothing about **how data is transported** from one Filter to the next, (i.e., the Pipes).
- Consider the following UML component for the same system to understand how the data is transported between the different filters (components) in the system

Pipes-and-Filters Architectural Pattern – Example2

•**Extra:**
•**The system uses .NET's FileSystemWatcher :** to monitor directories for new video files, triggering the data processing pipeline when new files are detected.

•**Processing Pipeline**: Consists of several components:

- •**Video Manager**: Handles initial video file management.
- •**Face Detection**: Detects faces in the videos.
- •**Face Recognition**: Identifies the detected faces.
- •**Identity Manager**: Manages identity data from recognized faces.



Pipes-and-Filters Arch Exar

Extra Explanation:

1. VideoManager
- `put(file)` → Puts new video file into the system.

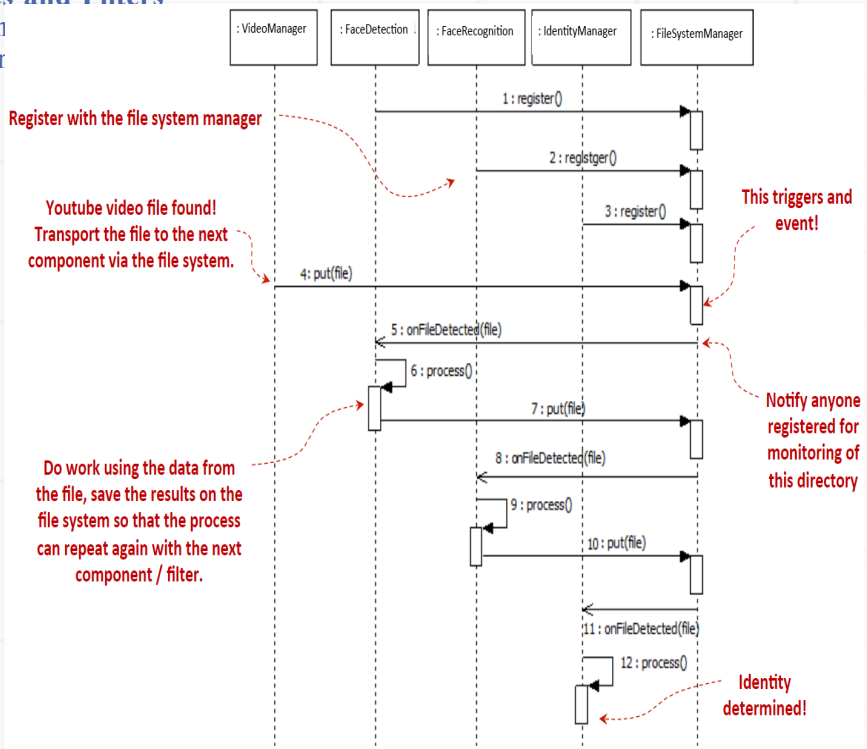
2. FaceDetection
- `register()` → Registers with FileSystemManager.
- `process()` → Processes video data for face features.
- `put(file)` → Passes processed file to next stage.

3. FaceRecognition
- `register()` → Registers with FileSystemManager.
- `process()` → Identifies and recognizes faces.
- `put(file)` → Passes recognized face data forward.

4. IdentityManager

`register()` → Registers with FileSystemManager
- `process()` → Determines identity of individuals from recognition data.

5. FileSystemManager it works like a **central coordinator**
- `onFileDetected(file)` → Detects new/processed files and triggers notifications.



Pipes-and-Filters Architectural Patter

Extra:

Extensibility – Easy to add new filters (e.g., spam filter).

Efficiency – Filters can run in parallel for faster processing.

Reusability – Filters can be reused across systems.

Modifiability – Easy to add/remove/modify filters.

Security – Security filters can be integrated anywhere.

Maintainability – Independent filters make updates simple.

➢ Quality properties of the Pipes-and-Filters architectural pattern include the ones specified below.

Quality	Description
Extensibility	Processing filters can be added easily for more capabilities.
Efficiency	By connecting filters in parallel, concurrency can be achieved to reduce latency in the system.
Reusability	By compartmentalizing pipes and filters, they can both be reused as-is in other systems.
Modifiability	Filters are compartmentalized and independent from each other; therefore, it is easy to add or remove filters to enhance the system.
Security	At any point during data-flow, security components can be injected to the work-flow to provide different types of security mechanisms to the data.
Maintainability	Allows for separation of concerns and independence of the Filters and Pipes; therefore, maintaining existing components becomes easier.

Distributed Systems

➢ Distributed systems are decomposed into multiple processes that (typically) collaborate through the network.

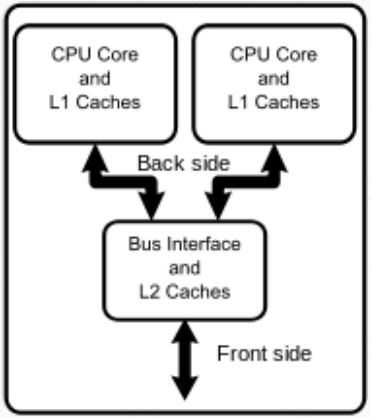
- In some distributed systems, one or more distributed processes perform work on behalf of client users and provide a bridge to some server computer, typically located remotely and performing work delegated to it by the client part of the system. (client-server)
- Other distributed systems may be composed of peer nodes, each with similar capabilities and collaborating together to provide enhanced services, such as music-sharing distributed applications. (peer-to-peer)

Peer-to-peer

client-server

Distributed Systems

- The previous types of distributed systems are easy to spot, since their deployment architecture entails multiple physical nodes.
- However, with the advent of multi-core processors, distributed architectures are also relevant to software that executes on a single node with multiprocessor capability.



Extra:

Distribution can be across many computers or across many cores(CPUs) in one computer.

Distributed Systems

- Some examples of distributed systems include:
  - Internet systems
  - Web services
  - File-or music-sharing systems
  - High-performance systems, etc.

- Common architectural patterns for distributed systems include:
  - Client-Server Pattern
  - Broker Pattern

Extra:

- Client–Server: The system has two roles. Clients request services, and a central server (or servers) provides them. Example: a web browser (client) requesting pages from a web server.
- Peer-to-Peer (P2P): All nodes act as both clients and servers. They share resources and collaborate without relying on a single central server. Example: music/file-sharing apps like BitTorrent.

Client-Server Pattern

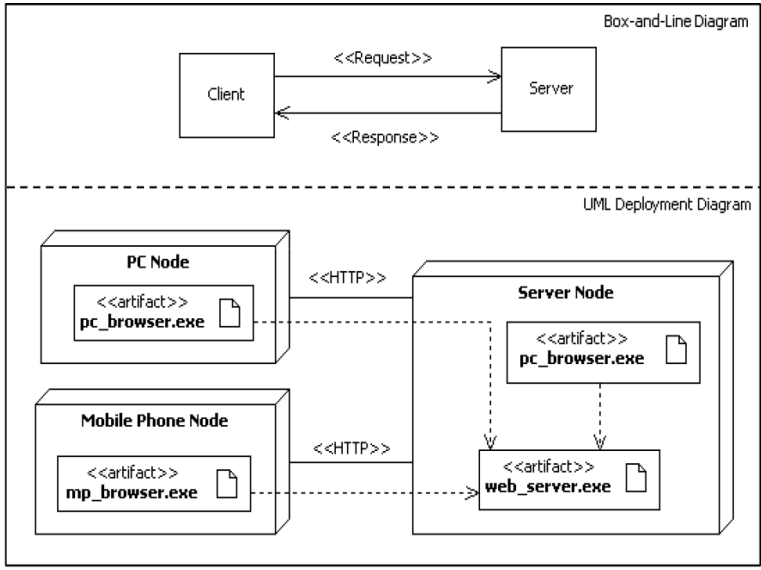
Server: a computer program or a device that provides functionality for other programs or devices, called "clients".

Client: is a piece of computer hardware or software that accesses a service made available by a "server". (Note that multiple clients will communicate with the server. These clients are independent and do not communicate with each other)

Extra:

1. **Bottom part (UML Deployment Diagram):**
   A **deployment diagram** shows how software components (**artifacts**) are deployed on **hardware nodes**.
   PC Node runs `pc_browser.exe` (client application).
   Mobile Phone Node runs `mp_browser.exe` (mobile client app).
   Server Node runs `web_server.exe` (server app) and `pc_browser.exe` (server-side browser process).
2. **Communication:**
   Clients (PC or Mobile) communicate with the server through **HTTP**.
   The server responds back with processed data.

Summary:
This figure shows a **client–server system**: PC and mobile clients send requests via HTTP to a web server. The UML deployment diagram maps **artifacts (programs)** like `browser.exe` and `web_server.exe` onto **nodes (PC, mobile, server)**, making it clear **where software runs and how they communicate**.



Client-Server Pattern

➢ Quality properties of the client-Server architectural pattern include the ones specified below.

Quality	Description
Interoperability	Allows clients on different platforms to interoperate with servers of different platforms.
Modifiability	Allows for centralized changes in the server and quick distribution among many clients.
Availability	By separating server data, multiple server nodes can be connected as backup to increase the server data or services' availability.
Reusability	By separating server from clients, services or data provided by the server can be reused in different applications.

- **Extra:**
- **Interoperability** → Clients on different platforms can work with servers on different platforms.
- **Modifiability** → Server changes are centralized and easily distributed to many clients.
- **Availability** → Backup server nodes increase service and data availability.
- **Reusability** → Server services/data can be reused in different applications.

Client-Server Pattern

➢ Although **Client-Server** Pattern supports the quality attributed described in the previous slide it has the following problems:

- Clients are tightly coupled with servers.
- This leads to complexity for systems that need to provide services from multiple servers hosted at different locations.
- Extra:
  - Client tightly depends(tightly coupled) on a specific server.
  - Must know server's exact IP/URL/protocol.
  - Server changes → client code updates required.
  - **Increases complexity** with multiple servers.
- To overcome this issue the **Broker** pattern was introduced.

Broker Pattern

- The **Broker** architectural pattern provides mechanisms for achieving better flexibility between clients and servers in a distributed environment by acting as an **mediator** between the two components.
- The Broker pattern decreases coupling between client and servers by mediating between them so that one client can transparently access the services of **multiple servers**

Extra: Client-Server versus Broker

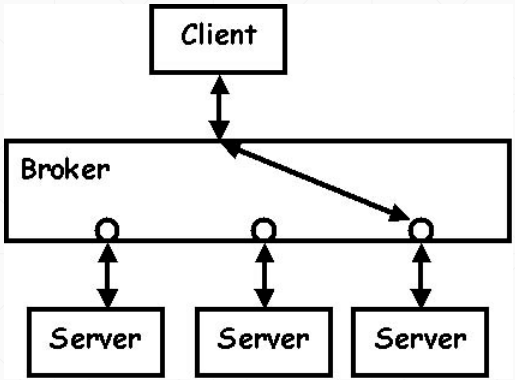
Client-Server

1. Direct communication between client and server.
2. Client must know server's exact address and protocol.
3. Tightly coupled → server changes require client updates.
4. Simpler but less flexible for scaling or multiple servers.

Broker

5. Communication goes through a **broker (middleware)**.
6. Clients don't need to know server details, only the broker.
7. Loosely coupled → server changes don't affect clients.
8. More scalable, flexible, and supports multiple servers/services.

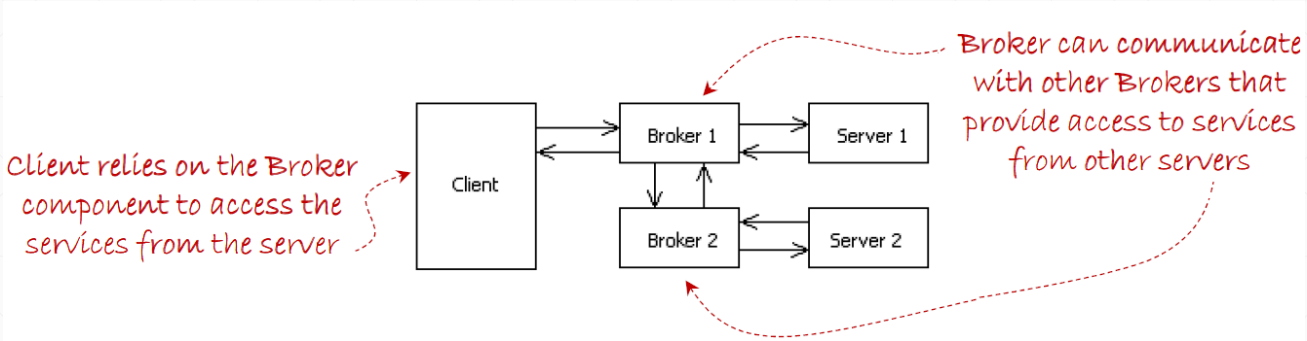
In short: **Client-Server = direct & tightly coupled**, while **Broker = indirect & loosely coupled via a broker**.



Broker Pattern

➢ The **broker** component is responsible for coordinating communication between a client and server(s), such as forwarding requests, as well as for transmitting results and exceptions.

- Leads to systems that are more flexible and interoperable.



Broker Pattern

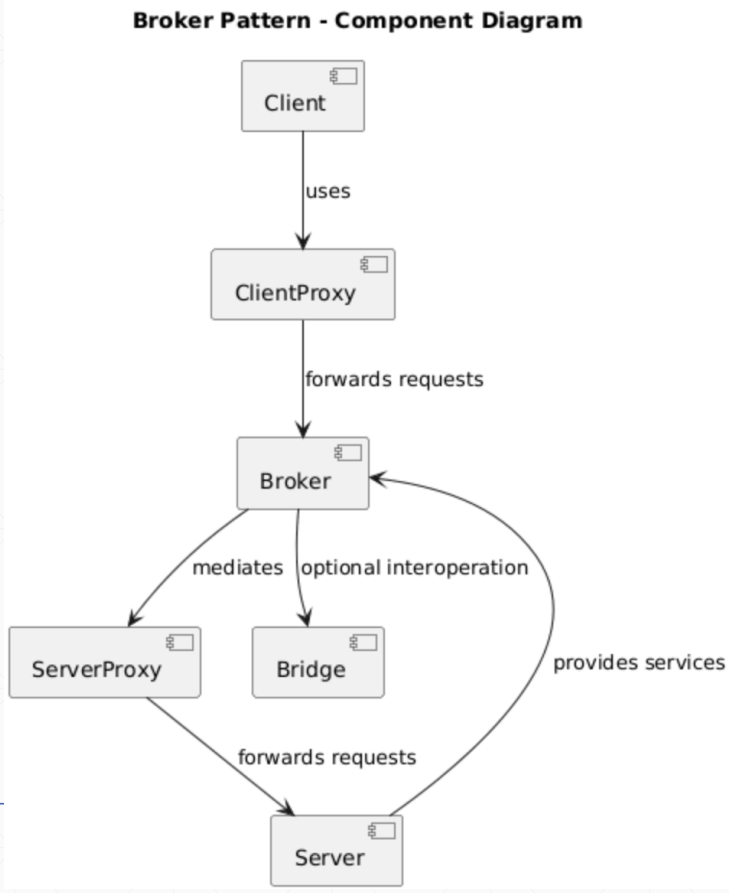
Extra:

- The Broker architectural pattern
- includes the following components:

Component	Description
Client	Applications that use the services provided by one or more servers.
ClientProxy	Component that provides transparency (at client) between remote and local components so that remote components appear as local ones.
Broker	Component that mediates between client and server components.
ServerProxy	Component that provides transparency (at server) between remote and local components so that remote components appear as local ones.
Server	Provide services to clients. May also act as client to the Broker.
Bridge	Optional component for encapsulating interoperation among Brokers.

•**Extra:**

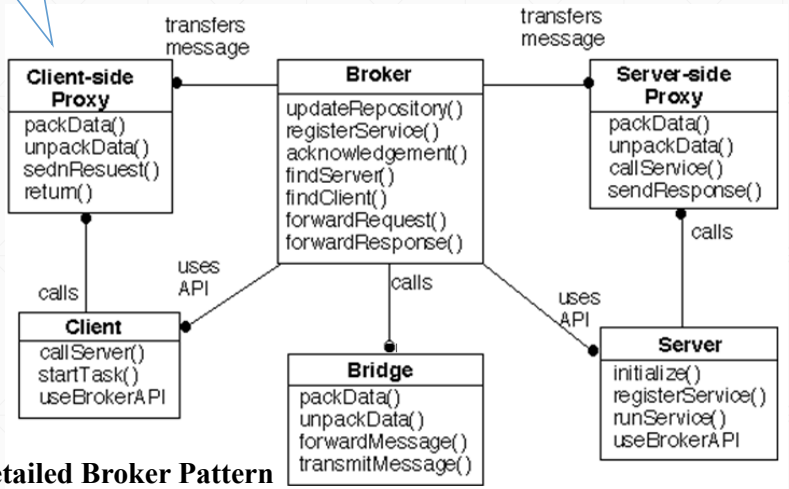
- **Client**
  - **Provided Interface:** None
  - **Required Interface:** Services from *ClientProxy*
- **ClientProxy**
  - **Provided Interface:** Local service to *Client*
  - **Required Interface:** *Broker*
- **Broker**
  - **Provided Interface:** Mediation service to *ClientProxy*
  - **Required Interface:** *ServerProxy*
- **ServerProxy**
  - **Provided Interface:** Local service to *Broker*
  - **Required Interface:** *Server*
- **Server**
  - **Provided Interface:** Real application services (exposed through *Broker/ServerProxy* to clients)
  - **Required Interface:** *Broker* (if the server acts as a client to another service)
- **Bridge**
  - **Provided Interface:** Interoperability between different *Brokers*



The client side proxy decides if the request can be resolved locally or to forward the request to the broker

Broker Pattern

- Separates system communication functionality from the main application functionality by providing a broker that isolates communication-related concerns.



Detailed Broker Pattern

Broker Pattern

➢ Quality properties of the Broker architectural pattern include the ones specified below.

Quality	Description
Interoperability	Allows clients on different platforms to interoperate with servers of different platforms. Also, allows clients to interoperate (transparently) with multiple servers.
Modifiability	Allows for centralized changes in the server and quick distribution among many clients.
Portability	By porting the broker to different platforms, services provided by the system can be easily acquired by new clients in different platforms.
Reusability	Brokers abstract many system calls required for providing communication between nodes. When using brokers, many complex services can be reused in other applications that require similar distributed operations.

Extra:

- **Interoperability** → Clients can work with servers across different platforms and multiple servers.
- **Modifiability** → Centralized server changes can be quickly distributed to many clients.
- **Portability** → Broker can be ported to different platforms, making services easily accessible to new clients.
- **Reusability** → Brokers handle system calls, enabling reuse of complex services in other applications.

Summary

- Data-Centered Systems
  - Blackboard pattern
- Data Flow Systems
  - Pipes-and-Filters pattern
- Distributed Systems
  - Client Server pattern
  - Broker pattern
- What's next…
  - Interactive Systems
  - Hierarchical Systems