# Software Design

- Deriving a solution which satisfies software requirements

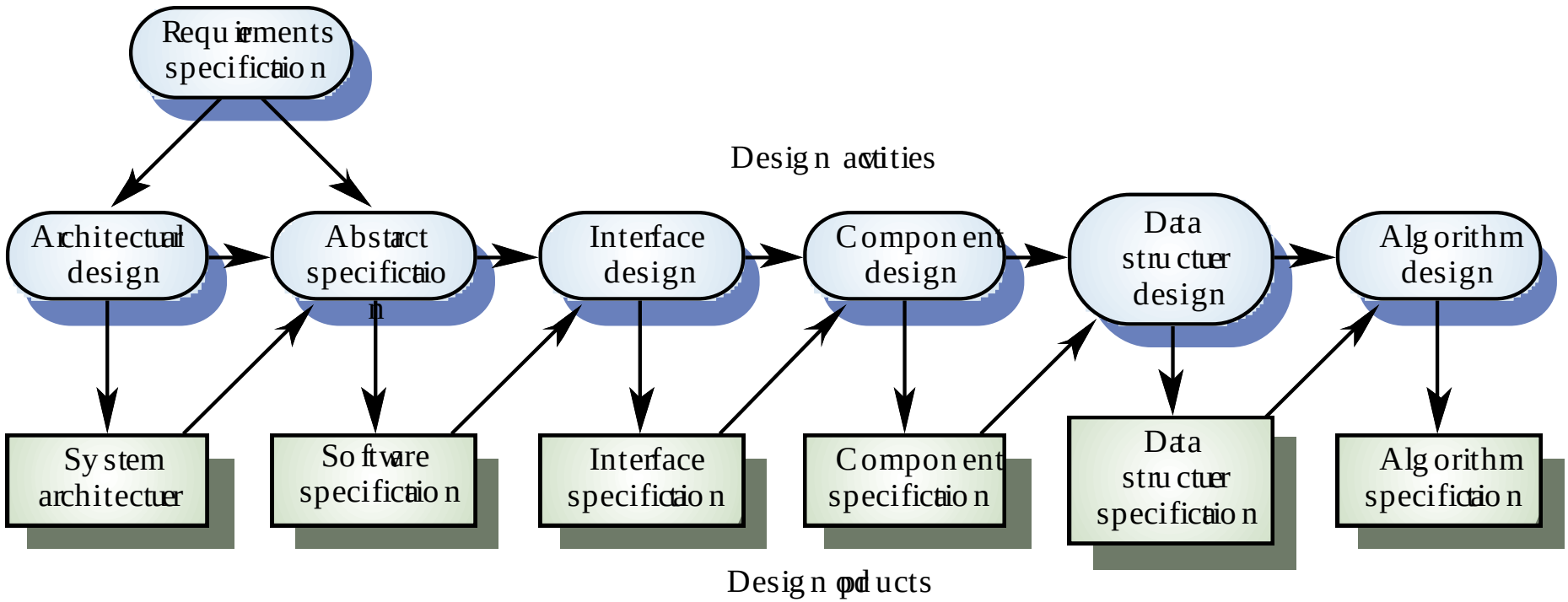
# Stages of Design

- Problem understanding
  - Look at the problem from different angles to discover the design requirements.
- Identify one or more solutions
  - Evaluate possible solutions and choose the most appropriate depending on the designer's experience and available resources.
- Describe solution abstractions
  - Use graphical, formal or other descriptive notations to describe the components of the design.
- Repeat process for each identified abstraction until the design is expressed in primitive terms.

# The Design Process

- Any design may be modelled as a directed graph made up of entities with attributes which participate in relationships.
- The system should be described at several different levels of abstraction.
- Design takes place in overlapping stages. It is artificial to separate it into distinct phases but some separation is usually necessary.

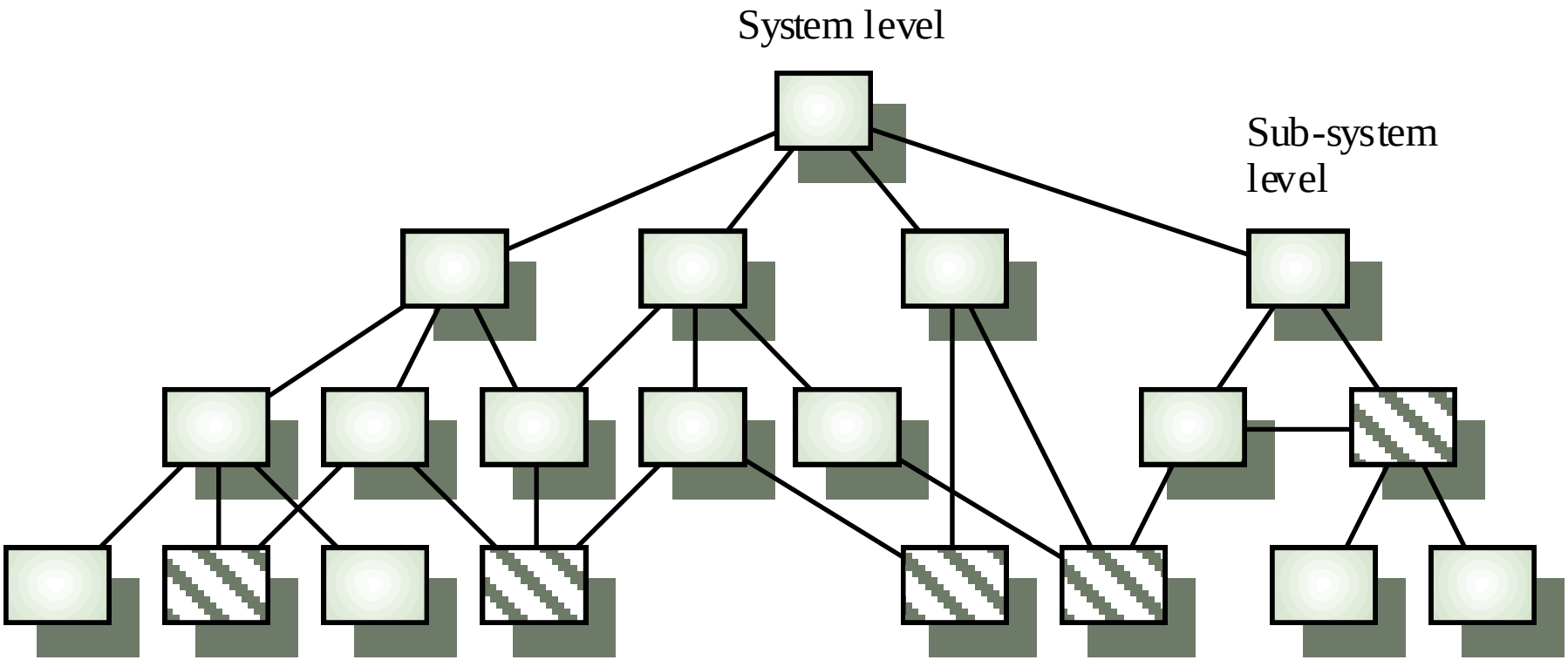
# Phases in the Design Process



# Design Phases

- ***Architectural design:*** Identify sub-systems.
- ***Abstract specification:*** Specify sub-systems.
- ***Interface design:*** Describe sub-system interfaces.
- ***Component design:*** Decompose sub-systems into components.
- ***Data structure design:*** Design data structures to hold problem data.
- ***Algorithm design:*** Design algorithms for problem functions.

# Hierarchical Design Structure



# Top-down Design

- In principle, top-down design involves starting at the uppermost components in the hierarchy and working down the hierarchy level by level.
- In practice, large systems design is never truly top-down. Some branches are designed before others. Designers reuse experience (and sometimes components) during the design process.

# Design Methods

- Structured methods are sets of notations for expressing a software design and guidelines for creating a design.
- Well-known methods include Structured Design (Yourdon), and JSD (Jackson Method).
- Can be applied successfully because the support standard notations and ensure designs follow a standard form.
- Structured methods may be supported with CASE tools.

# Method Components

- Many methods support comparable views of a system.
- A data flow view showing data transformations.
- An entity-relation view describing the logical data structures.
- A structural view showing system components and their interactions.

# Method Deficiencies

- They are guidelines rather than methods in the mathematical sense. Different designers create quite different system designs.
- They do not help much with the early, creative phase of design. Rather, they help the designer to structure and document his or her design ideas.

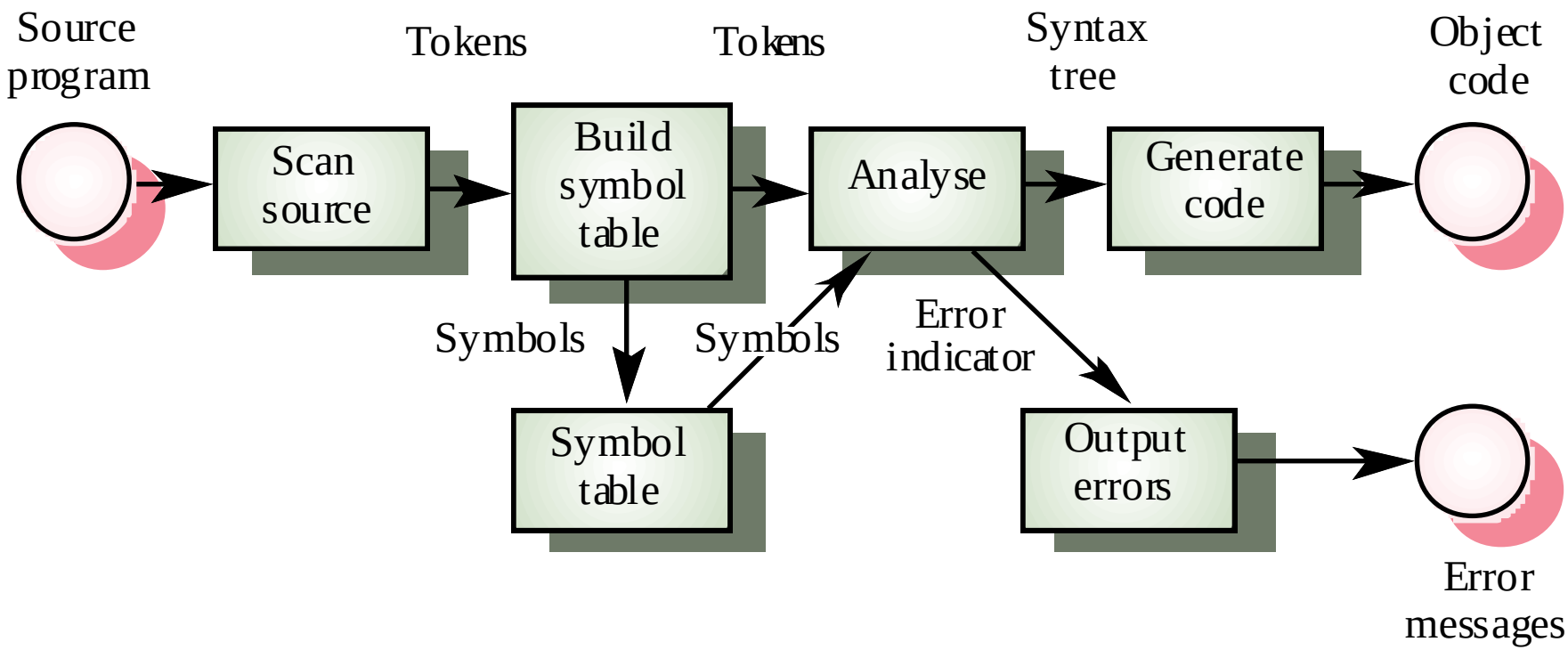
# Design Description

- ***Graphical notations:*** Used to display component relationships.
- ***Informal text:*** Natural language description.

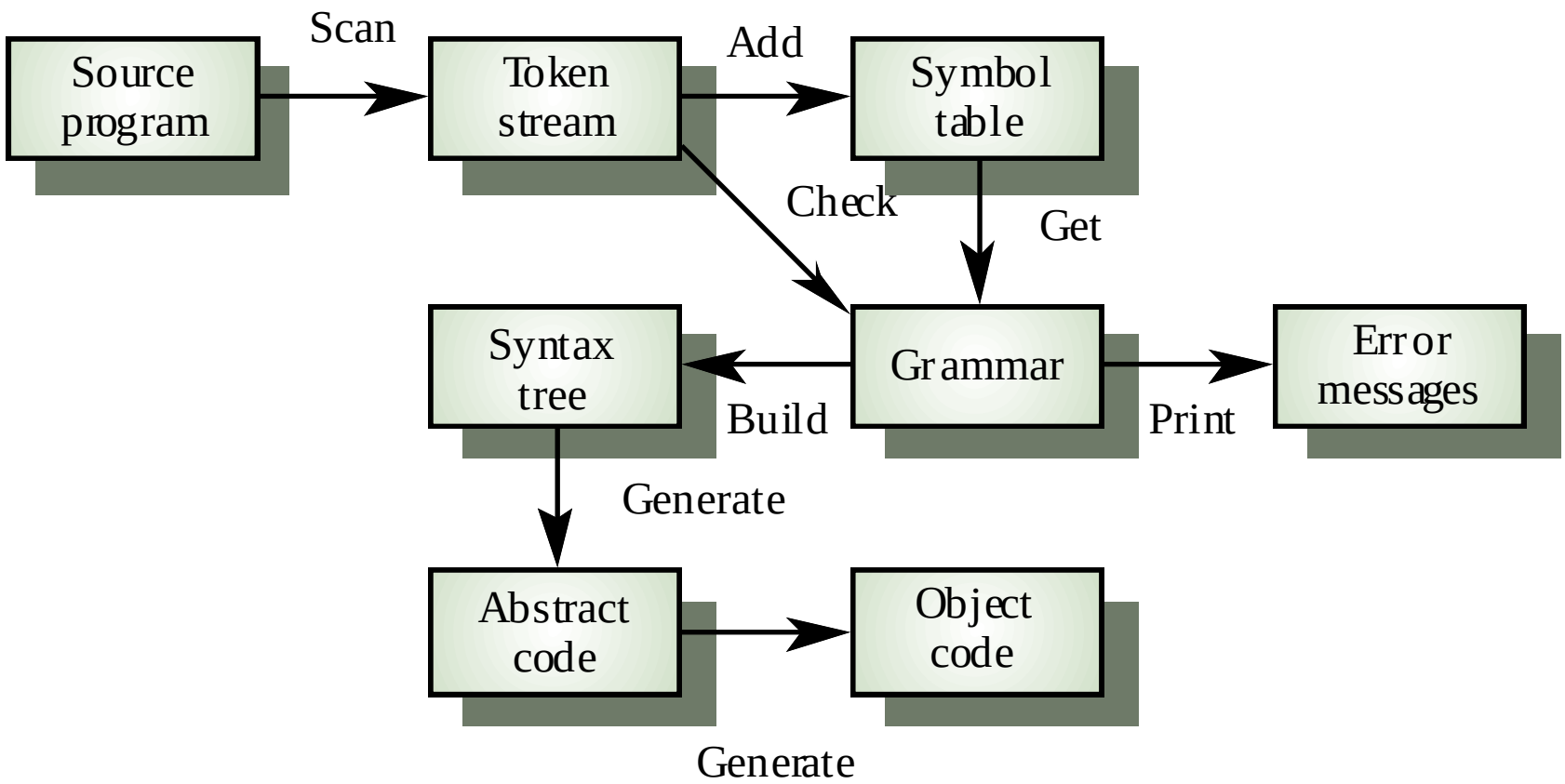
# Design Strategies

- Functional design
  - The system is designed from a functional viewpoint. The system state is centralized and shared between the functions operating on that state.
- Object-oriented design
  - The system is viewed as a collection of interacting objects. The system state is decentralized and each object manages its own state. Objects may be instances of an object class and communicate by exchanging methods.

# Functional View of a Compiler



# Object-oriented View of a Compiler



# Mixed-strategy Design

- Although it is sometimes suggested that one approach to design is superior, in practice, an object-oriented and a functional-oriented approach to design are complementary.
- Good software engineers should select the most appropriate approach for whatever sub-system is being designed.

# Design Quality

- Design quality is an elusive concept. Quality depends on specific organizational priorities.
- A “good” design may be the most efficient, the cheapest, the most maintainable, the most reliable, etc.
- The attributes discussed here are concerned with the maintainability of the design.
- Quality characteristics are equally applicable to function-oriented and object-oriented designs.

# Cohesion

- A measure of how well a component “fits together”.
- A component should implement a single logical entity or function.
- Cohesion is a desirable design component attribute as when a change has to be made, it is localized in a single cohesive component.
- Various levels of cohesion have been identified.

# Cohesion Levels

- Coincidental cohesion (weak)
  - Parts of a component are simply bundled together.
- Logical association (weak)
  - Components which perform similar functions are grouped.
- Temporal cohesion (weak)
  - Components which are activated at the same time are grouped.

# Cohesion Levels

- Communicational cohesion (medium)
  - All the elements of a component operate on the same input or produce the same output.
- Sequential cohesion (medium)
  - The output for one part of a component is the input to another part.
- Functional cohesion (strong)
  - Each part of a component is necessary for the execution of a single function.
- Object cohesion (strong)
  - Each operation provides functionality which allows object attributes to be modified or inspected.

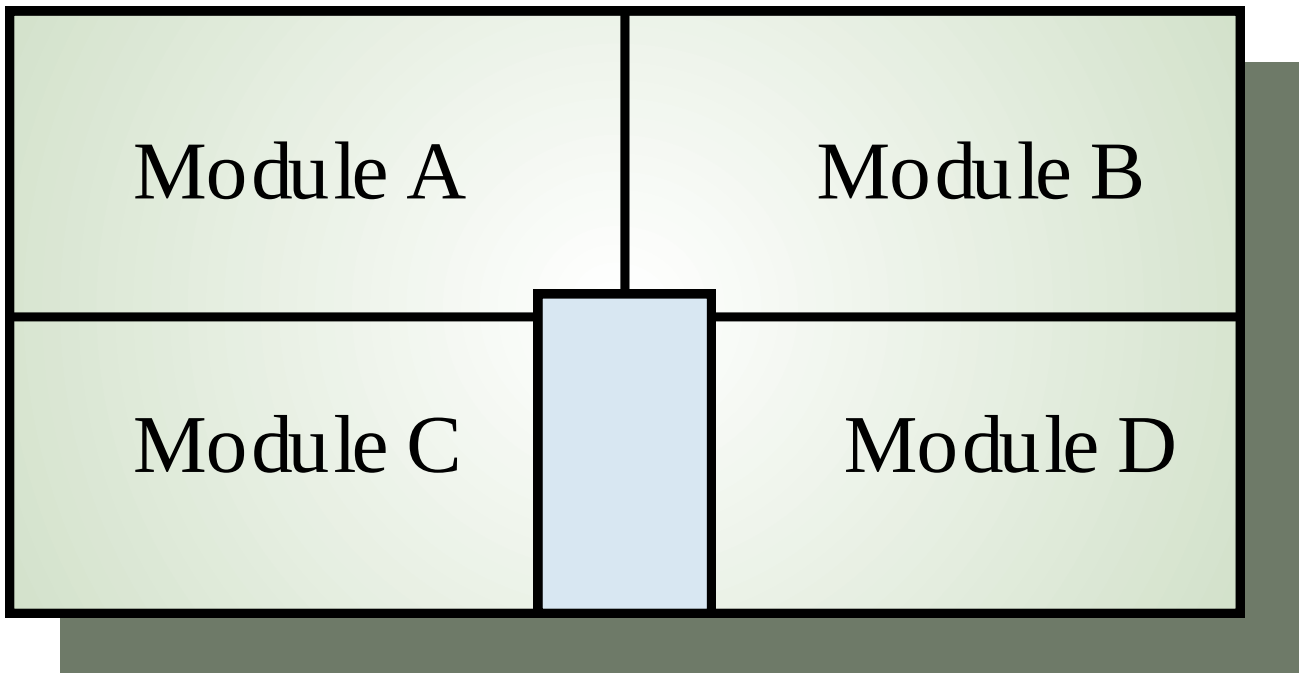
# Cohesion as a Design Attribute

- Not well-defined. Often difficult to classify cohesion.
- Inheriting attributes from super-classes weakens cohesion.
- To understand a component, the super-classes as well as the component class must be examined.
- Object class browsers assist with this process.

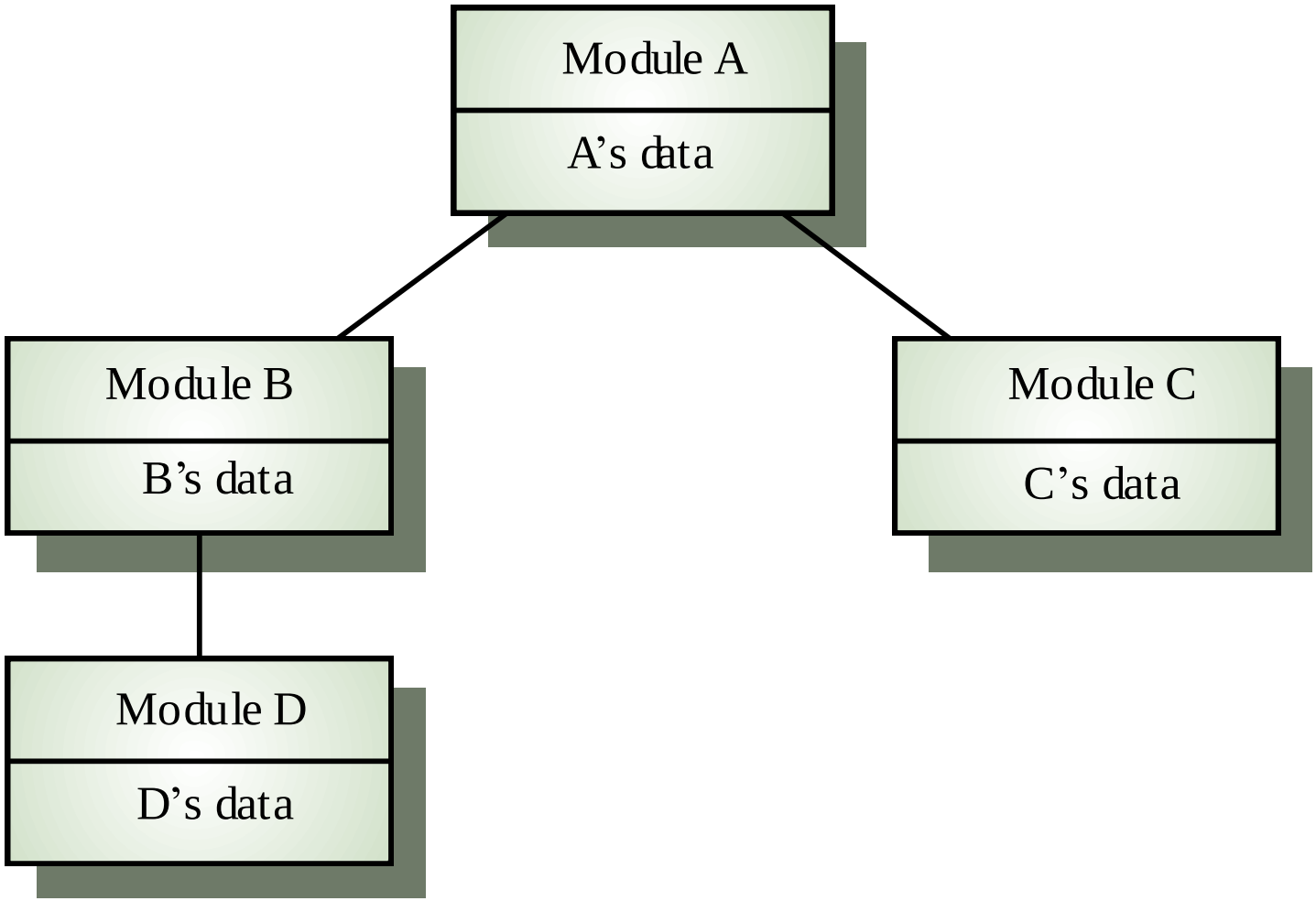
# Coupling

- A measure of the strength of the inter-connections between system components.
- Loose coupling means component changes are unlikely to affect other components.
- Shared variables or control information exchange lead to tight coupling.
- Loose coupling can be achieved by state decentralization (as in objects) and component communication via parameters or message passing.

# Tight Coupling



# Loose Coupling



# Coupling and Inheritance

- Object-oriented systems are loosely coupled because there is no shared state and objects communicate using message passing.
- However, an object class is coupled to its super-classes. Changes made to the attributes or operations in a super-class propagate to all sub-classes.

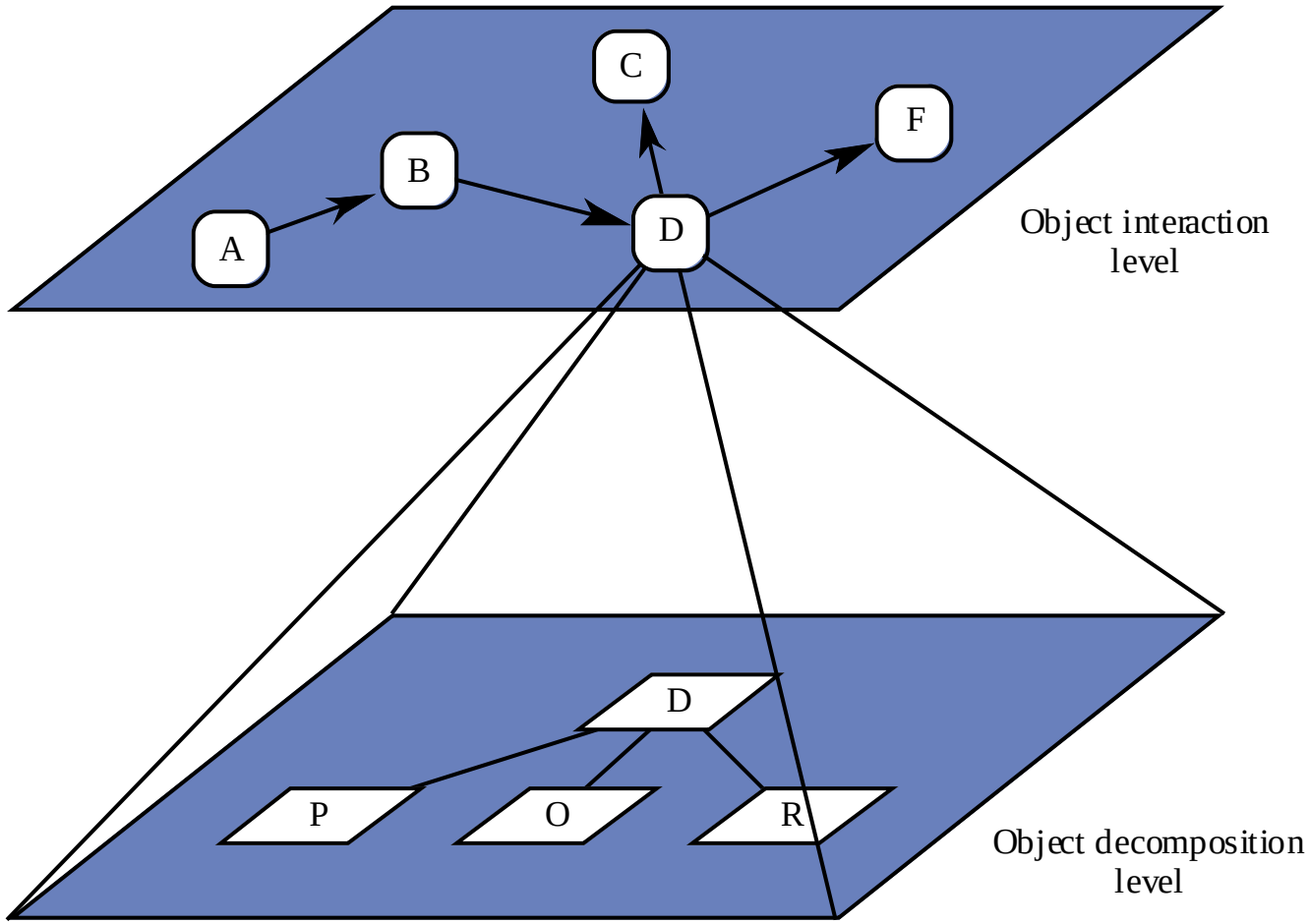
# Understandability

- Related to several component characteristics
  - Can the component be understood on its own?
  - Are meaningful names used?
  - Is the design well-documented?
  - Are complex algorithms used?
- Informally, high complexity means many relationships between different parts of the design.

# Adaptability

- A design is adaptable if:
  - Its components are loosely coupled.
  - It is well-documented and the documentation is up to date.
  - There is an obvious correspondence between design levels (design visibility).
  - Each component is a self-contained entity (tightly cohesive).
- To adapt a design, it must be possible to trace the links between design components so that change consequences can be analyzed.

# Design Traceability



# Adaptability and Inheritance

- Inheritance dramatically improves adaptability. Components may be adapted without change by deriving a sub-class and modifying that derived class.
- However, as the depth of the inheritance hierarchy increases, it becomes increasingly complex. It must be periodically reviewed and restructured.

# Architectural Design

- Establishing the overall structure of a software system

# Architectural Parallels

- Architects are the technical interface between the customer and the contractor building the system.
- A bad architectural design for a building cannot be rescued by good construction; the same is true for software.
- There are specialist types of building and software architects.
- There are schools or styles of building and software architecture.

# Sub-systems and Modules

- A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.
- A module is a system component that provides services to other components but would not normally be considered as a separate system.

# Architectural Models

- Structure, control and modular decomposition may be based on a particular model or architectural style.
- However, most systems are heterogeneous in that different parts of the system are based on different models and, in some cases, the system may follow a composite model.
- The architectural model used affects the performance, robustness, distributability and maintainability of the system.

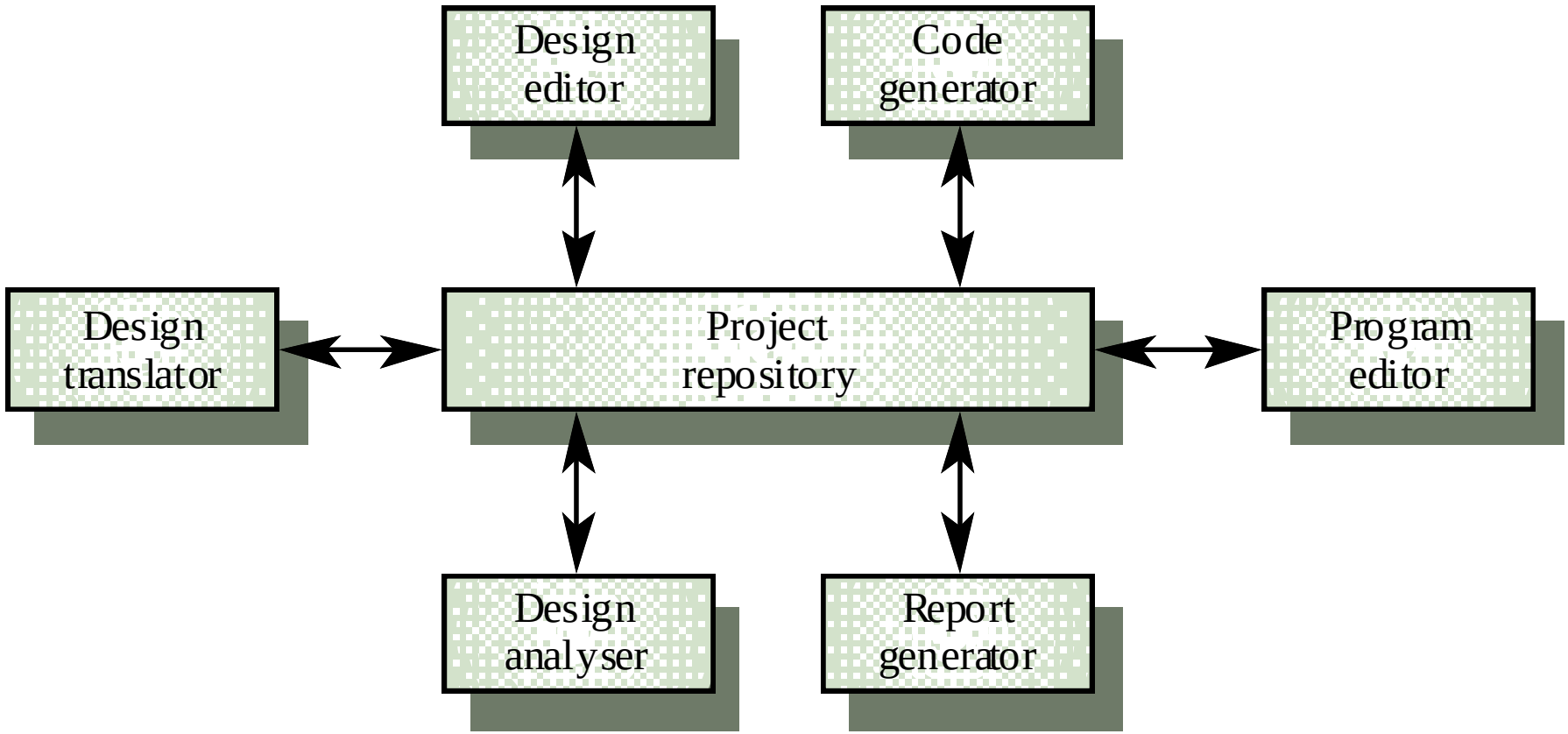
# System Structuring

- Concerned with decomposing the system into interacting sub-systems.
- The architectural design is normally expressed as a block diagram presenting an overview of the system structure.
- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed.

# The Repository Model

- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems.
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used.

# CASE Tool Set Architecture



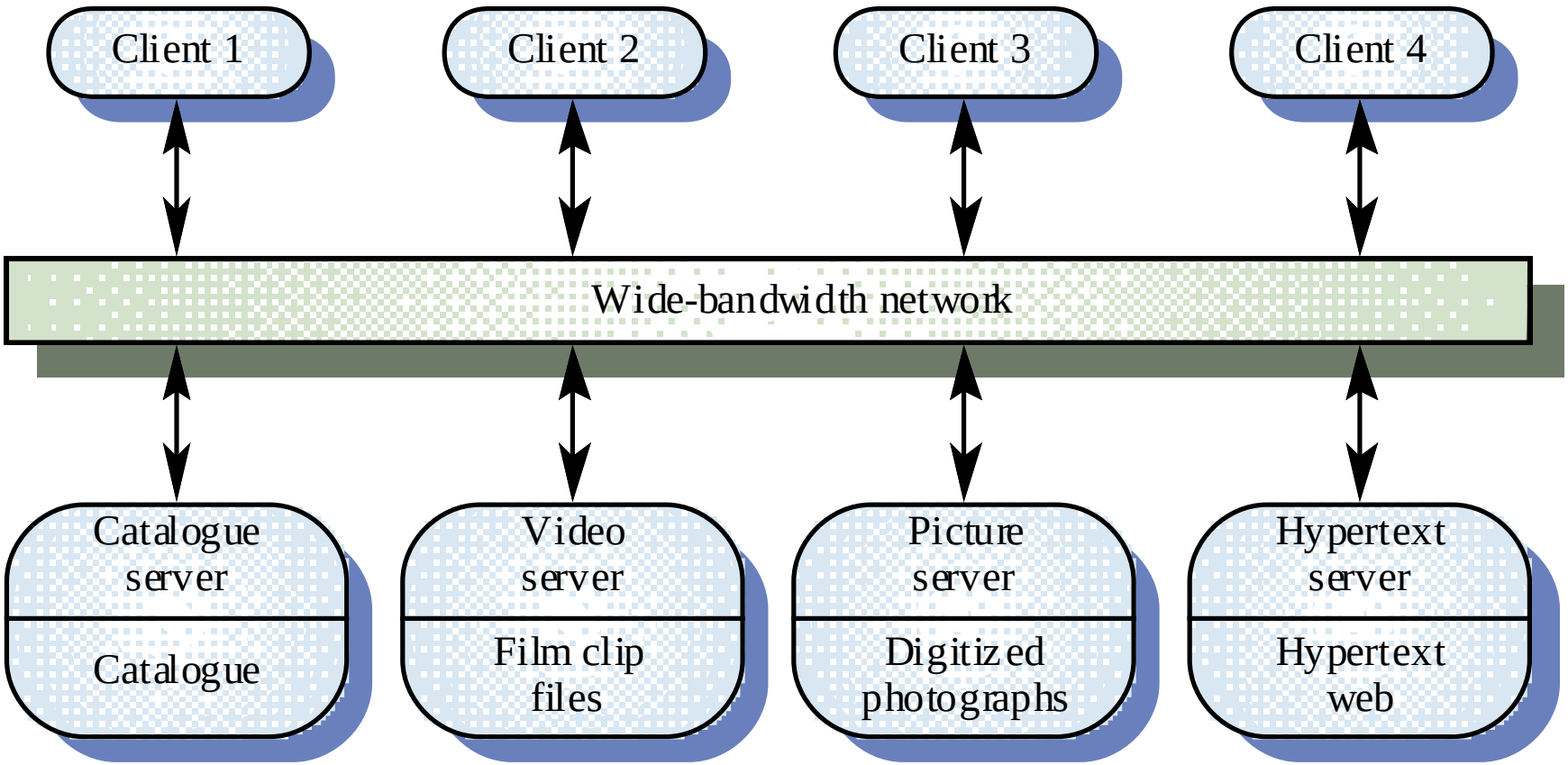
# Repository Model Characteristics

- **Advantages:**
  - Efficient way to share large amounts of data.
  - Sub-systems need not be concerned with how data is produced.
  - Centralized management e.g., backup, security, etc.
  - Sharing model is published as the repository schema.
- **Disadvantages:**
  - Sub-systems must agree on a repository data model.
  - Data evolution is difficult and expensive.
  - No scope for specific management policies.
  - Difficult to distribute efficiently.

# Client-server Architecture

- Distributed system model which shows how data and processing is distributed across a range of components.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

# Film and Picture Library



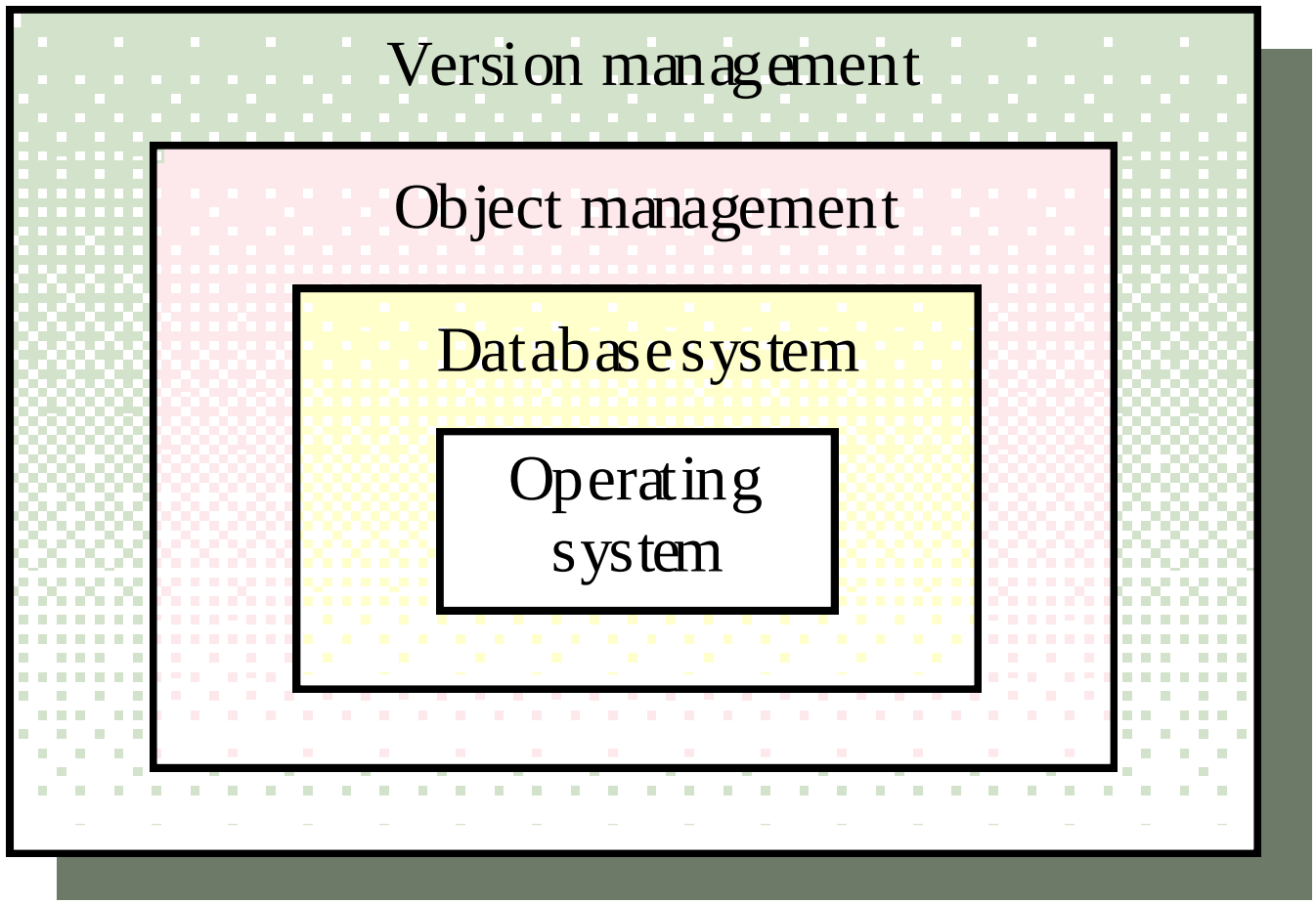
# Client-server Characteristics

- **Advantages:**
  - Distribution of data is straightforward.
  - Makes effective use of networked systems.
  - Easy to add new servers or upgrade existing servers.
- **Disadvantages:**
  - No shared data model so sub-systems may use different data organizations.
  - Redundant management in each server.
  - No central register of names and services - it may be hard to find out what servers and services are available.

# Abstract Machine Model

- Used to model the interfacing of sub-systems.
- Organizes the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often difficult to structure systems in this way.

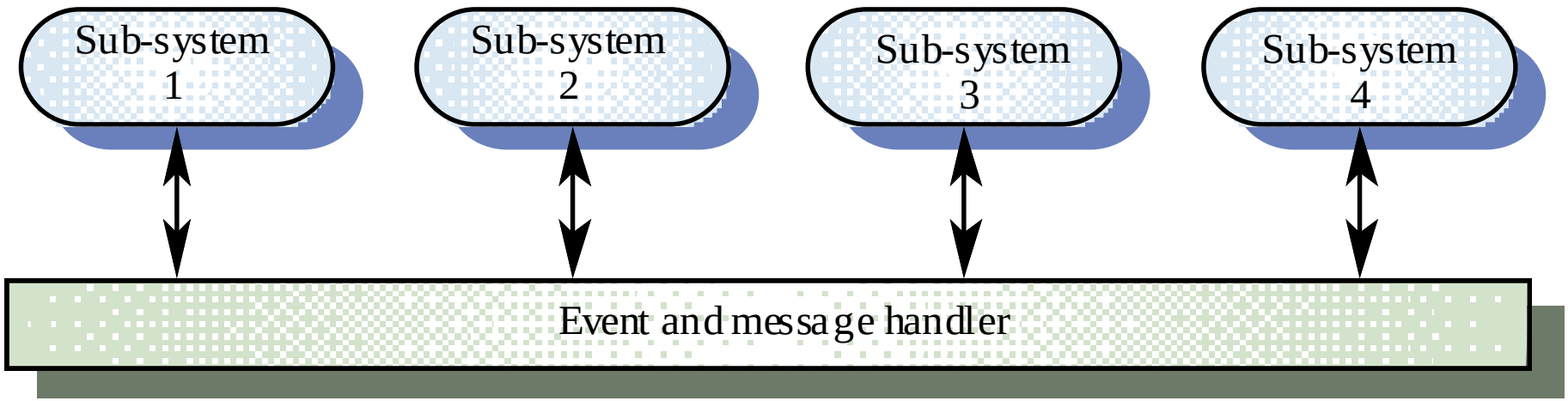
# Version Management System



# Broadcast Model

- Effective in integrating sub-systems on different computers in a network.
- Sub-systems register an interest in specific events. When these occur, control is transferred to the sub-system which can handle the event.
- Control policy is not embedded in the event and message handler. Sub-systems decide on events of interest to them.
- However, sub-systems don’t know if or when an event will be handled.

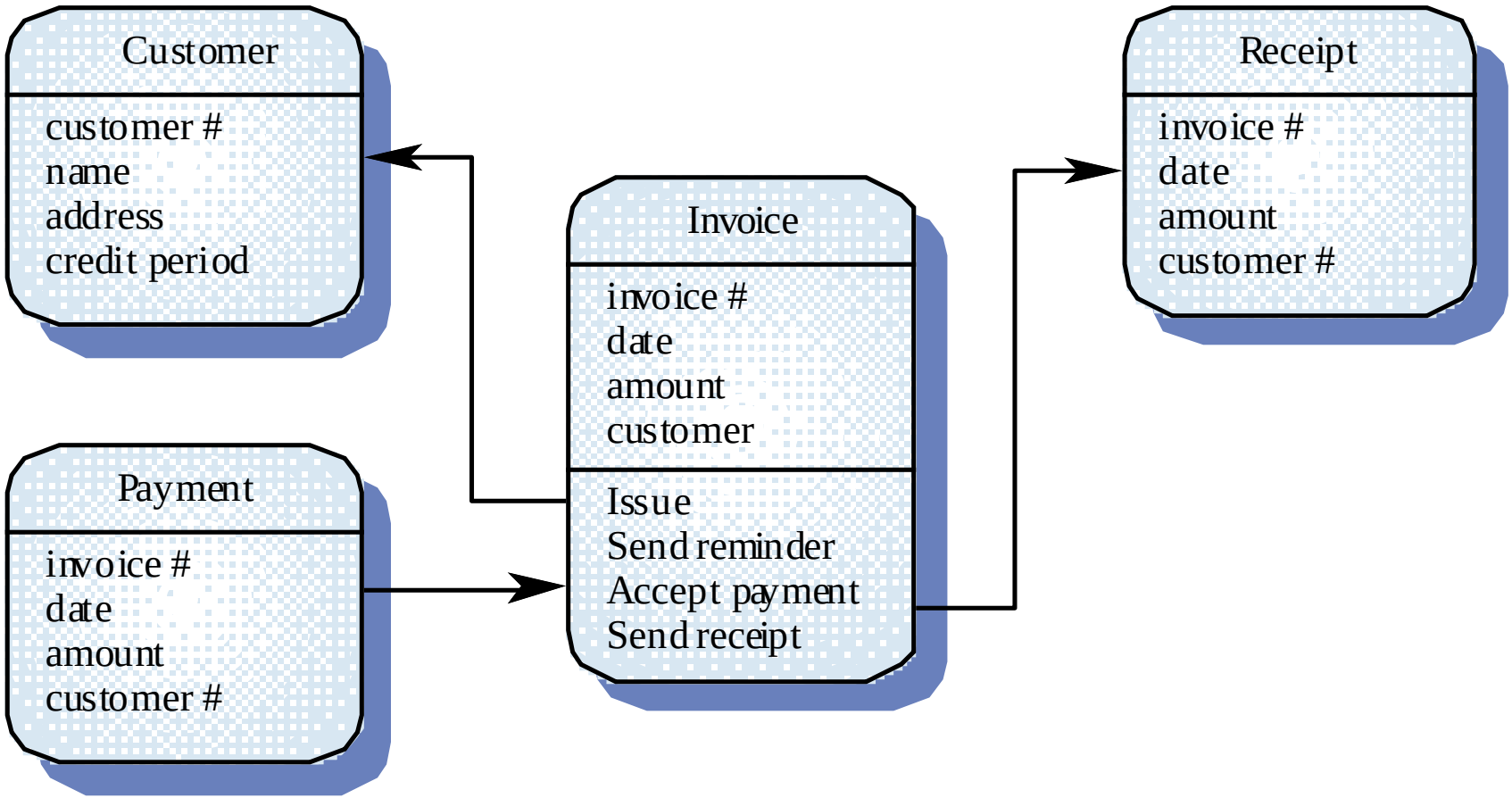
# Selective Broadcasting



# Object Models

- Structure the system into a set of loosely coupled objects with well-defined interfaces.
- Object-oriented decomposition is concerned with identifying object classes, their attributes and operations.
- When implemented, objects are created from these classes and some control model used to coordinate object operations.

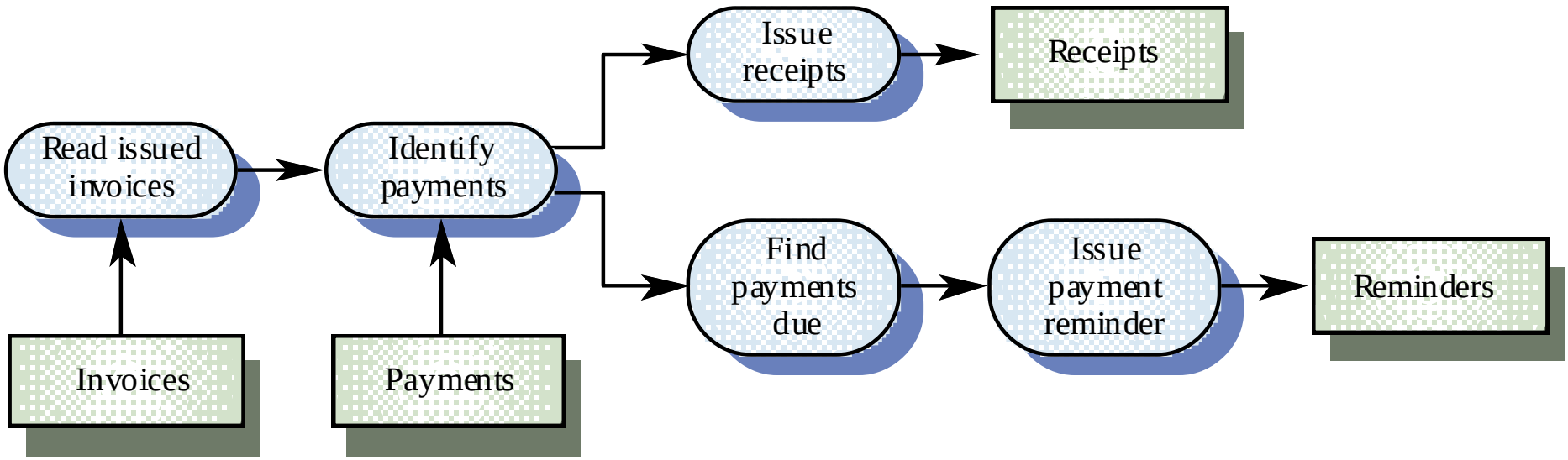
# Invoice Processing System



# Pipe and Filter Models

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

# Invoice Processing System



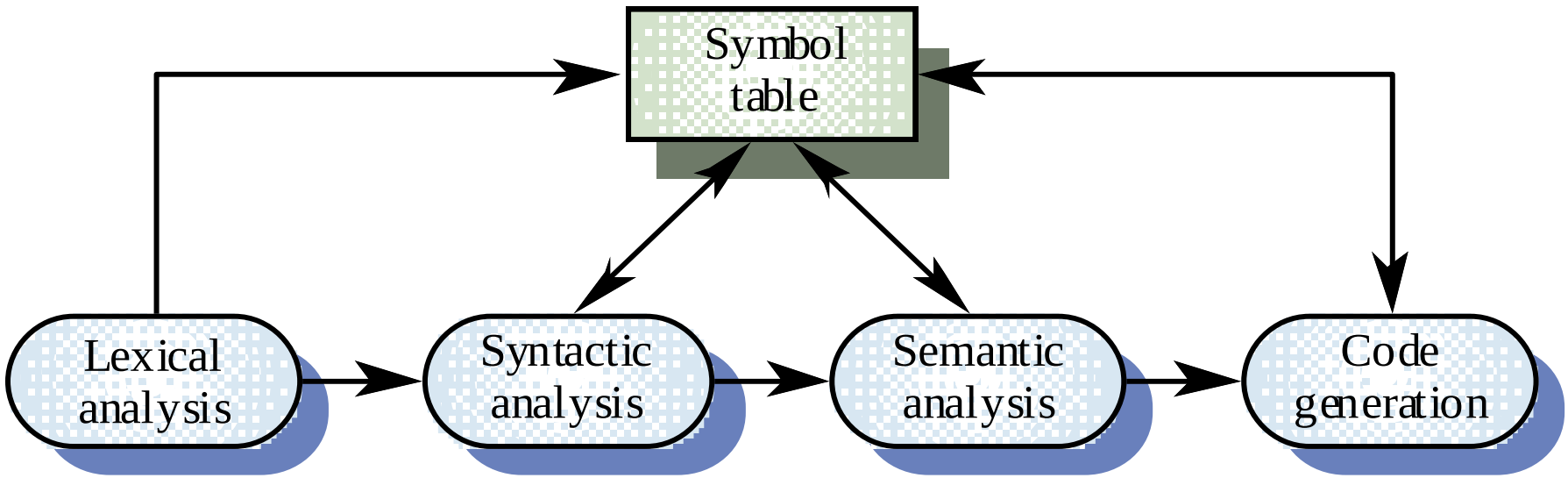
# Domain-specific Architectures

- Architectural models which are specific to some application domain.
- Two types of domain-specific model
  - **Generic models** which are abstractions from a number of real systems and which encapsulate the principal characteristics of these systems.
  - **Reference models** which are more abstract, idealized model. Provide a means of information about that class of system and of comparing different architectures.
- Generic models are usually bottom-up models.
- Reference models are top-down models.

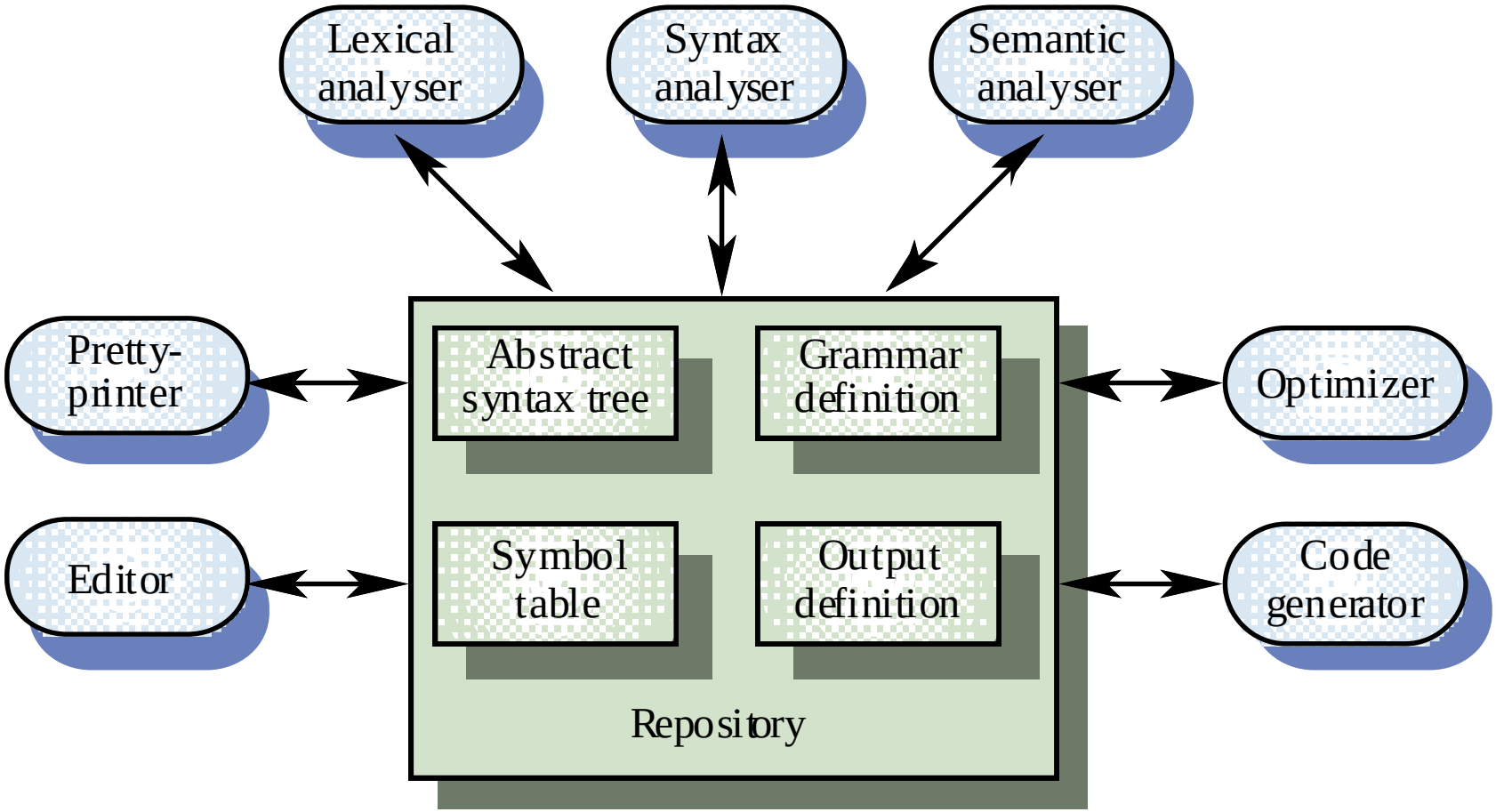
# Generic Models

- Compiler model is a well-known example although other models exist in more specialized application domains.
  - Lexical analyzer
  - Symbol table
  - Syntax analyzer
  - Syntax tree
  - Semantic analyzer
  - Code generator
- Generic compiler model may be organized according to different architectural models.

# Compiler Model



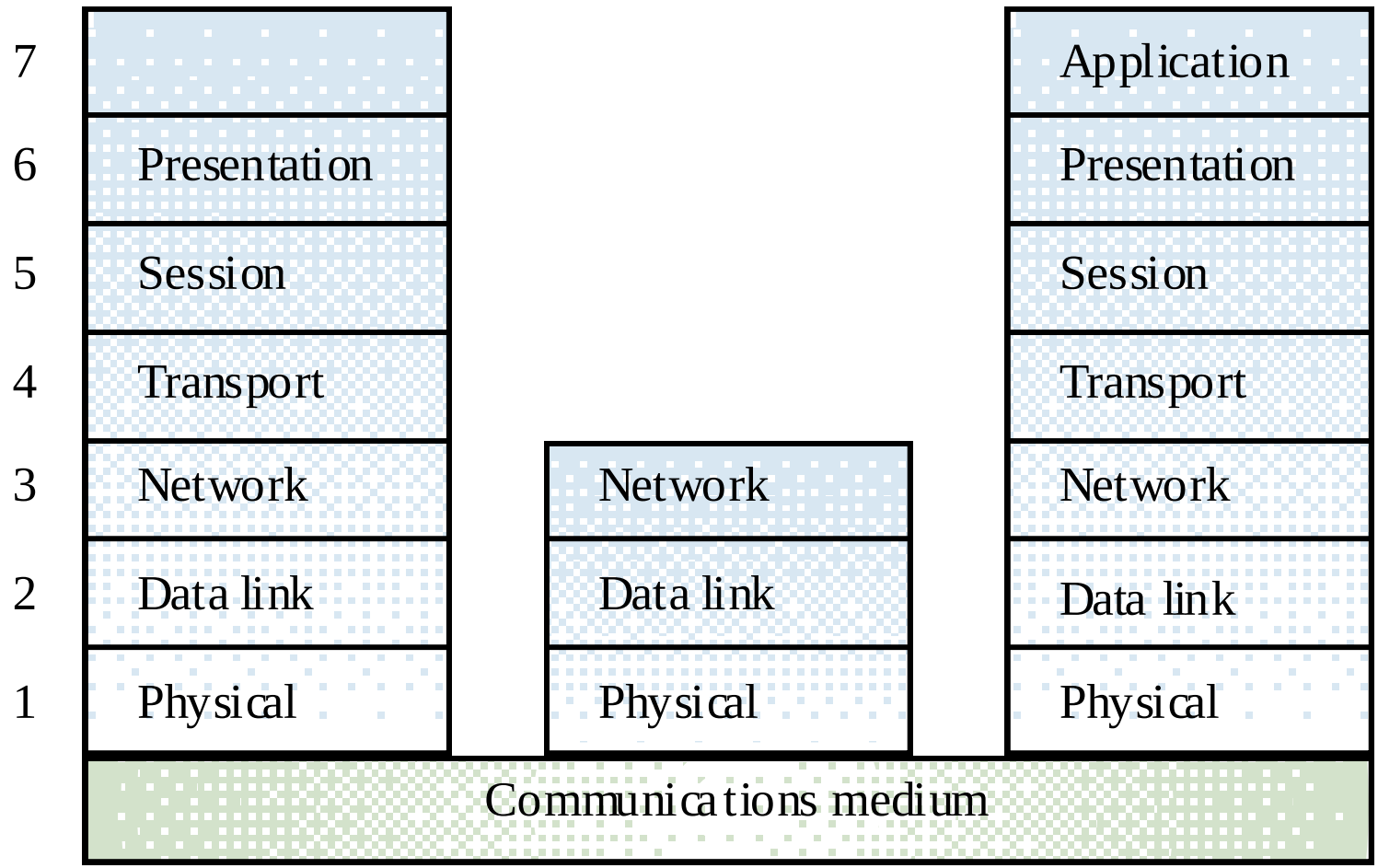
# Language Processing System



# Reference Architectures

- Reference models are derived from a study of the application domain rather than from existing systems.
- May be used as a basis for system implementation or to compare different systems. It acts as a standard against which systems can be evaluated.
- OSI model is a layered model for communication systems.

# OSI Reference Model



# Distributed Systems Architectures

Architectural design for software that executes on more than one processor

# Distributed systems

- Virtually all large computer-based systems are now distributed systems
- Information processing is distributed over several computers rather than confined to a single machine
- Distributed software engineering is now very important

# System types

- Personal systems that are not distributed and that are designed to run on a personal computer or workstation.
- Embedded systems that run on a single processor or on an integrated group of processors.
- Distributed systems where the system software runs on a loosely integrated group of cooperating processors linked by a network.

# Distributed system characteristics

- Resource sharing
- Openness
- Concurrency
- Scalability
- Fault tolerance
- Transparency

# Distributed system disadvantages

- Complexity
- Security
- Manageability
- Unpredictability

| Design issue | Description |
|---|---|
| *Resource identification* | The resources in a distributed system are spread across different computers and a naming scheme has to be devised so that users can discover and refer to the resources that they need. An example of such a naming scheme is the URL (Uniform Resource Locator) that is used to identify WWW pages. If a meaningful and universally understood identification scheme is not used then many of these resources will be inaccessible to system users. |
| *Communications* | The universal availability of the Internet and the efficient implementation of Internet TCP/IP communication protocols means that, for most distributed systems, these are the most effective way for the computers to communicate. However, where there are specific requirements for performance, reliability etc. alternative approaches to communications may be used. |
| *Quality of service* | The quality of service offered by a system reflects its performance, availability and reliability. It is affected by a number of factors such as the allocation of processes to processes in the system, the distribution of resources across the system, the network and the system hardware and the adaptability of the system. |
| *Software architectures* | The software architecture describes how the application functionality is distributed over a number of logical components and how these components are distributed across processors. Choosing the right architecture for an application is essential to achieve the desired quality of service. |

Issues in distributed system design

# Distributed systems archiectures

- Client-server architectures
  - Distributed services which are called on by clients. Servers that provide services are treated differently from clients that use services
- Distributed object architectures
  - No distinction between clients and servers. Any object on the system may provide and use services from other objects

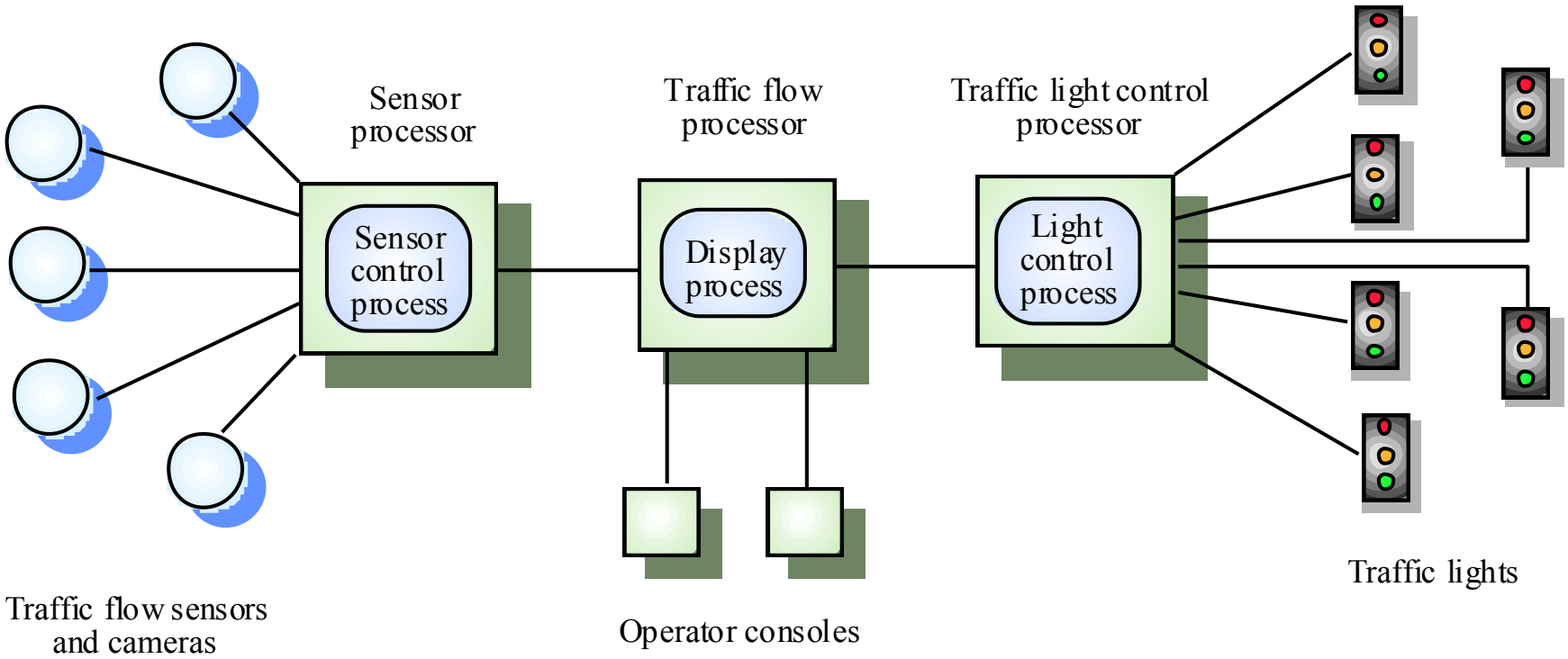
# Middleware

- Software that manages and supports the different components of a distributed system. In essence, it sits in the *middle* of the system
- Middleware is usually off-the-shelf rather than specially written software
- Examples
  - Transaction processing monitors
  - Data converters
  - Communication controllers

# Multiprocessor architectures

- Simplest distributed system model
- System composed of multiple processes which may (but need not) execute on different processors
- Architectural model of many large real-time systems
- Distribution of process to processor may be pre-ordered or may be under the control of a despatcher

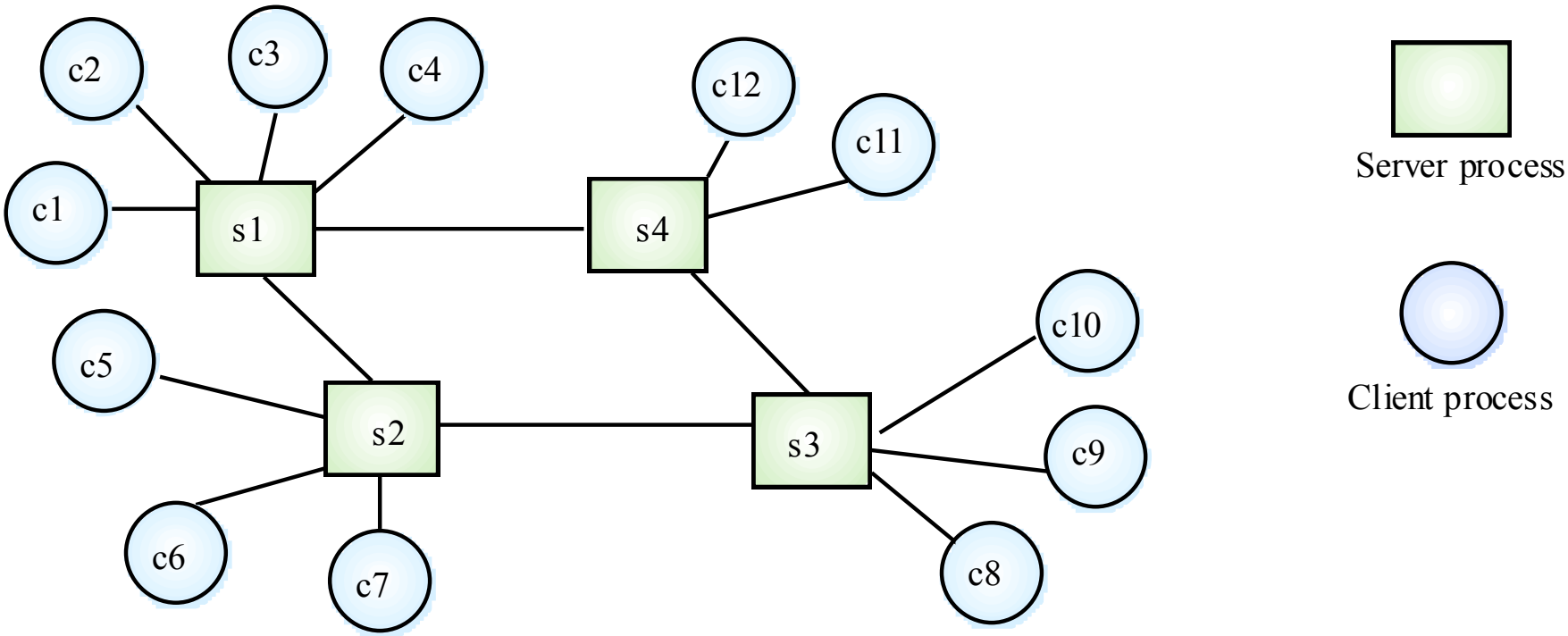
# A multiprocessor traffic control system



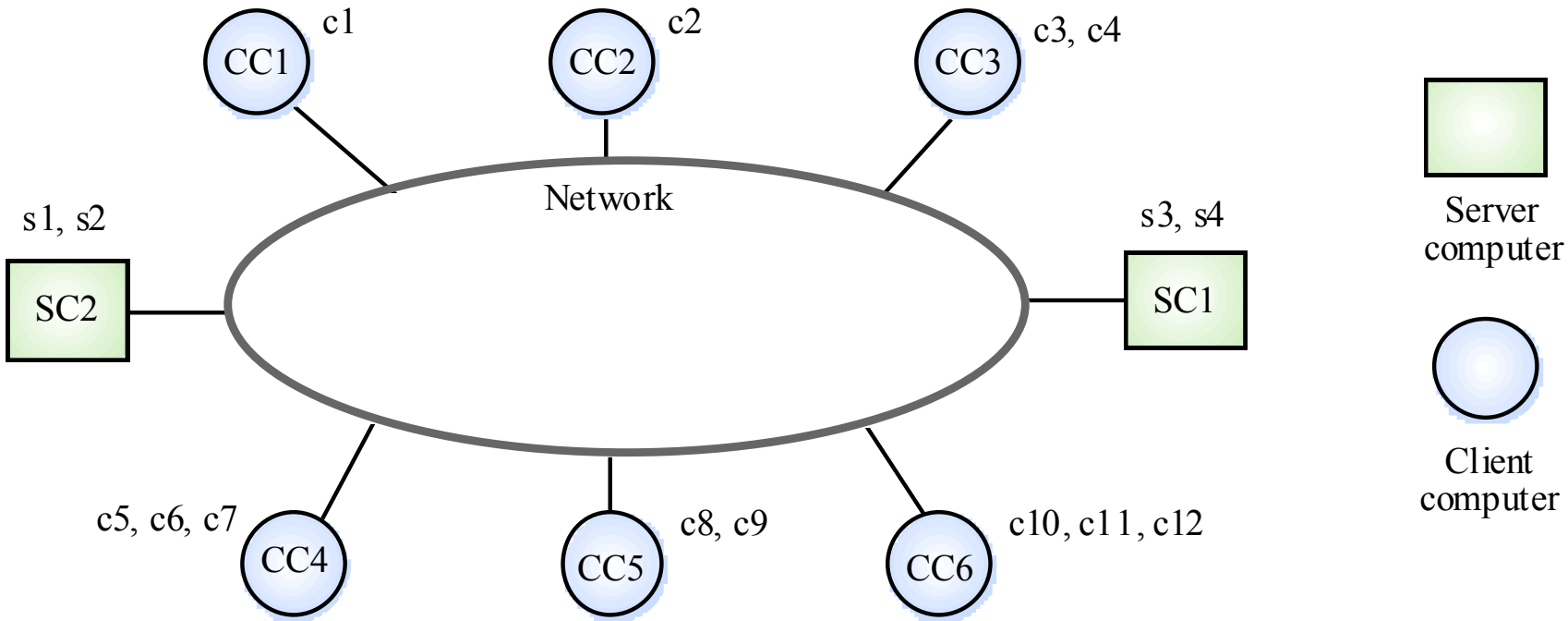
# Client-server architectures

- The application is modelled as a set of services that are provided by servers and a set of clients that use these services
- Clients know of servers but servers need not know of clients
- Clients and servers are logical processes
- The mapping of processors to processes is not necessarily 1 : 1

# A client-server system



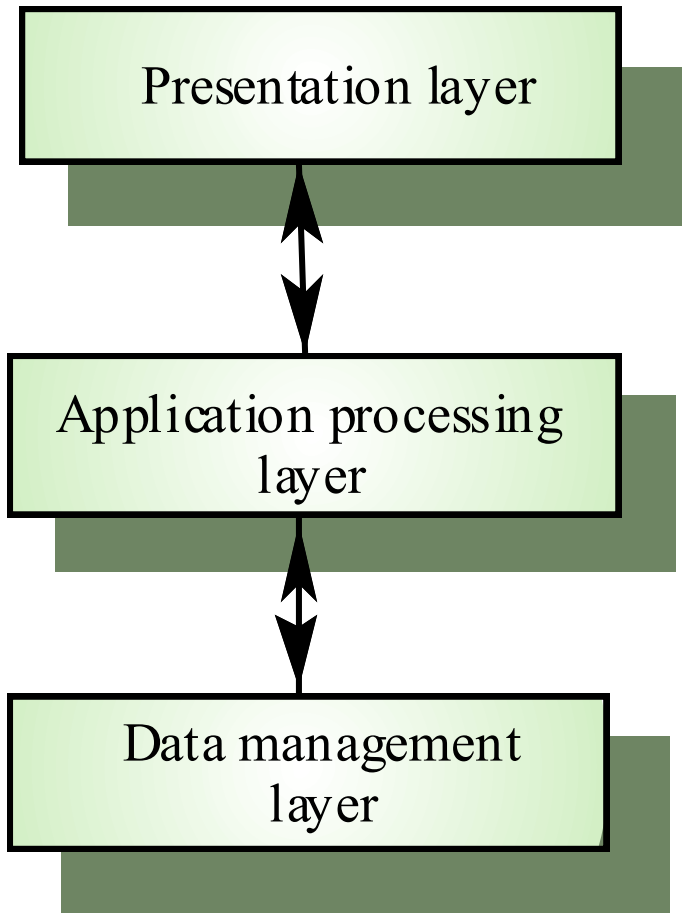
# Computers in a C/S network



# Layered application architecture

- Presentation layer
  - Concerned with presenting the results of a computation to system users and with collecting user inputs
- Application processing layer
  - Concerned with providing application specific functionality e.g., in a banking system, banking functions such as open account, close account, etc.
- Data management layer
  - Concerned with managing the system databases

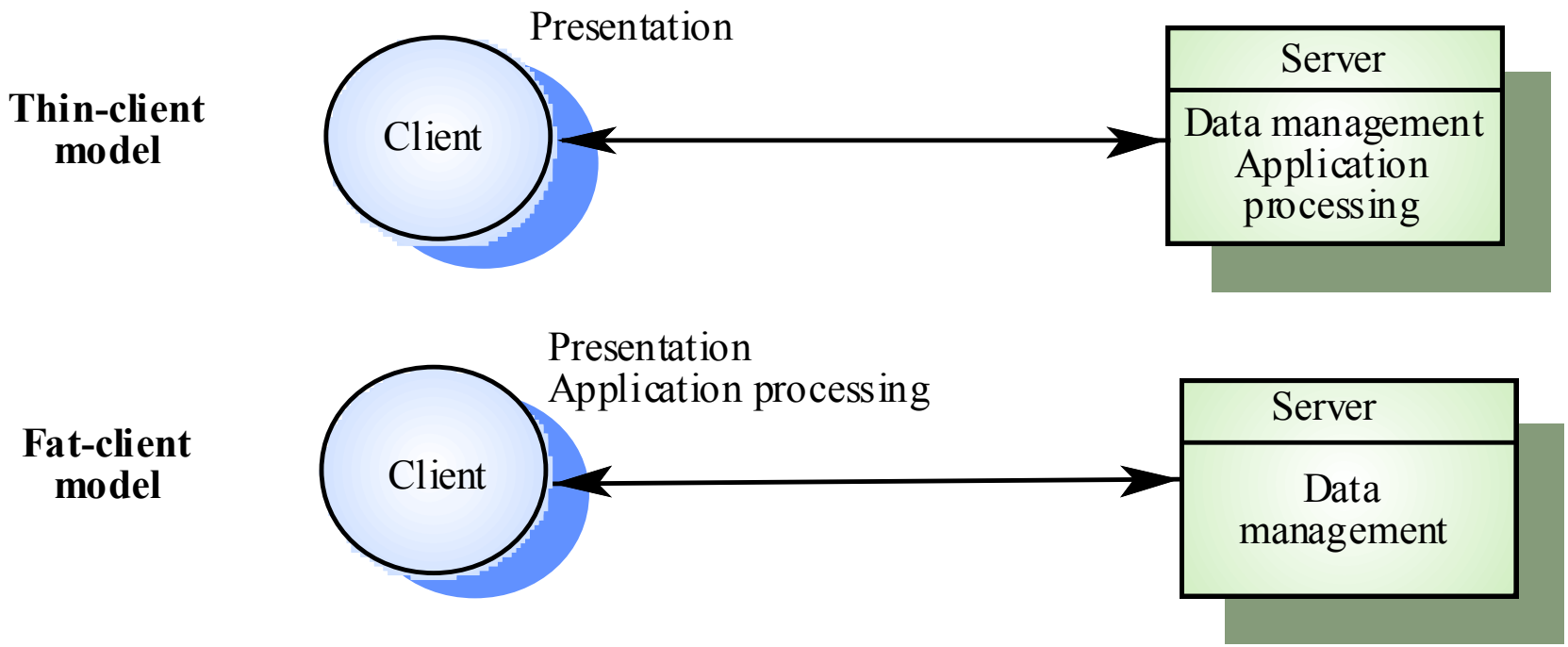
# Application layers



# Thin and fat clients

- *Thin-client model*
  - In a thin-client model, all of the application processing and data management is carried out on the server. The client is simply responsible for running the presentation software.
- *Fat-client model*
  - In this model, the server is only responsible for data management. The software on the client implements the application logic and the interactions with the system user.

# Thin and fat clients



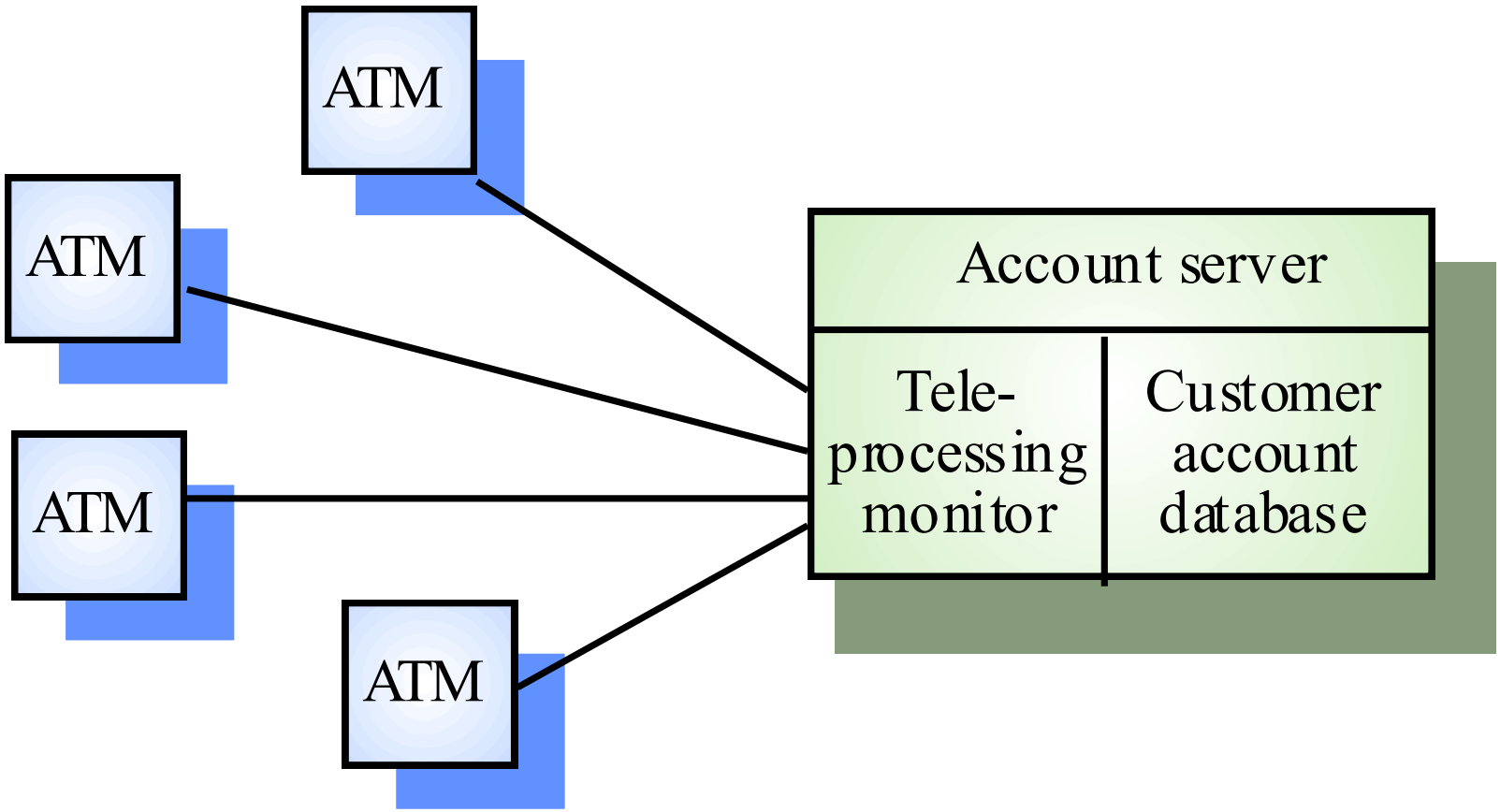
# Thin client model

- Used when legacy systems are migrated to client server architectures.
  - The legacy system acts as a server in its own right with a graphical interface implemented on a client
- A major disadvantage is that it places a heavy processing load on both the server and the network

# Fat client model

- More processing is delegated to the client as the application processing is locally executed
- Most suitable for new C/S systems where the capabilities of the client system are known in advance
- More complex than a thin client model especially for management. New versions of the application have to be installed on all clients

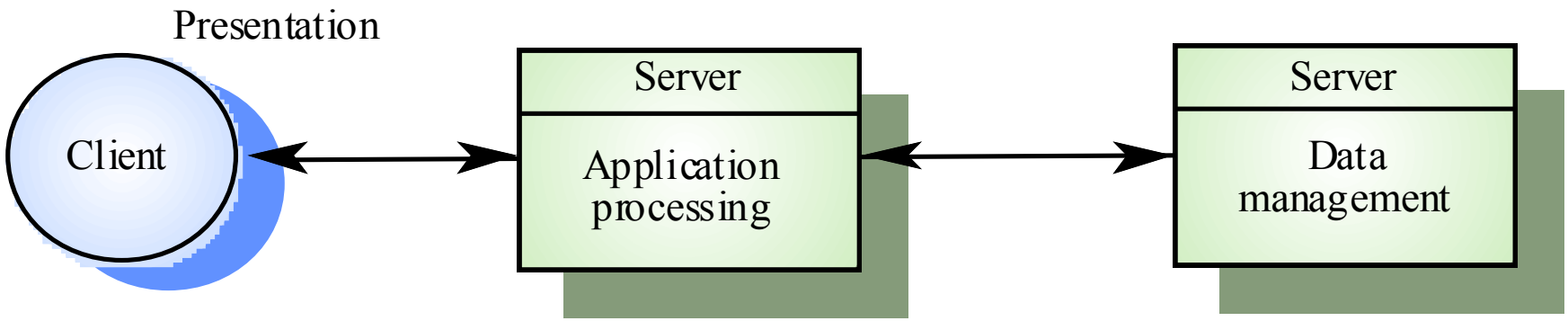
# A client-server ATM system



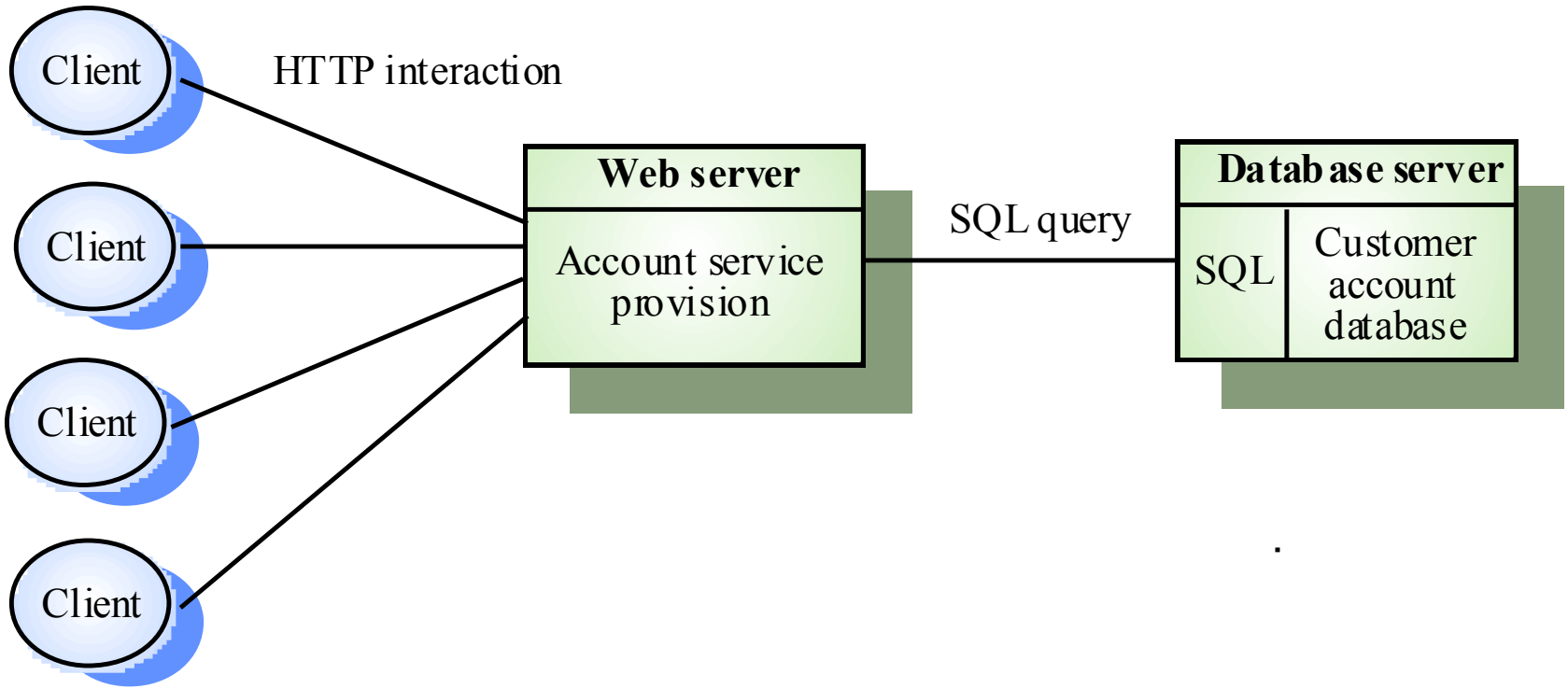
# Three-tier architectures

- In a three-tier architecture, each of the application architecture layers may execute on a separate processor
- Allows for better performance than a thin-client approach and is simpler to manage than a fat-client approach
- A more scalable architecture - as demands increase, extra servers can be added

# A 3-tier C/S architecture



# An internet banking system



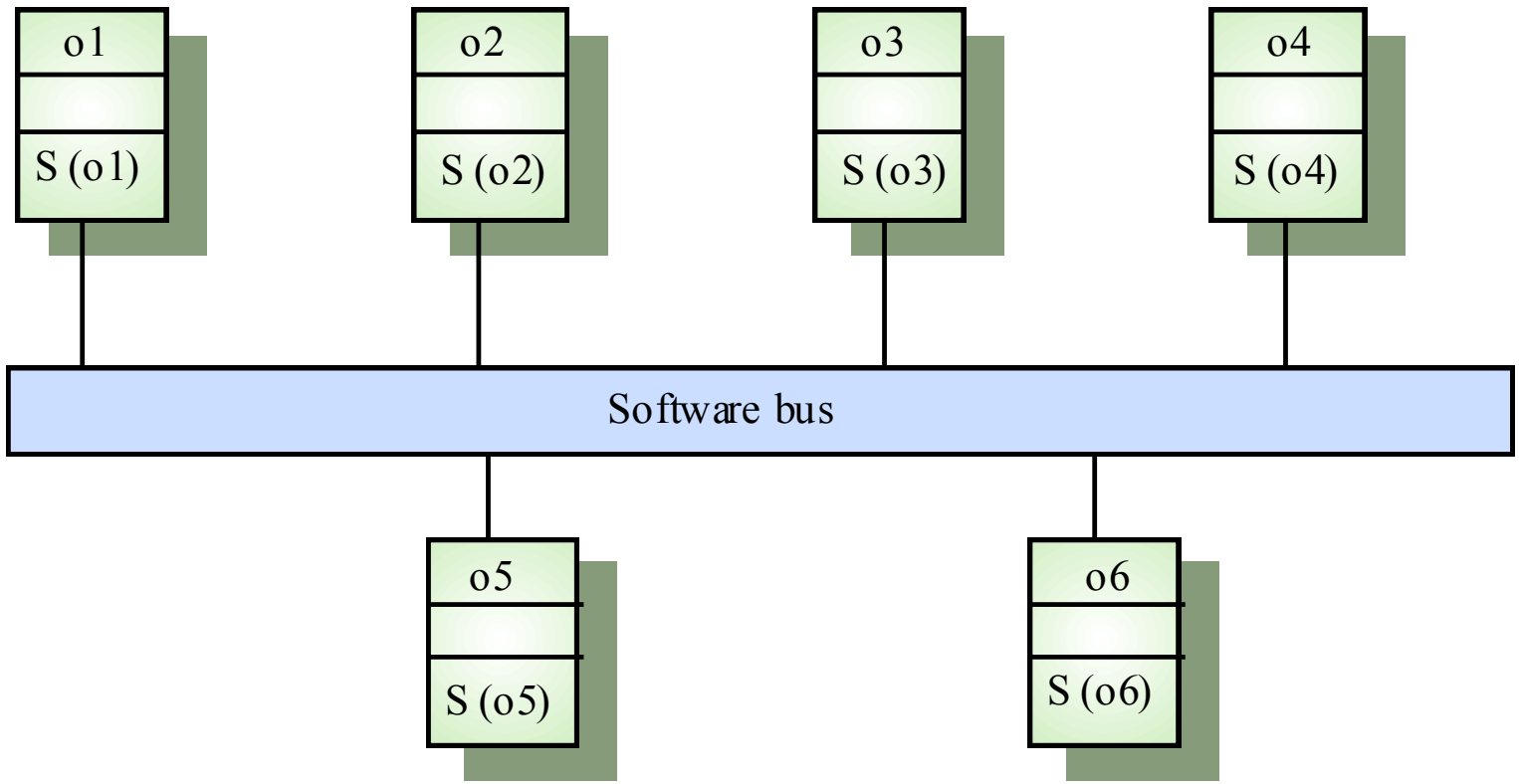
# Use of C/S architectures

| Architecture | Applications |
|---|---|
| Two-tier C/S architecture with thin clients | Legacy system applications where separating application processing and data management is impractical<br>Computationally-intensive applications such as compilers with little or no data management<br>Data-intensive applications (browsing and querying) with little or no application processing. |
| Two-tier C/S architecture with fat clients | Applications where application processing is provided by COTS (e.g. Microsoft Excel) on the client<br>Applications where computationally-intensive processing of data (e.g. data visualisation) is required.<br>Applications with relatively stable end-user functionality used in an environment with well-established system management |
| Three-tier or multi-tier C/S architecture | Large scale applications with hundreds or thousands of clients<br>Applications where both the data and the application are volatile.<br>Applications where data from multiple sources are integrated |

# Distributed object architectures

- There is no distinction in a distributed object architectures between clients and servers
- Each distributable entity is an object that provides services to other objects and receives services from other objects
- Object communication is through a middleware system called an object request broker (software bus)
- However, more complex to design than C/S systems

# Distributed object architecture



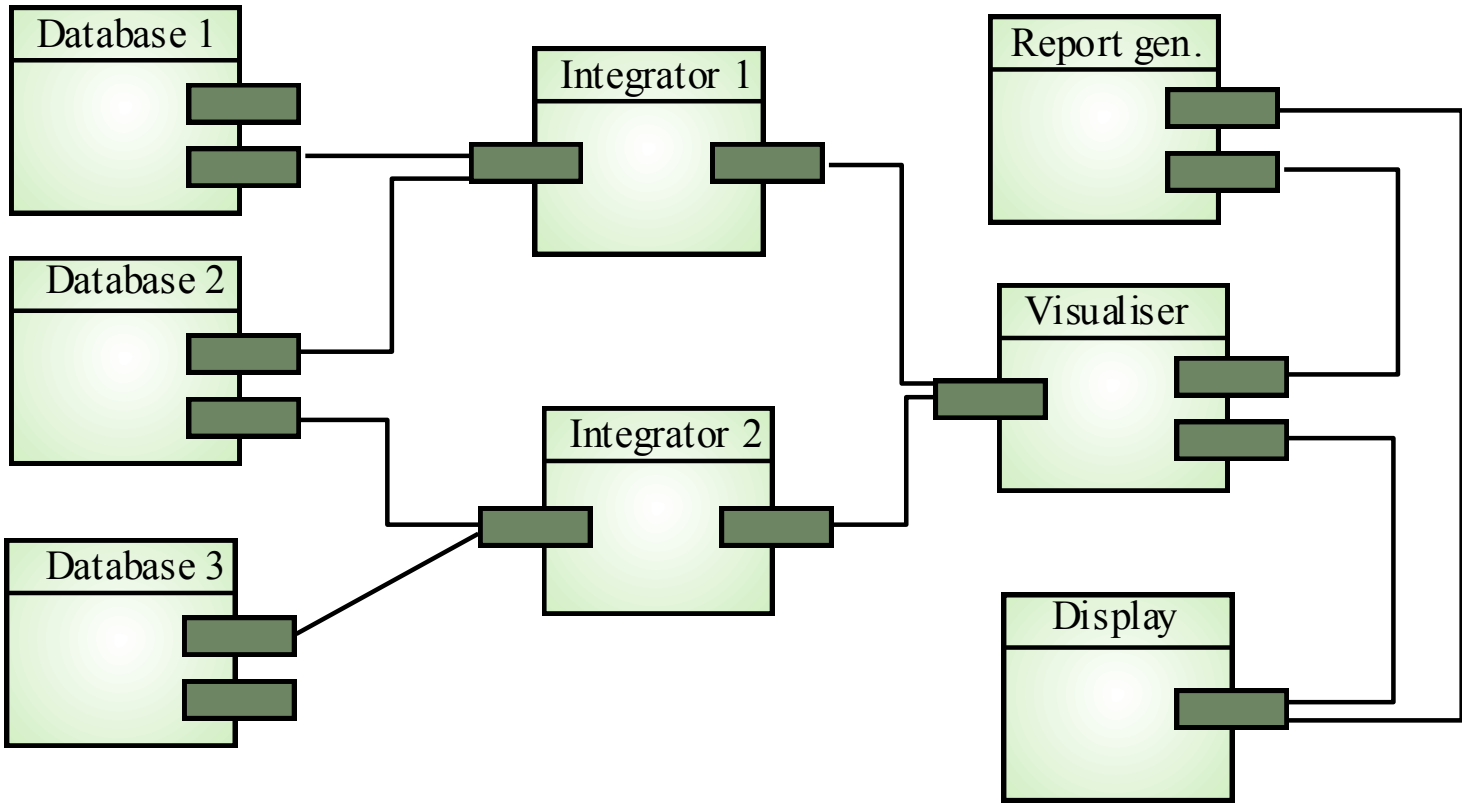
# Advantages of distributed object architecture

- It allows the system designer to delay decisions on where and how services should be provided
- It is a very open system architecture that allows new resources to be added to it as required
- The system is flexible and scaleable
- It is possible to reconfigure the system dynamically with objects migrating across the network as required

# Uses of distributed object architecture

- As a logical model that allows you to structure and organise the system. In this case, you think about how to provide application functionality solely in terms of services and combinations of services
- As a flexible approach to the implementation of client-server systems. The logical model of the system is a client-server model but both clients and servers are realised as distributed objects communicating through a software bus

# A data mining system



# Data mining system

- The logical model of the system is not one of service provision where there are distinguished data management services
- It allows the number of databases that are accessed to be increased without disrupting the system
- It allows new types of relationship to be mined by adding new integrator objects

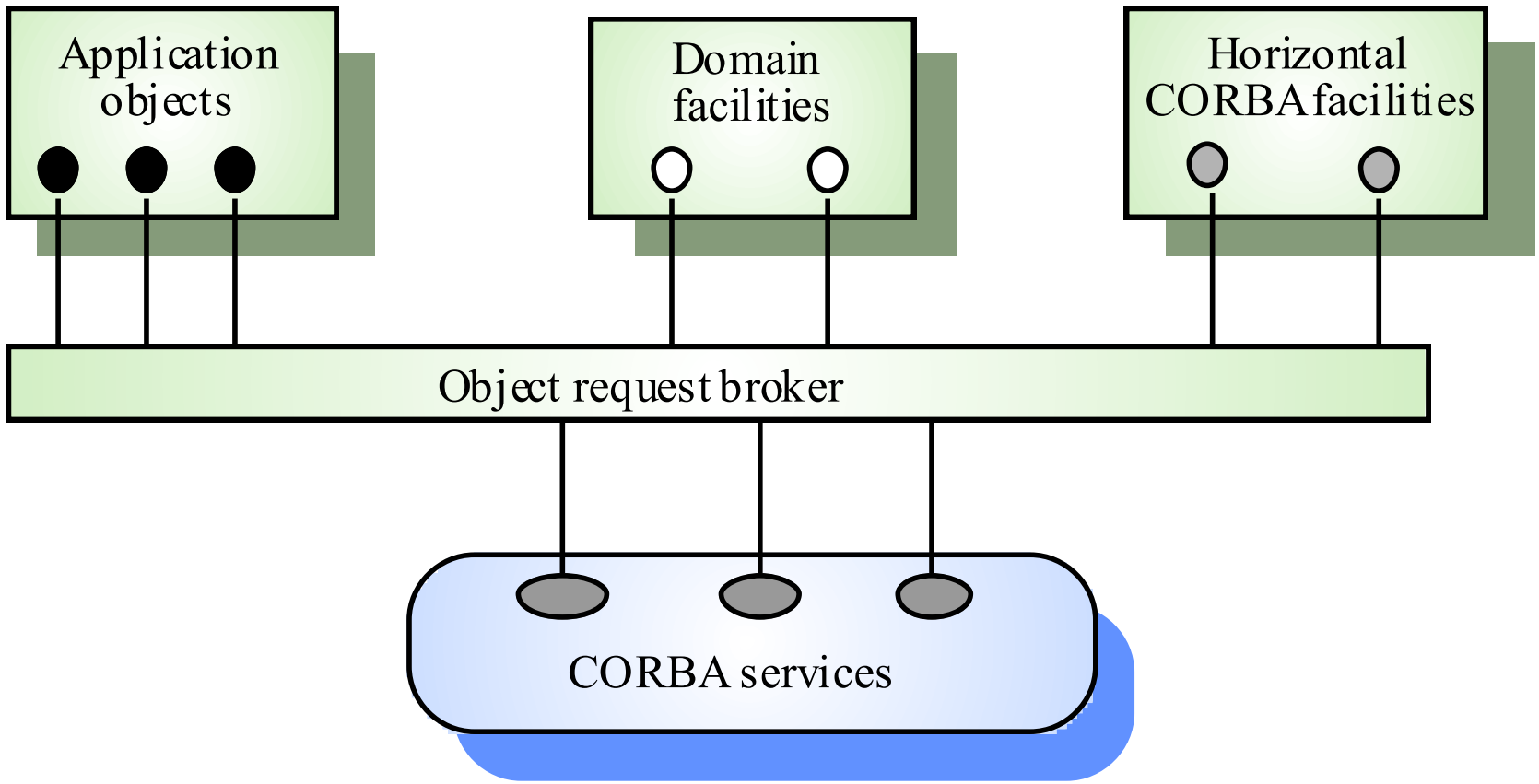
# CORBA

- CORBA is an international standard for an Object Request Broker - middleware to manage communications between distributed objects
- Several implementations of CORBA are available
- DCOM is an alternative approach by Microsoft to object request brokers
- CORBA has been defined by the Object Management Group

# Application structure

- Application objects
- Standard objects, defined by the OMG, for a specific domain e.g. insurance
- Fundamental CORBA services such as directories and security management
- Horizontal (i.e. cutting across applications) facilities such as user interface facilities

# CORBA application structure



# CORBA standards

- An object model for application objects
  - A CORBA object is an encapsulation of state with a well-defined, language-neutral interface defined in an IDL (interface definition language)
- An object request broker that manages requests for object services
- A set of general object services of use to many distributed applications
- A set of common components built on top of these services

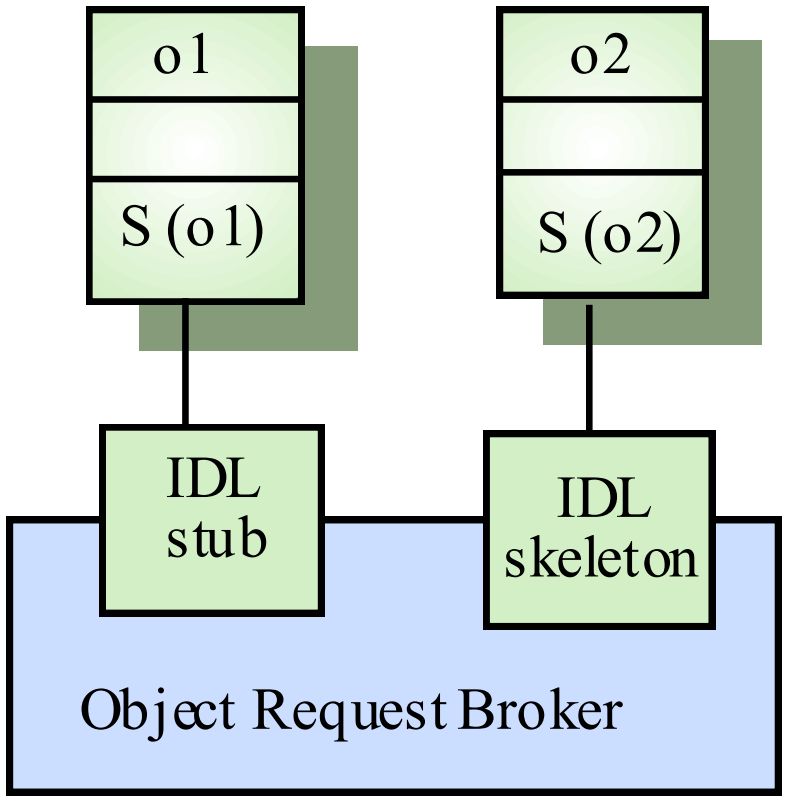
# CORBA objects

- CORBA objects are comparable, in principle, to objects in C++ and Java
- They MUST have a separate interface definition that is expressed using a common language (IDL) similar to C++
- There is a mapping from this IDL to programming languages (C++, Java, etc.)
- Therefore, objects written in different languages can communicate with each other

# Object request broker (ORB)

- The ORB handles object communications. It knows of all objects in the system and their interfaces
- Using an ORB, the calling object binds an IDL stub that defines the interface of the called object
- Calling this stub results in calls to the ORB which then calls the required object through a published IDL skeleton that links the interface to the service implementation

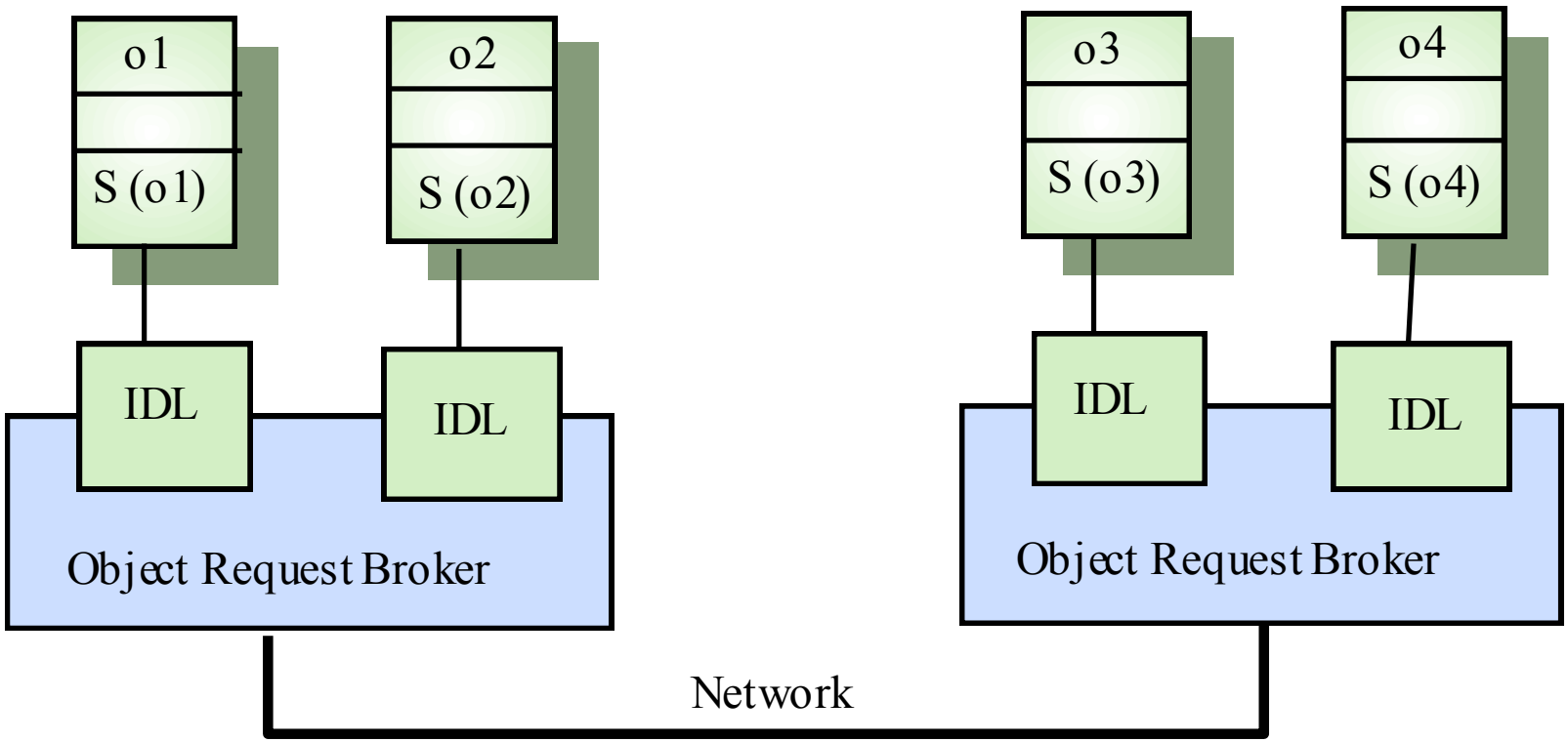
# ORB-based object communications



# Inter-ORB communications

- ORBs are not usually separate programs but are a set of objects in a library that are linked with an application when it is developed
- ORBs handle communications between objects executing on the same machine
- Several ORBS may be available and each computer in a distributed system will have its own ORB
- Inter-ORB communications are used for distributed object calls

# Inter-ORB communications



# CORBA services

- Naming and trading services
  - These allow objects to discover and refer to other objects on the network
- Notification services
  - These allow objects to notify other objects that an event has occurred
- Transaction services
  - These support atomic transactions and rollback on failure

# *Example of a Simple CORBA Application Written in Java*

# Starting Clients, Servers, and Name Servers

*# Compile the IDL into a Java stub*

**idltojava -fno-cpp Hello.idl**

*# Compile the Java program and stub*

**javac *.java HelloApp/*.java**

*# Initiate the **name server** on king.mcs.drexel.edu on port 1050.*

**tnameserv -ORBInitialPort 1050 -ORBInitialHost king.mcs.drexel.edu**

*# Initiate the **hello server** (relies on the name server being on king's 1050 port.)*

**java HelloServer -ORBInitialPort 1050 -ORBInitialHost king.mcs.drexel.edu**

*# Initiate the **hello client** (relies on the name server being on king's 1050 port.)*

**java HelloClient -ORBInitialPort 1050 -ORBInitialHost king.mcs.drexel.edu**

# “Hello World” IDL Specification

```
module HelloApp
{
    interface Hello
    {
        string sayHello();
    };
};
```

# "Hello World" Client Java Program

```
import HelloApp.*;
import org.omg.CosNaming.*;  // package used for name service
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class HelloClient {
    public static void main(String args[])
    {
        try{
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);  // args contain info. about
                                             // name server port/host
            // get the root naming context
            org.omg.CORBA.Object objRef =
                    orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            // casting ...
// continued on next page ...
```

# “Hello World” Client Java Program (Cont’d)

```
            // resolve the Object Reference in Naming
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            Hello HelloRef = HelloHelper.narrow(ncRef.resolve(path));

            // call the Hello server object and print results
            String Hello = HelloRef.sayHello();
            System.out.println(Hello);

        } catch (Exception e) {
            System.out.println("ERROR : " + e) ;
            e.printStackTrace(System.out);
        }
    }
}
```

# “Hello World” Server Java Program

```
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

class HelloServant extends _HelloImplBase
{
    public String sayHello()
    {
        return "\nHello world !!\n";
    }
}
```

# "Hello World" Server Java Program (Cont'd)

```
public class HelloServer {

    public static void main(String args[])
    {
        try{
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // create servant and register it with the ORB
            HelloServant HelloRef = new HelloServant();
            orb.connect(HelloRef); // HelloRef will be published ...

            // get the root naming context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
// continued on next slide ...
```

# "Hello World" Server Java Program (Cont'd)

```
            // bind the Object Reference in Naming
            NameComponent nc = new NameComponent("Hello", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, HelloRef);  // clobber old binding if any ...

            // wait for invocations from clients
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) { // aquires a mutex lock on thread
                sync.wait();        // server sleeps ...
            }

        } catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```