Software Design

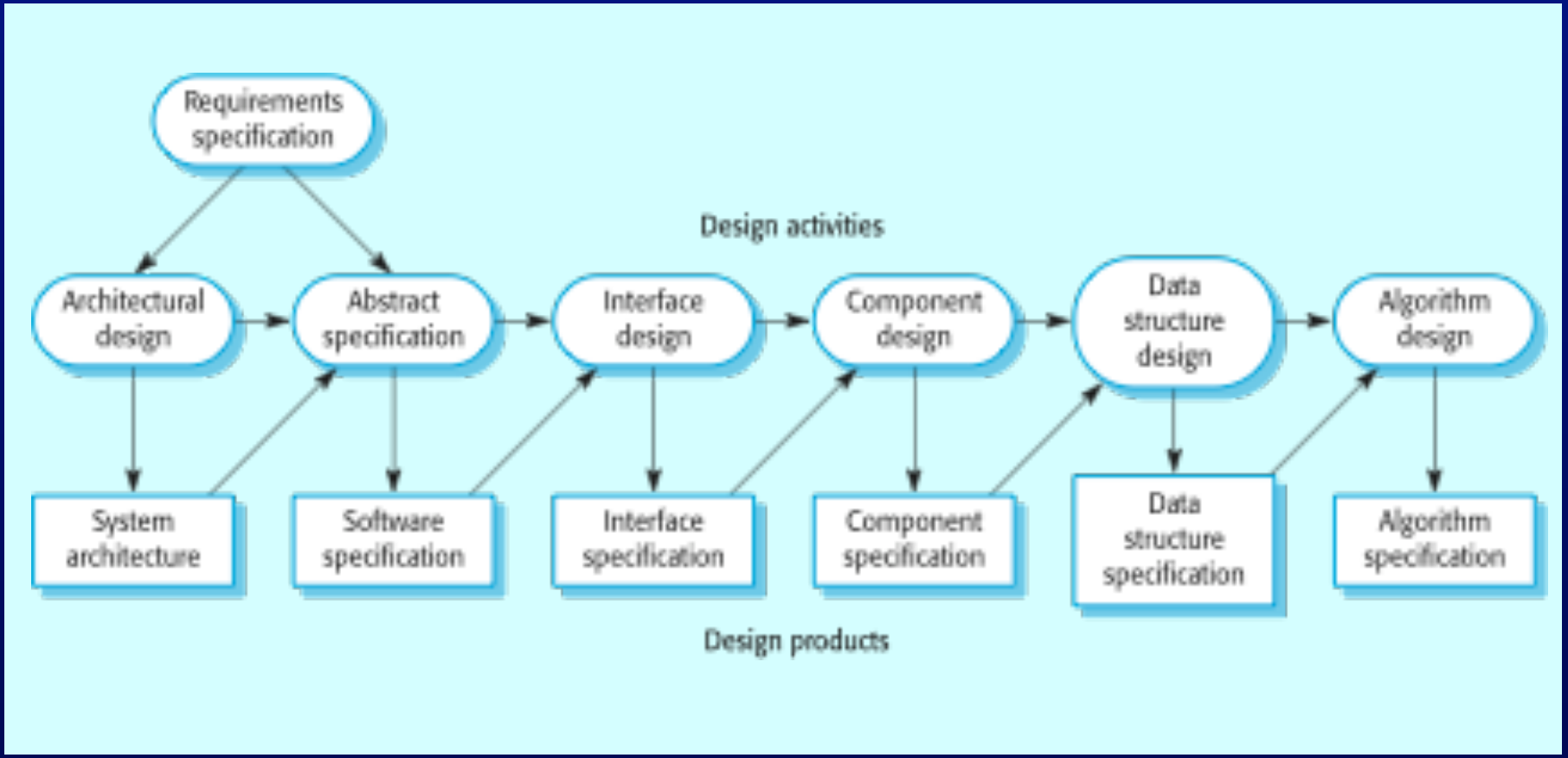
Software design and implementation

- The process of converting the system specification into an executable system.
- Software design
  - Design a software structure that realises the specification;
- Implementation
  - Translate this structure into an executable program;
- The activities of design and implementation are closely related and may be inter-leaved.

Design process activities

- Architectural design
- Abstract specification
- Interface design
- Component design
- Data structure design
- Algorithm design

The software design process



Software architecture

- Architectural design is:
  - Sub-systems identification
  - Control and communication frameworks specification
- A description of the software architecture is the output of this design process.

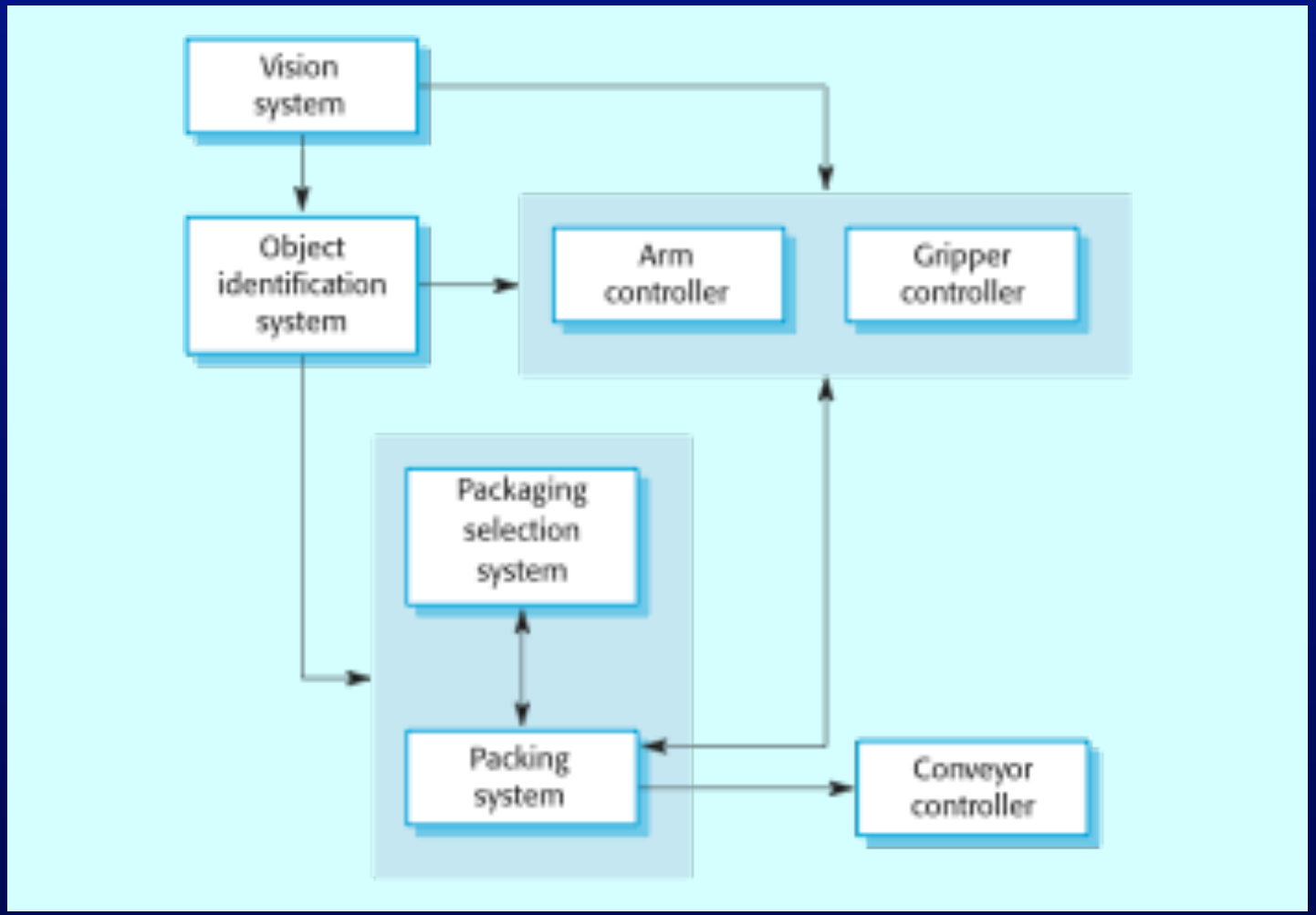
System structuring

- 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.

Box and line diagrams

- Very abstract - they do not show:
  - the nature of component relationships
  - the externally visible properties of the sub-systems.
- Useful for communication with stakeholders and for project planning.

Packing robot control system



Advantages of explicit architecture

- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
  - The architecture may be reusable across a range of systems.

Architecture and system characteristics

- Performance
  - Localise critical operations and minimise communications. Use large rather than fine-grain components.
- Security
  - Use a layered architecture with critical assets in the inner layers.
- Safety
  - Localise safety-critical features in a small number of sub-systems.
- Availability
  - Include redundant components and mechanisms for fault tolerance.
- Maintainability
  - Use fine-grain, replaceable components.

Architectural design decisions

- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- What approach will be used to structure the system?
- How will the system be decomposed into modules?
- What control strategy should be used?
- How will the architectural design be evaluated?
- How should the architecture be documented?

Architectural models (1)

- Used to document an architectural design.
- Static structural model that shows the major system components.
- Dynamic process model that shows the process structure of the system.
- Interface model that defines sub-system interfaces.
- Relationships model such as a data-flow model that shows sub-system relationships.
- Distribution model that shows how sub-systems are distributed across computers.

Architectural models (2)

- Different architectural models may be produced during the design process
- Each model presents different perspectives on the architecture

System organisation

- Reflects the basic strategy that is used to structure a system.
- Three organizational styles are widely used:
  - A shared data repository style;
  - A shared services and servers style;
  - An abstract machine or layered style.

The repository model

- Sub-systems must exchange data.
  - 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.

Repository model characteristics

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

Client-server model

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

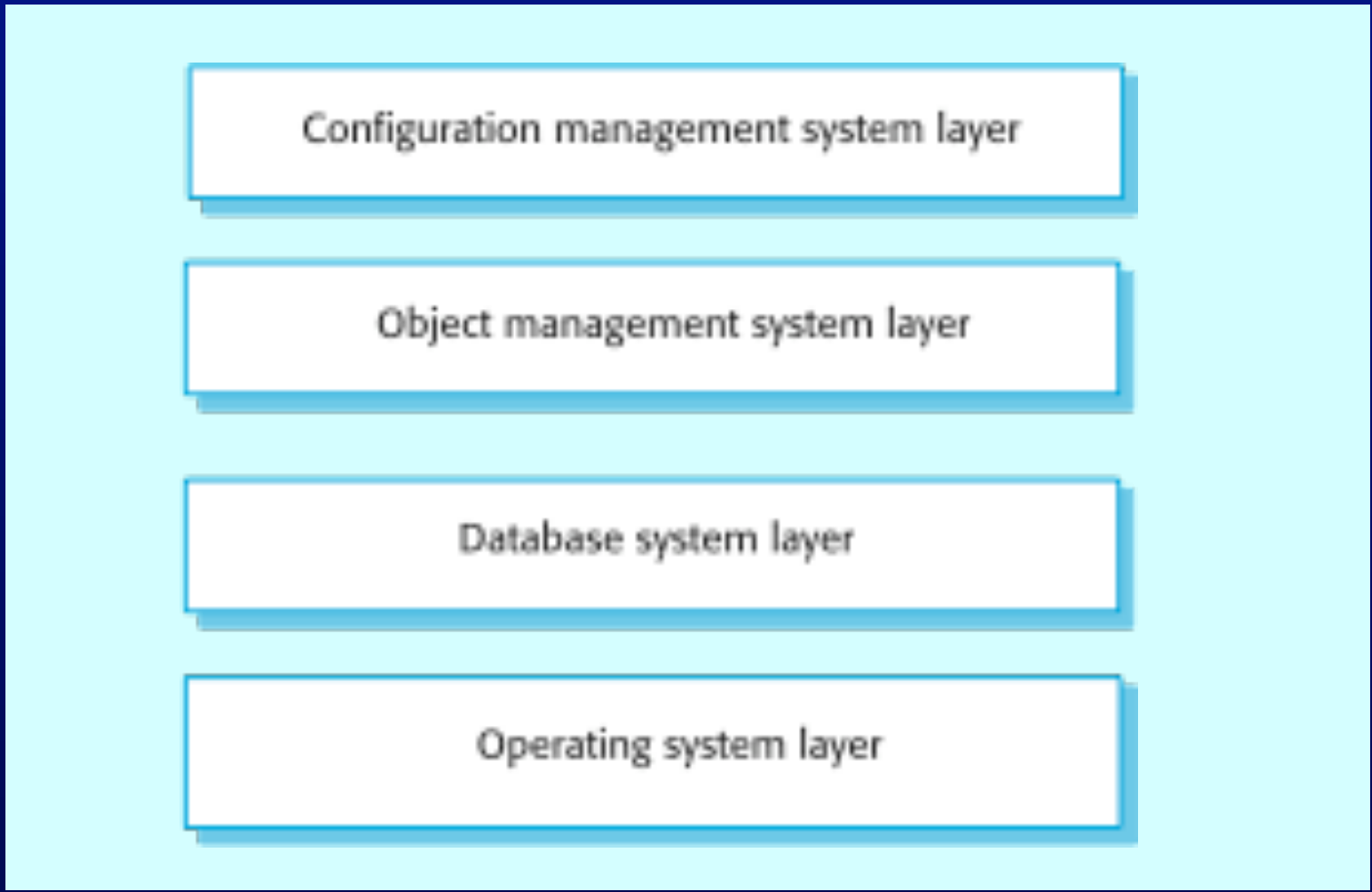
Client-server characteristics

- Advantages
  - Distribution of data is straightforward;
  - Makes effective use of networked systems. May require cheaper hardware;
  - Easy to add new servers or upgrade existing servers.
- Disadvantages
  - No shared data model so sub-systems use different data organisation. Data interchange may be inefficient;
  - 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 (layered) model

- Used to model the interfacing of sub-systems.
- Organises 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 artificial to structure systems in this way.

Version management system



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.

Modular decomposition styles

- Styles of decomposing sub-systems into modules.
- No rigid distinction between system organisation and modular decomposition.

Modular decomposition

- Another structural level where sub-systems are decomposed into modules.
- Two modular decomposition models covered
  - An object model where the system is decomposed into interacting object;
  - A pipeline or data-flow model where the system is decomposed into functional modules which transform inputs to outputs.
- If possible, decisions about concurrency should be delayed until modules are implemented.

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.

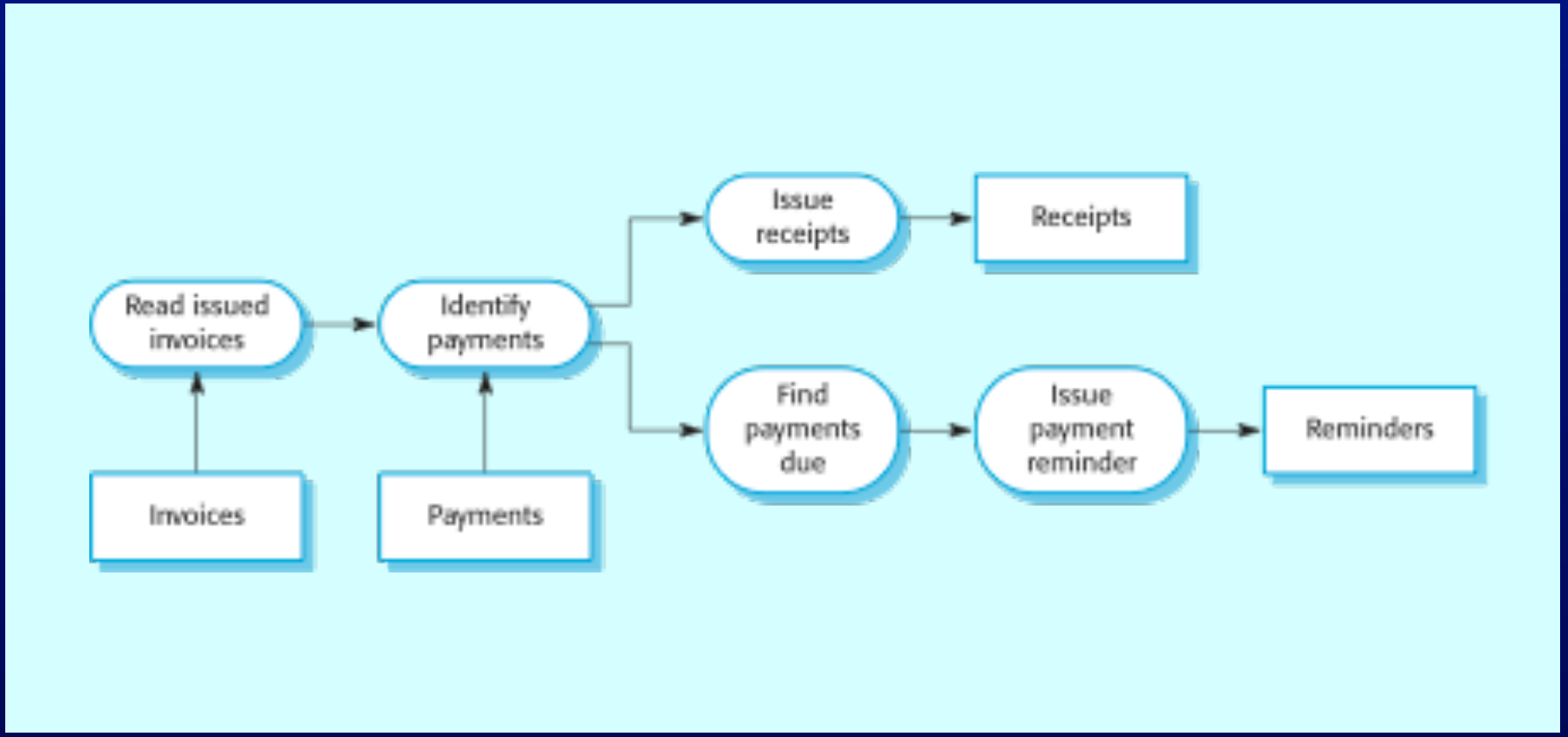
Object model advantages

- Objects are loosely coupled
  - their implementation can be modified without affecting other objects.
- The objects may reflect real-world entities.
- OO implementation languages are widely used.
- However, object interface changes may cause problems and complex entities may be hard to represent as objects.

Function-oriented pipelining

- 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



Pipeline model advantages

- Supports transformation reuse.
- Intuitive organisation for stakeholder communication.
- Easy to add new transformations.
- Relatively simple to implement as either a concurrent or sequential system.
- However, requires a common format for data transfer along the pipeline and difficult to support event-based interaction.

Control styles

- Are concerned with the control flow between sub-systems.
  - Distinct from the system decomposition model.
- Centralised control
  - One sub-system has overall responsibility for control and starts and stops other sub-systems.
- Event-based control
  - Each sub-system can respond to externally generated events from other sub-systems or the system's environment.

Centralised control

- A control sub-system takes responsibility for managing the execution of other sub-systems.
- Call-return model
  - Top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards. Applicable to sequential systems.
- Manager model
  - Applicable to concurrent systems. One system component controls the stopping, starting and coordination of other system processes. Can be implemented in sequential systems as a case statement.

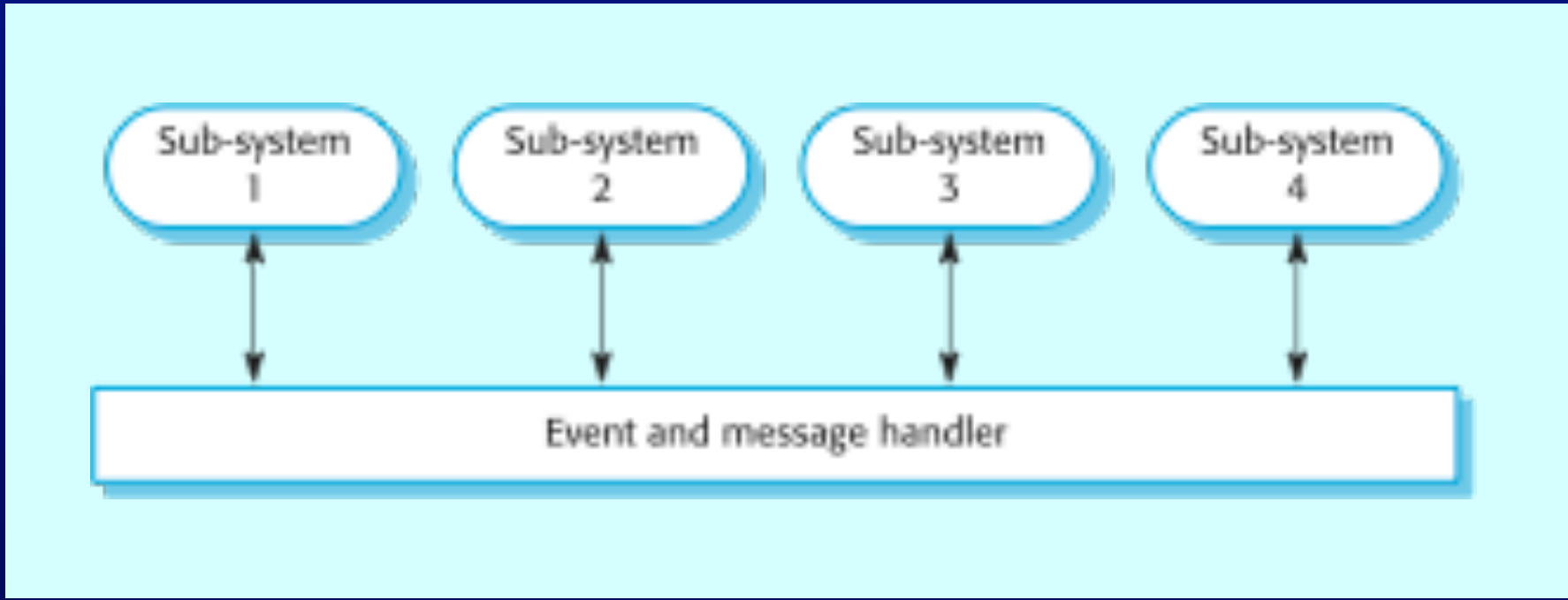
Event-driven systems

- Driven by externally generated events where the timing of the event is outwith the control of the sub-systems which process the event.
- Two principal event-driven models
  - Broadcast models. An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so;
  - Interrupt-driven models. Used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing.

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.

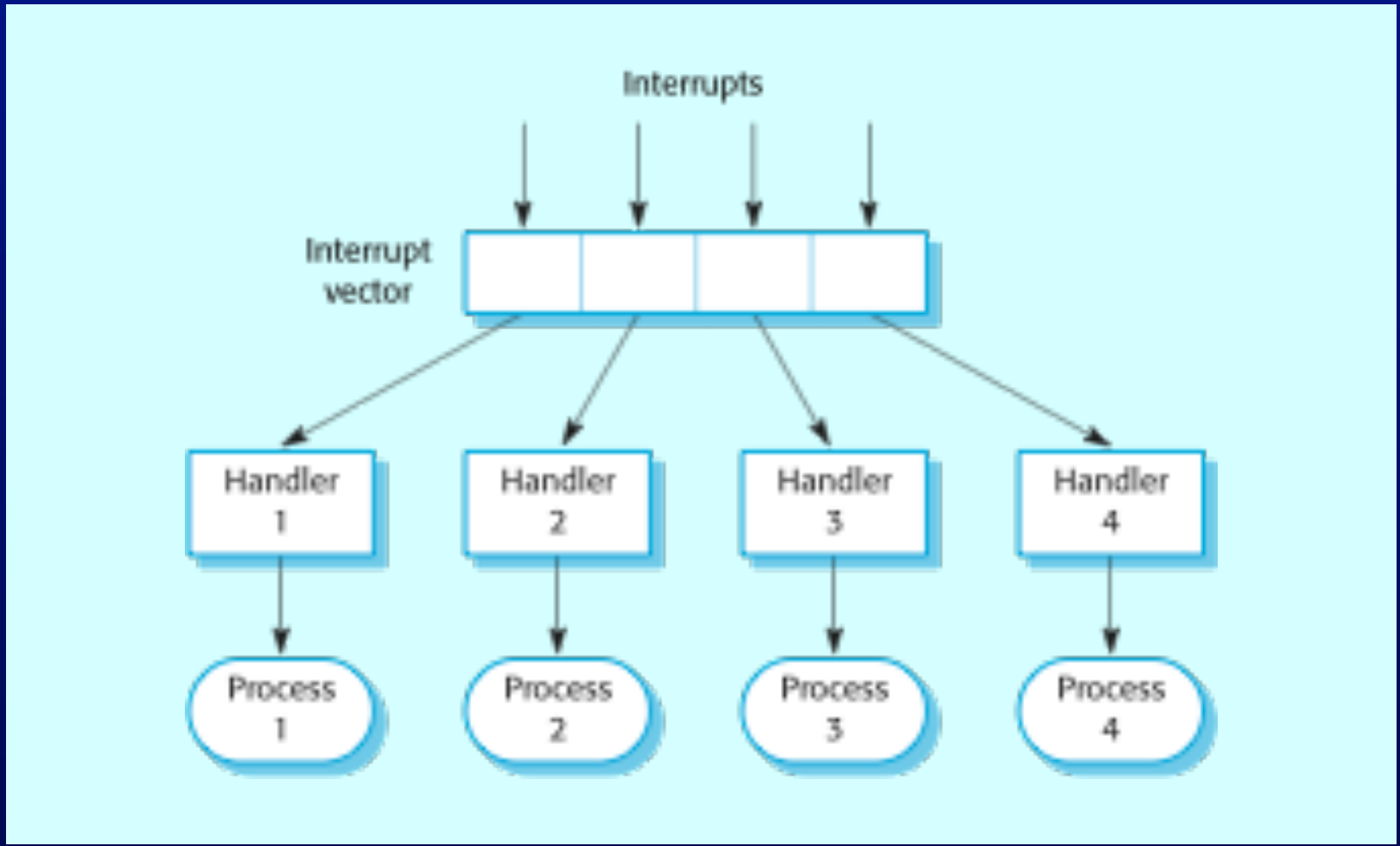
Selective broadcasting



Interrupt-driven systems

- Used in real-time systems where fast response to an event is essential.
- There are known interrupt types with a handler defined for each type.
- Each type is associated with a memory location and a hardware switch causes transfer to its handler.
- Allows fast response but complex to program and difficult to validate.

Interrupt-driven control



Distributed Systems Architectures

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 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 therefore very important for enterprise computing systems.

Distributed system characteristics

- Resource sharing
  - Sharing of hardware and software resources.
- Openness
  - Use of equipment and software from different vendors.
- Concurrency
  - Concurrent processing to enhance performance.
- Scalability
  - Increased throughput by adding new resources.
- Fault tolerance
  - The ability to continue in operation after a fault has occurred.

Distributed system disadvantages

- Complexity
  - Typically, distributed systems are more complex than centralised systems.
- Security
  - More susceptible to external attack.
- Manageability
  - More effort required for system management.
- Unpredictability
  - Unpredictable responses depending on the system organisation and network load.

Distributed systems architectures

- 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.

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 dispatcher.

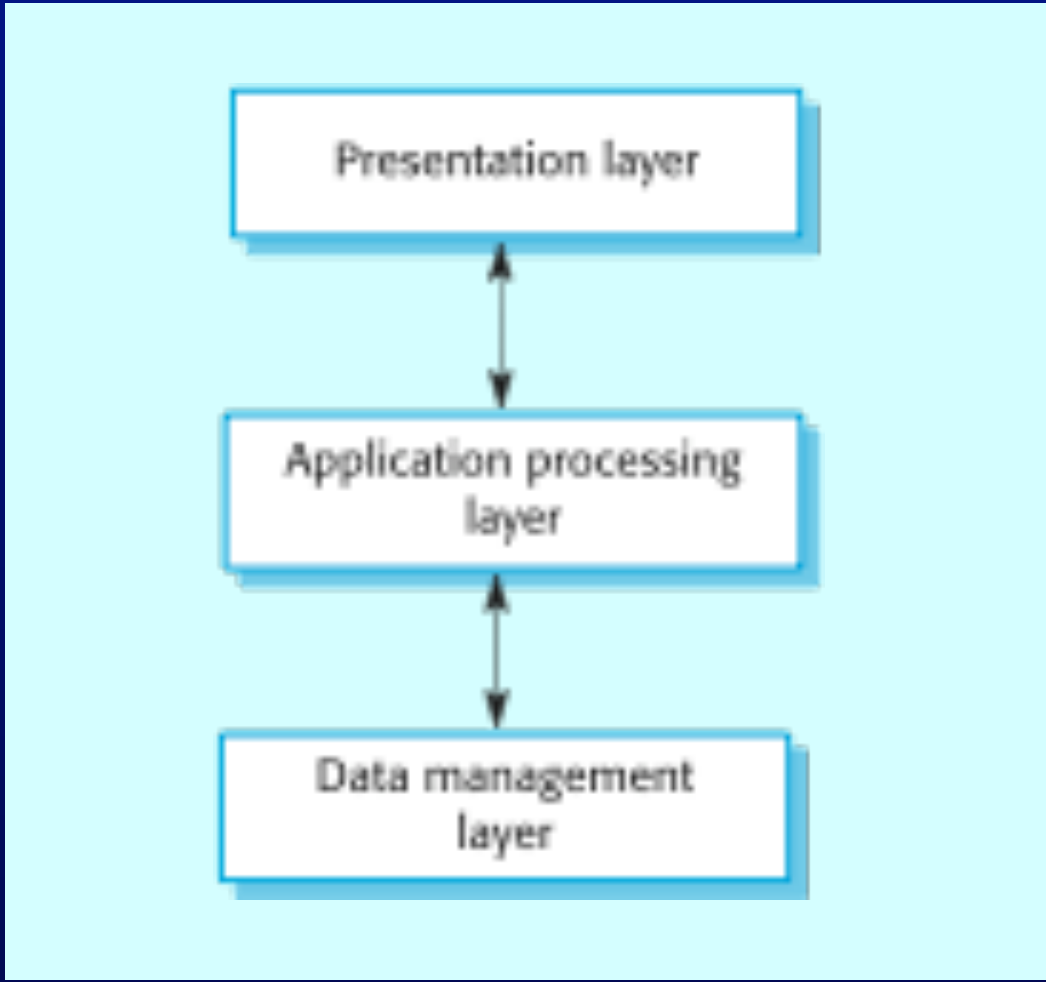
Client-server architectures

- 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.

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 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.

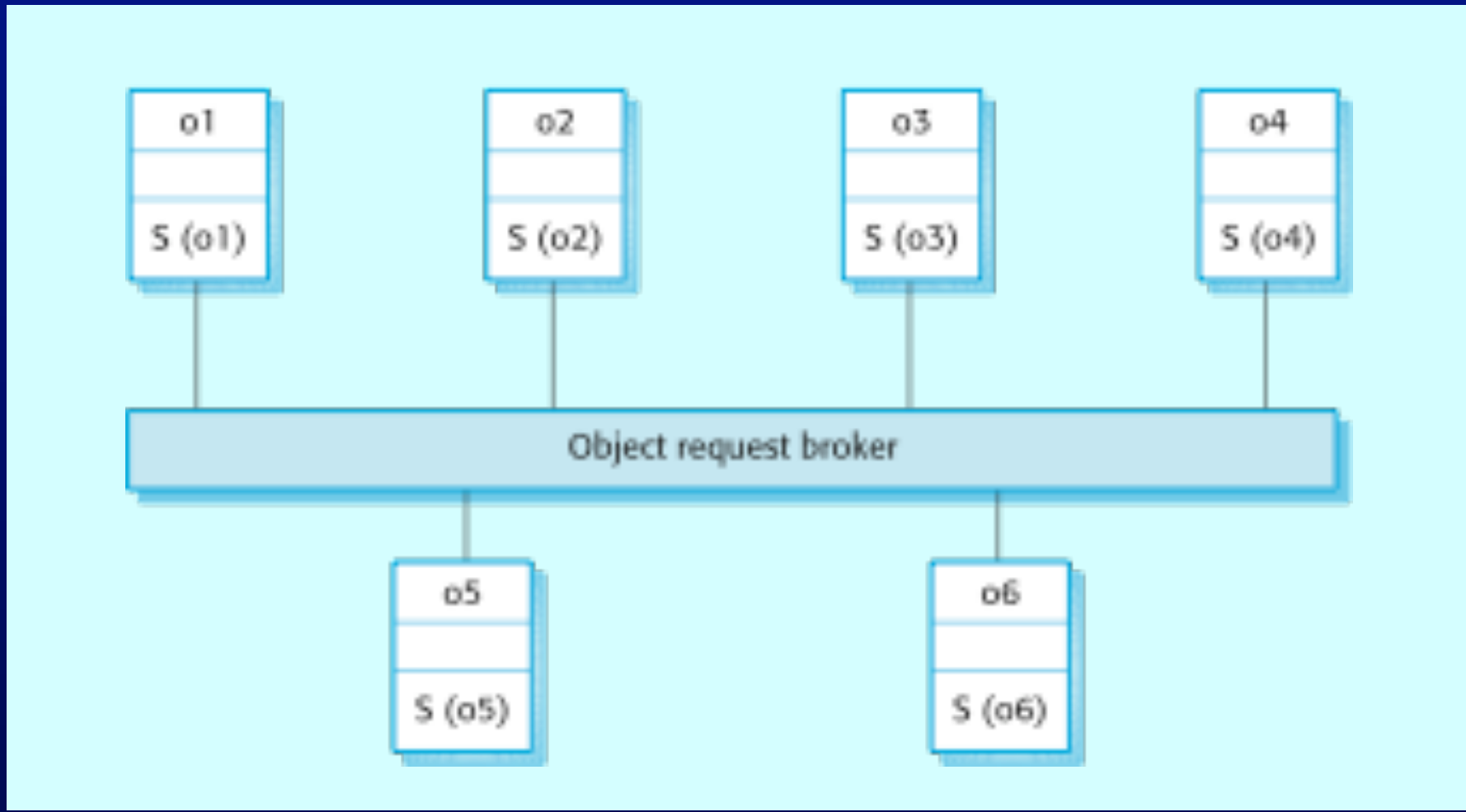
Three-tier architectures

- 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.

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.
- However, distributed object architectures are 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.

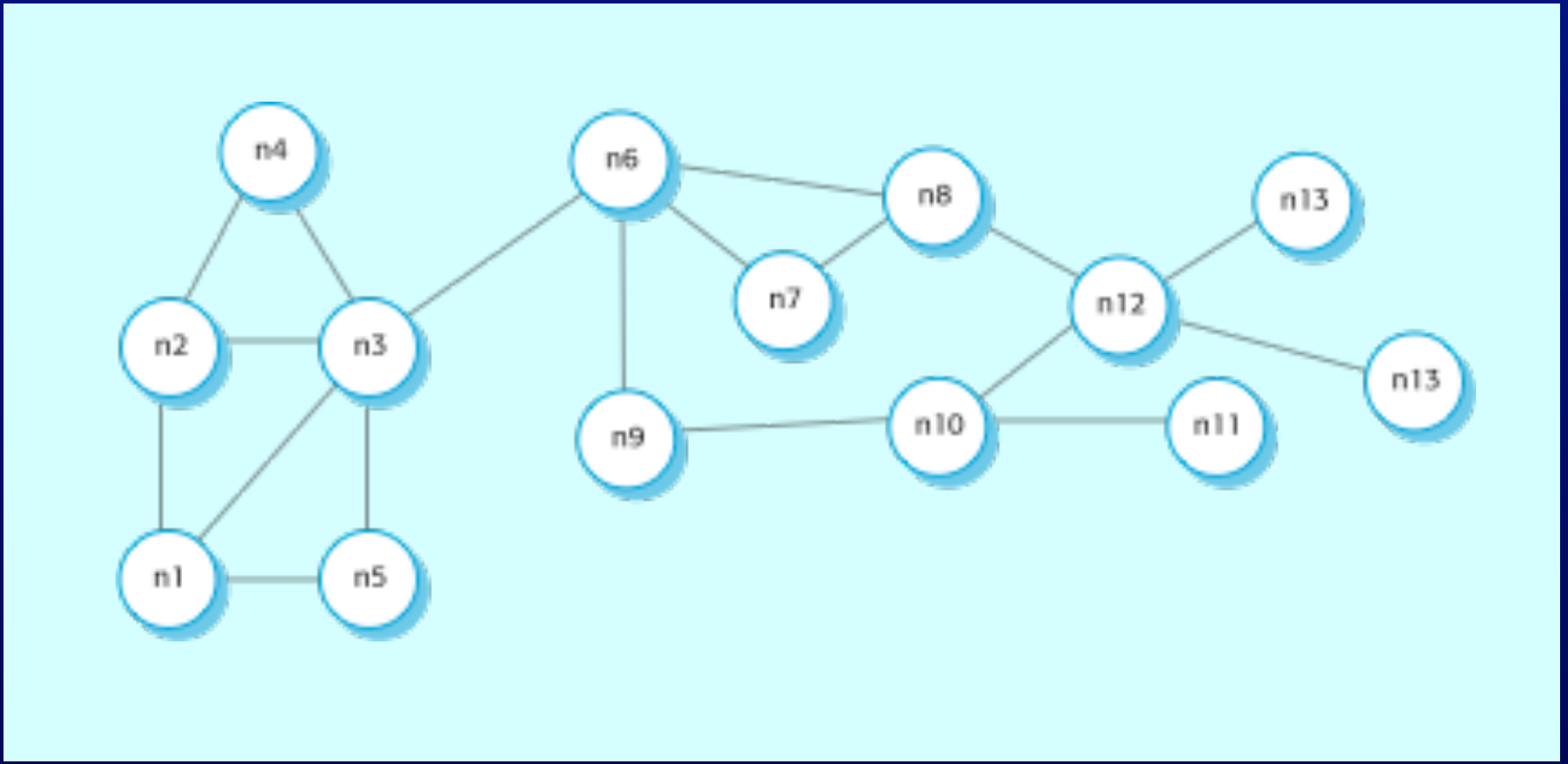
CORBA

- CORBA is an international standard for an Object Request Broker - middleware to manage communications between distributed objects.
- Middleware for distributed computing is required at 2 levels:
  - At the logical communication level, the middleware allows objects on different computers to exchange data and control information;
  - At the component level, the middleware provides a basis for developing compatible components. CORBA component standards have been defined.

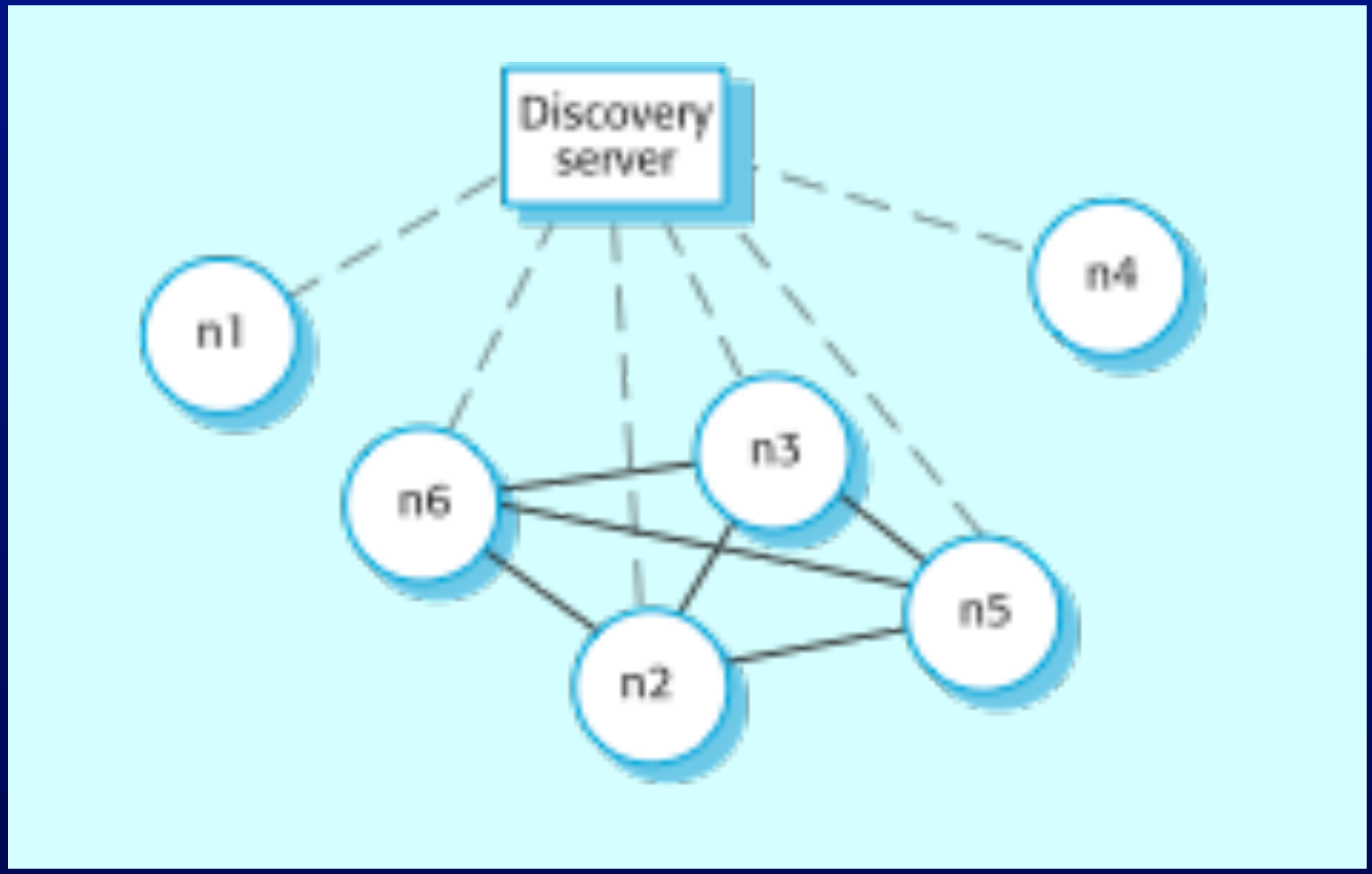
Peer-to-peer architectures

- Peer to peer (p2p) systems are decentralised systems where computations may be carried out by any node in the network.
- The overall system is designed to take advantage of the computational power and storage of a large number of networked computers.
- Most p2p systems have been personal systems but there is increasing business use of this technology.

Decentralised p2p architecture



Semi-centralised p2p architecture



Service-oriented architectures

- Based around the notion of externally provided services (web services).
- A web service is a standard approach to making a reusable component available and accessible across the web
  - A tax filing service could provide support for users to fill in their tax forms and submit these to the tax authorities.

Services standards

- Services are based on agreed, XML-based standards so can be provided on any platform and written in any programming language.
- Key standards
  - SOAP - Simple Object Access Protocol;
  - WSDL - Web Services Description Language;
  - UDDI - Universal Description, Discovery and Integration.

Object-oriented Design

Object-oriented development

- Object-Oriented Analysis, Design and Programming are related but distinct.
- OOA: developing an object model of the application domain.
- OOD: developing an object-oriented system model to implement requirements.
- OOP: realising an OOD using an OO programming language such as Java or C++.

Characteristics of OOD

- Objects are abstractions of real-world or system entities and manage themselves.
- Objects are independent and encapsulate state and representation information.
- System functionality is expressed in terms of object services.
- Shared data areas are eliminated. Objects communicate by message passing.
- Objects may be distributed and may execute sequentially or in parallel.

Advantages of OOD

- Easier maintenance
  - Objects may be understood as stand-alone entities.
- Reusability
  - Objects are potentially reusable components.
- Natural Metaphor
  - There may be an obvious mapping from real world entities to system objects.

The Unified Modeling Language

- Several different notations for describing object-oriented designs were proposed in the 1980s and 1990s.
- The Unified Modeling Language is an integration of these notations.
- It describes notations for a number of different models that may be produced during OO analysis and design.
- It is now a *de facto* standard for OO modelling.

An object-oriented design process

- Structured design processes involve developing a number of different system models.
- They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- However, for large systems developed by different groups design models are an essential communication mechanism.

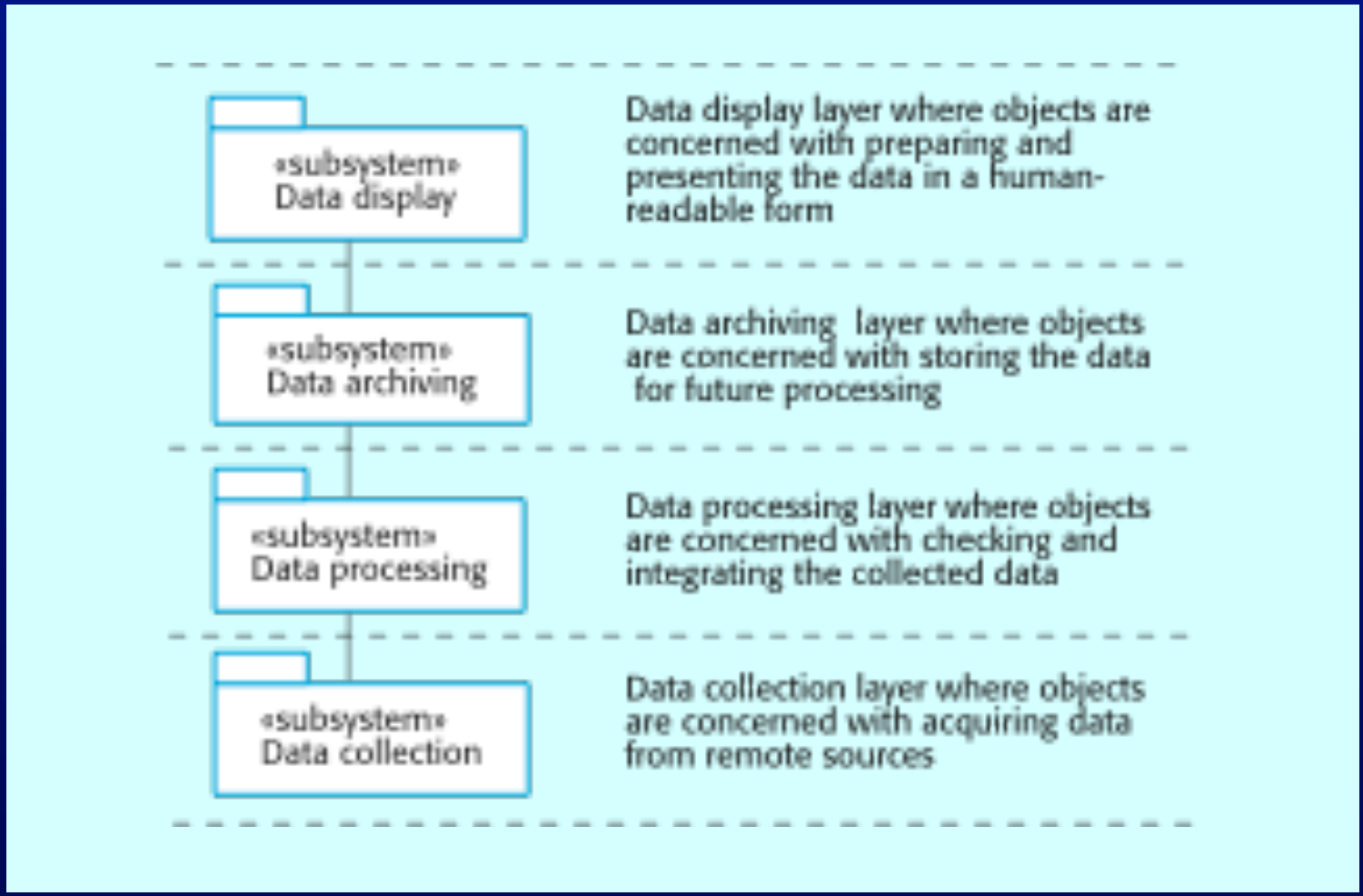
Process stages

- Highlights key activities without being tied to any proprietary process such as the RUP.
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.

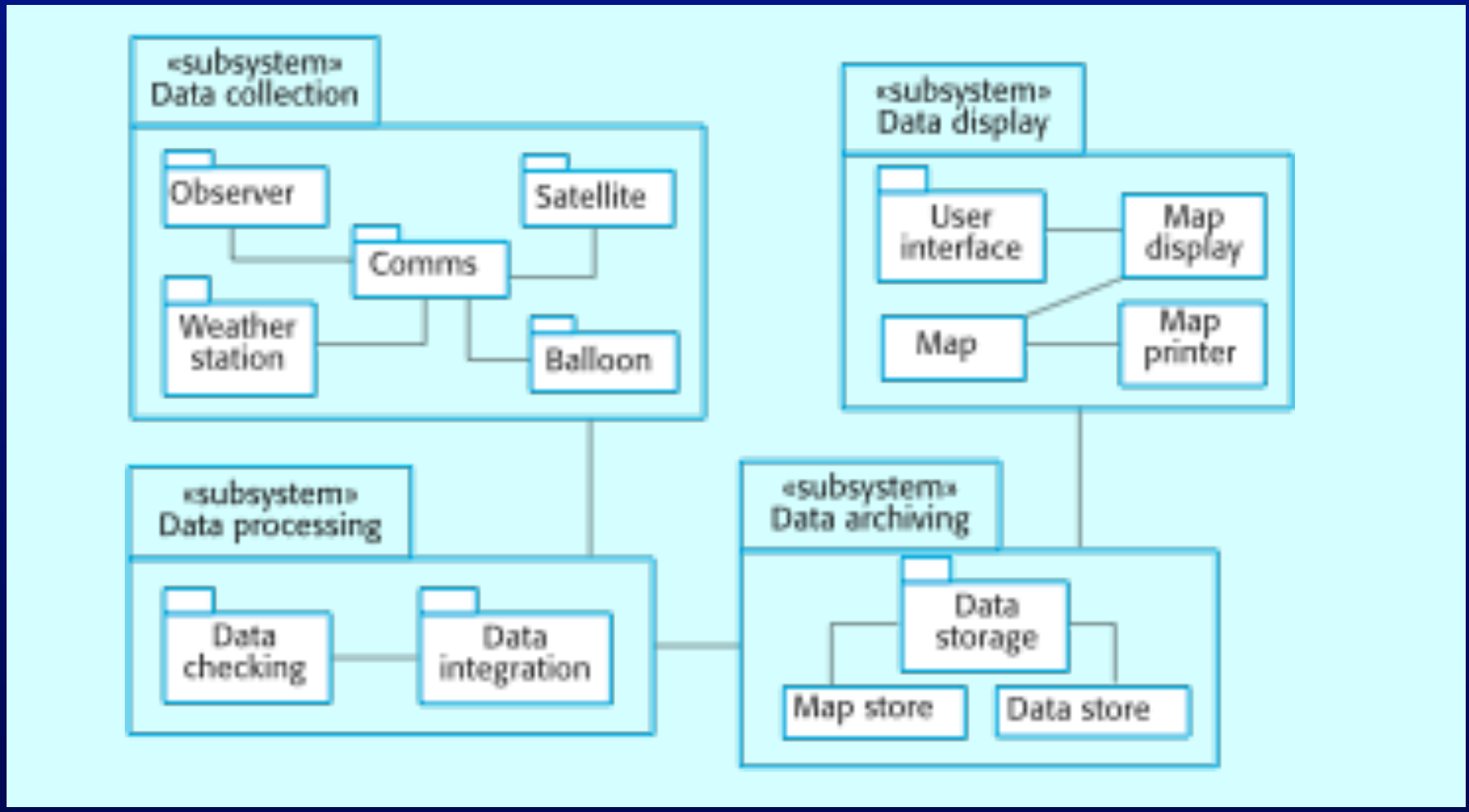
System context and models of use

- Develop an understanding of the relationships between the software being designed and its external environment
- System context
  - A static model that describes other systems in the environment. Use a subsystem model to show other systems. Following slide shows the systems around the weather station system.
- Model of system use
  - A dynamic model that describes how the system interacts with its environment. Use use-cases to show interactions

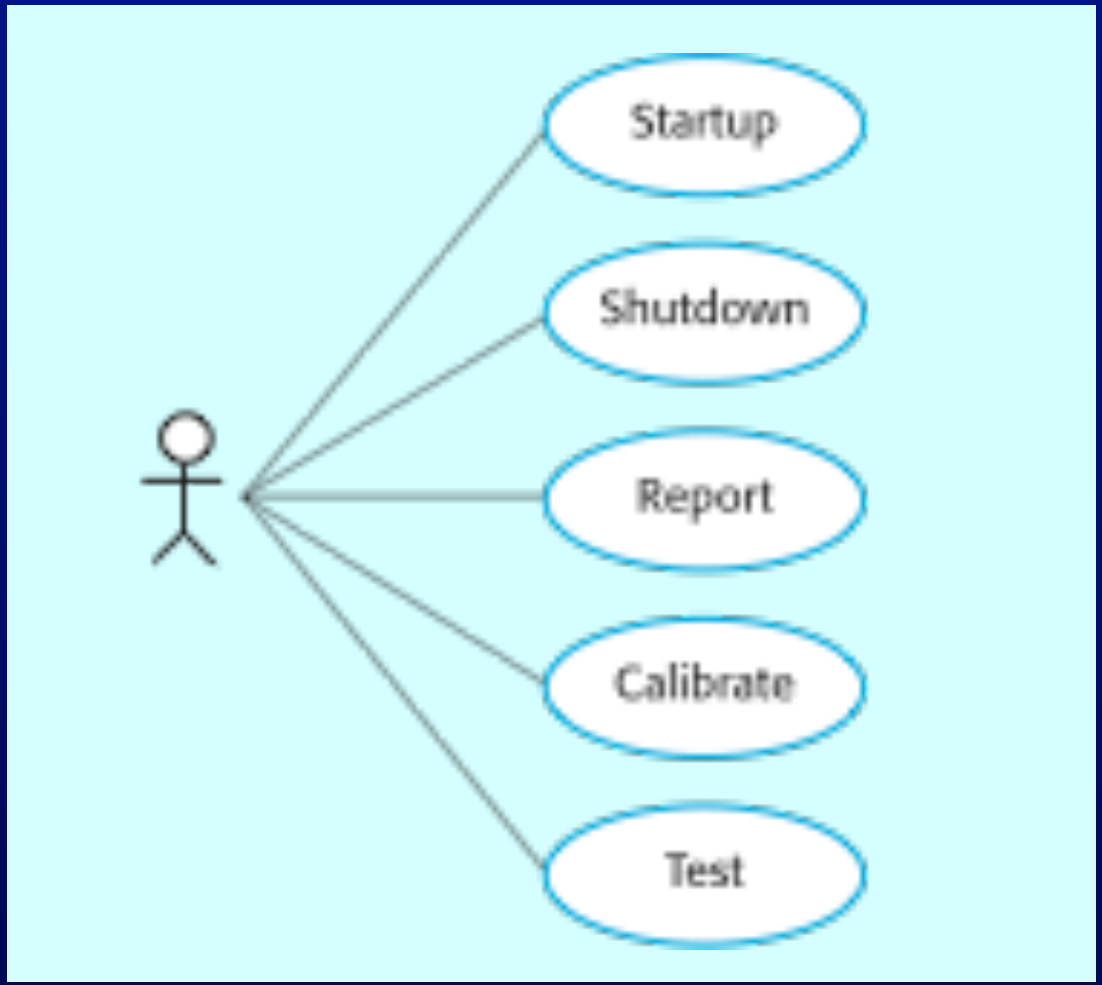
Layered architecture



Subsystems in the weather mapping system



Use-cases for the weather station



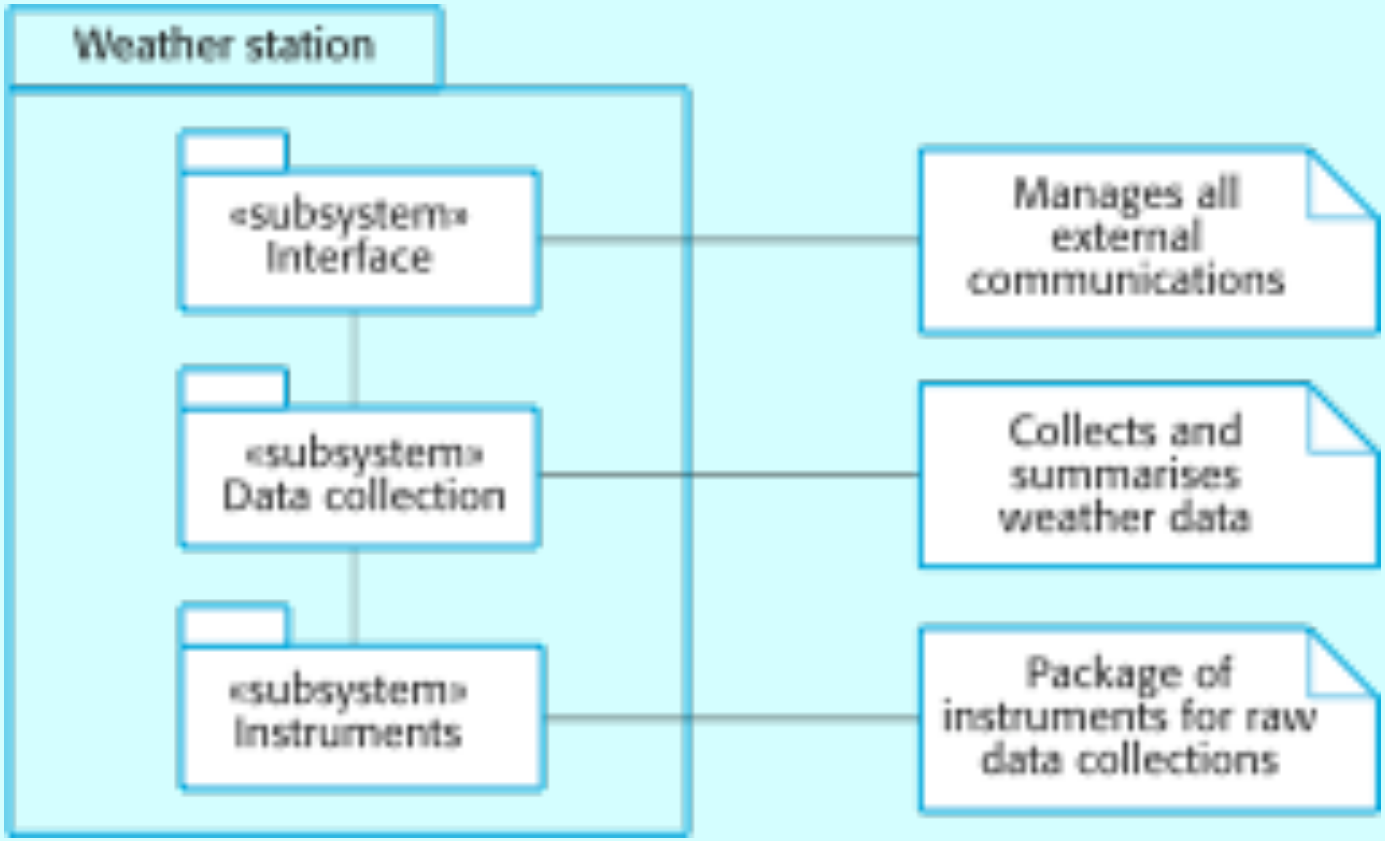
Use-case description

System	Weather station
Use-case	Report
Actors	Weather data collection system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather data collection system. The data sent are the maximum minimum and average ground and air temperatures, the maximum, minimum and average air pressures, the maximum, minimum and average wind speeds, the total rainfall and the wind direction as sampled at 5 minute intervals.
Stimulus	The weather data collection system establishes a modem link with the weather station and requests transmission of the data.
Response	The summarised data is sent to the weather data collection system
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to the other and may be modified in future.

Architectural design

- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- There should normally be no more than 7 entities in an architectural model.

Weather station architecture



Object identification

- Identifying objects (or object classes) is the most difficult part of object oriented design.
- There is no 'magic formula' for object identification.
  - It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.

Approaches to identification

- Use a grammatical approach based on a natural language description of the system (used in Hood OOD method).
- Base the identification on tangible things in the application domain.
- Use a behavioural approach and identify objects based on what participates in what behaviour.
- Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

Design models

- Design models show the objects and object classes and relationships between these entities.
  - Static models describe the static structure of the system in terms of object classes and relationships.
  - Dynamic models describe the dynamic interactions between objects.

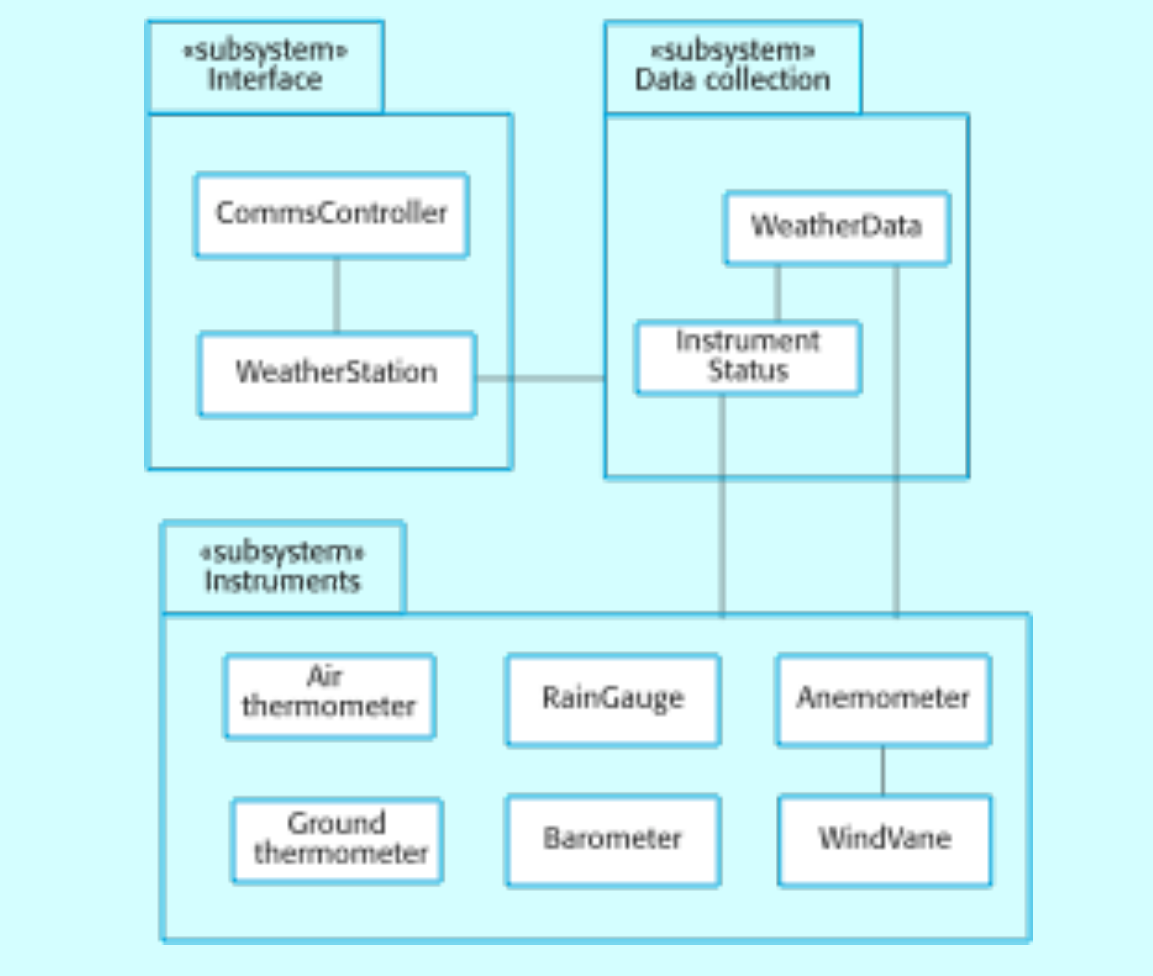
Examples of design models

- Sub-system models that show logical groupings of objects into coherent subsystems.
- Sequence models that show the sequence of object interactions.
- State machine models that show how individual objects change their state in response to events.
- Other models include use-case models, aggregation models, generalisation models, etc.

Subsystem models

- Shows how the design is organised into logically related groups of objects.
- In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

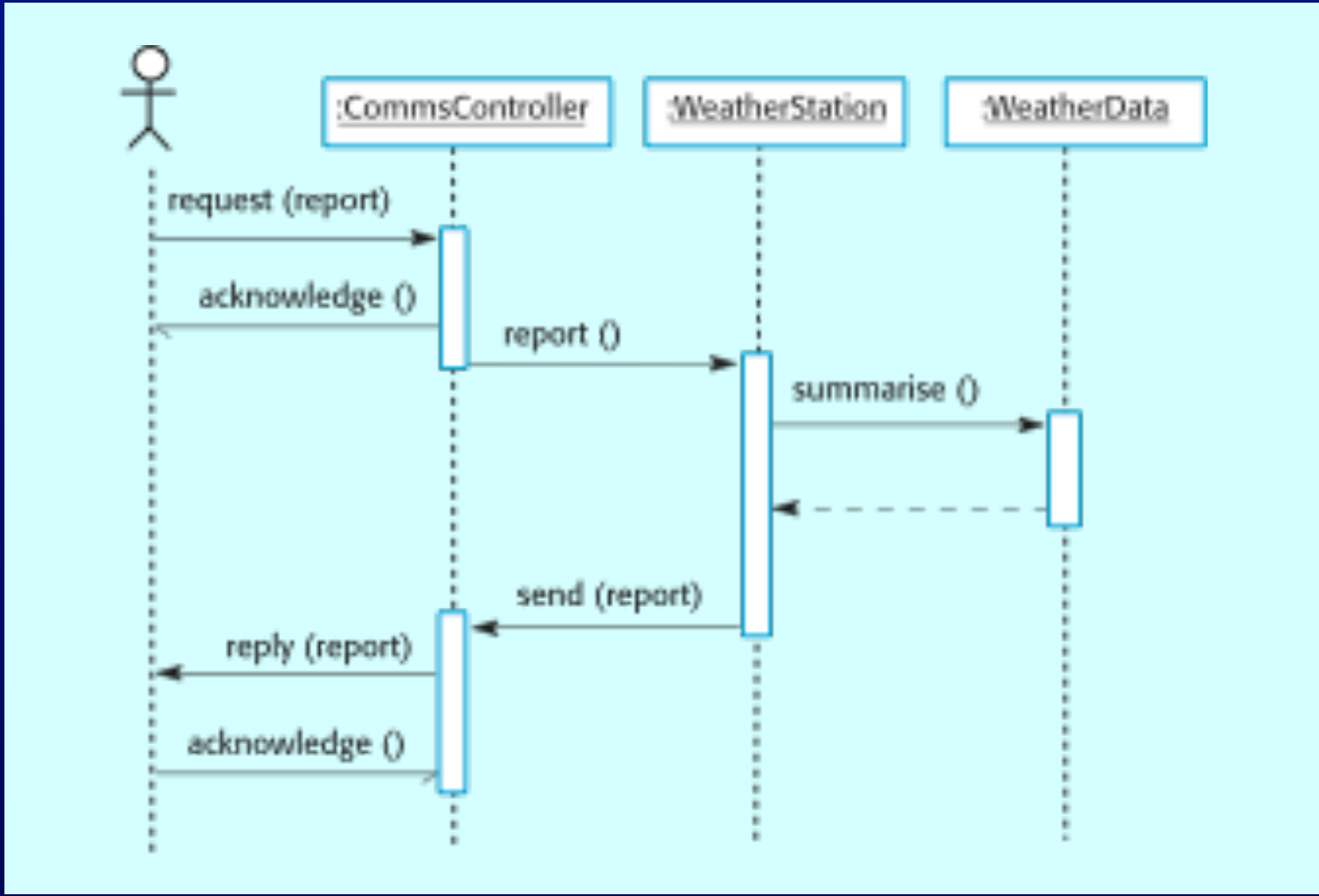
Weather station subsystems



Sequence models

- Sequence models show the sequence of object interactions that take place
  - Objects are arranged horizontally across the top;
  - Time is represented vertically so models are read top to bottom;
  - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;
  - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

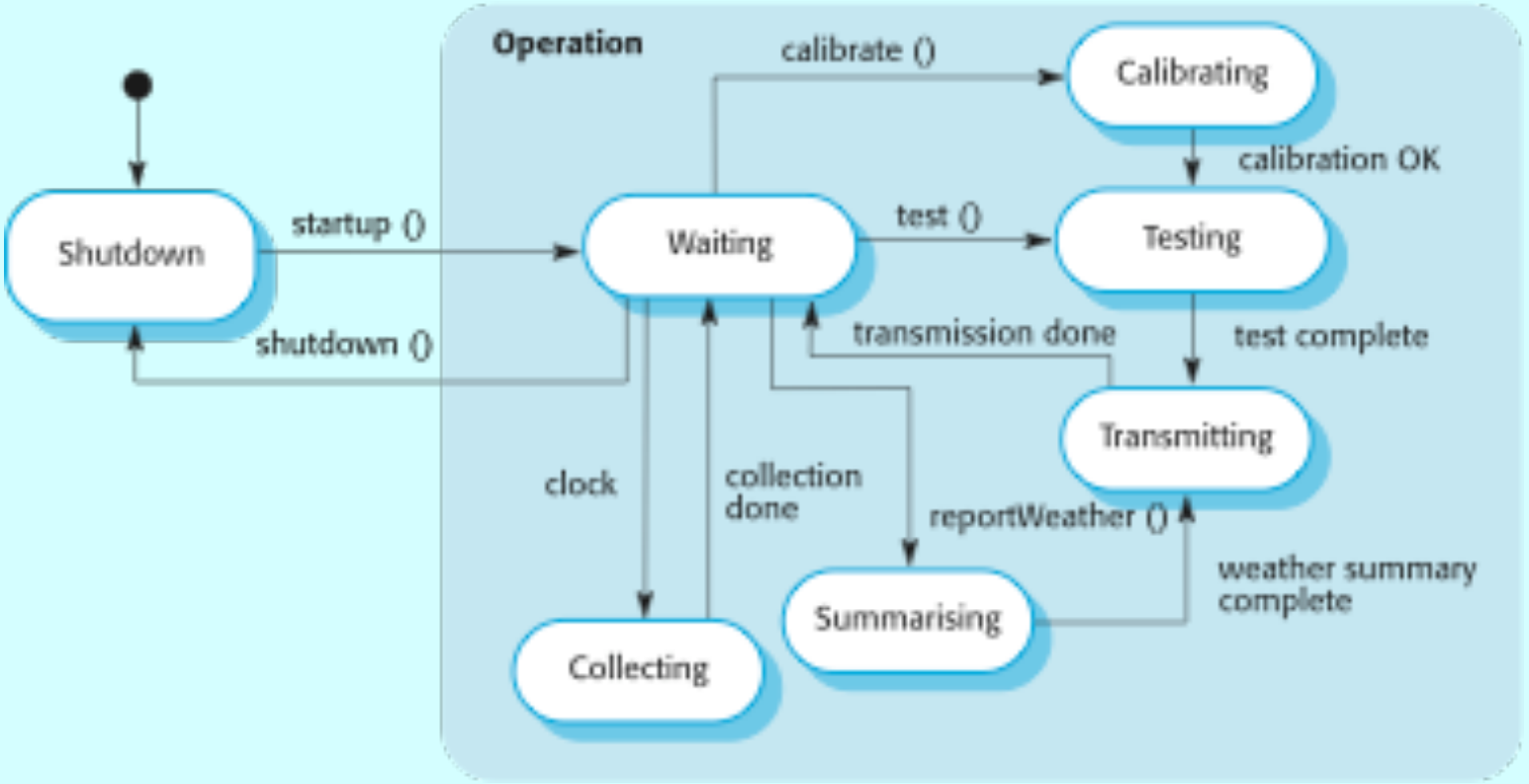
Data collection sequence



Statecharts

- Show how objects respond to different service requests and the state transitions triggered by these requests
  - If object state is Shutdown then it responds to a Startup() message;
  - In the waiting state the object is waiting for further messages;
  - If reportWeather () then system moves to summarising state;
  - If calibrate () the system moves to a calibrating state;
  - A collecting state is entered when a clock signal is received.

Weather station state diagram



Object interface specification

- Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- Designers should avoid designing the interface representation but should hide this in the object itself.
- Objects may have several interfaces which are viewpoints on the methods provided.
- The UML uses class diagrams for interface specification but Java may also be used.

Weather station interface

```
interface WeatherStation {

        public void WeatherStation () ;

        public void startup () ;
        public void startup (Instrument i) ;

        public void shutdown () ;
        public void shutdown (Instrument i) ;

        public void reportWeather ( ) ;

        public void test () ;
        public void test ( Instrument i ) ;

        public void calibrate ( Instrument i) ;

        public int getID () ;

} //WeatherStation
```