Software Design Methodologies and Testing

(Subject Code: 410449)

(Class: BE Computer Engineering)

2012 Pattern

Objectives and outcomes

- Course Objectives
  - To understand and apply different design methods and techniques
  - To understand architectural design and modeling
  - To understand and apply testing techniques
  - To implement design and testing using current tools and techniques in distributed, concurrent and parallel
  - Environments
- Course Outcomes
  - To present a survey on design techniques for software system
  - To present a design and model using UML for a given software system
  - To present a design of test cases and implement automated testing for client server, distributed, mobile applications

Other Information

- Teaching Scheme Lectures:
  - 3 Hrs/Week
- Examination Scheme
  - In Semester Assessment: 30
  - End Semester Assessment : 70

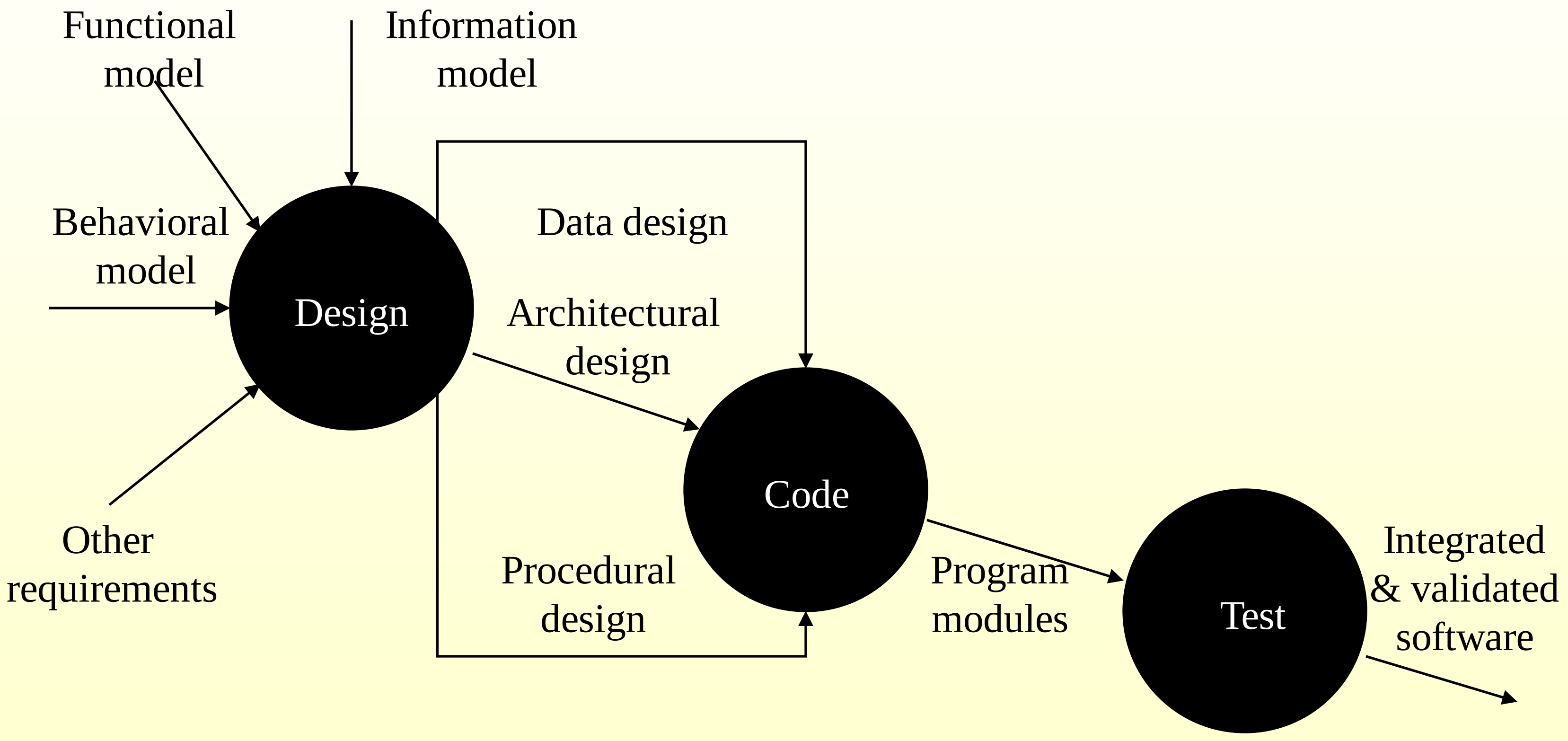
UNIT-II

Architectural Design

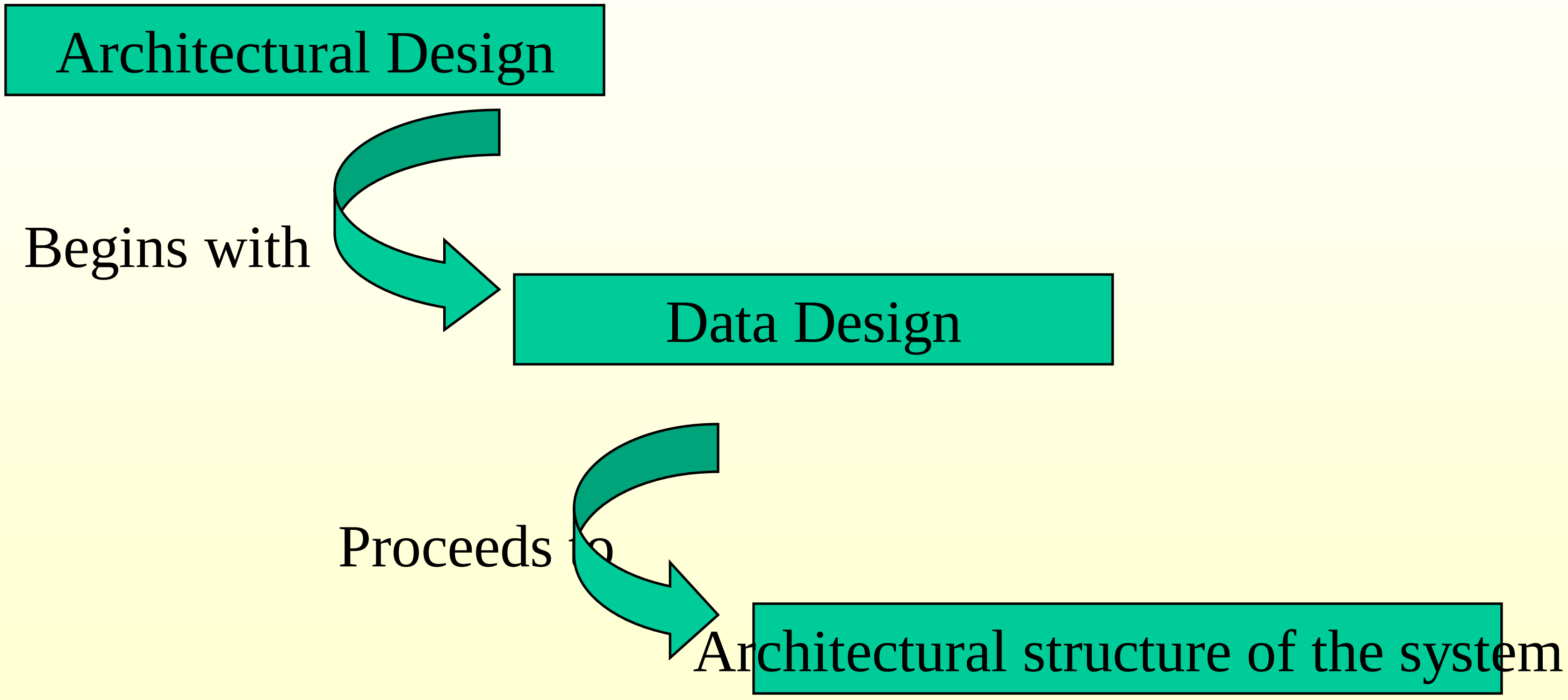
Architectural Design, importance and architecture views, client-server, service oriented, component based concurrent and real time software architecture with case studies

UNIT-II
Architectural Design

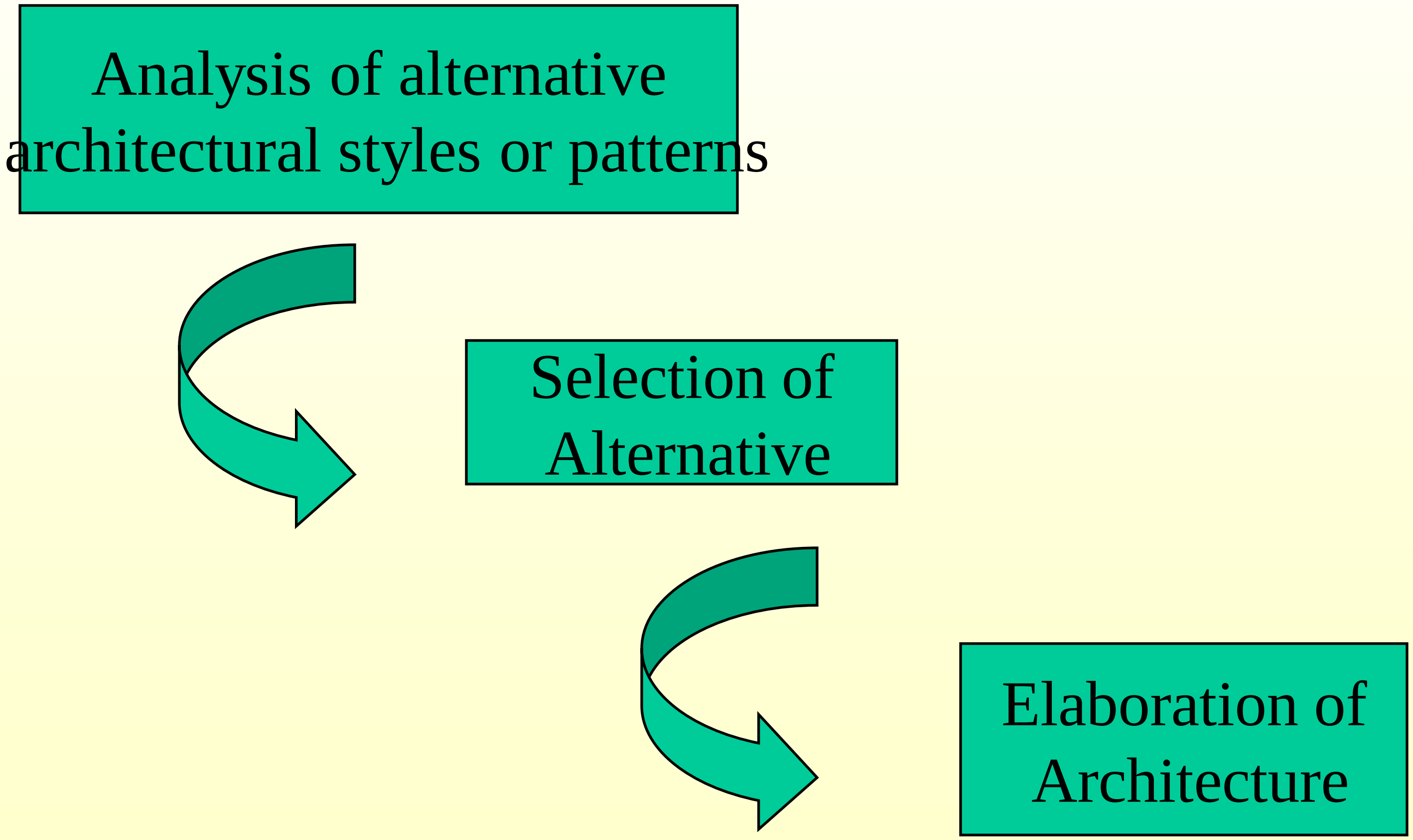
Software Design Model



Steps


Architectural Design
Begins with
Data Design
Proceeds to
Architectural structure of the system

Steps (contd.)


Analysis of alternative architectural styles or patterns
Selection of Alternative
Elaboration of Architecture

Software Architecture

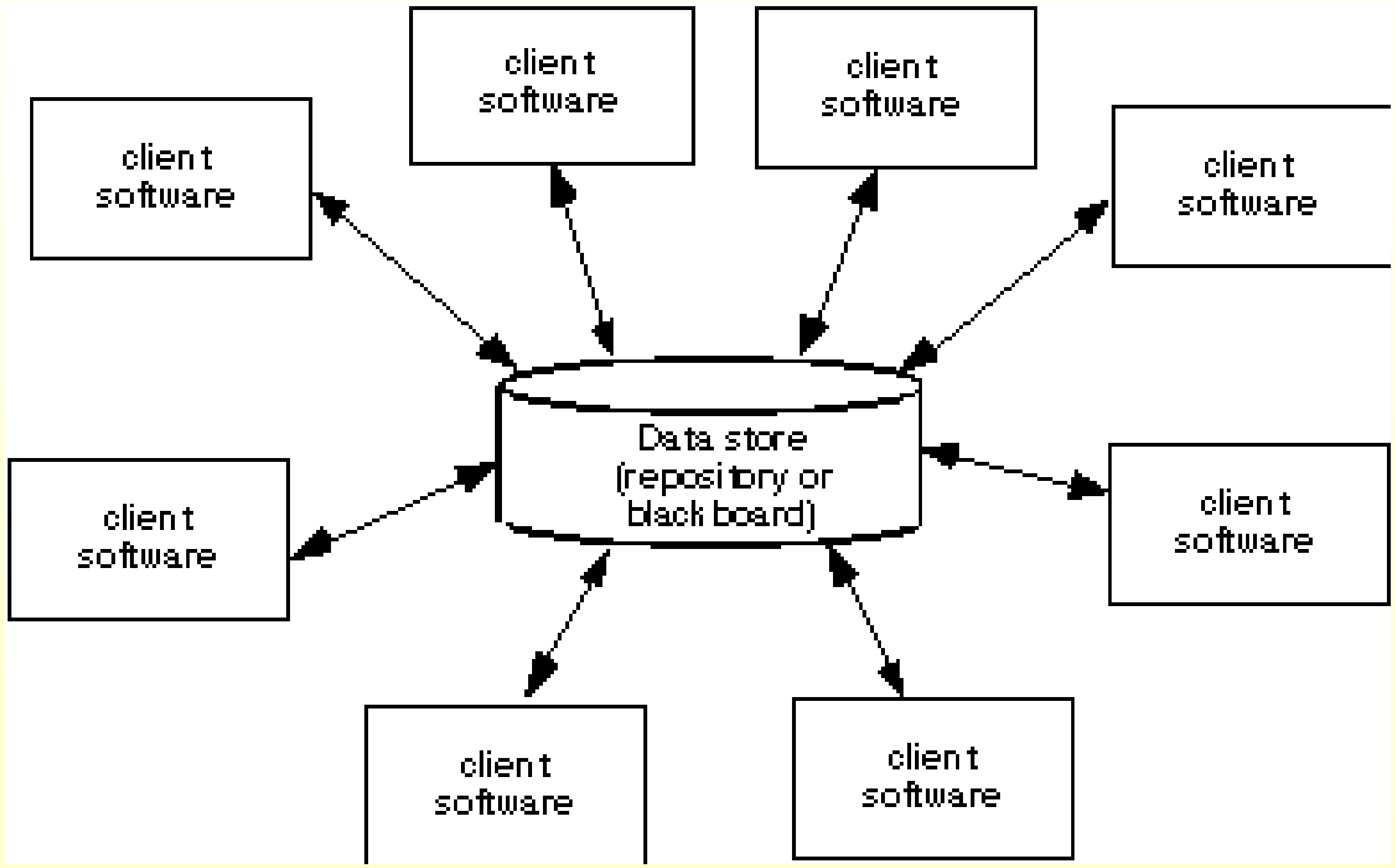
- What Is Architecture?
  - The software architecture of a program or computing system is the **structure or structures of the system**, which comprise **software components**, the externally visible properties of those components, and the relationships among them.
  - The architecture is not the operational software.

Architectural Styles

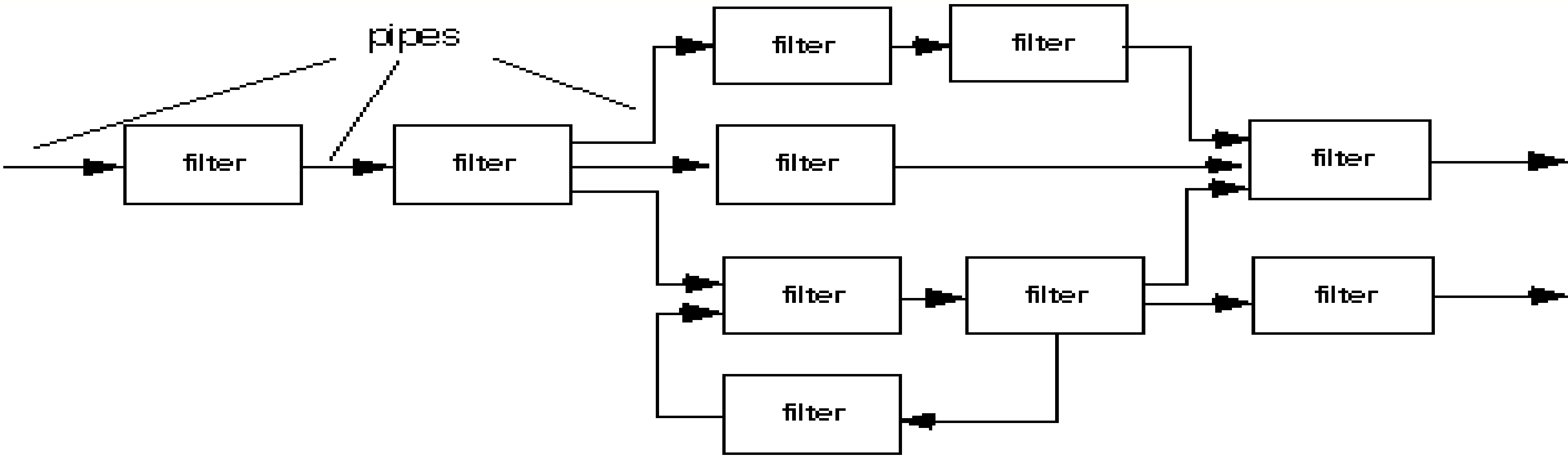
Each style describes a system category that encompasses: (1) a set of components (e.g., a database, computational modules) that perform a function required by a system, (2) a set of connectors that enable "communication, coordination and cooperation" among components, (3) constraints that define how components can be integrated to form the system, and (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

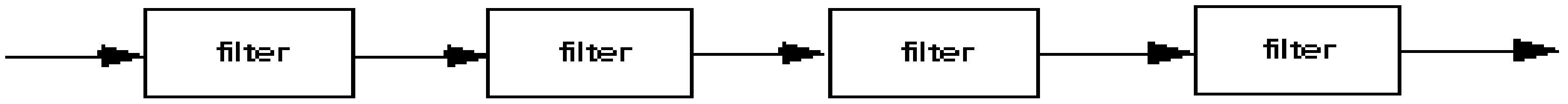
Data-Centered Architecture



Data Flow Architecture

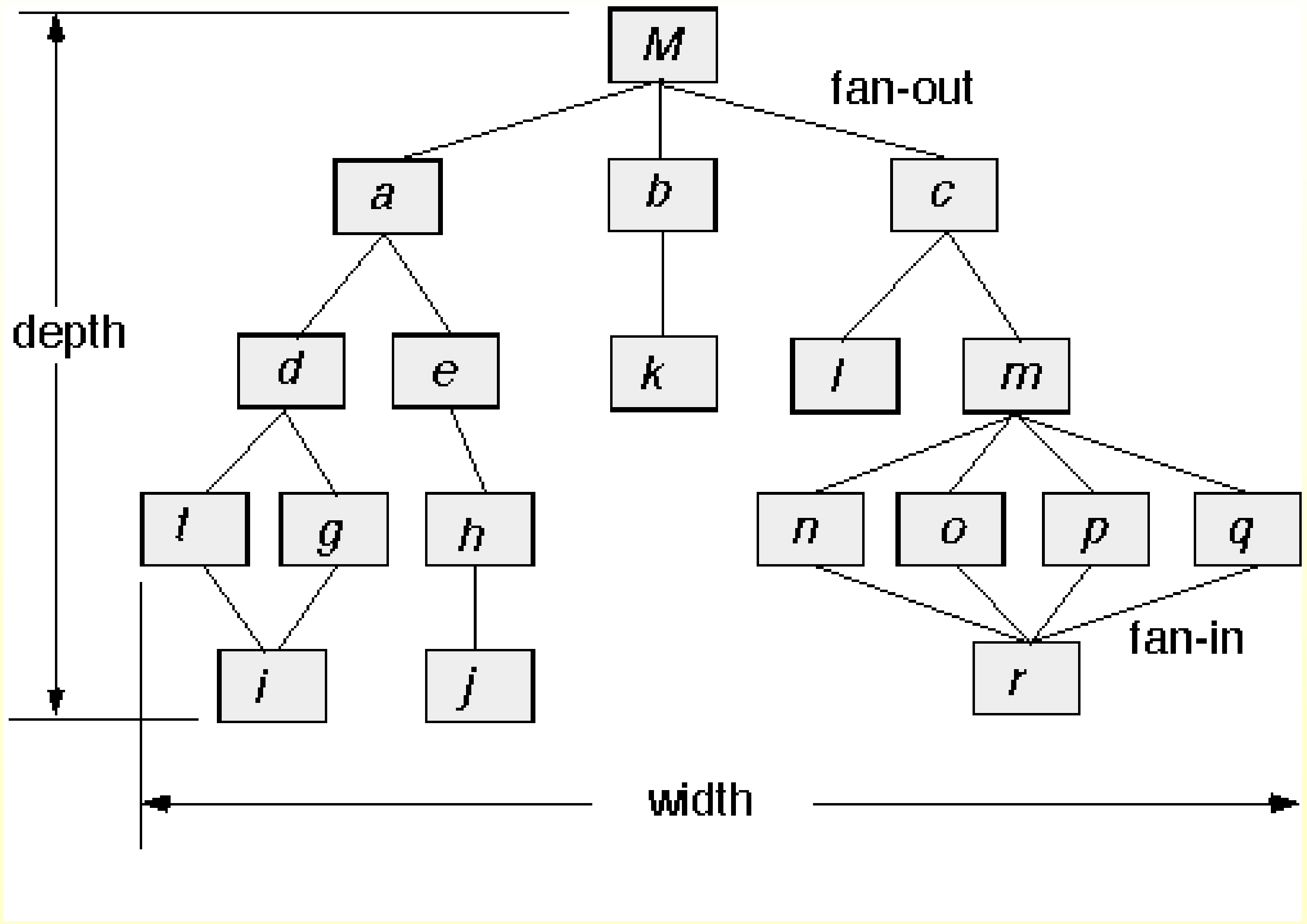


(a) pipes and filters

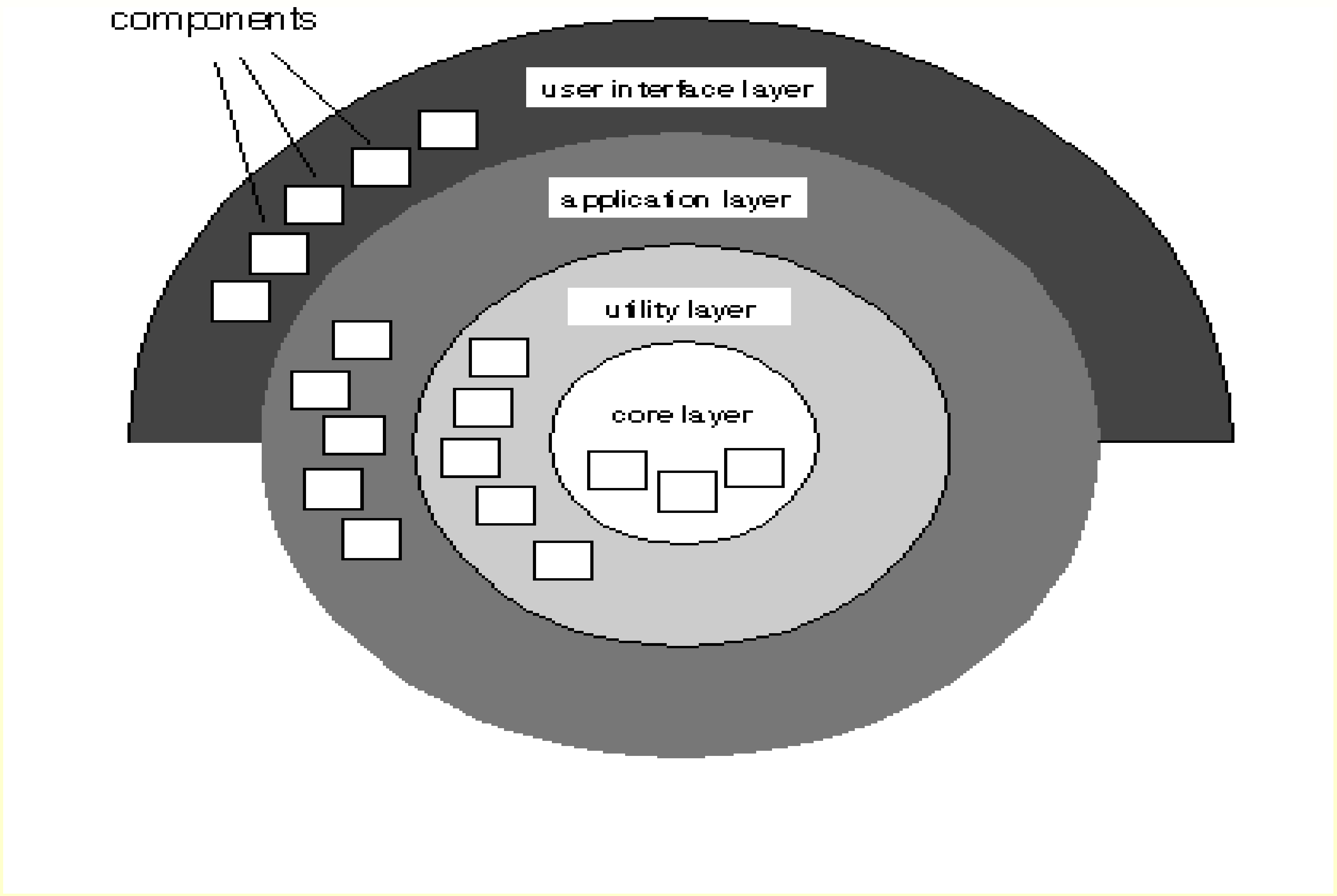


(b) batch sequential

Call and Return Architecture



Layered Architecture



ADL

- *Architectural description language* (ADL) provides a semantics and syntax for describing a software architecture
- Provide the designer with the ability to:
  - decompose architectural components
  - compose individual components into larger architectural blocks and
  - represent interfaces (connection mechanisms) between components.

Architectural Design-Introduction

- structure or structures of the system which comprise
  - The software components
  - The externally visible properties of those components
  - The relationships among the components

- Software architectural design represents the structure of the data and program components that are required to build a computer-based system

- The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is architectural design.
- The output of this design process is a description of the software architecture.

Architectural Design-Introduction

- A software architecture is defined by Bass, Clements, and Kazman (2003) as follows:

- "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

Architectural Design-Introduction

- Basic Steps
  - Creation of the data design
  - Derivation of one or more representations of the architectural structure of the system
  - Analysis of alternative architectural styles to choose the one best suited to customer requirements and quality attributes
  - Elaboration of the architecture based on the selected architectural style
- A database designer creates the data architecture for a system to represent the data components
- A system architect selects an appropriate architectural style derived during system engineering and software requirements analysis

Architectural Design-Introduction

- A software architecture enables a software engineer to
  - Analyze the <u>effectiveness</u> of the design in meeting its stated requirements
  - Consider architectural <u>alternatives</u> at a stage when making design changes is still relatively easy
  - Reduce the <u>risks</u> associated with the construction of the software
- Focus is placed on the software component
  - A program module
  - An object-oriented class
  - A database
  - Middleware

Architectural Design-Introduction

- An early stage of the system design process.
- Represents the link between specification and design processes.
- Often carried out in parallel with some specification activities.
- It involves identifying major system components and their communications.

Importance of Software Architecture

- Representations of software architecture are an enabler for communication between all stakeholders interested in the development of a computer-based system
- The software architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity
- The software architecture constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together

Importance of Software Architecture

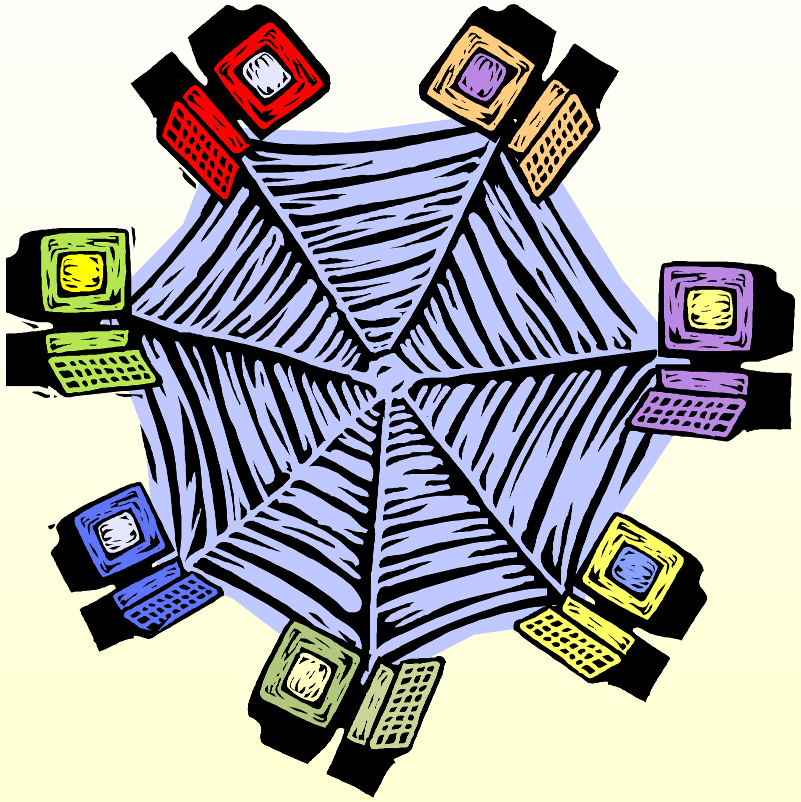
- A key to reducing development costs
  - Component-based development philosophy
  - Explicit system structure
- A natural evolution of design abstractions
  - Structure and interaction details overshadow the choice of algorithms and data structures in large/complex systems
- Benefits of explicit architectures
  - A framework for satisfying requirements
  - Technical basis for design
  - Managerial basis for cost estimation & process management
  - Effective basis for reuse
  - Basis for consistency, dependency, and tradeoff analysis
  - Avoidance of architectural loss

Key Architectural Styles

- Client/Server
  - Segregates the system into two applications, where the client makes requests to the server.
  - In many cases, the server is a database with application logic represented as stored procedures.
- *Service-Oriented Architecture (SOA)*
  - Refers to applications that expose and consume functionality as a service using contracts and messages.
- *Component-Based Architecture*
  - Decomposes application design into reusable functional or logical components that expose well-defined communication interfaces.

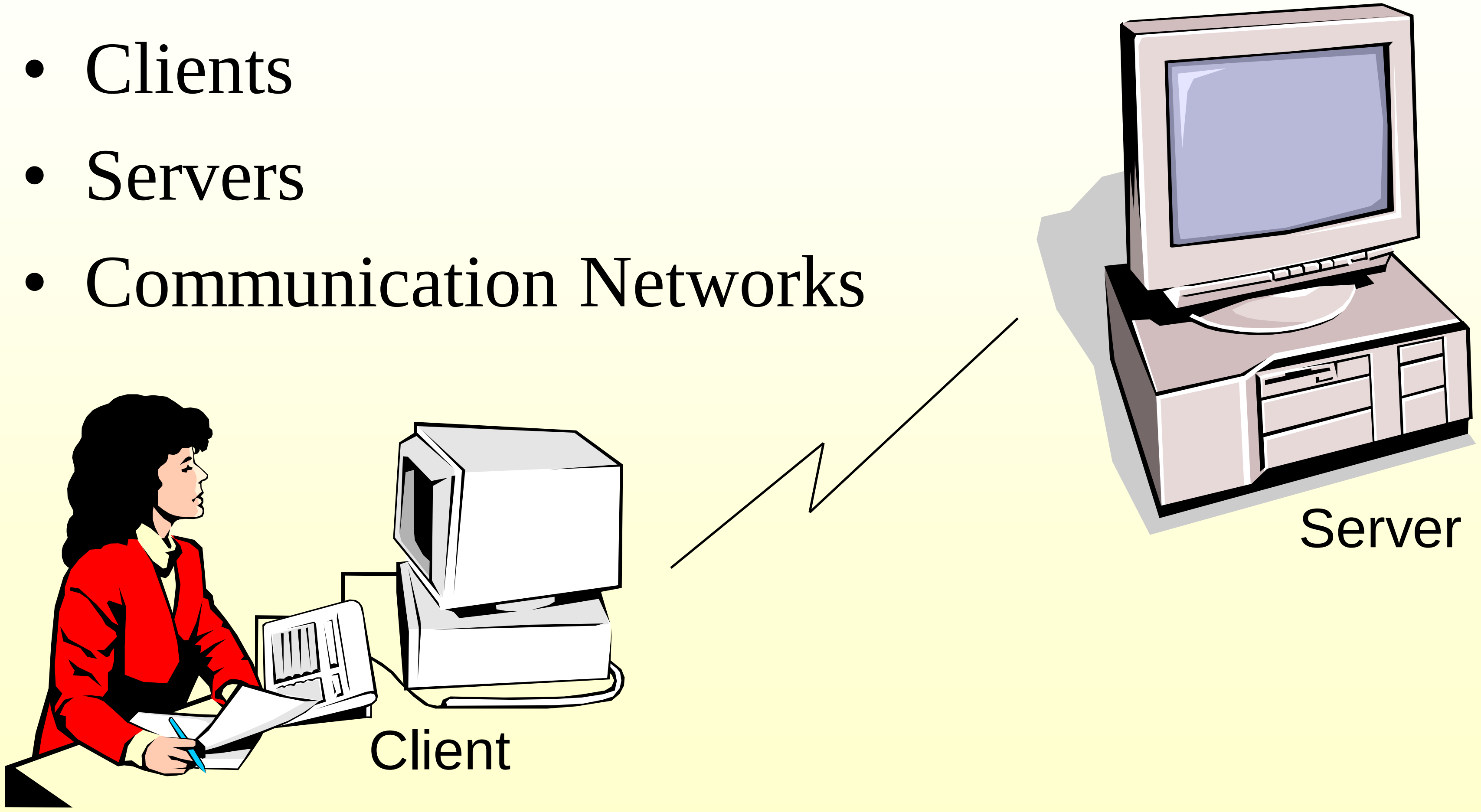
Client Server Architecture

- A network architecture in which each computer or process on the network is either a *client* or a *server*.
- The simplest client/server architecture has one service and many clients.
- More complex client/server systems might have multiple services.



Components

- Clients
- Servers
- Communication Networks



Clients

- Applications that run on computers
- Rely on servers for
  - Files
  - Devices
  - Processing power
- Example: E-mail client
  - An application that enables you to send and receive e-mail

Clients are Applications

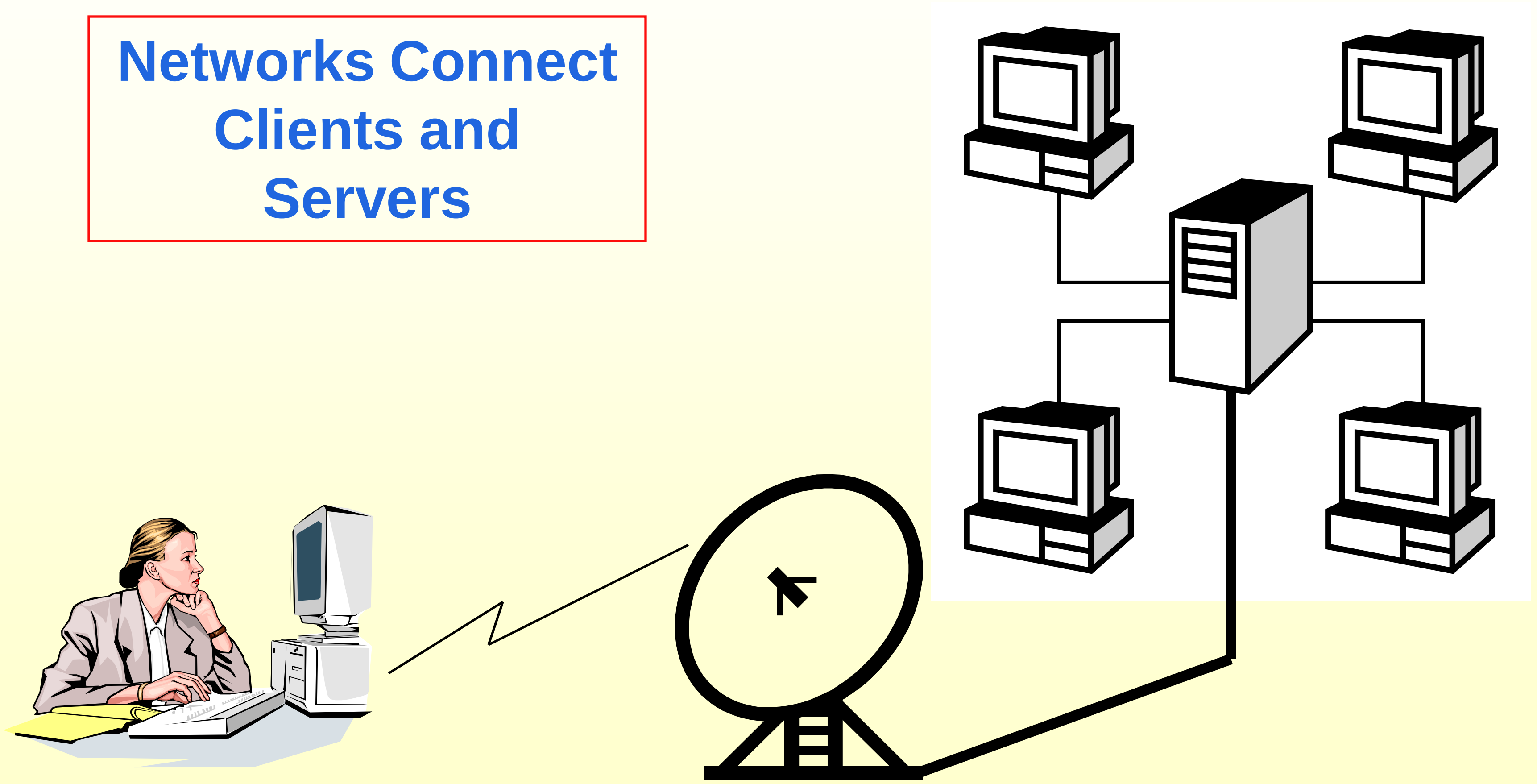
Servers

- Computers or processes that manage network resources
  - Disk drives (file servers)
  - Printers (print servers)
  - Network traffic (network servers)
- Example: Database Server
  - A computer system that processes database queries

Servers Manage Resources

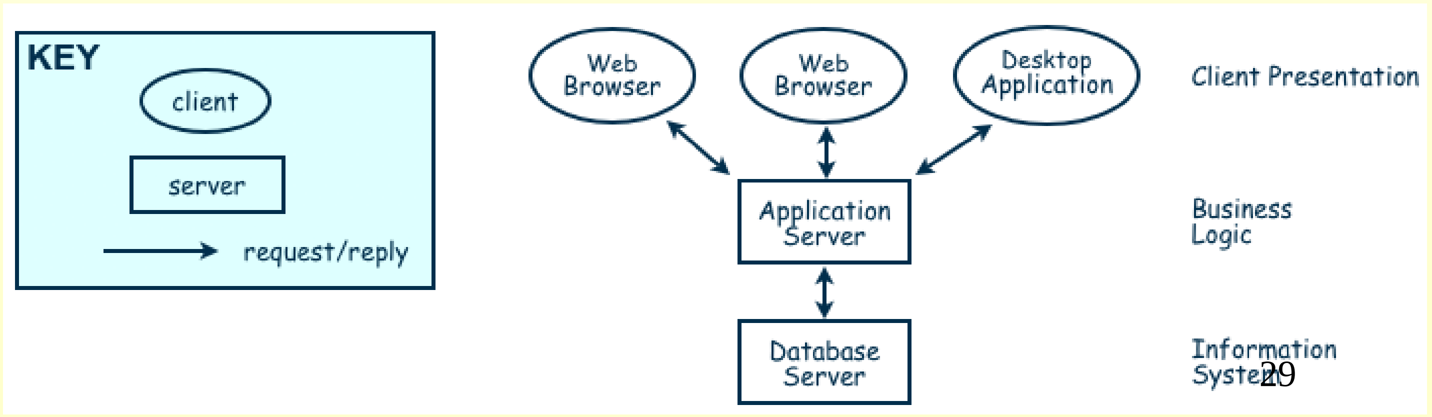
Communication Networks

Networks Connect Clients and Servers



Architectural Styles and Strategies

- **Client/server** architectures are based on client/service architectural patterns, the
- simplest of which consists of one service and multiple clients.
- Two types of components:
  - Server components offer services
  - Clients access them using a request/reply protocol
- Client may send the server an executable function, called a callback
  - The server subsequently calls under specific circumstances

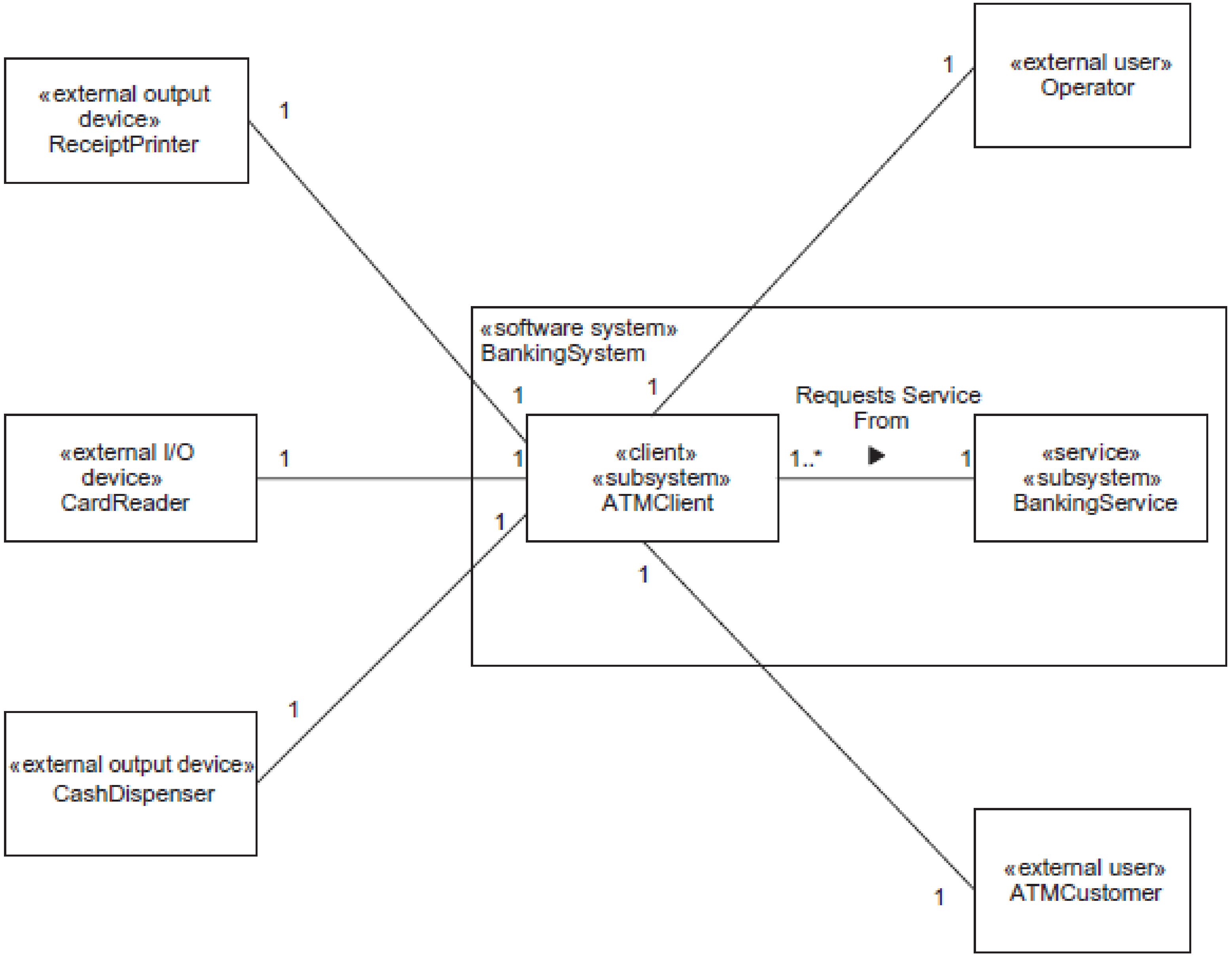


Multiple Client/Single Service Architectural Pattern

- The Multiple Client/Single Service architectural pattern consists of several clients that request a service and a service that fulfills client requests.
- The simplest and most common client/server architecture has one service and many clients, and for this reason the Multiple Client/Single Service architectural pattern is also known as the Client/Server or Client/Service pattern.

Example

- This system contains multiple ATMs and one banking service.
- For each ATM there is one ATM Client Subsystem, which handles customer requests by reading the ATM card and prompting for transaction details at the keyboard/ display.
- For an approved withdrawal request, the ATM dispenses cash, prints a receipt, and ejects the ATM card.
- The Banking Service maintains a database of customer accounts and customer ATM cards.
- It validates ATM transactions and either approves or rejects customer requests, depending on the status of the customer accounts.


«external output device» ReceiptPrinter
«external I/O device» CardReader
«external output device» CashDispenser
«external user» Operator
«external user» ATMCustomer
«software system» BankingSystem
«client» «subsystem» ATMClient
«service» «subsystem» BankingService
Requests Service From
1..*
1

Multiple Client/Multiple Service Architectural Pattern

- In the Multiple Client/Multiple Service pattern, in addition to clients requesting a service, a client might communicate with several services, and services might communicate with each other.
- With this pattern, a client could communicate with each service sequentially or could communicate with multiple services concurrently.

Example

- An example of the Multiple Client/Multiple Service architectural pattern is a banking group consisting of multiple interconnected banks
- Continuing with the ATM example, besides several ATM clients accessing the same bank service, it is possible for one ATM client to access multiple bank services.
- This feature allows customers to access their own bank service from a different bank’s ATM client. In this example, ATM customers from Bank of India can withdraw funds from State Bank of India in addition to their own Bank of India, and vice versa.

Benefits of Client-Server Architecture

- The main benefits of the client/server architectural style are:
- Higher security: All data is stored on the server, which generally offers a greater control of security than client machines.
- Centralized data access: Because data is stored only on the server, access and updates to the data are far easier to administer than in other architectural styles.
- Ease of maintenance: Roles and responsibilities of a computing system are distributed among several servers that are known to each other through a network.
  - This ensures that a client remains unaware and unaffected by a server repair, upgrade, or relocation.

Multi-tier Client/Service Architectural Pattern

- The Multi-tier Client/Service pattern has an intermediate tier (i.e., layer) that provides both a client and a service role.
- An intermediate tier is a client of its service tier and also provides a service for its clients.
- It is possible to have more than one intermediate tier.
- When viewed as a layered architecture, the client is considered to be at a higher layer than the service because the client depends on and uses the service.

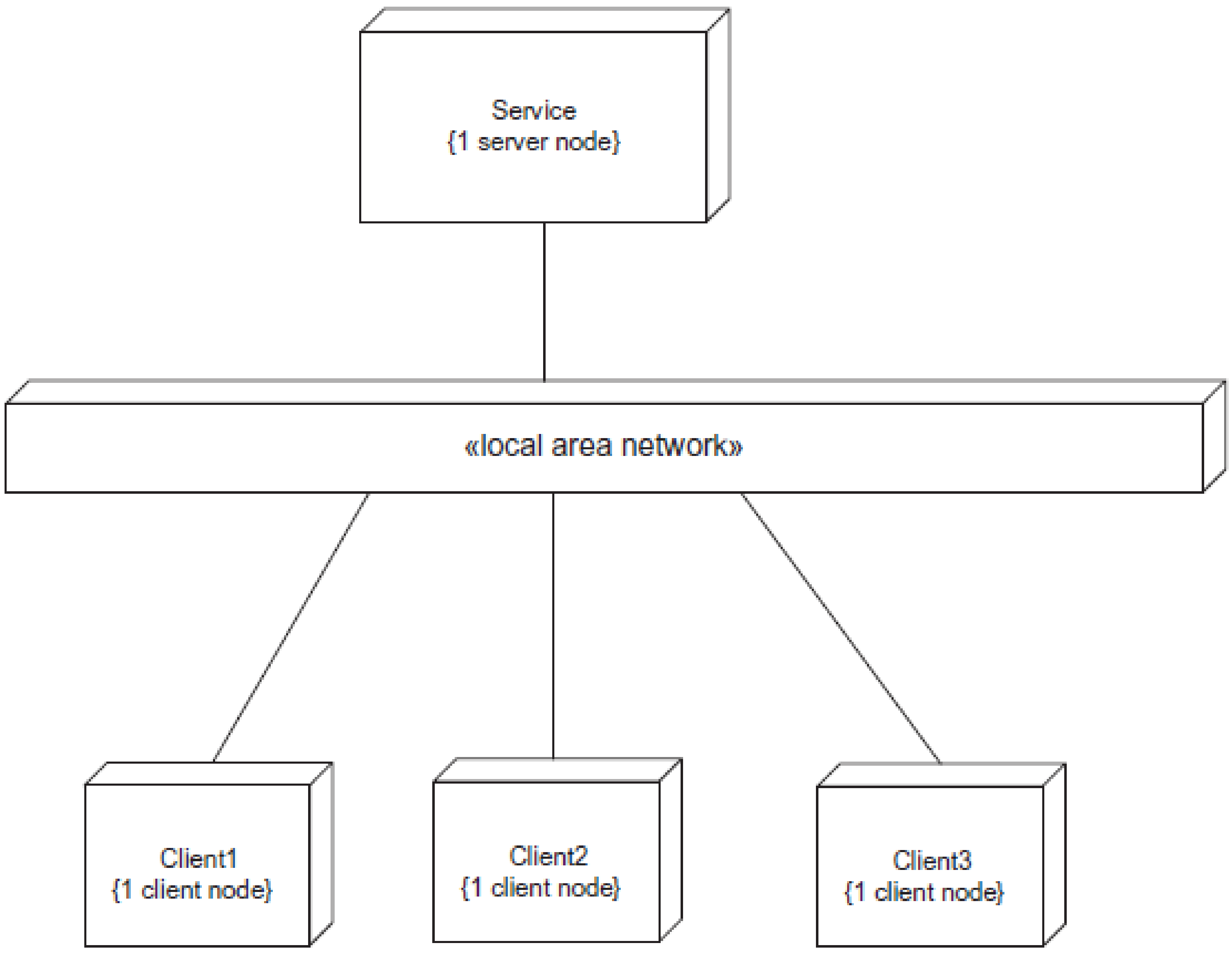
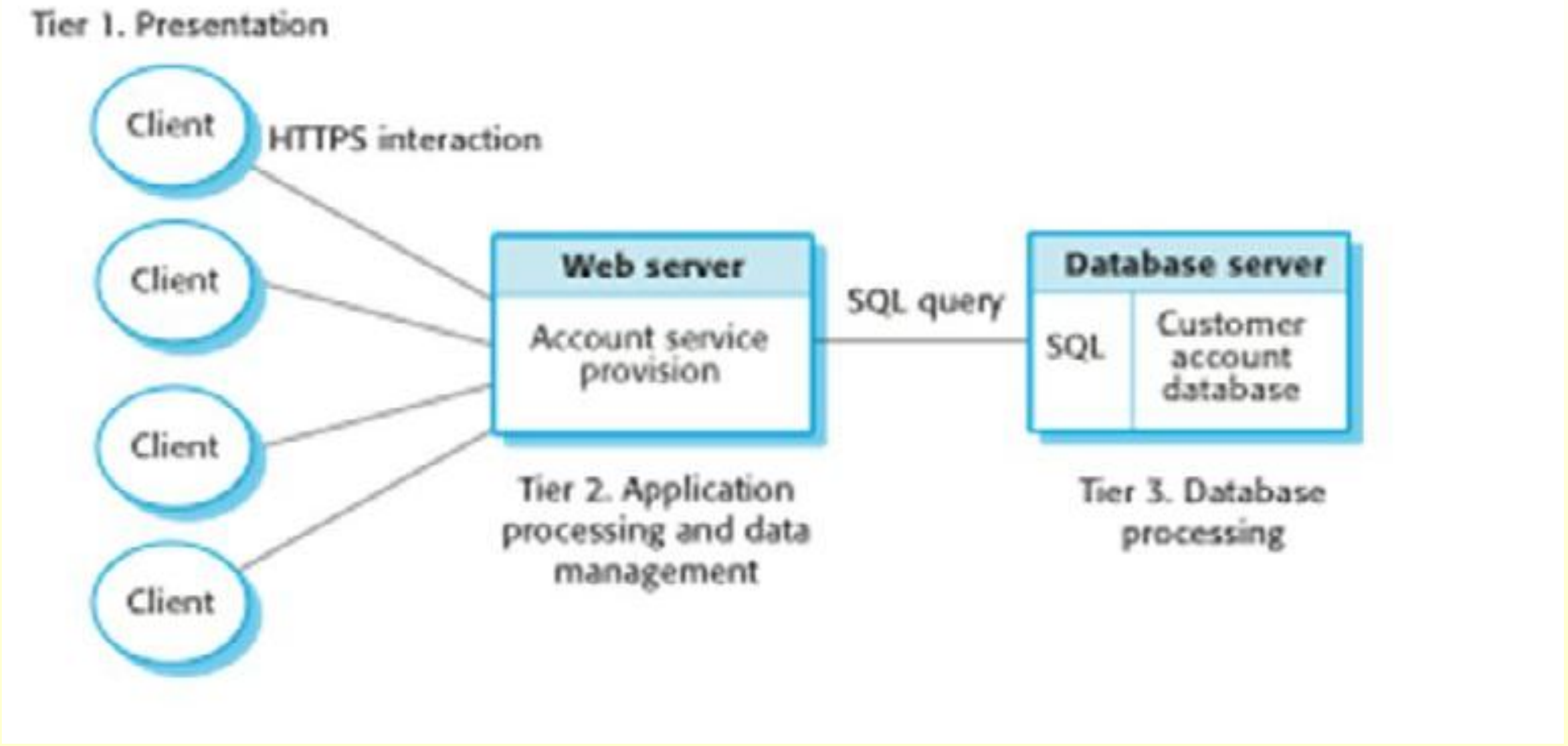


Figure 15.1. Multiple Client/Single Service architectural pattern

Three-tier architecture for an Internet banking system



Tier 1. Presentation
Client
HTTPS interaction
Client
Client
Client
Web server
Account service provision
SQL query
Database server
SQL
Customer account database
Tier 2. Application processing and data management
Tier 3. Database processing

Benefits of the N-tier/3-tier architectural style

- The main benefits of the N-tier/3-tier architectural style are:
- **Maintainability:** Because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole.
- **Scalability:** Because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward.
- **Flexibility**: Because each tier can be managed or scaled independently, flexibility is increased.
- **Availability**: Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

Service Oriented Architecture (SAO) Design

- A service-oriented architecture (SOA) is a distributed software architecture that consists of multiple autonomous services.
- The services are distributed such that they can execute on different nodes with different service providers.
- With a SOA, the goal is to develop software applications that are composed of distributed services, such that individual services can execute on different platforms and be implemented in different languages.

Service Oriented Architecture Design

- Standard protocols are provided to allow services to communicate with each other and to exchange information.
- In order to allow applications to discover and communicate with services, each service has a service description.
- The service description defines the name of the service, the location of the service, and its data exchange requirements.

Service Oriented Architecture Design

- A service provider supports services used by multiple clients.
- Usually, a client will sign up for a service provided by a service provider, such as an Internet, email, or Voice over Internet Protocol (VoIP) service.
- Unlike client/server architectures, in which a client communicates with a specific service provided on a fixed server configuration.
- SOAs, which build on the concept of loosely coupled services that can be discovered and linked to by clients (also referred to as service consumers or service requesters) with the assistance of service brokers.

Design Principles for Services

- Services need to be designed according to certain key principles.
- Many of these concepts are good software engineering and design principles, which have been incorporated into SOA design.
- **Loose coupling:** Services should be relatively independent of each other and a service should hold a minimum amount of information about other services and ideally should not depend on other services.
- **Service contract:** A service provides a contract, which a SOA application can rely on.
  - The contract is typically defined in the service interface in the form of a set of operations and each operation usually has input and output parameters, but it can also include quality of service parameters such as response time and availability.

Design Principles for Services

- Autonomy: Each service is self-contained, such that it can operate independently without the need of other services.
  - This concept can be achieved by separating services from coordination, so that services do not directly communicate with each other.
- Abstraction: As with object-oriented design, the details of a service are hidden, A service only reveals its interface in terms of the operations it provides, and for each operation, the inputs it needs, and the outputs it returns.
- Reusability: A key goal of SOA is to design services that are reusable.
  - The preceding design goals of services are intended to facilitate reuse.

Design Principles for Services

- Composability: Services are designed to be capable of being assembled into larger composite services.
  - In some cases, a composite service also needs to provide coordination of the individual services.
- Statelessness: Where possible, services maintain little or no information about specific client activities.
- Discoverability: A service provides an external description to help allow it to be discovered by a discovery mechanism.

SOFTWARE ARCHITECTURAL BROKER PATTERNS

- In a SOA, object brokers act as intermediaries between clients and services.
- The broker frees clients from having to maintain information about where a particular service is provided and how to obtain that service.
- Sophisticated brokers provide white pages (naming services) and yellow pages (trader services) so that clients can locate services more easily.

SOFTWARE ARCHITECTURAL BROKER PATTERNS

- In the Broker pattern (which is also known as the Object Broker or Object Request Broker pattern), the broker acts as an intermediary between the clients and services. Services register with the broker.
- Clients locate services through the broker.
- After the broker has brokered the connection between client and service, communication between client and service can be direct or via the broker

SOFTWARE ARCHITECTURAL BROKER PATTERNS

- The broker provides both location transparency and platform transparency.

- Location transparency means that if the service is moved to a different location, clients are unaware of the move and only the broker needs to be notified.

- Platform transparency means that each service can execute on a different hardware/software platform and does not need to maintain information about the platforms that other services execute on.

SOFTWARE ARCHITECTURAL BROKER PATTERNS (CONT…)

- The main benefits of the SOA architectural style are:
- Domain alignment- Reuse of common services with standard interfaces increases business and technology opportunities and reduces cost.
- Abstraction- Services are autonomous and accessed through a formal contract, which provides loose coupling and abstraction.
- Discoverability- Services can expose descriptions that allow other applications and services to locate them and automatically determine the interface.
- Interoperability- Because the protocols and data formats are based on industry standards, the provider and consumer of the service can be built and deployed on different platforms.
- Rationalization.-Services can be granular in order to provide specific functionality, rather than duplicating the functionality in number of applications, which removes duplication.

Designing Component-Based Software Architectures

- In distributed component-based software design, the component-based software architecture for the distributed application is developed.
- The software application is structured into components, and the interfaces between the components are defined.
- To assist with this process, guidelines are provided for determining the components.
- Components are designed to be configurable so that each component instance can be deployed to a different node in a geographically distributed environment

Designing Component-Based Software Architectures (Cont…)

- An important goal of a component-based software architecture is to provide a concurrent message-based design that is highly configurable.
- In other words, the objective is that the same software architecture should be capable of being deployed to many different distributed configurations.
- Thus, a given software application could be configured to have each component-based subsystem allocated to its own separate physical node, or, alternatively, to have all or some of its components allocated to the same physical node.
- To achieve this flexibility, it is necessary to design the software architecture in such a way that the decision about how components will be mapped to physical nodes is not made at design time but is made later, at system deployment time.

Designing Component-Based Software Architectures (Cont…)

- A component-based development approach, in which each subsystem is designed as a distributed self-contained component, helps achieve the goal of a distributed, highly configurable, message-based design.
- A distributed component is a concurrent object with a well-defined interface, which is a logical unit of distribution and deployment.
- A well-designed component is capable of being reused in applications other than the one for which it was originally developed.
- A component can be either a composite component or a simple component.
- A composite component is composed of other part components.
- A simple component has no part components within it.

Designing Distributed Component-based Software Architectures (Cont…)

- A distributed application consists of distributed components that can be configured to execute on distributed physical nodes.

- To successfully manage the inherent complexity of large-scale distributed applications, it is necessary to provide an approach for structuring the application into components in which each component can potentially execute on its own node.

- After this design is performed and the interfaces between the components are carefully defined, each component can be designed independently.

Designing Distributed Component-based Software Architectures (Cont…)

- The three main steps in designing a component-based software architecture for a distributed application are:
- 1. **Design distributed software architecture-** Structure the distributed application into constituent components that potentially could execute on separate nodes in a distributed environment.
- Because components can reside on separate nodes, all communication between components must be restricted to message communication. The interfaces between components are defined.
- Additional component structuring criteria are used to ensure that the components are designed as configurable components that can be effectively mapped to physical nodes.

Designing Distributed Component-based Software Architectures (Cont…)

- 2. **Design constituent components-** Because, by definition, a simple component can execute on only one node, the internals of each simple component can be designed by means of a design method for sequential object-oriented software Architectures.

- 3. **Deploy the application-** After a distributed application has been designed, instances of it can be defined and deployed.
-
- During this stage, the component instances of the application are defined, interconnected, and mapped onto a hardware configuration consisting of distributed physical nodes.

Composite Subsystems And Components

- A composite subsystem is a component and adheres to the principle of geographical distribution.

- Thus, objects that are part of a composite subsystem must reside at the same location, but objects in different geographical locations are never in the same composite subsystem.

- A **composite subsystem** is a component that encapsulates the internal components (objects) it contains.

- The component is both a logical and a physical container, but it adds no further functionality; thus, a component's functionality is provided entirely by the part components it contains.

Design of Component Interfaces

- An **interface** specifies the externally visible operations of a class or component without revealing the internal structure (implementation) of the operations

- If different components use a component differently, it is possible to design a separate interface for each component that requires a different interface.

Example

- An example of a component that provides more than one interface is Alarm Service.
- Two interfaces from the Emergency Monitoring System will be used in the examples that follow.
- Each interface consists of one or more operations, as follows:

1. **Interface:** IAlarmService

Operations provided:

alarmRequest (**in** request, **out** alarmData)

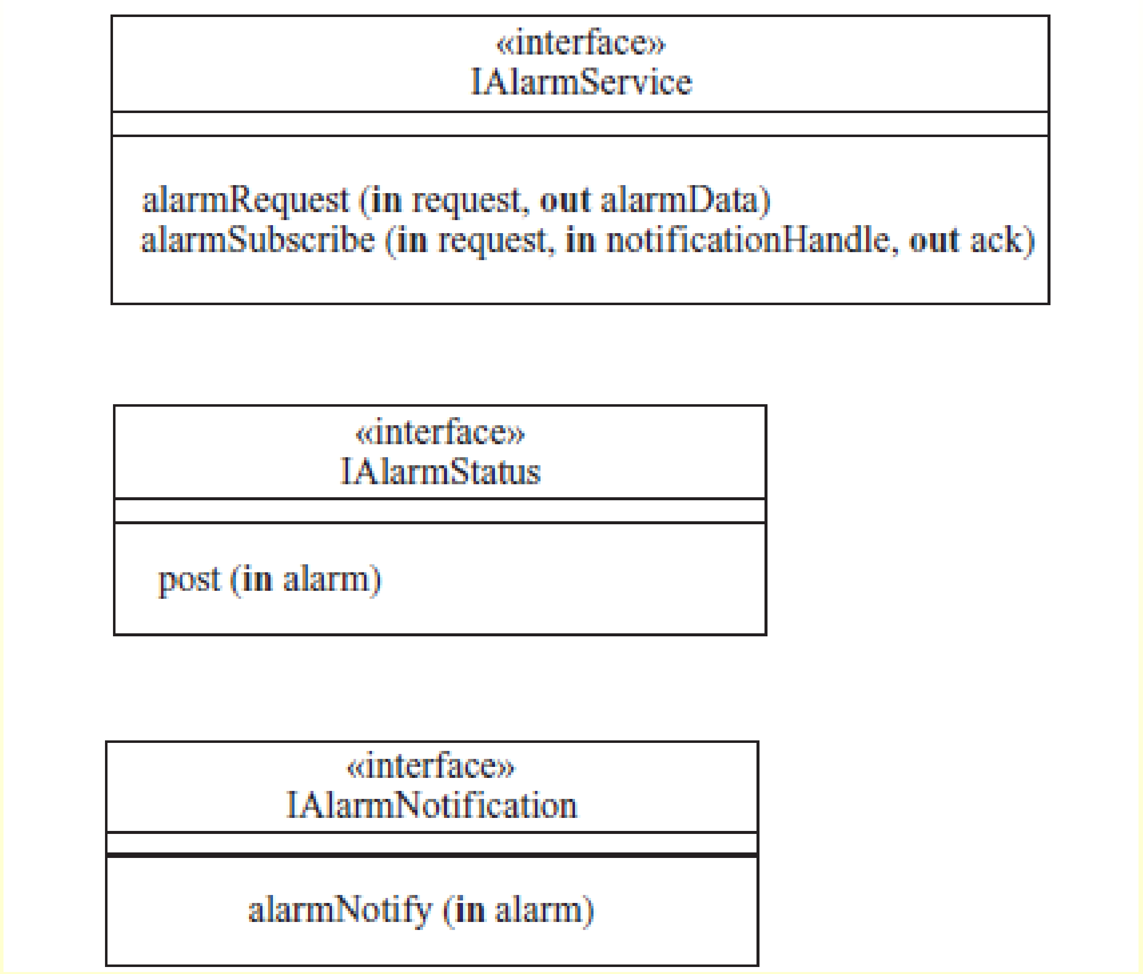
alarmSubscribe (**in** request, **in** notificationHandle, **out** ack)

2. **Interface:** IAlarmStatus

Operation provided: post (**in** alarm)

3. **Interface:** IAlarmNotification

Operation provided: alarmNotify (**in** alarm)


«interface»
IAlarmService
alarmRequest (**in** request, **out** alarmData)
alarmSubscribe (**in** request, **in** notificationHandle, **out** ack)
«interface»
IAlarmStatus
post (**in** alarm)
«interface»
IAlarmNotification
alarmNotify (**in** alarm)

Designing Concurrent and Real-Time Software Architectures

- An important activity in designing real-time software architectures is to design concurrent objects, which are referred to as concurrent tasks
- During concurrent software design, a **concurrent software architecture** is developed in which the system is structured into concurrent tasks, and the interfaces and interconnections between the concurrent tasks are defined.
- To help determine the concurrent tasks, concurrent task structuring criteria are provided to assist in mapping an object-oriented analysis model of the system to a concurrent software architecture.

Characteristics Of Real-time Systems

- Real-time systems are concurrent systems with timing constraints.
- They have widespread use in industrial, commercial, and military applications.
- The term real-time system usually refers to the whole system, including the real time application, real-time operating system, and the real-time I/O subsystem, with special-purpose device drivers to interface to the various sensors and actuators.

Characteristics Of Real-time Systems (Cont…)

- Real-time systems are often complex because they have to deal with multiple independent streams of input events and produce multiple independent outputs.
- These events have arrival rates that are often unpredictable, although they must be subject to timing constraints specified in the system requirements.
- Frequently, the order of incoming events is not predictable. Also, the input load might vary significantly and unpredictably with time.
- Real-time systems are frequently classified as hard real-time systems or soft real time systems.
- A hard real-time system has time-critical deadlines that must be met to prevent a catastrophic system failure. In a soft real-time system, missing deadlines occasionally is considered undesirable but not catastrophic, so it can be tolerated.

Control Patterns For Real-time Software Architectures

- Many real-time systems have a control function.
-
- It describes the different kinds of control patterns that could be used for this purpose: centralized control patterns, distributed control patterns, and hierarchical control patterns.

- To make the patterns applicable to component-based software architectures as well as real time software architectures, the «component» stereotype is used in these patterns.

Control Patterns For Real-time Software Architectures (Cont…)

- In the **Centralized Control** architectural pattern, there is one control component, which conceptually executes a state-chart and provides the overall control and sequencing of the system.

- The control component receives events from other components with which it interacts. These include events from various input components and user interface components that interact with the external environment – for example, through sensors that detect changes in the environment.

Control Patterns For Real-time Software Architectures (Cont…)

- An input event to a control component usually causes a state transition on its state chart, which results in one or more state-dependent actions.

- The control component uses these actions to control other components, such as output components, which output to the external environment – for example, to switch actuators on and off.

- Entity objects are also used to store any temporary data needed by the other objects.

Distributed Control Architectural Pattern

- The Distributed Control pattern contains several control components.
- Each of these components controls a given part of the system by conceptually executing a state chart.
- Control is distributed among the various control components, with no single component in overall control.
- To notify each other of important events, the control components communicate through peer-to-peer communication.

Distributed Control Architectural Pattern

- They also interact with the external environment as in the Centralized Control pattern.
- An example of the Distributed Control pattern is the control is distributed among the several distributed controller components.
- Each distributed controller executes a state machine, receiving inputs from the external environment through sensor components and controlling the external environment by sending outputs to actuator components.

- Each distributed controller communicates with the other distributed controller components by means of messages containing events.

Hierarchical Control Architectural Pattern (Cont…)

- The Hierarchical Control pattern (also known as the Multilevel Control pattern) contains several control components.

- Each component controls a given part of a system by conceptually executing a state machine. In addition, a coordinator component provides the overall system control by coordinating several control components.

- The coordinator provides high-level control by deciding the next job for each control component and communicating that information directly to the control component.

Hierarchical Control Architectural Pattern

- The coordinator also receives status information from the control components.
- One example of the Hierarchical Control pattern is the Hierarchical Controller sends high-level commands to each of the distributed controllers.
- The distributed controllers provide the low-level control, interacting with sensor and actuator components, and respond to the Hierarchical Controller when they have finished.
- They may also send progress reports to the Hierarchical Controller.