

OVERVIEW **Data-Centered, Data-Flow and Distributed Patterns: Blackboard, Pipes-and-Filters, Client-Server, Broker**

Four patterns for three system types. Data-centered systems revolve around shared data: in the blackboard pattern, independent knowledge sources (agents) improve a shared solution on the blackboard while a controller decides who works next. Data-flow systems move and transform a stream: pipes-and-filters chains independent filters with pipes from a data source to a data sink. Distributed systems spread work over the network: client-server is direct and tightly coupled, and the broker pattern puts a mediator (with client- and server-side proxies) between them to loosen that coupling.

Blackboard

Blackboard (shared data) + agents/knowledge sources (independent specialists) + controller (schedules them). Opportunistic: no fixed order of steps.

Pipes and filters

Data source → filter → pipe → filter → ... → data sink. Filters are independent, so they can be added, reordered, reused and run in parallel.

Client-server

Clients request, a central server provides. Interoperable and easy to update centrally, but clients are tightly coupled to the server.

Broker

Client → client proxy → broker → server proxy → server. Clients need only the broker, so servers can move or change.

HOW THE LECTURE IS ORGANIZED

Slides 2–5 Data-centered systems: data manager, workers, integrity

Slides 6–23 Blackboard: analogy, ER example, components, scheduling-system diagrams, speech-recognition exercise

Slides 24–36 Data flow and pipes-and-filters: compiler and face-recognition examples, qualities

Slides 37–42 Distributed systems and client-server

Slides 43–48 Broker pattern and summary

EXAM FOCUS

Every pattern here has been examined. Midterm 252 Q5 (6 marks) was a whole blackboard problem (name the pattern, give each component's role, build the component diagram, write an availability scenario) and Q6 (2 marks) a pipe-and-filter sequence diagram for the face-recognition system on slides 31–35. Final 252 Q4-2 asked for the pipe-and-filter diagram of a video-processing app, and Midterm 261 Q6 matched scenarios to blackboard, pipe-and-filter, broker and client-server.

HOW TO READ THIS PAGE

Every original slide is shown unchanged. **Yellow marks** on a slide point to a numbered note: hover or tap a mark to jump to its note, and hover a note to see its marks. Questions tagged **Past exam** appeared on a real paper; **Prediction** questions are likely angles, not confirmed.



Software Design and Architecture

[Architecture Patterns] – Chapter 04, L02

NOTES

- 1 Second patterns lecture: the patterns for the first three system types of Lecture 1. For each one learn its components (the roles you will be asked to name), how they communicate, and its quality properties.

SLIDE 2 / 48 Lecture Outlines- Architecture Patterns & Chapter 04

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

NOTES

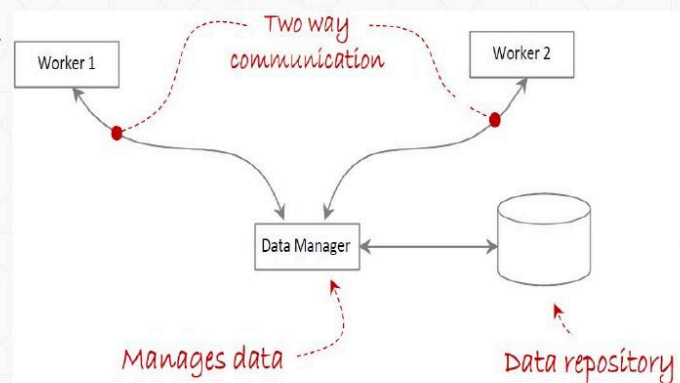
- 1 **Data-centered** → the blackboard pattern.
- 2 **Data flow** → pipes and filters.
- 3 **Distributed** → client-server and broker. Interactive and hierarchical systems come in Lecture 3.

SLIDE 3 / 48 Data-Centered Systems Overview

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.

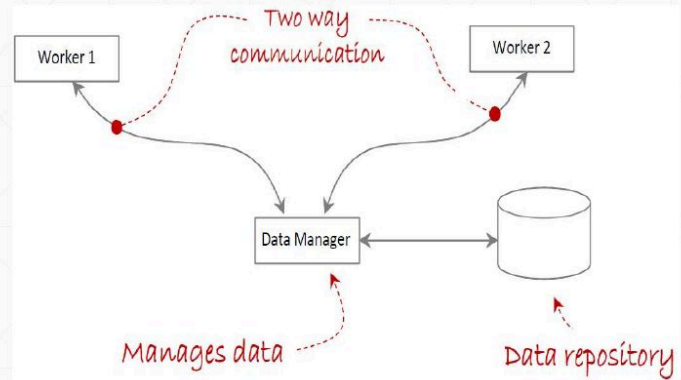


NOTES

- 1 A **data-centered** system is decomposed around one **central repository** of data.
- 2 The **data management component** (the "Data Manager" in the diagram) controls, provides and manages access to the system's data.
- 3 **Worker components** do the actual work, operating on that 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.



NOTES

- 1 Communication is **one-to-one and two-way** between each worker and the data manager (the "two way communication" arrows).
- 2 Workers **do not interact with each other** directly; everything goes through the data management component. That keeps workers independent, but makes the data manager critical: if it is slow or down, everyone is.

Data-Centered Systems – Overview

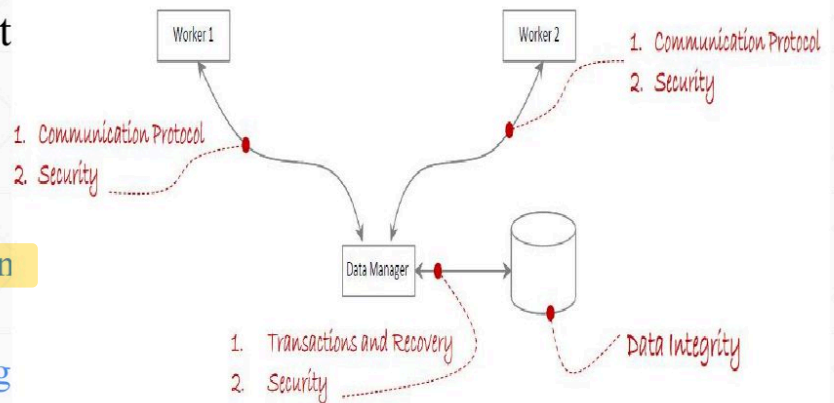
➤ Because of the architecture of these systems, they must consider issues with:

- **Data integrity**

– **Extra:** Integrity means that data is **complete, accurate, and consistent** across various databases, systems, or during any operation (like transfer, storage, or retrieval).

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.



between worker and data

management

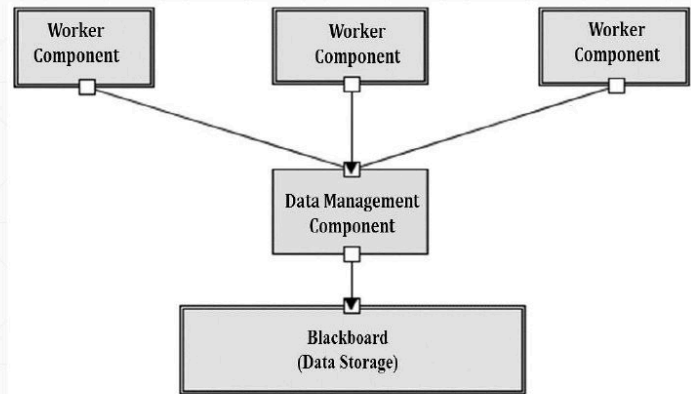
NOTES

- 1 Because everything depends on shared data, these systems must handle **data integrity**: data stays complete, accurate and consistent across operations, even when several workers update it (the diagram adds transactions and recovery, and security, at the data manager).
- 2 They must also define the **communication protocols** (and security) between the workers and the data manager. The Extra box repeats Lecture 1: a DBMS can be seen as a repository or as the server in client-server.

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.



Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

6

NOTES

- 1 The **blackboard** pattern decomposes a system into components that work around a **central data component** to solve complex problems.
- 2 The components work **independently**, each providing a **partial solution**, in an **opportunistic** way: whoever can contribute something useful given the current state does so.
- 3 There is **no predetermined sequence** of operations that leads to the answer. That is the key difference from pipes and filters, whose order is fixed.

EXAM ANGLE

Past exam · Midterm 252 · Q5a · 1 mark

Q1

Best Flight Itinerary system (Midterm 252): a Passenger App sends a trip request to the Trip Planning Service, which creates an empty itinerary draft in the Draft Itinerary Record. The Route Builder lists possible routes, the Constraint Checker filters out routes that break the passenger's or airline's rules, and the Best Option Selector picks the best one; each step retrieves the current draft, improves it and saves it back, until the draft is stable and the service returns it to the app. Write the name of the most suitable architectural pattern for this system.

Answer: Blackboard. Several independent components (Route Builder, Constraint Checker, Best Option Selector) repeatedly read and update one shared data store (the Draft Itinerary Record) until the solution is stable, and a controller (the Trip Planning Service) decides which one runs next. Pipe-and-filter is the tempting wrong answer: there, filters pass output directly to the next filter and no shared store is repeatedly updated.

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.

NOTES

- 1 The pattern copies how a **group of scientists** would attack a hard problem together at one blackboard.
- 2 A **mediator (controller)** controls access to the board. When it hands control to a scientist, that scientist looks at the current state and, if possible, advances the solution before releasing the board.
- 3 With the new knowledge on the board, control goes to the next scientist who can improve it, and so on until **no more progress** can be made. There is no fixed order: each contributes when the current state makes their expertise useful.
- 4 This is how **expert systems** work (AI that applies knowledge and inference rules like a human expert, in medicine or finance), so the blackboard pattern is a good fit for their logical architecture.

➤ **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.

NOTES

- 1 **Blackboard:** the shared whiteboard in a group project, where all information is displayed and updated and everyone can see it.
- 2 **Agents:** independent contributors with different skills who read the board and add their expertise.
- 3 **Controller:** the organiser who manages when and how agents access the board, so the project progresses in an orderly way.

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:

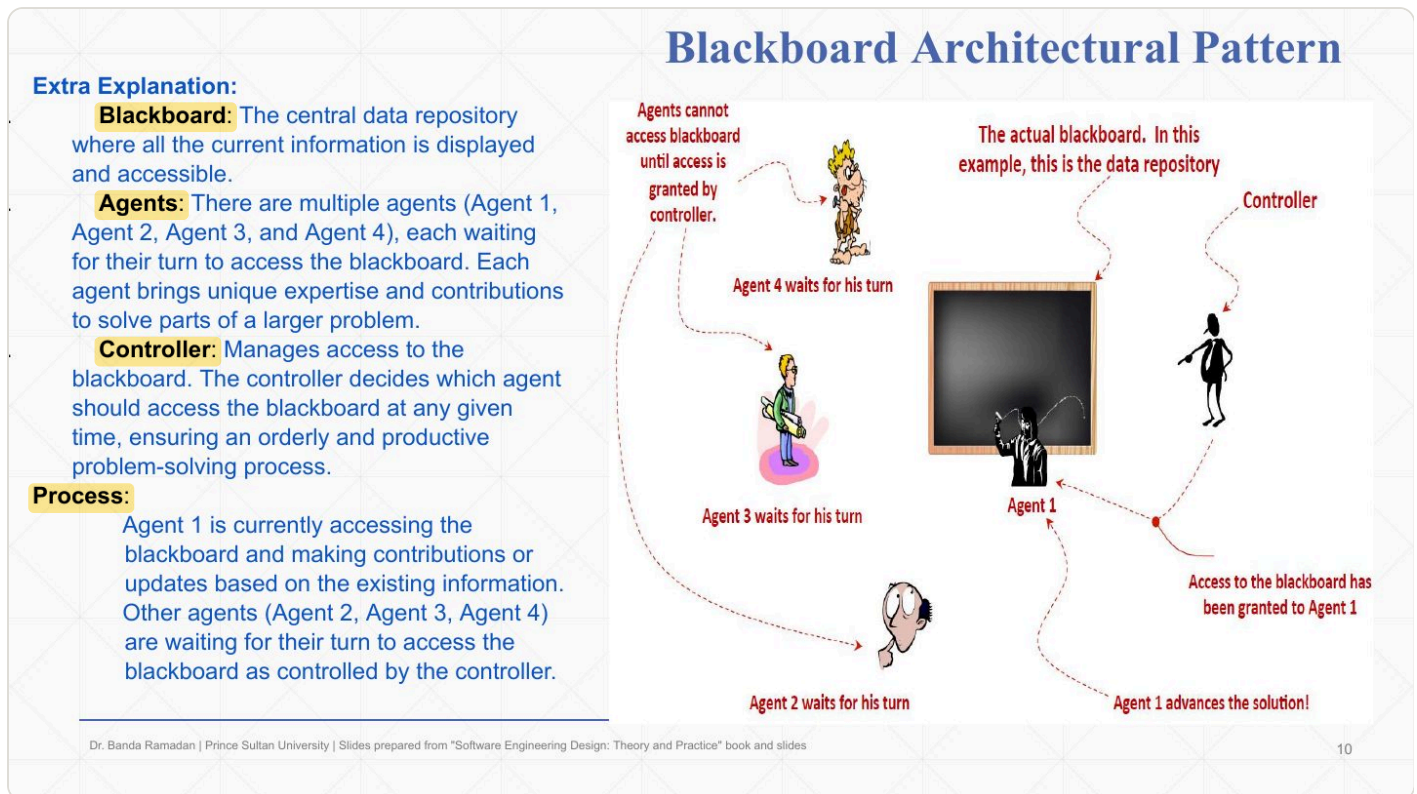
it will manage when and which agents have access to the blackboard (or patient chart) and may prioritize or schedule their contributions based on the patient's changing condition and the updates on the chart.



9

NOTES

- 1 An emergency room maps onto the pattern exactly.
- 2 **Blackboard:** the patient's chart, visible to all staff and continuously updated.
- 3 **Agents:** doctors, nurses and specialists. A nurse updates vital signs, a doctor orders tests or medication, a specialist adds an opinion. Each reads the chart and contributes from their own expertise.
- 4 **Controller:** decides who gets access to the chart and when, prioritising contributions as the patient's condition changes.

**NOTES**

- 1 **Blackboard:** the central data repository where all current information is displayed and accessible.
- 2 **Agents** (knowledge sources): Agent 1–4, each with unique expertise for part of the problem. In the cartoon, agents cannot touch the board until the controller grants access.
- 3 **Controller:** decides which agent accesses the board at any time, keeping problem solving orderly.
- 4 **Process:** Agent 1 currently has access and advances the solution; Agents 2–4 wait their turn.

EXAM ANGLE

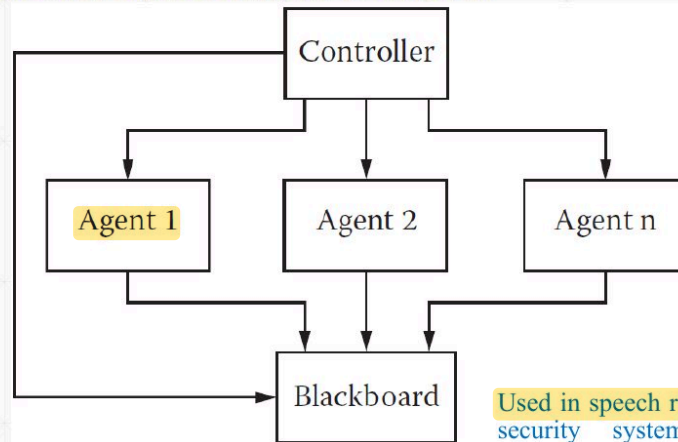
Past exam · Midterm 252 · Q5b · 1 mark

Q1

Best Flight Itinerary system (Midterm 252): a Passenger App sends a trip request to the Trip Planning Service, which creates an empty itinerary draft in the Draft Itinerary Record. The Route Builder lists possible routes, the Constraint Checker filters out routes that break the passenger's or airline's rules, and the Best Option Selector picks the best one; each step retrieves the current draft, improves it and saves it back, until the draft is stable and the service returns it to the app. Based on the pattern you selected, identify the role of each component.

Answer: **Passenger App** → client (sends the trip request, shows the result). **Trip Planning Service** → controller (manages the process and decides which component runs next). **Draft Itinerary Record** → blackboard (the shared data everyone reads and updates). **Route Builder, Constraint Checker, Best Option Selector** → workers / knowledge sources.

Blackboard Architectural Pattern



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

NOTES

- 1 Typical uses: speech recognition, image recognition, security systems, business resource management. All are problems where several kinds of knowledge must be combined with no fixed order.
- 2 The structure diagram: the **controller** at the top connects to every agent (to invoke them) and to the **blackboard**; Agents 1...n each connect to the blackboard to read and write. Agents never connect to each other.

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.

- 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.
- Agents**
Specialized problem solvers (e.g., StudentHistory, WorkSchedule, CourseOfferings).
They watch the Blackboard to see if they can contribute.
- 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)

NOTES

- Example: a students' scheduling system that builds a schedule from completed courses, the student's work schedule, and the courses offered.
- Blackboard** (ScheduleBlackboard): the shared knowledge base that controller and agents read and write.
- Agents** (StudentHistory, WorkSchedule, CourseOfferings): specialised problem solvers that contribute when they can.
- Controller** (ScheduleManager): decides when and which agent acts, and the order of problem solving.
- Interfaces in the component diagram: the **blackboard provides** an interface (IStudentHistory, IWorkSchedule... read/write/update) because both the controller and the agents need its data. The controller provides nothing to the agents: it orchestrates, it doesn't store data.
- The **agents provide** an interface (e.g. `run()`, `workOnSchedule()`) that the controller requires, so the controller can invoke them when it decides they should act.

Best Flight Itinerary system (Midterm 252): a Passenger App sends a trip request to the Trip Planning Service, which creates an empty itinerary draft in the Draft Itinerary Record. The Route Builder lists possible routes, the Constraint Checker filters out routes that break the passenger's or airline's rules, and the Best Option Selector picks the best one; each step retrieves the current draft, improves it and saves it back, until the draft is stable and the service returns it to the app. Build the component diagram with the correct provided and required interfaces.

Answer: Trip Planning Service **provides** ITripPlanningAPI and the Passenger App **requires** it (the client only talks to the controller). Draft Itinerary Record **provides** IItineraryStore; the Trip Planning Service and each worker (Route Builder, Constraint Checker, Best Option Selector) **require** it. The Trip Planning Service invokes the three workers. Wrong connections: Route Builder passing routes directly to the Constraint Checker (that is pipe-and-filter), the Passenger App using the store directly, the blackboard requiring an interface from the controller, a worker providing the API to the client.

SLIDE 13 / 48 Extra: Blackboard roles and interfaces

Extra: Continue

• Who does what (one line each)

- o **Blackboard:** shared data store; exposes an API to read/write/update facts.
- o **Agents:** specialized solvers; when invoked, they **use the Blackboard API** to read inputs and write results.
- o **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**.

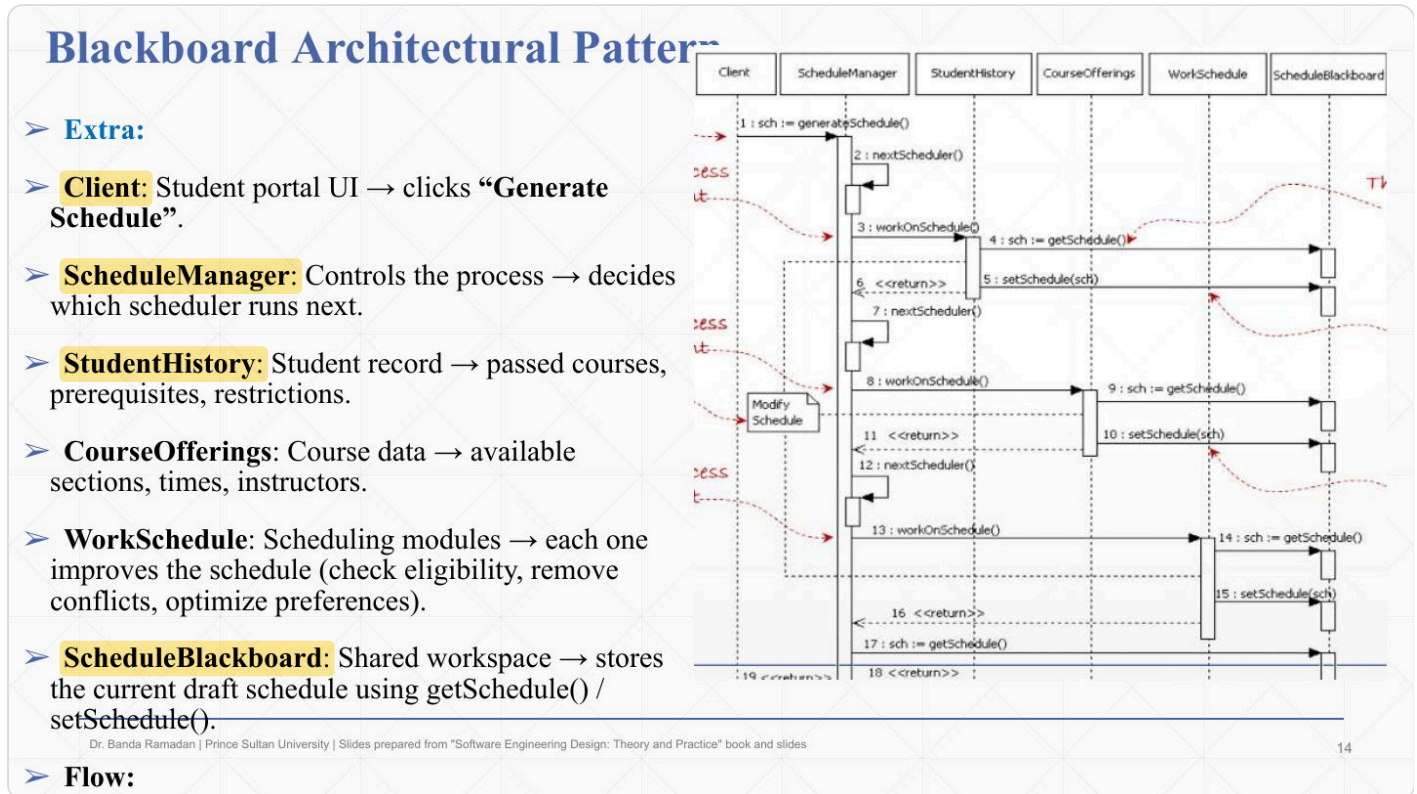
So: your "agents can't access the blackboard" interpretation is the only incorrect piece—agents **should** access it, but **only through the Blackboard's interface**, and **only when the Controller schedules them**.

NOTES

- 1 **Who does what:** the blackboard is the shared store with a read/write/update API; agents are specialised solvers that use that API when invoked; the controller (scheduler) orchestrates when and which agent runs, and does not pass data around.
- 2 **One cycle:** (1) the controller decides "Agent A runs now" and calls AgentA.run() ; (2) A reads what it needs from the blackboard; (3) A computes and writes its results back; (4) the controller checks the board (or is notified) and picks the next agent.

- 3 **Interfaces:** the blackboard provides the data API (agents and controller require it); agents provide a service interface such as `run()` (the controller requires it); the controller generally provides nothing to agents.
- 4 **Guideline:** control flows controller → agents (invocation); data flows agents ↔ blackboard (read/write). Agents never talk to each other; they communicate only through the blackboard, and only when scheduled.

SLIDE 14 / 48 Blackboard: scheduling system sequence diagram



NOTES

- 1 The sequence diagram for the scheduling system. **Client:** the student portal; the student clicks “Generate Schedule”, which calls `generateSchedule()`.
- 2 **SchedulerManager** (controller) runs the process and decides which scheduler goes next (`nextScheduler()`).
- 3 **StudentHistory, CourseOfferings, WorkSchedule:** the agents. Each improves the draft: check eligibility from passed courses, keep only sections actually offered, remove conflicts with work hours.
- 4 **ScheduleBlackboard** holds the current draft, accessed through `getSchedule()` and `setSchedule()`. Slides 15–17 walk through the 19 messages.

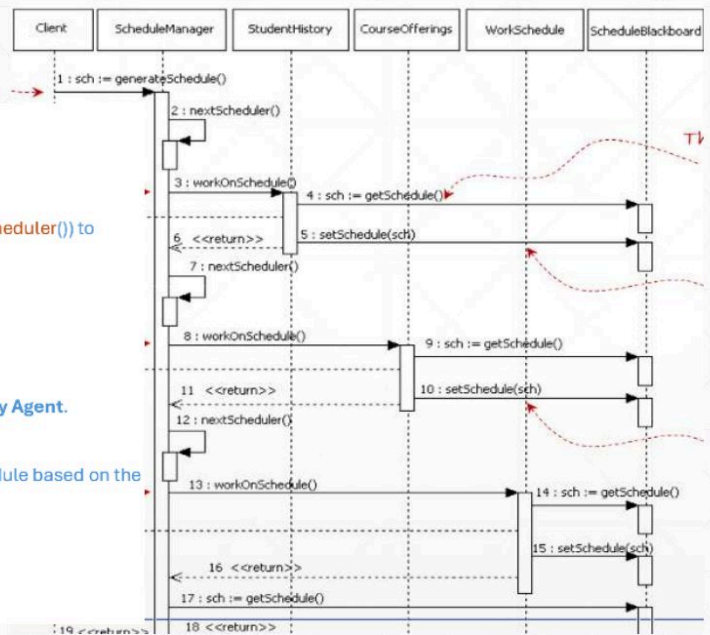
Extra Explanation:

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**.



15

NOTES

- 1 Messages 1–2: the client calls `generateSchedule()` on the **ScheduleManager**, which calls `nextScheduler()` to choose the first agent.
- 2 **Phase 1** (messages 3–6): the manager calls `workOnScheduler()` on **StudentHistory**; **StudentHistory** reads the draft with `getScheduled()`, adjusts it for past courses and degree requirements, saves it with `setSchedule(sch)`, and returns control to the manager.

Phase 2: CourseOfferings Agent Works

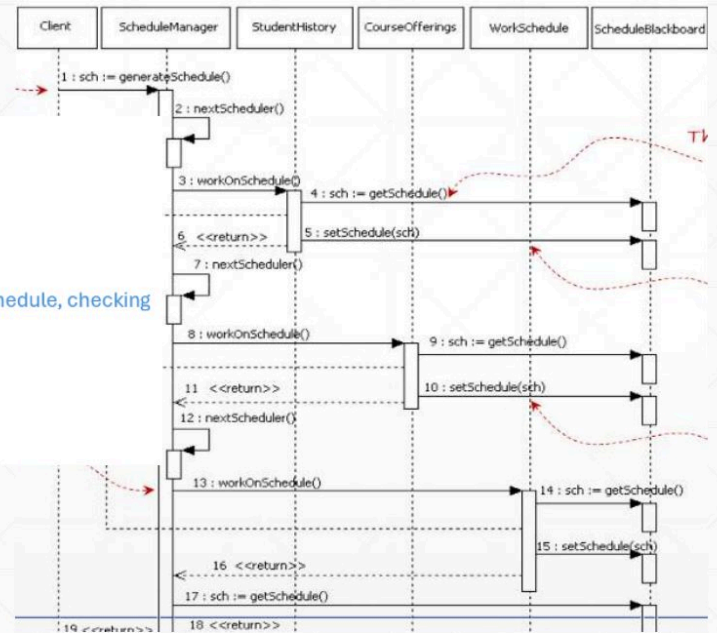
7-ScheduleManager gets the next agent (nextScheduler())

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.

**NOTES**

- Phase 2** (messages 7–11): the manager picks the next agent (nextScheduler()) and calls workOnScheduler() on CourseOfferings. It reads the draft, keeps only courses actually offered this semester, saves the realistic version back, and returns control. Note the pattern repeating: manager invokes → agent reads → agent writes → control returns.

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).

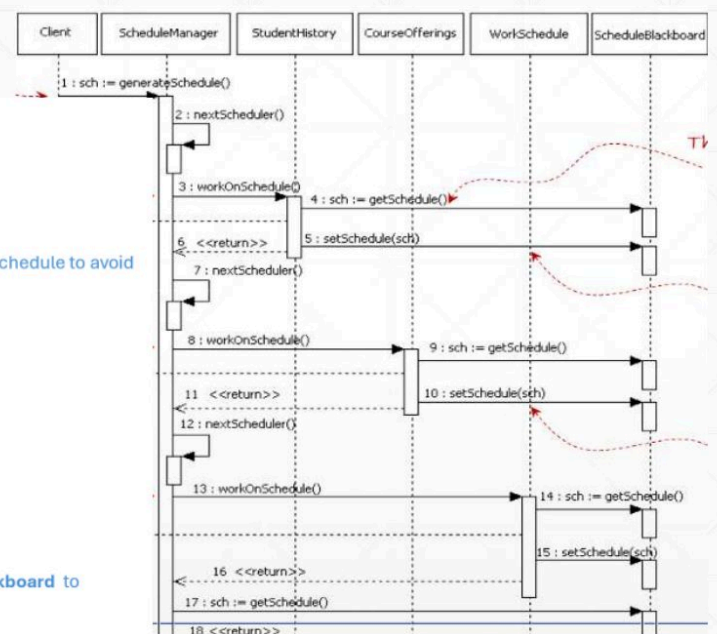
16-It returns control to the ScheduleManager.

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.



NOTES

- 1 **Phase 3** (messages 12–16): WorkSchedule reads the draft, removes conflicts with the student's work commitments, and saves the final conflict-free version.
- 2 **Phase 4** (messages 17–19): the manager retrieves the finished schedule from the blackboard with one last `getSchedule()` and returns it to the client.

SLIDE 18 / 48 Blackboard Architectural Pattern

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.

18

NOTES

- 1 **Modifiability:** agents are independent, so adding or removing one is easy, e.g. a new SportsSchedule agent without touching the others.
- 2 **Reusability:** specialised agents can be reused elsewhere, e.g. StudentHistory in a graduation checker.
- 3 **Maintainability:** knowledge is separated, so fixing a bug in CourseOfferings doesn't touch WorkSchedule. (The cost: the controller's scheduling makes behaviour harder to predict and test.)

EXAM ANGLE

Prediction

Q1

Give two quality properties of the blackboard pattern with a short justification.

Answer: Modifiability: agents are independent, so new ones can be added without changing others. Reusability: specialised agents can be reused in other systems. Maintainability: knowledge sources are separated, so one can be fixed without touching the rest.

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.

20

NOTES

- 1 A **deployment diagram** shows the physical deployment of artifacts (programs, libraries) onto hardware: servers, devices, run-time nodes and the communication links between them. Web-app example: how the web server, application server and database server are configured and connected.
- 2 A **component diagram** shows how software components are organised and depend on each other through interfaces. Web-app example: the authentication library, data-access objects and business logic, and how they interact. Component = logical structure; deployment = where it runs.

EXAM ANGLE**Prediction****Q1**

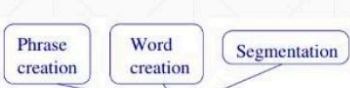
Which diagram shows hardware nodes and the software deployed on them?

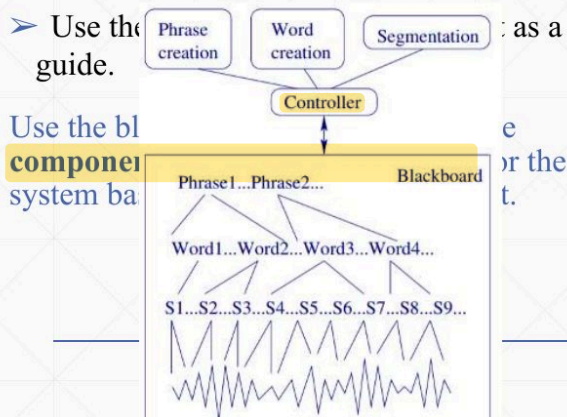
- A. Deployment diagram
- B. Component diagram
- C. Class diagram
- D. Sequence diagram

Answer: A. Deployment diagram

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.

[illegible]

Speech Recognition System

NOTES

- 1 The exercise: a speech-recognition system. Input is speech as a waveform (single words or whole sentences); output is English text on the screen.
- 2 Using the structure on the left (Segmentation, Word creation and Phrase creation agents around a controller and a blackboard holding segments S1...S9, words and phrases), draw the **component and sequence diagrams** with the blackboard pattern.
- 3 (The table on the right is unreadable in the PDF because of a font problem; slides 21–23 give its content and the solution.) The **controller** coordinates the agents; the **blackboard** is the shared memory where segments, words and phrases are stored and refined.

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 lessons."*

➤ 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, transforming education]

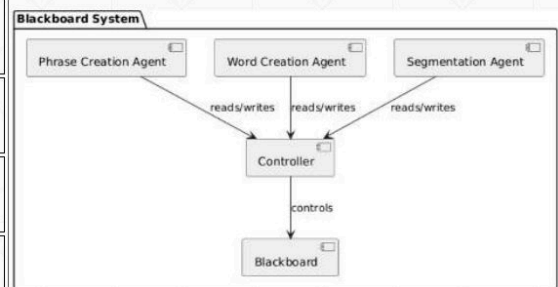
Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides.

22

NOTES

- 1 A walk-through on sample text. **Segmentation agent:** splits the input into sentences ("Artificial intelligence is transforming education." ...).
- 2 The **word creation agent** (between them) builds words from segments; the **phrase creation agent** groups words into grammatically correct phrases: NP "Artificial intelligence", VP "is transforming", NP "education".
- 3 The **final structured output** on the blackboard holds the sentence, its words and its phrases, each written by a different agent.

Extra: Solution – Simple Component diagram without Interfaces

[illegible]

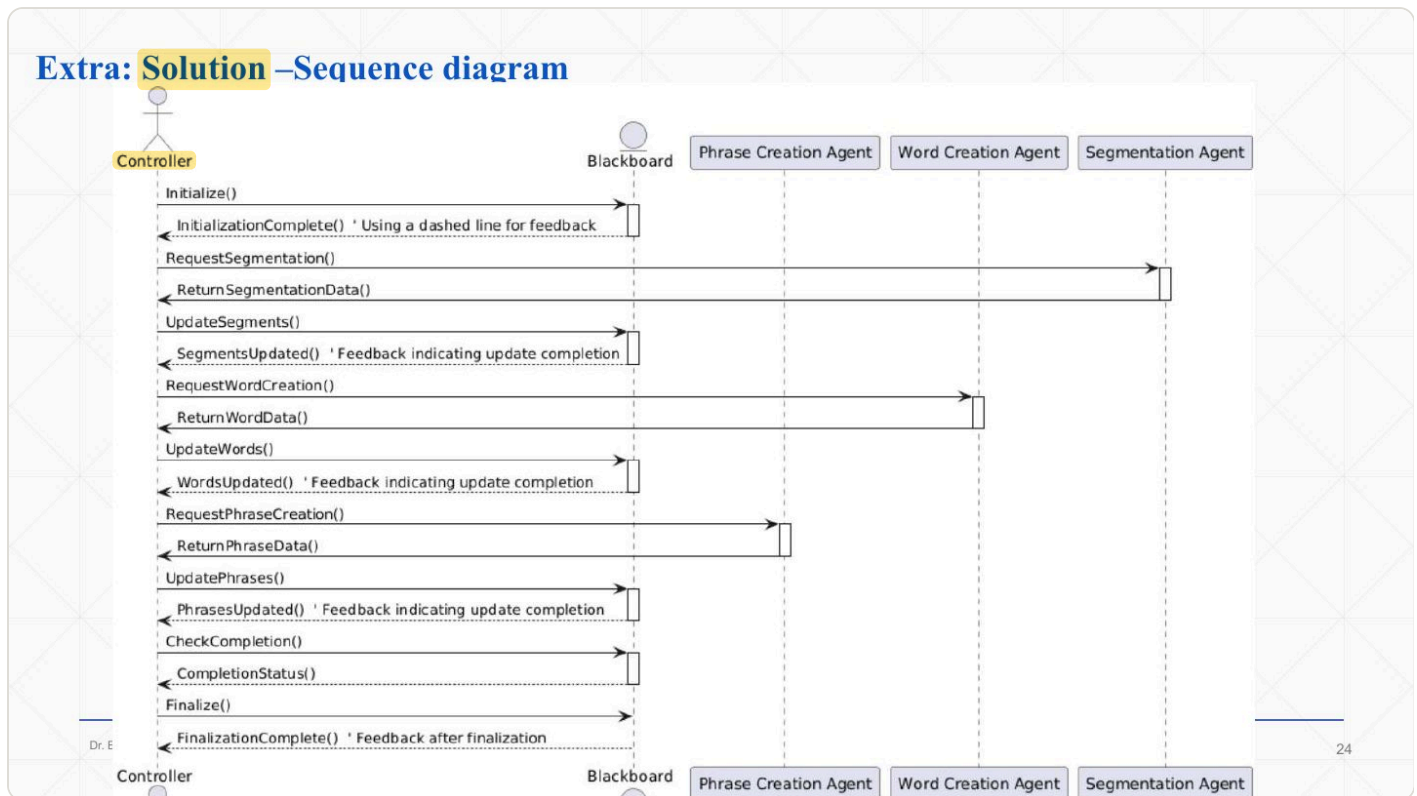
Note

- 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**.

23

NOTES

- 1 The component solution: Segmentation, Word Creation and Phrase Creation agents each require the blackboard (reads/writes) and provide their own operation; the controller requires the agents and the blackboard and orchestrates them; the blackboard provides read/write and event services. (The table is garbled in the PDF; the diagram on the right shows the same structure.)
- 2 The note answers a common doubt: give the controller a **provided interface** only if it is **active** (offers services such as scheduling or policies to others); if it just manages agents and the blackboard internally, it has none.

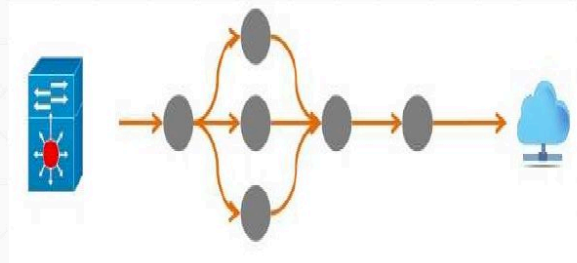


NOTES

- 1 The sequence solution, top to bottom: the controller initialises the blackboard, then repeats the same cycle for each agent: **request** (segmentation, word creation, phrase creation) → the agent returns its data → the controller **updates** the blackboard → the blackboard confirms (dashed feedback arrows).
- 2 It ends with `CheckCompletion()` / `CompletionStatus()` and `Finalize()`. In this solution the controller writes to the blackboard on the agents' behalf; in the scheduling example the agents wrote themselves. Both are accepted variants: what matters is that agents never call each other.

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

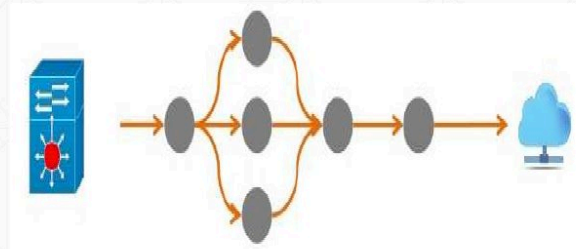


NOTES

- 1 **Data flow** systems are decomposed around **transporting** data streams and **transforming** them on the way to meet application needs.
- 2 **Worker components** perform work on the data.
- 3 **Transport components** move the 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

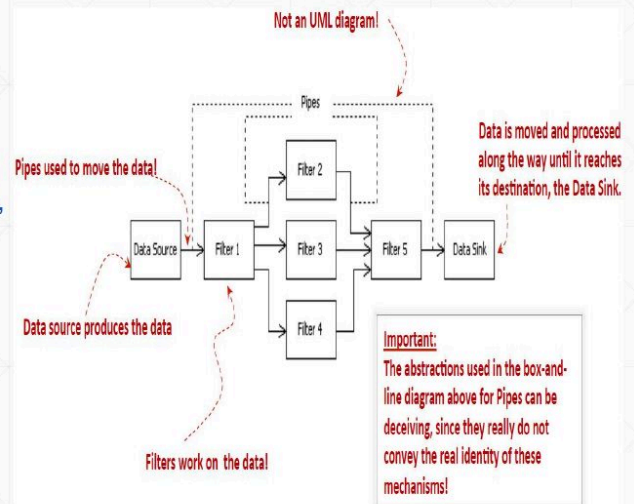
NOTES

- 1 Worker components abstract the transformations applied before data is forwarded: encryption/de-encryption, compression/decompression, changing format (binary to XML, raw data to information), enhancing, modifying or storing data.
- 2 Transport components abstract how data is moved and controlled. The data-flow pattern studied here is **pipes and filters**: filters are the workers, pipes the transport.

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.



27

NOTES

- 1 **Data source:** produces the input data.
- 2 **Pipes:** carry data from one step to the next.
- 3 **Filters:** process the data (transform, clean, analyse) and pass it on. In the diagram data branches: Filter 1 feeds Filters 2, 3 and 4, which rejoin at Filter 5.
- 4 **Data sink:** the final destination where the processed data is stored or used.
- 5 **Important:** the box-and-line diagram is not UML and hides how pipes really work; in real systems pipes can be function calls, files, queues or sockets.

EXAM ANGLE

Past exam · Final 252 · Q4-2 · 2.5 marks

Q1

StreamFlow: a desktop app for broadcasters processes live video in real time. Frames are modified as they pass through the application before being sent to the streaming server, through transformations in a specific order: Noise Reduction, Color Grading, Watermarking and Compression. Identify the architecture pattern and draw its component diagram.

Answer: Pipe-and-filter. Each transformation is an independent filter; frames flow between them through pipes in a fixed order, so filters can be replaced or reordered without touching the others: Live feed (source) → Noise Reduction → Color Grading → Watermarking → Compression → Streaming server (sink).

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

NOTES

- 1 The four components, formally. **Data source**: produces the data.
- 2 **Filter**: processes, enhances or modifies the data.
- 3 **Pipe**: connects source → filter, filter → filter, and filter → sink.
- 4 **Data sink**: consumes the data.

EXAM ANGLE

Prediction

Q1

In pipes and filters, which component transforms the data?

- A. Filter
- B. Pipe
- C. Data sink
- D. Data source

Answer: A. Filter

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

NOTES

- 1 Example 1: a **language processor** (compiler or interpreter) is the classic pipes-and-filters system: source code flows through lexical analysis, parsing, code generation and optimisation.

SLIDE 29 / 48 Pipes-and-Filters – Example 1: language processor

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:

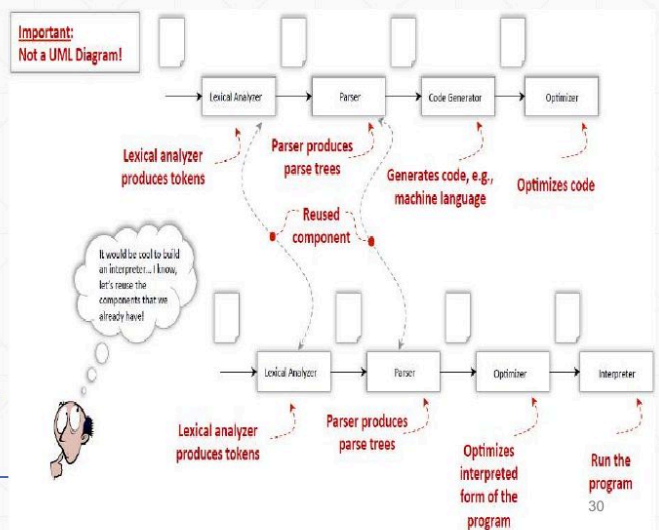
- **Function:** Makes the code run faster and use fewer resources.
 - **Example:** For the code `int x = 5;`, the optimizer might simplify memory operations so more efficient instruction that does both at once, reducing the overall steps the machine must perform.
- instead of:**
 Step 1: Reserve memory 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`.

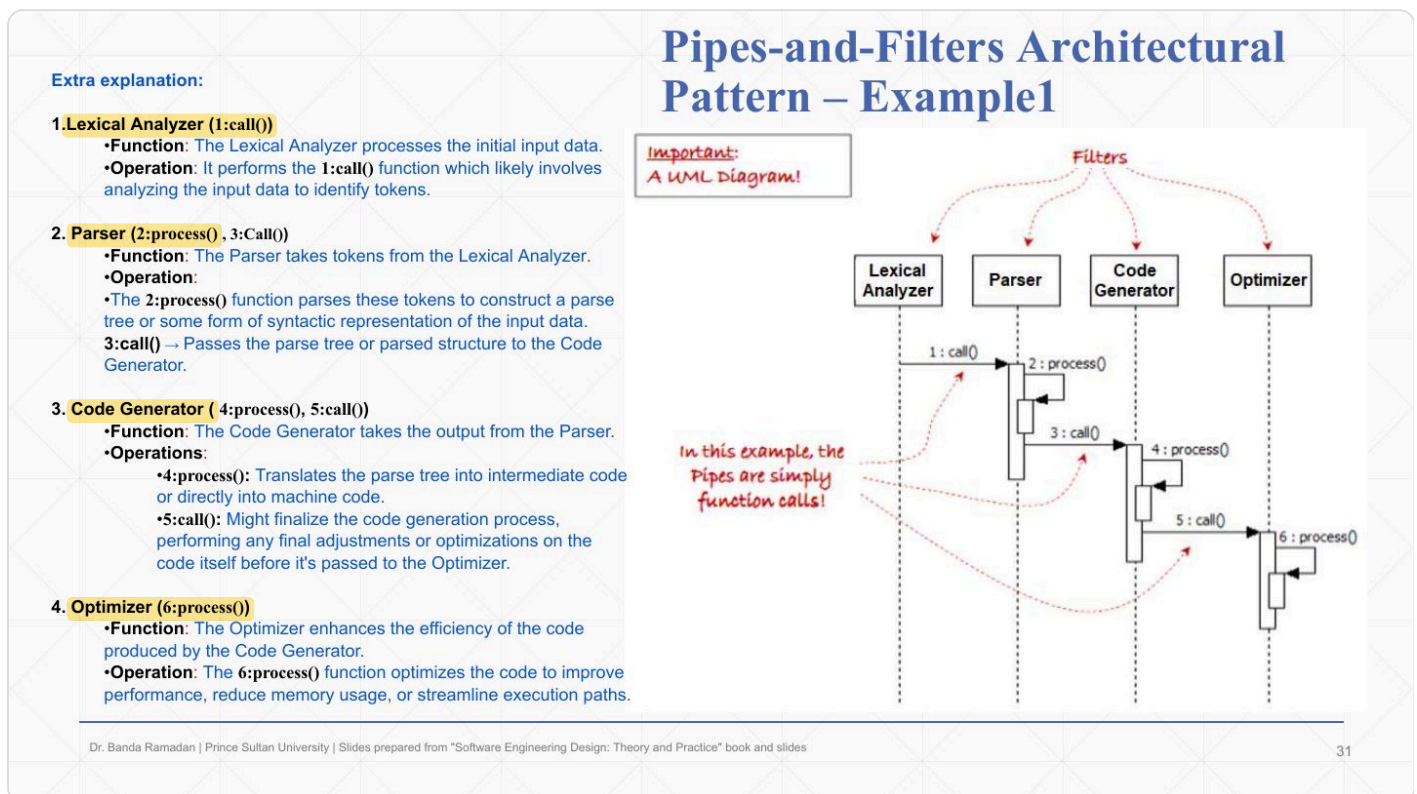
Pipes-and-Filters Architectural Pattern – Example1



NOTES

- 1 Lexical analyser** (filter 1): breaks source code into tokens. `int x = 5;` becomes `int`, `x`, `=`, `5`, `;`.
- 2 Parser**: builds a parse tree from the tokens (a declaration node for `int x`, an initialisation node for `= 5`). **Code generator**: turns the tree into machine code or an intermediate form (allocate memory for `x`, move 5 into it).
- 3 Optimizer**: makes the code faster or smaller (e.g. one instruction that both reserves and initialises `x`). In an **interpreter** pipeline the last filter executes the intermediate form directly instead. The diagram shows that filters are reusable: the same lexer and parser serve both pipelines.

SLIDE 30 / 48 Pipes-and-Filters – Example 1 as UML



NOTES

- The same pipeline as a UML sequence diagram ("Important: a UML diagram!"). **1: call()** feeds input to the lexical analyser.
- The parser **2: process()**es the tokens into a parse tree and **3: call()**s the code generator.
- The code generator **4: process()**es the tree into code and **5: call()**s the optimizer.
- The optimizer **6: process()**es the code. Here the pipes are simply function calls between filters.

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**

Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

32

NOTES

- Example 2: software that identifies people automatically. In order: it **accesses videos** (with audio) from YouTube, **detects faces** (is there a face?), **recognises faces** (whose face is it?), and **predicts the identity** of the individuals.
- Each step consumes the previous step's output, which makes it a pipes-and-filters system; the next slides model it.

EXAM ANGLE

Past exam · Midterm 252 · Q6 · 2 marks

Q1

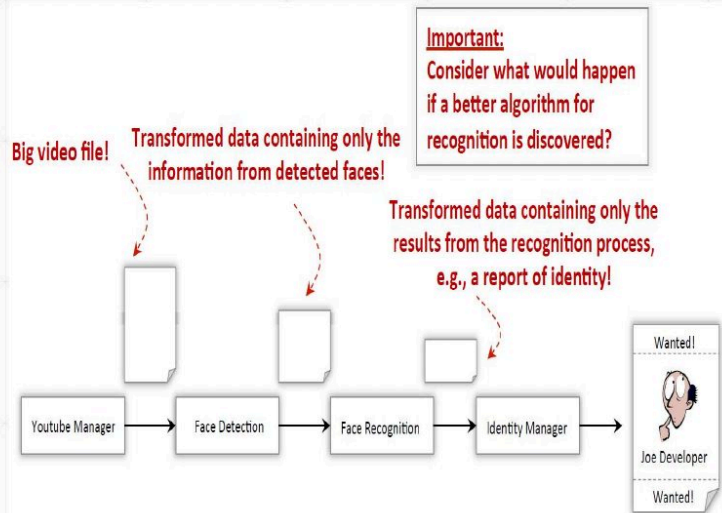
A piece of software works out who a person is automatically with a pipe-and-filter architecture: it accesses videos (with audio) from YouTube, detects faces, recognizes faces, and predicts the identity. (1) Place the lifelines of the sequence diagram from left to right in the order the data flows. (2) Which statements about the correct diagram are true?

Answer: Order: access videos → detects faces → recognizes faces → predicts the identity. **True statements:** "access videos" sends the video/audio stream (message 1) to "detects faces"; "detects faces" has a self-message (1.1) for its own processing before passing detected faces on (1.2); "recognizes faces" receives the detected faces and sends recognized faces (1.2.2) to "predicts the identity"; the flow ends with "predicts the identity" running its own prediction step (1.2.2.1), forwarding nothing (it is the sink).

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.



Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

33

NOTES

- 1 The box-and-line view: a **big video file** enters the **YouTube Manager**, which prepares it.
- 2 **Face detection** turns the video into data containing only the detected faces.
- 3 **Face recognition** matches the detected faces against known ones.
- 4 **Identity manager** produces the final output, such as an identity report ("Wanted: Joe Developer"). The callout asks the design question: if a better recognition algorithm appears, only that one filter changes.

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

NOTES

- The box-and-line diagram shows the components but says nothing about **how data is transported** between filters, i.e. the pipes.
- A UML component diagram (next slide) makes the transport explicit.

Pipes-and-Filters Architectural Pattern – Example2

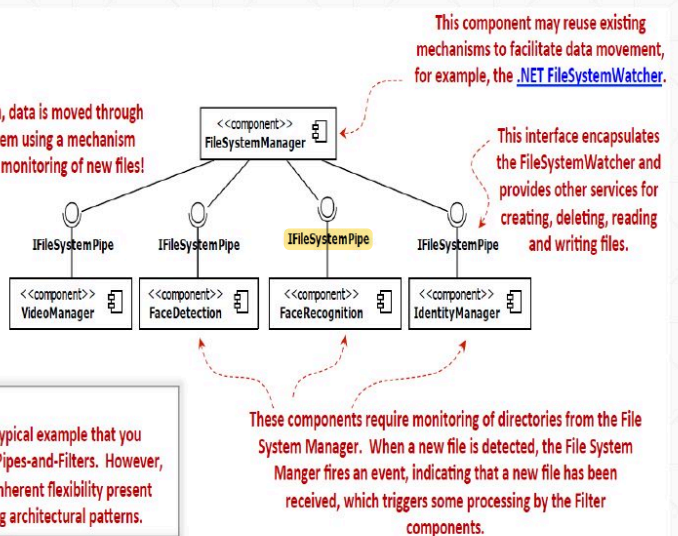
•Extra:

•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.

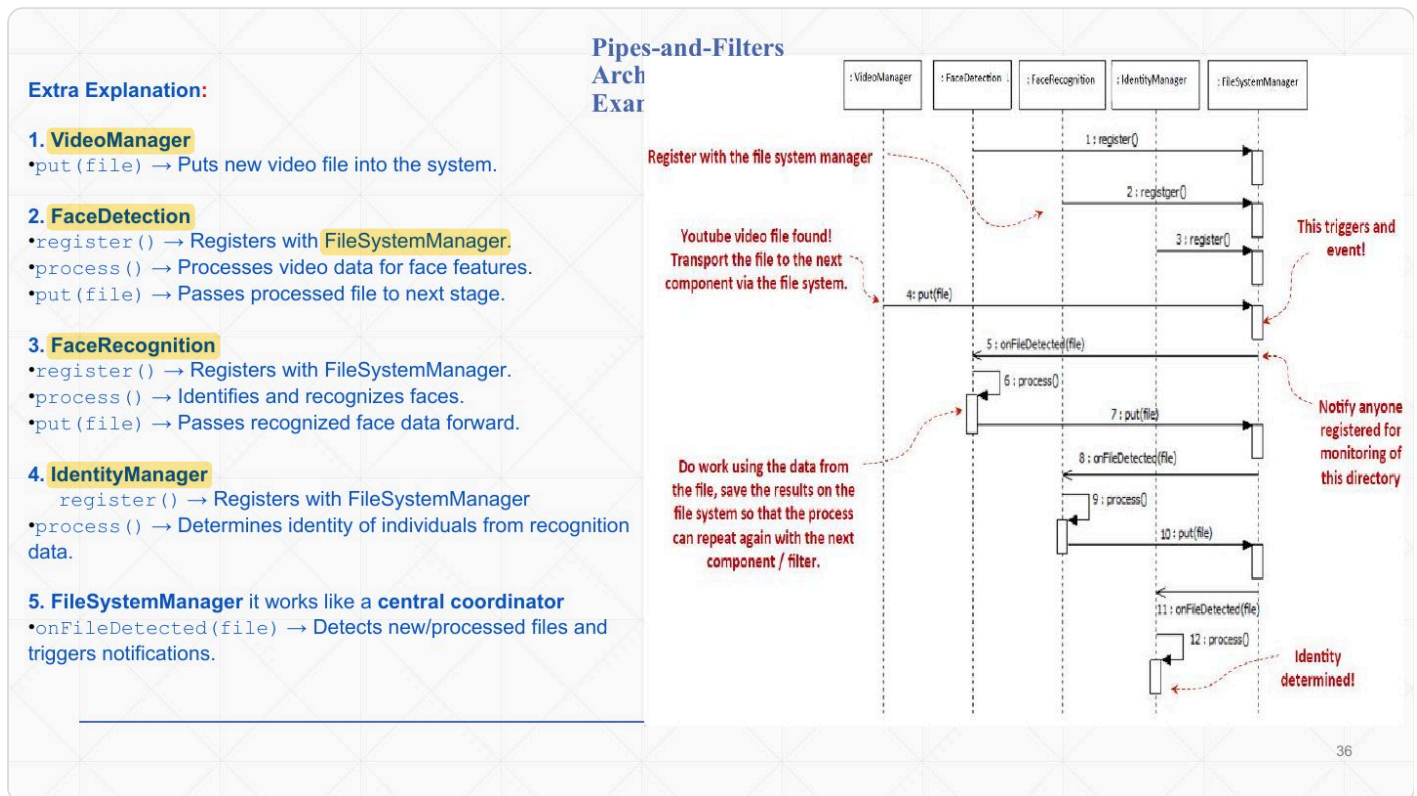
In this system, data is moved through the File System using a mechanism that relies on monitoring of new files!



NOTES

- 1 Here the pipe is the **file system**: a `FileSystemManager` component (built on .NET's `FileSystemWatcher`) monitors directories and fires an event when a new file appears.
- 2 The pipeline components (`VideoManager`, `FaceDetection`, `FaceRecognition`, `IdentityManager`) each require the **`IFileSystemPipe`** interface the manager provides. A filter writes its output as a file; the manager detects it and triggers the next filter. The warning notes this is not the typical pipes-and-filters example but shows how flexible the pattern is.

SLIDE 35 / 48 Pipes-and-Filters – Example 2 sequence diagram



NOTES

- 1 **VideoManager:** `put(file)` places a new video file into the system (message 4).
- 2 **FaceDetection:** `register()`s with the `FileSystemManager` at start-up, then, when notified, `process()`es the video for faces and `put(file)`s its result.
- 3 **FaceRecognition:** registers, processes the detected faces when notified, and puts the recognised-face data forward.
- 4 **IdentityManager:** registers and, when notified, processes the recognition data to determine identity (the last step: "identity determined!").
- 5 **FileSystemManager:** the coordinator. Its `onFileDetected(file)` notifies the registered filter whenever a new or processed file appears, which is what moves data along the pipeline.

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.

37

NOTES

- 1 **Extensibility:** add new filters easily (a spam filter). **Efficiency:** filters can run in parallel. **Reusability:** filters are reused across systems. **Modifiability:** add, remove or change filters independently. **Security:** security filters can be inserted anywhere in the chain. **Maintainability:** independent filters make updates simple. Trade-off: every boundary needs agreed data formats, and the pattern suits batch-like flows, not interactive ones where the user steers the work midway.

EXAM ANGLE

Prediction

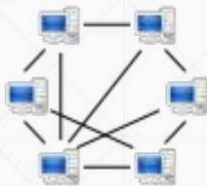
Q1

List three quality properties of the pipes-and-filters pattern.

Answer: Any three of: extensibility (new filters), efficiency (filters run concurrently), reusability, modifiability (add/remove/reorder filters), security (insert security filters), maintainability.

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

Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

38

NOTES

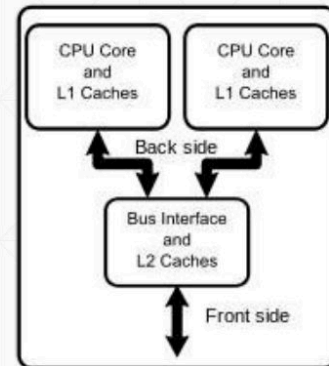
- 1 Distributed** systems are decomposed into multiple processes that usually collaborate over a network.
- 2 Client-server:** client-side processes work for users and act as a bridge to a remote server that does the work delegated to it.
- 3 Peer-to-peer:** peer nodes with similar capabilities collaborate as equals, e.g. music-sharing applications. The pictures contrast a mesh of peers with a star around one 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.~~



Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

39

NOTES

- 1 Those systems are easy to spot because they are deployed on **multiple physical nodes**.
- 2 With **multi-core processors**, distributed architecture also applies inside a single machine: work is spread across cores instead of computers.

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.

Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

40

NOTES

- 1 Examples: Internet systems, web services, file- and music-sharing systems, high-performance computing.
- 2 Common patterns: **client-server** (clients request, a central server provides, e.g. browser and web server) and **broker**.
- 3 **Peer-to-peer**: every node is both client and server, sharing resources with no central server (BitTorrent).

EXAM ANGLE

Prediction

Q1

How does peer-to-peer differ from client-server?

Answer: In client-server, clients request services from a central server. In peer-to-peer every node acts as both client and server and they collaborate without relying on one central node, so the system keeps working when any single peer leaves.

SLIDE 40 / 48 Client-Server Pattern

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:

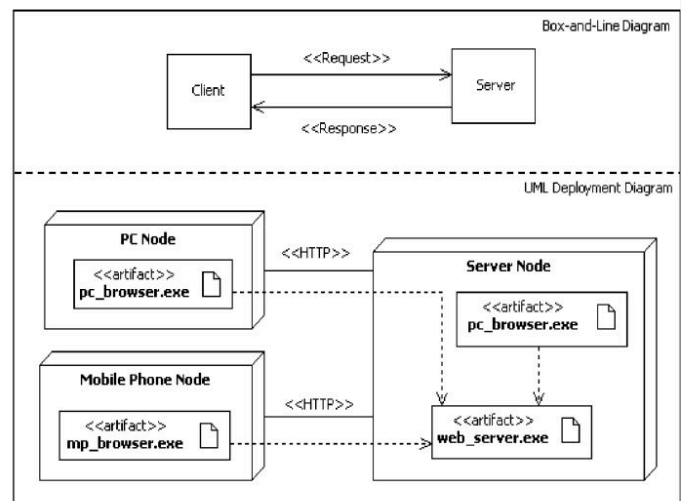
Clients (PC or Mobile) communicate with the server through

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**.



41

NOTES

- 1 **Server:** a program or device that provides functionality to other programs or devices, its clients.
- 2 **Client:** hardware or software that accesses a service the server makes available. Many independent clients talk to the server, and **clients do not talk to each other**.
- 3 **Deployment diagram** (bottom): the PC node runs `pc_browser.exe`, the mobile node runs `mp_browser.exe`, and the server node runs `web_server.exe` (artifacts on nodes).

- 4 Clients communicate with the server over **HTTP**; the server responds with processed data. The box-and-line diagram above shows the same thing as request/response arrows.

SLIDE 41 / 48 Client-Server Pattern

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.

Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

42

NOTES

- 1 **Interoperability:** clients on different platforms work with servers on different platforms (any browser, any OS).
- 2 **Modifiability:** changes are made once at the server and reach all clients.
- 3 **Availability:** backup server nodes keep the service and data available.
- 4 **Reusability:** server services and data can be reused by different applications.

EXAM ANGLE

Past exam · Midterm 261 · Q6 (one of seven)

Q1

A classroom response application allows hundreds of students to submit answers from their phones while an instructor watches the results on a dashboard. Scores and question status must come from one authoritative course session so that every participant sees consistent information. Which architectural style?

Answer: Client-server: one authoritative central service that many clients use.

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.

NOTES

- 1 The main problem: **clients are tightly coupled to servers**, which becomes complex when services come from many servers in different places.
- 2 The client must know the server's exact IP/URL/protocol, and any server change forces client code updates.
- 3 The **broker** pattern was introduced to remove that coupling.

EXAM ANGLE

Prediction

Q1

What is the main drawback of client-server that the broker pattern fixes?

- A. Clients are tightly coupled to specific servers
- B. It cannot store data
- C. It needs a blackboard
- D. Clients talk to each other

Answer: A. Clients are tightly coupled to specific servers

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

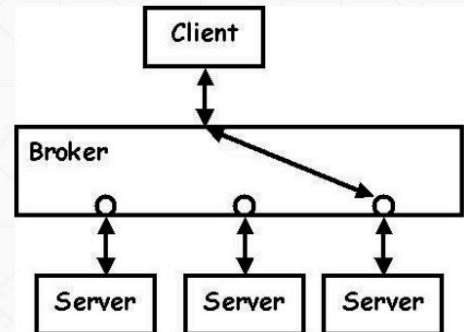
Client-Server

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

Broker

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

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



NOTES

- 1 The **broker** pattern gives more flexibility between clients and servers in a distributed environment by acting as a **mediator** between them.
- 2 It **decreases coupling**: one client can transparently use the services of many servers, knowing only the broker.
- 3 Comparison: **client-server** is direct, the client must know the server's address and protocol, server changes need client updates, simpler but less flexible. **Broker** goes through middleware, the client knows only the broker, server changes don't affect clients, more scalable. In short: direct and tight vs. indirect and loose.

EXAM ANGLE

Past exam · Midterm 261 · Q6 (one of seven)

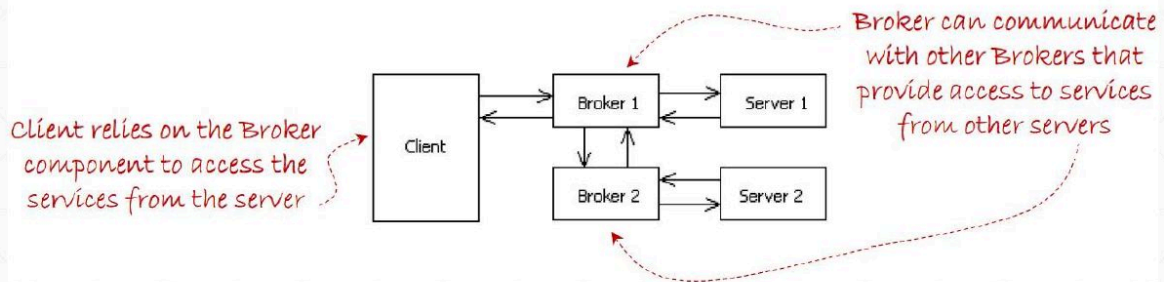
Q1

A hospital application sends requests to several types of laboratory equipment supplied by different vendors. Equipment can be replaced or added over time, but the clinical application should continue using the same request format and should not need vendor-specific logic. Which architectural style?

Answer: Broker: the broker hides vendor- and service-specific communication behind one request format.

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**.



Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

45

NOTES

- 1 The **broker** coordinates communication between clients and servers: it forwards requests and transmits results and exceptions back.
- 2 This makes systems more **flexible and interoperable**. In the diagram the client relies on Broker 1 for services from Server 1, and Broker 1 can talk to Broker 2 to reach Server 2's services too.

SLIDE 45 / 48 Broker Pattern components

➤ Broker Pattern

➤ includes the following components:

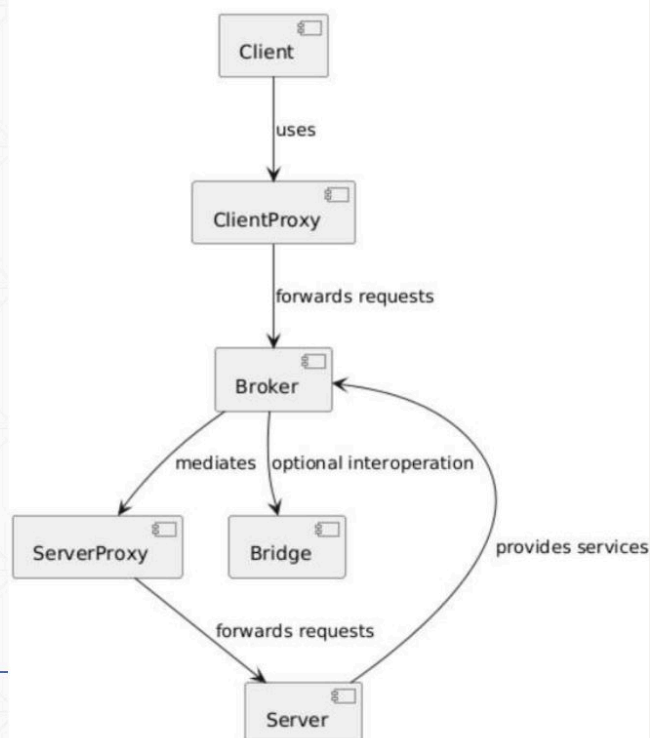
Component	Description
Client	Applications that use the services provided by one or more servers.
ClientProxy	Component that provides transparency (at client) between remote and local components so that remote components appear as local ones.
Broker	Component that mediates between client and server components.
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

Extra:

Broker Pattern - Component Diagram



NOTES

- 1 **Client:** uses services of one or more servers. Provides nothing; requires the client proxy.
- 2 **Client proxy:** makes remote components look local to the client (transparency). Provides a local service to the client; requires the broker.
- 3 **Broker:** mediates between client and server components. Provides mediation to the client proxy; requires the server proxy.
- 4 **Server proxy:** makes remote components look local on the server side. Provides a local service to the broker; requires the server.
- 5 **Server:** provides the real application services (it may also act as a client of the broker).
- 6 **Bridge** (optional): connects different brokers so they can interoperate.

EXAM ANGLE

Prediction

Q1

List the components of the broker pattern in the order a request passes through them.

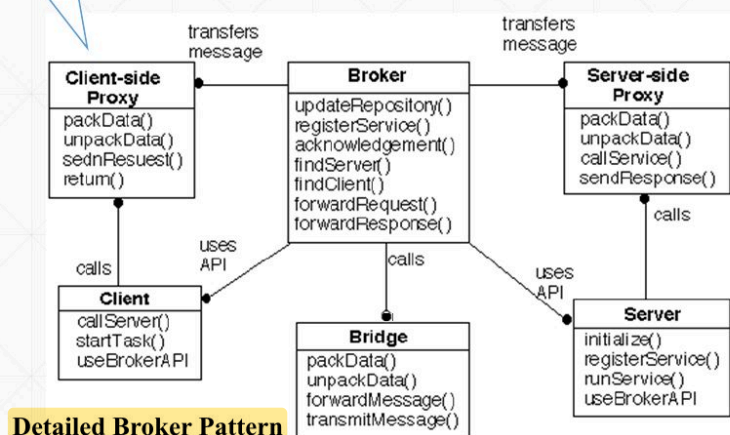
Answer: Client → client-side proxy → broker → server-side proxy → server (a bridge optionally connects brokers). The response returns the same way.

SLIDE 46 / 48 Broker Pattern

Broker Pattern

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

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



NOTES

- 1 The **client-side proxy** decides whether a request can be resolved locally or must be forwarded to the broker.

2 The pattern **separates communication functionality from application functionality**: everything about networks and messages sits in the broker and proxies, so clients and servers contain only application logic.

3 Reading the detailed class diagram:

Client-side proxy

`packData()`, `unpackData()`, `sendRequest()`, `return()` : marshals requests into messages and results back out.

Broker

`registerService()`, `findServer()`, `findClient()`, `forwardRequest()`, `forwardResponse()` : knows where services are and routes messages.

Server-side proxy

`packData()`, `unpackData()`, `callService()`, `sendResponse()` : calls the real service.

Client / Server / Bridge

The client calls `callServer()` via the broker API; the server `registerService()` s and `runService()` s; the bridge transmits messages between brokers.

SLIDE 47 / 48 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.

NOTES

1 **Interoperability**: clients work with servers on different platforms, and with multiple servers.

2 **Modifiability**: server changes are centralised and reach many clients quickly.

3 **Portability**: the broker can be ported to new platforms, opening the services to new clients.

- 4 **Reusability:** because brokers handle the system calls, complex services can be reused in other applications.

EXAM ANGLE

Prediction

Q1

Which is a quality property of the broker pattern?

- A. Interoperability across platforms
- B. Tighter client–server coupling
- C. No need for middleware
- D. Guaranteed fastest performance

Answer: A. Interoperability across platforms

SLIDE 48 / 48 Summary

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

Dr. Banda Ramadan | Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

50

NOTES

- 1 Data-centered → **blackboard**.
- 2 Data flow → **pipes and filters**.
- 3 Distributed → **client–server** and **broker**.
- 4 Next: interactive systems (MVC) and hierarchical systems (layered).

Original slides from the SE322 lecture deck, with study notes. Check exact wording against the [original deck](#).
Notes and questions are study aids. Made by Shoug Alomran.