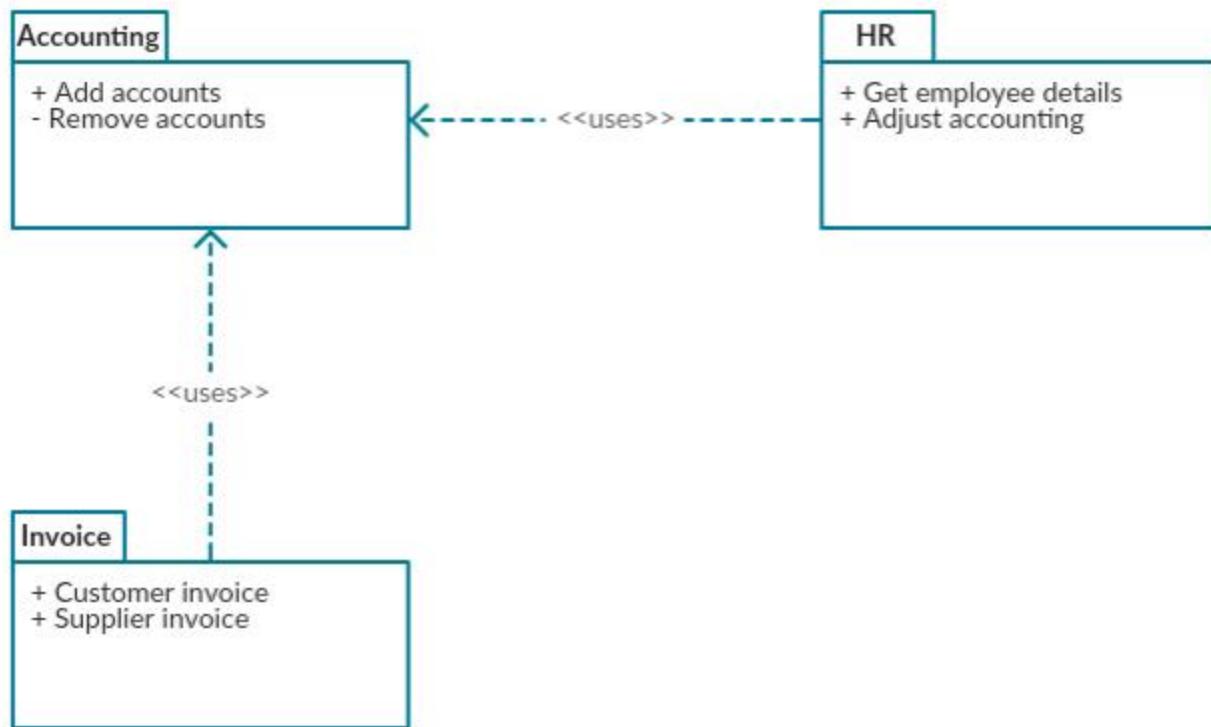
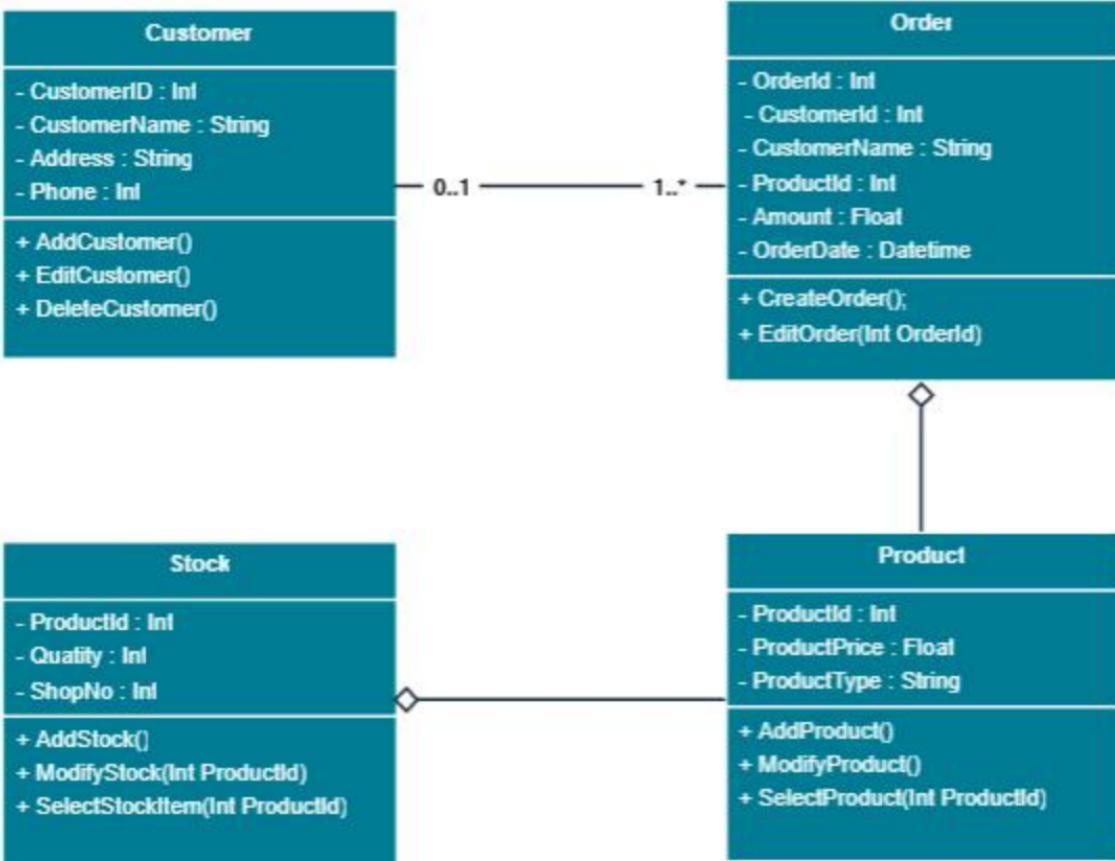


Package Diagram

As the name suggests, a package diagram shows the dependencies between different packages in a system. Check out [this wiki article](#) to learn more about the dependencies and elements found in package diagrams.



Class Diagram for Order Processing System



Please submit your answers as a word document into D2L in the folder ICAIII before the end of class today. For Q6 you can hand-draw the Fault Tree, take and copy the picture in the word document, or simply conduct the FTA descriptively.

Q1. The reliability testing of a product yielded an exception/error 5, 4, 7, 5, 6, 8, 11, 14, 17, 23 hours since the last error. Compute the reliability that the system will not have any failure for 10 continuous hours after starting correctly. (0.5 mark)

Ans:

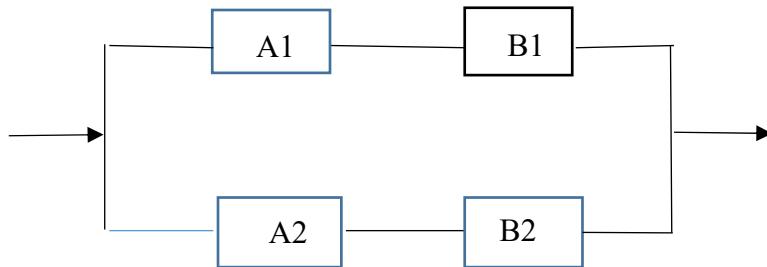
$$MTBF = [5+4+7+5+6+8+11+14+17+23] / 10 = 10.$$

Probability that the product is still functional after 10 hours since successful start is then $R(10) = e^{-10/10} = e^{-1} = 0.36$

Q2. Fault injection is a common technique for developing reliability growth models. Let's suppose a test manager deliberately injects 20 faults/bugs into the software system. After a period of testing, the testers were able to identify 10 of these injected faults but also discovered 20 of non-injected faults. Assuming that the probability of detecting an injected fault over this test duration is same as that of finding a non-injected fault, estimate the remaining non-injected faults in the system. (0.5 mark)

Ans: 20 non-injected bugs are still potentially there to be discovered.

Q3. Consider a distributed systems architecture depicted below. Determine the overall availability of the system given that the availability of each component is 0.9. (0.5 mark)



$$\text{Ans: } A = [1 - [(1 - (A1 * B1)) * (1 - (A2 * B2))]] = [1 - (0.19)(0.19)] = [1 - .0361] = 0.9639$$

Q4. Propose an operational profile of the application you are developing for the course. [0.5 mark]

Q5. Propose a stress test for the Photo Gallery app or for the application that you are developing for the course. [0.5 mark]

Q6. Consider a smartphone application that continuously monitors the onboard accelerometer and gyroscope sensor for the purposes of the fall detection in the elderly and notifies the remote staff, via an SMS message, if a fall is detected. Conduct the safety/hazard analysis of this application, or, if applicable, conduct the safety analysis of the application that you are developing for the course, using FTA (Fault Tree Analysis) based deductive approach. [0.5 mark]

Ans:

fall occurred but no help <= [alert didn't arrive OR alert was ignored]

alert didn't arrive <= [fall not detected OR SMS failed]

fall not detected <= [sensor failed OR incorrect sensor readings OR algorithmic error]

SMS Failed <= [no signal OR low battery OR phone off OR SMS infrastructure failure]

alert was ignored <= [too many false alarms OR staff not available]

too many false alarms <= [algorithm not accurate OR duplicates]

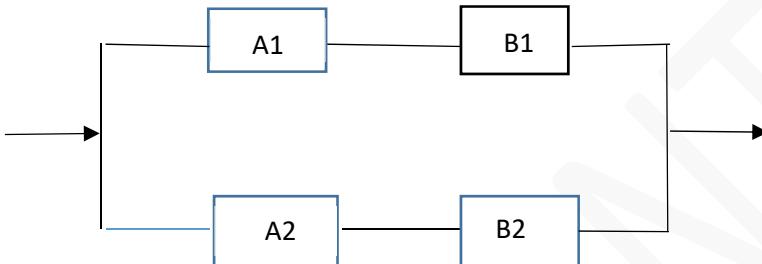
Resources: lec2-4.ppt, lec6-3.ppt, lec4-1.ppt, lec4-2.ppt, Reliability.pdf and Availability.pdf in the “lecture notes” subfolder, and FTA_of_Clinical_Alarms-Hyman_Johnson.pdf in “General Papers” sub folder under comp7081 share out full time BTech folder.

In Class Assignment: Please handwrite the answers on loose sheets and hand in the assignment.

Q1 (a) [0.5 mark] Propose a load test for the photogallery app or the app that you are developing for the course.

Ans: Incrementally increasing the size of the repository of the photos and evaluating its impact on the performance of the search.

(b) [0.5 mark] Assuming that each component, in the architecture depicted below, on the average can process 10 requests per second and the average arrival rate of the requests to the system is 3 requests per second, estimate the average latency or time it takes for a request to be handled by the system assuming that the system could be modeled as a network of queues.



Ans: $0.5 * [1/(10 - 1.5) + 1/(10 - 1.5)] + 0.5 * [1/(10 - 1.5) + 1/(10 - 1.5)]$

Where 0.5 is the probability that a request takes a particular path.

Q2. Illustrate the static composition, in the form of a UML class diagram [2 marks], and the dynamic behavior of the system, in the form of both the UML sequence diagram [2 marks] as well as the UML state machine/chart diagram [2 marks], for a simple Android application whose code is listed below. Marks will be based on how accurately and completely UML was utilized in capturing the correct composition and dynamic behavior of this software.

Note: For simplicity, assume that there is no other activity in the system and when this activity comes to the foreground it stays there forever. Also due to irrelevance, some of the xml layout attributes have been removed.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    tools:context="com.example.server.sampleapp.MainActivity">
    <TextView
        android:id="@+id/textView"
        android:text="Hello World!" />
</RelativeLayout>
```

```
package com.example.server.sampleapp;
interface Callback {
    © Tejinder Randhawa, 2017
```

```

    void onCallback(String result);
}

package com.example.server.sampleapp;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;
import android.os.AsyncTask;
import java.lang.Thread;
public class MainActivity extends AppCompatActivity implements Callback{
    TextView textView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        textView = (TextView) findViewById(R.id.textView);
        IntStore intStore = new IntStore();
        Depositor depositor = new Depositor(this);
        depositor.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, intStore);
        Withdrawer withdrawer = new Withdrawer(this);
        withdrawer.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, intStore);
    }
    @Override
    public void onCallback(String result) {
        textView.setText(result);
    }
}

package com.example.server.sampleapp;
public class IntStore {
    private int contents;
    private boolean available = false;
    public synchronized int withdraw() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        notifyAll();
        return contents;
    }
    public synchronized void deposit(int value) {
        while (available == true) {
            try {

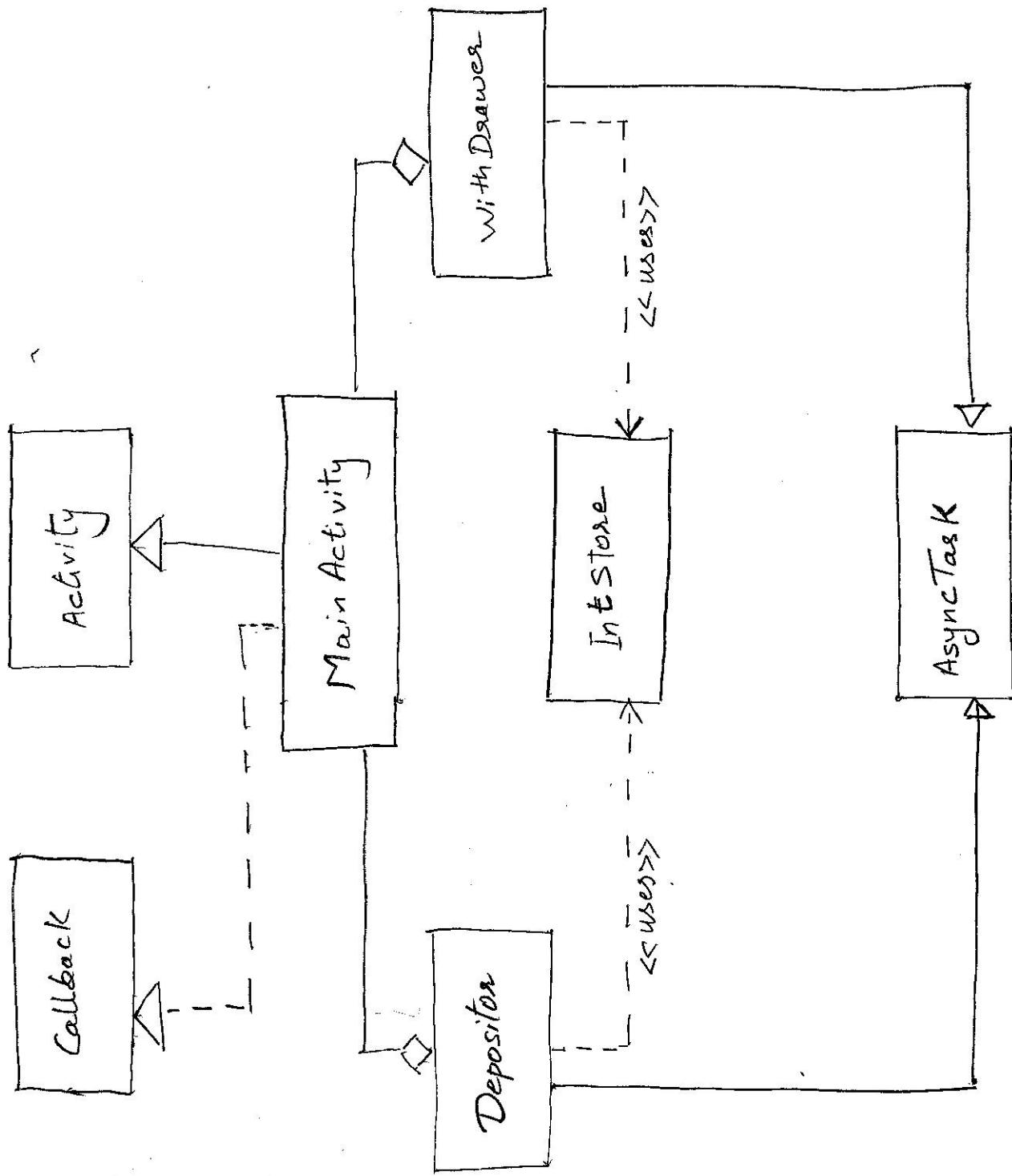
```

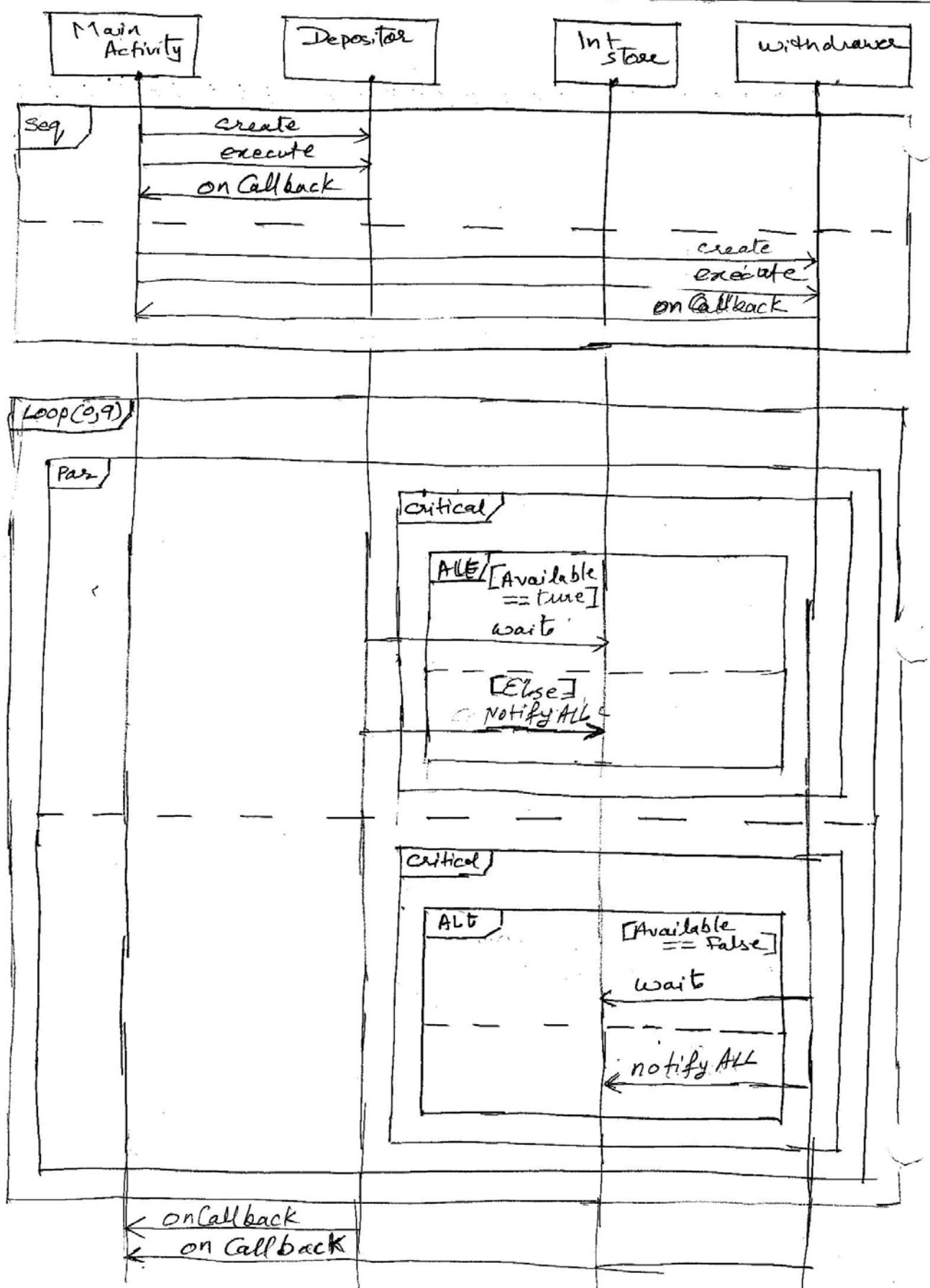
```
        wait();
    } catch (InterruptedException e) { }
}
contents = value;
available = true;
notifyAll();
}
}

package com.example.server.sampleapp;
import android.os.AsyncTask;
import android.util.Log;
public class Withdrawer extends AsyncTask<IntStore, Integer, String> {
    Callback callback;
    public Withdrawer(Callback callback){
        this.callback = callback;
    }
    @Override
    protected void onPreExecute() {
        callback.onCallback("Withdrawer Start");
    }
    @Override
    protected String doInBackground(IntStore... intStores) {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = intStores[0].withdraw();
            publishProgress(value);
        }
        return "End";
    }
    @Override
    protected void onPostExecute(String message) {
        callback.onCallback("Withdrawer " + message);
    }
    @Override
    protected void onProgressUpdate(Integer... progress) {
        Log.i("Withdrawn: ", String.valueOf(progress[0]));
    }
}
```

```
package com.example.server.sampleapp;
import android.os.AsyncTask;
import android.util.Log;
class Depositor extends AsyncTask<IntStore, Integer, String> {
    Callback callback;
    public Depositor(Callback callback){
        this.callback = callback;
    }
    @Override
    protected void onPreExecute() {
        callback.onCallback("Depositor Start");
    }
    @Override
    protected String doInBackground(IntStore... intStores) {
        for (int i = 0; i < 10; i++) {
            intStores[0].deposit(i);
            publishProgress(i);
            try {
                Thread.sleep((int) (Math.random() * 100));
            } catch (InterruptedException e) {
            }
        }
        return "End";
    }
    @Override
    protected void onPostExecute(String message) {
        callback.onCallback("Depositor " + message);
    }
    @Override
    protected void onProgressUpdate(Integer... progress) {
        Log.i("Depositer", String.valueOf(progress[0]));
    }
}
```

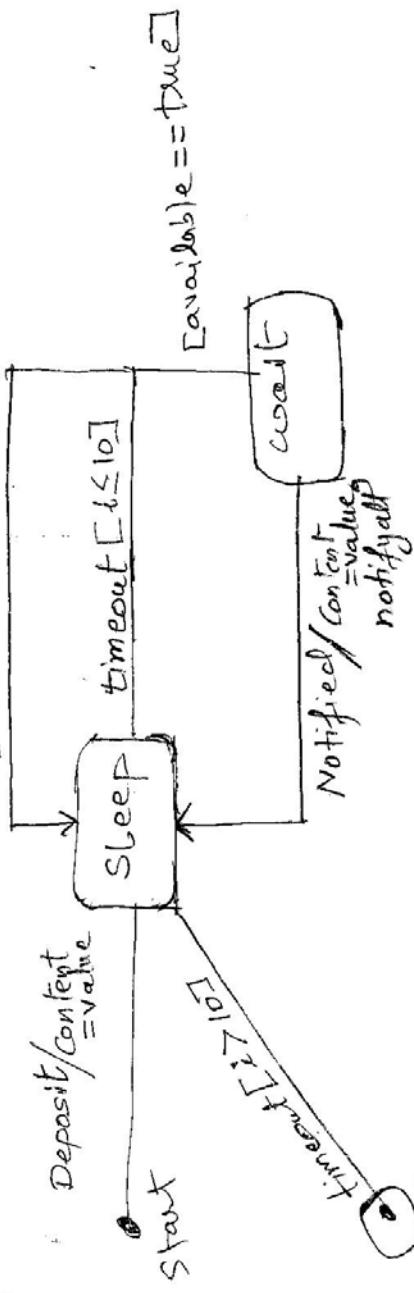
CONFIDENTIAL





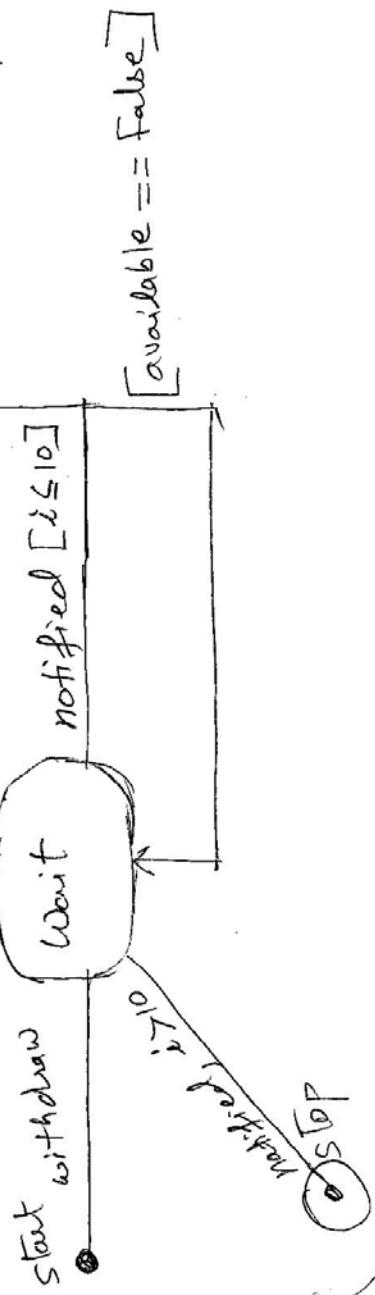
Producer

[available == false] / content [= value]



Consumer

[available == true] / return content



Assignment 1

Suppose following needs were expressed by a focus group:

"As a hobbyist photographer, I would like to tag or caption my smartphone pictures effectively so that I am able to search them conveniently".

"Additionally, as a professional travel blogger, I would also like to upload select pictures to my travel blog (web) site where visitors are also able to search pictures based on categories that would be obvious/relevant to other travelers".

Q1. [2 marks] Derive a set of user stories from these problem statements. Include the associated acceptance tests for each user story. Your marks will be based on the viability of the acceptance tests.

Q2. [2 marks] Capture the functionality as a UML Use Case Diagram. Your marks will be based on how effectively you use the possible relationships among actors and use cases in the use case diagrams.

Solution:

User Story 1

As a <photographer> I want to <capture pictures using my smartphone that are automatically timestamped, geotagged and given a globally unique ID> so that <I can view them later and reminisce>

User Story 2

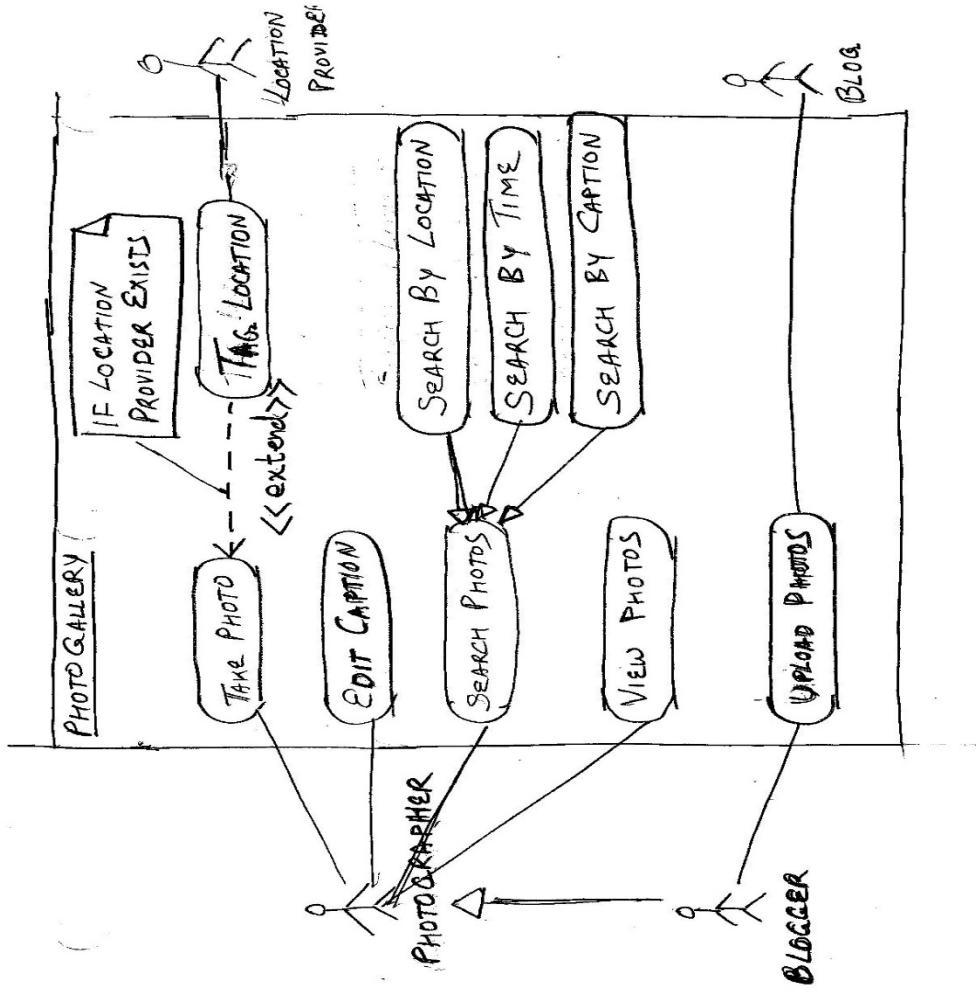
As a <photographer> I would like to <assign a caption for any captured picture> so that <I can search them by specifying keywords>

User Story 3

As a <photographer> I would like to <search pictures based on time, location and/or keywords> so that <I can view pictures from my collection that I am interested in>

User Story 4

As a <blogger> I <want pictures to be uploaded to twitter> so that <it reaches larger audience>



Assignment II

[Part A] Assuming that you have captured the desired functionality of PhotoGallery app as a set of user stories, provide a comprehensive acceptance criteria for some of the high priority user stories so that they are ready for development. The acceptance criteria should be spelled out in user friendly GWT (Given-When-Then) Gherkin style. In case the user stories you wrote are somewhat different from the ones that I suggested in the class, the high priority user stories would be ones that are functionally equivalent to the first 4 (out of the total 5) user stories that I suggested. In other words, the user story that captured the requirement of uploading the photos to the social media sites, could be delayed for a later sprint. [2 marks]

[Part B] Also, sketch out a visual narrative, in the form of wireframes and storyboard, depicting how a user will interact with your app. [2 marks]

Solution [Part A]:

Feature: Take Picture

- Scenario: User Story 1

Given I am at the gallery screen of the PhotoGallery app
When I press the button labeled “Snap”
Then I see the view of the android camera app
When I take a picture using the android camera app
Then I go back to the main screen of the PhotoGallery app
And see the picture just taken
And the current time
And the current location
And a globally unique picture ID

Feature: Assign Caption

- Scenario: User Story 2

Given I am at the Gallery screen of the PhotoGallery app
And I see an editable text view labeled “caption” next to the picture
When I edit the caption by typing “My Test Caption”
Then I see the new caption “My Test Caption”
When I scroll away from the picture
And scroll back to the picture
Then I see the new caption “My Test Caption”

Feature: Search Pictures

- Scenario: User Story 3

Given I am at the Search screen of the PhotoGallery app
And I see StartDate Calendar view
And I see EndDate Calendar view
When I select Today as StartDate

And I select Today as EndDate
And I press the button labeled "Search"
Then I see the gallery screen of the PhotoGallery app
And I see the pictures that have today as the creation date

Given I am at the Search screen of the PhotoGallery app
And I see Google Map view to select the Top Left corner of the search area
And I see Google Map view to select the Bottom Right corner of the search area
When I select the Top Left corner of the search area that includes the current location
And I select Bottom Right corner of the search area that includes the current location
And I press the button labeled "Search"
Then I see the gallery screen of the PhotoGallery app
And I see the pictures that were taken at the current location

Given I am at the Search screen of the PhotoGallery app
And I see editable text view labeled "keywords"
When I type characters "My Test Caption"
And I press the button labeled "Search"
Then I see the gallery screen of the PhotoGallery app
And I see the pictures with the Caption "My Test Caption"

Feature: Upload Picture

Scenario: User Story 4

Given I am at the gallery screen of the PhotoGallery app
And the smartphone indicates wireless coverage
And I see the button labeled "Upload"
And the button "Upload" is enabled.
When I press the button "Upload"
Then I see the button change its label first to "Uploading"
And then to "Uploaded"
And become disabled.
When I check my blog
Then I see the picture on my blog.

Solution [Part B]

It is expected that this android application would contain at least the following activities.

Main Activity should be composed of an ImageView with a navigation button on each side to scroll the images left and right. In addition, the activity should have snap, enlarge, filter and settings buttons. A multi-select checkbox list should also exist to view/add tags/keywords for each picture being displayed.

Snap button shall send an intent to the onboard camera app to take a picture, and perhaps also specify additional tags/keywords.

Enlarge button shall display the same picture on a new activity with an image view but enlarged.

Filter button shall present the user with an activity that allows the user to select the pictures to view based on geographical area, time span and/or keywords/tags.

Settings activity shall allow the user to specify the path of the folder where the pictures are stored. In addition it should also allow the user to add to its list of tags or keywords.

What is software?

- Computer programs and associated documentation such as requirements, design models and user manuals.
- Software products may be developed for a particular customer or may be developed for a general market.
- Software products may be
 - Generic - developed to be sold to a range of different customers e.g. PC software such as Excel or Word.
 - Bespoke (custom) - developed for a single customer according to their specification.
- New software can be created by developing new programs, configuring generic software systems or reusing existing software.

What is software engineering?

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- Software engineers should adopt a systematic and organised approach to their work and use appropriate tools and techniques depending on the problem to be solved, the development constraints and the resources available.

What is the difference between software engineering and computer science?

- Computer science is concerned with theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
- Computer science theories are still insufficient to act as a complete underpinning for software engineering (unlike e.g. physics and electrical engineering).

What is the difference between software engineering and system engineering?

- System engineering is concerned with all aspects of computer-based systems development including hardware, software and process engineering. Software engineering is part of this process concerned with developing the software infrastructure, control, applications and databases in the system.
- System engineers are involved in system specification, architectural design, integration and deployment.

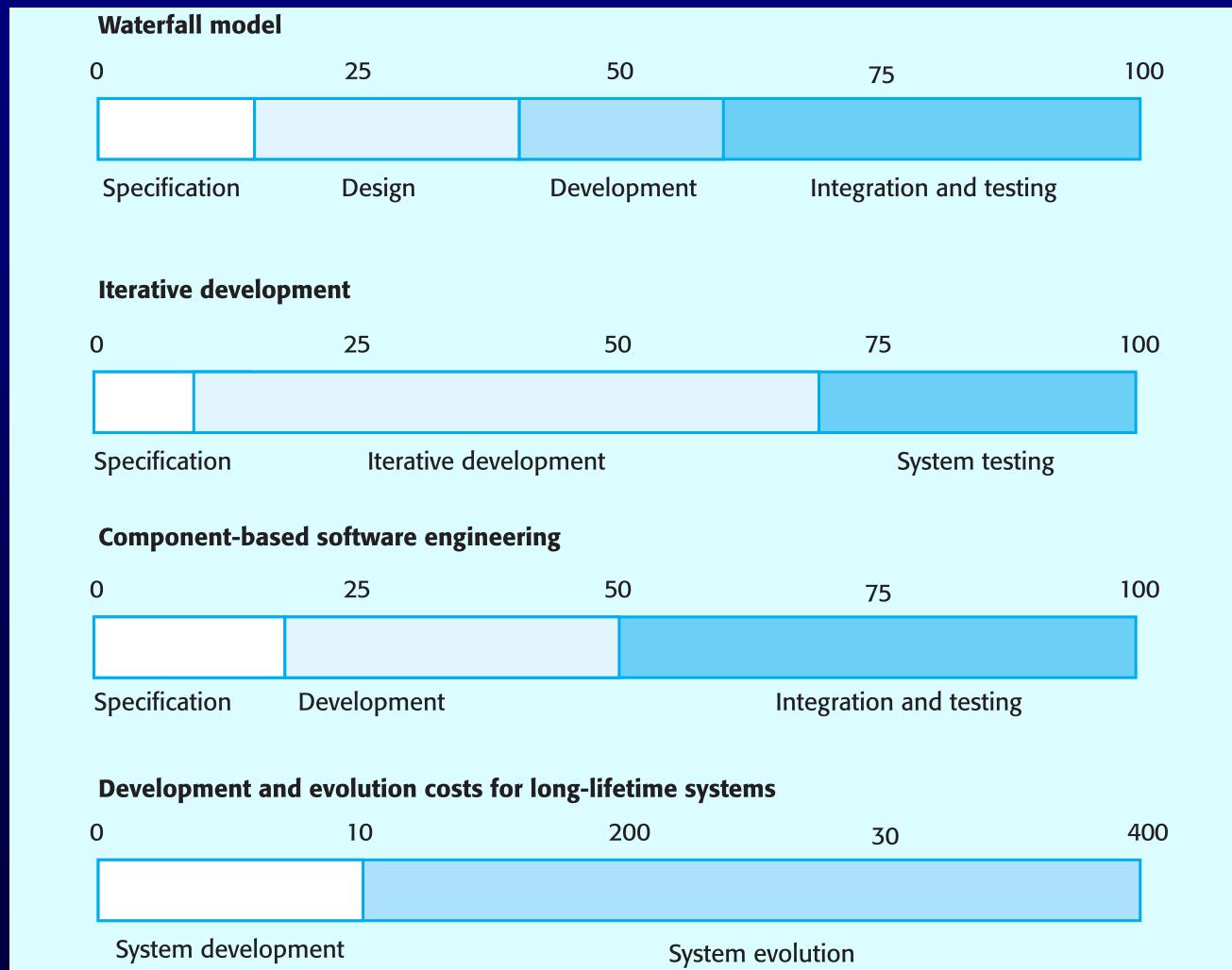
What is a software process?

- A set of activities whose goal is the development or evolution of software.
- Generic activities in all software processes are:
 - Specification - what the system should do and its development constraints
 - Development - production of the software system
 - Validation - checking that the software is what the customer wants
 - Evolution - changing the software in response to changing demands.

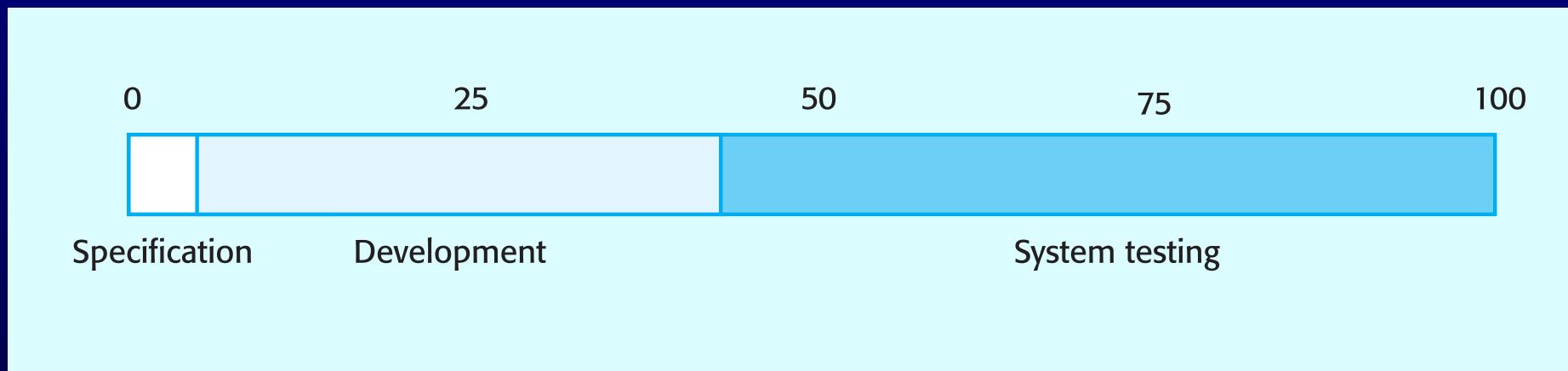
What is a software process model?

- A simplified representation of a software process, presented from a specific perspective.
- Examples of process perspectives are
 - Workflow perspective - sequence of activities;
 - Data-flow perspective - information flow;
 - Role/action perspective - who does what.
- Generic process models
 - Waterfall;
 - Iterative development;
 - Component-based software engineering.

Activity cost distribution



Product development costs



What are software engineering methods?

- Structured approaches to software development which include system models, notations, rules, design advice and process guidance.
- Model descriptions
 - Descriptions of graphical models which should be produced;
- Rules
 - Constraints applied to system models;
- Recommendations
 - Advice on good design practice;
- Process guidance
 - What activities to follow.

What is CASE (Computer-Aided Software Engineering)

- Software systems that are intended to provide automated support for software process activities.
- CASE systems are often used for method support.
- Upper-CASE
 - Tools to support the early process activities of requirements and design;
- Lower-CASE
 - Tools to support later activities such as programming, debugging and testing.

What are the attributes of good software?

- The software should deliver the required functionality and performance to the user and should be maintainable, dependable and acceptable.
- Maintainability
 - Software must evolve to meet changing needs;
- Dependability
 - Software must be trustworthy;
- Efficiency
 - Software should not make wasteful use of system resources;
- Acceptability
 - Software must be accepted by the users for which it was designed. This means it must be understandable, usable and compatible with other systems.

What are the key challenges facing software engineering?

- Heterogeneity, delivery and trust.
- Heterogeneity
 - Developing techniques for building software that can cope with heterogeneous platforms and execution environments;
- Delivery
 - Developing techniques that lead to faster delivery of software;
- Trust
 - Developing techniques that demonstrate that software can be trusted by its users.

Key points

- Software engineering is an engineering discipline that is concerned with all aspects of software production.
- Software products consist of developed programs and associated documentation. Essential product attributes are maintainability, dependability, efficiency and usability.
- The software process consists of activities that are involved in developing software products. Basic activities are software specification, development, validation and evolution.
- Methods are organised ways of producing software. They include suggestions for the process to be followed, the notations to be used, rules governing the system descriptions which are produced and design guidelines.

Key points

- CASE tools are software systems which are designed to support routine activities in the software process such as editing design diagrams, checking diagram consistency and keeping track of program tests which have been run.
- Software engineers have responsibilities to the engineering profession and society. They should not simply be concerned with technical issues.
- Professional societies publish codes of conduct which set out the standards of behaviour expected of their members.

Software Processes

Objectives

- To introduce software process models
- To describe three generic process models and when they may be used
- To describe outline process models for requirements engineering, software development, testing and evolution
- To explain the Rational Unified Process model
- To introduce CASE technology to support software process activities

Topics covered

- Software process models
- Process iteration
- Process activities
- The Rational Unified Process
- Computer-aided software engineering

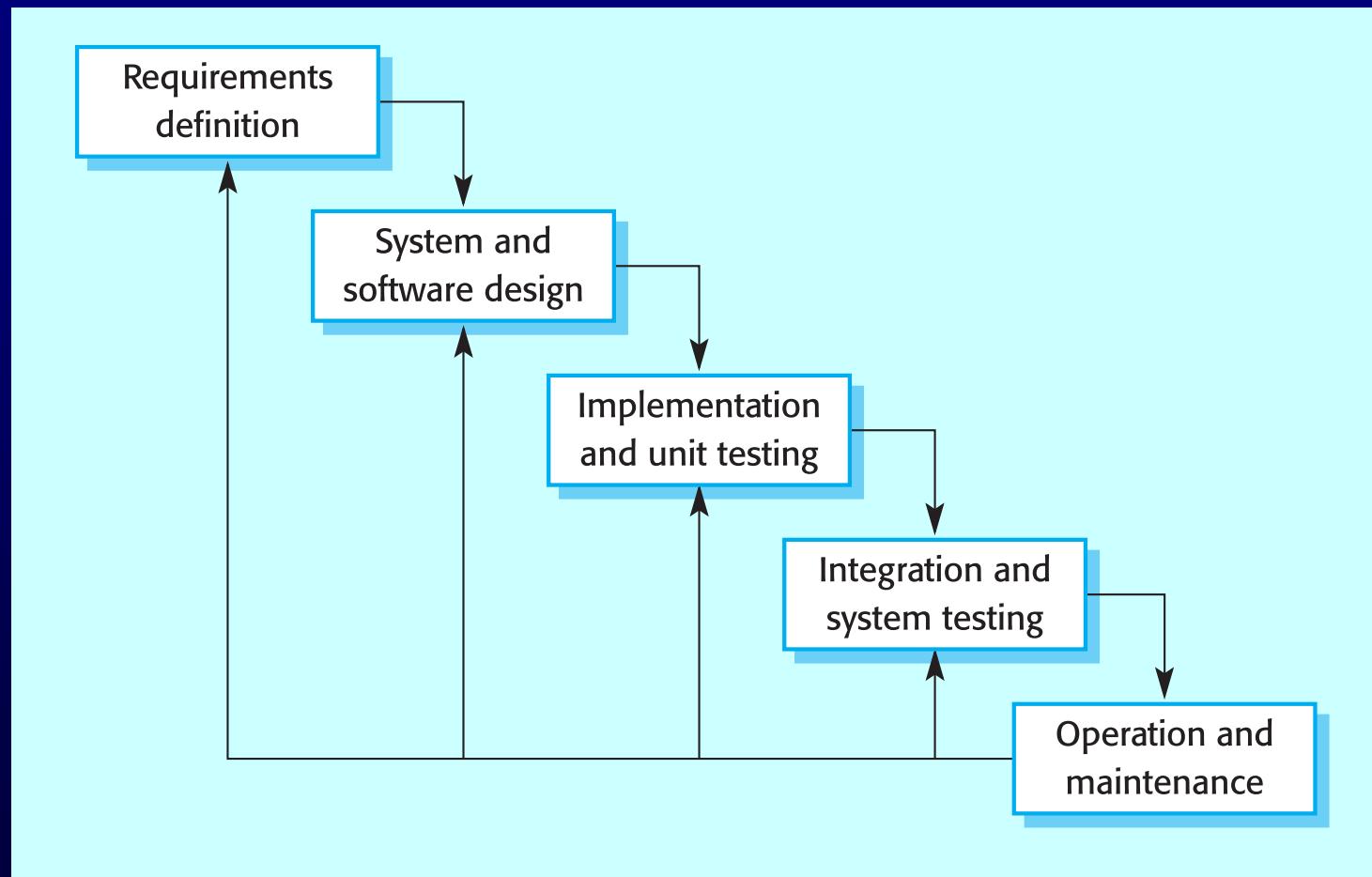
The software process

- A structured set of activities required to develop a software system
 - Specification;
 - Design;
 - Validation;
 - Evolution.
- A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

Generic software process models

- The waterfall model
 - Separate and distinct phases of specification and development.
- Evolutionary development
 - Specification, development and validation are interleaved.
- Component-based software engineering
 - The system is assembled from existing components.
- There are many variants of these models e.g. formal development where a waterfall-like process is used but the specification is a formal specification that is refined through several stages to an implementable design.

Waterfall model



Waterfall model phases

- Requirements analysis and definition
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance
- The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway. One phase has to be complete before moving onto the next phase.

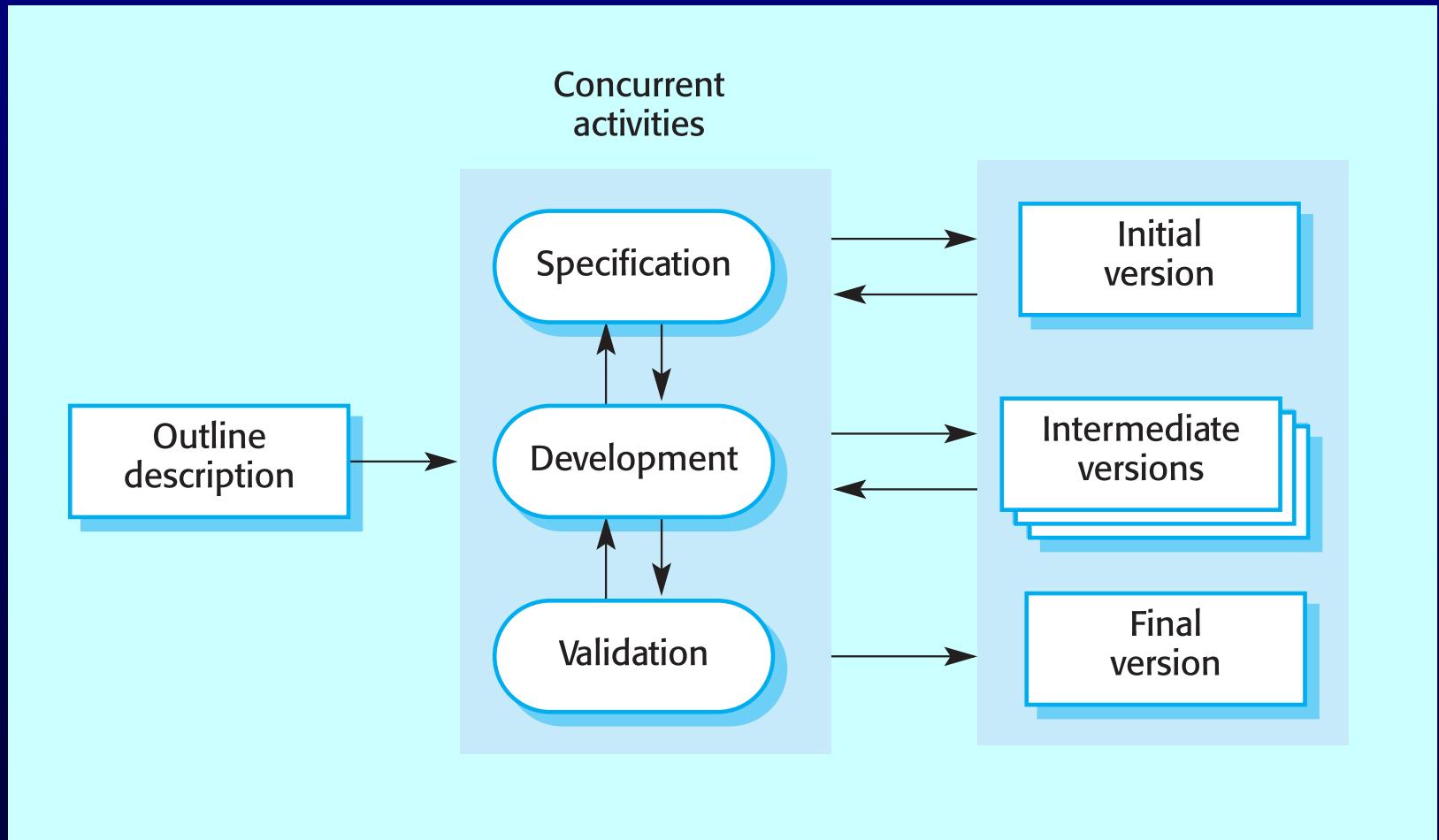
Waterfall model problems

- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
- Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
- Few business systems have stable requirements.
- The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.

Evolutionary development

- Exploratory development
 - Objective is to work with customers and to evolve a final system from an initial outline specification. Should start with well-understood requirements and add new features as proposed by the customer.
- Throw-away prototyping
 - Objective is to understand the system requirements. Should start with poorly understood requirements to clarify what is really needed.

Evolutionary development



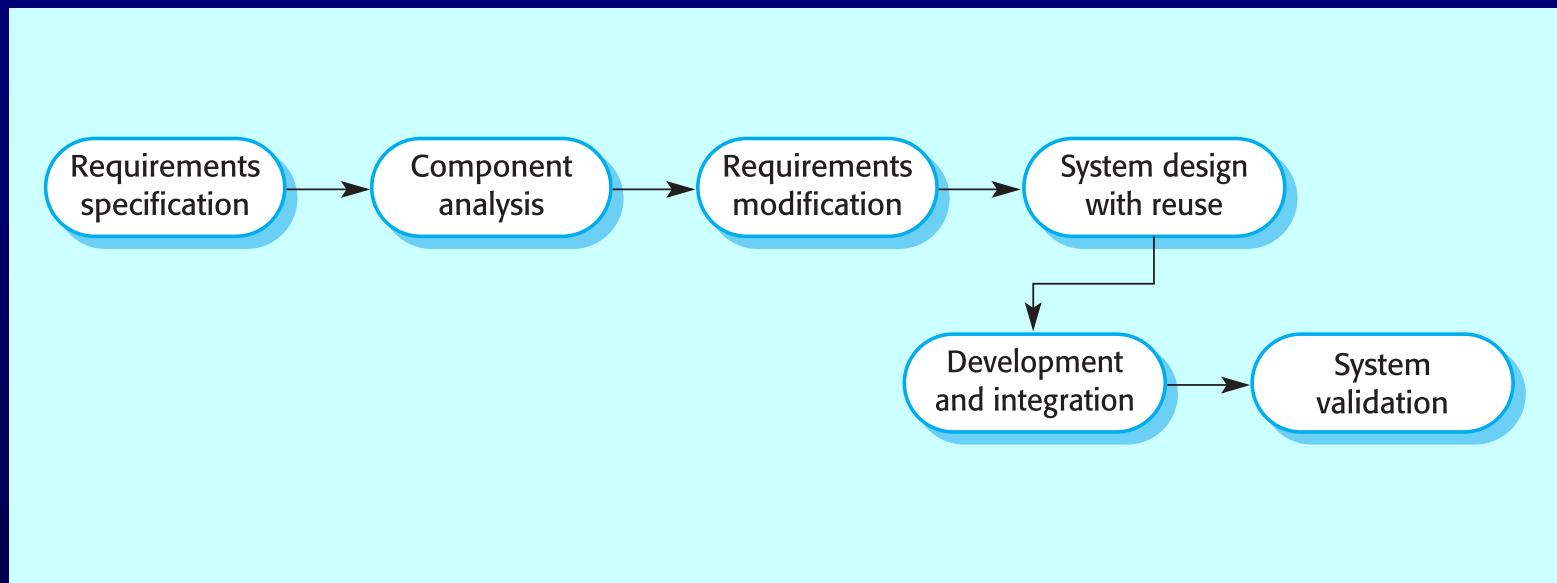
Evolutionary development

- Problems
 - Lack of process visibility;
 - Systems are often poorly structured;
 - Special skills (e.g. in languages for rapid prototyping) may be required.
- Applicability
 - For small or medium-size interactive systems;
 - For parts of large systems (e.g. the user interface);
 - For short-lifetime systems.

Component-based software engineering

- Based on systematic reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems.
- Process stages
 - Component analysis;
 - Requirements modification;
 - System design with reuse;
 - Development and integration.
- This approach is becoming increasingly used as component standards have emerged.

Reuse-oriented development



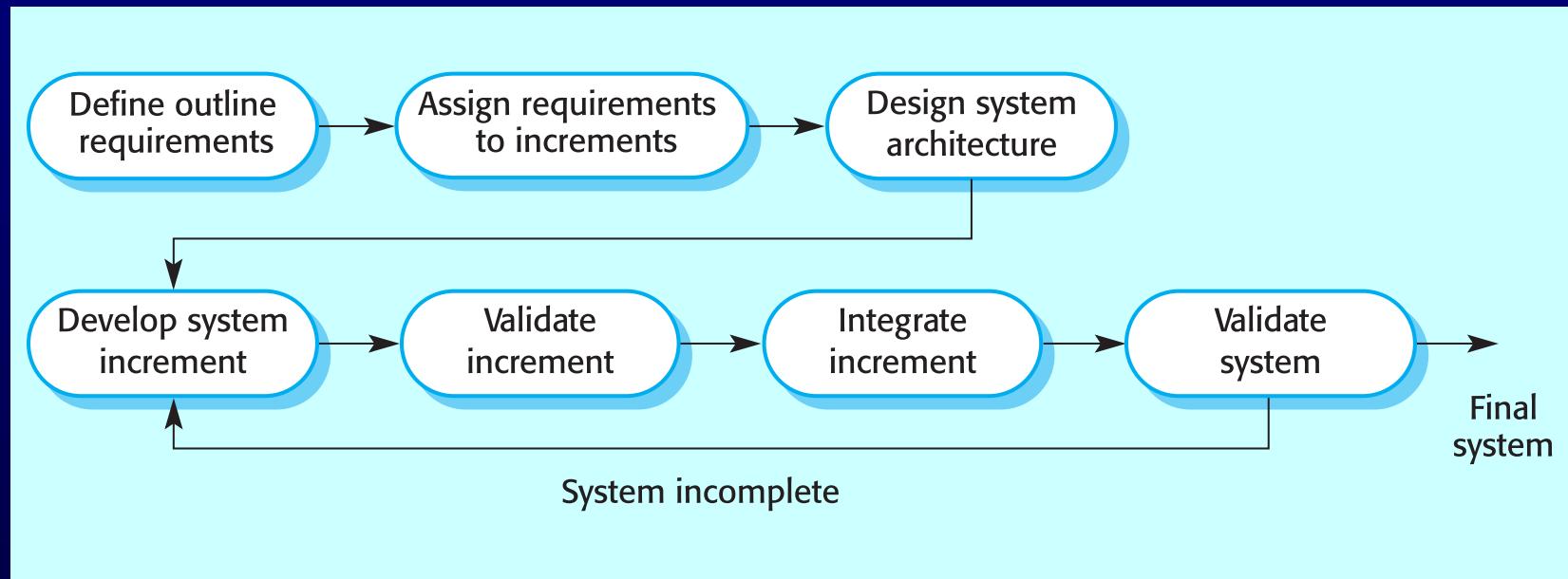
Process iteration

- System requirements **ALWAYS** evolve in the course of a project so process iteration where earlier stages are reworked is always part of the process for large systems.
- Iteration can be applied to any of the generic process models.
- Two (related) approaches
 - Incremental delivery;
 - Spiral development.

Incremental delivery

- Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- User requirements are prioritised and the highest priority requirements are included in early increments.
- Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.

Incremental development



Incremental development advantages

- Customer value can be delivered with each increment so system functionality is available earlier.
- Early increments act as a prototype to help elicit requirements for later increments.
- Lower risk of overall project failure.
- The highest priority system services tend to receive the most testing.

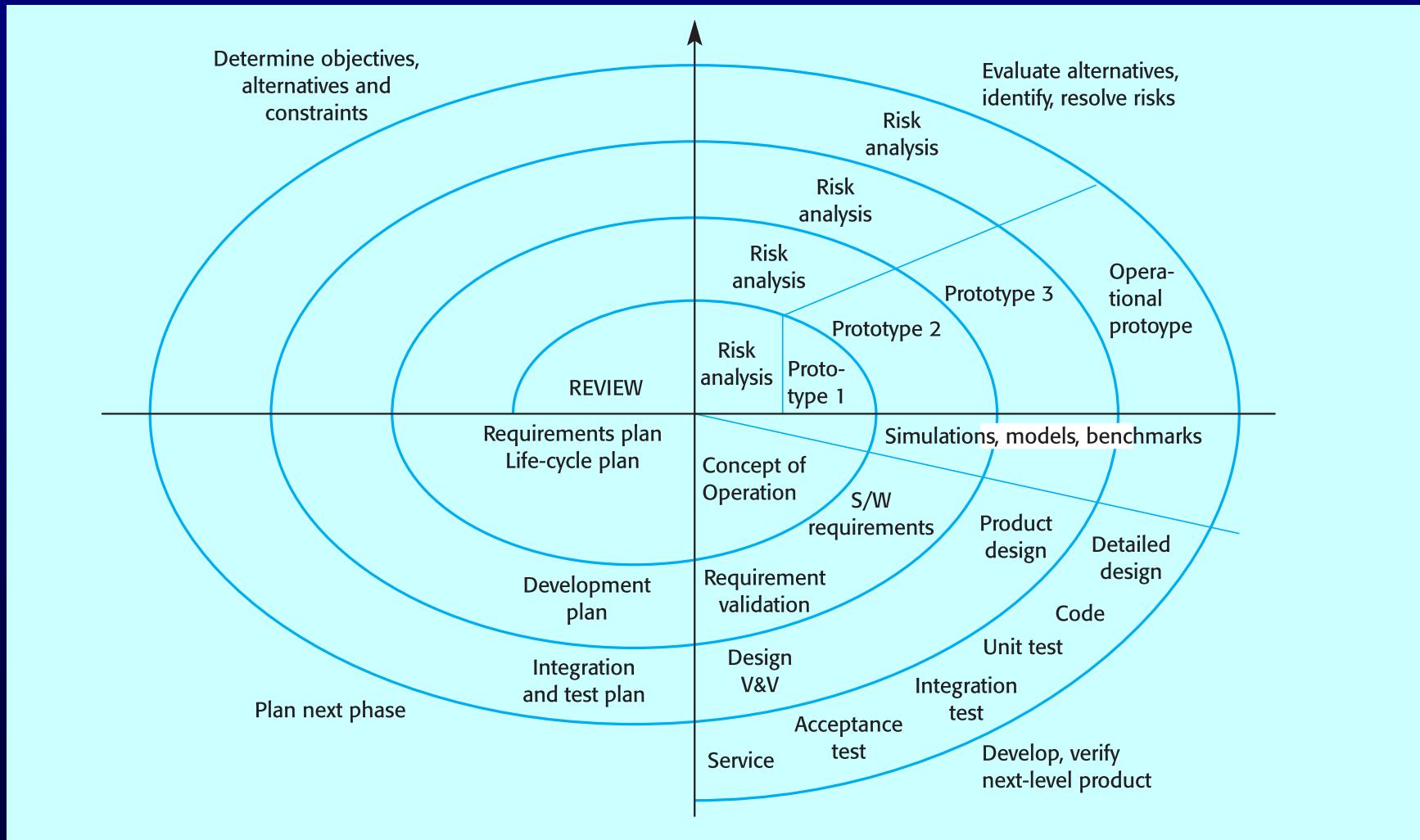
Extreme programming

- An approach to development based on the development and delivery of very small increments of functionality.
- Relies on constant code improvement, user involvement in the development team and pairwise programming.

Spiral development

- Process is represented as a spiral rather than as a sequence of activities with backtracking.
- Each loop in the spiral represents a phase in the process.
- No fixed phases such as specification or design - loops in the spiral are chosen depending on what is required.
- Risks are explicitly assessed and resolved throughout the process.

Spiral model of the software process



Spiral model sectors

- Objective setting
 - Specific objectives for the phase are identified.
- Risk assessment and reduction
 - Risks are assessed and activities put in place to reduce the key risks.
- Development and validation
 - A development model for the system is chosen which can be any of the generic models.
- Planning
 - The project is reviewed and the next phase of the spiral is planned.

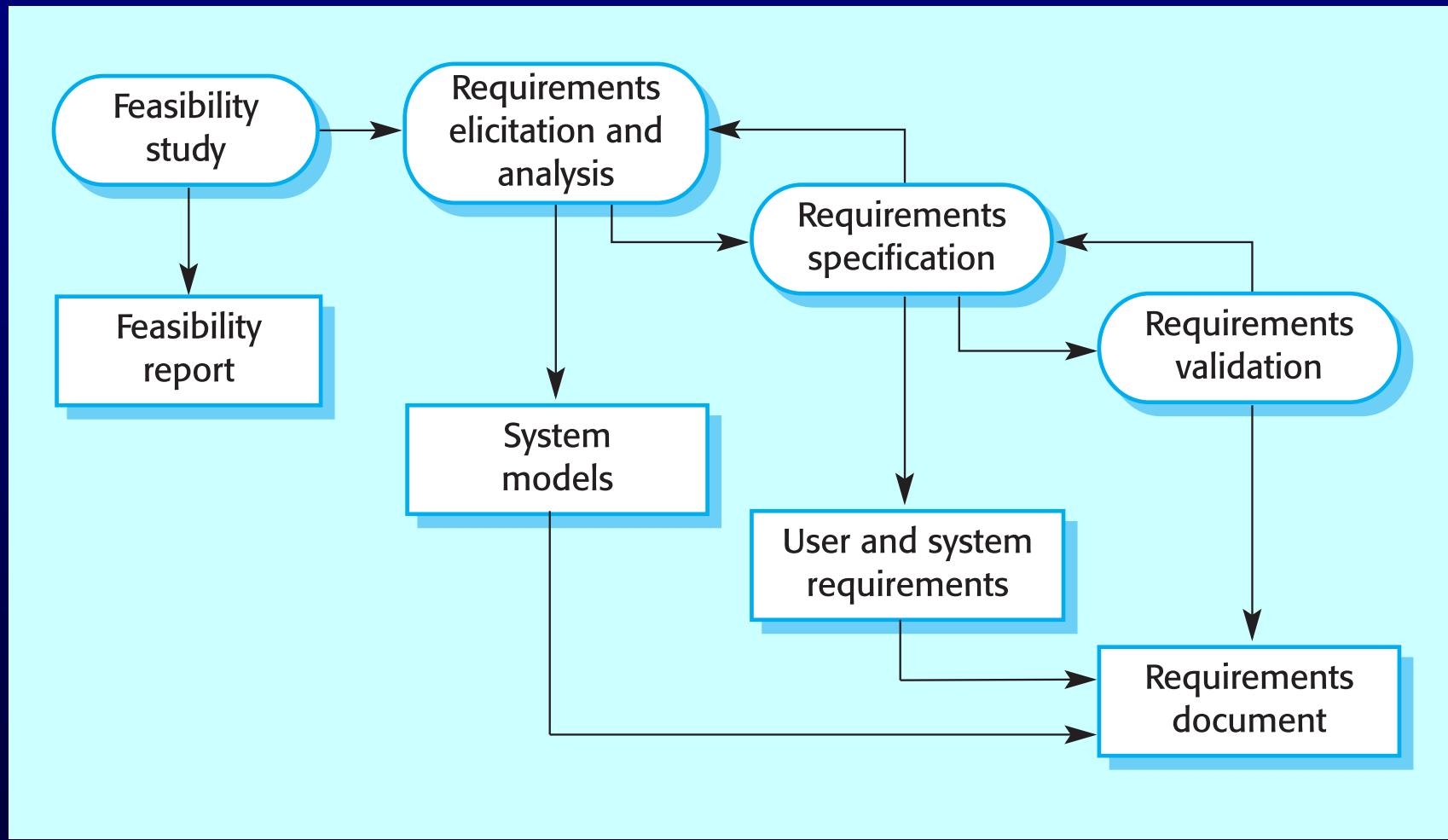
Process activities

- Software specification
- Software design and implementation
- Software validation
- Software evolution

Software specification

- The process of establishing what services are required and the constraints on the system's operation and development.
- Requirements engineering process
 - Feasibility study;
 - Requirements elicitation and analysis;
 - Requirements specification;
 - Requirements validation.

The requirements engineering process



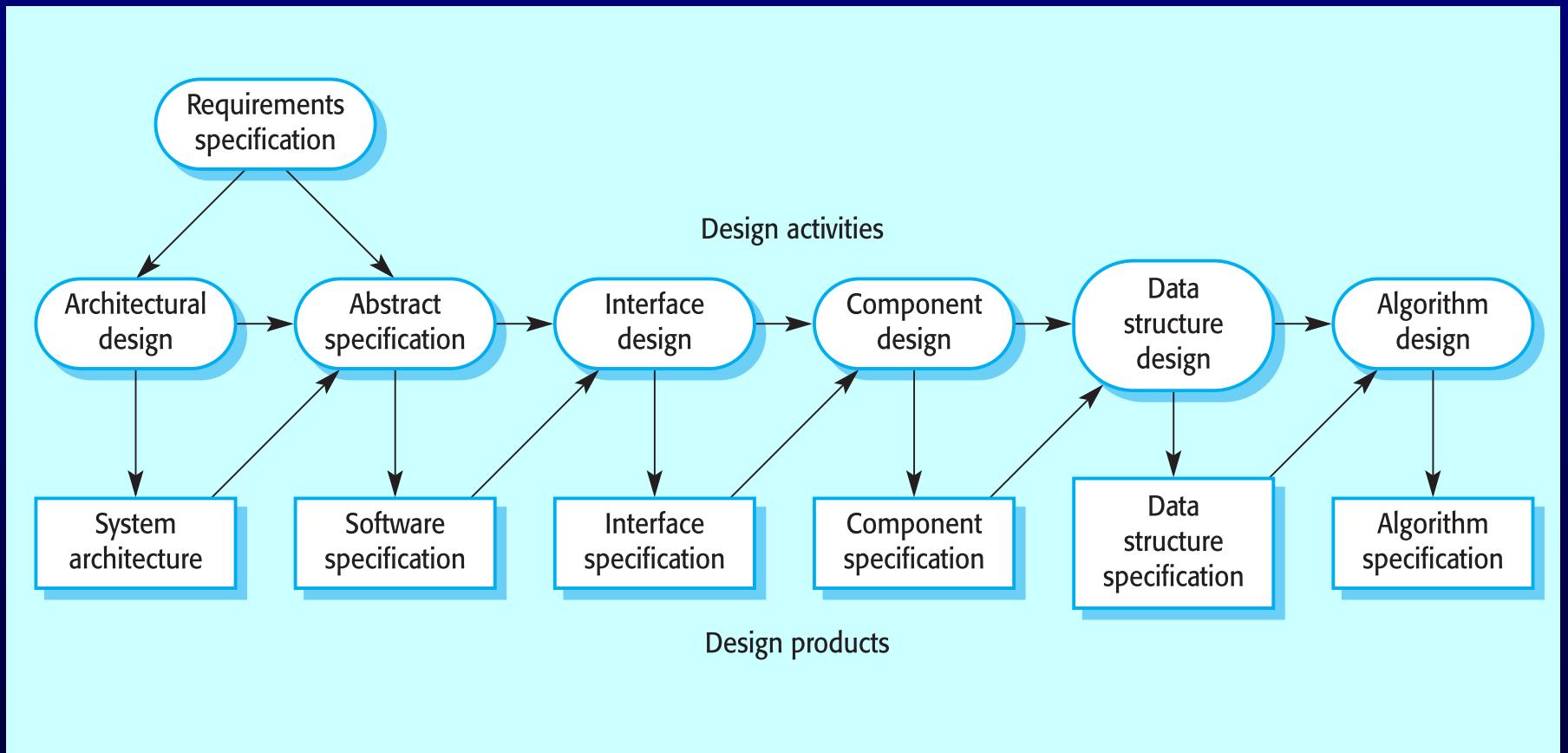
Software design and implementation

- 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



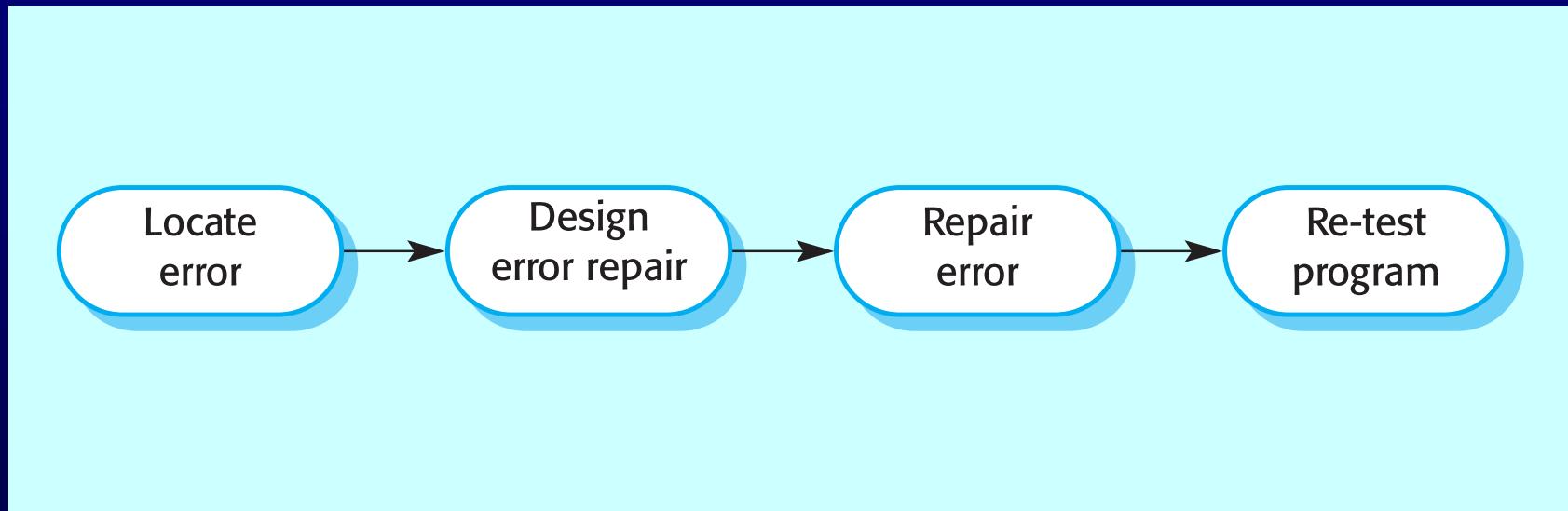
Structured methods

- Systematic approaches to developing a software design.
- The design is usually documented as a set of graphical models.
- Possible models
 - Object model;
 - Sequence model;
 - State transition model;
 - Structural model;
 - Data-flow model.

Programming and debugging

- Translating a design into a program and removing errors from that program.
- Programming is a personal activity - there is no generic programming process.
- Programmers carry out some program testing to discover faults in the program and remove these faults in the debugging process.

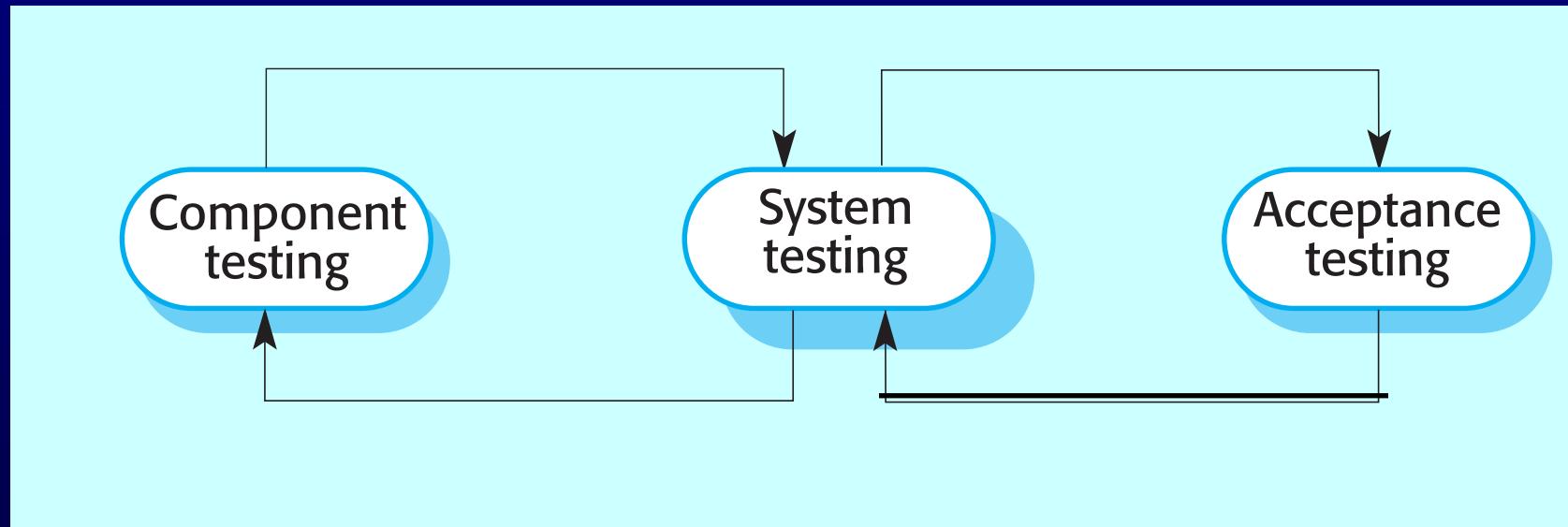
The debugging process



Software validation

- Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- Involves checking and review processes and system testing.
- System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.

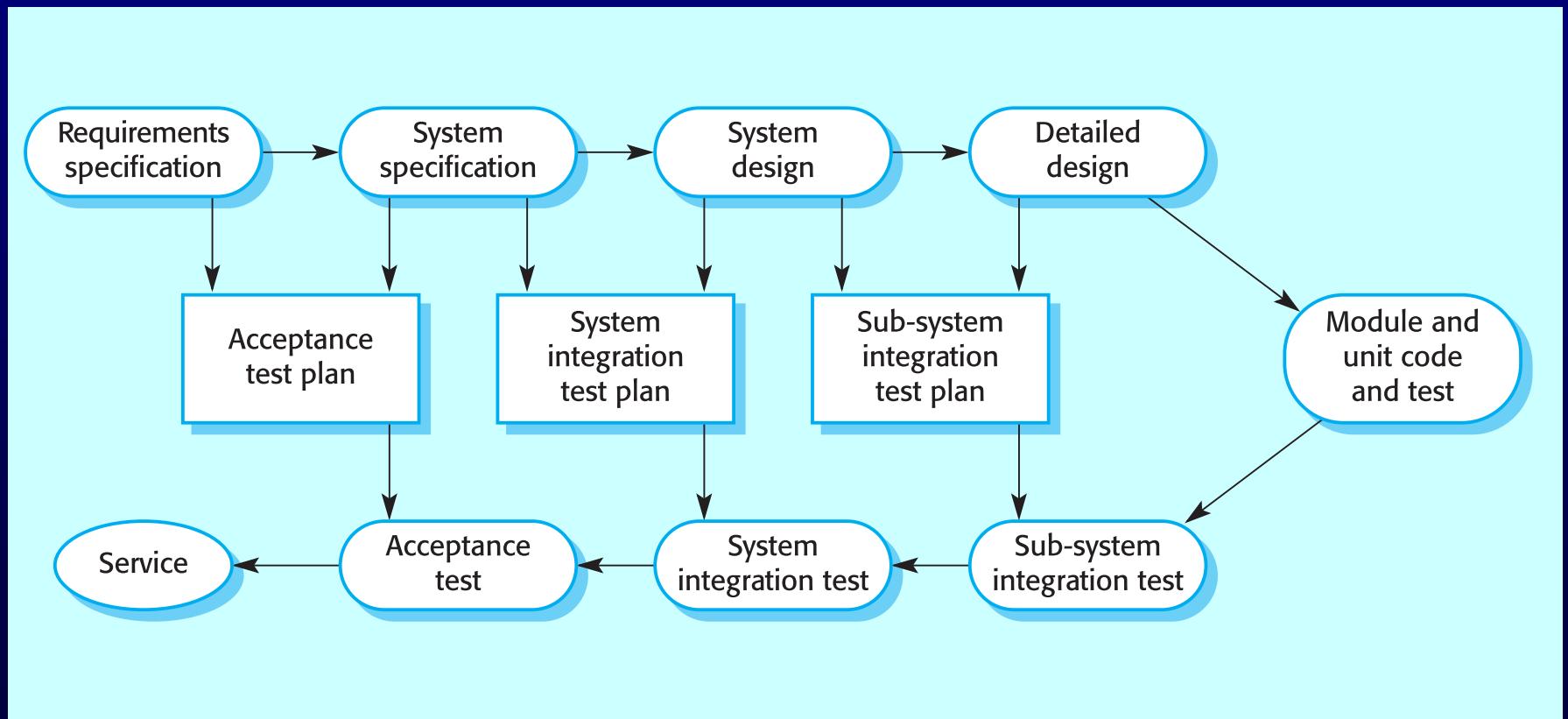
The testing process



Testing stages

- Component or unit testing
 - Individual components are tested independently;
 - Components may be functions or objects or coherent groupings of these entities.
- System testing
 - Testing of the system as a whole. Testing of emergent properties is particularly important.
- Acceptance testing
 - Testing with customer data to check that the system meets the customer's needs.

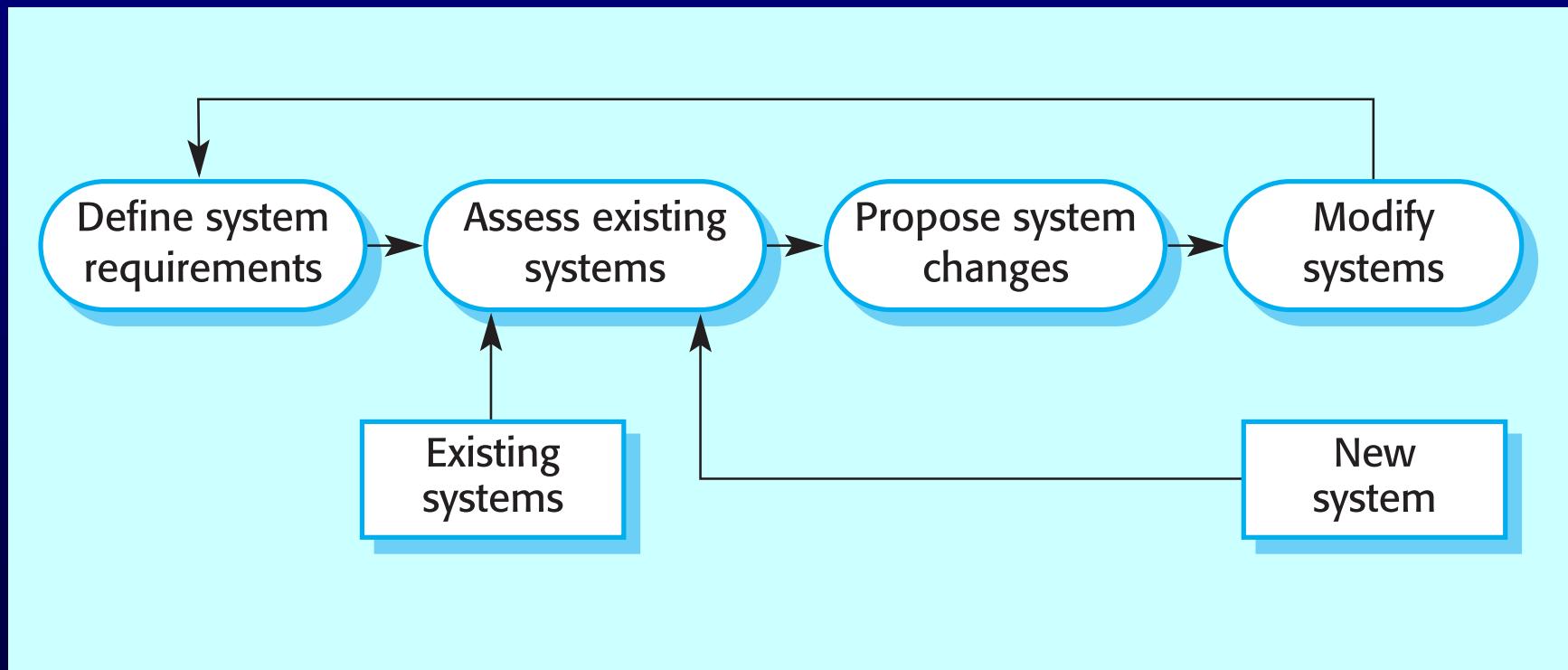
Testing phases



Software evolution

- Software is inherently flexible and can change.
- As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.

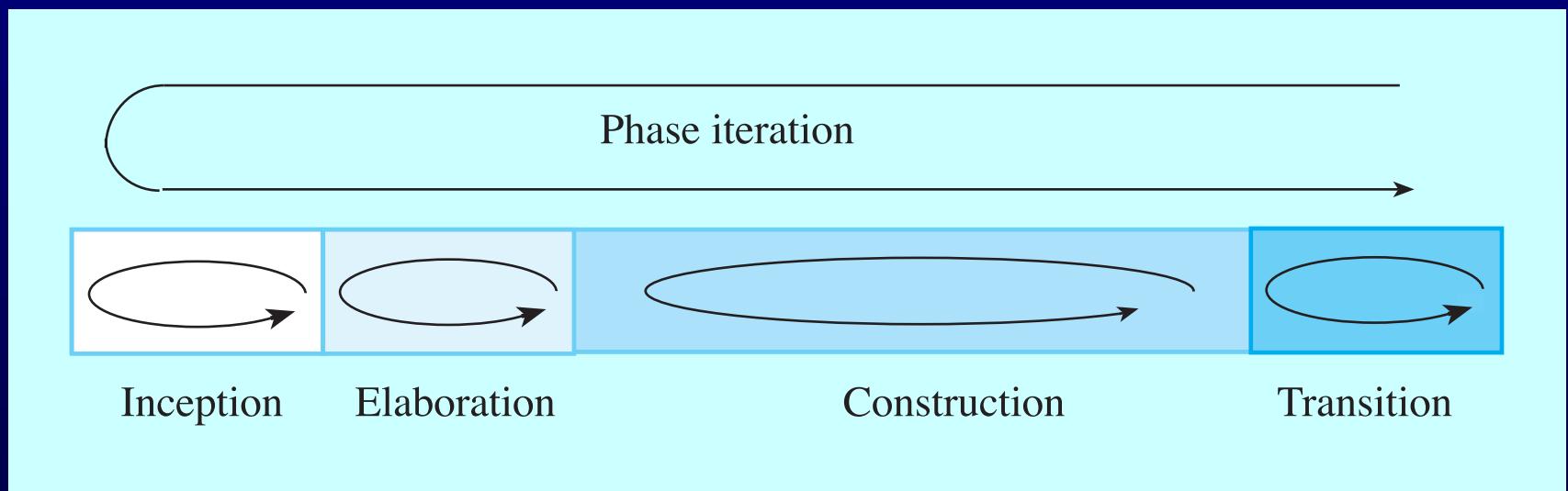
System evolution



The Rational Unified Process

- A modern process model derived from the work on the UML and associated process.
- Normally described from 3 perspectives
 - A dynamic perspective that shows phases over time;
 - A static perspective that shows process activities;
 - A practice perspective that suggests good practice.

RUP phase model



RUP phases

- Inception
 - Establish the business case for the system.
- Elaboration
 - Develop an understanding of the problem domain and the system architecture.
- Construction
 - System design, programming and testing.
- Transition
 - Deploy the system in its operating environment.

RUP good practice

- Develop software iteratively
- Manage requirements
- Use component-based architectures
- Visually model software
- Verify software quality
- Control changes to software

Static workflows

Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Test	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users and installed in their workplace.
Configuration and change management	This supporting workflow manages changes to the system (see Chapter 29).
Project management	This supporting workflow manages the system development (see Chapter 5).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

Computer-aided software engineering

- Computer-aided software engineering (CASE) is software to support software development and evolution processes.
- Activity automation
 - Graphical editors for system model development;
 - Data dictionary to manage design entities;
 - Graphical UI builder for user interface construction;
 - Debuggers to support program fault finding;
 - Automated translators to generate new versions of a program.

Case technology

- Case technology has led to significant improvements in the software process. However, these are not the order of magnitude improvements that were once predicted
 - Software engineering requires creative thought - this is not readily automated;
 - Software engineering is a team activity and, for large projects, much time is spent in team interactions. CASE technology does not really support these.

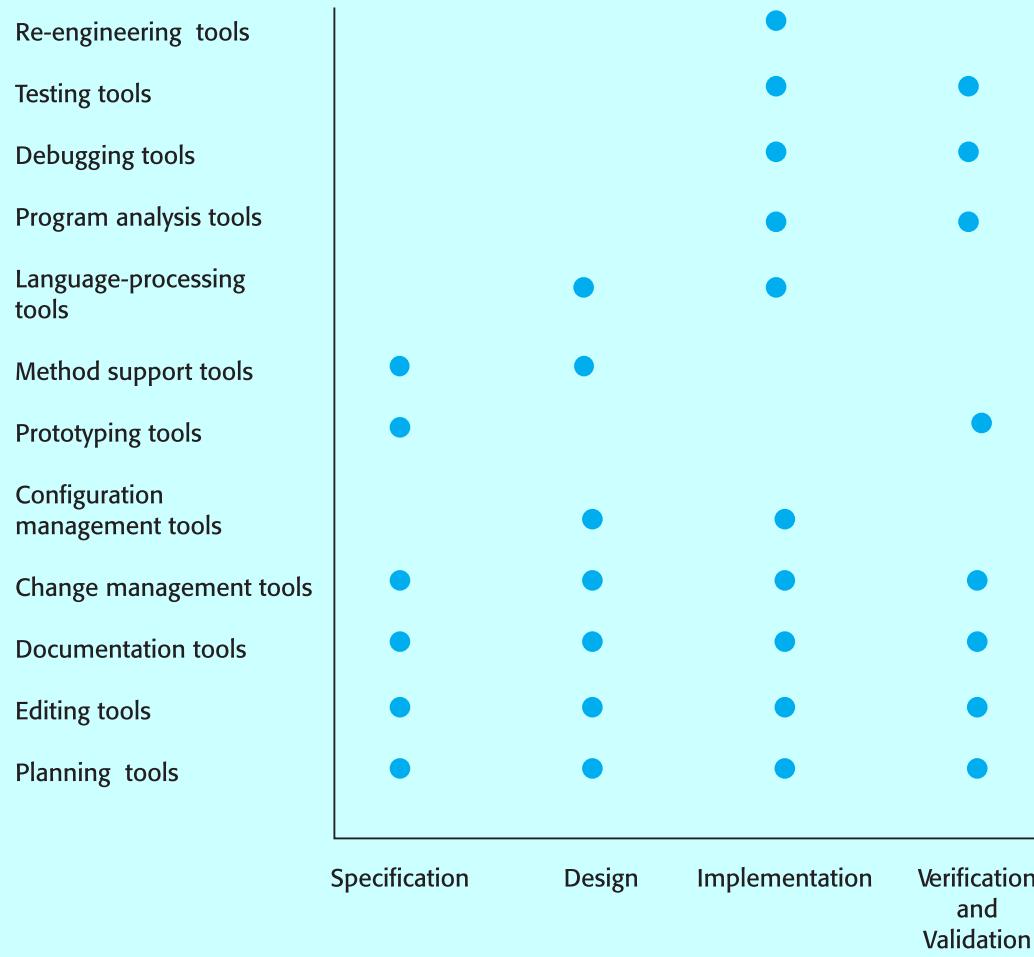
CASE classification

- Classification helps us understand the different types of CASE tools and their support for process activities.
- Functional perspective
 - Tools are classified according to their specific function.
- Process perspective
 - Tools are classified according to process activities that are supported.
- Integration perspective
 - Tools are classified according to their organisation into integrated units.

Functional tool classification

Tool type	Examples
Planning tools	PERT tools, estimation tools, spreadsheets
Editing tools	Text editors, diagram editors, word processors
Change management tools	Requirements traceability tools, change control systems
Configuration management tools	Version management systems, system building tools
Prototyping tools	Very high-level languages, user interface generators
Method-support tools	Design editors, data dictionaries, code generators
Language-processing tools	Compilers, interpreters
Program analysis tools	Cross reference generators, static analysers, dynamic analysers
Testing tools	Test data generators, file comparators
Debugging tools	Interactive debugging systems
Documentation tools	Page layout programs, image editors
Re-engineering tools	Cross-reference systems, program re-structuring systems

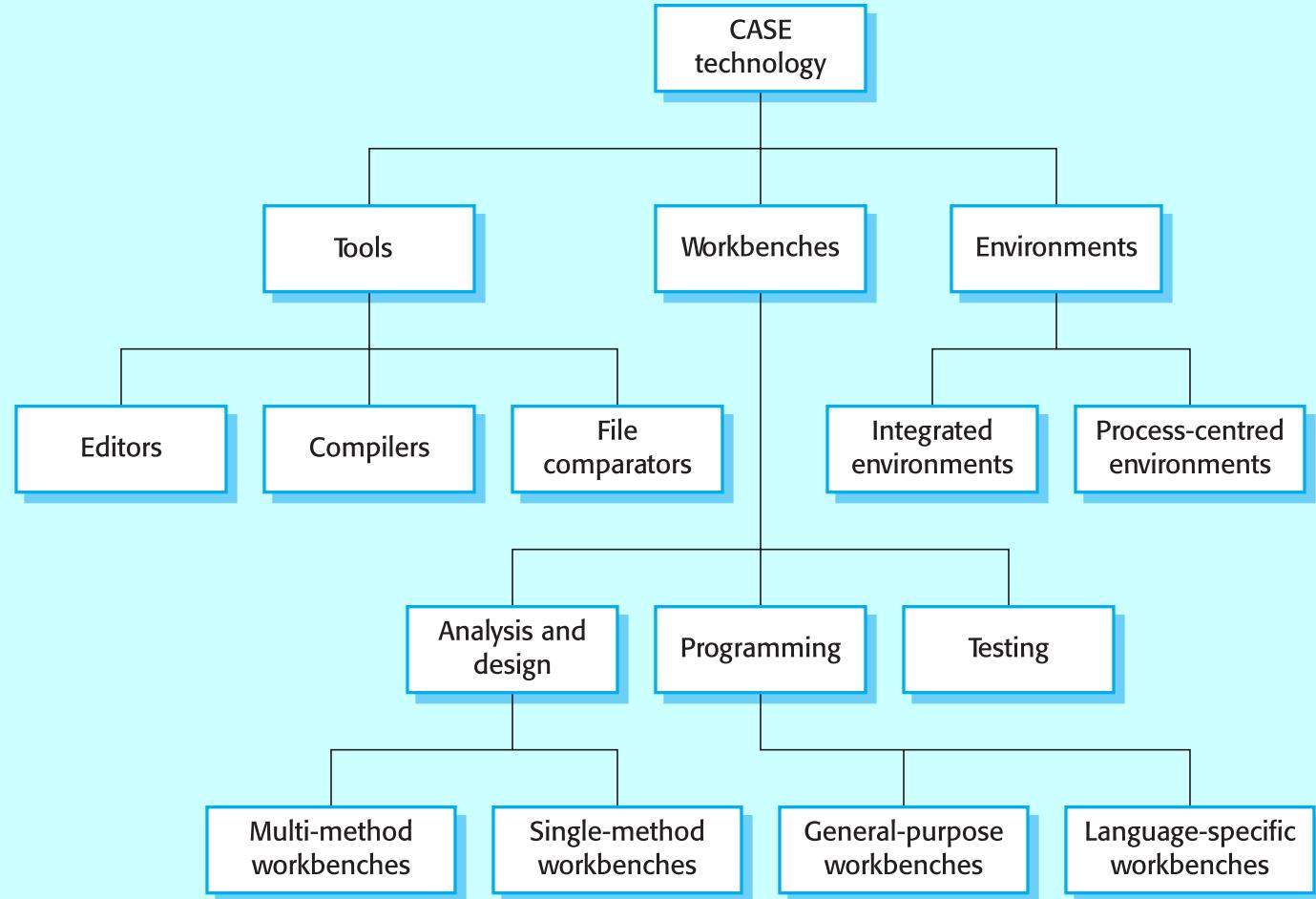
Activity-based tool classification



CASE integration

- Tools
 - Support individual process tasks such as design consistency checking, text editing, etc.
- Workbenches
 - Support a process phase such as specification or design, Normally include a number of integrated tools.
- Environments
 - Support all or a substantial part of an entire software process. Normally include several integrated workbenches.

Tools, workbenches, environments



Key points

- Software processes are the activities involved in producing and evolving a software system.
- Software process models are abstract representations of these processes.
- General activities are specification, design and implementation, validation and evolution.
- Generic process models describe the organisation of software processes. Examples include the waterfall model, evolutionary development and component-based software engineering.
- Iterative process models describe the software process as a cycle of activities.

Key points

- Requirements engineering is the process of developing a software specification.
- Design and implementation processes transform the specification to an executable program.
- Validation involves checking that the system meets to its specification and user needs.
- Evolution is concerned with modifying the system after it is in use.
- The Rational Unified Process is a generic process model that separates activities from phases.
- CASE technology supports software process activities.

Software Requirements

Requirements engineering

- The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.

What is a requirement?

- It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- This is inevitable as requirements may serve a dual function
 - May be the basis for a bid for a contract - therefore must be open to interpretation;
 - May be the basis for the contract itself - therefore must be defined in detail;
 - Both these statements may be called requirements.

Types of requirement

- User requirements
 - Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
- System requirements
 - A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

Definitions and specifications

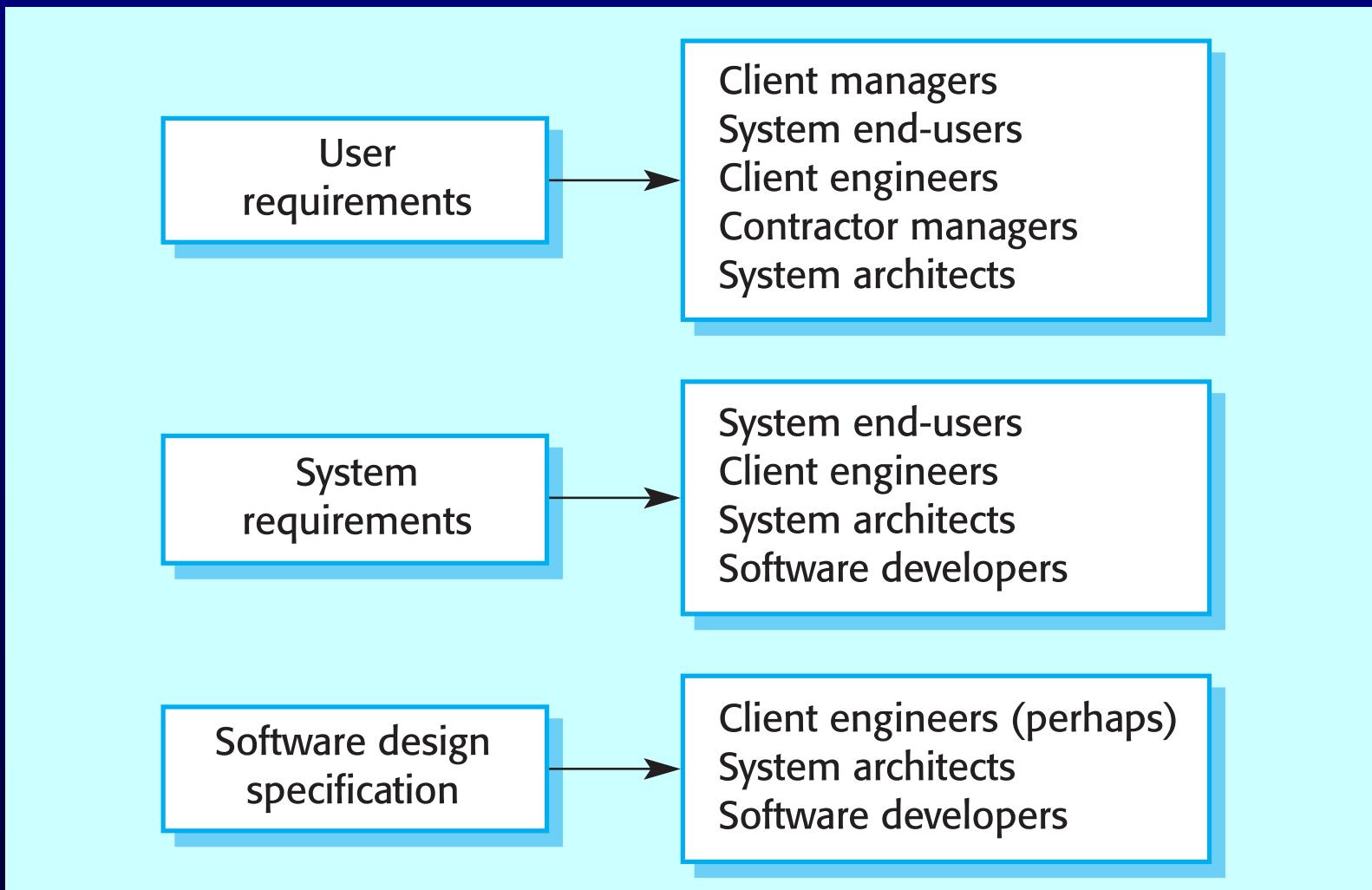
User requirement definition

1. The software must provide a means of representing and accessing external files created by other tools.

System requirements specification

- 1.1 The user should be provided with facilities to define the type of external files.
- 1.2 Each external file type may have an associated tool which may be applied to the file.
- 1.3 Each external file type may be represented as a specific icon on the user's display.
- 1.4 Facilities should be provided for the icon representing an external file type to be defined by the user.
- 1.5 When a user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.

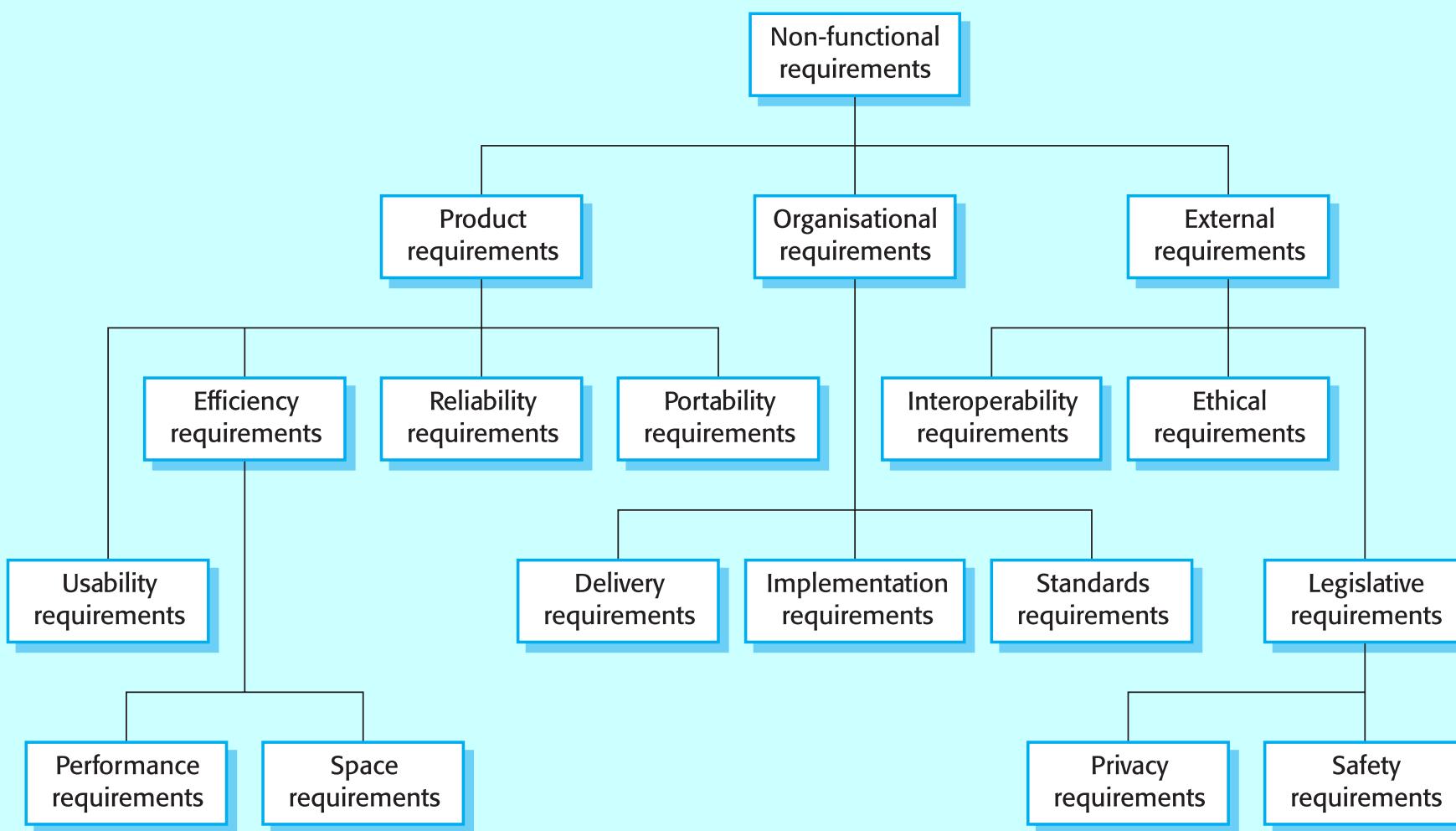
Requirements readers



Functional and non-functional requirements

- Functional requirements
 - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- Non-functional requirements
 - constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Domain requirements
 - Requirements that come from the application domain of the system and that reflect characteristics of that domain.

Non-functional requirement types



Requirements completeness and consistency

- In principle, requirements should be both complete and consistent.
- Complete
 - They should include descriptions of all facilities required.
- Consistent
 - There should be no conflicts or contradictions in the descriptions of the system facilities.
- In practice, it is impossible to produce a complete and consistent requirements document.

Requirements measures

Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	M Bytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Problems with natural language

- Lack of clarity
 - Precision is difficult without making the document difficult to read.
- Requirements confusion
 - Functional and non-functional requirements tend to be mixed-up.
- Requirements amalgamation
 - Several different requirements may be expressed together.

Guidelines for writing requirements

- Invent a standard format and use it for all requirements.
- Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- Use text highlighting to identify key parts of the requirement.
- Avoid the use of computer jargon.

Requirements and design

- In principle, requirements should state what the system should do and the design should describe how it does this.
- In practice, requirements and design are inseparable
 - A system architecture may be designed to structure the requirements;
 - The system may inter-operate with other systems that generate design requirements;
 - The use of a specific design may be a domain requirement.

Alternatives to NL specification

Notation	Description
Structured natural language	This approach depends on defining standard forms or templates to express the requirements specification.
Design description languages	This approach uses a language like a programming language but with more abstract features to specify the requirements by defining an operational model of the system. This approach is not now widely used although it can be useful for interface specifications.
Graphical notations	A graphical language, supplemented by text annotations is used to define the functional requirements for the system. An early example of such a graphical language was SADT. Now, use-case descriptions and sequence diagrams are commonly used .
Mathematical specifications	These are notations based on mathematical concepts such as finite-state machines or sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality. However, most customers don't understand formal specifications and are reluctant to accept it as a system contract.

Form-based node specification

<i>Insulin Pump/Control Software/SRS/3.3.2</i>	
Function	Compute insulin dose: Safe sugar level
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1)
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose → the dose in insulin to be delivered
Destination	Main control loop
Action:	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requires	Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin..
Post-condition	r0 is replaced by r1 then r1 is replaced by r2
Side-effects	None

Tabular specification

- Used to supplement natural language.
- Particularly useful when you have to define a number of possible alternative courses of action.

Tabular specification

Condition	Action
Sugar level falling ($r_2 < r_1$)	$\text{CompDose} = 0$
Sugar level stable ($r_2 = r_1$)	$\text{CompDose} = 0$
Sugar level increasing and rate of increase decreasing $((r_2-r_1) < (r_1-r_0))$	$\text{CompDose} = 0$
Sugar level increasing and rate of increase stable or increasing. $((r_2-r_1) \geq (r_1-r_0))$	$\text{CompDose} = \text{round}((r_2-r_1)/4)$ If rounded result = 0 then $\text{CompDose} = \text{MinimumDose}$

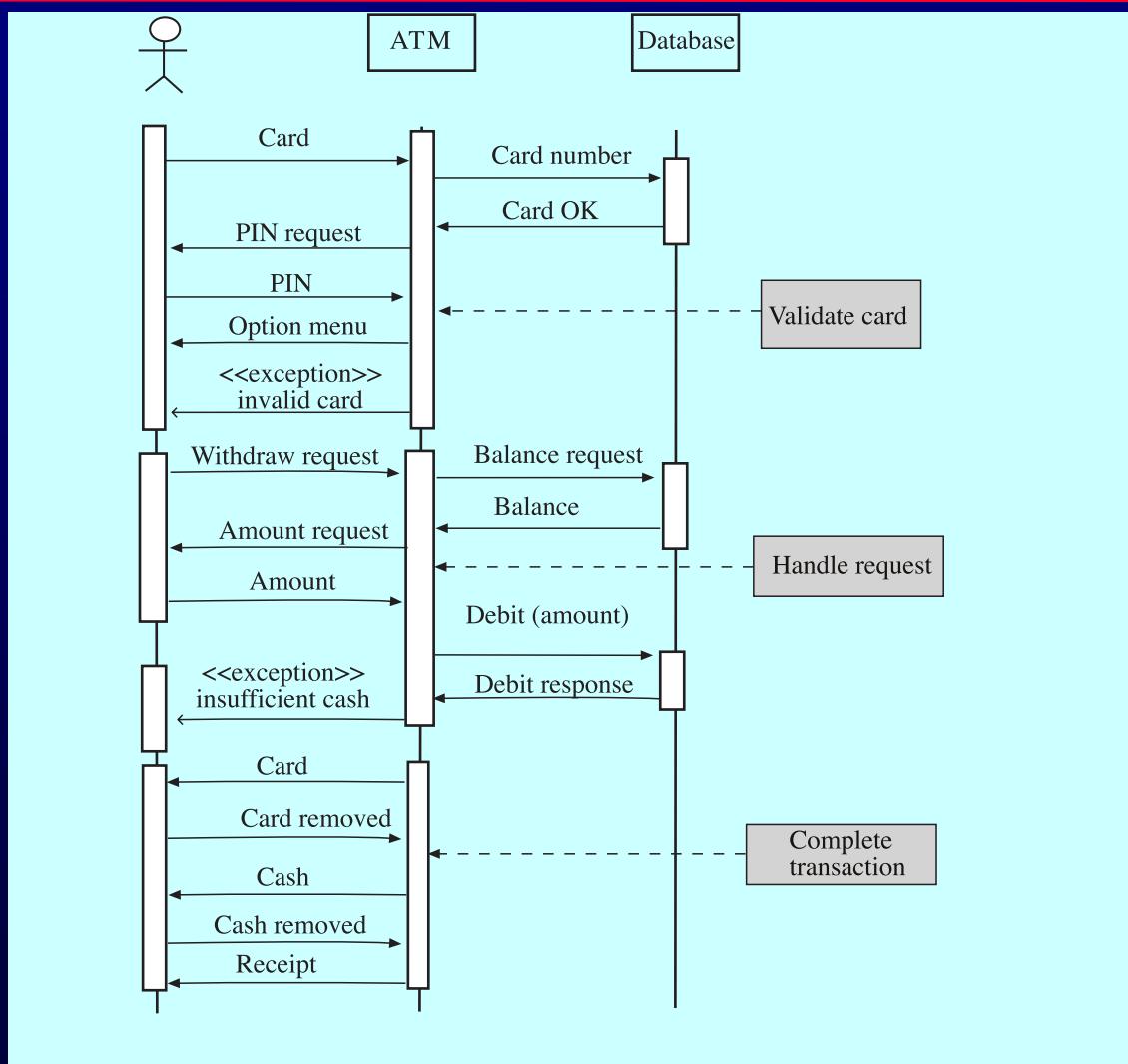
Graphical models

- Graphical models are most useful when you need to show how state changes or where you need to describe a sequence of actions.
- Different graphical models are explained in Chapter 8.

Sequence diagrams

- These show the sequence of events that take place during some user interaction with a system.
- You read them from top to bottom to see the order of the actions that take place.
- Cash withdrawal from an ATM
 - Validate card;
 - Handle request;
 - Complete transaction.

Sequence diagram of ATM withdrawal



Interface specification

- Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements.
- Three types of interface may have to be defined
 - Procedural interfaces;
 - Data structures that are exchanged;
 - Data representations.
- Formal notations are an effective technique for interface specification.

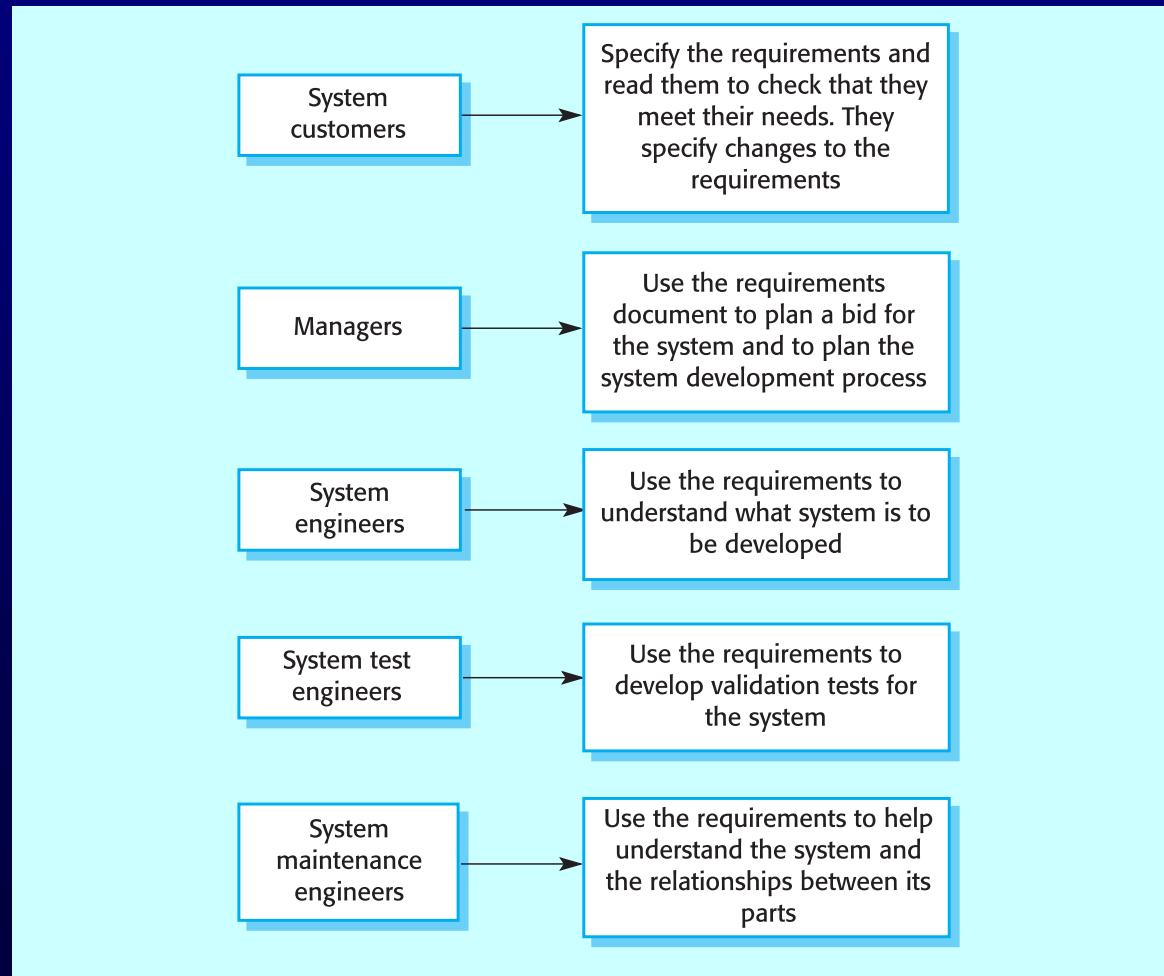
PDL interface description

```
interface PrintServer {  
  
    // defines an abstract printer server  
    // requires:      interface Printer, interface PrintDoc  
    // provides: initialize, print, displayPrintQueue, cancelPrintJob, switchPrinter  
  
    void initialize ( Printer p ) ;  
    void print ( Printer p, PrintDoc d ) ;  
    void displayPrintQueue ( Printer p ) ;  
    void cancelPrintJob (Printer p, PrintDoc d) ;  
    void switchPrinter (Printer p1, Printer p2, PrintDoc d) ;  
} //PrintServer
```

The requirements document

- The requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it

Users of a requirements document



IEEE requirements standard

- Defines a generic structure for a requirements document that must be instantiated for each specific system.
 - Introduction.
 - General description.
 - Specific requirements.
 - Appendices.
 - Index.

Requirements document structure

- Preface
- Introduction
- Glossary
- User requirements definition
- System architecture
- System requirements specification
- System models
- System evolution
- Appendices
- Index

Key points

- Requirements set out what the system should do and define constraints on its operation and implementation.
- Functional requirements set out services the system should provide.
- Non-functional requirements constrain the system being developed or the development process.
- User requirements are high-level statements of what the system should do. User requirements should be written using natural language, tables and diagrams.

Key points

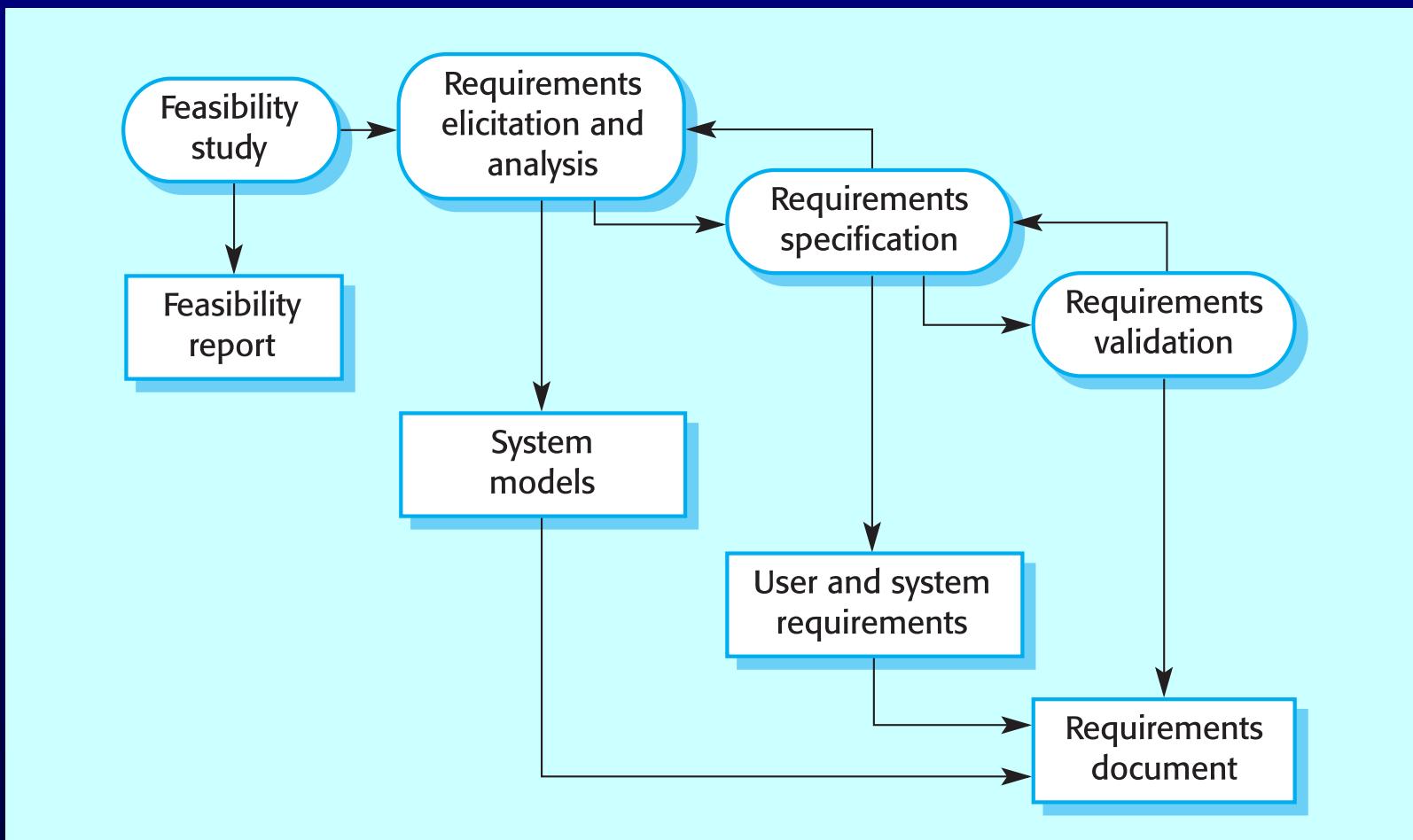
- System requirements are intended to communicate the functions that the system should provide.
- A software requirements document is an agreed statement of the system requirements.
- The IEEE standard is a useful starting point for defining more detailed specific requirements standards.

Requirements Engineering Processes

Topics covered

- Feasibility studies
- Requirements elicitation and analysis
- Requirements validation
- Requirements management

The requirements engineering process



Feasibility studies

- A feasibility study decides whether or not the proposed system is worthwhile.
- A short focused study that checks
 - If the system contributes to organisational objectives;
 - If the system can be engineered using current technology and within budget;
 - If the system can be integrated with other systems that are used.

Elicitation and analysis

- Sometimes called requirements elicitation or requirements discovery.
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*.

Problems of requirements analysis

- Stakeholders don't know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organisational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders may emerge and the business environment change.

Process activities

- Requirements discovery
 - Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
- Requirements classification and organisation
 - Groups related requirements and organises them into coherent clusters.
- Prioritisation and negotiation
 - Prioritising requirements and resolving requirements conflicts.
- Requirements documentation
 - Requirements are documented and input into the next round of the spiral.

Requirements discovery

- The process of gathering information about the proposed and existing systems and distilling the user and system requirements from this information.
- Sources of information include documentation, system stakeholders and the specifications of similar systems.

ATM stakeholders

- Bank customers
- Representatives of other banks
- Bank managers
- Counter staff
- Database administrators
- Security managers
- Marketing department
- Hardware and software maintenance engineers
- Banking regulators

Viewpoints

- Viewpoints are a way of structuring the requirements to represent the perspectives of different stakeholders. Stakeholders may be classified under different viewpoints.
- This multi-perspective analysis is important as there is no single correct way to analyse system requirements.

Interviewing

- In formal or informal interviewing, the RE team puts questions to stakeholders about the system that they use and the system to be developed.
- There are two types of interview
 - Closed interviews where a pre-defined set of questions are answered.
 - Open interviews where there is no pre-defined agenda and a range of issues are explored with stakeholders.

Scenarios

- Scenarios are real-life examples of how a system can be used.
- They should include
 - A description of the starting situation;
 - A description of the normal flow of events;
 - A description of what can go wrong;
 - Information about other concurrent activities;
 - A description of the state when the scenario finishes.

LIBSYS scenario (1)

Initial assumption: The user has logged on to the LIBSYS system and has located the journal containing the copy of the article.

Normal: The user selects the article to be copied. He or she is then prompted by the system to either provide subscriber information for the journal or to indicate how they will pay for the article. Alternative payment methods are by credit card or by quoting an organisational account number.

The user is then asked to fill in a copyright form that maintains details of the transaction and they then submit this to the LIBSYS system.

The copyright form is checked and, if OK, the PDF version of the article is downloaded to the LIBSYS working area on the user's computer and the user is informed that it is available. The user is asked to select a printer and a copy of the article is printed. If the article has been flagged as 'print-only' it is deleted from the user's system once the user has confirmed that printing is complete.

LIBSYS scenario (2)

What can go wrong: The user may fail to fill in the copyright form correctly. In this case, the form should be re-presented to the user for correction. If the resubmitted form is still incorrect then the user's request for the article is rejected.

The payment may be rejected by the system. The user's request for the article is rejected.

The article download may fail. Retry until successful or the user terminates the session.

It may not be possible to print the article. If the article is not flagged as "print-only" then it is held in the LIBSYS workspace. Otherwise, the article is deleted and the user's account credited with the cost of the article.

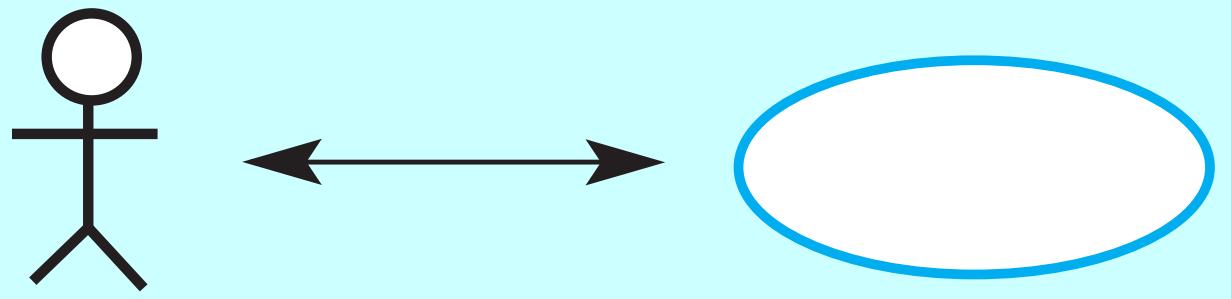
Other activities: Simultaneous downloads of other articles.

System state on completion: User is logged on. The downloaded article has been deleted from LIBSYS workspace if it has been flagged as print-only.

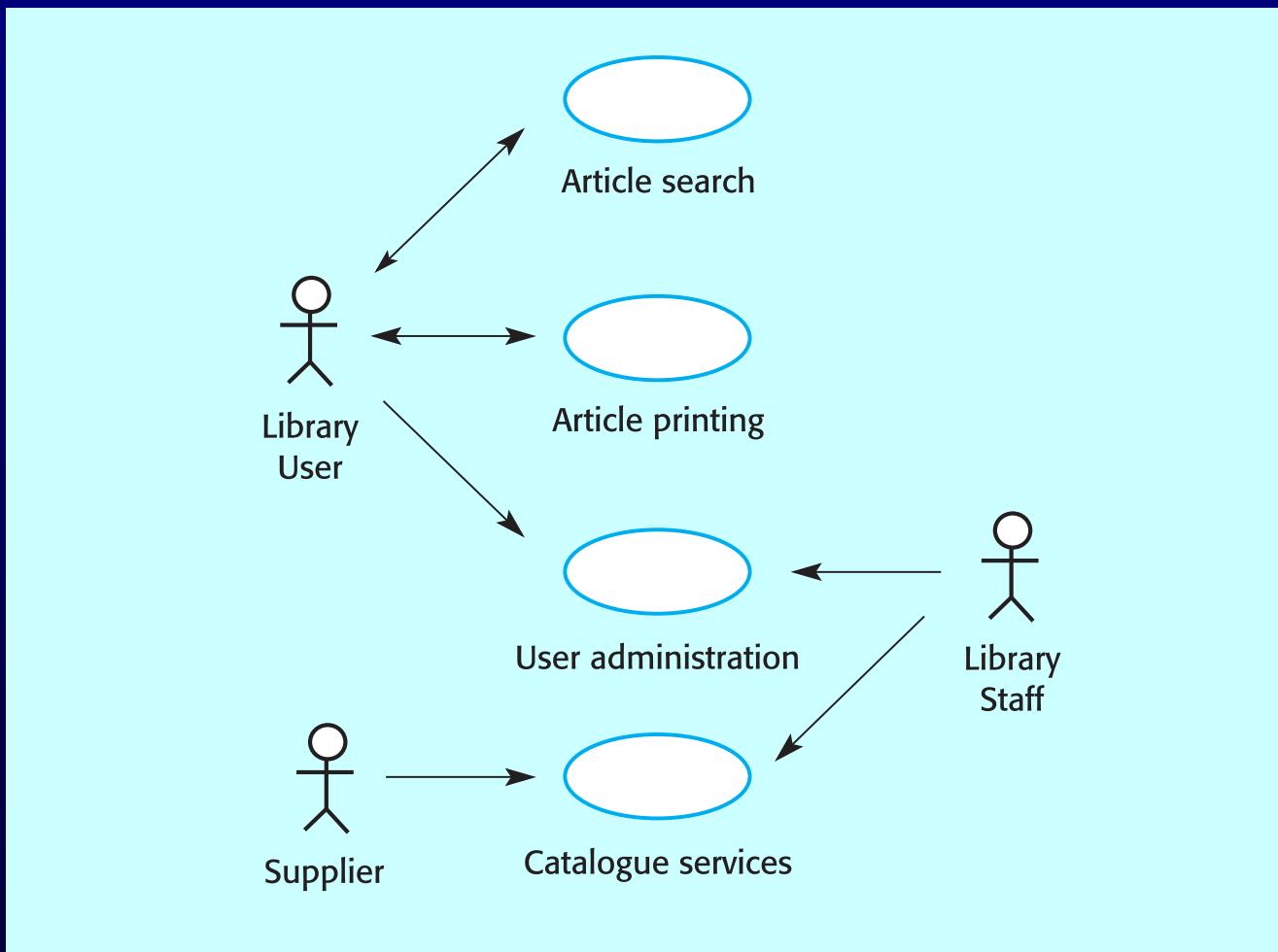
Use cases

- Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.
- A set of use cases should describe all possible interactions with the system.
- Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

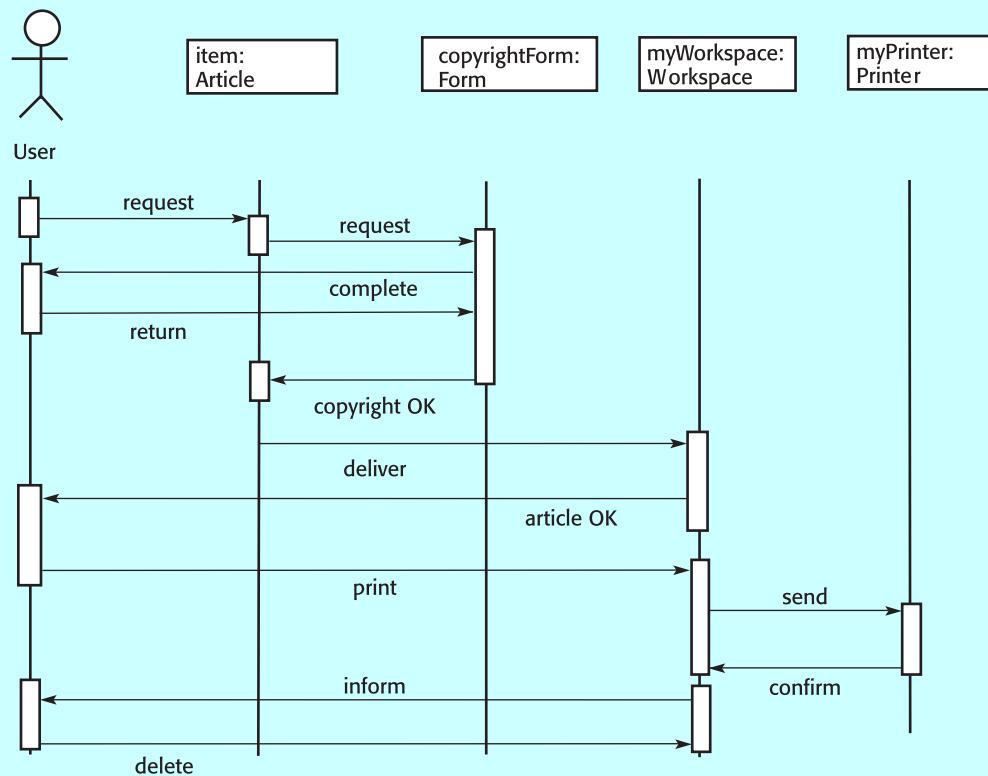
Article printing use-case



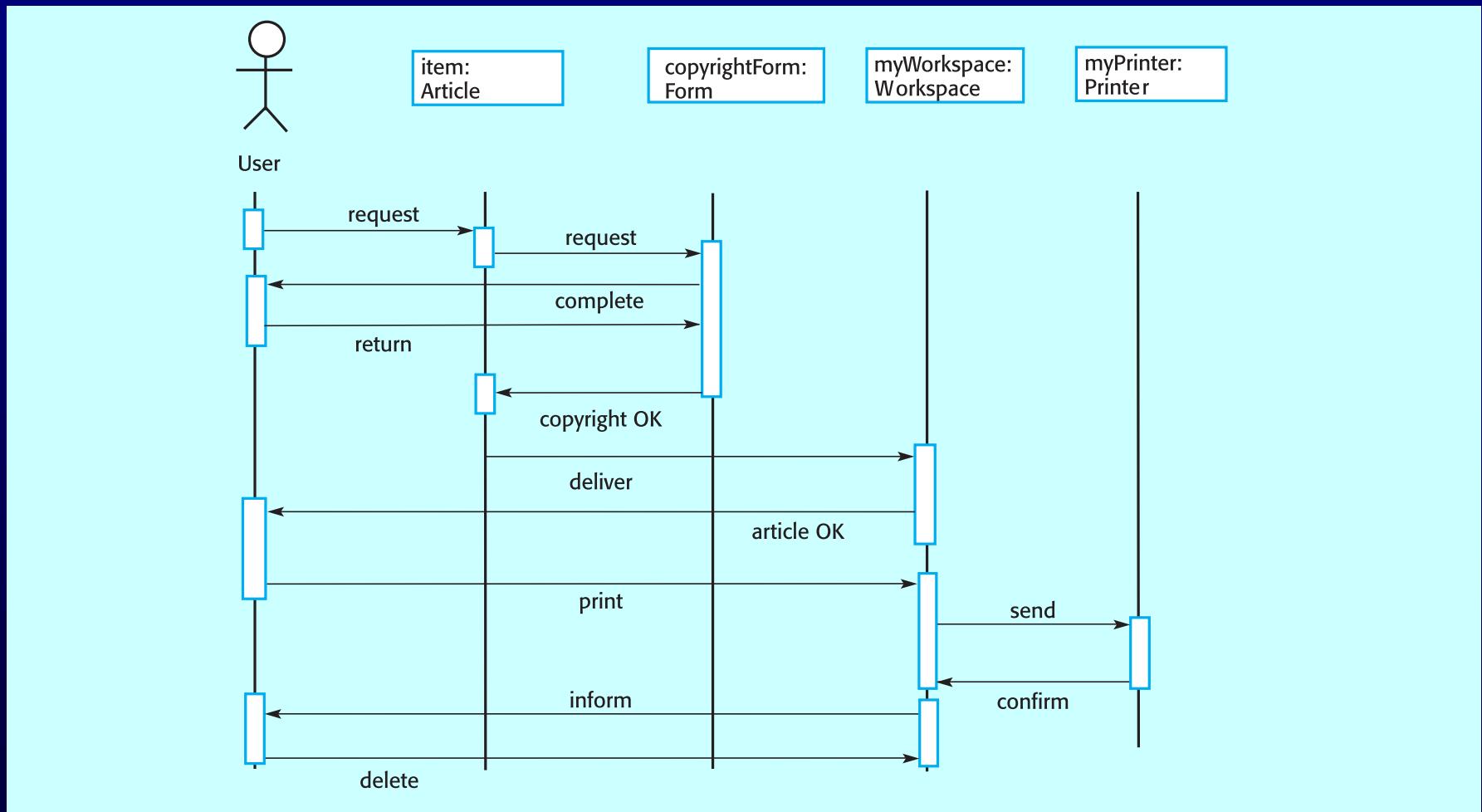
LIBSYS use cases



Article printing



Print article sequence



Ethnography

- A social scientist spends a considerable time observing and analysing how people actually work.
- People do not have to explain or articulate their work.
- Social and organisational factors of importance may be observed.
- Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

Requirements validation

- Concerned with demonstrating that the requirements define the system that the customer really wants.
- Requirements error costs are high so validation is very important
 - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking

- **Validity.** Does the system provide the functions which best support the customer's needs?
- **Consistency.** Are there any requirements conflicts?
- **Completeness.** Are all functions required by the customer included?
- **Realism.** Can the requirements be implemented given available budget and technology
- **Verifiability.** Can the requirements be checked?

Requirements validation techniques

- Requirements reviews
 - Systematic manual analysis of the requirements.
- Prototyping
 - Using an executable model of the system to check requirements. Covered in Chapter 17.
- Test-case generation
 - Developing tests for requirements to check testability.

Review checks

- **Verifiability.** Is the requirement realistically testable?
- **Comprehensibility.** Is the requirement properly understood?
- **Traceability.** Is the origin of the requirement clearly stated?
- **Adaptability.** Can the requirement be changed without a large impact on other requirements?

Requirements management

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development.
- Requirements are inevitably incomplete and inconsistent
 - New requirements emerge during the process as business needs change and a better understanding of the system is developed;
 - Different viewpoints have different requirements and these are often contradictory.

Requirements classification

Requirement Type	Description
Mutable requirements	Requirements that change because of changes to the environment in which the organisation is operating. For example, in hospital systems, the funding of patient care may change and thus require different treatment information to be collected.
Emergent requirements	Requirements that emerge as the customer's understanding of the system develops during the system development. The design process may reveal new emergent requirements.
Consequential requirements	Requirements that result from the introduction of the computer system. Introducing the computer system may change the organisations processes and open up new ways of working which generate new system requirements
Compatibility requirements	Requirements that depend on the particular systems or business processes within an organisation. As these change, the compatibility requirements on the commissioned or delivered system may also have to evolve.

Requirements management planning

- During the requirements engineering process, you have to plan:
 - Requirements identification
 - How requirements are individually identified;
 - A change management process
 - The process followed when analysing a requirements change;
 - Traceability policies
 - The amount of information about requirements relationships that is maintained;
 - CASE tool support
 - The tool support required to help manage requirements change;

Traceability

- Traceability is concerned with the relationships between requirements, their sources and the system design
- Source traceability
 - Links from requirements to stakeholders who proposed these requirements;
- Requirements traceability
 - Links between dependent requirements;
- Design traceability
 - Links from the requirements to the design;

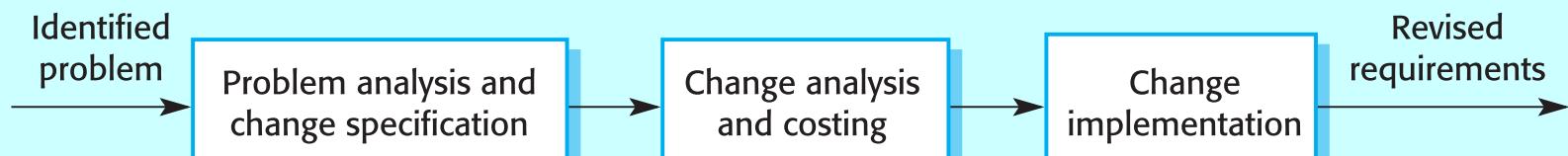
A traceability matrix

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		D	R					
1.2			D			D		D
1.3	R			R				
2.1			R		D			D
2.2								D
2.3		R		D				
3.1							R	
3.2						R		

CASE tool support

- Requirements storage
 - Requirements should be managed in a secure, managed data store.
- Change management
 - The process of change management is a workflow process whose stages can be defined and information flow between these stages partially automated.
- Traceability management
 - Automated retrieval of the links between requirements.

Change management



Key points

- The requirements engineering process includes a feasibility study, requirements elicitation and analysis, requirements specification and requirements management.
- Requirements elicitation and analysis is iterative involving domain understanding, requirements collection, classification, structuring, prioritisation and validation.
- Systems have multiple stakeholders with different requirements.

Key points

- Social and organisation factors influence system requirements.
- Requirements validation is concerned with checks for validity, consistency, completeness, realism and verifiability.
- Business changes inevitably lead to changing requirements.
- Requirements management includes planning and change management.

Critical Systems Specification

Topics covered

- Risk-driven specification
- Safety specification
- Security specification
- Software reliability specification

Dependability requirements

- **Functional requirements** to define error checking and recovery facilities and protection against system failures.
- **Non-functional requirements** defining the required reliability and availability of the system.
- **Excluding requirements** that define states and conditions that must not arise.

Stages of risk-based analysis

- Risk identification
 - Identify potential risks that may arise.
- Risk analysis and classification
 - Assess the seriousness of each risk.
- Risk decomposition
 - Decompose risks to discover their potential root causes.
- Risk reduction assessment
 - Define how each risk must be taken into account or reduced when the system is designed.

Risk identification

- Identify the risks faced by the critical system.
- In safety-critical systems, the risks are the hazards that can lead to accidents.
- In security-critical systems, the risks are the potential attacks on the system.
- In risk identification, you should identify risk classes and position risks in these classes
 - Service failure;
 - Electrical risks;
 - ...

Insulin pump risks

- Insulin overdose (service failure).
- Insulin underdose (service failure).
- Power failure due to exhausted battery (electrical).
- Electrical interference with other medical equipment (electrical).
- Poor sensor and actuator contact (physical).
- Parts of machine break off in body (physical).
- Infection caused by introduction of machine (biological).
- Allergic reaction to materials or insulin (biological).

Risk analysis and classification

- The process is concerned with understanding the likelihood that a risk will arise and the potential consequences if an accident or incident should occur.
- Risks may be categorised as:
 - **Intolerable.** Must never arise or result in an accident
 - **As low as reasonably practical(ALARP).** Must minimise the possibility of risk given cost and schedule constraints
 - **Acceptable.** The consequences of the risk are acceptable and no extra costs should be incurred to reduce hazard probability

Risk assessment

- Estimate the risk probability and the risk severity.
- It is not normally possible to do this precisely so relative values are used such as ‘unlikely’, ‘rare’, ‘very high’, etc.
- The aim must be to exclude risks that are likely to arise or that have high severity.

Risk assessment - insulin pump

Identified hazard	Hazard probability	Hazard severity	Estimated risk	Acceptability
1. Insulin overdose	Medium	High	High	Intolerable
2. Insulin underdose	Medium	Low	Low	Acceptable
3. Power failure	High	Low	Low	Acceptable
4. Machine incorrectly fitted	High	High	High	Intolerable
5. Machine breaks in patient	Low	High	Medium	ALARP
6. Machine causes infection	Medium	Medium	Medium	ALARP
7. Electrical interference	Low	High	Medium	ALARP
8. Allergic reaction	Low	Low	Low	Acceptable

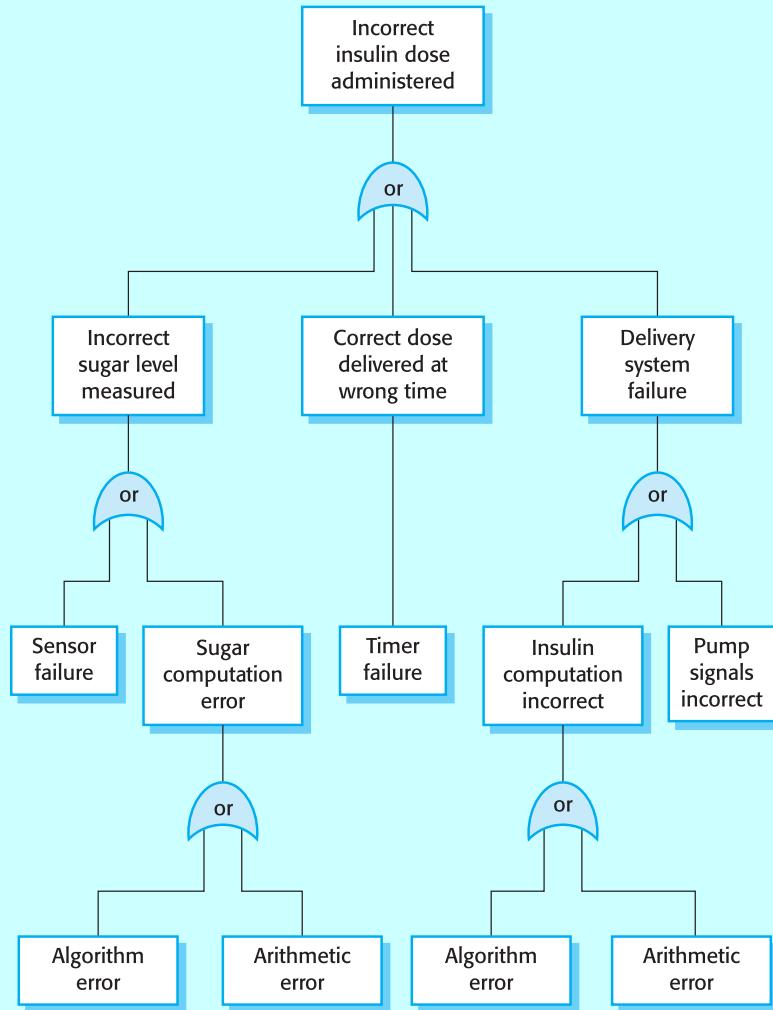
Risk decomposition

- Concerned with discovering the root causes of risks in a particular system.
- Techniques have been mostly derived from safety-critical systems and can be
 - Inductive, bottom-up techniques. Start with a proposed system failure and assess the hazards that could arise from that failure;
 - Deductive, top-down techniques. Start with a hazard and deduce what the causes of this could be.

Fault-tree analysis

- A deductive top-down technique.
- Put the risk or hazard at the root of the tree and identify the system states that could lead to that hazard.
- Where appropriate, link these with ‘and’ or ‘or’ conditions.
- A goal should be to minimise the number of single causes of system failure.

Insulin pump fault tree



Risk reduction assessment

- The aim of this process is to identify dependability requirements that specify how the risks should be managed and ensure that accidents/incidents do not arise.
- Risk reduction strategies
 - Risk avoidance;
 - Risk detection and removal;
 - Damage limitation.

Insulin pump - software risks

- Arithmetic error
 - A computation causes the value of a variable to overflow or underflow;
 - Maybe include an exception handler for each type of arithmetic error.
- Algorithmic error
 - Compare dose to be delivered with previous dose or safe maximum doses. Reduce dose if too high.

Safety requirements - insulin pump

- SR1:** The system shall not deliver a single dose of insulin that is greater than a specified maximum dose for a system user.
- SR2:** The system shall not deliver a daily cumulative dose of insulin that is greater than a specified maximum for a system user.
- SR3:** The system shall include a hardware diagnostic facility that shall be executed at least 4 times per hour.
- SR4:** The system shall include an exception handler for all of the exceptions that are identified in Table 3.
- SR5:** The audible alarm shall be sounded when any hardware or software anomaly is discovered and a diagnostic message as defined in Table 4 should be displayed.
- SR6:** In the event of an alarm in the system, insulin delivery shall be suspended until the user has reset the system and cleared the alarm.

Safety specification

- The safety requirements of a system should be separately specified.
- These requirements should be based on an analysis of the possible hazards and risks as previously discussed.
- Safety requirements usually apply to the system as a whole rather than to individual sub-systems. In systems engineering terms, the safety of a system is an emergent property.

IEC 61508

- An international standard for safety management that was specifically designed for protection systems - it is not applicable to all safety-critical systems.
- Incorporates a model of the safety life cycle and covers all aspects of safety management from scope definition to system decommissioning.

Safety requirements

- Functional safety requirements
 - These define the safety functions of the protection system i.e. they define how the system should provide protection.
- Safety integrity requirements
 - These define the reliability and availability of the protection system. They are based on expected usage and are classified using a safety integrity level from 1 to 4.

Security specification

- Has some similarities to safety specification
 - Not possible to specify security requirements quantitatively;
 - The requirements are often ‘shall not’ rather than ‘shall’ requirements.
- Differences
 - No well-defined notion of a security life cycle for security management; No standards;
 - Generic threats rather than system specific hazards;
 - Mature security technology (encryption, etc.). However, there are problems in transferring this into general use;
 - The dominance of a single supplier (Microsoft) means that huge numbers of systems may be affected by security failure.

Stages in security specification

- Asset identification and evaluation
 - The assets (data and programs) and their required degree of protection are identified. The degree of required protection depends on the asset value so that a password file (say) is more valuable than a set of public web pages.
- Threat analysis and risk assessment
 - Possible security threats are identified and the risks associated with each of these threats is estimated.
- Threat assignment
 - Identified threats are related to the assets so that, for each identified asset, there is a list of associated threats.

Stages in security specification

- **Technology analysis**
 - Available security technologies and their applicability against the identified threats are assessed.
- **Security requirements specification**
 - The security requirements are specified. Where appropriate, these will explicitly identify the security technologies that may be used to protect against different threats to the system.

Types of security requirement

- Identification requirements.
- Authentication requirements.
- Authorisation requirements.
- Immunity requirements.
- Integrity requirements.
- Intrusion detection requirements.
- Non-repudiation requirements.
- Privacy requirements.
- Security auditing requirements.
- System maintenance security requirements.

LIBSYS security requirements

- SEC1:** All system users shall be identified using their library card number and personal password.
- SEC2:** Users privileges shall be assigned according to the class of user (student, staff, library staff).
- SEC3:** Before execution of any command, LIBSYS shall check that the user has sufficient privileges to access and execute that command.
- SEC4:** When a user orders a document, the order request shall be logged. The log data maintained shall include the time of order, the user's identification and the articles ordered.
- SEC5:** All system data shall be backed up once per day and backups stored off-site in a secure storage area.
- SEC6:** Users shall not be permitted to have more than 1 simultaneous login to LIBSYS.

System reliability specification

- **Hardware reliability**
 - What is the probability of a hardware component failing and how long does it take to repair that component?
- **Software reliability**
 - How likely is it that a software component will produce an incorrect output. Software failures are different from hardware failures in that software does not wear out. It can continue in operation even after an incorrect result has been produced.
- **Operator reliability**
 - How likely is it that the operator of a system will make an error?

Reliability metrics

- Reliability metrics are units of measurement of system reliability.
- System reliability is measured by counting the number of operational failures and, where appropriate, relating these to the demands made on the system and the time that the system has been operational.
- A long-term measurement programme is required to assess the reliability of critical systems.

Reliability metrics

Metric	Explanation
POFOD Probability of failure on demand	The likelihood that the system will fail when a service request is made. A POFOD of 0.001 means that 1 out of a thousand service requests may result in failure.
ROCOF Rate of failure occurrence	The frequency of occurrence with which unexpected behaviour is likely to occur. A ROCOF of 2/100 means that 2 failures are likely to occur in each 100 operational time units. This metric is sometimes called the failure intensity.
MTTF Mean time to failure	The average time between observed system failures. An MTTF of 500 means that 1 failure can be expected every 500 time units.
AVAIL Availability	The probability that the system is available for use at a given time. Availability of 0.998 means that in every 1000 time units, the system is likely to be available for 998 of these.

Failure consequences

- When specifying reliability, it is not just the number of system failures that matter but the consequences of these failures.
- Failures that have serious consequences are clearly more damaging than those where repair and recovery is straightforward.
- In some cases, therefore, different reliability specifications for different types of failure may be defined.

Failure classification

Failure class	Description
Transient	Occurs only with certain inputs
Permanent	Occurs with all inputs
Recoverable	System can recover without operator intervention
Unrecoverable	Operator intervention needed to recover from failure
Non-corrupting	Failure does not corrupt system state or data
Corrupting	Failure corrupts system state or data

Steps to a reliability specification

- For each sub-system, analyse the consequences of possible system failures.
- From the system failure analysis, partition failures into appropriate classes.
- For each failure class identified, set out the reliability using an appropriate metric. Different metrics may be used for different reliability requirements.
- Identify functional reliability requirements to reduce the chances of critical failures.

Bank auto-teller system

- Each machine in a network is used 300 times a day
- Bank has 1000 machines
- Lifetime of software release is 2 years
- Each machine handles about 200, 000 transactions
- About 300, 000 database transactions in total per day

Reliability specification for an ATM

Failure class	Example	Reliability metric
Permanent, non-corrupting.	The system fails to operate with any card that is input. Software must be restarted to correct failure.	ROCOF 1 occurrence/1000 days
Transient, non-corrupting	The magnetic stripe data cannot be read on an undamaged card that is input.	ROCOF 1 in 1000 transactions
Transient, corrupting	A pattern of transactions across the network causes database corruption.	Unquantifiable! Should never happen in the lifetime of the system

Specification validation

- It is impossible to empirically validate very high reliability specifications.
- No database corruptions means POFOD of less than 1 in 200 million.
- If a transaction takes 1 second, then simulating one day's transactions takes 3.5 days.
- It would take longer than the system's lifetime to test it for reliability.

System models

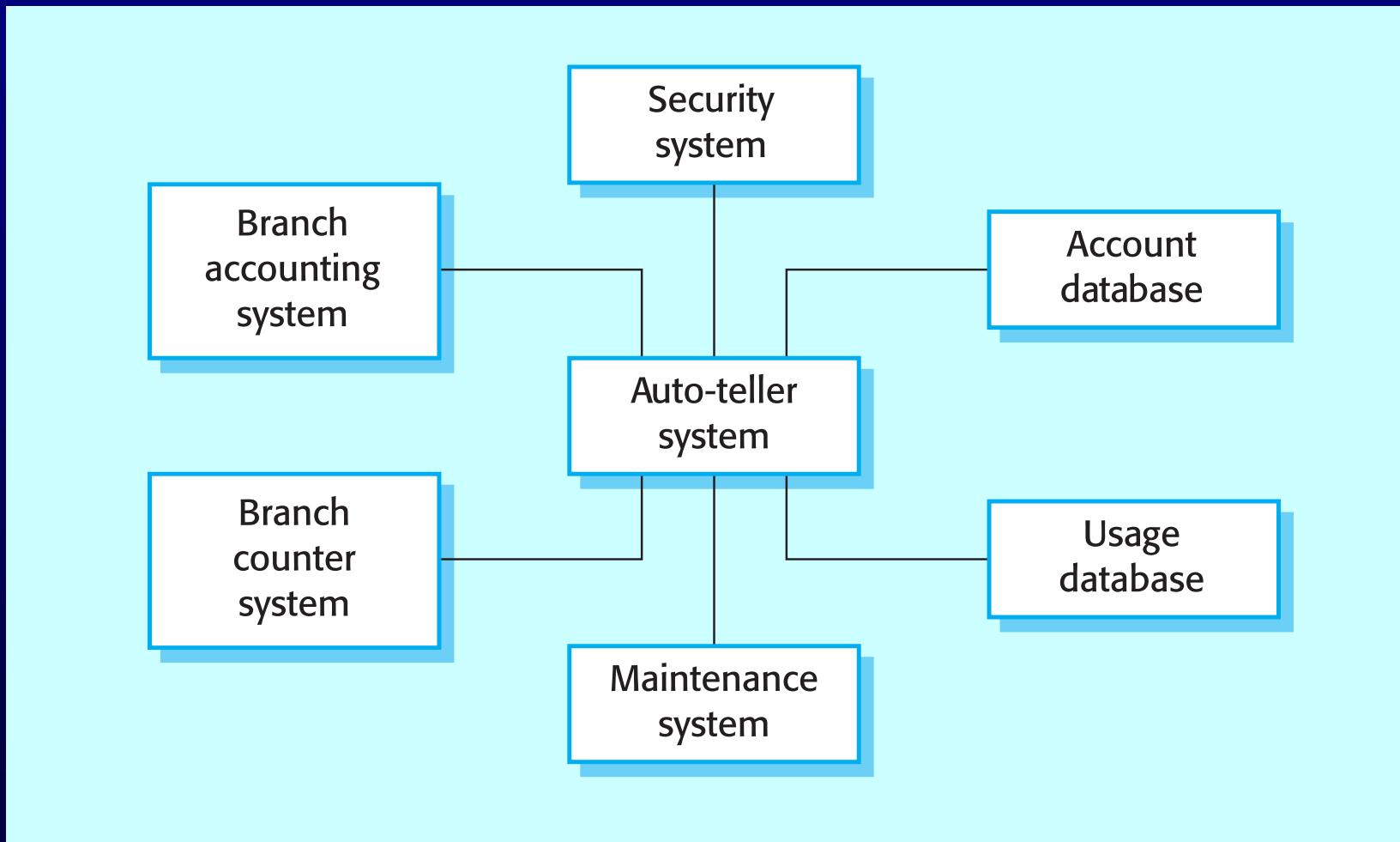
System modelling

- System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.
- Different models present the system from different perspectives
 - External perspective showing the system's context or environment;
 - Behavioural perspective showing the behaviour of the system;
 - Structural perspective showing the system or data architecture.

Context models

- Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- Social and organisational concerns may affect the decision on where to position system boundaries.

The context of an ATM system



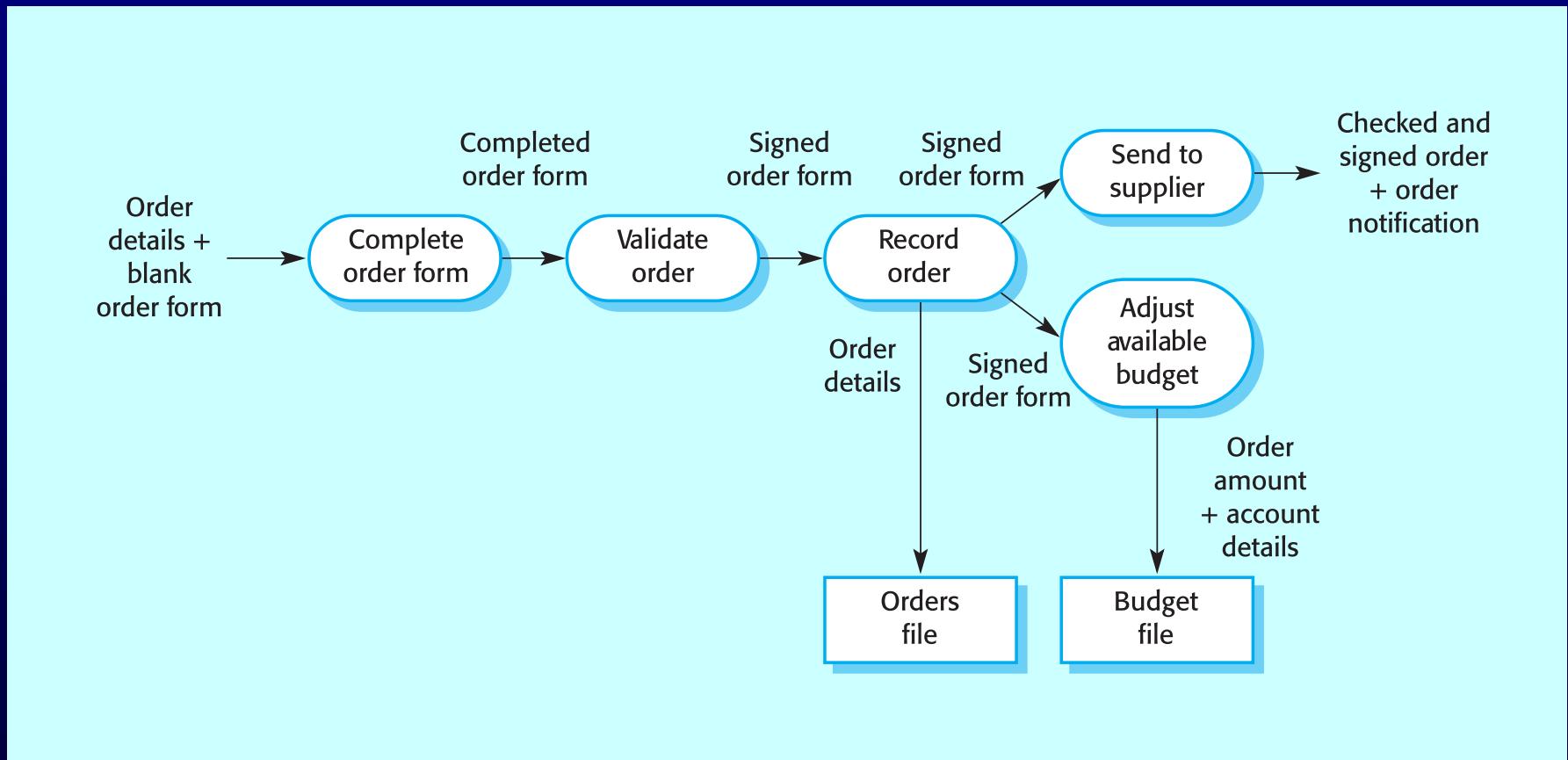
Behavioural models

- Behavioural models are used to describe the overall behaviour of a system.
- Two types of behavioural model are:
 - Data processing models that show how data is processed as it moves through the system;
 - State machine models that show the systems response to events.
- These models show different perspectives so both of them are required to describe the system's behaviour.

Data-processing models

- Data flow diagrams (DFDs) may be used to model the system's data processing.
- These show the processing steps as data flows through a system.
- DFDs are an intrinsic part of many analysis methods.
- Simple and intuitive notation that customers can understand.
- Show end-to-end processing of data.

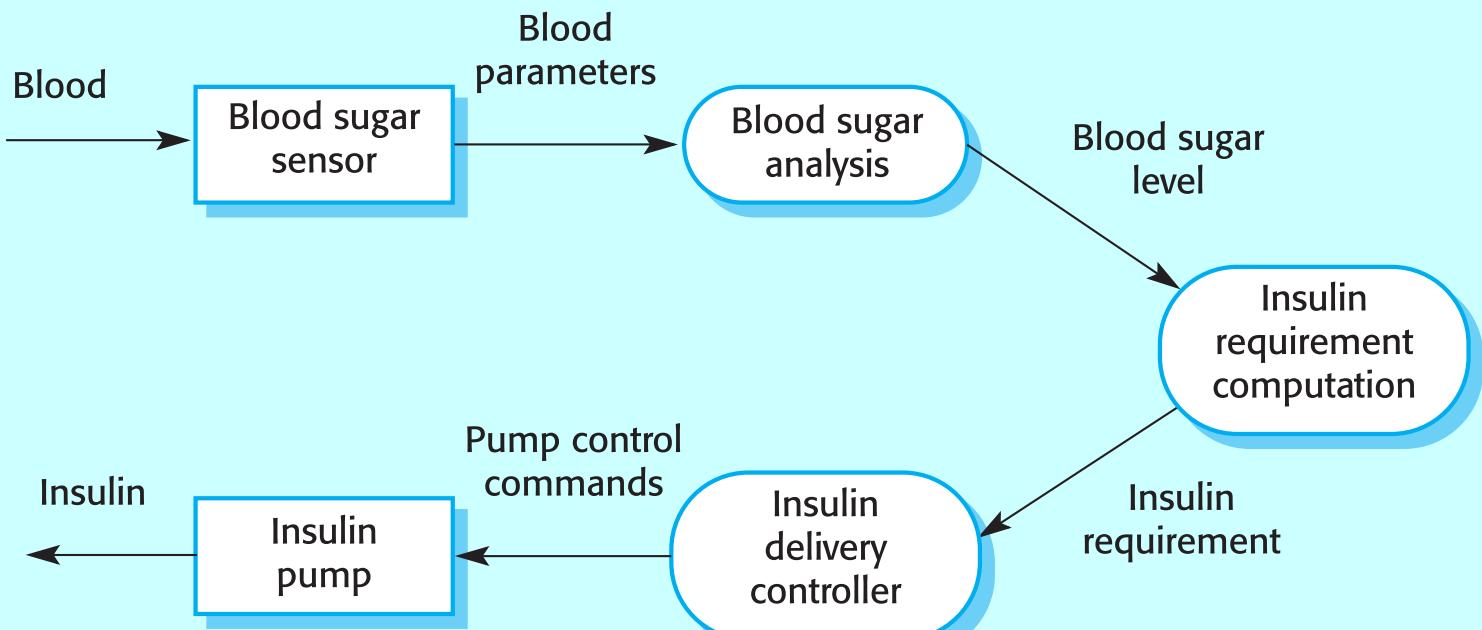
Order processing DFD



Data flow diagrams

- DFDs model the system from a functional perspective.
- Tracking and documenting how the data associated with a process is helpful to develop an overall understanding of the system.
- Data flow diagrams may also be used in showing the data exchange between a system and other systems in its environment.

Insulin pump DFD



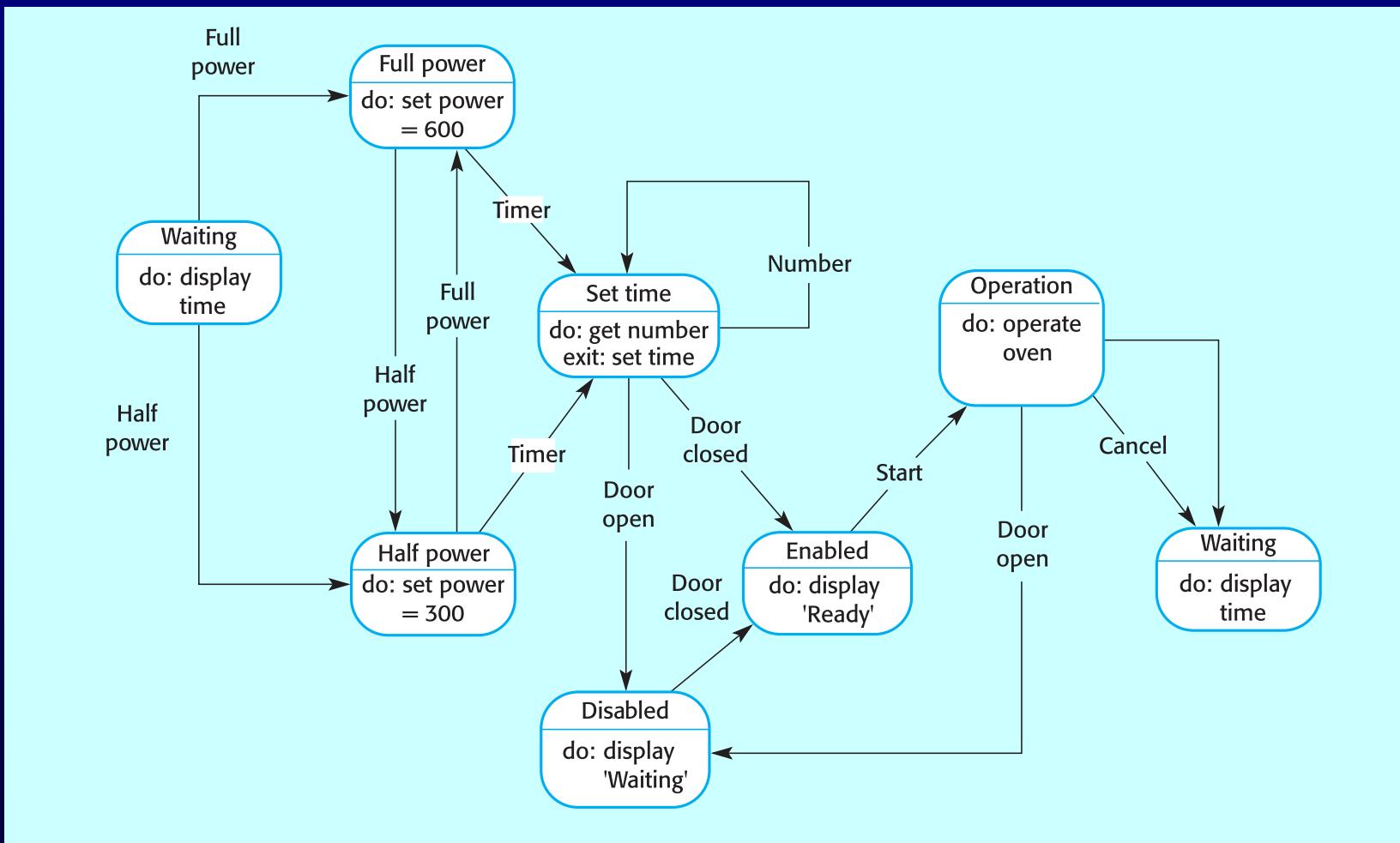
State machine models

- These model the behaviour of the system in response to external and internal events.
- They show the system's responses to stimuli so are often used for modelling real-time systems.
- State machine models show system states as nodes and events as arcs between these nodes. When an event occurs, the system moves from one state to another.
- Statecharts are an integral part of the UML and are used to represent state machine models.

Statecharts

- Allow the decomposition of a model into sub-models (see following slide).
- A brief description of the actions is included following the ‘do’ in each state.
- Can be complemented by tables describing the states and the stimuli.

Microwave oven model



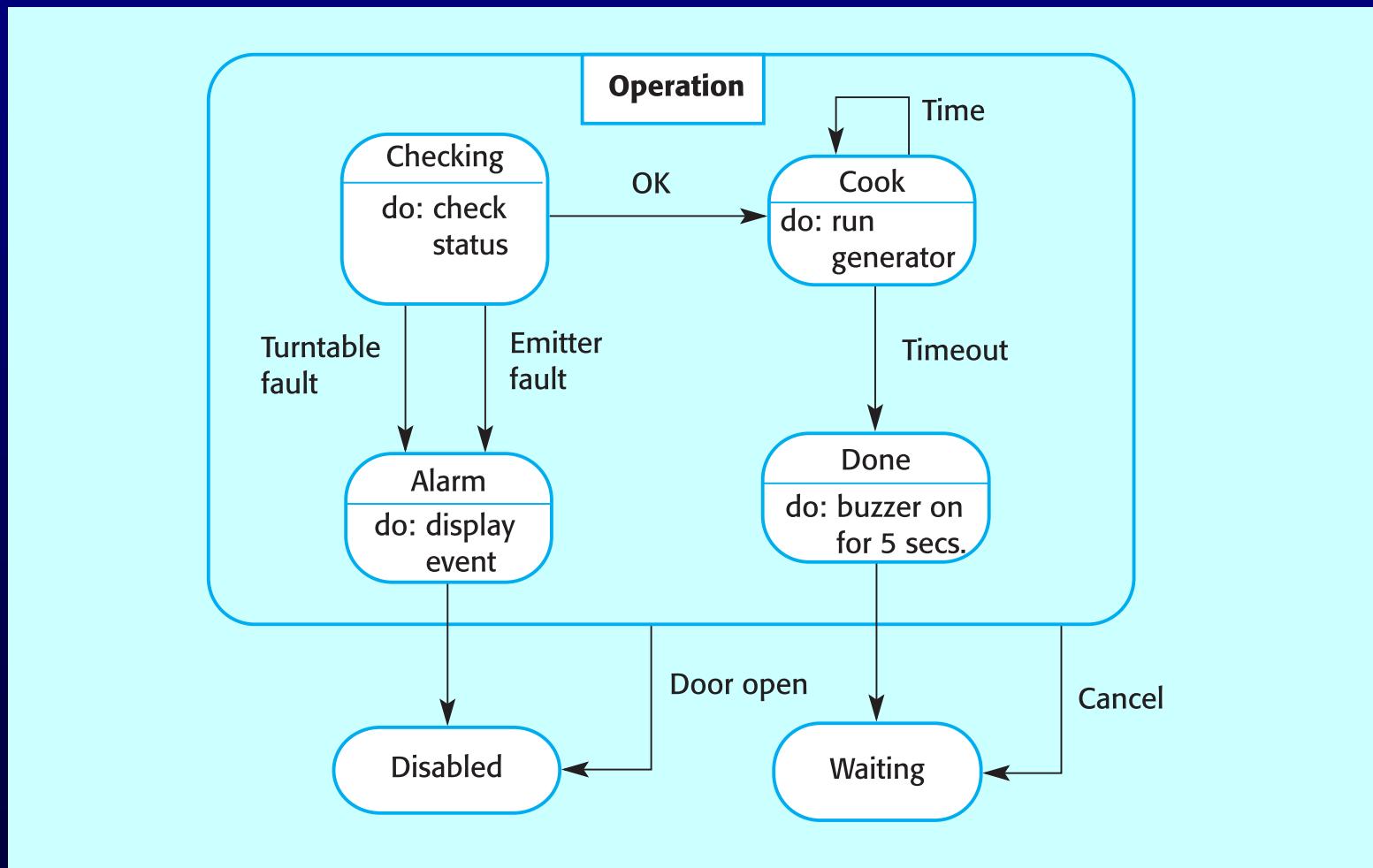
Microwave oven state description

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows "Half power"
Full power	The oven power is set to 600 watts. The display shows "Full power"
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows "Not ready"
Enabled	Oven operation is enabled. Interior oven light is off. Display shows "Ready to cook"
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows "Cooking complete" while buzzer is sounding.

Microwave oven stimuli

Stimulus	Description
Half power	The user has pressed the half power button
Full power	The user has pressed the full power button
Timer	The user has pressed one of the timer buttons
Number	The user has pressed a numeric key
Door open	The oven door switch is not closed
Door closed	The oven door switch is closed
Start	The user has pressed the start button
Cancel	The user has pressed the cancel button

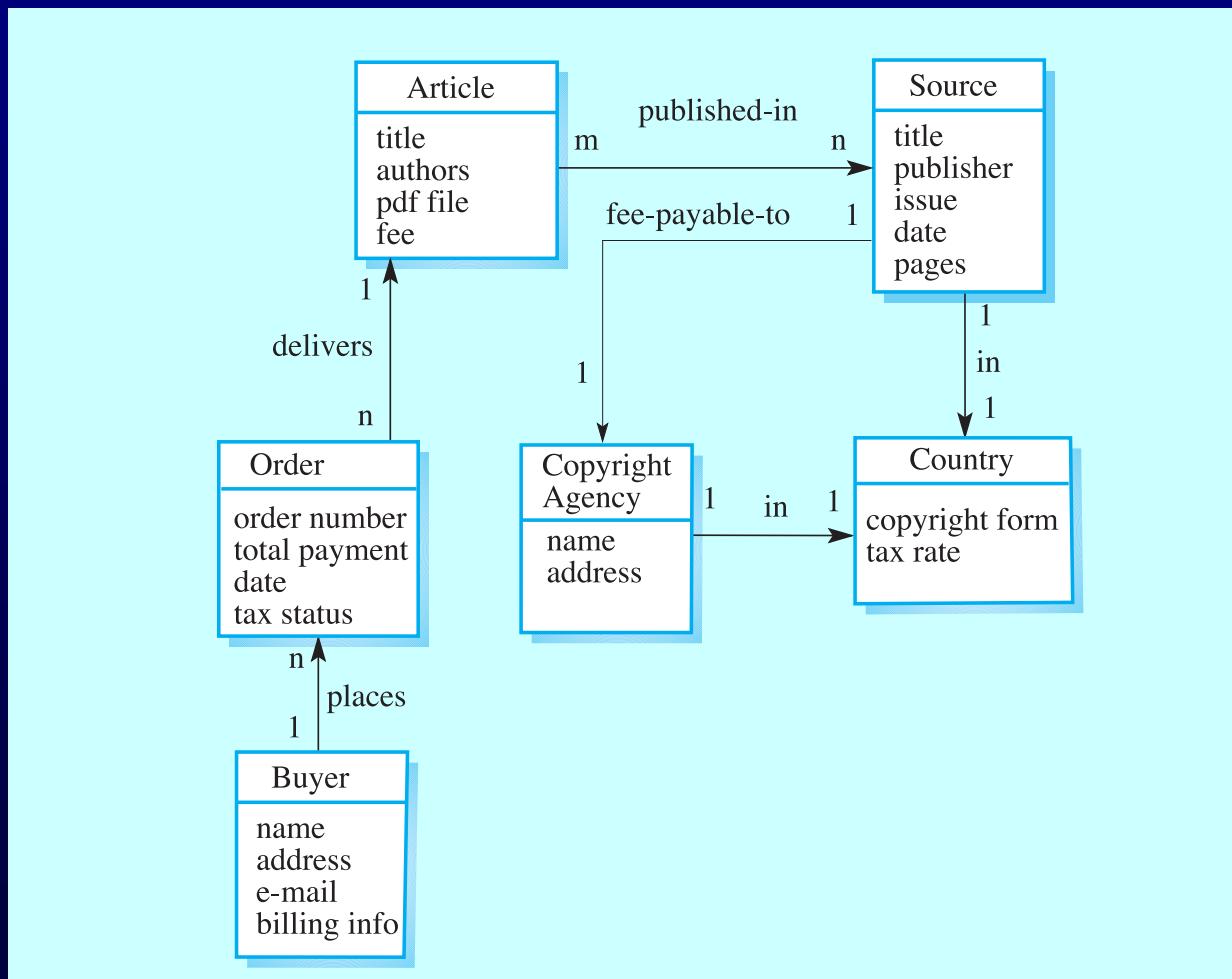
Microwave oven operation



Semantic data models

- Used to describe the logical structure of data processed by the system.
- An entity-relation-attribute model sets out the entities in the system, the relationships between these entities and the entity attributes
- Widely used in database design. Can readily be implemented using relational databases.
- No specific notation provided in the UML but objects and associations can be used.

Library semantic model



Data dictionaries

- Data dictionaries are lists of all of the names used in the system models. Descriptions of the entities, relationships and attributes are also included.
- Advantages
 - Support name management and avoid duplication;
 - Store of organisational knowledge linking analysis, design and implementation;
- Many CASE workbenches support data dictionaries.

Data dictionary entries

Name	Description	Type	Date
Article	Details of the published article that may be ordered by people using LIBSYS.	Entity	30.12.2002
authors	The names of the authors of the article who may be due a share of the fee.	Attribute	30.12.2002
Buyer	The person or organisation that orders a copy of the article.	Entity	30.12.2002
fee-payable-to	A 1:1 relationship between Article and the Copyright Agency who should be paid the copyright fee.	Relation	29.12.2002
Address (Buyer)	The address of the buyer. This is used to any paper billing information that is required.	Attribute	31.12.2002

Object models

- Object models describe the system in terms of object classes and their associations.
- An object class is an abstraction over a set of objects with common attributes and the services (operations) provided by each object.
- Various object models may be produced
 - Inheritance models;
 - Aggregation models;
 - Interaction models.

Object models

- Natural ways of reflecting the real-world entities manipulated by the system
- More abstract entities are more difficult to model using this approach
- Object class identification is recognised as a difficult process requiring a deep understanding of the application domain
- Object classes reflecting domain entities are reusable across systems

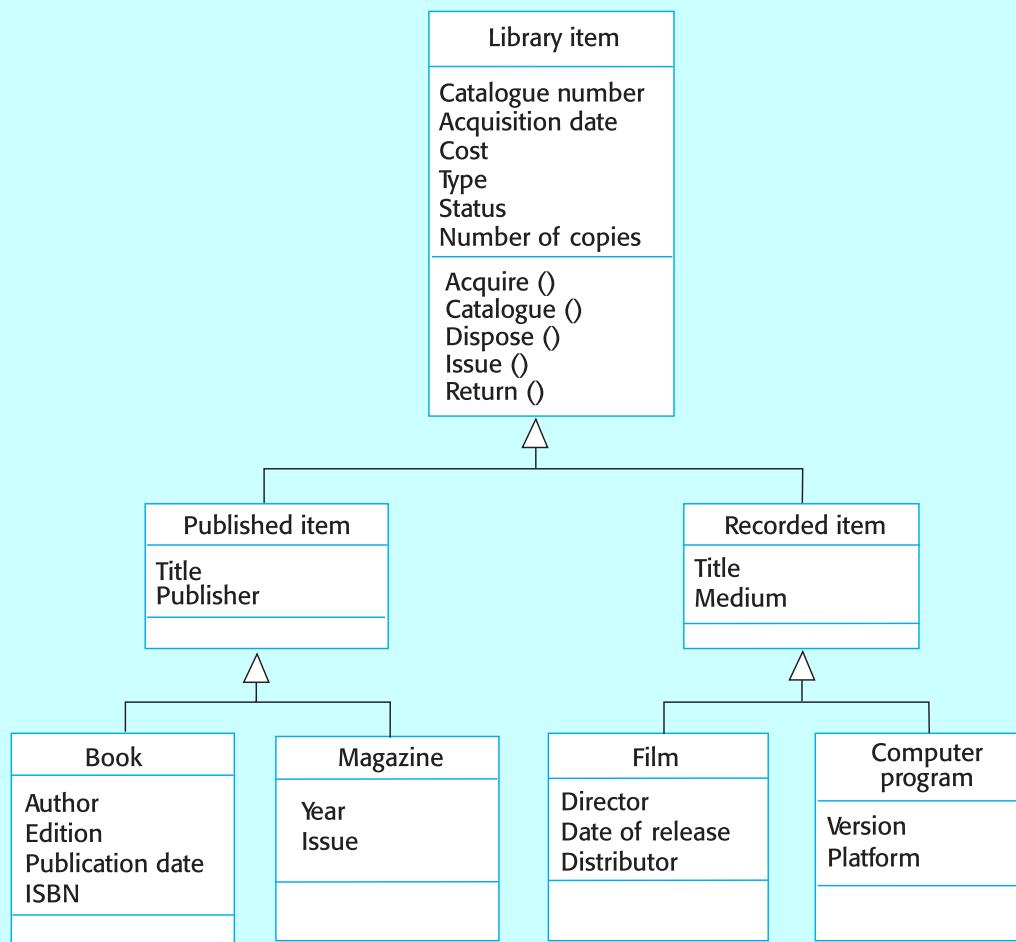
Inheritance models

- Organise the domain object classes into a hierarchy.
- Classes at the top of the hierarchy reflect the common features of all classes.
- Object classes inherit their attributes and services from one or more super-classes. these may then be specialised as necessary.
- Class hierarchy design can be a difficult process if duplication in different branches is to be avoided.

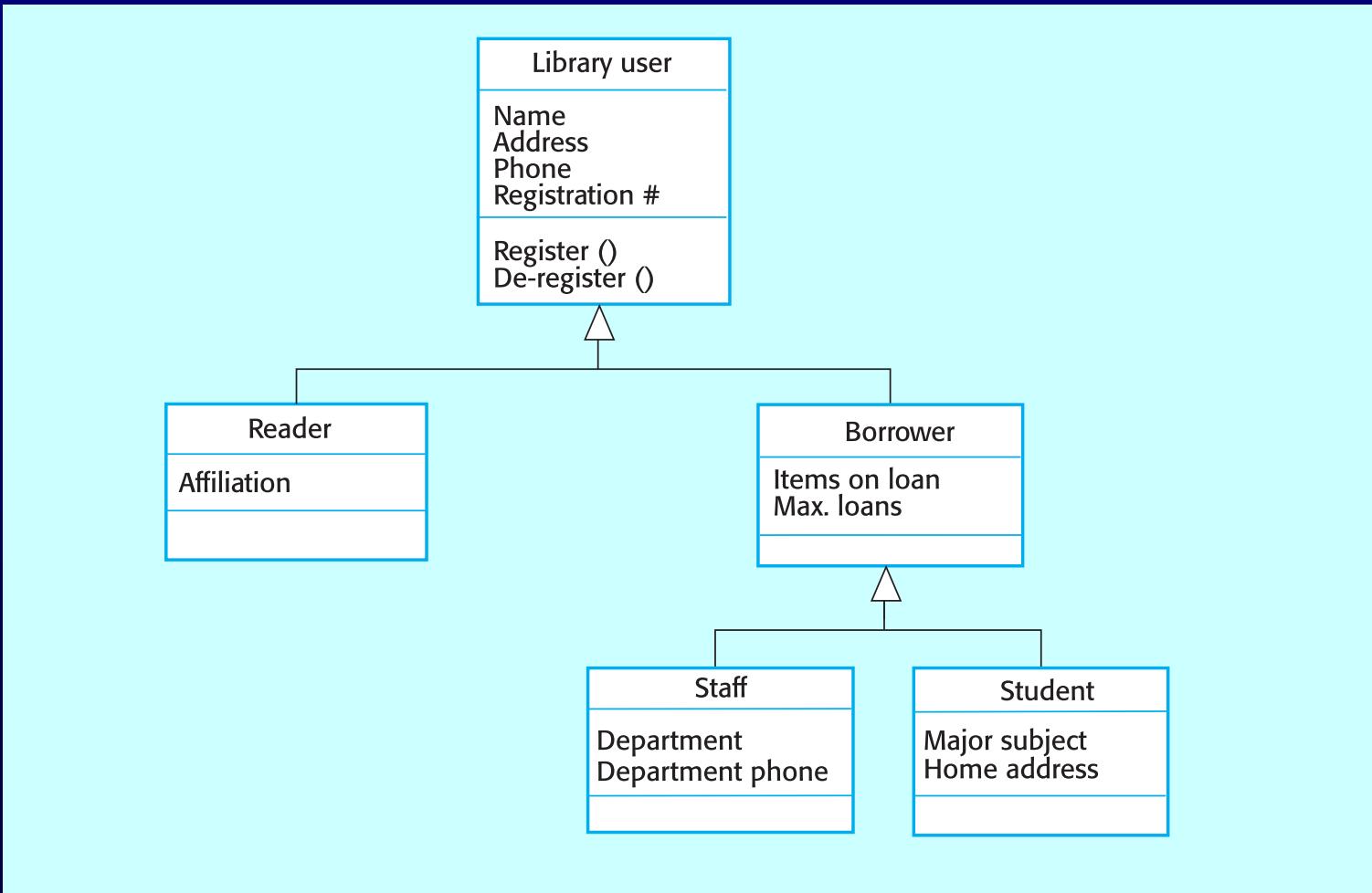
Object models and the UML

- The UML is a standard representation devised by the developers of widely used object-oriented analysis and design methods.
- It has become an effective standard for object-oriented modelling.
- Notation
 - Object classes are rectangles with the name at the top, attributes in the middle section and operations in the bottom section;
 - Relationships between object classes (known as associations) are shown as lines linking objects;
 - Inheritance is referred to as generalisation and is shown ‘upwards’ rather than ‘downwards’ in a hierarchy.

Library class hierarchy



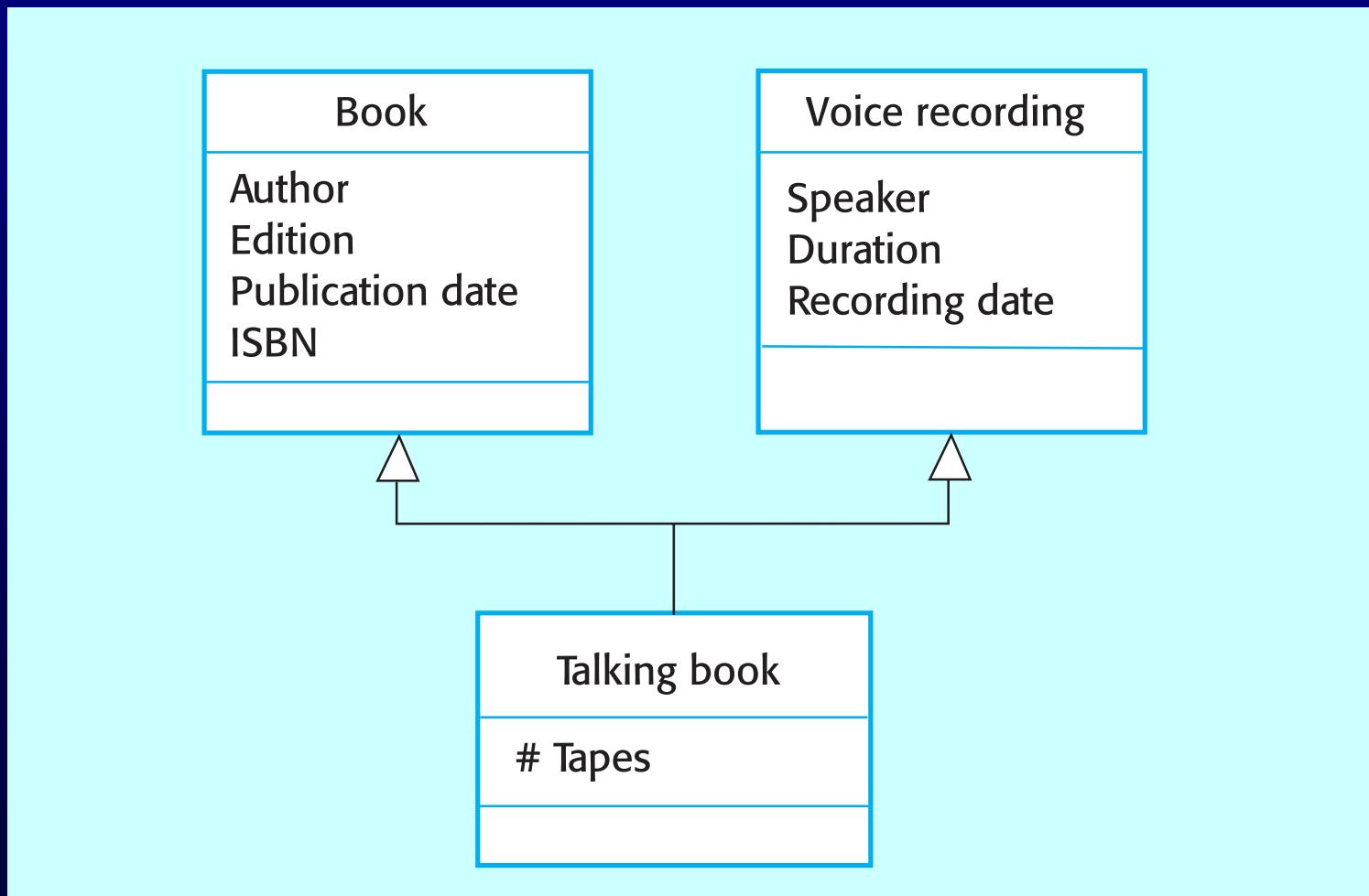
User class hierarchy



Multiple inheritance

- Rather than inheriting the attributes and services from a single parent class, a system which supports multiple inheritance allows object classes to inherit from several super-classes.
- This can lead to semantic conflicts where attributes/services with the same name in different super-classes have different semantics.
- Multiple inheritance makes class hierarchy reorganisation more complex.

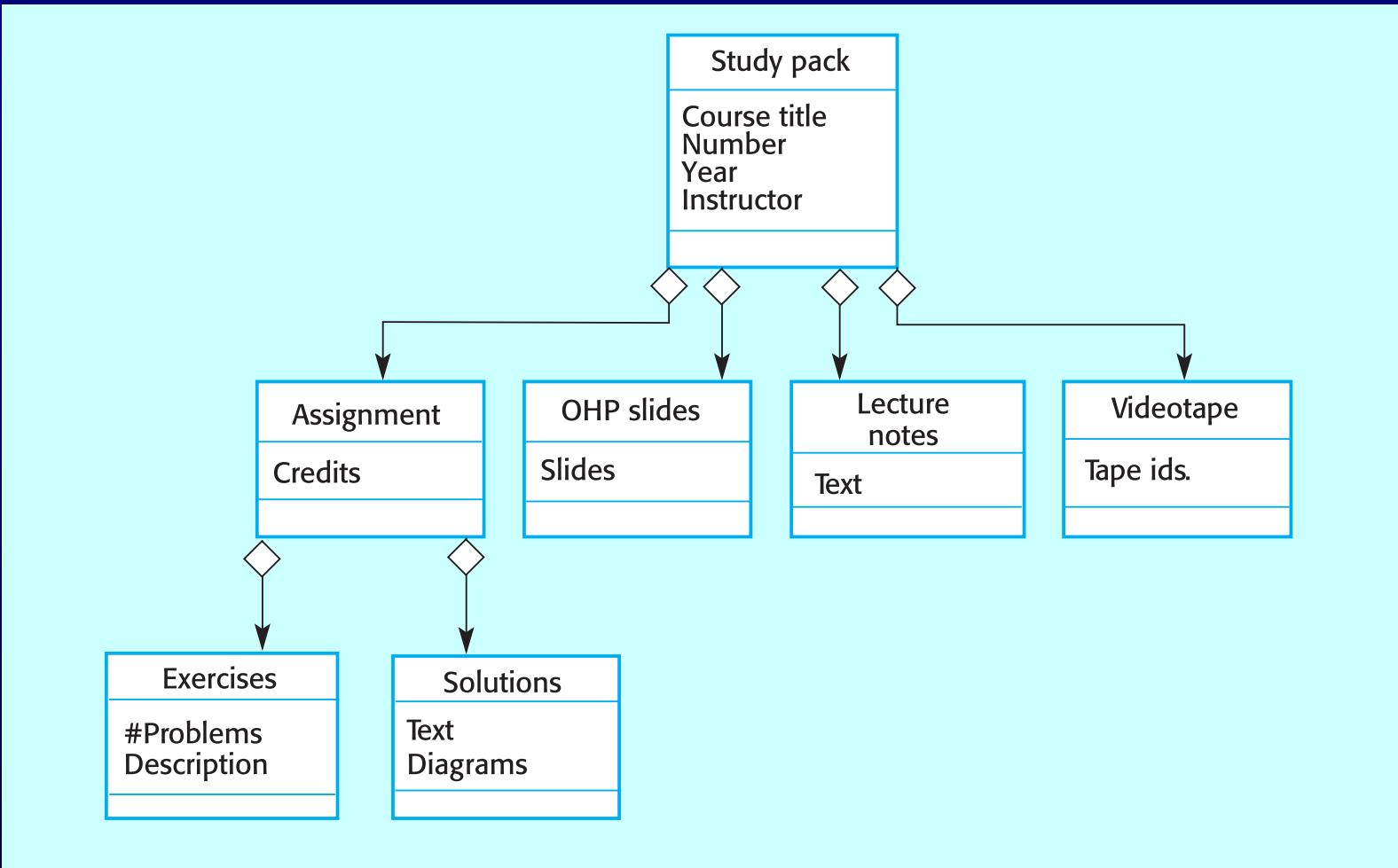
Multiple inheritance



Object aggregation

- An aggregation model shows how classes that are collections are composed of other classes.
- Aggregation models are similar to the part-of relationship in semantic data models.

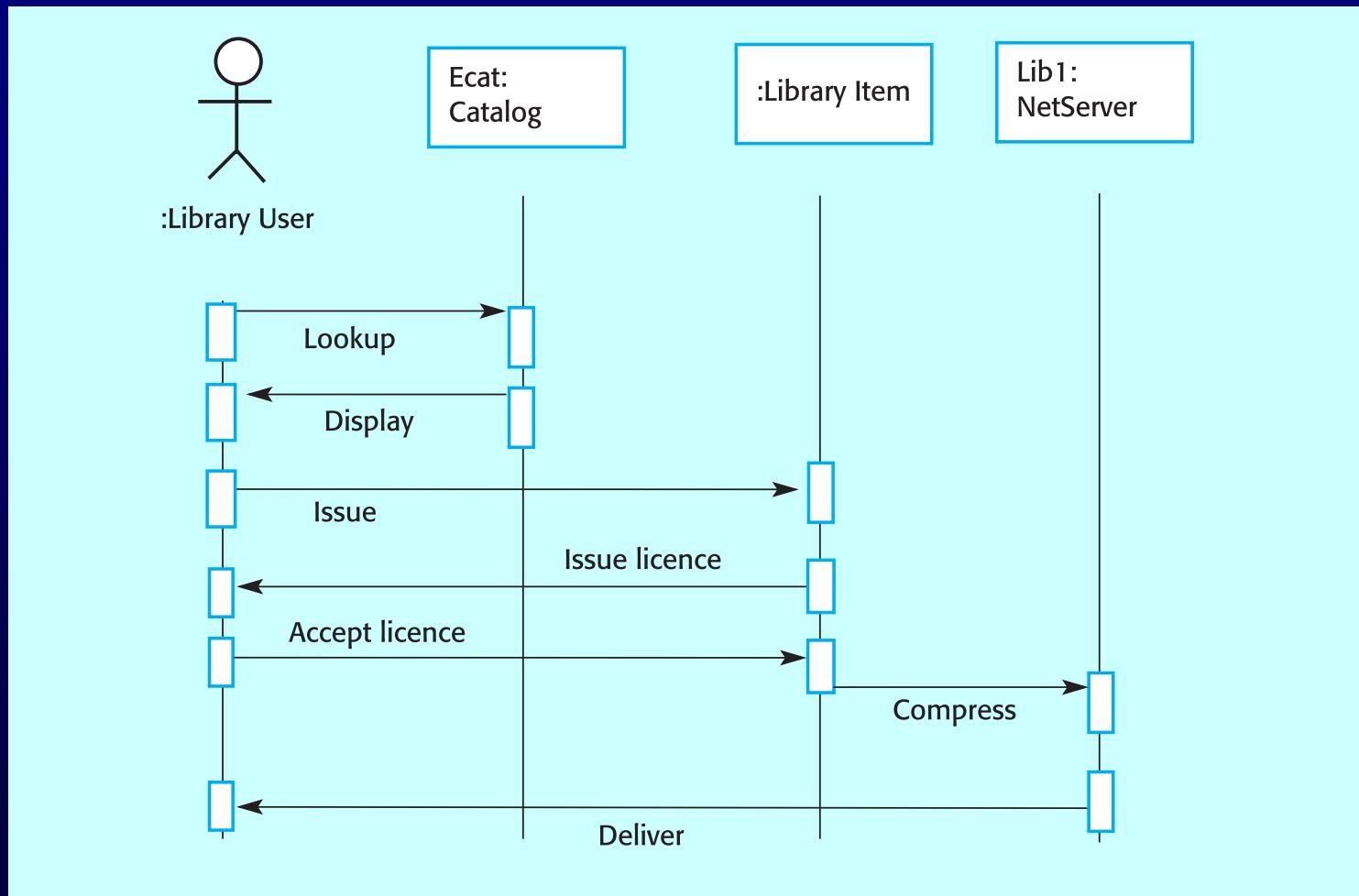
Object aggregation



Object behaviour modelling

- A behavioural model shows the interactions between objects to produce some particular system behaviour that is specified as a use-case.
- Sequence diagrams (or collaboration diagrams) in the UML are used to model interaction between objects.

Issue of electronic items



Structured methods

- Structured methods incorporate system modelling as an inherent part of the method.
- Methods define a set of models, a process for deriving these models and rules and guidelines that should apply to the models.
- CASE tools support system modelling as part of a structured method.

Method weaknesses

- They do not model non-functional system requirements.
- They do not usually include information about whether a method is appropriate for a given problem.
- They may produce too much documentation.
- The system models are sometimes too detailed and difficult for users to understand.

Architectural Design

Software architecture

- 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**.

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 conflicts

- Using large-grain components improves performance but reduces maintainability.
- Introducing redundant data improves availability but makes security more difficult.
- Localising safety-related features usually means more communication so degraded performance.

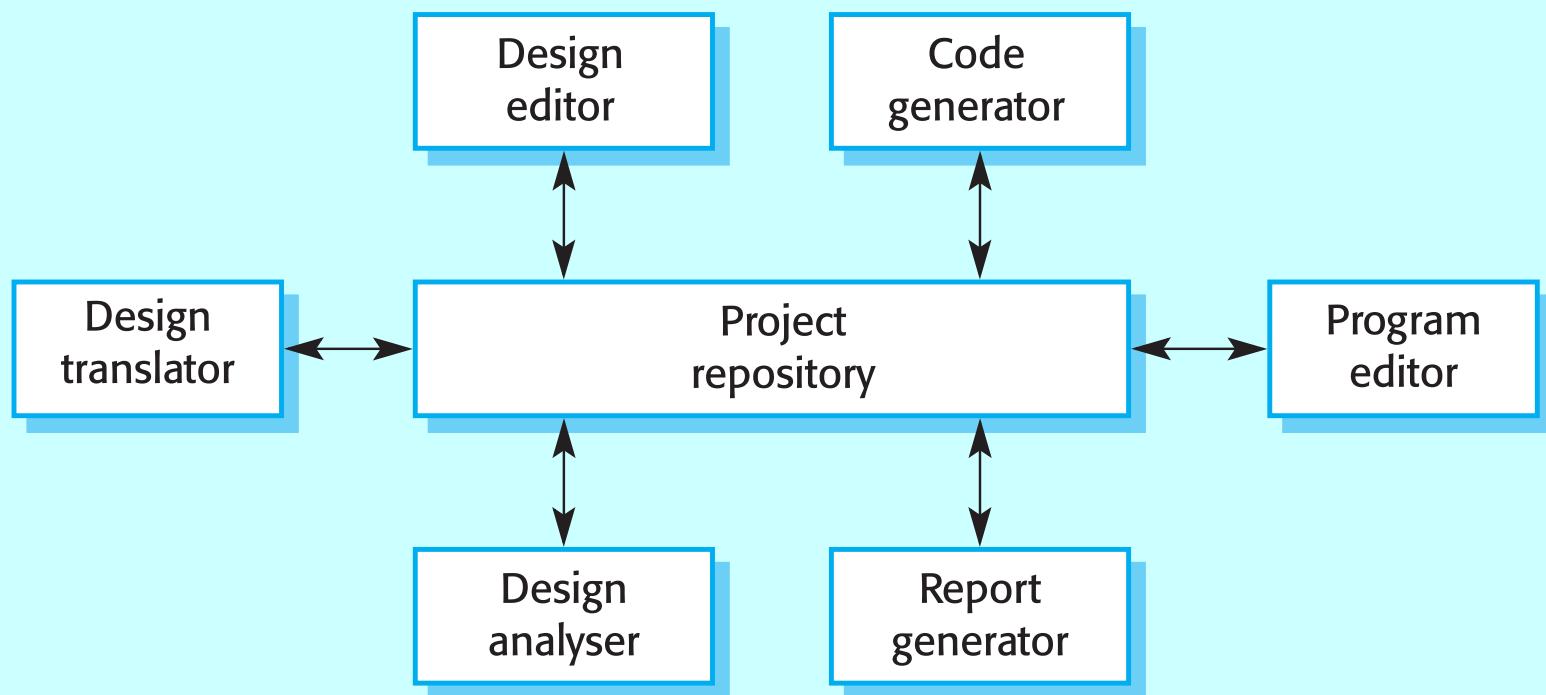
Architectural models

- 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.

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 toolset architecture



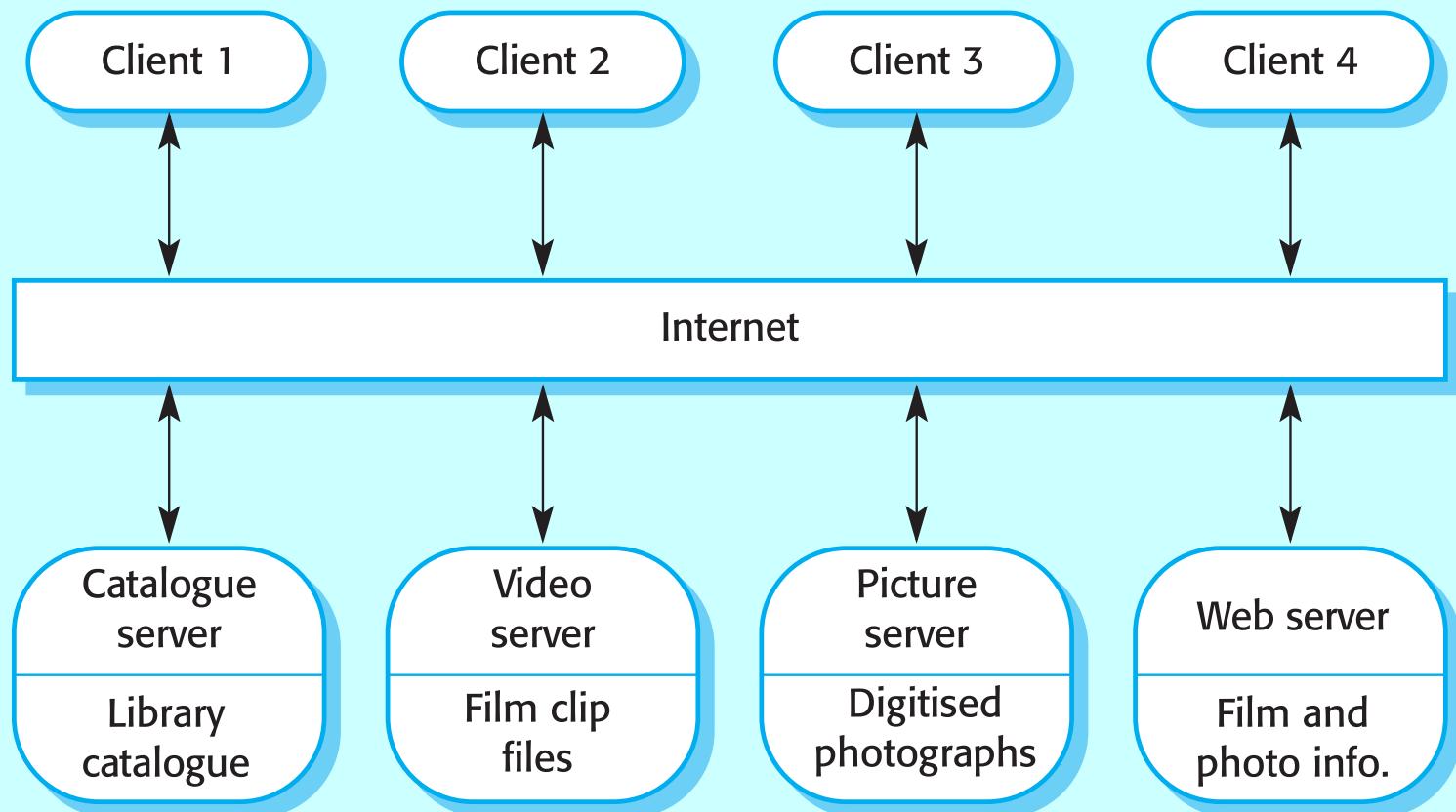
Repository model characteristics

- 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 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



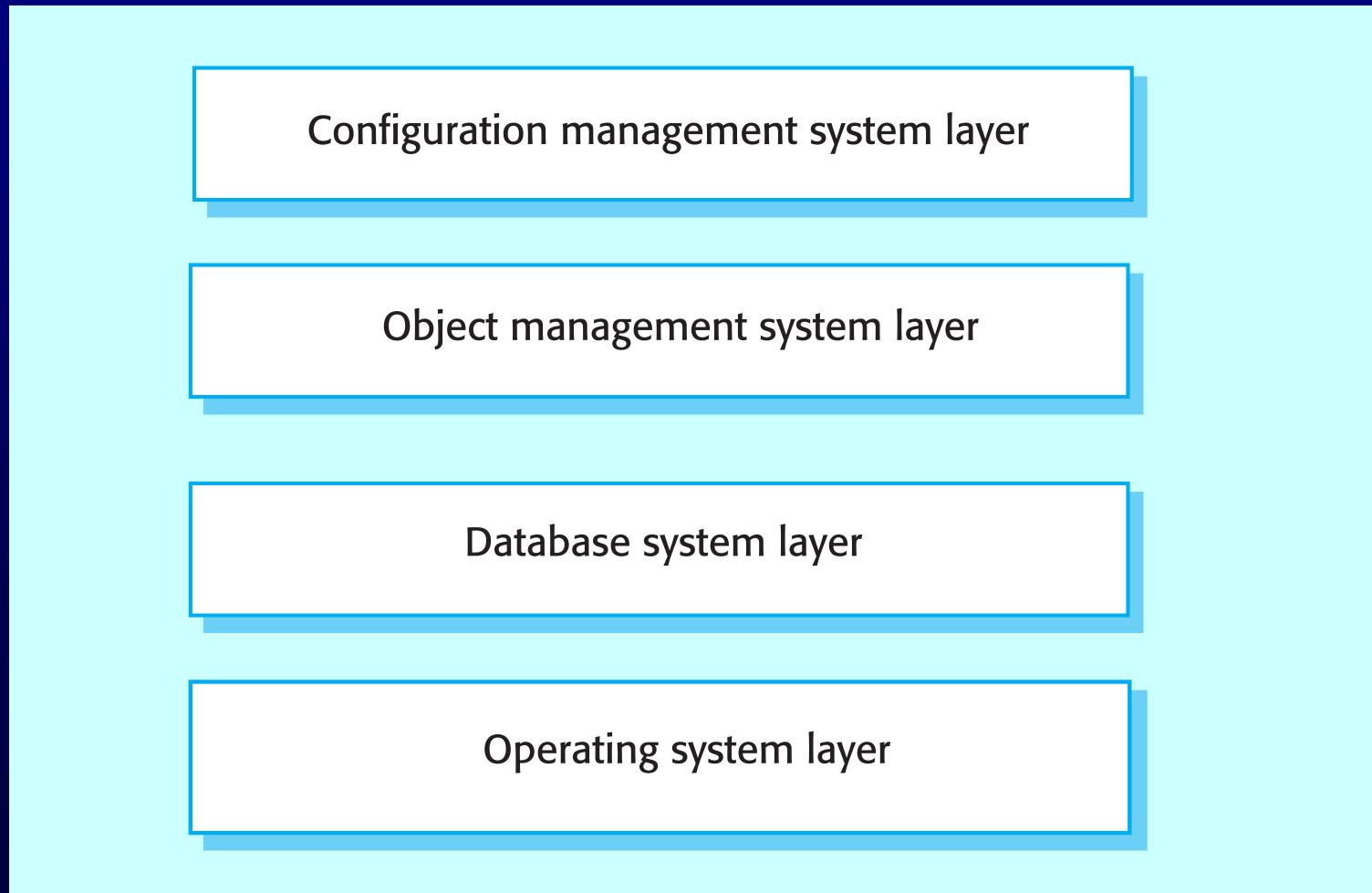
Client-server characteristics

- Advantages
 - Distribution of data is straightforward;
 - Makes effective use of networked systems. 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



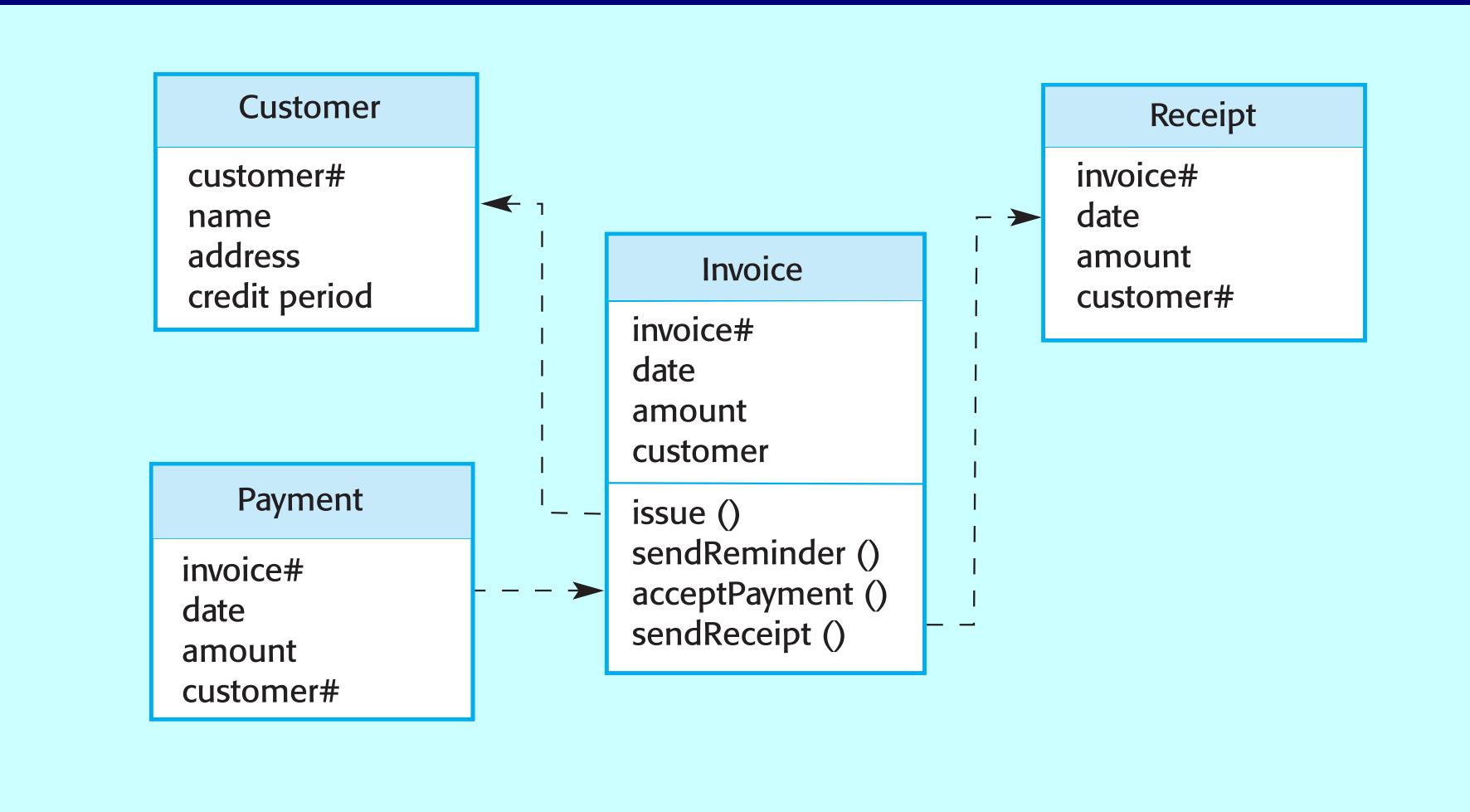
Modular decomposition

- 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 objects;
 - 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.

Invoice processing system



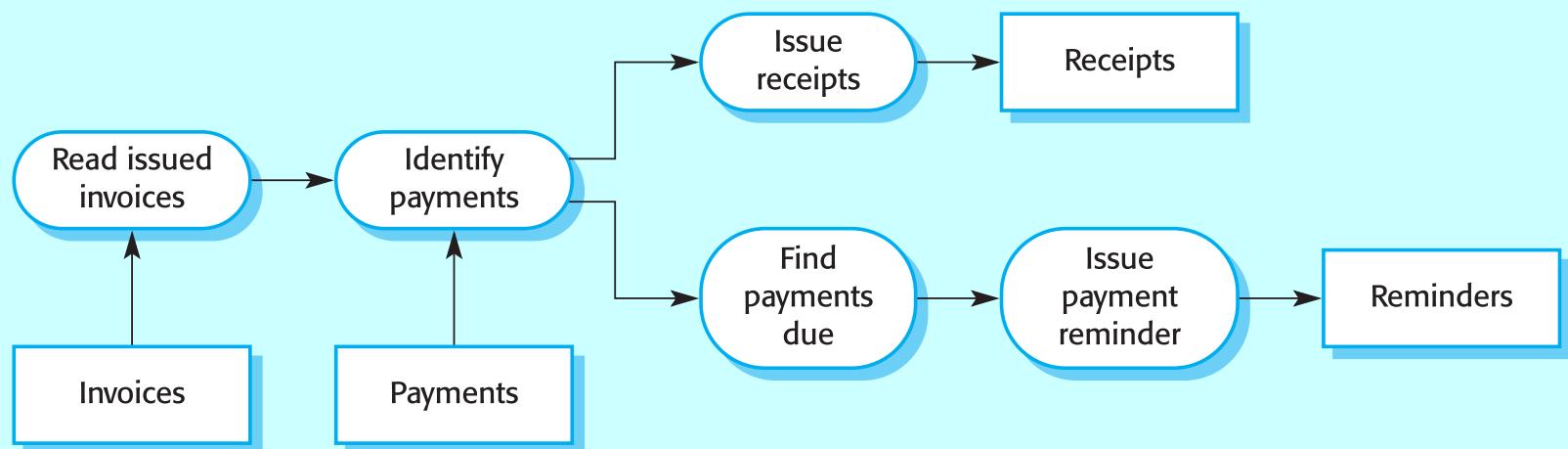
Object model advantages

- Objects are loosely coupled so 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.

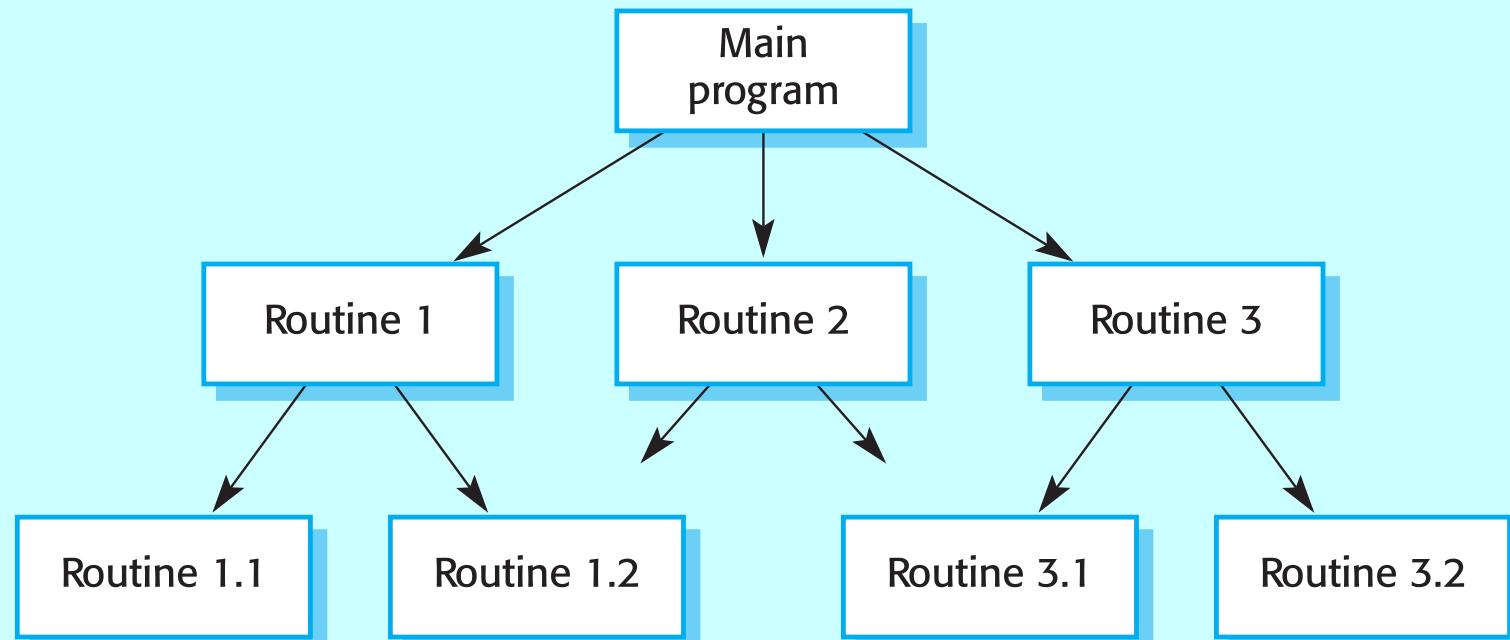
Control styles

- 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.

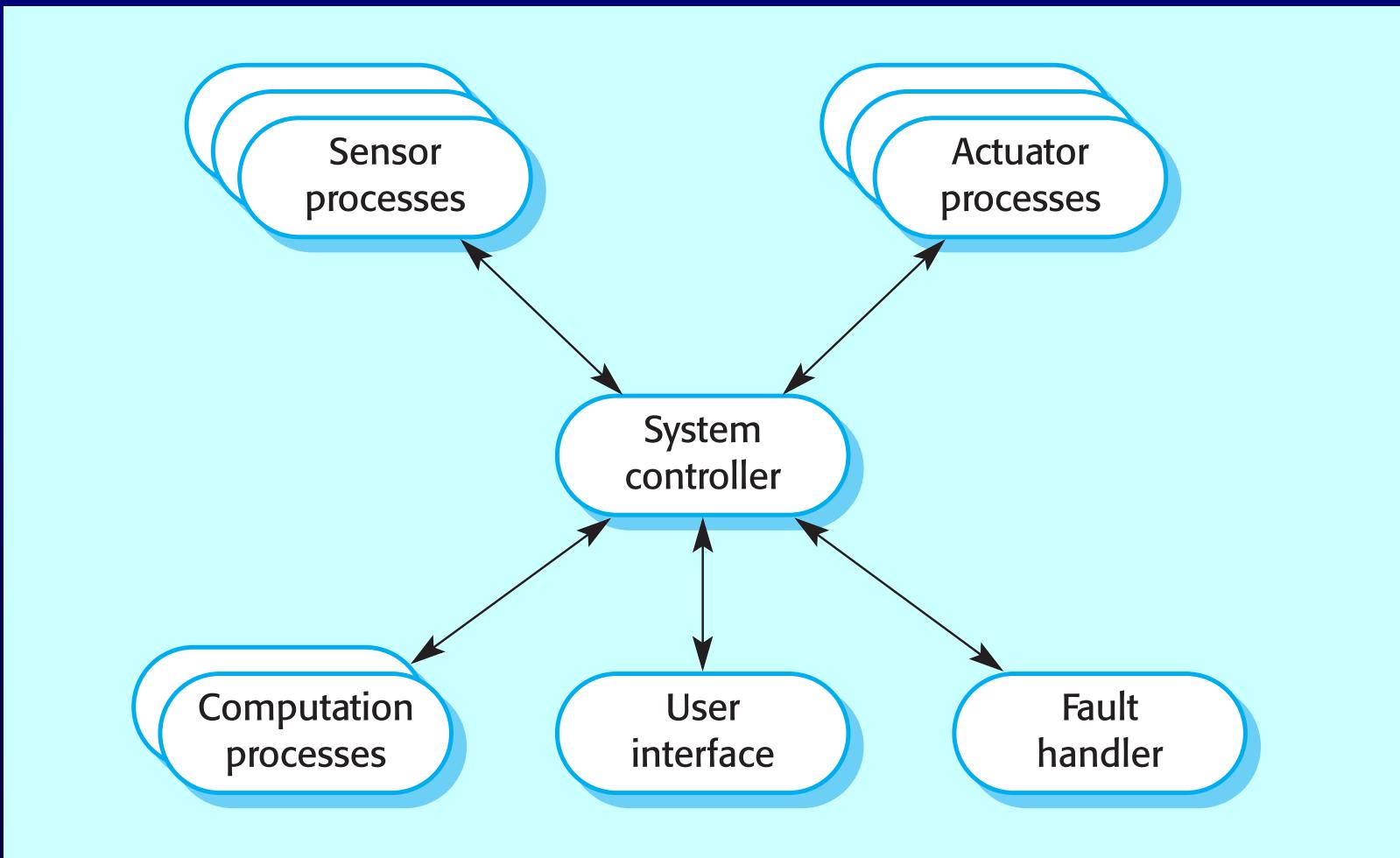
Centralised control

- 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.

Call-return model



Real-time system control



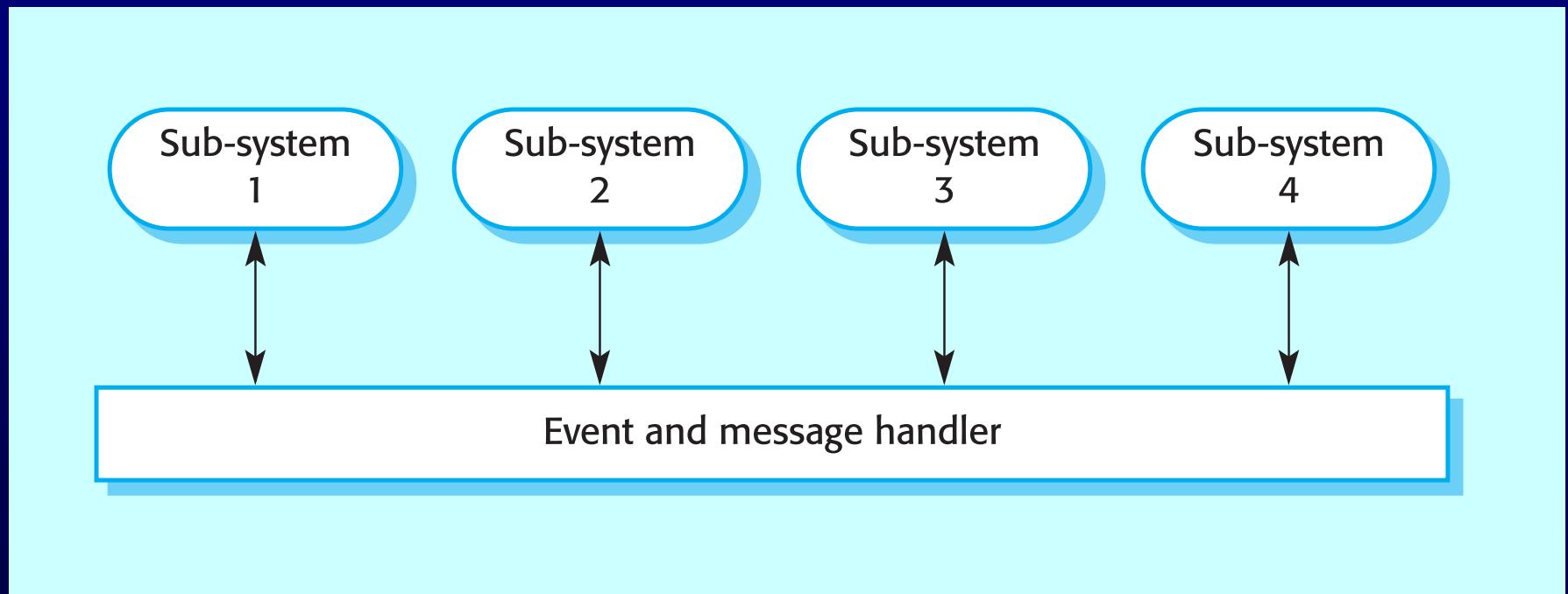
Event-driven systems

- 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.
- Other event driven models include spreadsheets and production systems.

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.

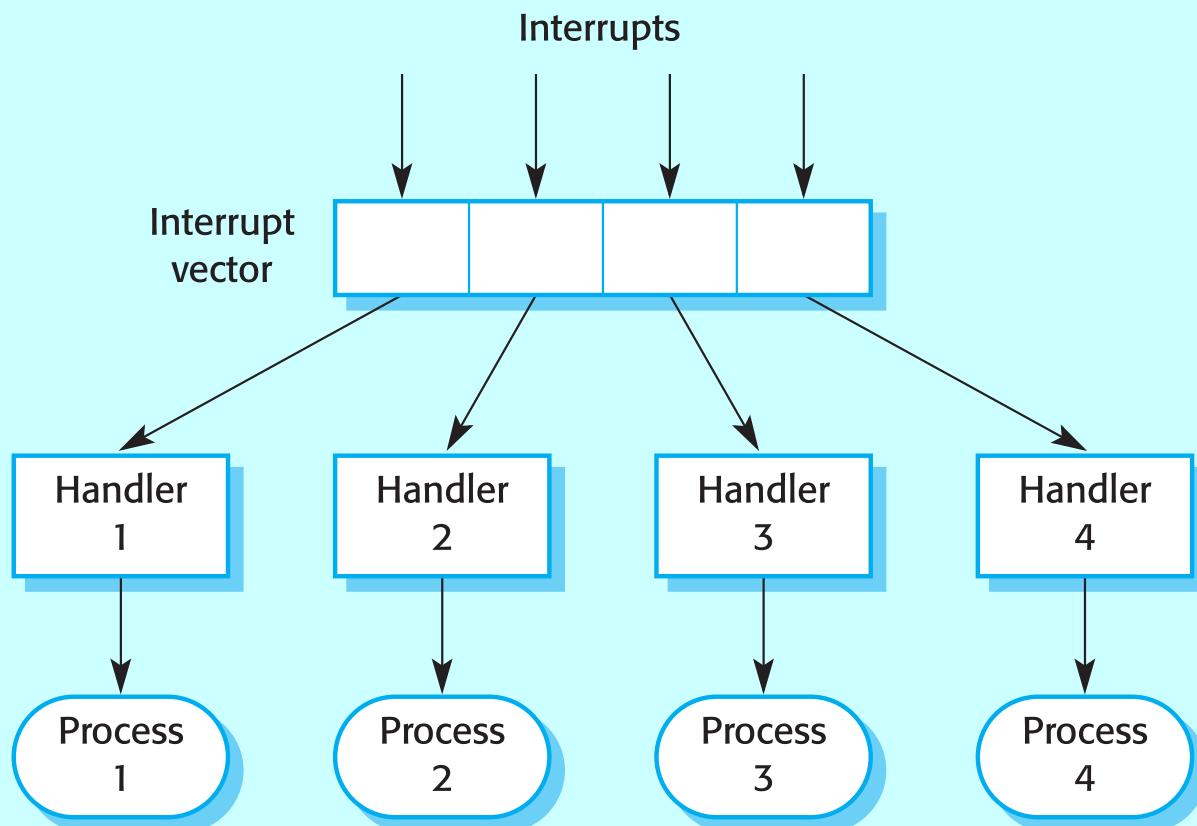
Selective broadcasting



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



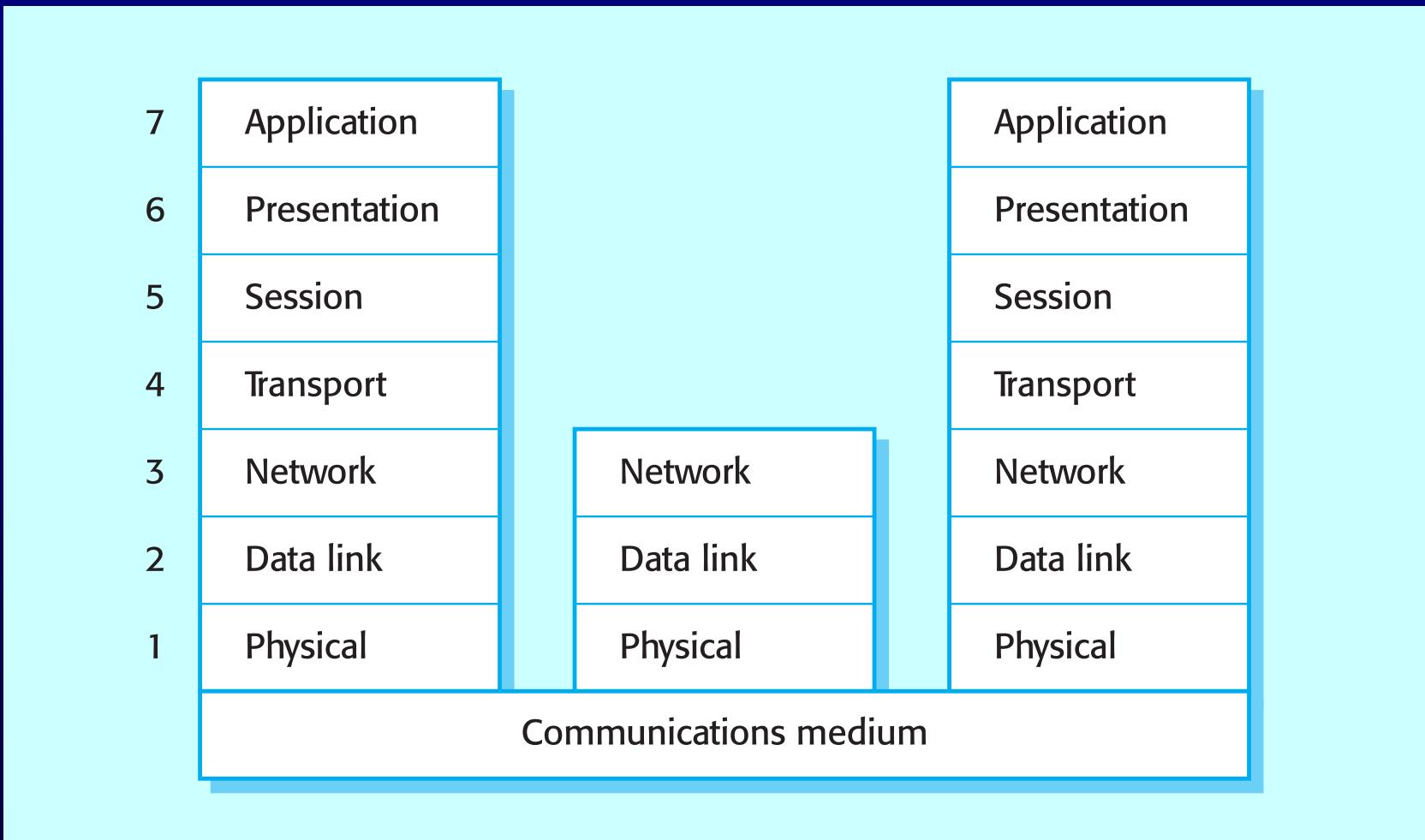
Reference architectures

- Architectural models may be 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. Covered in Chapter 13.
 - Reference models which are more abstract, idealised 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.

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

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.

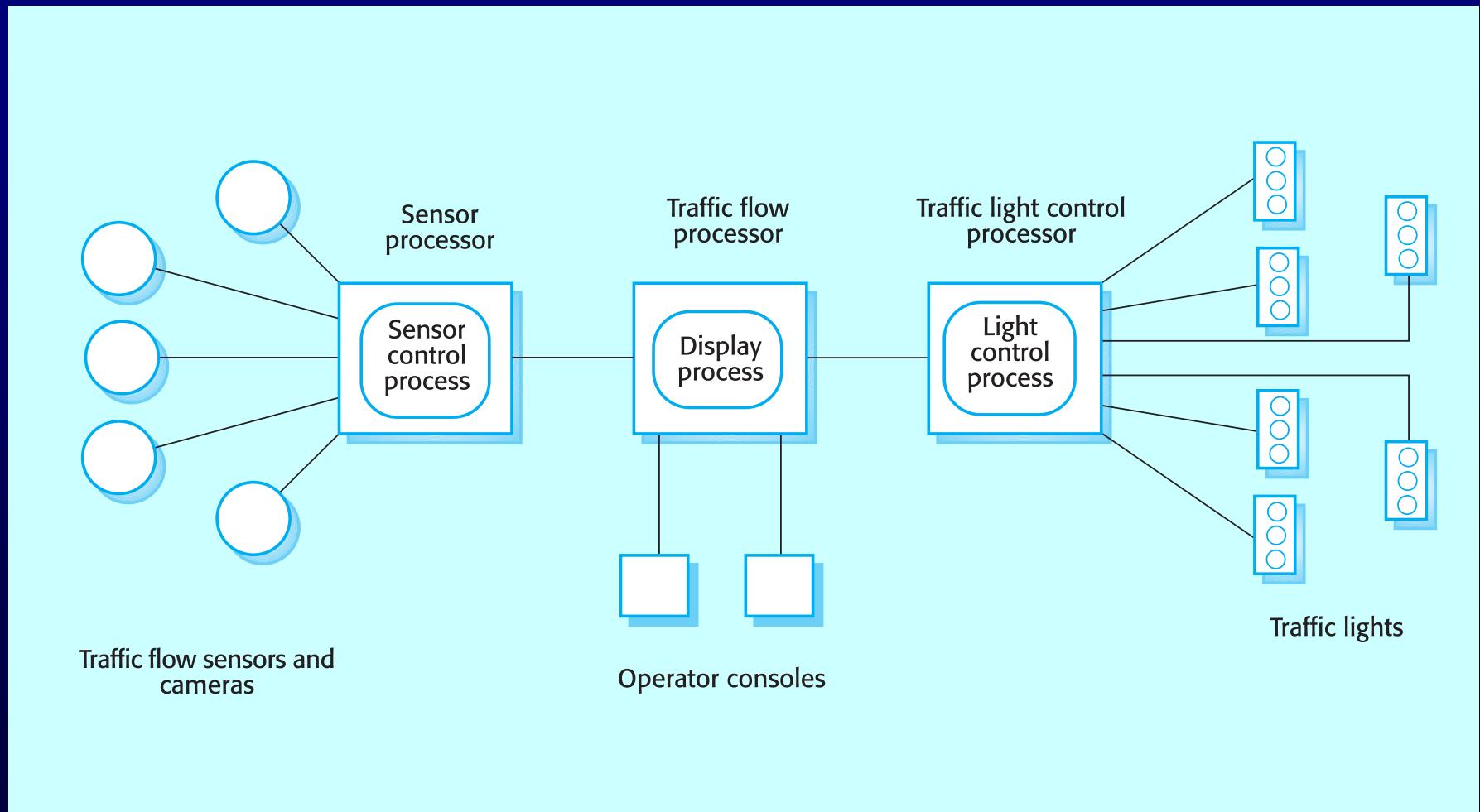
Distributed systems architectures

- 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.

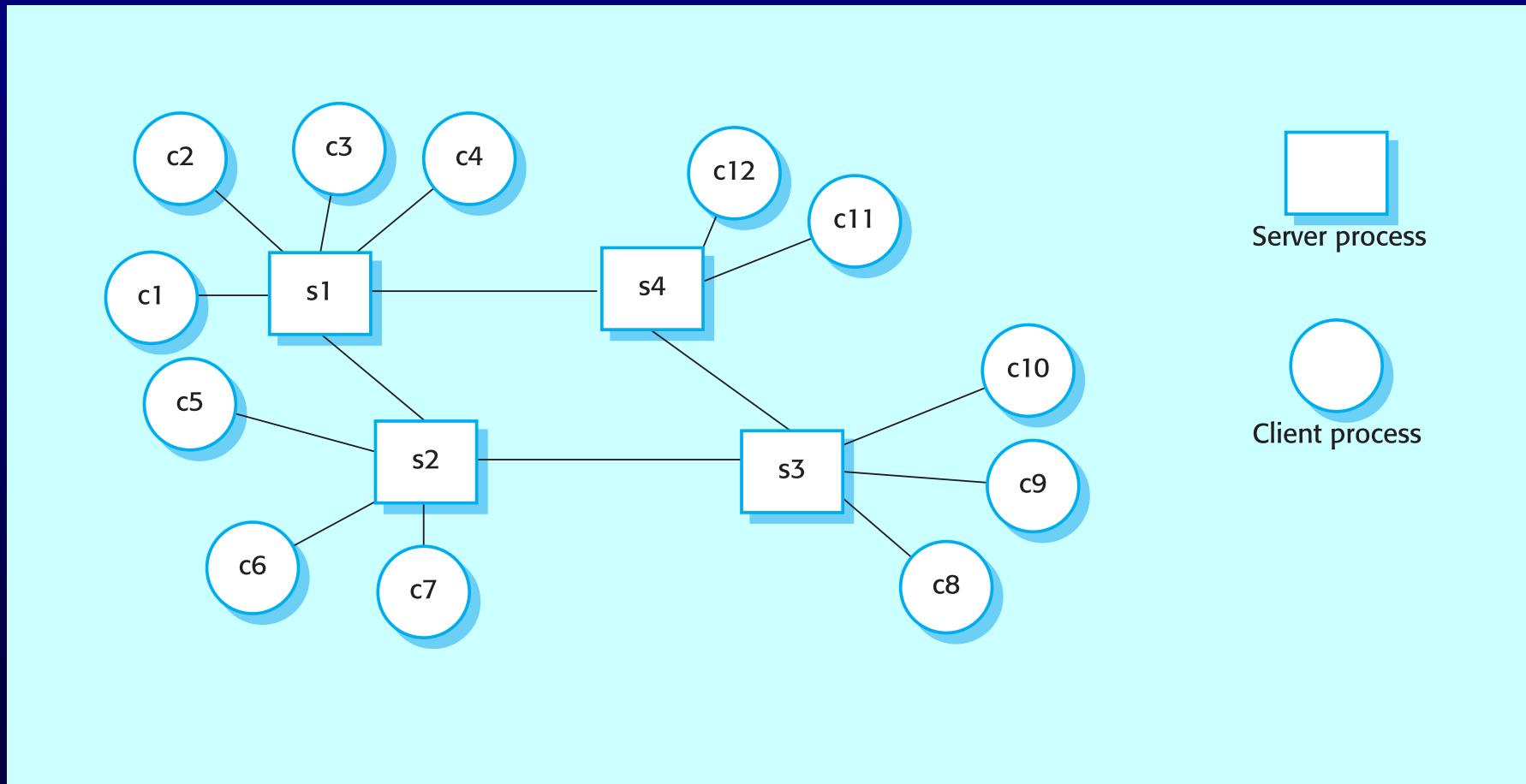
A multiprocessor traffic control system



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.

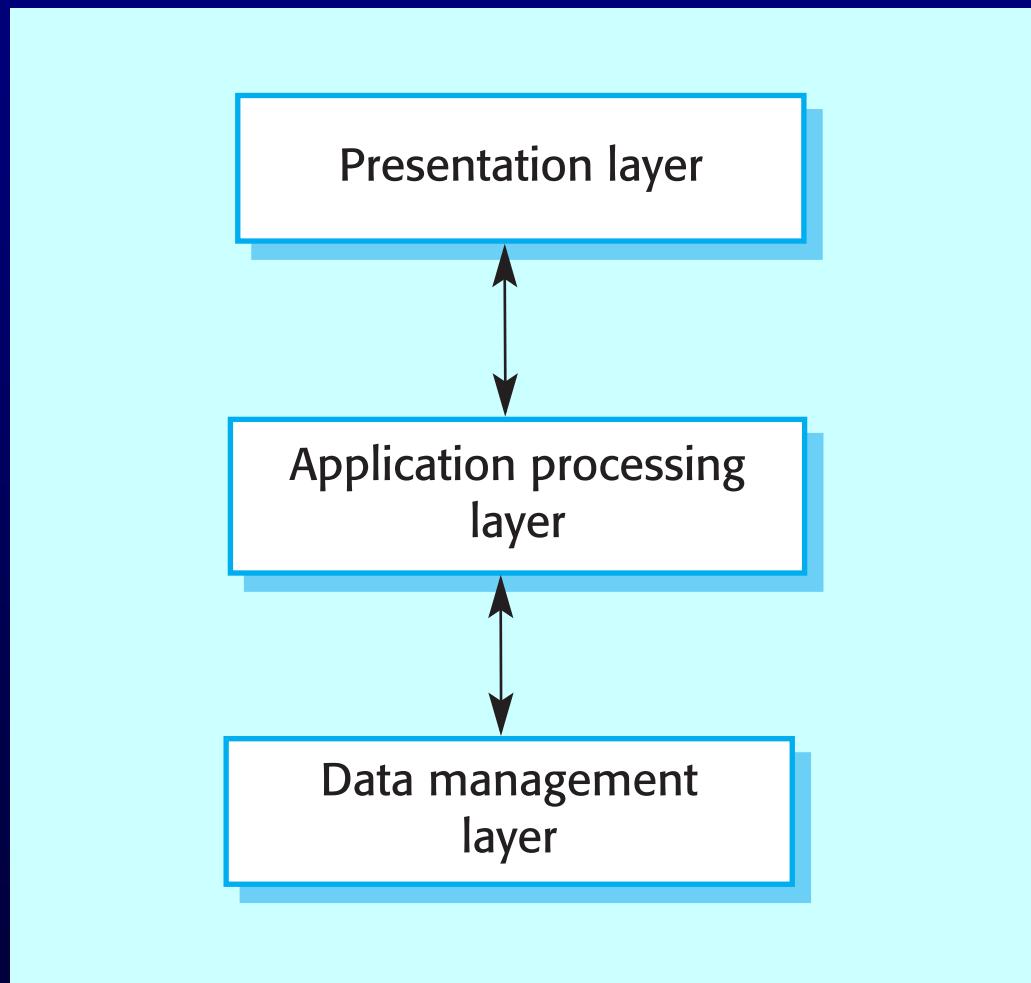
A client-server system



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.

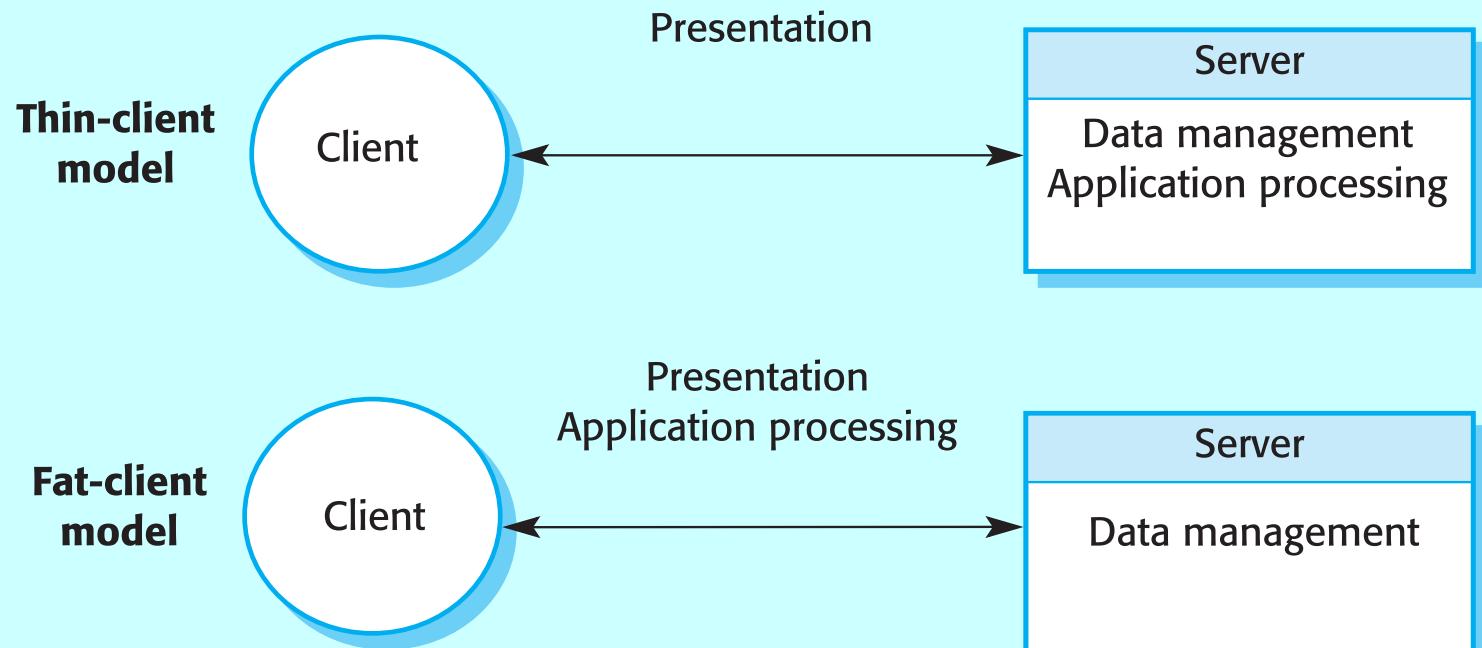
Application layers



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.

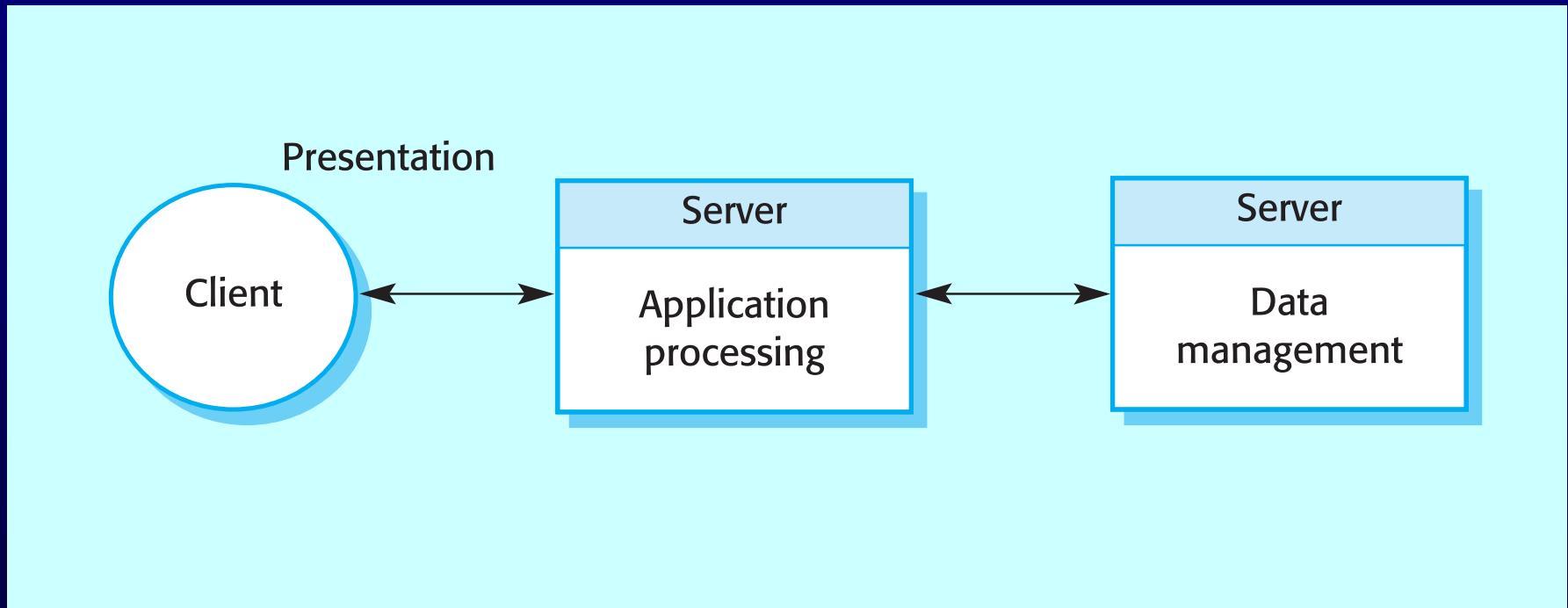
Thin and fat clients



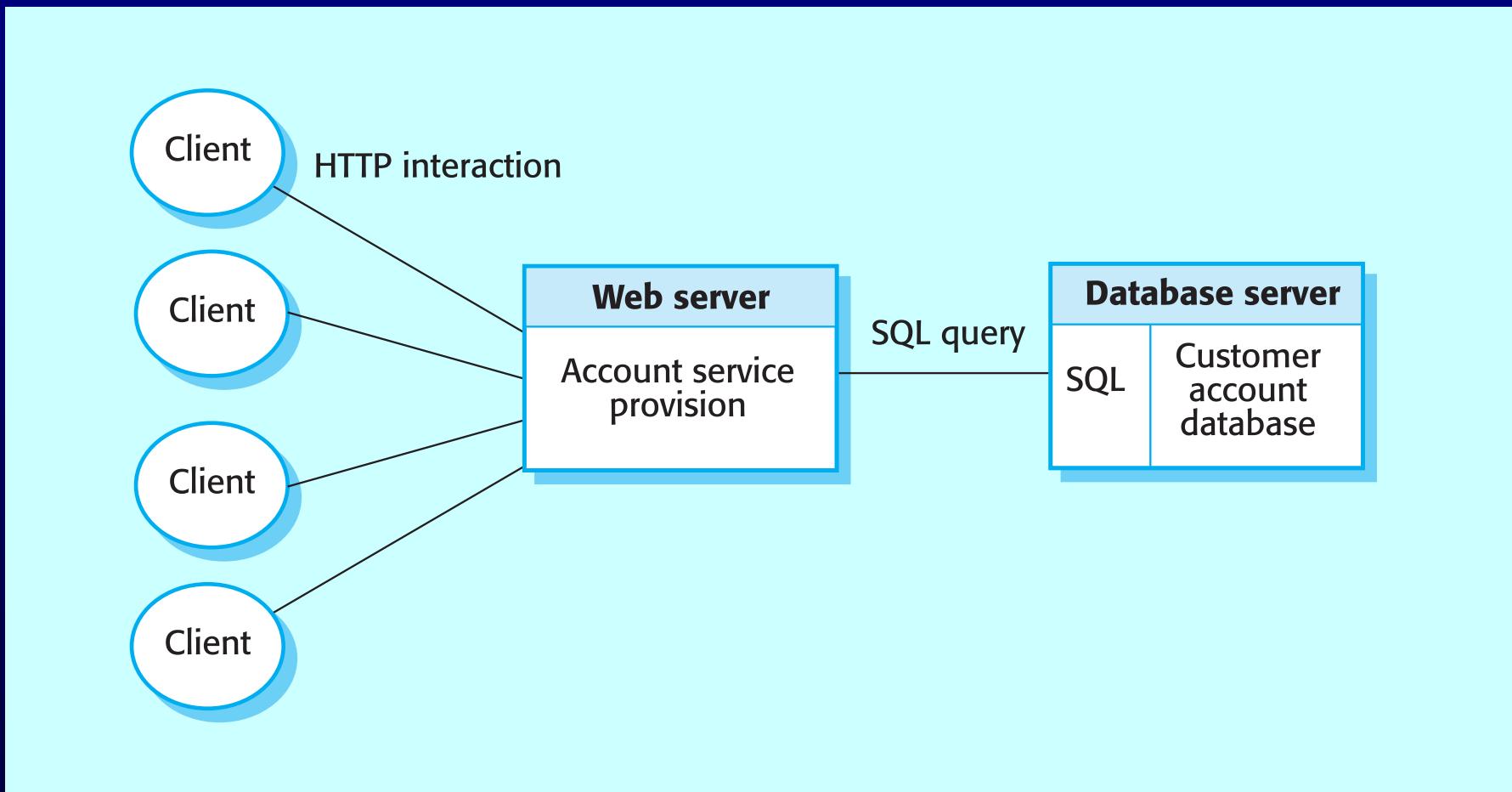
Three-tier architectures

- 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



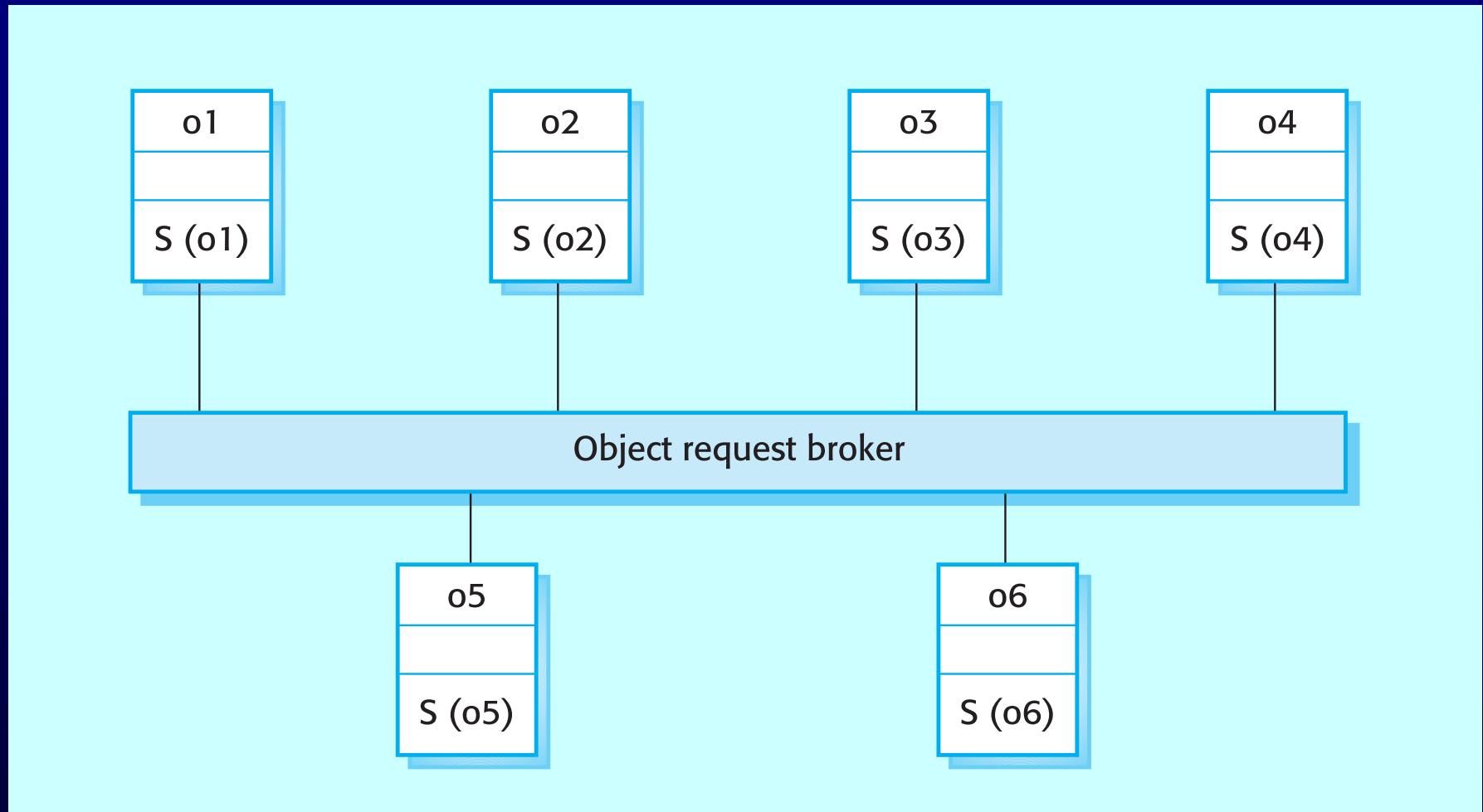
Use of C/S architectures

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

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.

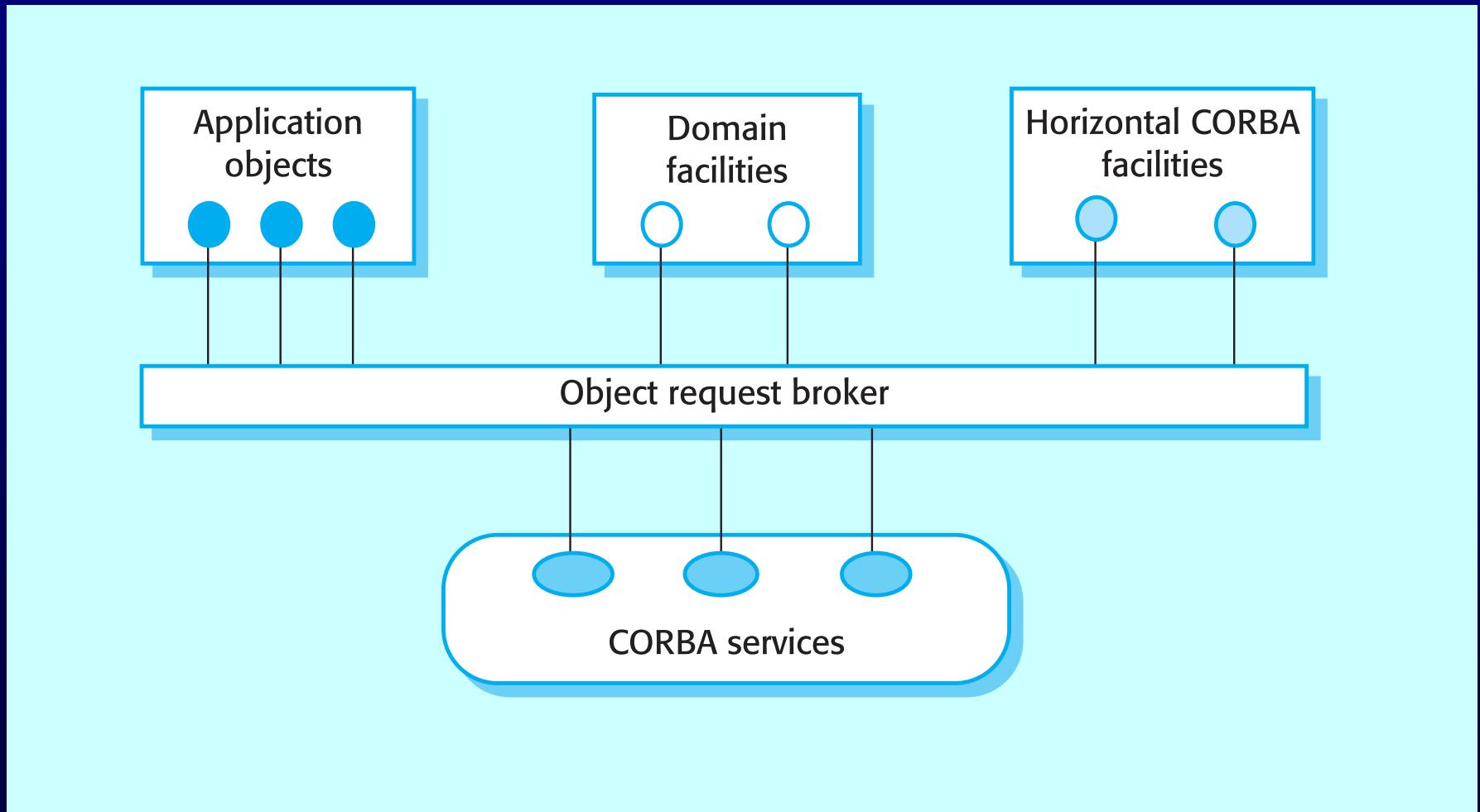
Distributed object architecture



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.

CORBA application structure



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 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.

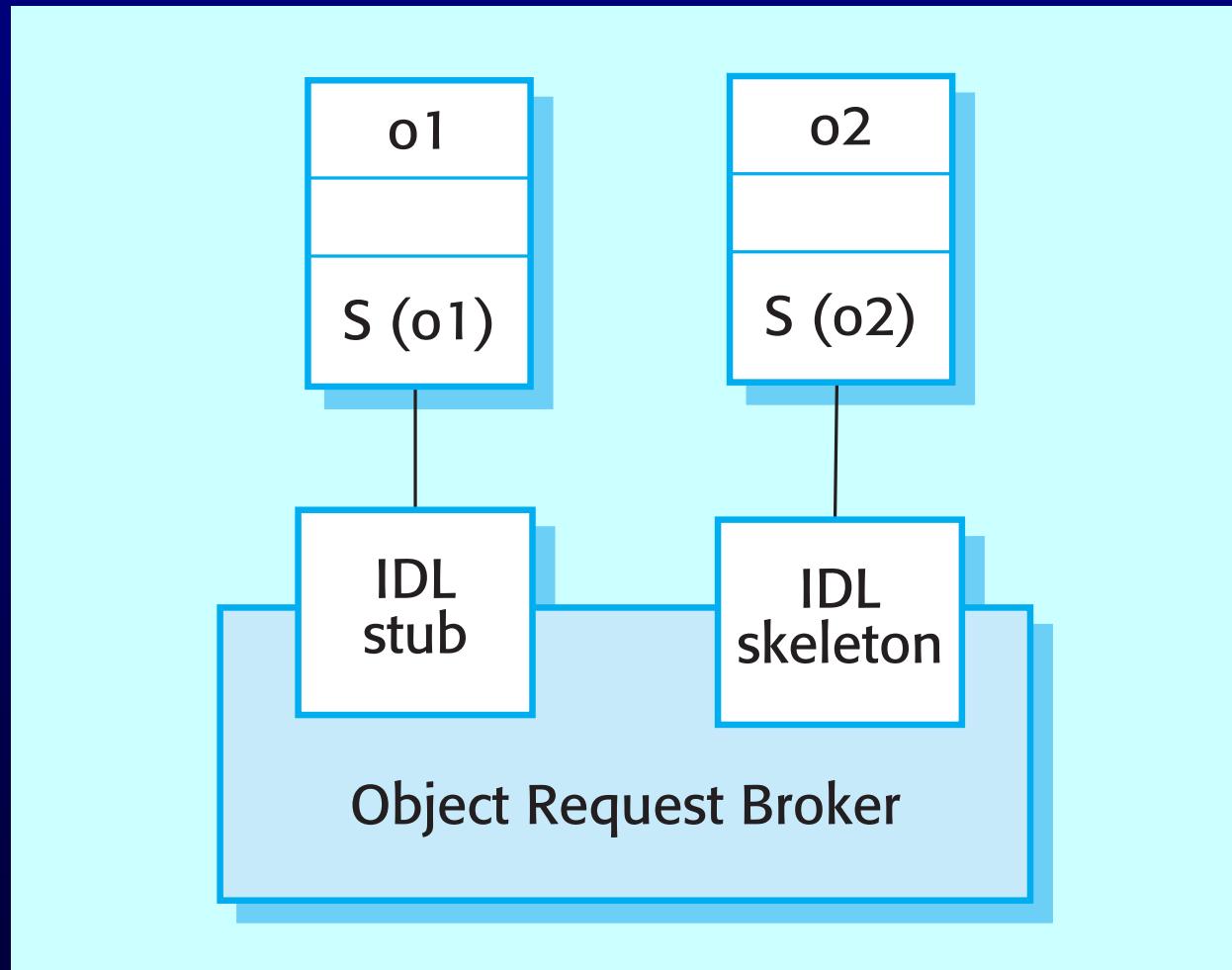
CORBA objects

- 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.

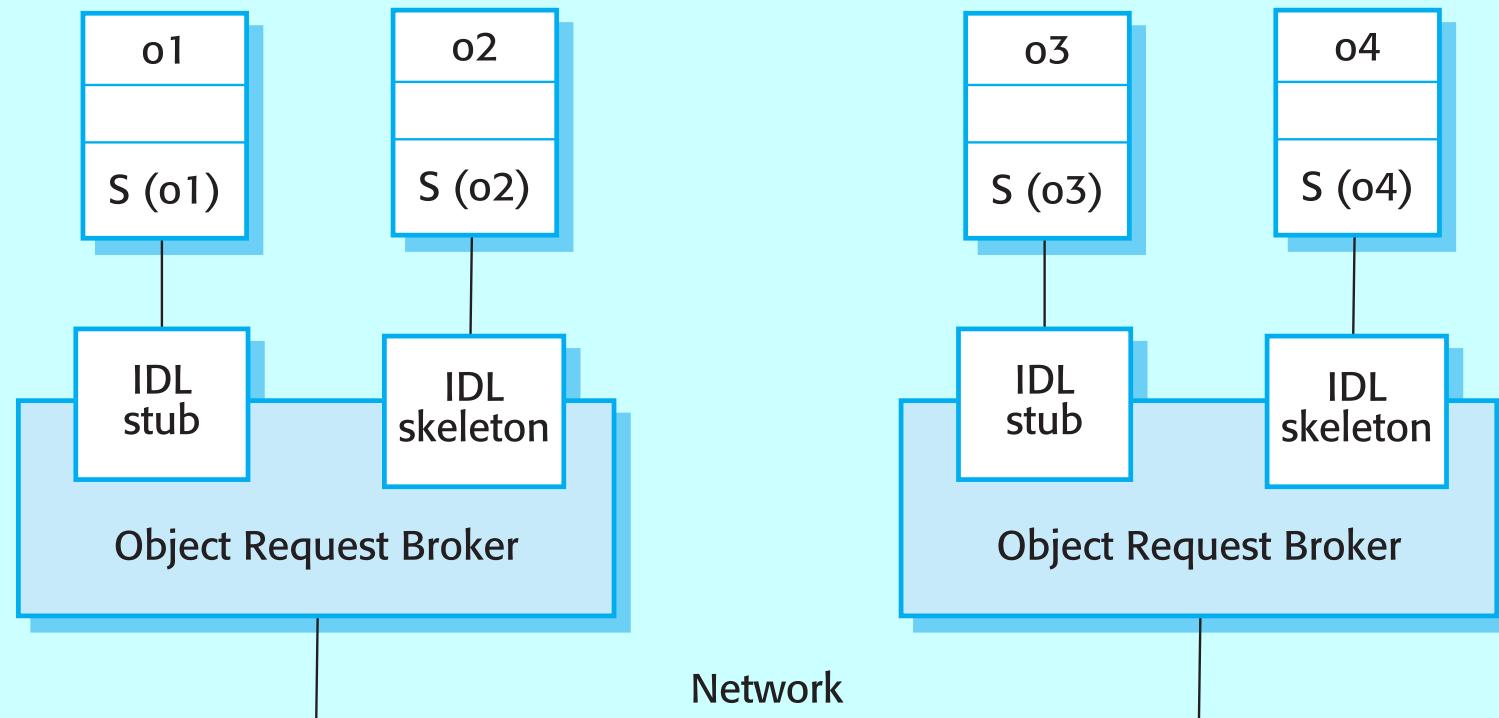
ORB-based object communications



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.

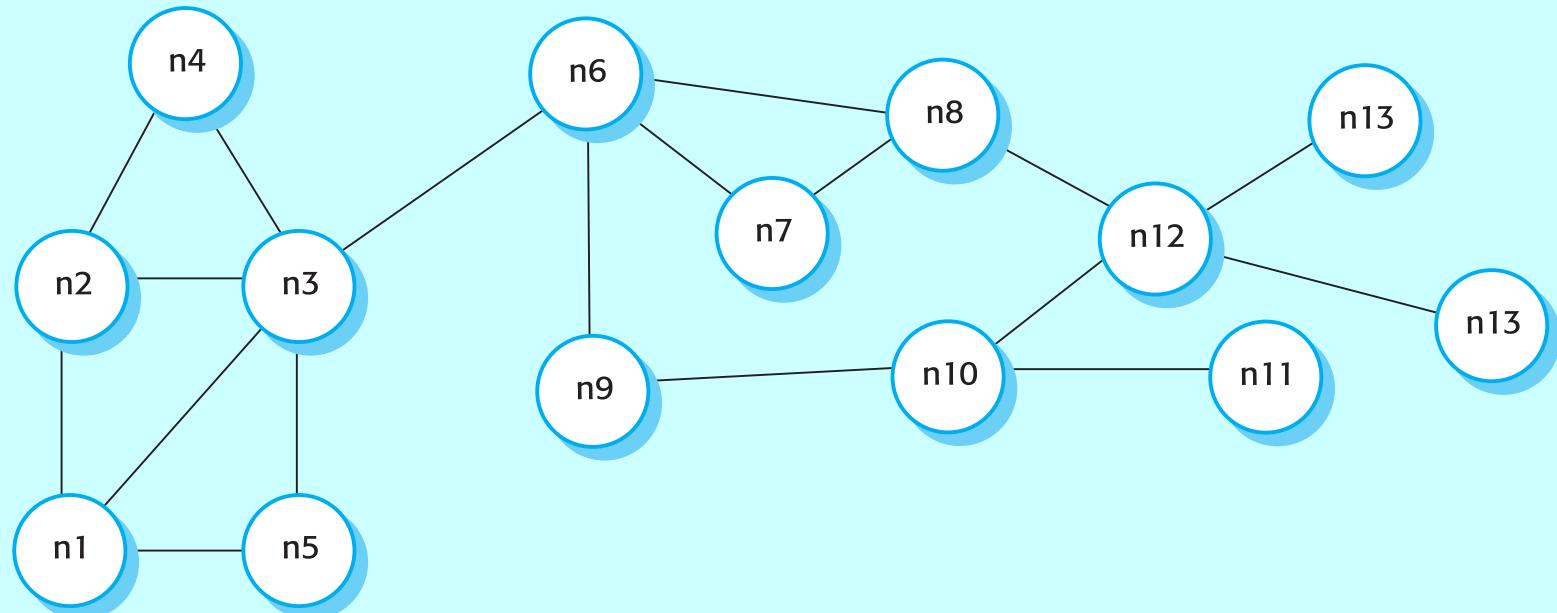
Inter-organisational computing

- For security and inter-operability reasons, most distributed computing has been implemented at the enterprise level.
- Local standards, management and operational processes apply.
- Newer models of distributed computing have been designed to support inter-organisational computing where different nodes are located in different organisations.

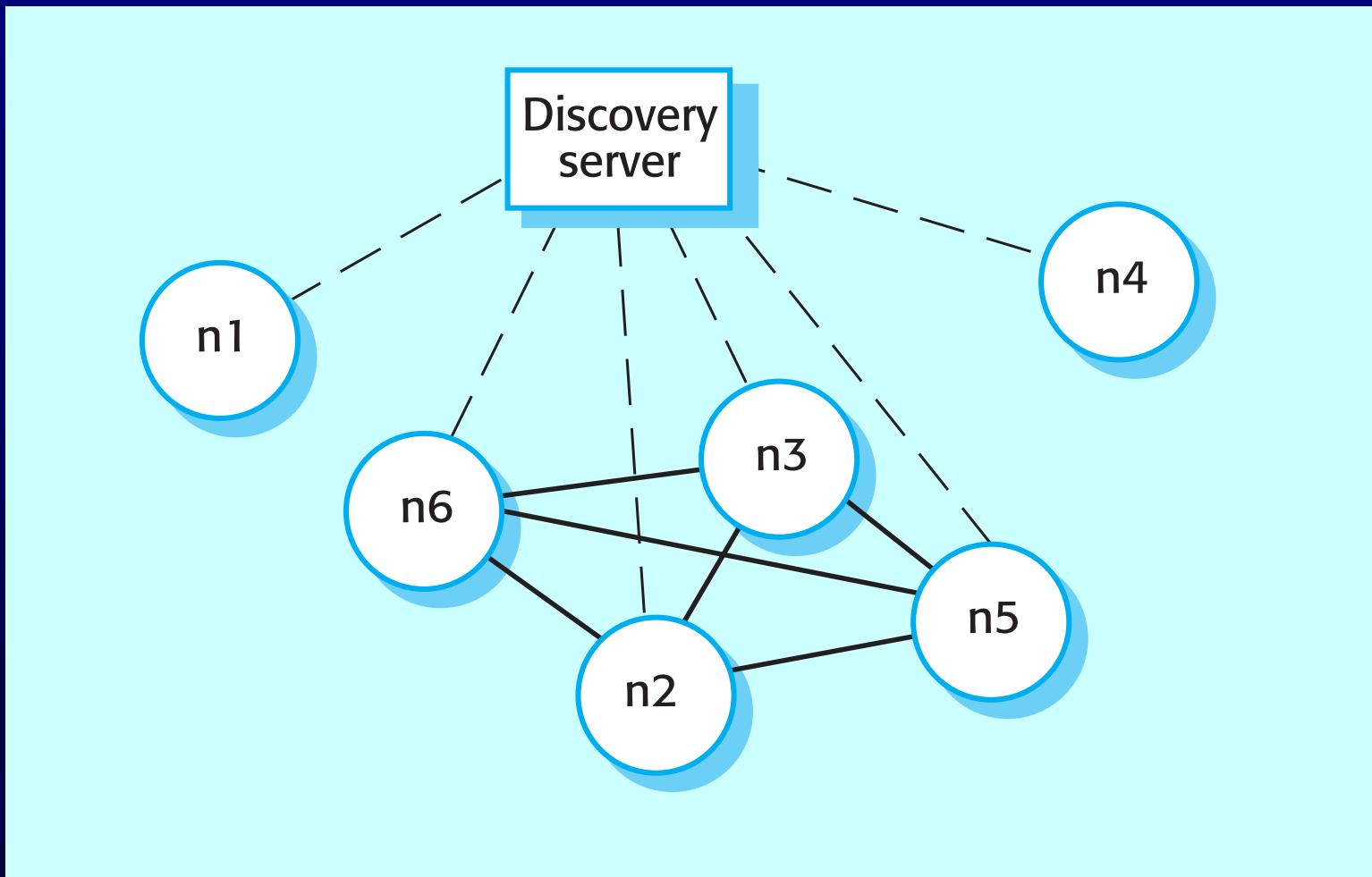
Peer-to-peer architectures

- 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



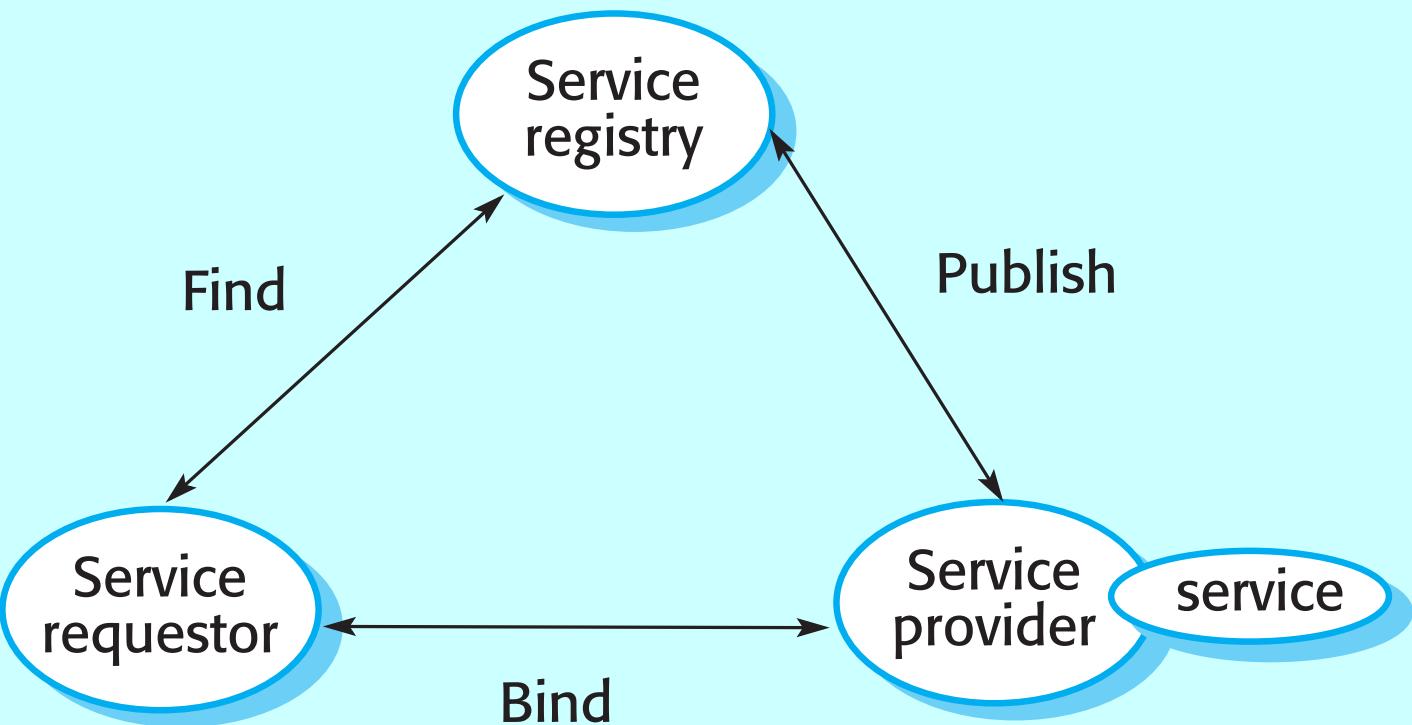
Service-oriented architectures

- 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.

A generic service

- *An act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production.*
- Service provision is therefore independent of the application using the service.

Web services



Services and distributed objects

- Provider independence.
- Public advertising of service availability.
- Potentially, run-time service binding.
- Opportunistic construction of new services through composition.
- Pay for use of services.
- Smaller, more compact applications.
- Reactive and adaptive applications.

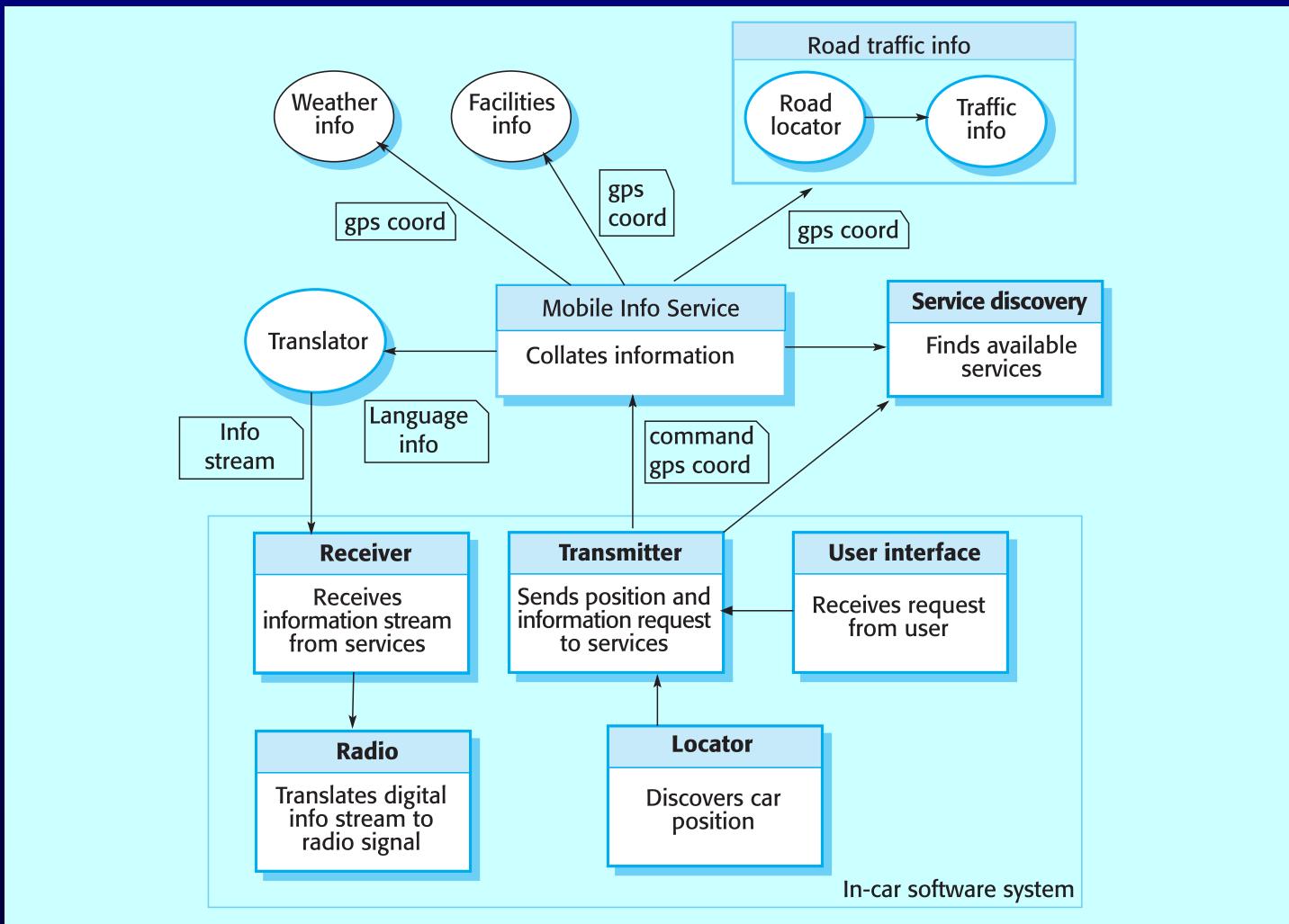
Services standards

- 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.

Services scenario

- An in-car information system provides drivers with information on weather, road traffic conditions, local information etc. This is linked to car radio so that information is delivered as a signal on a specific radio channel.
- The car is equipped with GPS receiver to discover its position and, based on that position, the system accesses a range of information services. Information may be delivered in the driver's specified language.

Automotive system



Key points

- Distributed systems support resource sharing, openness, concurrency, scalability, fault tolerance and transparency.
- Client-server architectures involve services being delivered by servers to programs operating on clients.
- User interface software always runs on the client and data management on the server. Application functionality may be on the client or the server.
- In a distributed object architecture, there is no distinction between clients and servers.

Key points

- Distributed object systems require middleware to handle object communications and to add and remove system objects.
- The CORBA standards are a set of middleware standards that support distributed object architectures.
- Peer to peer architectures are decentralised architectures where there is no distinction between clients and servers.
- Service-oriented systems are created by linking software services provided by different service suppliers.

Object-oriented Design

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.
- Objects are potentially reusable components.
- For some systems, there may be an obvious mapping from real world entities to system objects.

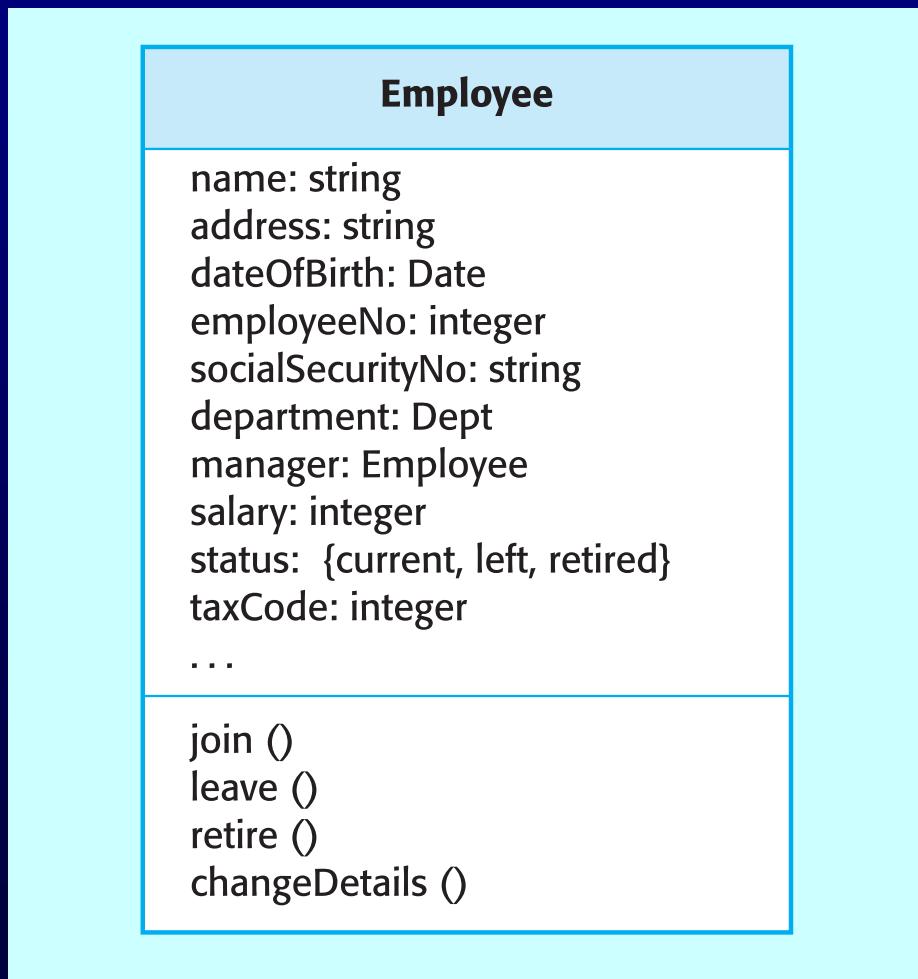
Objects and object classes

- Objects are entities in a software system which represent instances of real-world and system entities.
- Object classes are templates for objects. They may be used to create objects.
- Object classes may inherit attributes and services from other object classes.

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.

Employee object class (UML)



Object communication

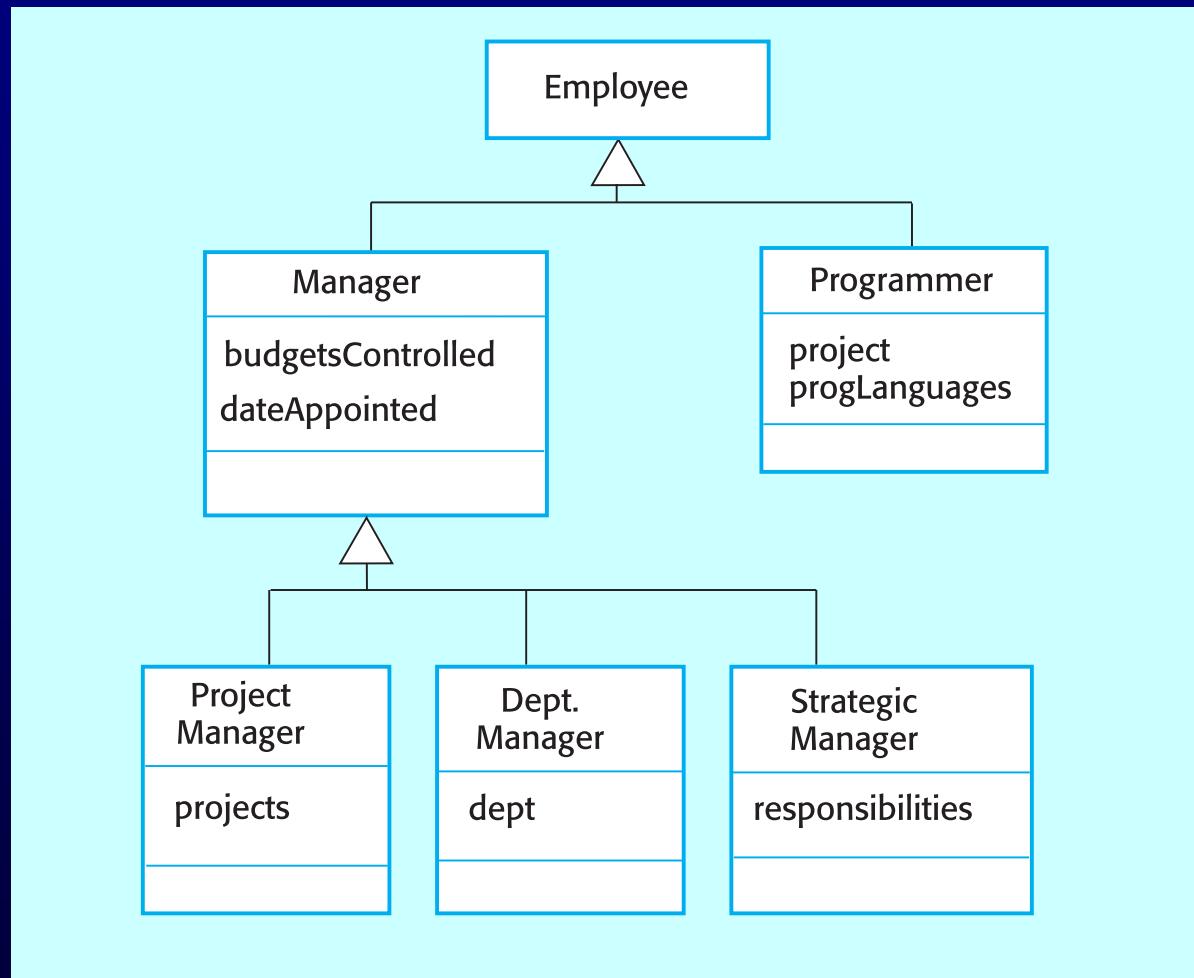
```
// Call a method associated with a buffer  
// object that returns the next value  
// in the buffer  
v = circularBuffer.Get () ;
```

```
// Call the method associated with a  
// thermostat object that sets the  
// temperature to be maintained  
thermostat.setTemp (20) ;
```

Generalisation and inheritance

- Objects are members of classes that define attribute types and operations.
- Classes may be arranged in a class hierarchy where one class (a super-class) is a generalisation of one or more other classes (sub-classes).
- A sub-class inherits the attributes and operations from its super class and may add new methods or attributes of its own.
- Generalisation in the UML is implemented as inheritance in OO programming languages.

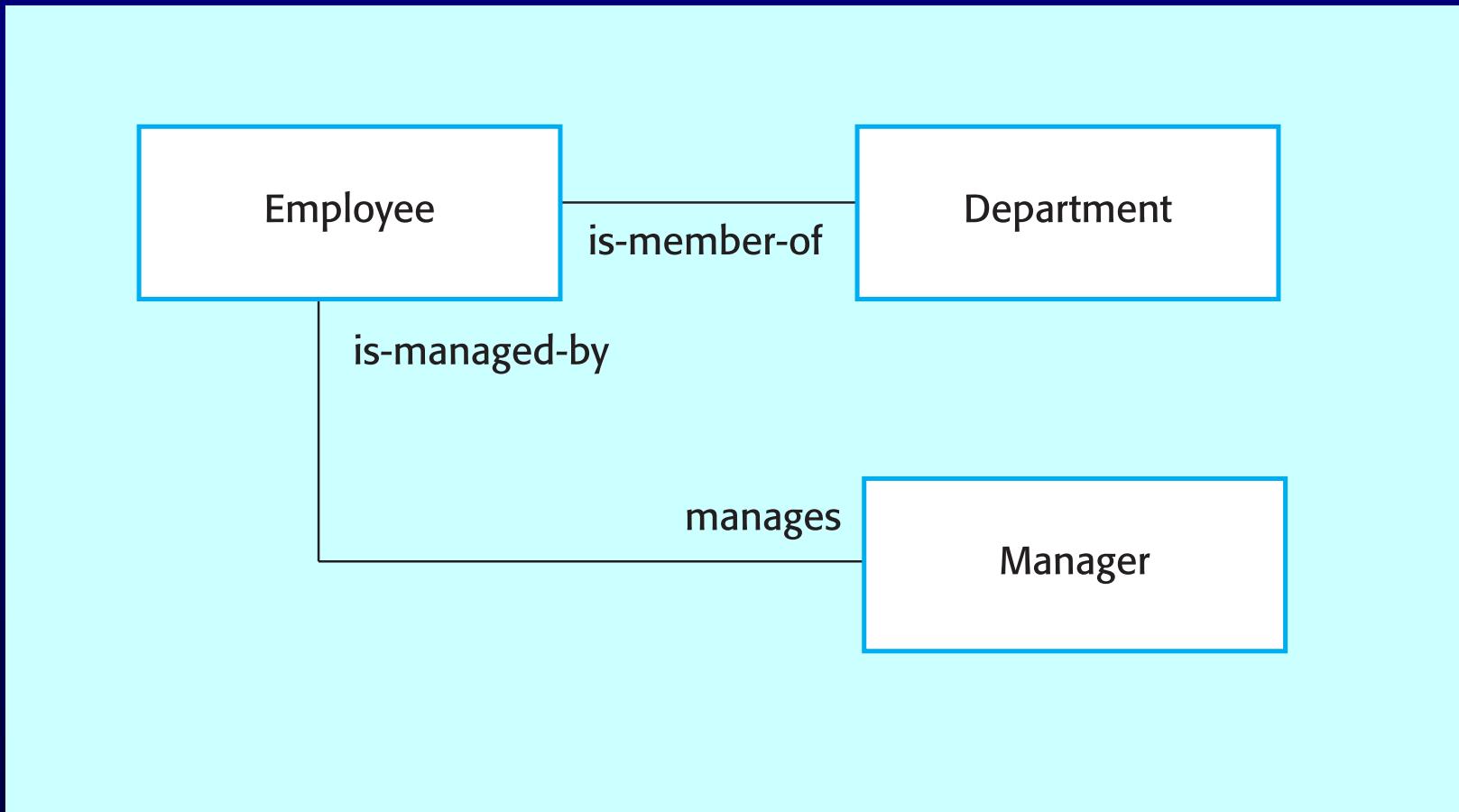
A generalisation hierarchy



UML associations

- Objects and object classes participate in relationships with other objects and object classes.
- In the UML, a generalised relationship is indicated by an association.
- Associations may be annotated with information that describes the association.
- Associations are general but may indicate that an attribute of an object is an associated object or that a method relies on an associated object.

An association model



Concurrent objects

- The nature of objects as self-contained entities make them suitable for concurrent implementation.
- The message-passing model of object communication can be implemented directly if objects are running on separate processors in a distributed system.

Servers and active objects

- Servers.
 - The object is implemented as a parallel process (server) with entry points corresponding to object operations. If no calls are made to it, the object suspends itself and waits for further requests for service.
- Active objects
 - Objects are implemented as parallel processes and the internal object state may be changed by the object itself and not simply by external calls.

Weather system description

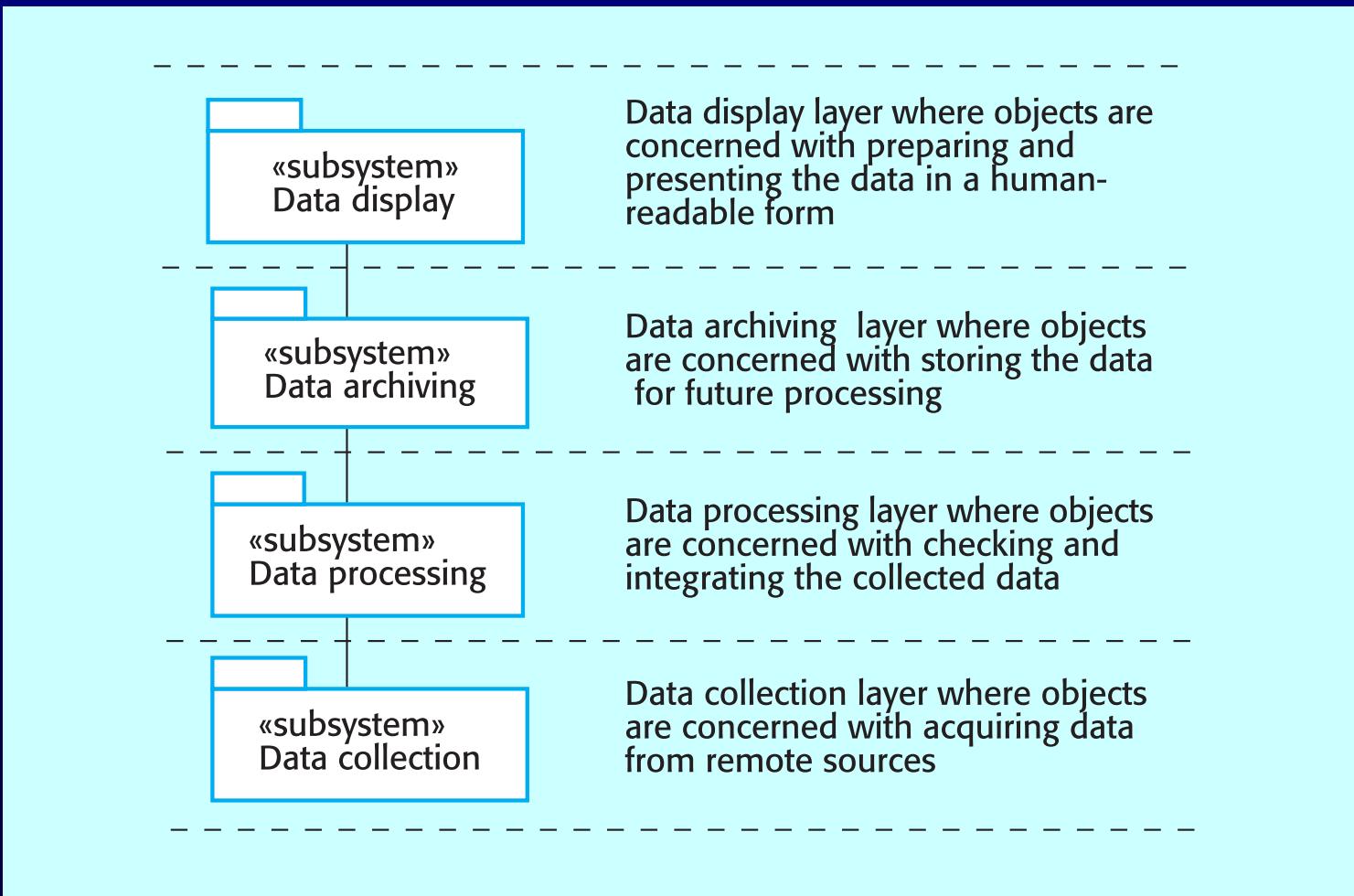
A **weather mapping system** is required to generate weather maps on a regular basis using data collected from remote, unattended weather stations and other data sources such as weather observers, balloons and satellites. Weather stations transmit their data to the area computer in response to a request from that machine.

The area computer system validates the collected data and integrates it with the data from different sources. The integrated data is archived and, using data from this archive and a digitised map database a set of local weather maps is created. Maps may be printed for distribution on a special-purpose map printer or may be displayed in a number of different formats.

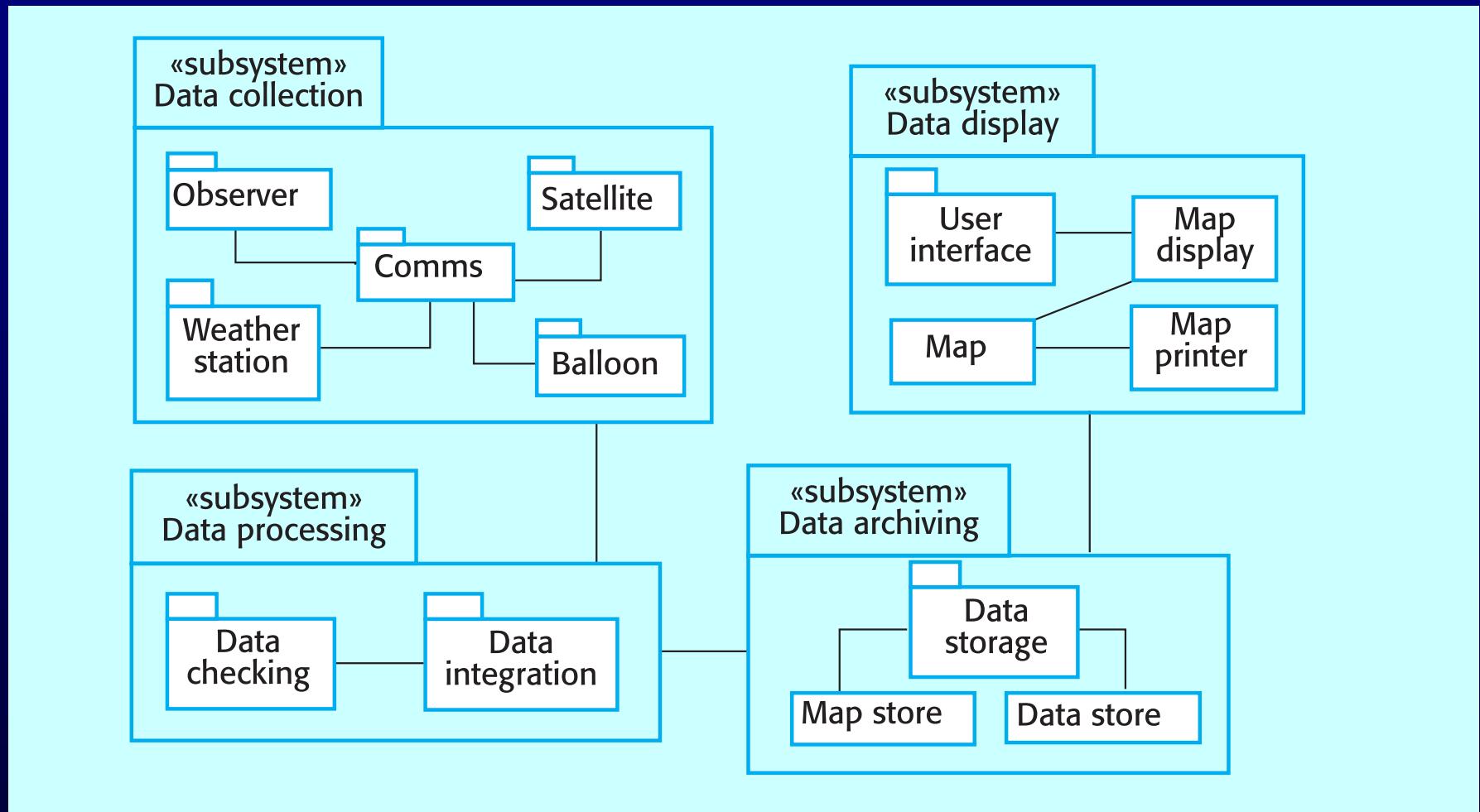
System context and models of use

- 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



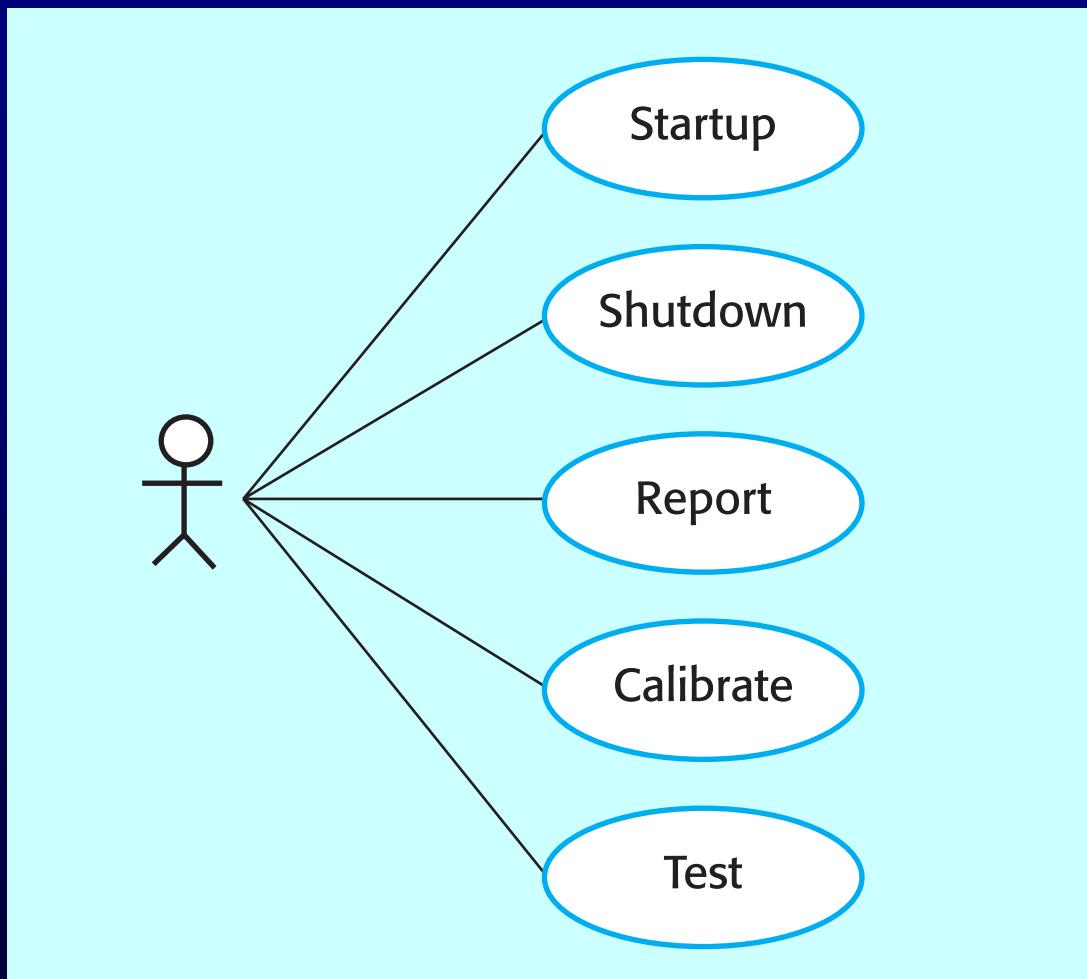
Subsystems in the weather mapping system



Use-case models

- Use-case models are used to represent each interaction with the system.
- A use-case model shows the system features as ellipses and the interacting entity as a stick figure.

Use-cases for the weather station



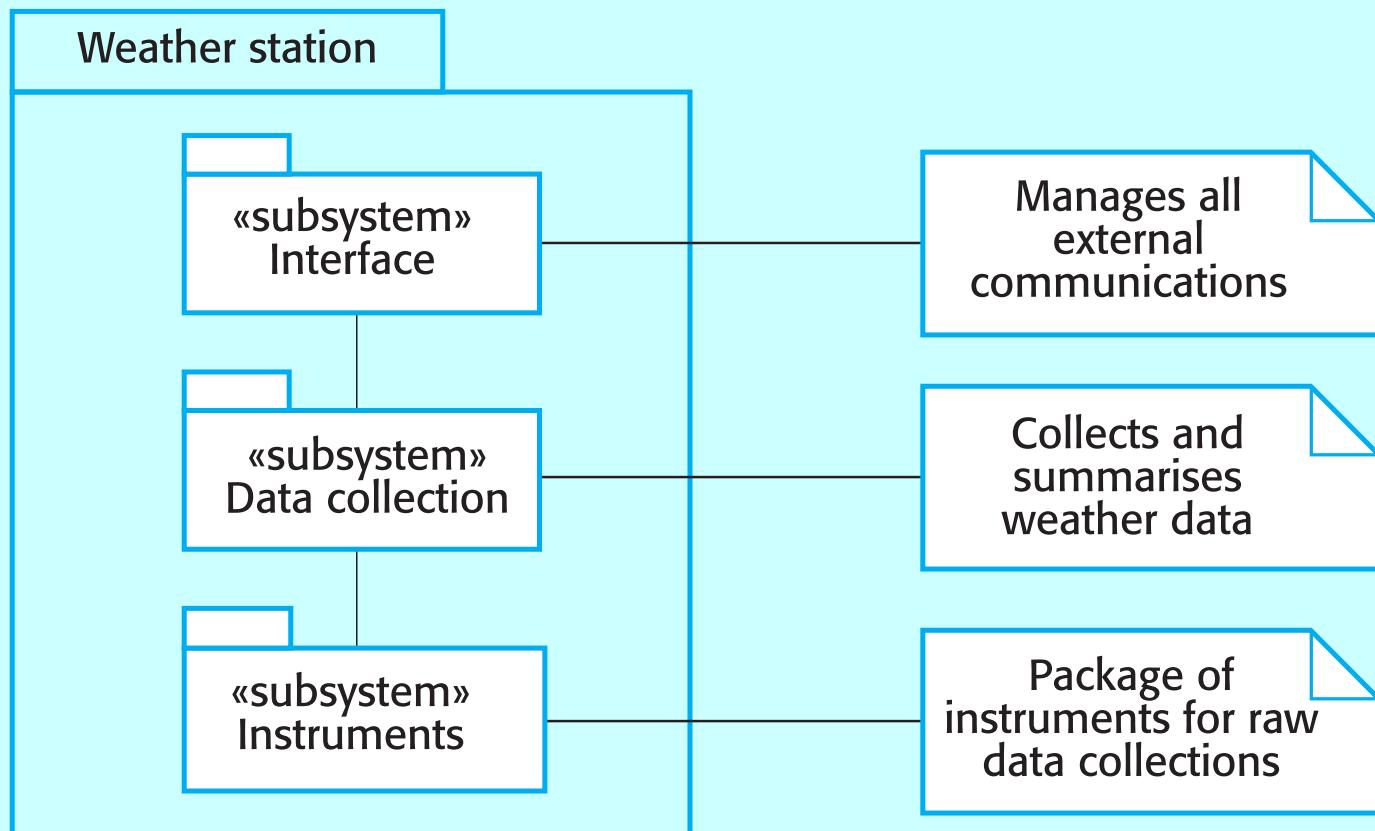
Use-case description

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.
- A layered architecture as discussed in Chapter 11 is appropriate for the weather station
 - Interface layer for handling communications;
 - Data collection layer for managing instruments;
 - Instruments layer for collecting data.
- 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.

Weather station description

A **weather station** is a package of software controlled instruments which collects data, performs some data processing and transmits this data for further processing. The instruments include air and ground thermometers, an anemometer, a wind vane, a barometer and a rain gauge. Data is collected periodically.

When a command is issued to transmit the weather data, the weather station processes and summarises the collected data. The summarised data is transmitted to the mapping computer when a request is received.

Weather station object classes

- Ground thermometer, Anemometer, Barometer
 - Application domain objects that are ‘hardware’ objects related to the instruments in the system.
- Weather station
 - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
- Weather data
 - Encapsulates the summarised data from the instruments.

Weather station object classes

WeatherStation

identifier
reportWeather ()
calibrate (instruments)
test ()
startup (instruments)
shutdown (instruments)

WeatherData

airTemperatures
groundTemperatures
windSpeeds
windDirections
pressures
rainfall

collect ()
summarise ()

Ground thermometer

temperature
test ()
calibrate ()

Anemometer

windSpeed
windDirection
test ()

Barometer

pressure
height
test ()
calibrate ()

Further objects and object refinement

- Use domain knowledge to identify more objects and operations
 - Weather stations should have a unique identifier;
 - Weather stations are remotely situated so instrument failures have to be reported automatically. Therefore attributes and operations for self-checking are required.
- Active or passive objects
 - In this case, objects are passive and collect data on request rather than autonomously. This introduces flexibility at the expense of controller processing time.

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.

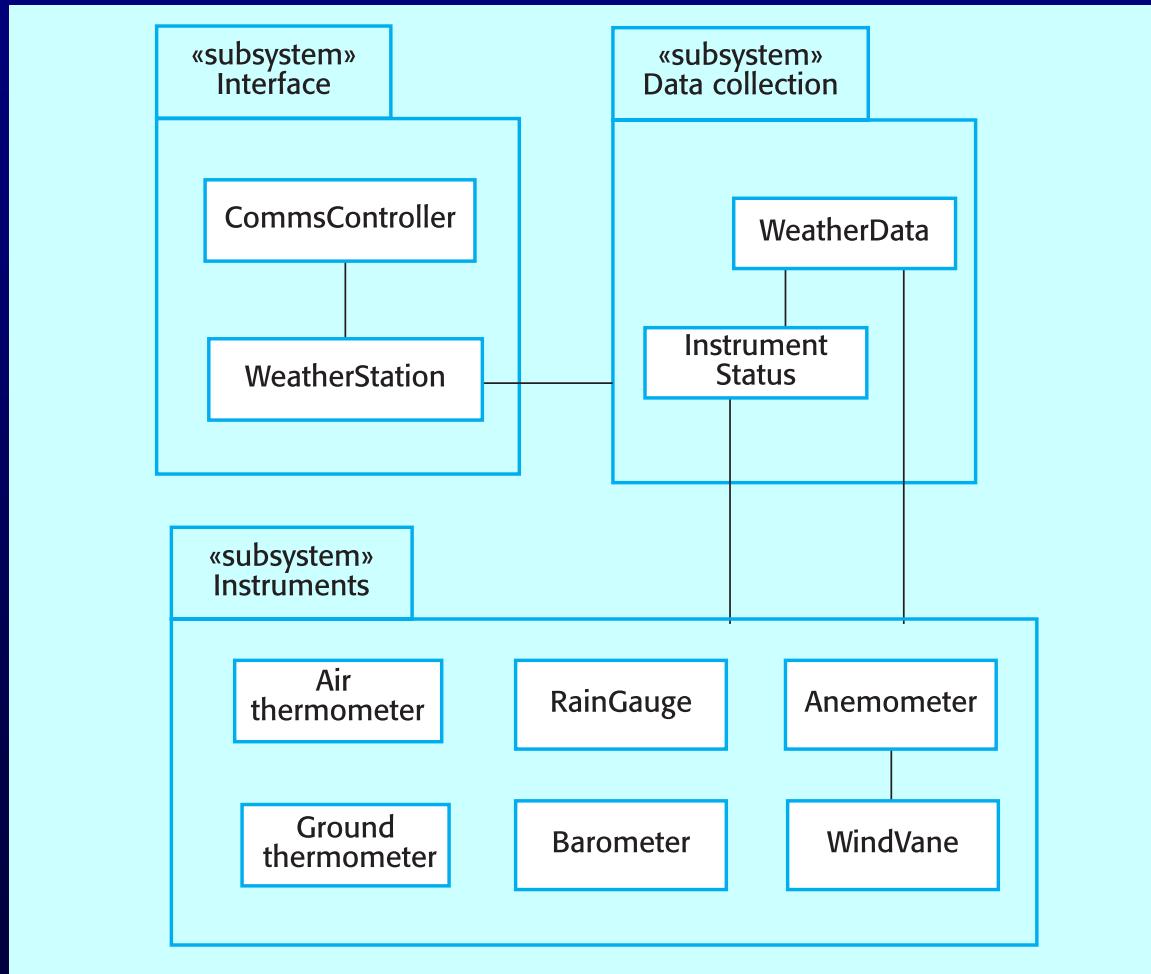
Examples of design models

- 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.

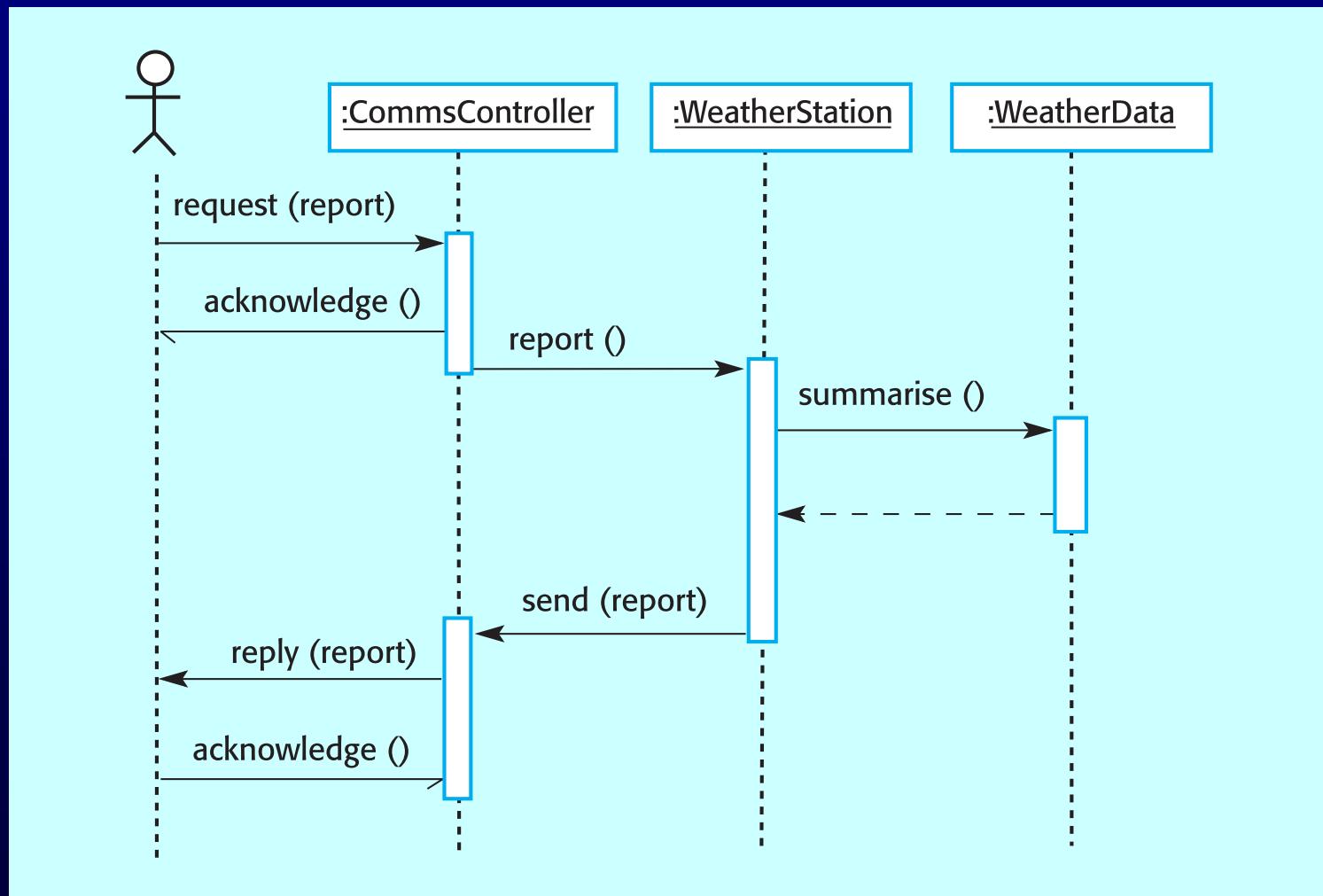
Weather station subsystems



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.

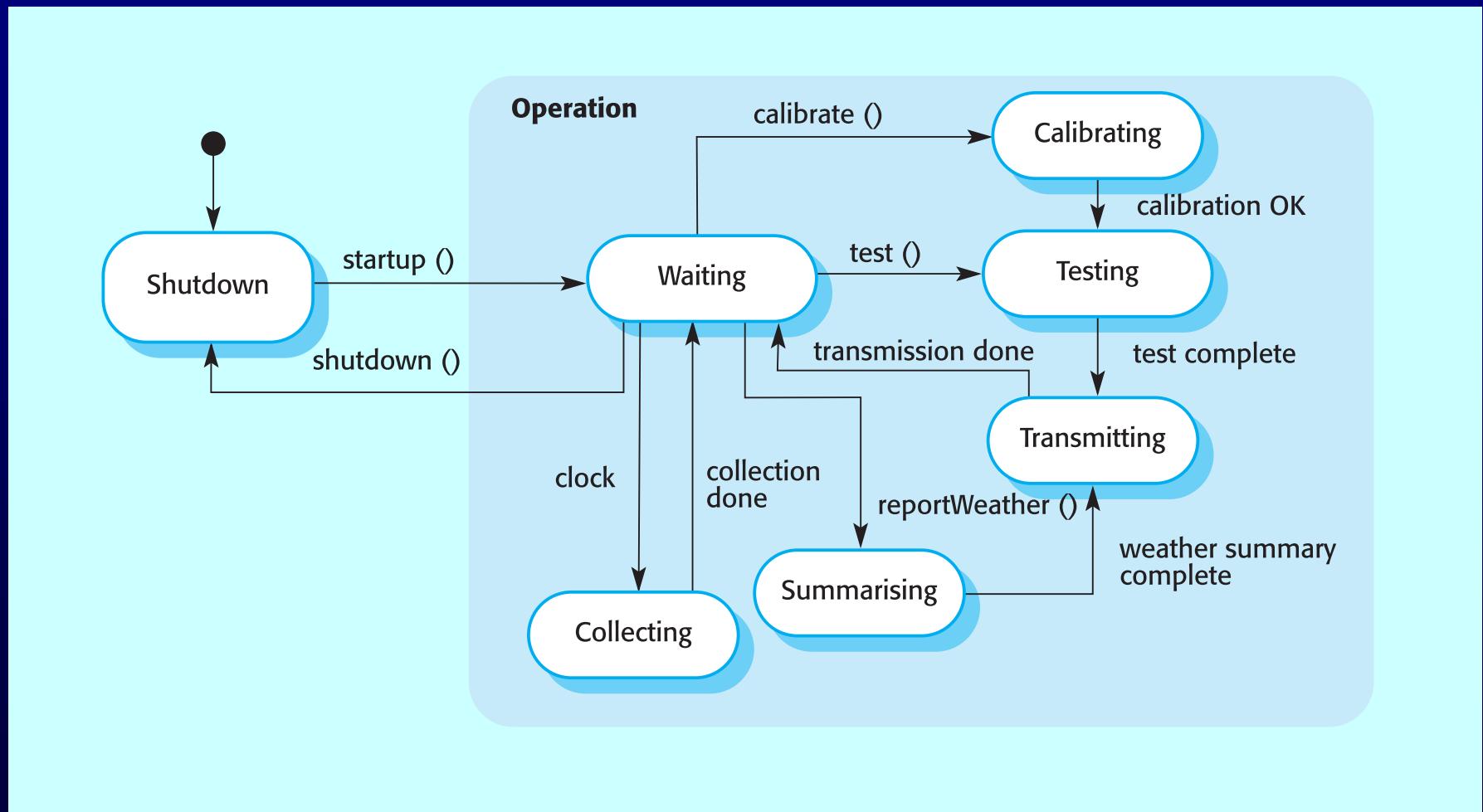
Data collection sequence



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
```

Design evolution

- Hiding information inside objects means that changes made to an object do not affect other objects in an unpredictable way.
- Assume pollution monitoring facilities are to be added to weather stations. These sample the air and compute the amount of different pollutants in the atmosphere.
- Pollution readings are transmitted with weather data.

Changes required

- Add an object class called **Air quality** as part of **WeatherStation**.
- Add an operation **reportAirQuality** to **WeatherStation**. Modify the control software to collect pollution readings.
- Add objects representing pollution monitoring instruments.

Pollution monitoring

WeatherStation

identifier
reportWeather ()
reportAirQuality ()
calibrate (instruments)
test ()
startup (instruments)
shutdown (instruments)

Air quality

NOData
smokeData
benzeneData

collect ()
summarise ()

Pollution monitoring instruments

NOmeter

SmokeMeter

BenzeneMeter

Key points

- OOD is an approach to design so that design components have their own private state and operations.
- Objects should have constructor and inspection operations. They provide services to other objects.
- Objects may be implemented sequentially or concurrently.
- The Unified Modeling Language provides different notations for defining different object models.

Key points

- A range of different models may be produced during an object-oriented design process. These include static and dynamic system models.
- Object interfaces should be defined precisely using e.g. a programming language like Java.
- Object-oriented design potentially simplifies system evolution.

Real-time Software Design

Objectives

- To explain the concept of a real-time system and why these systems are usually implemented as concurrent processes
- To describe a design process for real-time systems
- To explain the role of a real-time operating system
- To introduce generic process architectures for monitoring and control and data acquisition systems

Topics covered

- System design
- Real-time operating systems
- Monitoring and control systems
- Data acquisition systems

Real-time systems

- Systems which monitor and control their environment.
- Inevitably associated with hardware devices
 - **Sensors**: Collect data from the system environment;
 - **Actuators**: Change (in some way) the system's environment;
- Time is critical. Real-time systems **MUST** respond within specified times.

Definition

- A **real-time system** is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced.
- A **soft real-time system** is a system whose operation is degraded if results are not produced according to the specified timing requirements.
- A **hard real-time system** is a system whose operation is incorrect if results are not produced according to the timing specification.

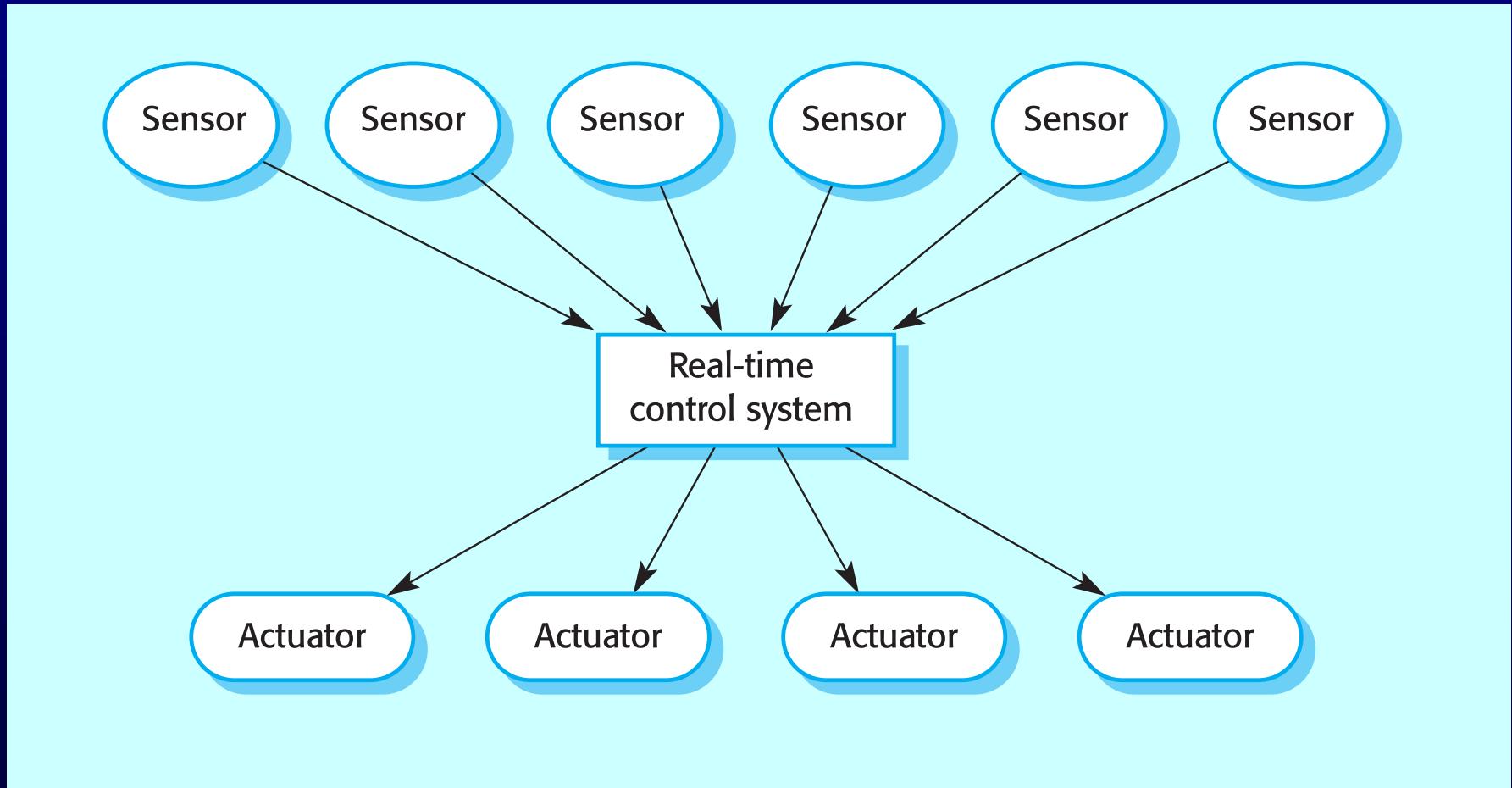
Stimulus/Response Systems

- Given a stimulus, the system must produce a response within a specified time.
- **Periodic stimuli.** Stimuli which occur at predictable time intervals
 - For example, a temperature sensor may be polled 10 times per second.
- **Aperiodic stimuli.** Stimuli which occur at unpredictable times
 - For example, a system power failure may trigger an interrupt which must be processed by the system.

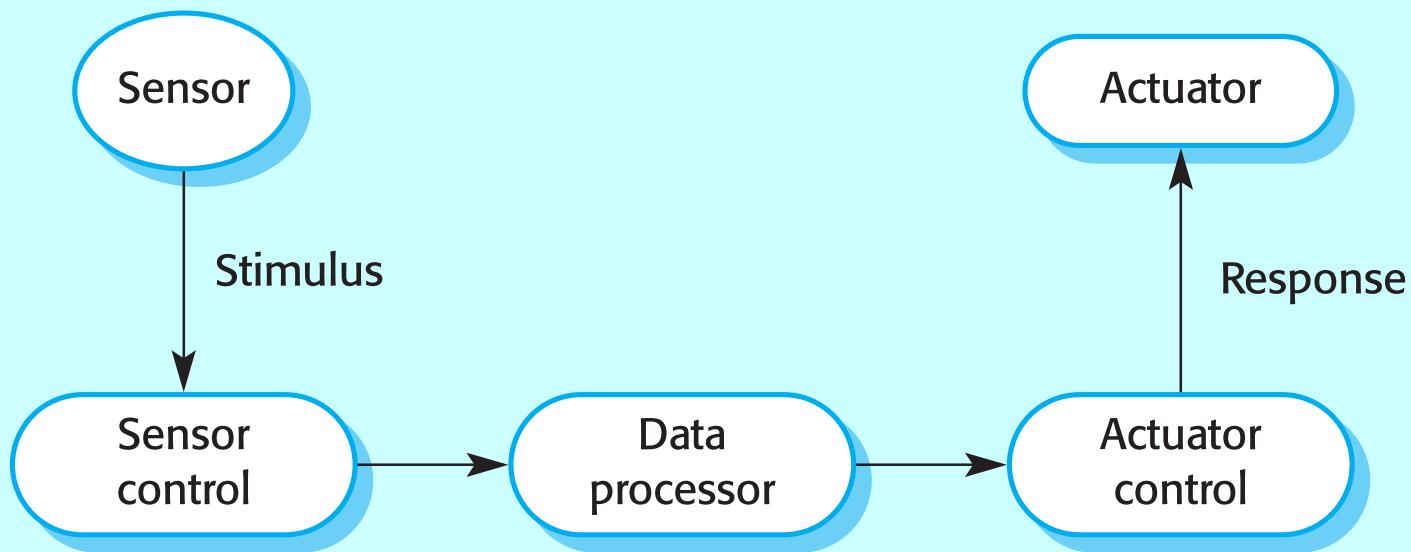
Architectural considerations

- Because of the need to respond to timing demands made by different stimuli/responses, the system architecture must allow for fast switching between stimulus handlers.
- Timing demands of different stimuli are different so a simple sequential loop is not usually adequate.
- Real-time systems are therefore usually designed as cooperating processes with a real-time executive controlling these processes.

A real-time system model



Sensor/actuator processes



System elements

- Sensor control processes
 - Collect information from sensors. May buffer information collected in response to a sensor stimulus.
- Data processor
 - Carries out processing of collected information and computes the system response.
- Actuator control processes
 - Generates control signals for the actuators.

Real-time programming

- Hard-real time systems may have to programmed in assembly language to ensure that deadlines are met.
- Languages such as C allow efficient programs to be written but do not have constructs to support concurrency or shared resource management.

Java as a real-time language

- Java supports lightweight concurrency (threads and synchronized methods) and can be used for some soft real-time systems.
- Java 2.0 is not suitable for hard RT programming but real-time versions of Java are now available that address problems such as
 - Not possible to specify thread execution time;
 - Different timing in different virtual machines;
 - Uncontrollable garbage collection;
 - Not possible to discover queue sizes for shared resources;
 - Not possible to access system hardware;
 - Not possible to do space or timing analysis.

System design

- Design both the hardware and the software associated with system. Partition functions to either hardware or software.
- Design decisions should be made on the basis on non-functional system requirements.
- Hardware delivers better performance but potentially longer development and less scope for change.

R-T systems design process

- Identify the stimuli to be processed and the required responses to these stimuli.
- For each stimulus and response, identify the timing constraints.
- Aggregate the stimulus and response processing into concurrent processes. A process may be associated with each class of stimulus and response.

R-T systems design process

- Design algorithms to process each class of stimulus and response. These must meet the given timing requirements.
- Design a scheduling system which will ensure that processes are started in time to meet their deadlines.
- Integrate using a real-time operating system.

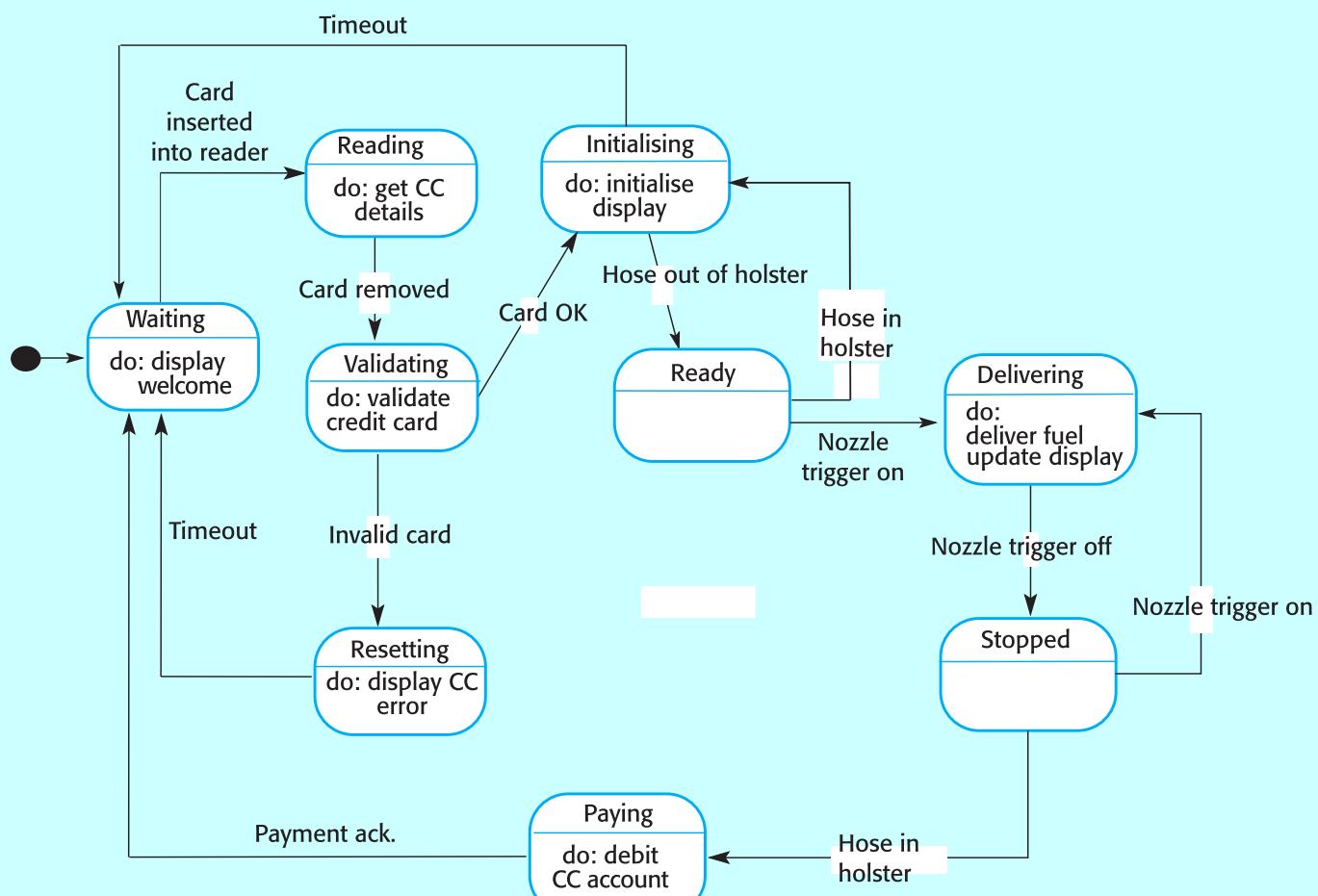
Timing constraints

- May require extensive simulation and experiment to ensure that these are met by the system.
- May mean that certain design strategies such as object-oriented design cannot be used because of the additional overhead involved.
- May mean that low-level programming language features have to be used for performance reasons.

Real-time system modelling

- The effect of a stimulus in a real-time system may trigger a transition from one state to another.
- Finite state machines can be used for modelling real-time systems.
- However, FSM models lack structure. Even simple systems can have a complex model.
- The UML includes notations for defining state machine models
- See Chapter 8 for further examples of state machine models.

Petrol pump state model



Real-time operating systems

- Real-time operating systems are specialised operating systems which manage the processes in the RTS.
- Responsible for process management and resource (processor and memory) allocation.
- May be based on a standard kernel which is used unchanged or modified for a particular application.
- Do not normally include facilities such as file management.

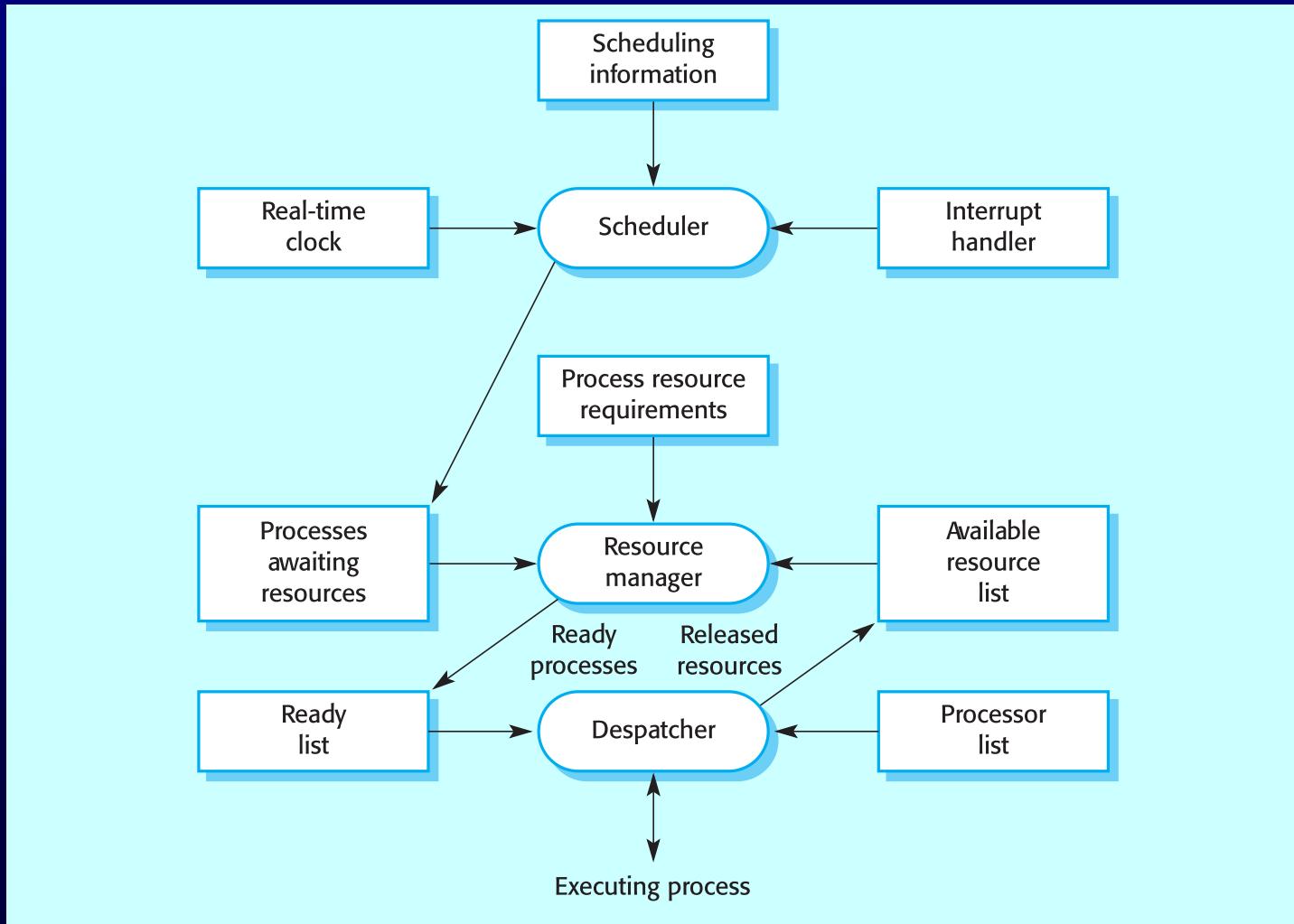
Operating system components

- Real-time clock
 - Provides information for process scheduling.
- Interrupt handler
 - Manages aperiodic requests for service.
- Scheduler
 - Chooses the next process to be run.
- Resource manager
 - Allocates memory and processor resources.
- Dispatcher
 - Starts process execution.

Non-stop system components

- Configuration manager
 - Responsible for the dynamic reconfiguration of the system software and hardware. Hardware modules may be replaced and software upgraded without stopping the systems.
- Fault manager
 - Responsible for detecting software and hardware faults and taking appropriate actions (e.g. switching to backup disks) to ensure that the system continues in operation.

Real-time OS components



Process priority

- The processing of some types of stimuli must sometimes take priority.
- Interrupt level priority. Highest priority which is allocated to processes requiring a very fast response.
- Clock level priority. Allocated to periodic processes.
- Within these, further levels of priority may be assigned.

Interrupt servicing

- Control is transferred automatically to a pre-determined memory location.
- This location contains an instruction to jump to an interrupt service routine.
- Further interrupts are disabled, the interrupt serviced and control returned to the interrupted process.
- Interrupt service routines **MUST** be short, simple and fast.

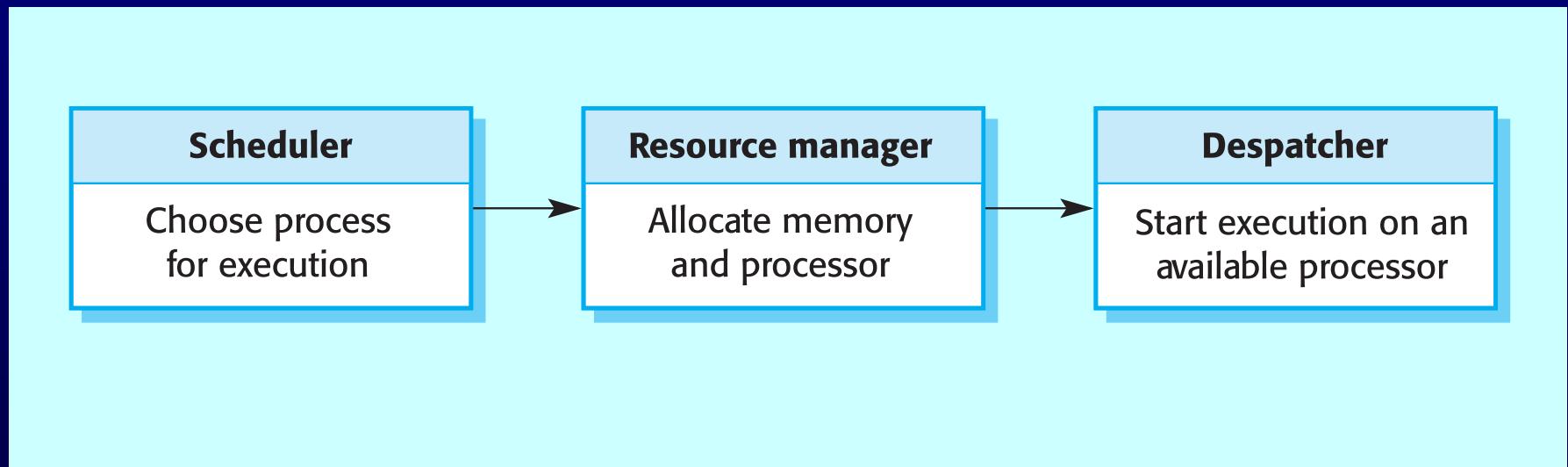
Periodic process servicing

- In most real-time systems, there will be several classes of periodic process, each with different periods (the time between executions), execution times and deadlines (the time by which processing must be completed).
- The real-time clock ticks periodically and each tick causes an interrupt which schedules the process manager for periodic processes.
- The process manager selects a process which is ready for execution.

Process management

- Concerned with managing the set of concurrent processes.
- Periodic processes are executed at pre-specified time intervals.
- The RTOS uses the real-time clock to determine when to execute a process taking into account:
 - Process period - time between executions.
 - Process deadline - the time by which processing must be complete.

RTE process management



Process switching

- The scheduler chooses the next process to be executed by the processor. This depends on a scheduling strategy which may take the process priority into account.
- The resource manager allocates memory and a processor for the process to be executed.
- The dispatcher takes the process from ready list, loads it onto a processor and starts execution.

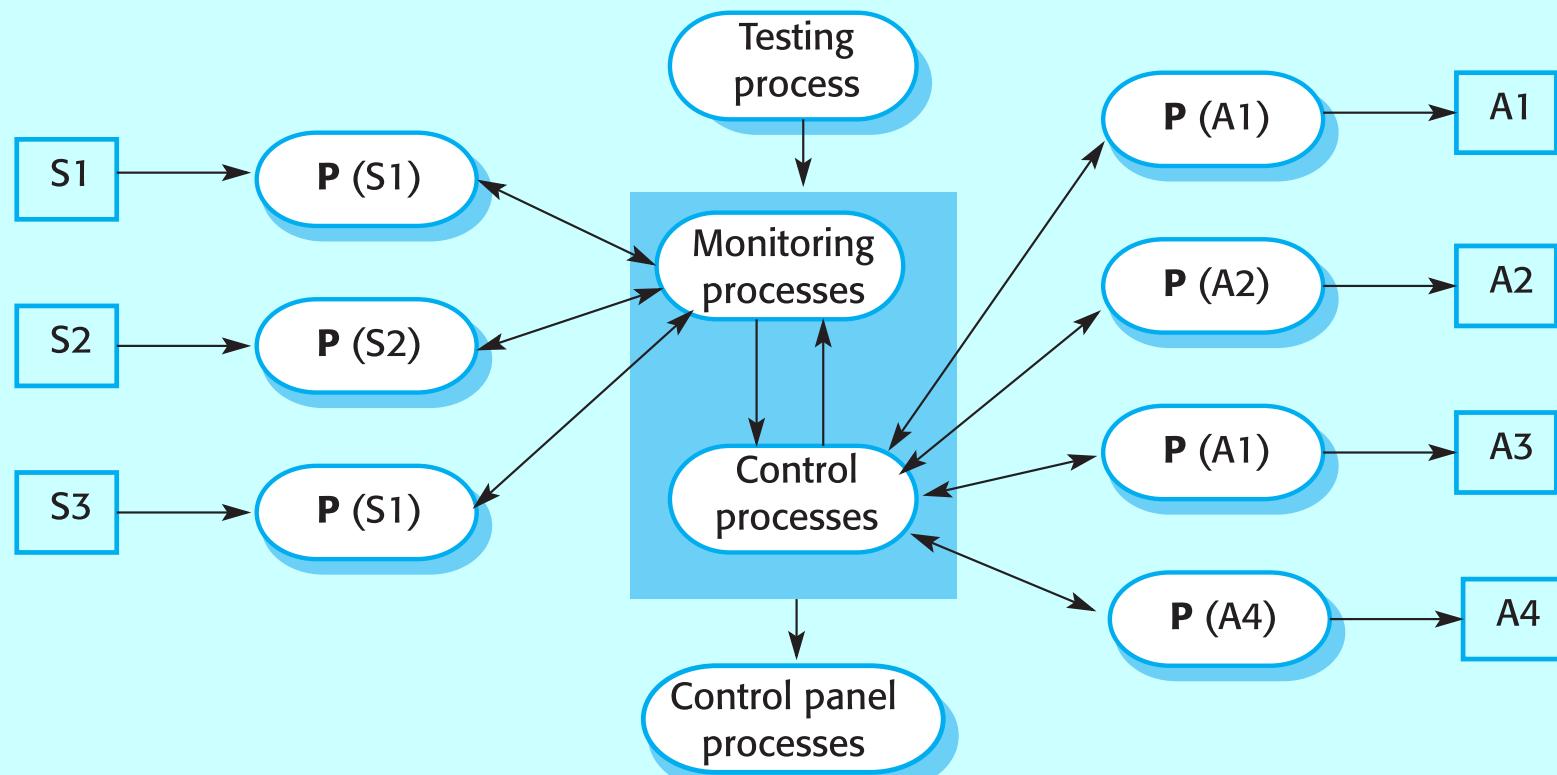
Scheduling strategies

- Non pre-emptive scheduling
 - Once a process has been scheduled for execution, it runs to completion or until it is blocked for some reason (e.g. waiting for I/O).
- Pre-emptive scheduling
 - The execution of an executing processes may be stopped if a higher priority process requires service.
- Scheduling algorithms
 - Round-robin;
 - Rate monotonic;
 - Shortest deadline first.

Monitoring and control systems

- Important class of real-time systems.
- Continuously check sensors and take actions depending on sensor values.
- Monitoring systems examine sensors and report their results.
- Control systems take sensor values and control hardware actuators.

Generic architecture



Burglar alarm system

- A system is required to monitor sensors on doors and windows to detect the presence of intruders in a building.
- When a sensor indicates a break-in, the system switches on lights around the area and calls police automatically.
- The system should include provision for operation without a mains power supply.

Burglar alarm system

- Sensors
 - Movement detectors, window sensors, door sensors;
 - 50 window sensors, 30 door sensors and 200 movement detectors;
 - Voltage drop sensor.
- Actions
 - When an intruder is detected, police are called automatically;
 - Lights are switched on in rooms with active sensors;
 - An audible alarm is switched on;
 - The system switches automatically to backup power when a voltage drop is detected.

The R-T system design process

- Identify stimuli and associated responses.
- Define the timing constraints associated with each stimulus and response.
- Allocate system functions to concurrent processes.
- Design algorithms for stimulus processing and response generation.
- Design a scheduling system which ensures that processes will always be scheduled to meet their deadlines.

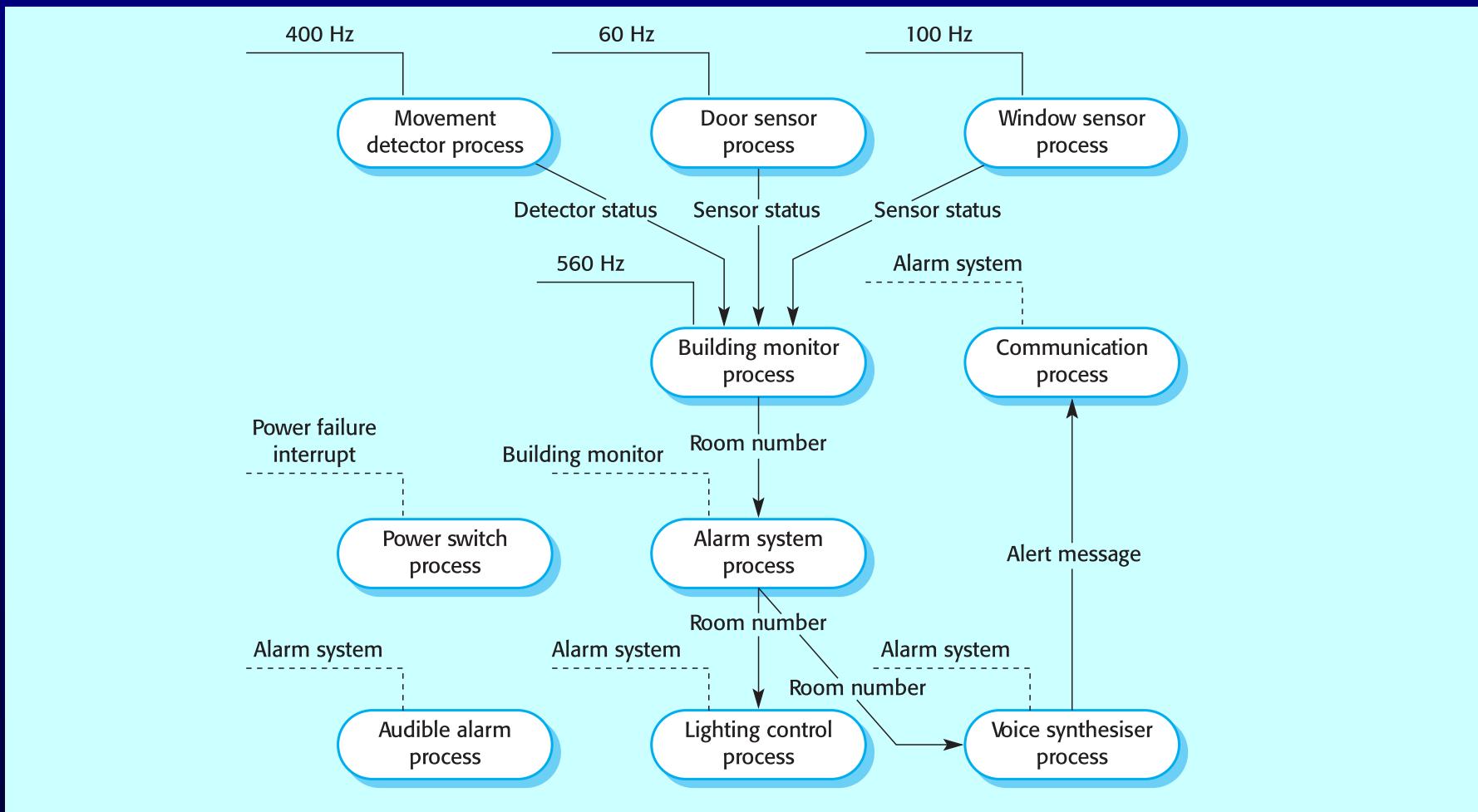
Stimuli to be processed

- Power failure
 - Generated aperiodically by a circuit monitor. When received, the system must switch to backup power within 50 ms.
- Intruder alarm
 - Stimulus generated by system sensors. Response is to call the police, switch on building lights and the audible alarm.

Timing requirements

Stimulus/Response	Timing requirements
Power fail interrupt	The switch to backup power must be completed within a deadline of 50 ms.
Door alarm	Each door alarm should be polled twice per second.
Window alarm	Each window alarm should be polled twice per second.
Movement detector	Each movement detector should be polled twice per second.
Audible alarm	The audible alarm should be switched on within 1/2 second of an alarm being raised by a sensor.
Lights switch	The lights should be switched on within 1/2 second of an alarm being raised by a sensor.
Communications	The call to the police should be started within 2 seconds of an alarm being raised by a sensor.
Voice synthesiser	A synthesised message should be available within 4 seconds of an alarm being raised by a sensor.

Burglar alarm system processes



Building_monitor process 1

```
class BuildingMonitor extends Thread {  
  
    BuildingSensor win, door, move ;  
  
    Siren      siren = new Siren () ;  
    Lights     lights = new Lights () ;  
    Synthesizer synthesizer = new Synthesizer () ;  
    DoorSensors doors = new DoorSensors (30) ;  
    WindowSensors windows = new WindowSensors (50) ;  
    MovementSensors movements = new MovementSensors (200) ;  
    PowerMonitor pm = new PowerMonitor () ;  
  
    BuildingMonitor()  
{  
        // initialise all the sensors and start the processes  
        siren.start () ; lights.start () ;  
        synthesizer.start () ; windows.start () ;  
        doors.start () ; movements.start () ; pm.start () ;  
    }  
}
```

Building monitor process 2

```
public void run ()  
{  
    int room = 0 ;  
    while (true)  
    {  
        // poll the movement sensors at least twice per second (400 Hz)  
        move = movements.getVal () ;  
        // poll the window sensors at least twice/second (100 Hz)  
        win = windows.getVal () ;  
        // poll the door sensors at least twice per second (60 Hz)  
        door = doors.getVal () ;  
        if (move.sensorVal == 1 | door.sensorVal == 1 | win.sensorVal == 1)  
        {  
            // a sensor has indicated an intruder  
            if (move.sensorVal == 1)      room = move.room ;  
            if (door.sensorVal == 1)     room = door.room ;  
            if (win.sensorVal == 1 )       room = win.room ;  
  
            lights.on (room) ; siren.on () ; synthesizer.on (room) ;  
            break ;  
        }  
    }  
}
```

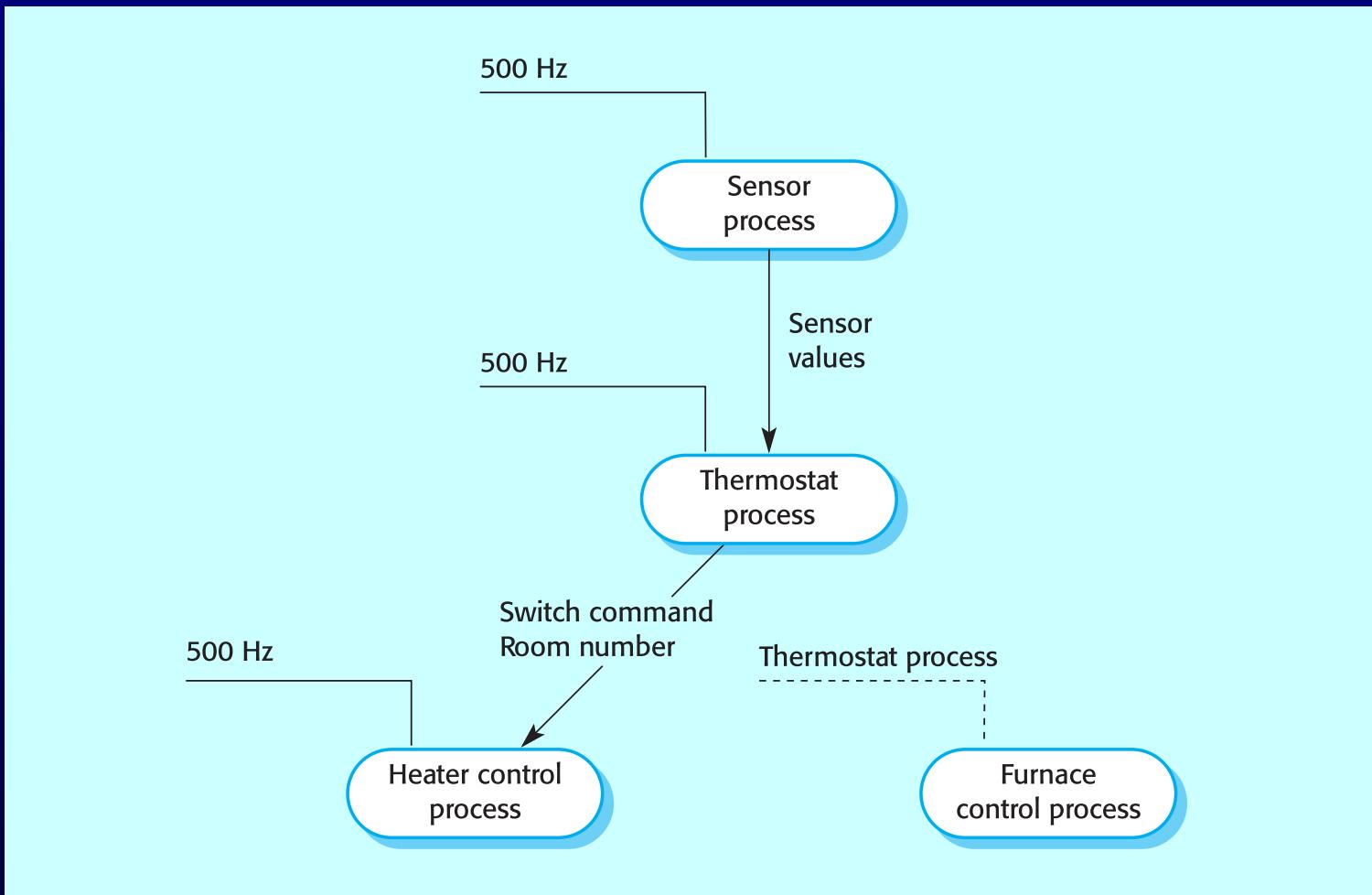
Building_monitor process 3

```
    lights.shutdown () ; siren.shutdown () ; synthesizer.shutdown () ;  
    windows.shutdown () ; doors.shutdown () ; movements.shutdown () ;  
  
} // run  
} //BuildingMonitor
```

Control systems

- A burglar alarm system is primarily a monitoring system. It collects data from sensors but no real-time actuator control.
- Control systems are similar but, in response to sensor values, the system sends control signals to actuators.
- An example of a monitoring and control system is a system that monitors temperature and switches heaters on and off.

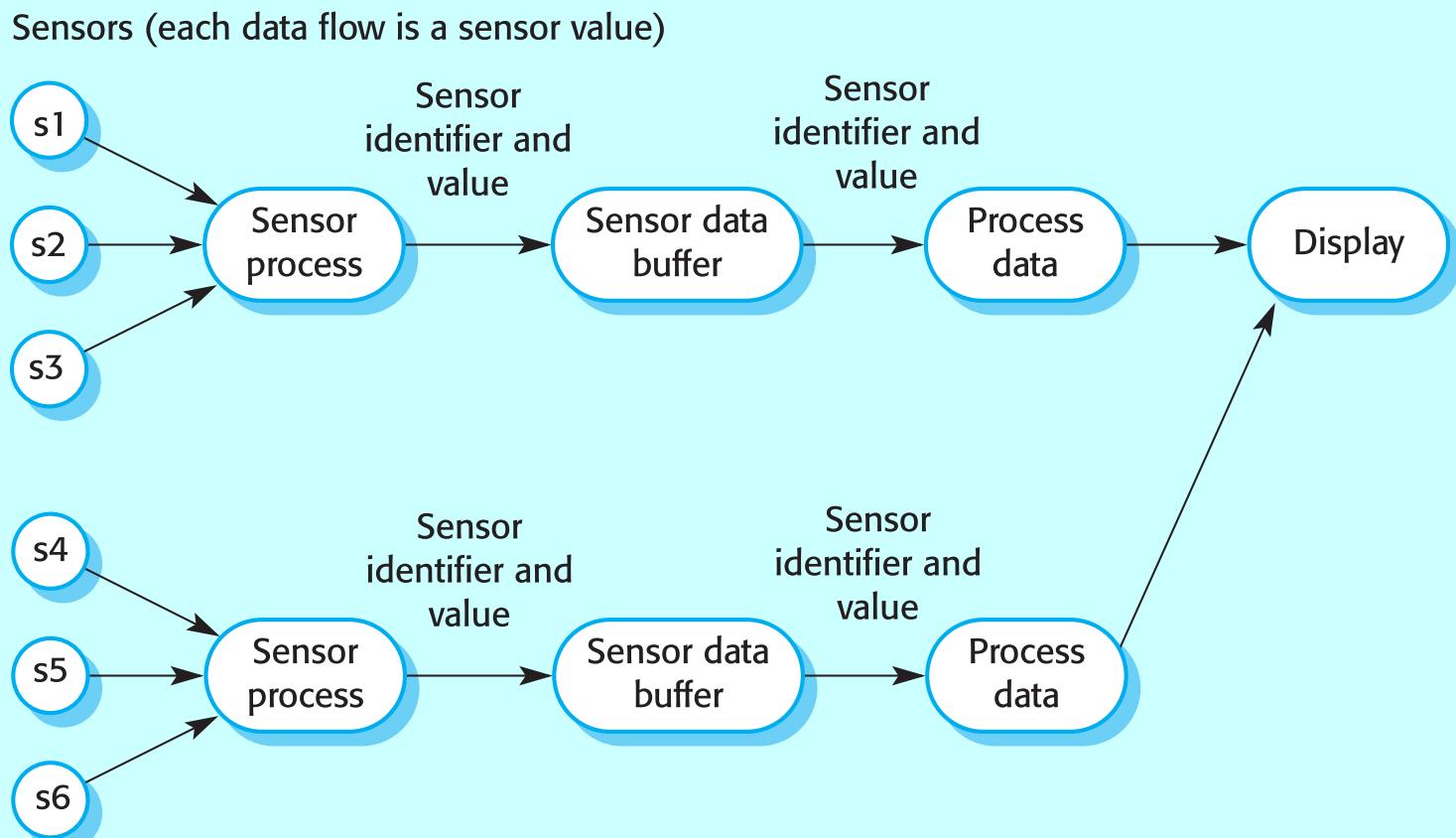
A temperature control system



Data acquisition systems

- Collect data from sensors for subsequent processing and analysis.
- Data collection processes and processing processes may have different periods and deadlines.
- Data collection may be faster than processing e.g. collecting information about an explosion.
- Circular or ring buffers are a mechanism for smoothing speed differences.

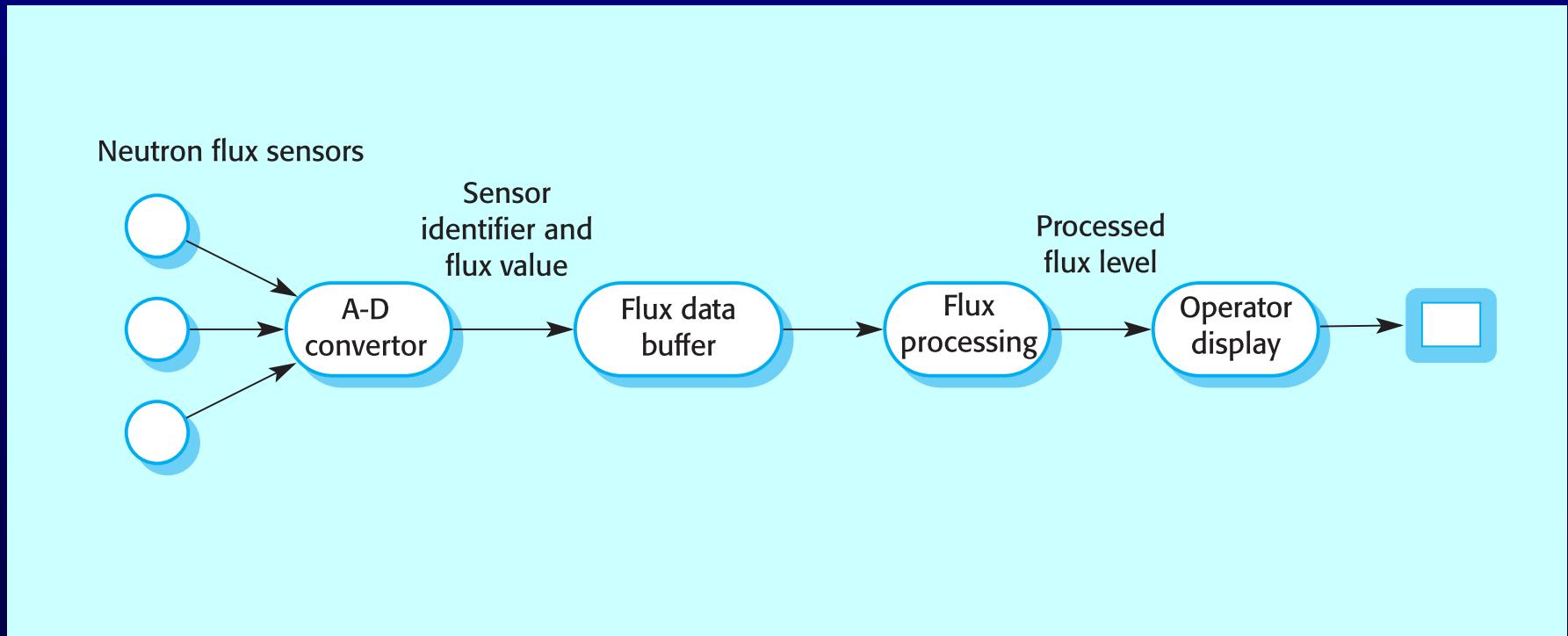
Data acquisition architecture



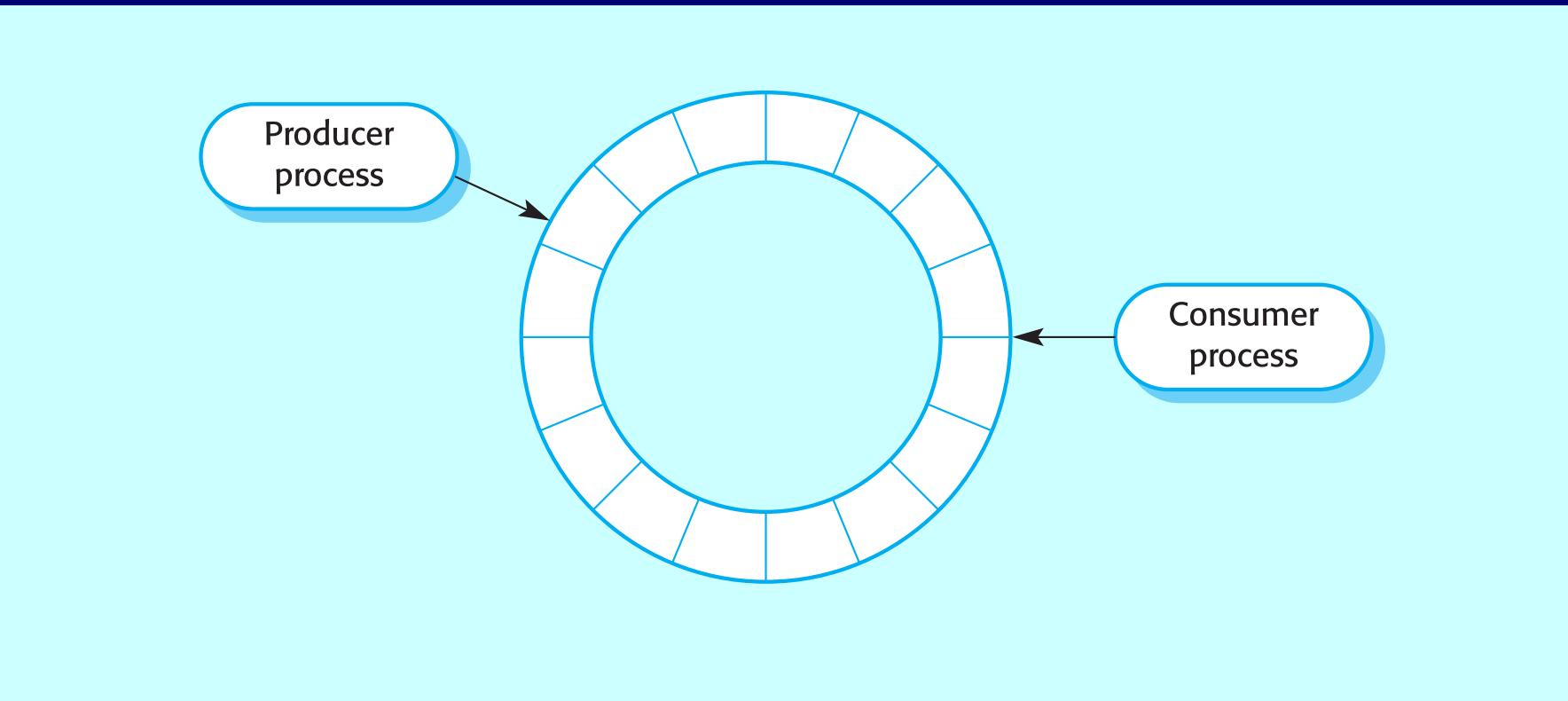
Reactor data collection

- A system collects data from a set of sensors monitoring the neutron flux from a nuclear reactor.
- Flux data is placed in a ring buffer for later processing.
- The ring buffer is itself implemented as a concurrent process so that the collection and processing processes may be synchronized.

Reactor flux monitoring



A ring buffer



Mutual exclusion

- Producer processes collect data and add it to the buffer. Consumer processes take data from the buffer and make elements available.
- Producer and consumer processes must be mutually excluded from accessing the same element.
- The buffer must stop producer processes adding information to a full buffer and consumer processes trying to take information from an empty buffer.

Ring buffer implementation 1

```
class CircularBuffer
{
    int bufsize ;
    SensorRecord [] store ;
    int numberOfEntries = 0 ;
    int front = 0, back = 0 ;

    CircularBuffer (int n) {
        bufsize = n ;
        store = new SensorRecord [bufsize] ;
    } // CircularBuffer
```

Ring buffer implementation 2

```
synchronized void put (SensorRecord rec )
    throws InterruptedException
{
    if ( numberOfEntries == bufsize)
        wait () ;
    store [back] = new SensorRecord (rec.sensorId, rec
    back = back + 1 ;
    if (back == bufsize)
        back = 0 ;
    numberOfEntries = numberOfEntries + 1 ;
    notify () ;
} // put
```

Ring buffer implementation 3

```
synchronized SensorRecord get () throws InterruptedException
{
    SensorRecord result = new SensorRecord (-1, -1) ;
    if (numberOfEntries == 0)
        wait () ;
    result = store [front] ;
    front = front + 1 ;
    if (front == bufsize)
        front = 0 ;
    numberOfEntries = numberOfEntries - 1 ;
    notify () ;
    return result ;
} // get
} // CircularBuffer
```

Key points

- Real-time system correctness depends not just on what the system does but also on how fast it reacts.
- A general RT system model involves associating processes with sensors and actuators.
- Real-time systems architectures are usually designed as a number of concurrent processes.

Key points

- Real-time operating systems are responsible for process and resource management.
- Monitoring and control systems poll sensors and send control signal to actuators.
- Data acquisition systems are usually organised according to a producer consumer model.

User interface design

Objectives

- To suggest some general design principles for user interface design
- To explain different interaction styles and their use
- To explain when to use graphical and textual information presentation
- To explain the principal activities in the user interface design process
- To introduce usability attributes and approaches to system evaluation

Topics covered

- Design issues
- The user interface design process
- User analysis
- User interface prototyping
- Interface evaluation

The user interface

- User interfaces should be designed to match the skills, experience and expectations of its anticipated users.
- System users often judge a system by its interface rather than its functionality.
- A poorly designed interface can cause a user to make catastrophic errors.
- Poor user interface design is the reason why so many software systems are never used.

Human factors in interface design

- Limited short-term memory
 - People can instantaneously remember about 7 items of information. If you present more than this, they are more liable to make mistakes.
- People make mistakes
 - When people make mistakes and systems go wrong, inappropriate alarms and messages can increase stress and hence the likelihood of more mistakes.
- People are different
 - People have a wide range of physical capabilities. Designers should not just design for their own capabilities.
- People have different interaction preferences
 - Some like pictures, some like text.

UI design principles

- UI design must take account of the needs, experience and capabilities of the system users.
- Designers should be aware of people's physical and mental limitations (e.g. limited short-term memory) and should recognise that people make mistakes.
- UI design principles underlie interface designs although not all principles are applicable to all designs.

User interface design principles

Principle	Description
User familiarity	The interface should use terms and concepts which are drawn from the experience of the people who will make most use of the system.
Consistency	The interface should be consistent in that, wherever possible, comparable operations should be activated in the same way.
Minimal surprise	Users should never be surprised by the behaviour of a system.
Recoverability	The interface should include mechanisms to allow users to recover from errors.
User guidance	The interface should provide meaningful feedback when errors occur and provide context-sensitive user help facilities.
User diversity	The interface should provide appropriate interaction facilities for different types of system user.

Design principles

- User familiarity
 - The interface should be based on user-oriented terms and concepts rather than computer concepts. For example, an office system should use concepts such as letters, documents, folders etc. rather than directories, file identifiers, etc.
- Consistency
 - The system should display an appropriate level of consistency. Commands and menus should have the same format, command punctuation should be similar, etc.
- Minimal surprise
 - If a command operates in a known way, the user should be able to predict the operation of comparable commands

Design principles

- Recoverability
 - The system should provide some resilience to user errors and allow the user to recover from errors. This might include an undo facility, confirmation of destructive actions, 'soft' deletes, etc.
- User guidance
 - Some user guidance such as help systems, on-line manuals, etc. should be supplied
- User diversity
 - Interaction facilities for different types of user should be supported. For example, some users have seeing difficulties and so larger text should be available

Design issues in UIs

- Two problems must be addressed in interactive systems design
 - How should information from the user be provided to the computer system?
 - How should information from the computer system be presented to the user?
- User interaction and information presentation may be integrated through a coherent framework such as a user interface metaphor.

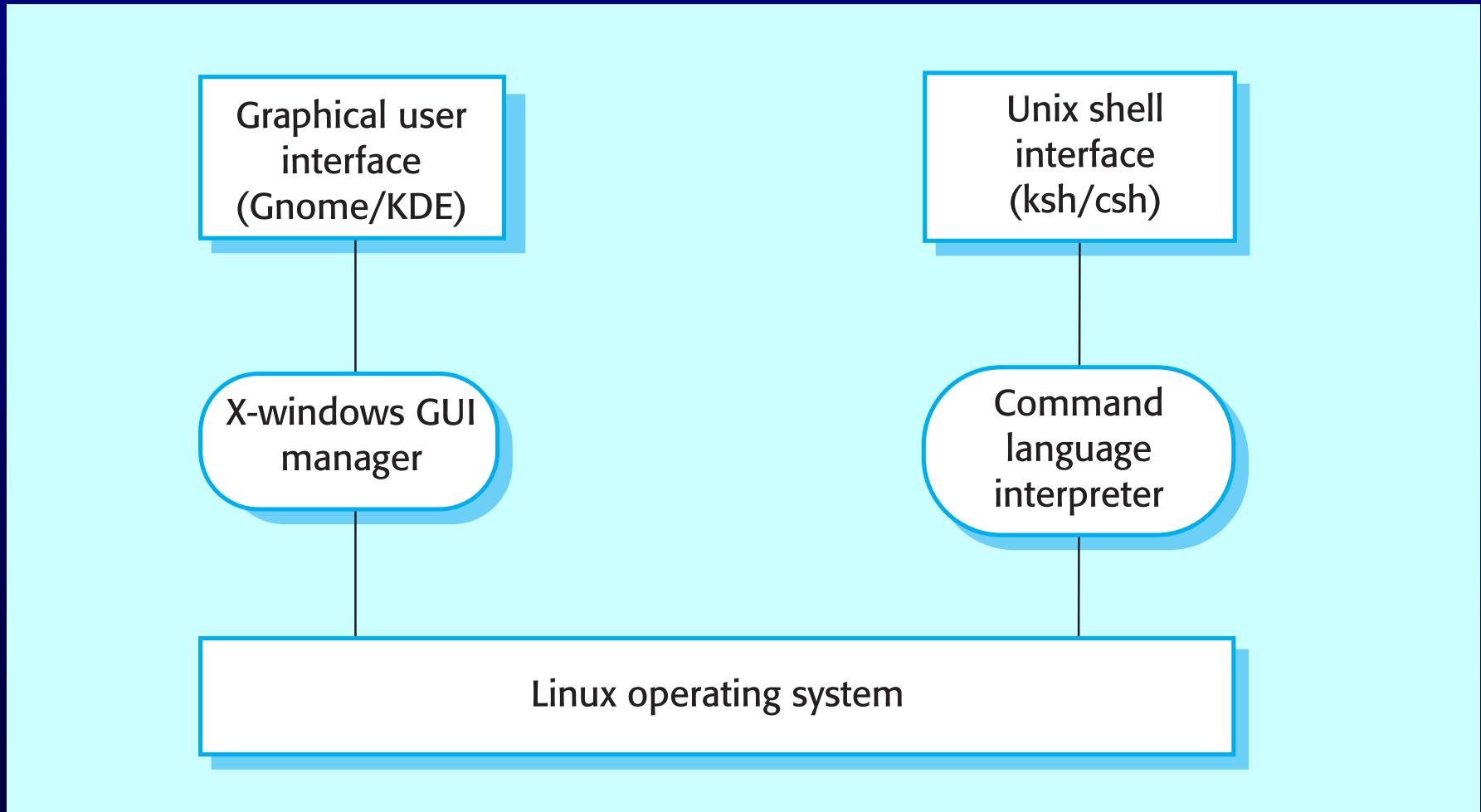
Interaction styles

- Direct manipulation
- Menu selection
- Form fill-in
- Command language
- Natural language

Interaction styles

Interaction style	Main advantages	Main disadvantages	Application examples
Direct manipulation	Fast and intuitive interaction Easy to learn	May be hard to implement. Only suitable where there is a visual metaphor for tasks and objects.	Video games CAD systems
Menu selection	Avoids user error Little typing required	Slow for experienced users. Can become complex if many menu options.	Most general-purpose systems
Form fill-in	Simple data entry Easy to learn Checkable	Takes up a lot of screen space. Causes problems where user options do not match the form fields.	Stock control, Personal loan processing
Command language	Powerful and flexible	Hard to learn. Poor error management.	Operating systems, Command and control systems
Natural language	Accessible to casual users Easily extended	Requires more typing. Natural language understanding systems are unreliable.	Information retrieval systems

Multiple user interfaces



LIBSYS interaction

- Document search
 - Users need to be able to use the search facilities to find the documents that they need.
- Document request
 - Users request that a document be delivered to their machine or to a server for printing.

Web-based interfaces

- Many web-based systems have interfaces based on web forms.
- Form field can be menus, free text input, radio buttons, etc.
- In the LIBSYS example, users make a choice of where to search from a menu and type the search phrase into a free text field.

LIBSYS search form

LIBSYS: Search

Choose collection

All



Keyword or phrase

Search using

Title



Adjacent words

Yes

No

Search

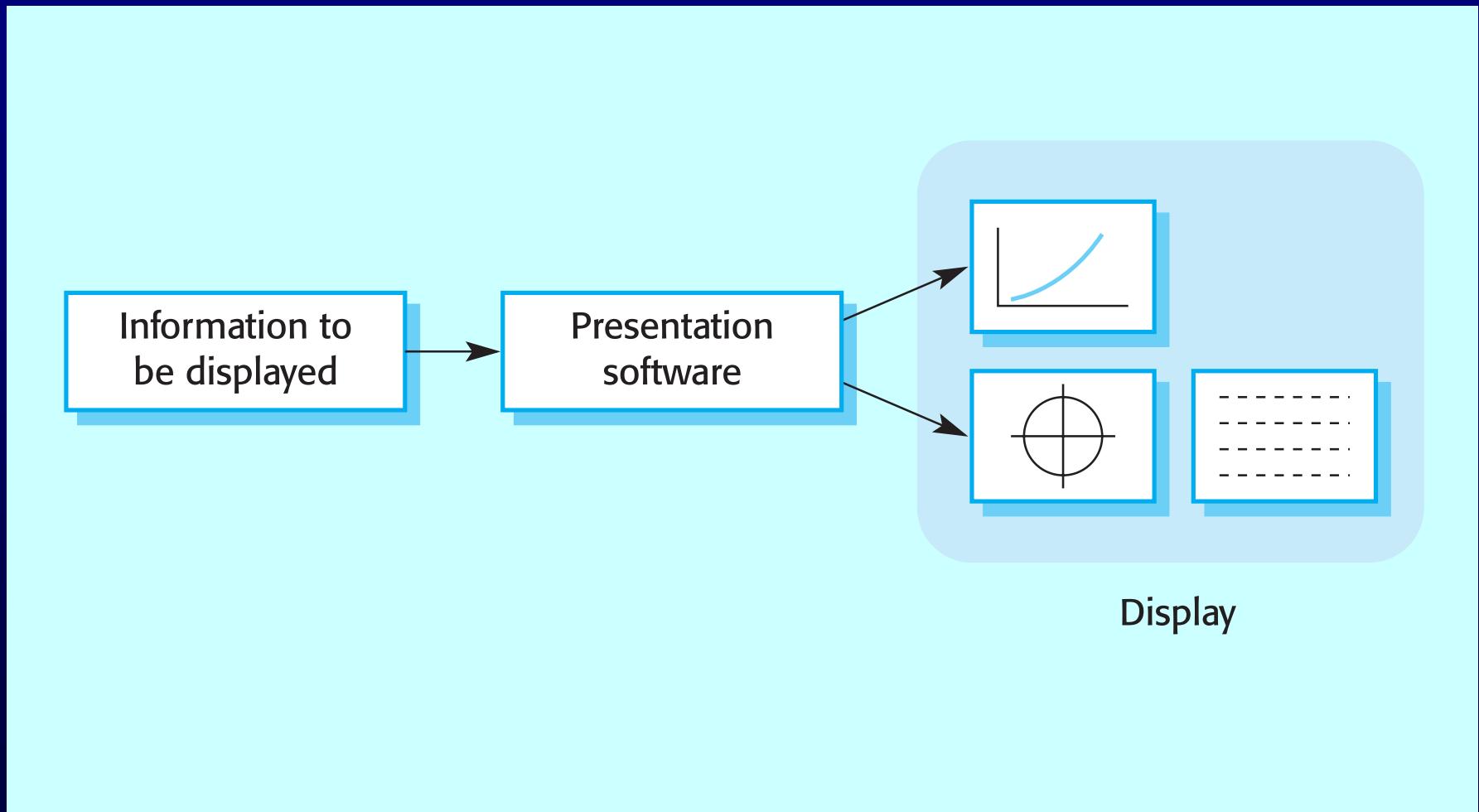
Reset

Cancel

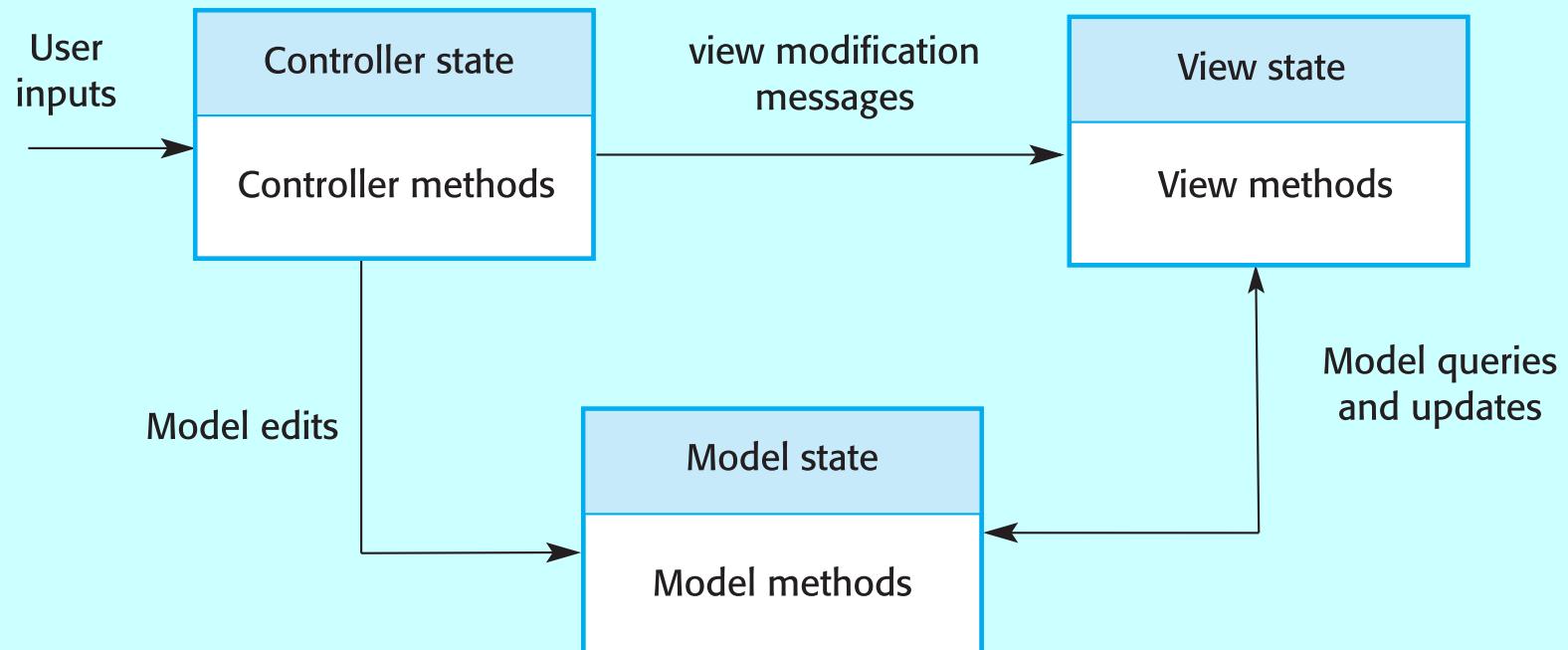
Information presentation

- Information presentation is concerned with presenting system information to system users.
- The information may be presented directly (e.g. text in a word processor) or may be transformed in some way for presentation (e.g. in some graphical form).
- The Model-View-Controller approach is a way of supporting multiple presentations of data.

Information presentation



Model-view-controller



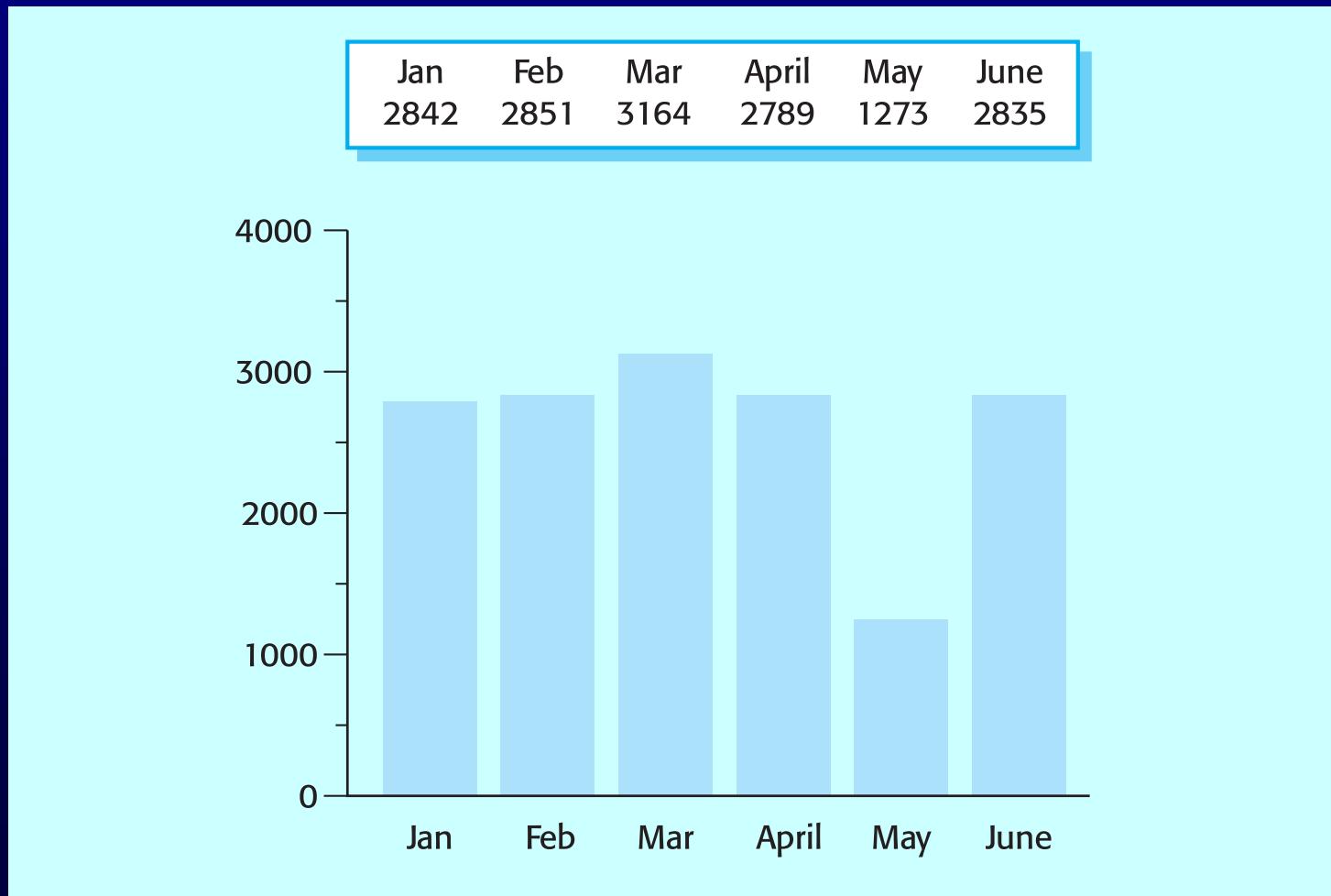
Information presentation

- Static information
 - Initialised at the beginning of a session. It does not change during the session.
 - May be either numeric or textual.
- Dynamic information
 - Changes during a session and the changes must be communicated to the system user.
 - May be either numeric or textual.

Information display factors

- Is the user interested in precise information or data relationships?
- How quickly do information values change?
Must the change be indicated immediately?
- Must the user take some action in response to a change?
- Is there a direct manipulation interface?
- Is the information textual or numeric? Are relative values important?

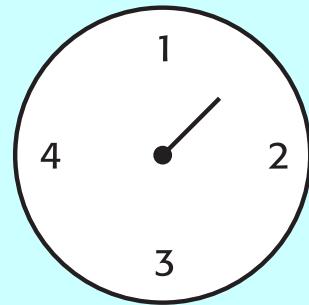
Alternative information presentations



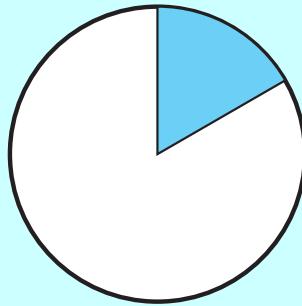
Analogue or digital presentation?

- Digital presentation
 - Compact - takes up little screen space;
 - Precise values can be communicated.
- Analogue presentation
 - Easier to get an 'at a glance' impression of a value;
 - Possible to show relative values;
 - Easier to see exceptional data values.

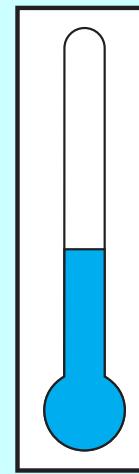
Presentation methods



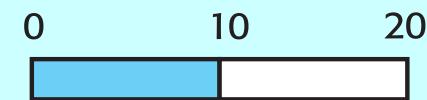
Dial with needle



Pie chart

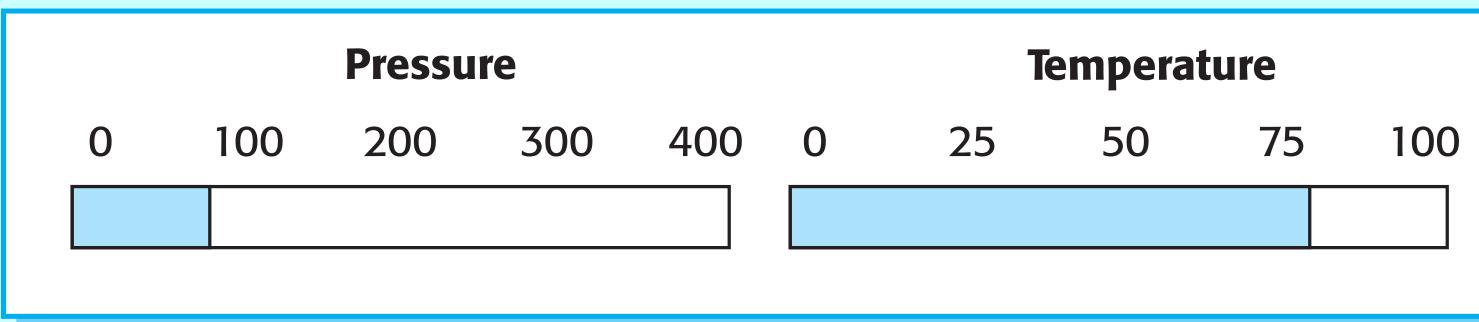


Thermometer



Horizontal bar

Displaying relative values



Data visualisation

- Concerned with techniques for displaying large amounts of information.
- Visualisation can reveal relationships between entities and trends in the data.
- Possible data visualisations are:
 - Weather information collected from a number of sources;
 - The state of a telephone network as a linked set of nodes;
 - Chemical plant visualised by showing pressures and temperatures in a linked set of tanks and pipes;
 - A model of a molecule displayed in 3 dimensions;
 - Web pages displayed as a hyperbolic tree.

Colour displays

- Colour adds an extra dimension to an interface and can help the user understand complex information structures.
- Colour can be used to highlight exceptional events.
- Common mistakes in the use of colour in interface design include:
 - The use of colour to communicate meaning;
 - The over-use of colour in the display.

Colour use guidelines

- Limit the number of colours used and be conservative in their use.
- Use colour change to show a change in system status.
- Use colour coding to support the task that users are trying to perform.
- Use colour coding in a thoughtful and consistent way.
- Be careful about colour pairings.

Error messages

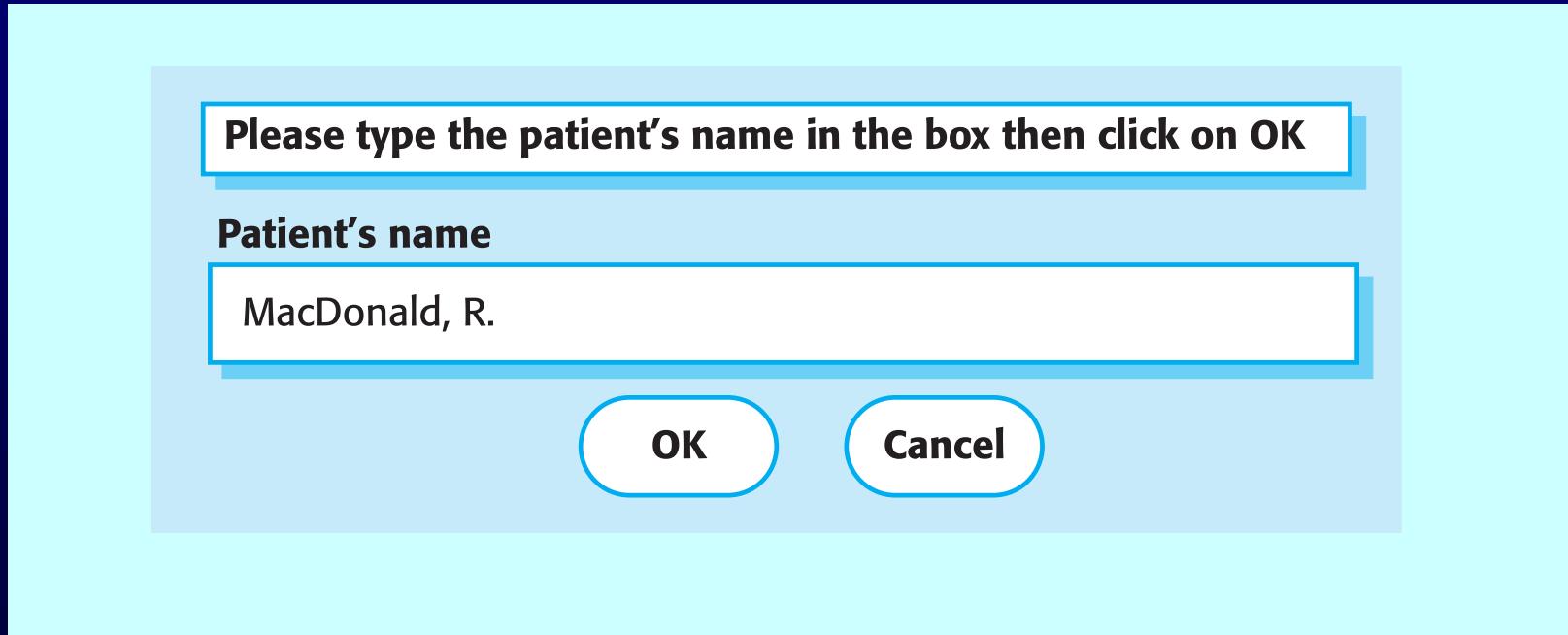
- Error message design is critically important. Poor error messages can mean that a user rejects rather than accepts a system.
- Messages should be polite, concise, consistent and constructive.
- The background and experience of users should be the determining factor in message design.

Design factors in message wording

Factor	Description
Context	Wherever possible, the messages generated by the system should reflect the current user context. As far as is possible, the system should be aware of what the user is doing and should generate messages that are relevant to their current activity.
Experience	As users become familiar with a system they become irritated by long, “meaningful” messages. However, beginners find it difficult to understand short terse statements of a problem. You should provide both types of message and allow the user to control message conciseness.
Skill level	Messages should be tailored to the user’s skills as well as their experience. Messages for the different classes of user may be expressed in different ways depending on the terminology that is familiar to the reader.
Style	Messages should be positive rather than negative. They should use the active rather than the passive mode of address. They should never be insulting or try to be funny.
Culture	Wherever possible, the designer of messages should be familiar with the culture of the country where the system is sold. There are distinct cultural differences between Europe, Asia and America. A suitable message for one culture might be unacceptable in another.

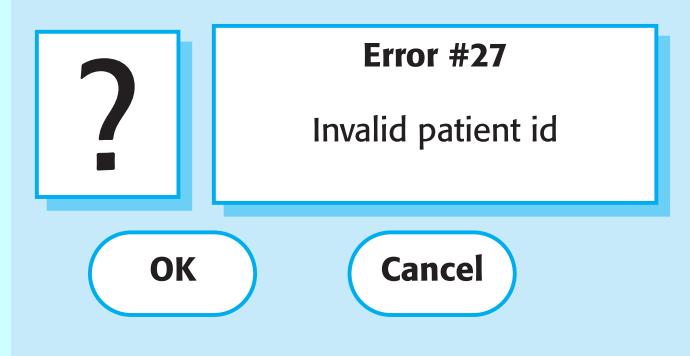
User error

- Assume that a nurse misspells the name of a patient whose records he is trying to retrieve.

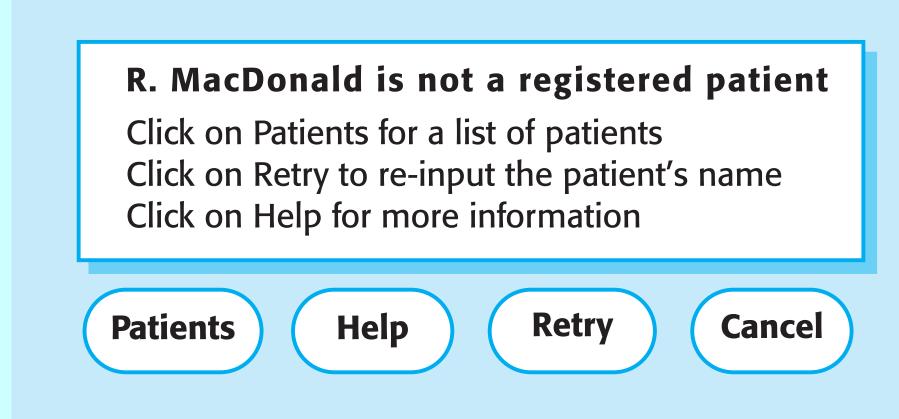


Good and bad message design

System-oriented error message



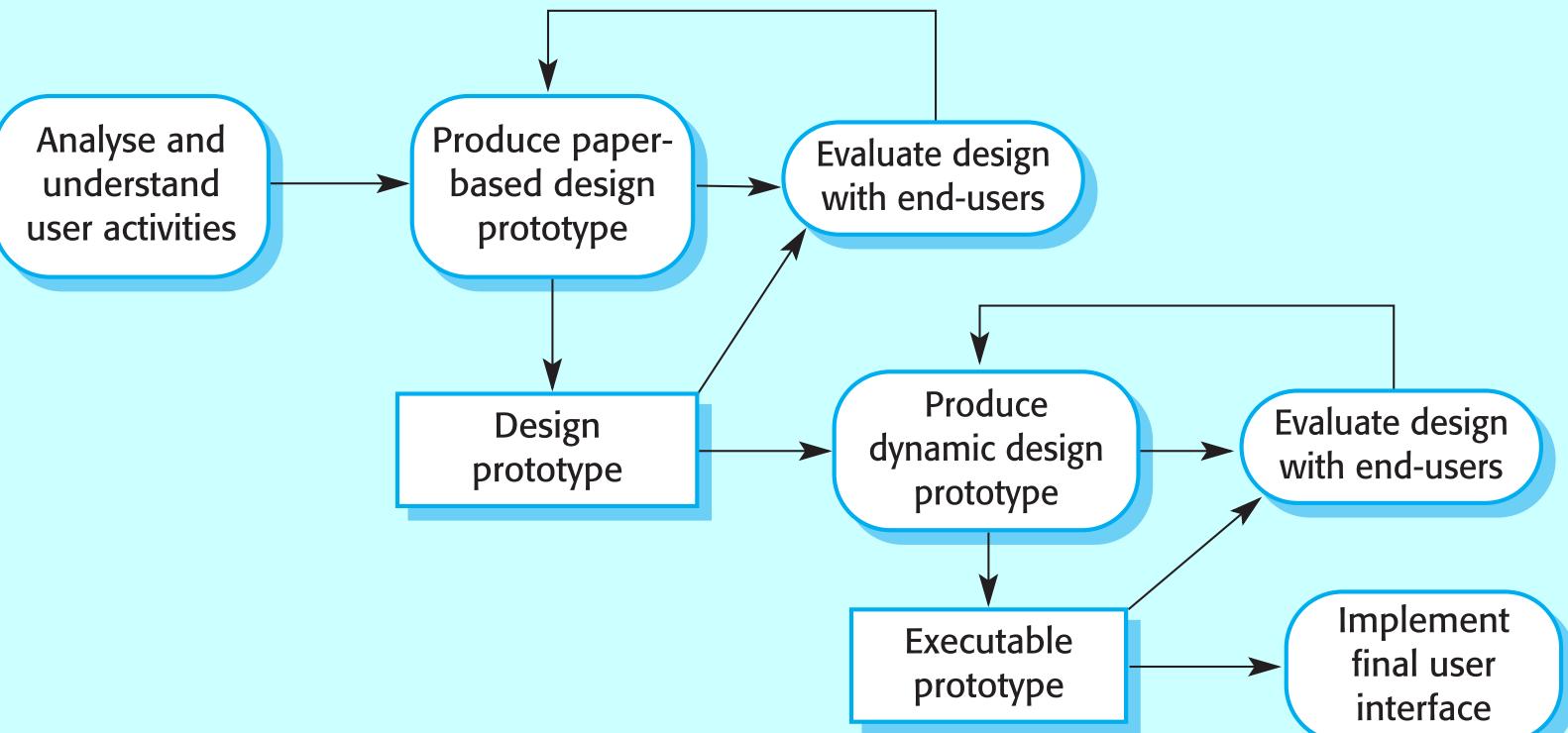
User-oriented error message



The UI design process

- UI design is an iterative process involving close liaisons between users and designers.
- The 3 core activities in this process are:
 - **User analysis.** Understand what the users will do with the system;
 - **System prototyping.** Develop a series of prototypes for experiment;
 - **Interface evaluation.** Experiment with these prototypes with users.

The design process



User analysis

- If you don't understand what the users want to do with a system, you have no realistic prospect of designing an effective interface.
- User analyses have to be described in terms that users and other designers can understand.
- Scenarios where you describe typical episodes of use, are one way of describing these analyses.

User interaction scenario

Jane is a student of Religious Studies and is working on an essay on Indian architecture and how it has been influenced by religious practices. To help her understand this, she would like to access some pictures of details on notable buildings but can't find anything in her local library.

She approaches the subject librarian to discuss her needs and he suggests some search terms that might be used. He also suggests some libraries in New Delhi and London that might have this material so they log on to the library catalogues and do some searching using these terms. They find some source material and place a request for photocopies of the pictures with architectural detail to be posted directly to Jane.

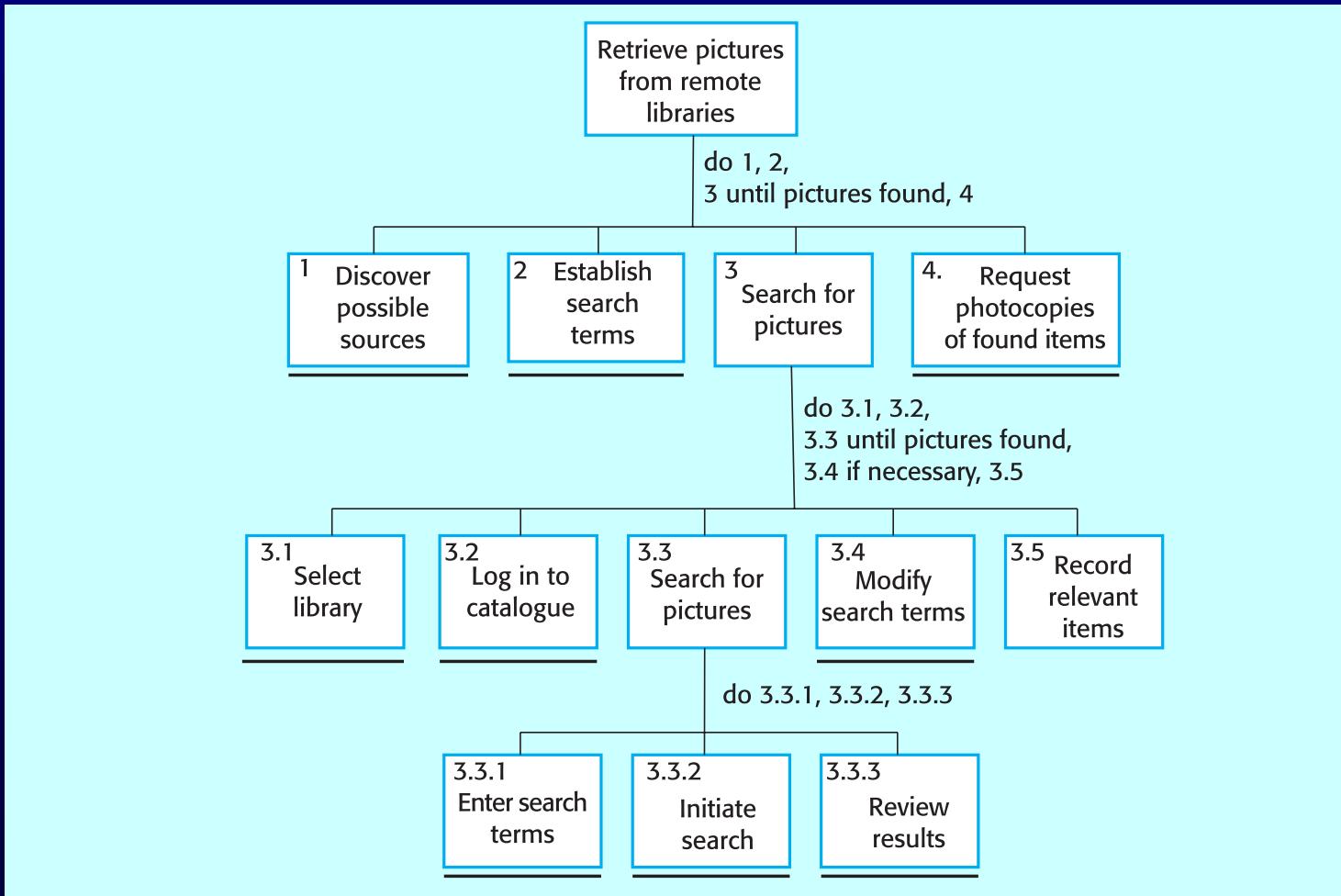
Requirements from the scenario

- Users may not be aware of appropriate search terms so need a way of helping them choose terms.
- Users have to be able to select collections to search.
- Users need to be able to carry out searches and request copies of relevant material.

Analysis techniques

- Task analysis
 - Models the steps involved in completing a task.
- Interviewing and questionnaires
 - Asks the users about the work they do.
- Ethnography
 - Observes the user at work.

Hierarchical task analysis



Interviewing

- Design semi-structured interviews based on open-ended questions.
- Users can then provide information that they think is essential; not just information that you have thought of collecting.
- Group interviews or focus groups allow users to discuss with each other what they do.

Ethnography

- Involves an external observer watching users at work and questioning them in an unscripted way about their work.
- Valuable because many user tasks are intuitive and they find these very difficult to describe and explain.
- Also helps understand the role of social and organisational influences on work.

Ethnographic records

Air traffic control involves a number of control ‘suites’ where the suites controlling adjacent sectors of airspace are physically located next to each other. Flights in a sector are represented by paper strips that are fitted into wooden racks in an order that reflects their position in the sector. If there are not enough slots in the rack (i.e. when the airspace is very busy), controllers spread the strips out on the desk in front of the rack.

When we were observing controllers, we noticed that controllers regularly glanced at the strip racks in the adjacent sector. We pointed this out to them and asked them why they did this. They replied that, if the adjacent controller has strips on their desk, then this meant that they would have a lot of flights entering their sector. They therefore tried to increase the speed of aircraft in the sector to ‘clear space’ for the incoming aircraft.

Insights from ethnography

- Controllers had to see all flights in a sector. Therefore, scrolling displays where flights disappeared off the top or bottom of the display should be avoided.
- The interface had to have some way of telling controllers how many flights were in adjacent sectors so that they could plan their workload.

User interface prototyping

- The aim of prototyping is to allow users to gain direct experience with the interface.
- Without such direct experience, it is impossible to judge the usability of an interface.
- Prototyping may be a two-stage process:
 - Early in the process, paper prototypes may be used;
 - The design is then refined and increasingly sophisticated automated prototypes are then developed.

Paper prototyping

- Work through scenarios using sketches of the interface.
- Use a storyboard to present a series of interactions with the system.
- Paper prototyping is an effective way of getting user reactions to a design proposal.

Prototyping techniques

- Script-driven prototyping
 - Develop a set of scripts and screens using a tool such as Macromedia Director. When the user interacts with these, the screen changes to the next display.
- Visual programming
 - Use a language designed for rapid development such as Visual Basic. See Chapter 17.
- Internet-based prototyping
 - Use a web browser and associated scripts.

User interface evaluation

- Some evaluation of a user interface design should be carried out to assess its suitability.
- Full scale evaluation is very expensive and impractical for most systems.
- Ideally, an interface should be evaluated against a usability specification. However, it is rare for such specifications to be produced.

Usability attributes

Attribute	Description
Learnability	How long does it take a new user to become productive with the system?
Speed of operation	How well does the system response match the user's work practice?
Robustness	How tolerant is the system of user error?
Recoverability	How good is the system at recovering from user errors?
Adaptability	How closely is the system tied to a single model of work?

Simple evaluation techniques

- Questionnaires for user feedback.
- Video recording of system use and subsequent tape evaluation.
- Instrumentation of code to collect information about facility use and user errors.
- The provision of code in the software to collect on-line user feedback.

Key points

- User interface design principles should help guide the design of user interfaces.
- Interaction styles include direct manipulation, menu systems form fill-in, command languages and natural language.
- Graphical displays should be used to present trends and approximate values. Digital displays when precision is required.
- Colour should be used sparingly and consistently.

Key points

- The user interface design process involves user analysis, system prototyping and prototype evaluation.
- The aim of user analysis is to sensitise designers to the ways in which users actually work.
- UI prototyping should be a staged process with early paper prototypes used as a basis for automated prototypes of the interface.
- The goals of UI evaluation are to obtain feedback on how to improve the interface design and to assess if the interface meets its usability requirements.

Verification and Validation

Objectives

- To introduce software verification and validation and to discuss the distinction between them
- To describe the program inspection process and its role in V & V
- To explain static analysis as a verification technique
- To describe the Cleanroom software development process

Topics covered

- Verification and validation planning
- Software inspections
- Automated static analysis
- Cleanroom software development

Verification vs validation

- **Verification:**
"Are we building the product right".
- The software should conform to its specification.
- **Validation:**
"Are we building the right product".
- The software should do what the user really requires.

The V & V process

- Is a whole life-cycle process - V & V must be applied at each stage in the software process.
- Has two principal objectives
 - The discovery of defects in a system;
 - The assessment of whether or not the system is useful and useable in an operational situation.

V& V goals

- Verification and validation should establish confidence that the software is fit for purpose.
- This does NOT mean completely free of defects.
- Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed.

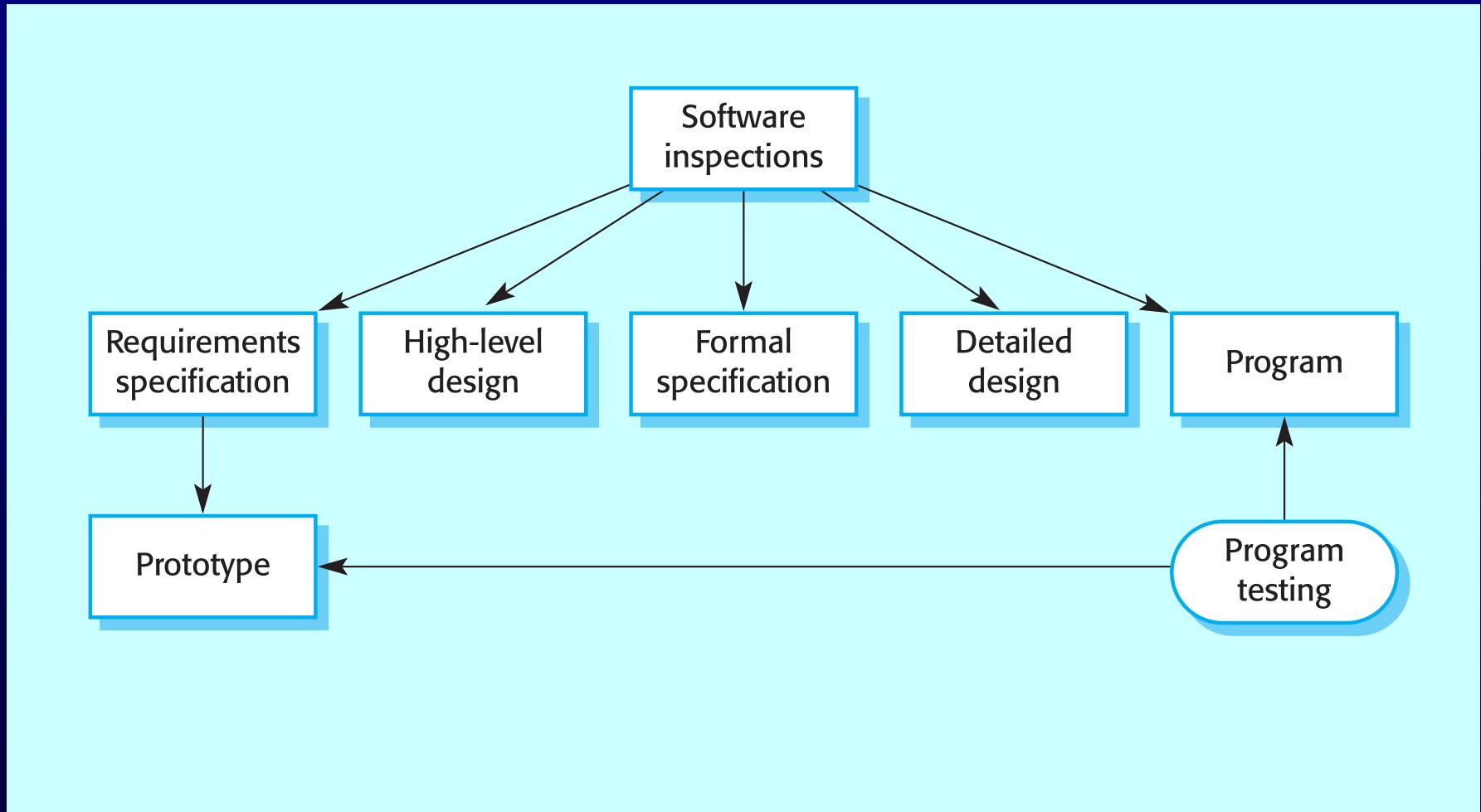
V & V confidence

- Depends on system's purpose, user expectations and marketing environment
 - Software function
 - The level of confidence depends on how critical the software is to an organisation.
 - User expectations
 - Users may have low expectations of certain kinds of software.
 - Marketing environment
 - Getting a product to market early may be more important than finding defects in the program.

Static and dynamic verification

- **Software inspections.** Concerned with analysis of the static system representation to discover problems (static verification)
 - May be supplement by tool-based document and code analysis
- **Software testing.** Concerned with exercising and observing product behaviour (dynamic verification)
 - The system is executed with test data and its operational behaviour is observed

Static and dynamic V&V



Program testing

- Can reveal the presence of errors NOT their absence.
- The only validation technique for non-functional requirements as the software has to be executed to see how it behaves.
- Should be used in conjunction with static verification to provide full V&V coverage.

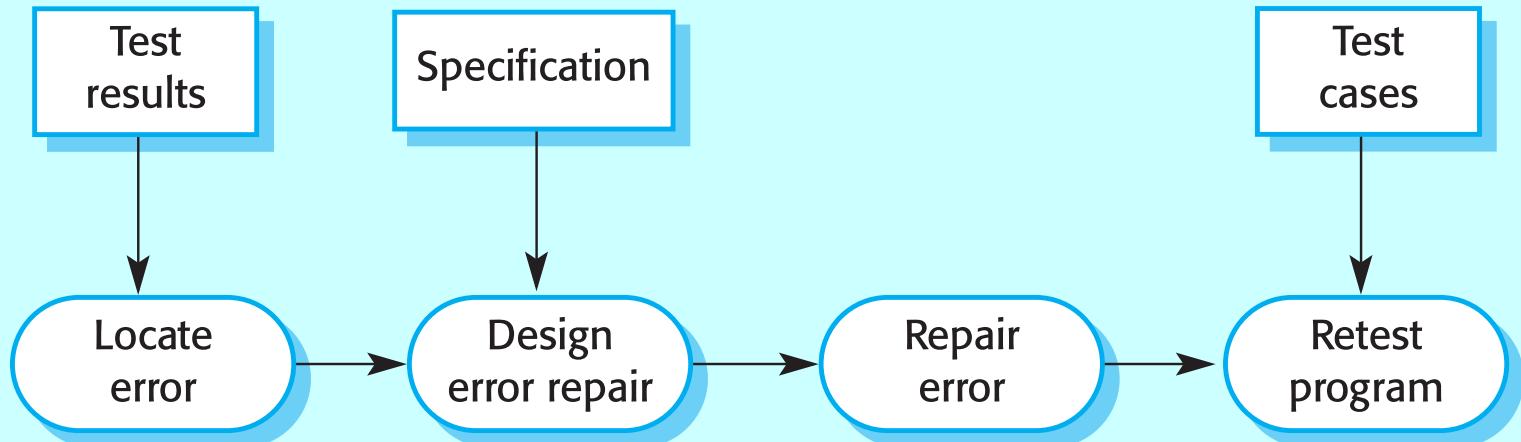
Types of testing

- **Defect testing**
 - Tests designed to discover system defects.
 - A successful defect test is one which reveals the presence of defects in a system.
 - Covered in Chapter 23
- **Validation testing**
 - Intended to show that the software meets its requirements.
 - A successful test is one that shows that a requirements has been properly implemented.

Testing and debugging

- Defect testing and debugging are distinct processes.
- Verification and validation is concerned with establishing the existence of defects in a program.
- Debugging is concerned with locating and repairing these errors.
- Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error.

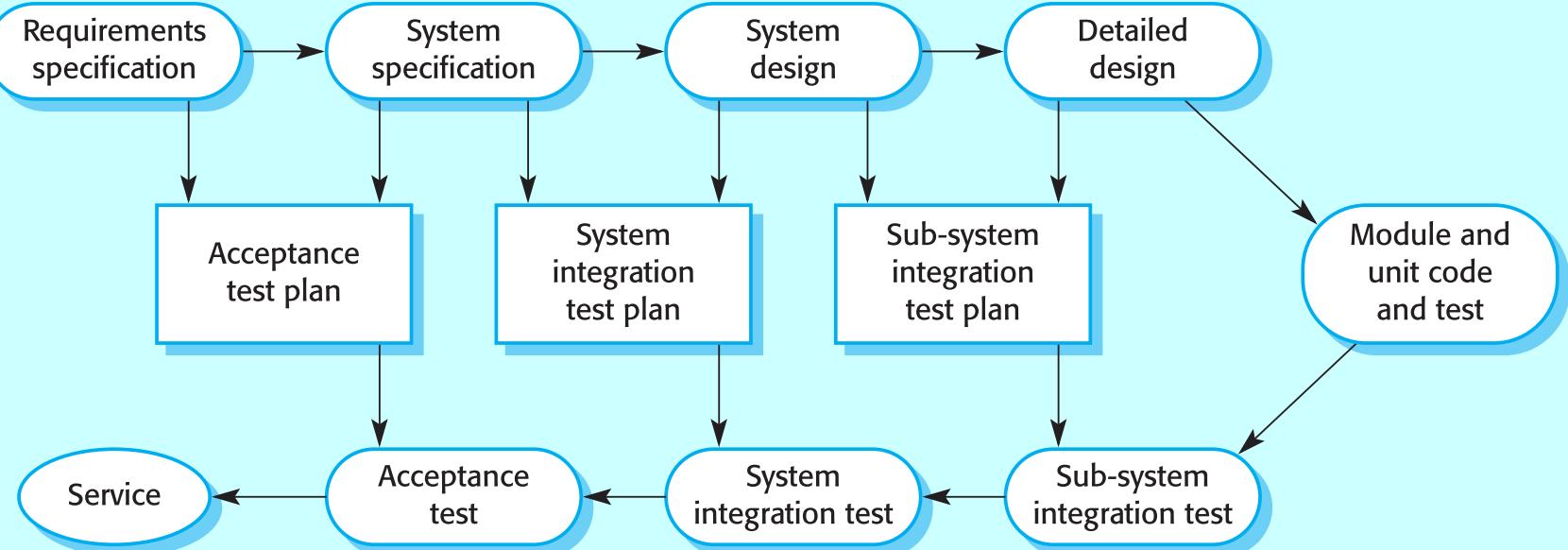
The debugging process



V & V planning

- Careful planning is required to get the most out of testing and inspection processes.
- Planning should start early in the development process.
- The plan should identify the balance between static verification and testing.
- Test planning is about defining standards for the testing process rather than describing product tests.

The V-model of development



The structure of a software test plan

- The testing process.
- Requirements traceability.
- Tested items.
- Testing schedule.
- Test recording procedures.
- Hardware and software requirements.
- Constraints.

The software test plan

The testing process

A description of the major phases of the testing process. These might be as described earlier in this chapter.

Requirements traceability

Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.

Tested items

The products of the software process that are to be tested should be specified.

Testing schedule

An overall testing schedule and resource allocation for this schedule. This, obviously, is linked to the more general project development schedule.

Test recording procedures

It is not enough simply to run tests. The results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it has been carried out correctly.

Hardware and software requirements

This section should set out software tools required and estimated hardware utilisation.

Constraints

Constraints affecting the testing process such as staff shortages should be anticipated in this section.

Software inspections

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

Inspection success

- Many different defects may be discovered in a single inspection. In testing, one defect may mask another so several executions are required.
- The reuse domain and programming knowledge so reviewers are likely to have seen the types of error that commonly arise.

Inspections and testing

- Inspections and testing are complementary and not opposing verification techniques.
- Both should be used during the V & V process.
- Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

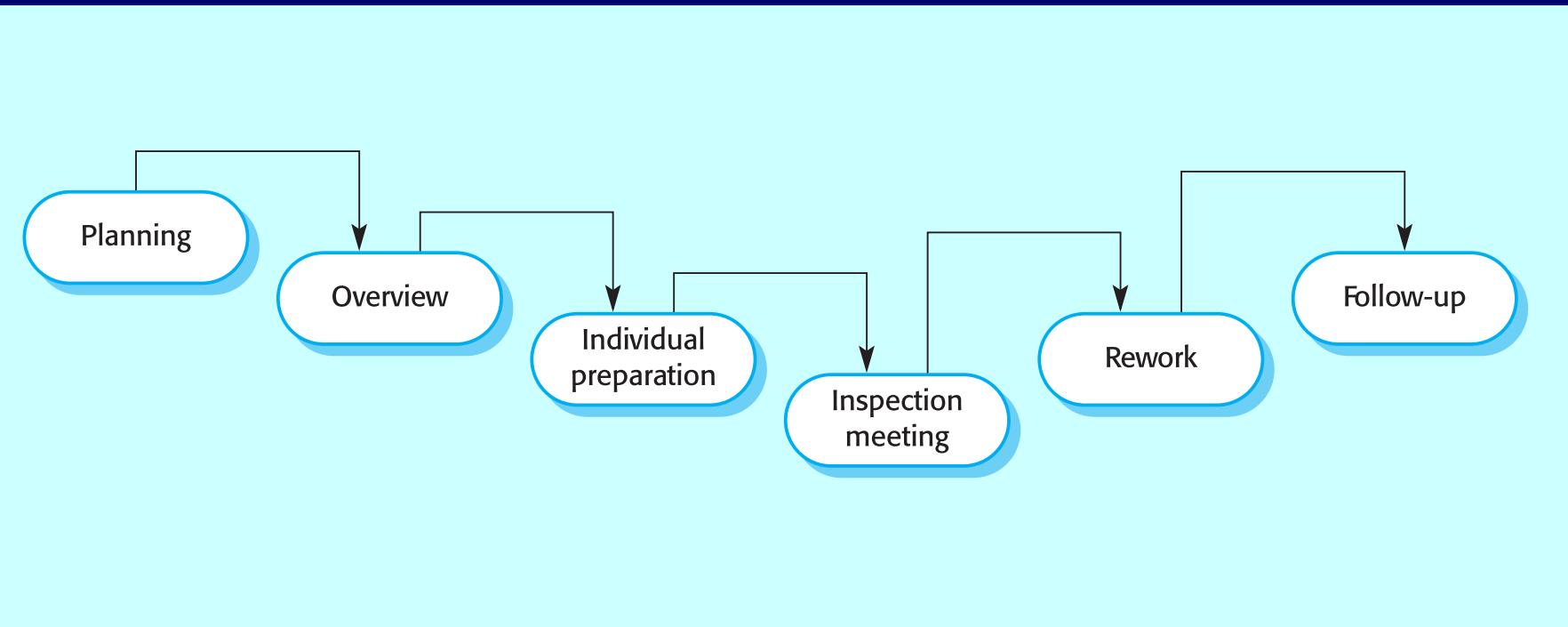
Program inspections

- Formalised approach to document reviews
- Intended explicitly for defect **detection** (not correction).
- Defects may be logical errors, anomalies in the code that might indicate an erroneous condition (e.g. an uninitialised variable) or non-compliance with standards.

Inspection pre-conditions

- A precise specification must be available.
- Team members must be familiar with the organisation standards.
- Syntactically correct code or other system representations must be available.
- An error checklist should be prepared.
- Management must accept that inspection will increase costs early in the software process.
- Management should not use inspections for staff appraisal ie finding out who makes mistakes.

The inspection process



Inspection procedure

- System overview presented to inspection team.
- Code and associated documents are distributed to inspection team in advance.
- Inspection takes place and discovered errors are noted.
- Modifications are made to repair discovered errors.
- Re-inspection may or may not be required.

Inspection roles

Author or owner

The programmer or designer responsible for producing the program or document. Responsible for fixing defects discovered during the inspection process.

Inspector

Finds errors, omissions and inconsistencies in programs and documents. May also identify broader issues that are outside the scope of the inspection team.

Reader

Presents the code or document at an inspection meeting.

Scribe

Records the results of the inspection meeting.

Chairman or moderator

Manages the process and facilitates the inspection. Reports process results to the Chief moderator.

Chief moderator

Responsible for inspection process improvements, checklist updating, standards development etc.

Inspection checklists

- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
- In general, the 'weaker' the type checking, the larger the checklist.
- Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

Inspection checks 1

Data faults	<p>Are all program variables initialised before their values are used?</p> <p>Have all constants been named?</p> <p>Should the upper bound of arrays be equal to the size of the array or Size -1?</p> <p>If character strings are used, is a de limiter explicitly assigned?</p> <p>Is there any possibility of buffer overflow?</p>
Control faults	<p>For each conditional statement, is the condition correct?</p> <p>Is each loop certain to terminate?</p> <p>Are compound statements correctly bracketed?</p> <p>In case statements, are all possible cases accounted for?</p> <p>If a break is required after each case in case statements, has it been included?</p>
Input/output faults	<p>Are all input variables used?</p> <p>Are all output variables assigned a value before they are output?</p> <p>Can unexpected inputs cause corruption?</p>

Inspection checks 2

Interface faults	<p>Do all function and method calls have the correct number of parameters?</p> <p>Do formal and actual parameter types match?</p> <p>Are the parameters in the right order?</p> <p>If components access shared memory, do they have the same model of the shared memory structure?</p>
Storage management faults	<p>If a linked structure is modified, have all links been correctly reassigned?</p> <p>If dynamic storage is used, has space been allocated correctly?</p> <p>Is space explicitly de-allocated after it is no longer required?</p>
Exception management faults	<p>Have all possible error conditions been taken into account?</p>

Inspection rate

- 500 statements/hour during overview.
- 125 source statement/hour during individual preparation.
- 90-125 statements/hour can be inspected.
- Inspection is therefore an expensive process.
- Inspecting 500 lines costs about 40 man/hours effort - about £2800 at UK rates.

Automated static analysis

- Static analysers are software tools for source text processing.
- They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team.
- They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

Static analysis checks

Fault class	Static analysis check
Data faults	Variables used before initialisation Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic

Stages of static analysis

- **Control flow analysis.** Checks for loops with multiple exit or entry points, finds unreachable code, etc.
- **Data use analysis.** Detects uninitialised variables, variables written twice without an intervening assignment, variables which are declared but never used, etc.
- **Interface analysis.** Checks the consistency of routine and procedure declarations and their use

Stages of static analysis

- **Information flow analysis.** Identifies the dependencies of output variables. Does not detect anomalies itself but highlights information for code inspection or review
- **Path analysis.** Identifies paths through the program and sets out the statements executed in that path. Again, potentially useful in the review process
- Both these stages generate vast amounts of information. They must be used with care.

LINT static analysis

```
138% more lint_ex.c
#include <stdio.h>
printarray (Anarray)
int Anarray;
{ printf("%d",Anarray); }

main ()
{
int Anarray[5]; int i; char c;
printarray (Anarray, i, c);
printarray (Anarray) ;
}

139% cc lint_ex.c
140% lint lint_ex.c
```

```
lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

Use of static analysis

- Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler,
- Less cost-effective for languages like Java that have strong type checking and can therefore detect many errors during compilation.

Verification and formal methods

- Formal methods can be used when a mathematical specification of the system is produced.
- They are the ultimate static verification technique.
- They involve detailed mathematical analysis of the specification and may develop formal arguments that a program conforms to its mathematical specification.

Arguments for formal methods

- Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors.
- They can detect implementation errors before testing when the program is analysed alongside the specification.

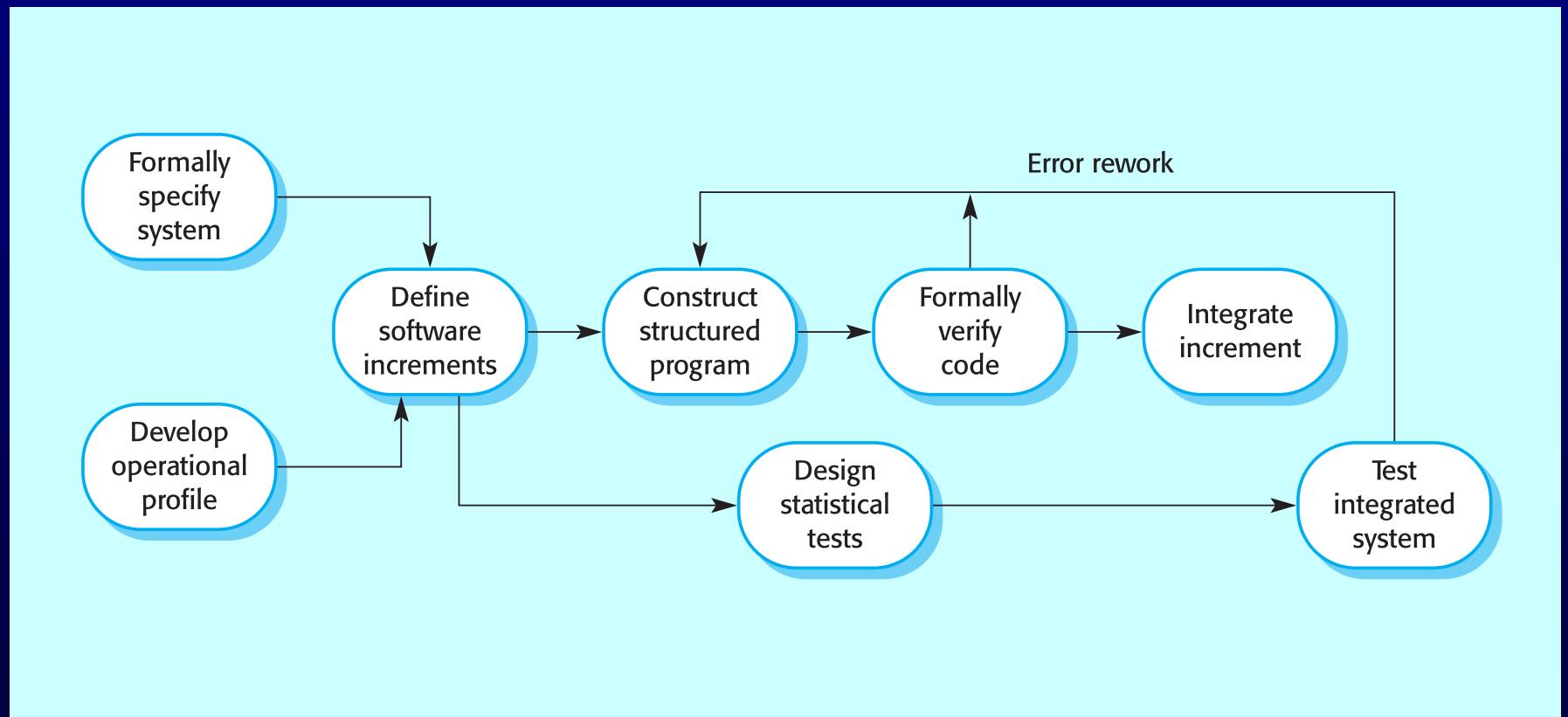
Arguments against formal methods

- Require specialised notations that cannot be understood by domain experts.
- Very expensive to develop a specification and even more expensive to show that a program meets that specification.
- It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques.

Cleanroom software development

- The name is derived from the 'Cleanroom' process in semiconductor fabrication. The philosophy is defect avoidance rather than defect removal.
- This software development process is based on:
 - Incremental development;
 - Formal specification;
 - Static verification using correctness arguments;
 - Statistical testing to determine program reliability.

The Cleanroom process



Cleanroom process characteristics

- Formal specification using a state transition model.
- Incremental development where the customer prioritises increments.
- Structured programming - limited control and abstraction constructs are used in the program.
- Static verification using rigorous inspections.
- Statistical testing of the system (covered in Ch. 24).

Formal specification and inspections

- The state based model is a system specification and the inspection process checks the program against this model.
- The programming approach is defined so that the correspondence between the model and the system is clear.
- Mathematical arguments (not proofs) are used to increase confidence in the inspection process.

Cleanroom process teams

- **Specification team.** Responsible for developing and maintaining the system specification.
- **Development team.** Responsible for developing and verifying the software. The software is NOT executed or even compiled during this process.
- **Certification team.** Responsible for developing a set of statistical tests to exercise the software after development. Reliability growth models used to determine when reliability is acceptable.

Cleanroom process evaluation

- The results of using the Cleanroom process have been very impressive with few discovered faults in delivered systems.
- Independent assessment shows that the process is no more expensive than other approaches.
- There were fewer errors than in a 'traditional' development process.
- However, the process is not widely used. It is not clear how this approach can be transferred to an environment with less skilled or less motivated software engineers.

Key points

- Verification and validation are not the same thing. Verification shows conformance with specification; validation shows that the program meets the customer's needs.
- Test plans should be drawn up to guide the testing process.
- Static verification techniques involve examination and analysis of the program for error detection.

Key points

- Program inspections are very effective in discovering errors.
- Program code in inspections is systematically checked by a small team to locate software faults.
- Static analysis tools can discover program anomalies which may be an indication of faults in the code.
- The Cleanroom development process depends on incremental development, static verification and statistical testing.

Software testing

Objectives

- To discuss the distinctions between validation testing and defect testing
- To describe the principles of system and component testing
- To describe strategies for generating system test cases
- To understand the essential characteristics of tool used for test automation

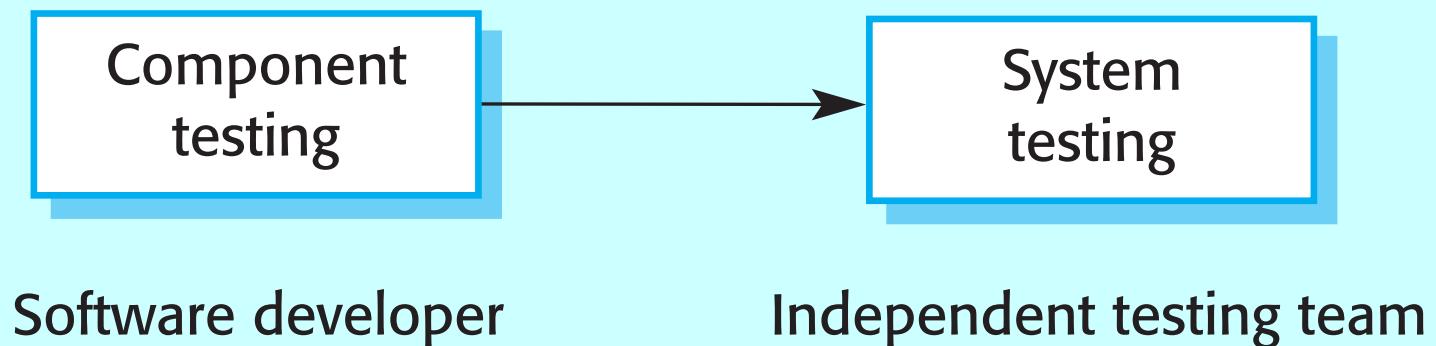
Topics covered

- System testing
- Component testing
- Test case design
- Test automation

The testing process

- Component testing
 - Testing of individual program components;
 - Usually the responsibility of the component developer (except sometimes for critical systems);
 - Tests are derived from the developer's experience.
- System testing
 - Testing of groups of components integrated to create a system or sub-system;
 - The responsibility of an independent testing team;
 - Tests are based on a system specification.

Testing phases



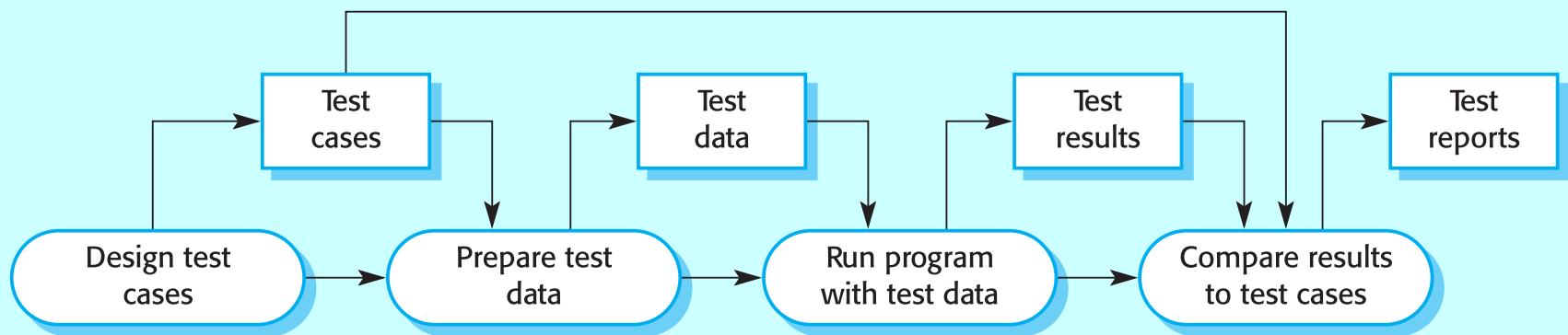
Defect testing

- The goal of defect testing is to discover defects in programs
- A *successful* defect test is a test which causes a program to behave in an anomalous way
- Tests show the presence not the absence of defects

Testing process goals

- **Validation testing**
 - To demonstrate to the developer and the system customer that the software meets its requirements;
 - A successful test shows that the system operates as intended.
- **Defect testing**
 - To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification;
 - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

The software testing process



Testing policies

- Only exhaustive testing can show a program is free from defects. However, exhaustive testing is impossible,
- Testing policies define the approach to be used in selecting system tests:
 - All functions accessed through menus should be tested;
 - Combinations of functions accessed through the same menu should be tested;
 - Where user input is required, all functions must be tested with correct and incorrect input.

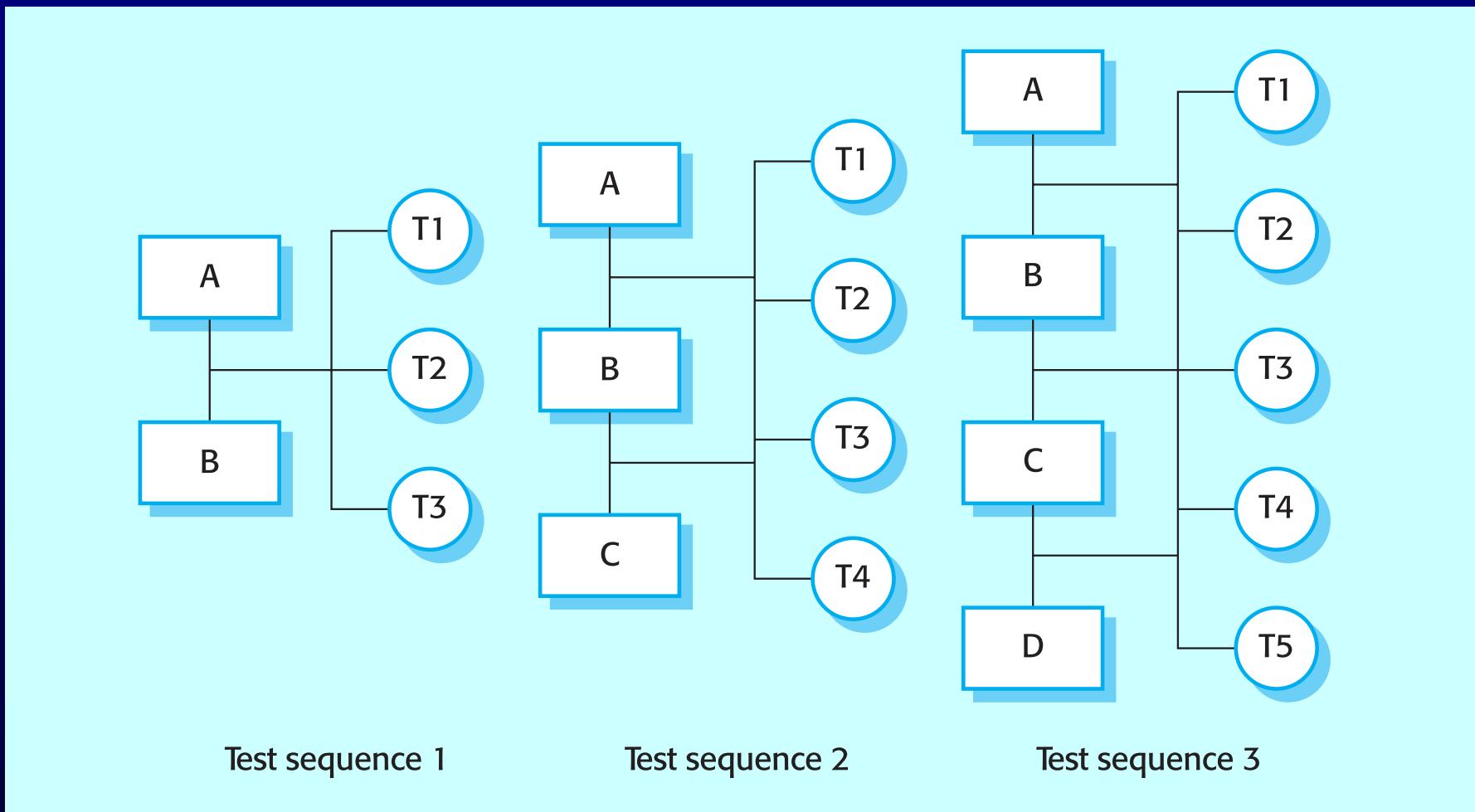
System testing

- Involves integrating components to create a system or sub-system.
- May involve testing an increment to be delivered to the customer.
- Two phases:
 - **Integration testing** - the test team have access to the system source code. The system is tested as components are integrated.
 - **Release testing** - the test team test the complete system to be delivered as a black-box.

Integration testing

- Involves building a system from its components and testing it for problems that arise from component interactions.
- Top-down integration
 - Develop the skeleton of the system and populate it with components.
- Bottom-up integration
 - Integrate infrastructure components then add functional components.
- To simplify error localisation, systems should be incrementally integrated.

Incremental integration testing



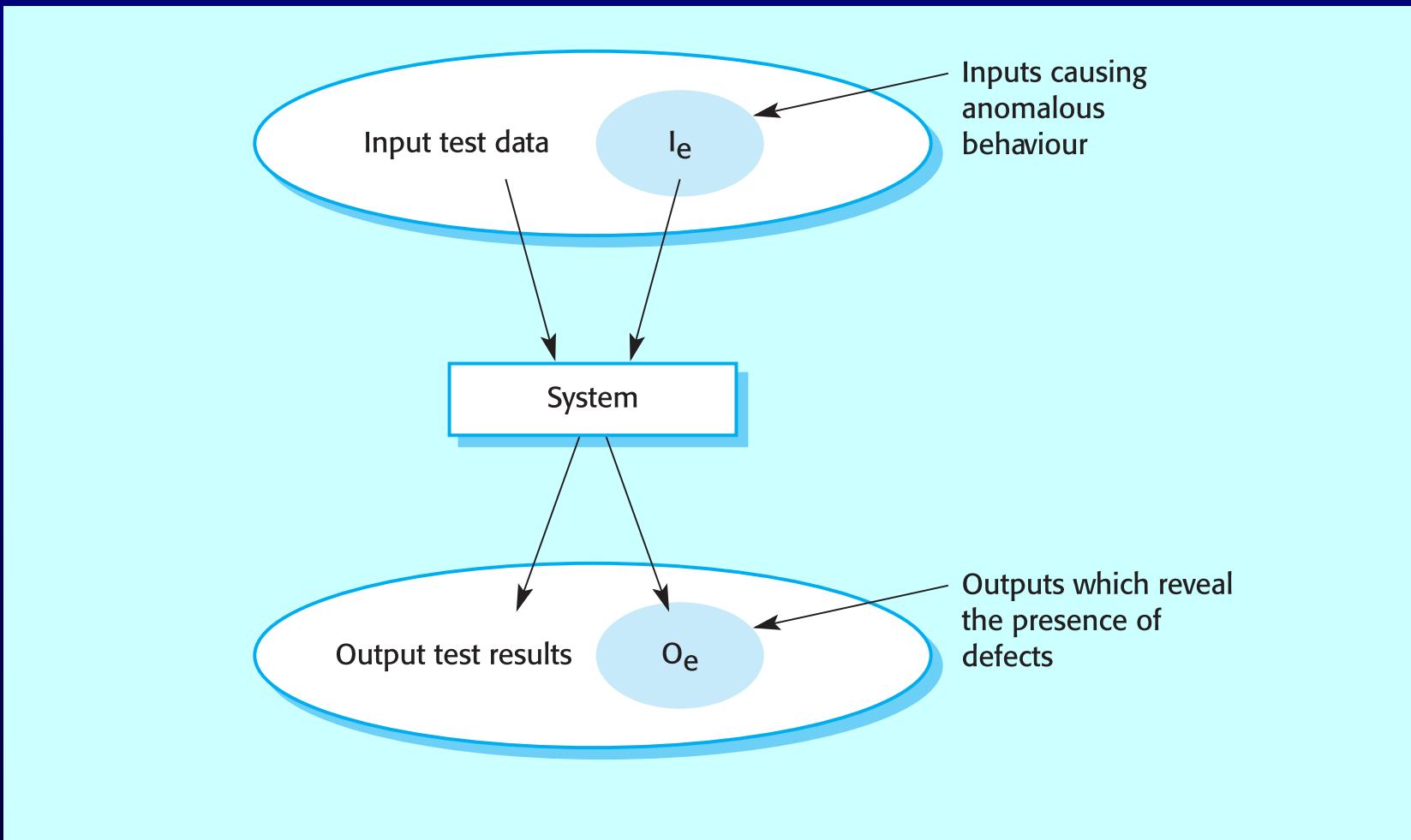
Testing approaches

- Architectural validation
 - Top-down integration testing is better at discovering errors in the system architecture.
- System demonstration
 - Top-down integration testing allows a limited demonstration at an early stage in the development.
- Test implementation
 - Often easier with bottom-up integration testing.
- Test observation
 - Problems with both approaches. Extra code may be required to observe tests.

Release testing

- The process of testing a release of a system that will be distributed to customers.
- Primary goal is to increase the supplier's confidence that the system meets its requirements.
- Release testing is usually black-box or functional testing
 - Based on the system specification only;
 - Testers do not have knowledge of the system implementation.

Black-box testing



Testing guidelines

- Testing guidelines are hints for the testing team to help them choose tests that will reveal defects in the system
 - Choose inputs that force the system to generate all error messages;
 - Design inputs that cause buffers to overflow;
 - Repeat the same input or input series several times;
 - Force invalid outputs to be generated;
 - Force computation results to be too large or too small.

Testing scenario

A student in Scotland is studying American History and has been asked to write a paper on Frontier mentality in the American West from 1840 to 1880. To do this, she needs to find sources from a range of libraries. She logs on to the LIBSYS system and uses the search facility to discover if she can access original documents from that time. She discovers sources in various US university libraries and downloads copies of some of these. However, for one document, she needs to have confirmation from her university that she is a genuine student and that use is for non-commercial purposes. The student then uses the facility in LIBSYS that can request such permission and registers her request. If granted, the document will be downloaded to the registered library server and printed for her. She receives a message from LIBSYS telling her that she will receive an e-mail message when the printed document is available for collection.

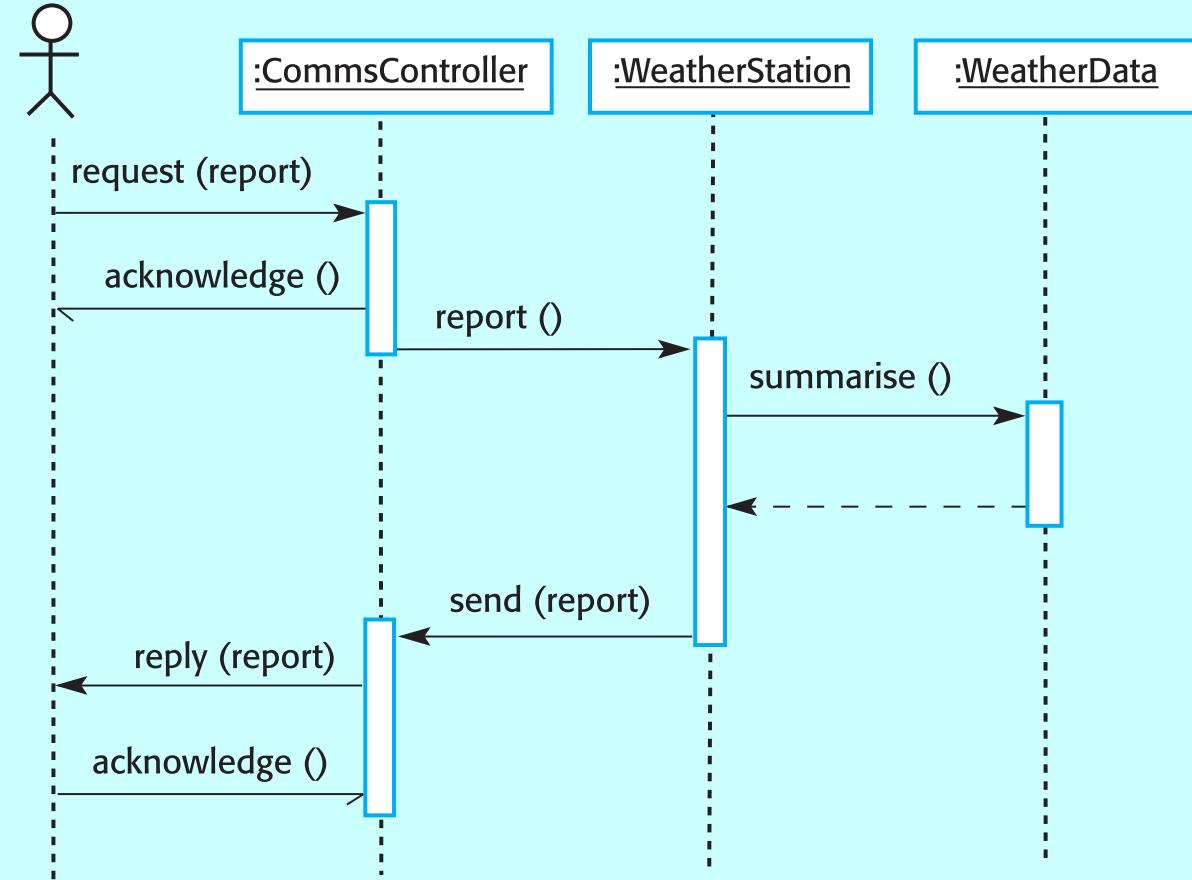
System tests

1. Test the login mechanism using correct and incorrect logins to check that valid users are accepted and invalid users are rejected.
2. Test the search facility using different queries against known sources to check that the search mechanism is actually finding documents.
3. Test the system presentation facility to check that information about documents is displayed properly.
4. Test the mechanism to request permission for downloading.
5. Test the e-mail response indicating that the downloaded document is available.

Use cases

- Use cases can be a basis for deriving the tests for a system. They help identify operations to be tested and help design the required test cases.
- From an associated sequence diagram, the inputs and outputs to be created for the tests can be identified.

Collect weather data sequence chart



Performance testing

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.

Stress testing

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light.
- Stressing the system test failure behaviour.. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data.
- Stress testing is particularly relevant to distributed systems that can exhibit severe degradation as a network becomes overloaded.

Component testing

- Component or unit testing is the process of testing individual components in isolation.
- It is a defect testing process.
- Components may be:
 - Individual functions or methods within an object;
 - Object classes with several attributes and methods;
 - Composite components with defined interfaces used to access their functionality.

Object class testing

- Complete test coverage of a class involves
 - Testing all operations associated with an object;
 - Setting and interrogating all object attributes;
 - Exercising the object in all possible states.
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised.

Weather station object interface

WeatherStation

identifier

reportWeather ()
calibrate (instruments)
test ()
startup (instruments)
shutdown (instruments)

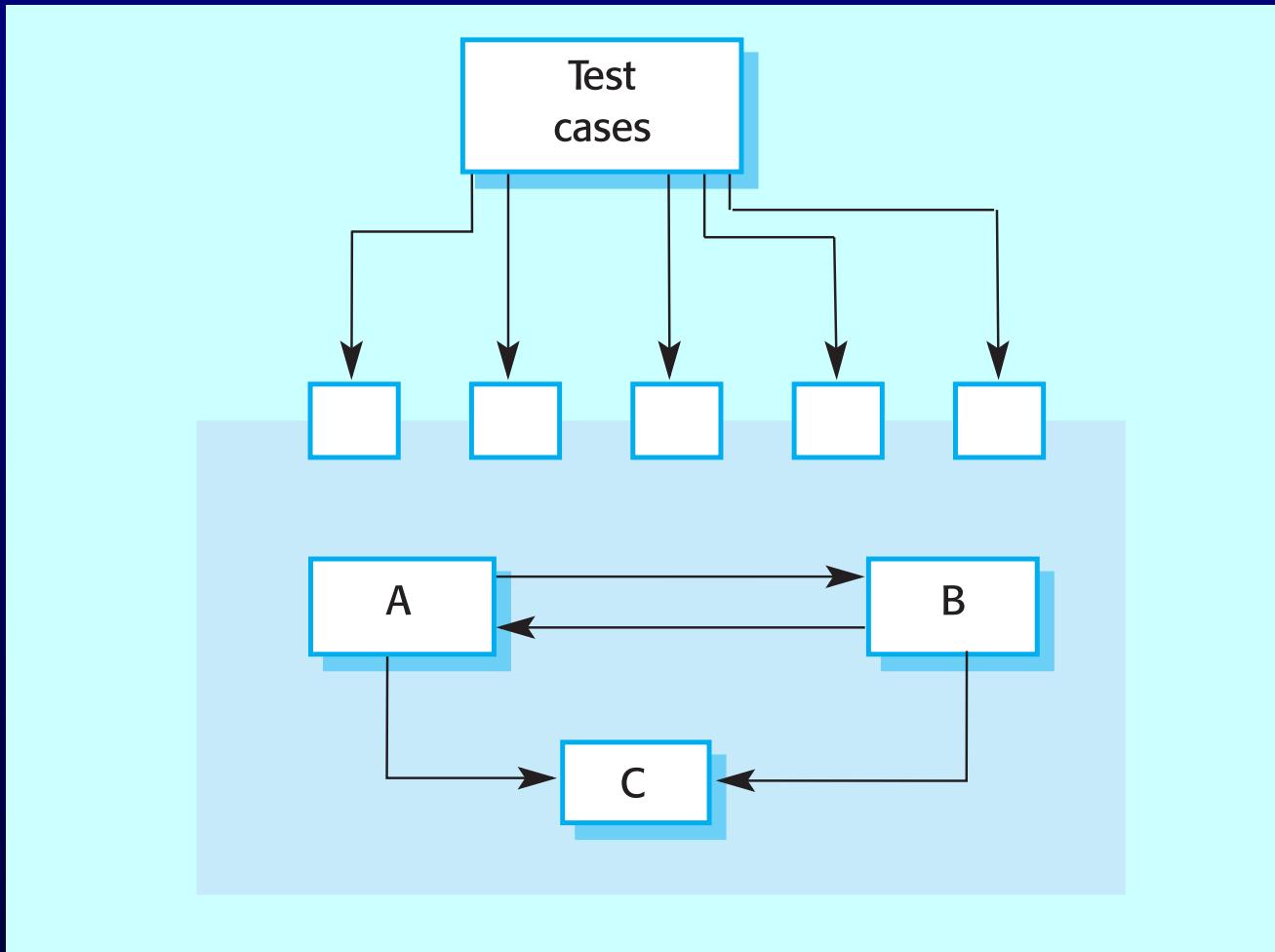
Weather station testing

- Need to define test cases for reportWeather, calibrate, test, startup and shutdown.
- Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions
- For example:
 - Waiting -> Calibrating -> Testing -> Transmitting -> Waiting

Interface testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Particularly important for object-oriented development as objects are defined by their interfaces.

Interface testing



Interface types

- Parameter interfaces
 - Data passed from one procedure to another.
- Shared memory interfaces
 - Block of memory is shared between procedures or functions.
- Procedural interfaces
 - Sub-system encapsulates a set of procedures to be called by other sub-systems.
- Message passing interfaces
 - Sub-systems request services from other sub-systems

Interface errors

- Interface misuse
 - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.
- Interface misunderstanding
 - A calling component embeds assumptions about the behaviour of the called component which are incorrect.
- Timing errors
 - The called and the calling component operate at different speeds and out-of-date information is accessed.

Interface testing guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- Use stress testing in message passing systems.
- In shared memory systems, vary the order in which components are activated.

Test case design

- Involves designing the test cases (inputs and outputs) used to test the system.
- The goal of test case design is to create a set of tests that are effective in validation and defect testing.
- Design approaches:
 - Requirements-based testing;
 - Partition testing;
 - Structural testing.

Requirements based testing

- A general principle of requirements engineering is that requirements should be testable.
- Requirements-based testing is a validation testing technique where you consider each requirement and derive a set of tests for that requirement.

LIBSYS requirements

The user shall be able to search either all of the initial set of databases or select a subset from it.

The system shall provide appropriate viewers for the user to read documents in the document store.

Every order shall be allocated a unique identifier (ORDER_ID) that the user shall be able to copy to the account's permanent storage area.

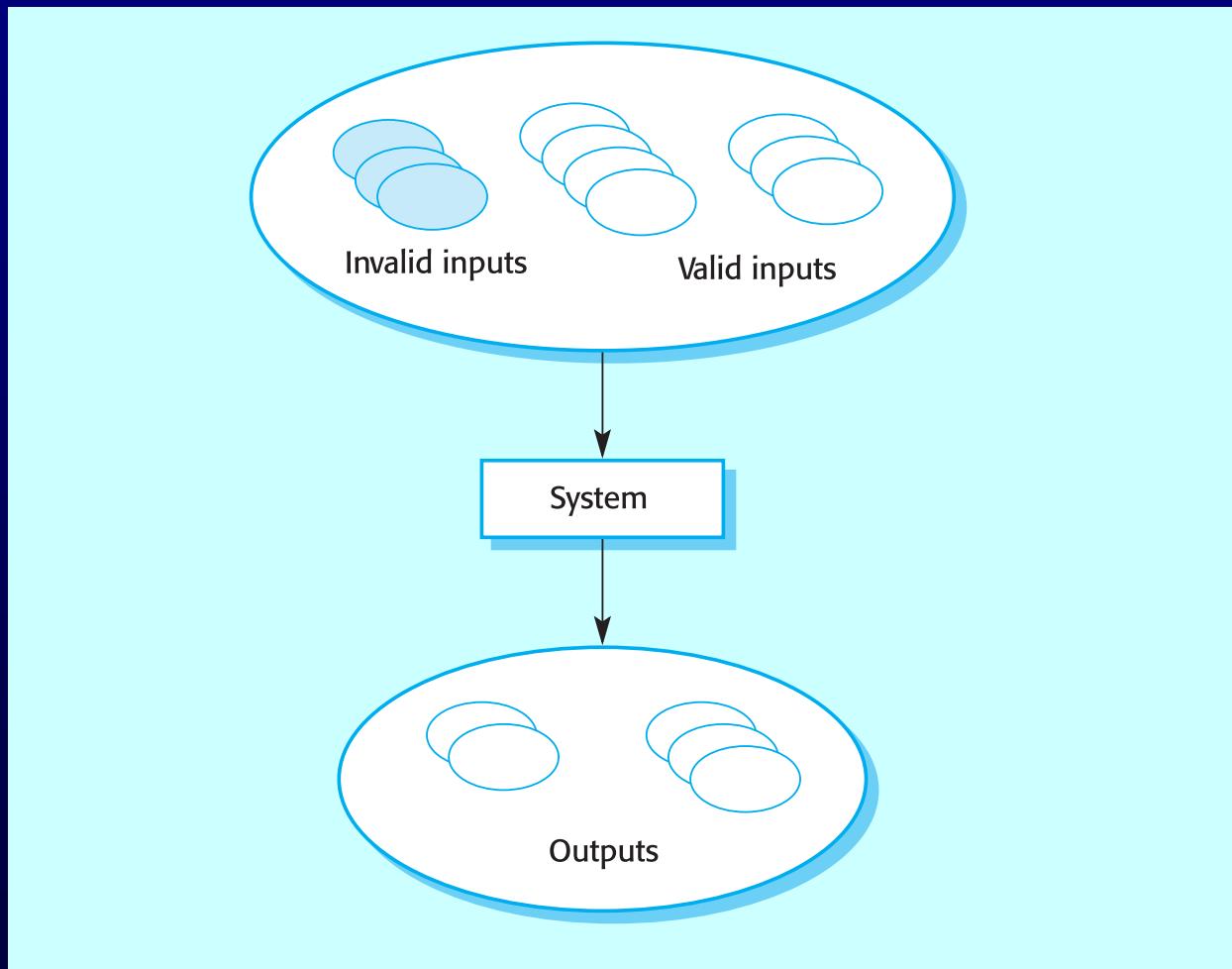
LIBSYS tests

- Initiate user search for searches for items that are known to be present and known not to be present, where the set of databases includes 1 database.
- Initiate user searches for items that are known to be present and known not to be present, where the set of databases includes 2 databases
- Initiate user searches for items that are known to be present and known not to be present where the set of databases includes more than 2 databases.
- Select one database from the set of databases and initiate user searches for items that are known to be present and known not to be present.
- Select more than one database from the set of databases and initiate searches for items that are known to be present and known not to be present.

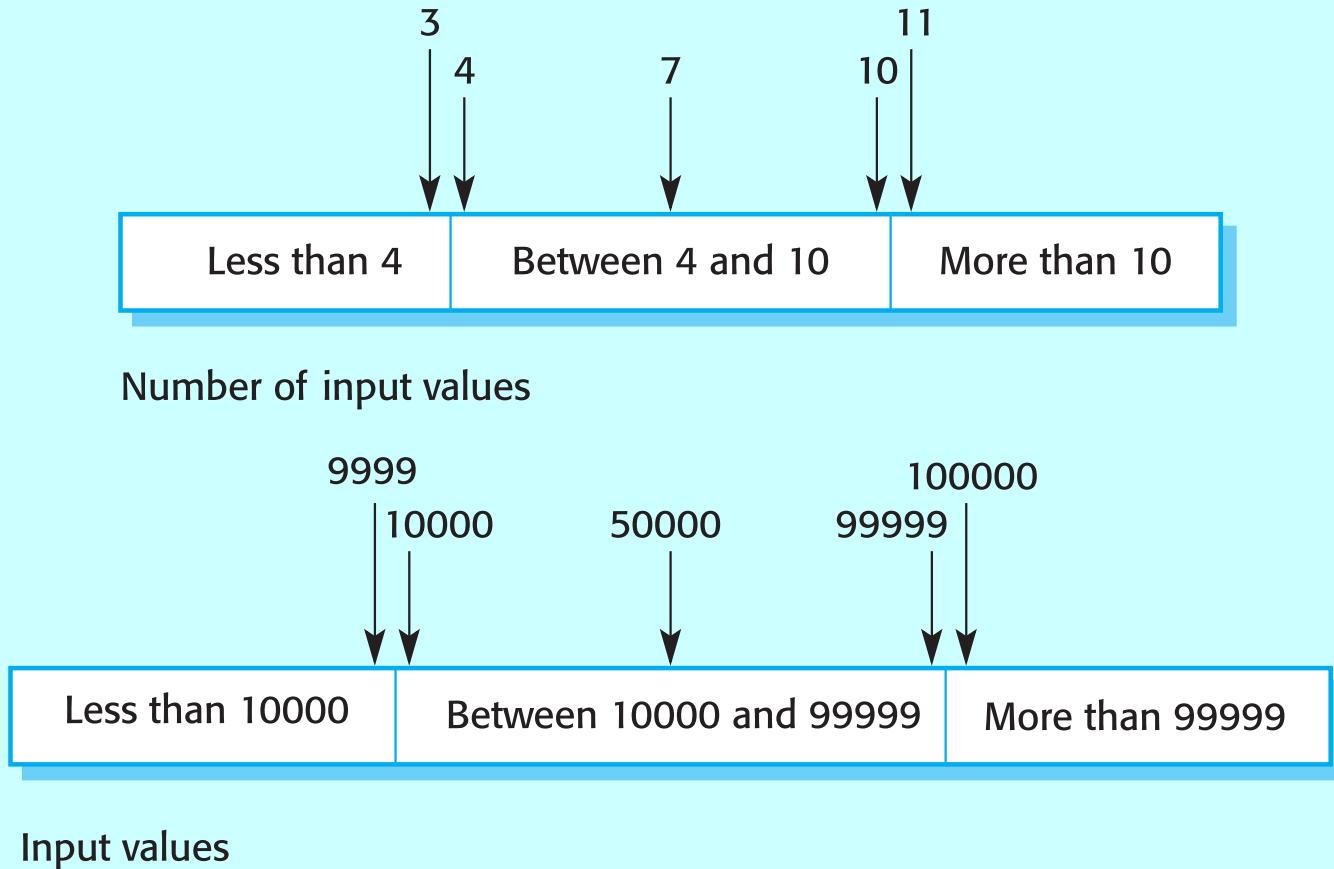
Partition testing

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an **equivalence partition** or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.

Equivalence partitioning



Equivalence partitions



Search routine specification

```
procedure Search (Key : ELEM ; T: SEQ of ELEM;  
    Found : in out BOOLEAN; L: in out ELEM_INDEX) ;
```

Pre-condition

- the sequence has at least one element
- T'FIRST <= T'LAST

Post-condition

- the element is found and is referenced by L
- (Found and T (L) = Key)

or

- the element is not in the array
- (**not** Found **and**
- not** (**exists** i, T'FIRST >= i <= T'LAST, T (i) = Key))

Search routine - input partitions

- Inputs which conform to the pre-conditions.
- Inputs where a pre-condition does not hold.
- Inputs where the key element is a member of the array.
- Inputs where the key element is not a member of the array.

Testing guidelines (sequences)

- Test software with sequences which have only a single value.
- Use sequences of different sizes in different tests.
- Derive tests so that the first, middle and last elements of the sequence are accessed.
- Test with sequences of zero length.

Search routine - input partitions

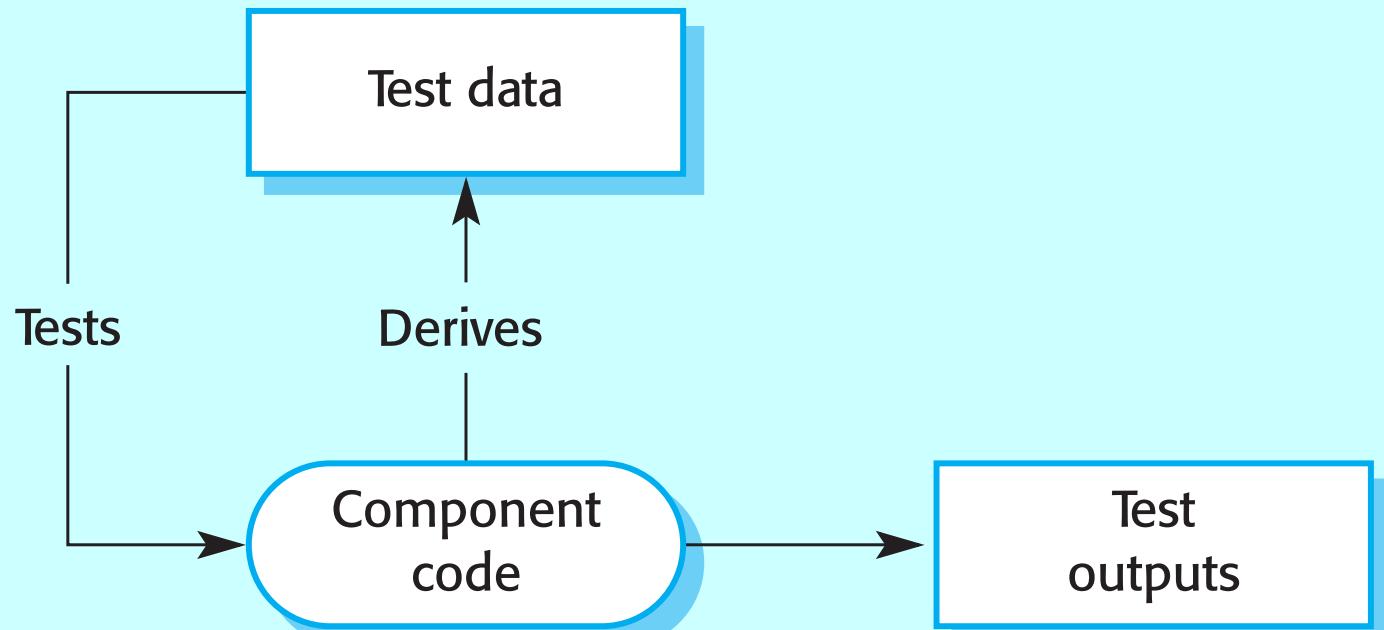
Sequence	Element
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Structural testing

- Sometime called white-box testing.
- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases.
- Objective is to exercise all program statements (not all path combinations).

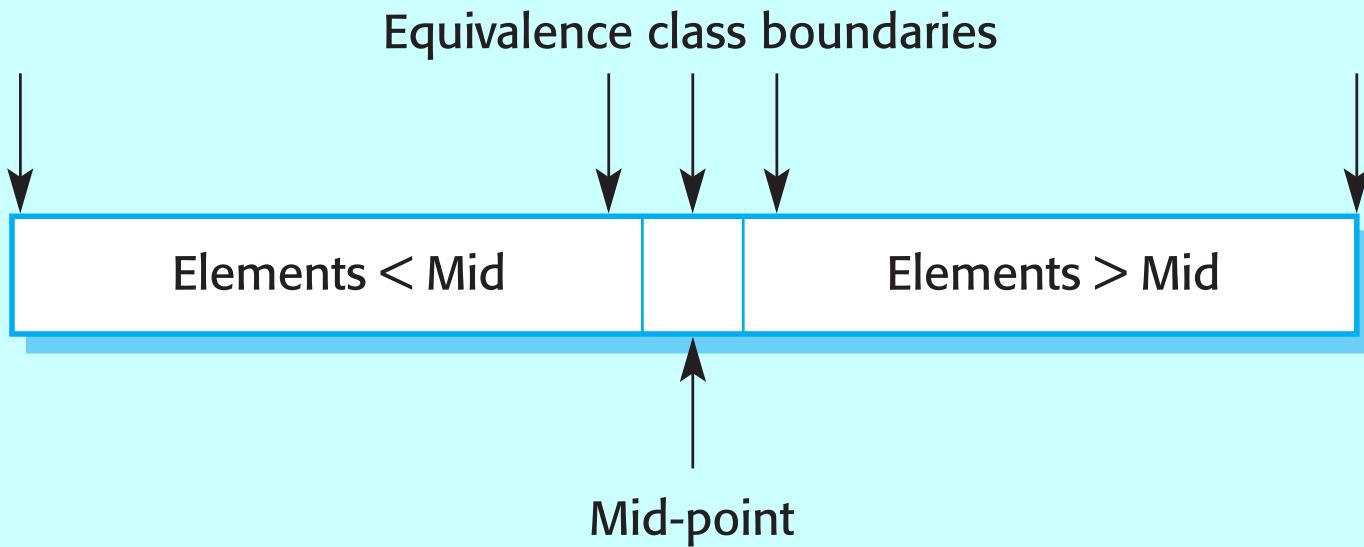
Structural testing



Binary search - equiv. partitions

- Pre-conditions satisfied, key element in array.
- Pre-conditions satisfied, key element not in array.
- Pre-conditions unsatisfied, key element in array.
- Pre-conditions unsatisfied, key element not in array.
- Input array has a single value.
- Input array has an even number of values.
- Input array has an odd number of values.

Binary search equiv. partitions



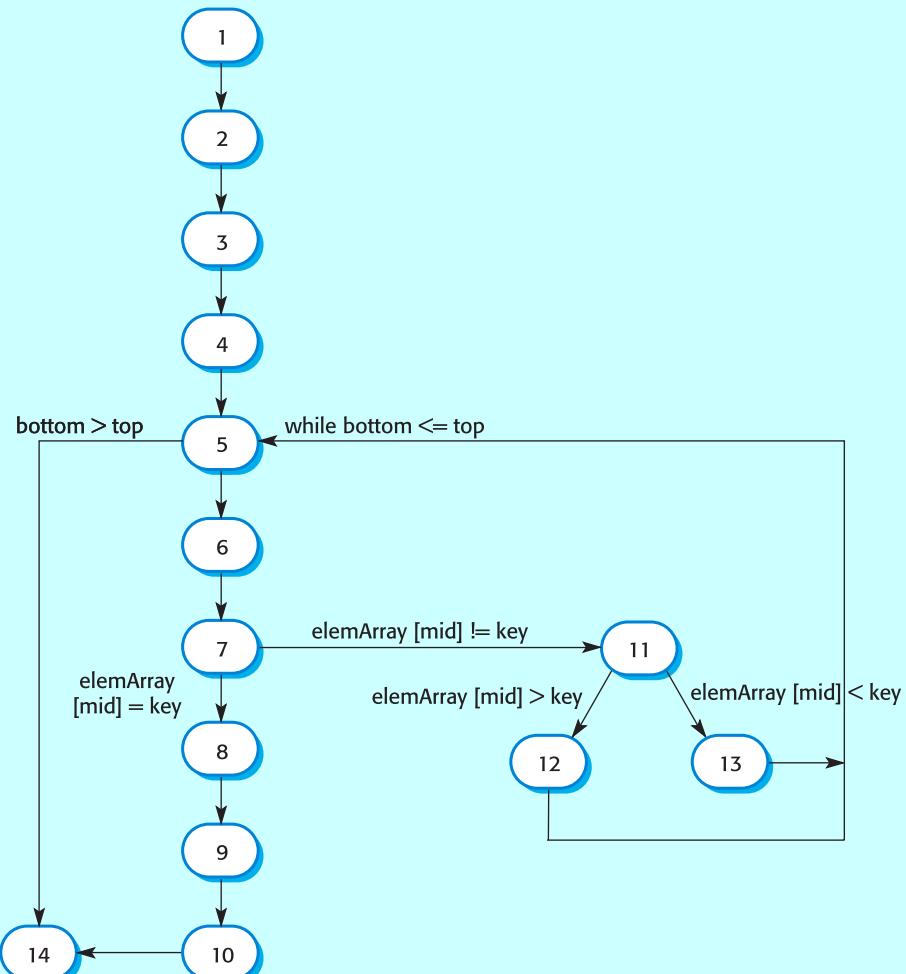
Binary search - test cases

Input array (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

Path testing

- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once.
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control.
- Statements with conditions are therefore nodes in the flow graph.

Binary search flow graph



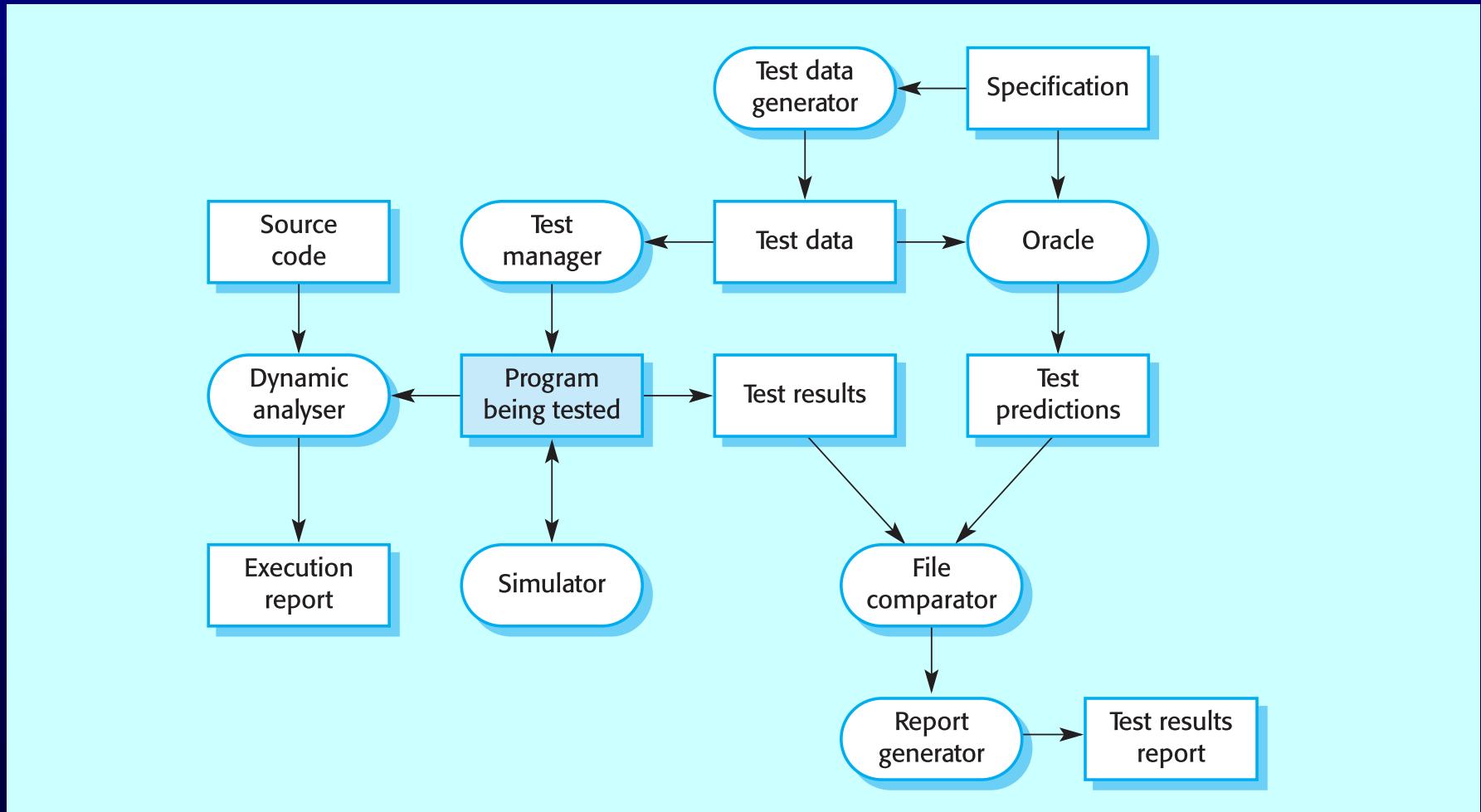
Independent paths

- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
- 1, 2, 3, 4, 5, 14
- 1, 2, 3, 4, 5, 6, 7, 11, 12, 5, ...
- 1, 2, 3, 4, 6, 7, 2, 11, 13, 5, ...
- Test cases should be derived so that all of these paths are executed
- A dynamic program analyser may be used to check that paths have been executed

Test automation

- Testing is an expensive process phase. Testing workbenches provide a range of tools to reduce the time required and total testing costs.
- Systems such as Junit support the automatic execution of tests.
- Most testing workbenches are open systems because testing needs are organisation-specific.
- They are sometimes difficult to integrate with closed design and analysis workbenches.

A testing workbench



Testing workbench adaptation

- Scripts may be developed for user interface simulators and patterns for test data generators.
- Test outputs may have to be prepared manually for comparison.
- Special-purpose file comparators may be developed.

Key points

- Testing can show the presence of faults in a system; it cannot prove there are no remaining faults.
- Component developers are responsible for component testing; system testing is the responsibility of a separate team.
- Integration testing is testing increments of the system; release testing involves testing a system to be released to a customer.
- Use experience and guidelines to design test cases in defect testing.

Key points

- Interface testing is designed to discover defects in the interfaces of composite components.
- Equivalence partitioning is a way of discovering test cases - all cases in a partition should behave in the same way.
- Structural analysis relies on analysing a program and deriving tests from this analysis.
- Test automation reduces testing costs by supporting the test process with a range of software tools.

Critical Systems Validation

Objectives

- To explain how system reliability can be measured and how reliability growth models can be used for reliability prediction
- To describe safety arguments and how these are used
- To discuss the problems of safety assurance
- To introduce safety cases and how these are used in safety validation

Topics covered

- Reliability validation
- Safety assurance
- Security assessment
- Safety and dependability cases

Validation of critical systems

- The verification and validation costs for critical systems involves additional validation processes and analysis than for non-critical systems:
 - The costs and consequences of failure are high so it is cheaper to find and remove faults than to pay for system failure;
 - You may have to make a formal case to customers or to a regulator that the system meets its dependability requirements. This dependability case may require specific V & V activities to be carried out.

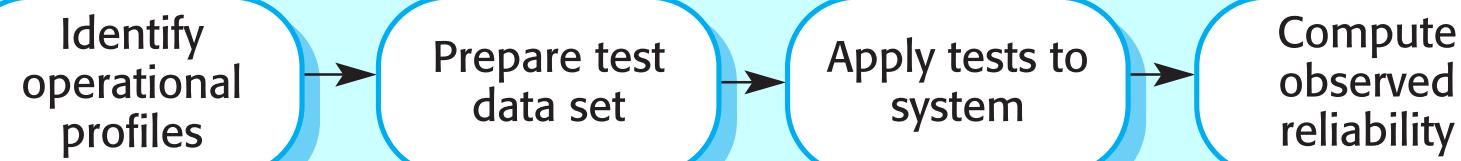
Validation costs

- Because of the additional activities involved, the validation costs for critical systems are usually significantly higher than for non-critical systems.
- Normally, V & V costs take up more than 50% of the total system development costs.

Reliability validation

- Reliability validation involves exercising the program to assess whether or not it has reached the required level of reliability.
- This cannot normally be included as part of a normal defect testing process because data for defect testing is (usually) atypical of actual usage data.
- Reliability measurement therefore requires a specially designed data set that replicates the pattern of inputs to be processed by the system.

The reliability measurement process



Reliability validation activities

- Establish the operational profile for the system.
- Construct test data reflecting the operational profile.
- Test the system and observe the number of failures and the times of these failures.
- Compute the reliability after a statistically significant number of failures have been observed.

Statistical testing

- Testing software for reliability rather than fault detection.
- Measuring the number of errors allows the reliability of the software to be predicted. Note that, for statistical reasons, more errors than are allowed for in the reliability specification must be induced.
- An acceptable level of reliability should be specified and the software tested and amended until that level of reliability is reached.

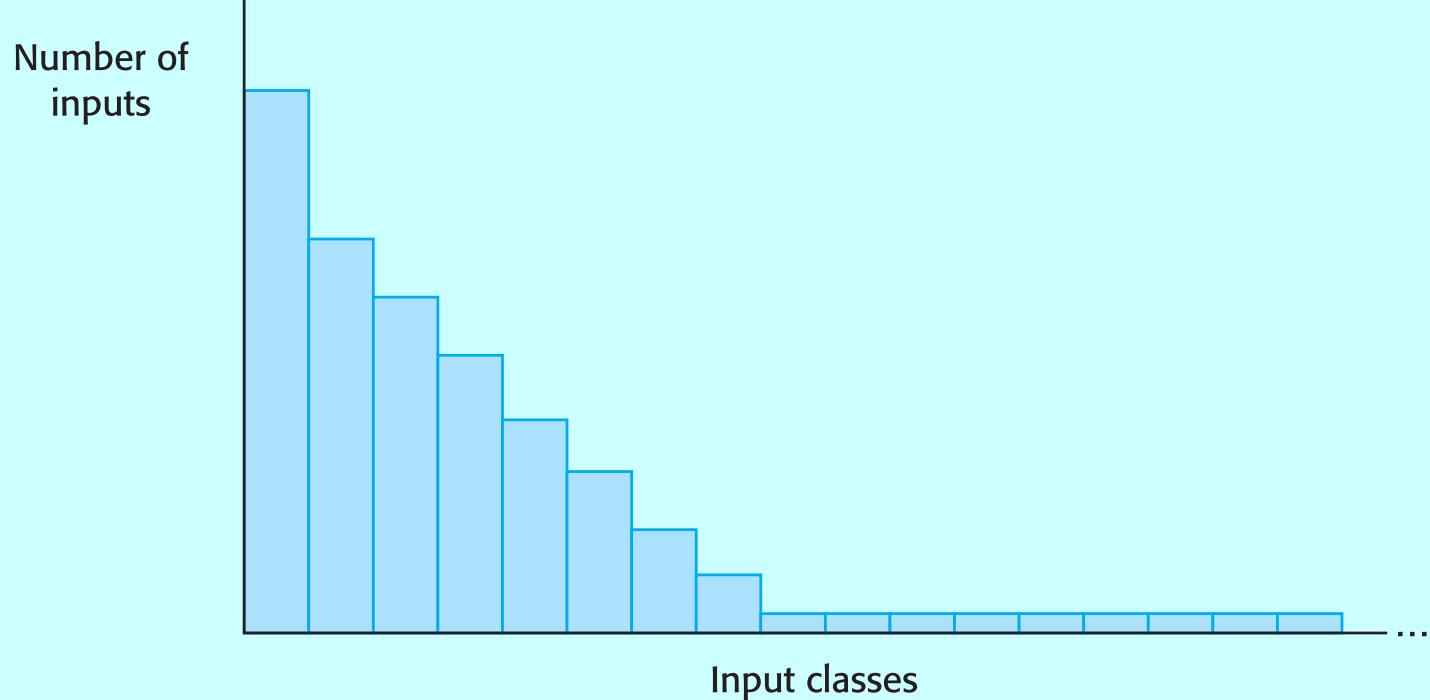
Reliability measurement problems

- Operational profile uncertainty
 - The operational profile may not be an accurate reflection of the real use of the system.
- High costs of test data generation
 - Costs can be very high if the test data for the system cannot be generated automatically.
- Statistical uncertainty
 - You need a statistically significant number of failures to compute the reliability but highly reliable systems will rarely fail.

Operational profiles

- An operational profile is a set of test data whose frequency matches the actual frequency of these inputs from ‘normal’ usage of the system. A close match with actual usage is necessary otherwise the measured reliability will not be reflected in the actual usage of the system.
- It can be generated from real data collected from an existing system or (more often) depends on assumptions made about the pattern of usage of a system.

An operational profile



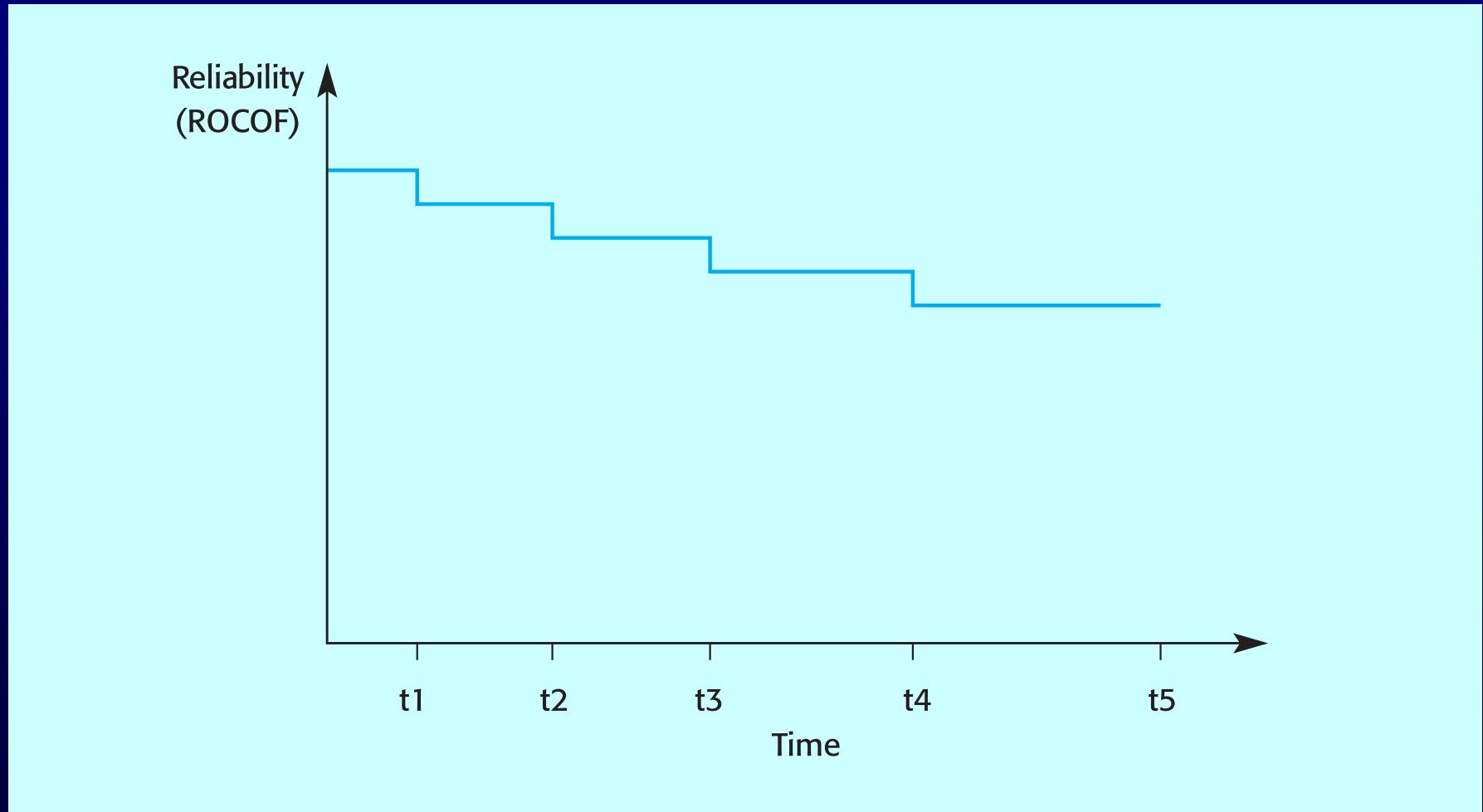
Operational profile generation

- Should be generated automatically whenever possible.
- Automatic profile generation is difficult for interactive systems.
- May be straightforward for ‘normal’ inputs but it is difficult to predict ‘unlikely’ inputs and to create test data for them.

Reliability prediction

- A reliability growth model is a mathematical model of the system reliability change as it is tested and faults are removed.
- It is used as a means of reliability prediction by extrapolating from current data
 - Simplifies test planning and customer negotiations.
 - You can predict when testing will be completed and demonstrate to customers whether or not the reliability growth will ever be achieved.
- Prediction depends on the use of statistical testing to measure the reliability of a system version.

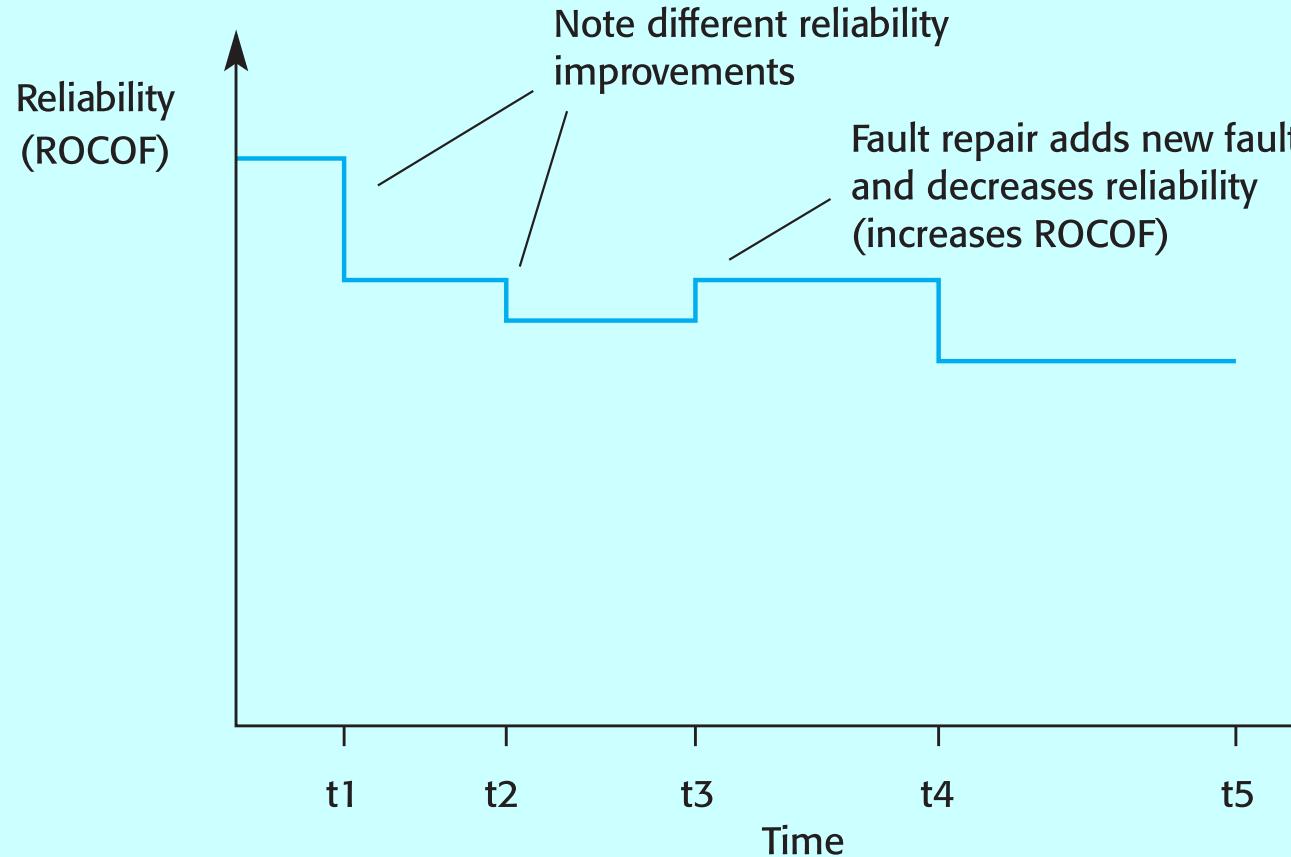
Equal-step reliability growth



Observed reliability growth

- The equal-step growth model is simple but it does not normally reflect reality.
- Reliability does not necessarily increase with change as the change can introduce new faults.
- The rate of reliability growth tends to slow down with time as frequently occurring faults are discovered and removed from the software.
- A random-growth model where reliability changes fluctuate may be a more accurate reflection of real changes to reliability.

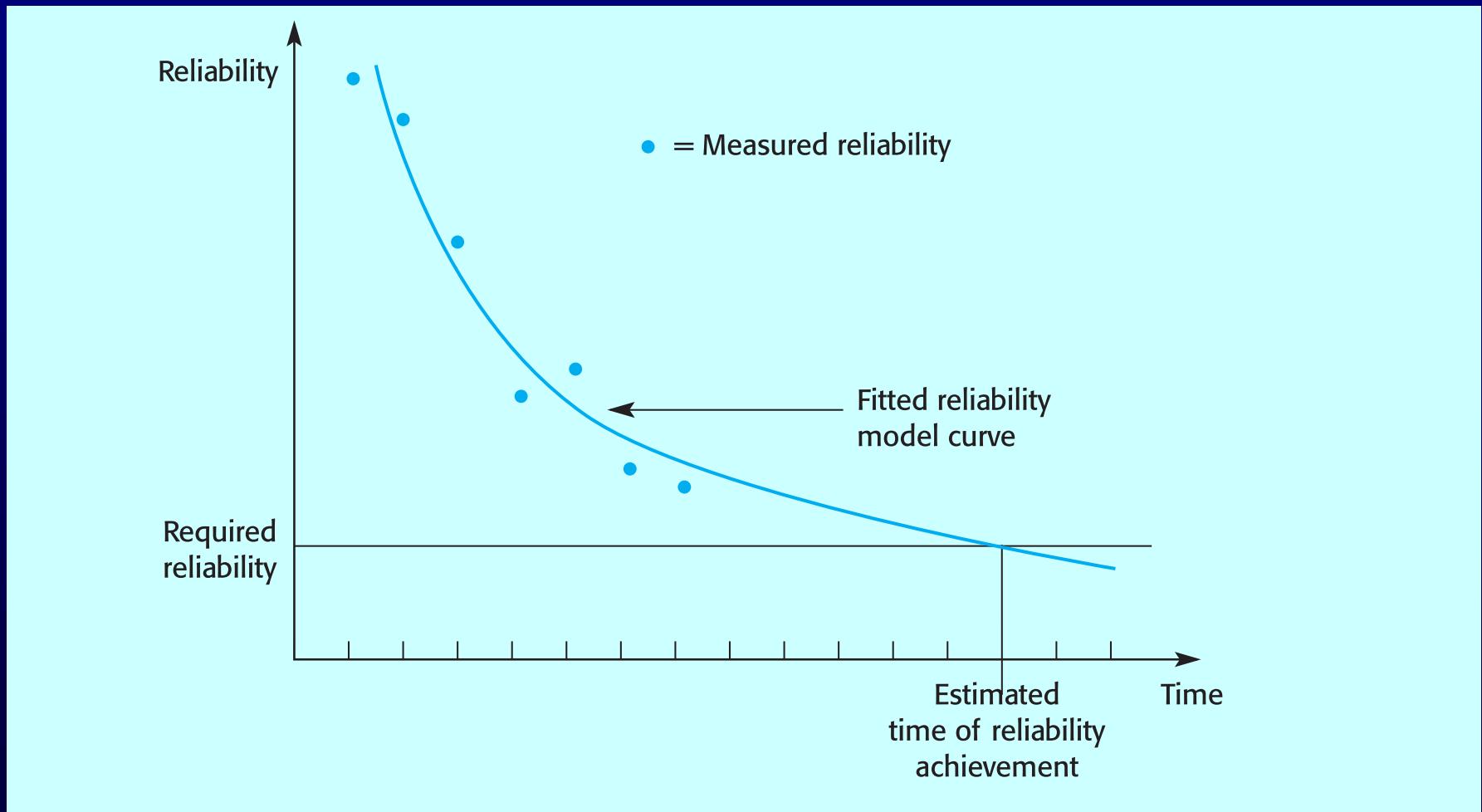
Random-step reliability growth



Growth model selection

- Many different reliability growth models have been proposed.
- There is no universally applicable growth model.
- Reliability should be measured and observed data should be fitted to several models.
- The best-fit model can then be used for reliability prediction.

Reliability prediction



Safety assurance

- Safety assurance and reliability measurement are quite different:
 - Within the limits of measurement error, you know whether or not a required level of reliability has been achieved;
 - However, quantitative measurement of safety is impossible. Safety assurance is concerned with establishing a confidence level in the system.

Safety confidence

- Confidence in the safety of a system can vary from very low to very high.
- Confidence is developed through:
 - Past experience with the company developing the software;
 - The use of dependable processes and process activities geared to safety;
 - Extensive V & V including both static and dynamic validation techniques.

Safety reviews

- Review for correct intended function.
- Review for maintainable, understandable structure.
- Review to verify algorithm and data structure design against specification.
- Review to check code consistency with algorithm and data structure design.
- Review adequacy of system testing.

Review guidance

- Make software as simple as possible.
- Use simple techniques for software development avoiding error-prone constructs such as pointers and recursion.
- Use information hiding to localise the effect of any data corruption.
- Make appropriate use of fault-tolerant techniques but do not be seduced into thinking that fault-tolerant software is necessarily safe.

Safety arguments

- Safety arguments are intended to show that the system cannot reach an unsafe state.
- These are weaker than correctness arguments which must show that the system code conforms to its specification.
- They are generally based on proof by contradiction
 - Assume that an unsafe state can be reached;
 - Show that this is contradicted by the program code.
- A graphical model of the safety argument may be developed.

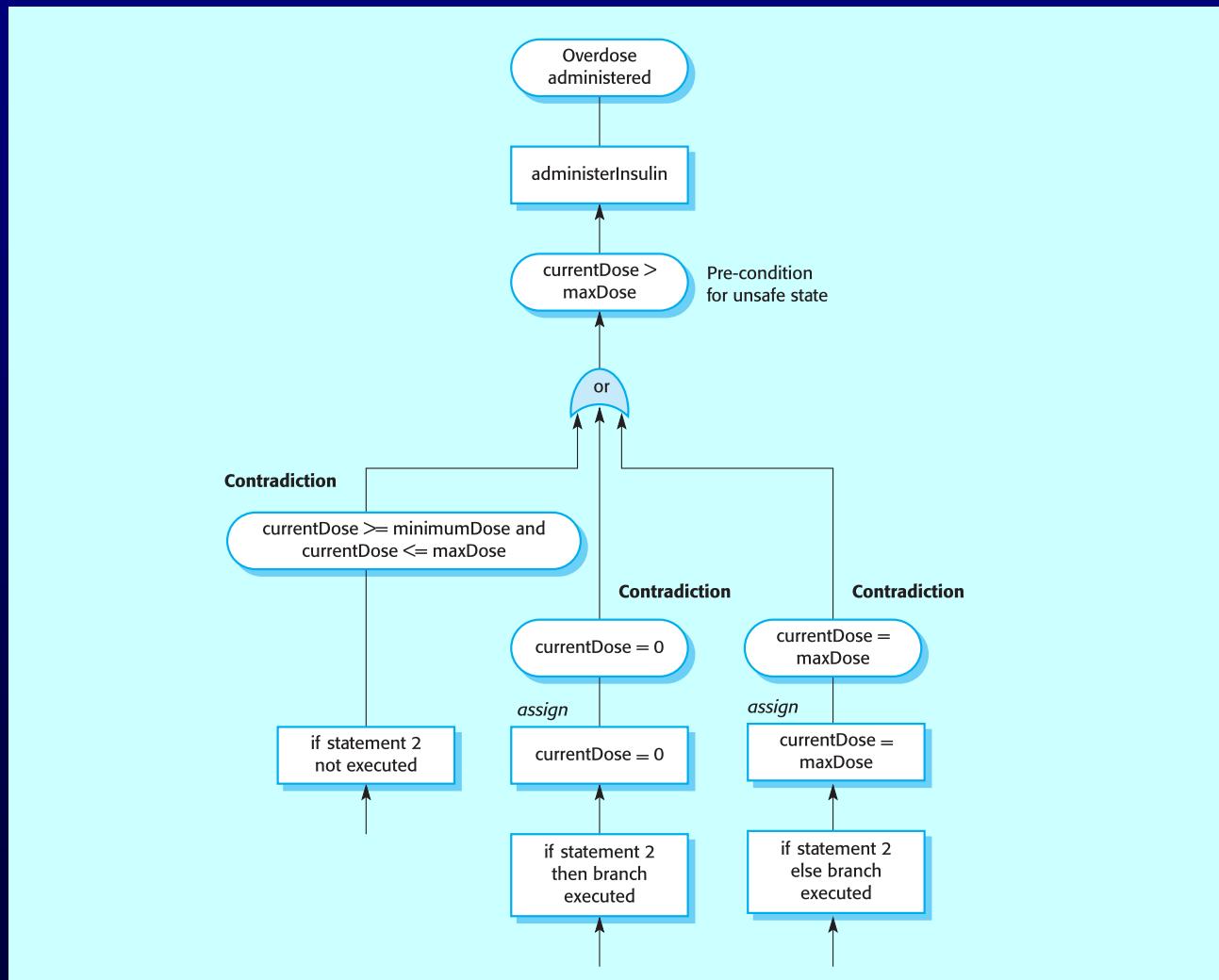
Construction of a safety argument

- Establish the safe exit conditions for a component or a program.
- Starting from the END of the code, work backwards until you have identified all paths that lead to the exit of the code.
- Assume that the exit condition is false.
- Show that, for each path leading to the exit that the assignments made in that path contradict the assumption of an unsafe exit from the component.

Insulin delivery code

```
currentDose = computeInsulin () ;  
// Safety check - adjust currentDose if necessary  
// if statement 1  
if (previousDose == 0)  
{  
    if (currentDose > 16)  
        currentDose = 16 ;  
}  
else  
    if (currentDose > (previousDose * 2) )  
        currentDose = previousDose * 2 ;  
// if statement 2  
if ( currentDose < minimumDose )  
    currentDose = 0 ;  
else if ( currentDose > maxDose )  
    currentDose = maxDose ;  
administerInsulin (currentDose) ;
```

Safety argument model



Program paths

- Neither branch of if-statement 2 is executed
 - Can only happen if CurrentDose is \geq minimumDose and \leq maxDose.
- then branch of if-statement 2 is executed
 - currentDose = 0.
- else branch of if-statement 2 is executed
 - currentDose = maxDose.
- In all cases, the post conditions contradict the unsafe condition that the dose administered is greater than maxDose.

Process assurance

- Process assurance involves defining a dependable process and ensuring that this process is followed during the system development.
- As discussed in Chapter 20, the use of a safe process is a mechanism for reducing the chances that errors are introduced into a system.
 - Accidents are rare events so testing may not find all problems;
 - Safety requirements are sometimes ‘shall not’ requirements so cannot be demonstrated through testing.

Safety related process activities

- Creation of a hazard logging and monitoring system.
- Appointment of project safety engineers.
- Extensive use of safety reviews.
- Creation of a safety certification system.
- Detailed configuration management (see Chapter 29).

Hazard analysis

- Hazard analysis involves identifying hazards and their root causes.
- There should be clear traceability from identified hazards through their analysis to the actions taken during the process to ensure that these hazards have been covered.
- A hazard log may be used to track hazards throughout the process.

Hazard log entry

Hazard Log.

System: Insulin Pump System

File: InsulinPump/Safety/HazardLog

Safety Engineer: James Brown

Log version: 1/3

Identified Hazard Insulin overdose delivered to patient

Identified by Jane Williams

Criticality class

Identified risk High

Fault tree identified YES *Date* 24.01.99 *Location* Hazard Log,
Page 5

Fault tree creators Jane Williams and Bill Smith

Fault tree checked YES Date 28.01.99 Checker James Brown

Page 4: Printed 20.02.20

System safety design requirements

1. The system shall include self-testing software that will test the sensor system, the clock and the insulin delivery system.
 2. The self-checking software shall be executed once per minute
 3. In the event of the self-checking software discovering a fault in any of the system components, an audible warning shall be issued and the pump display should indicate the name of the component where the fault has been discovered. The delivery of insulin should be suspended.
 4. The system shall incorporate an override system that allows the system user to modify the computed dose of insulin that is to be delivered by the system.
 5. The amount of override should be limited to be no greater than a pre-set value that is set when the system is configured by medical staff.

Run-time safety checking

- During program execution, safety checks can be incorporated as assertions to check that the program is executing within a safe operating ‘envelope’.
- Assertions can be included as comments (or using an assert statement in some languages). Code can be generated automatically to check these assertions.

Insulin administration with assertions

```
static void administerInsulin ( ) throws SafetyException {  
  
    int maxIncrements = InsulinPump.maxDose / 8 ;  
    int increments = InsulinPump.currentDose / 8 ;  
  
    // assert currentDose <= InsulinPump.maxDose  
  
    if (InsulinPump.currentDose > InsulinPump.maxDose)  
        throw new SafetyException (Pump.doseHigh);  
    else  
        for (int i=1; i<= increments; i++)  
        {  
            generateSignal () ;  
            if (i > maxIncrements)  
                throw new SafetyException ( Pump.incorrectIncrements);  
        } // for loop  
  
} //administerInsulin
```

Security assessment

- Security assessment has something in common with safety assessment.
- It is intended to demonstrate that the system cannot enter some state (an unsafe or an insecure state) rather than to demonstrate that the system can do something.
- However, there are differences
 - Safety problems are accidental; security problems are deliberate;
 - Security problems are more generic - many systems suffer from the same problems; Safety problems are mostly related to the application domain

Security validation

- **Experience-based validation**
 - The system is reviewed and analysed against the types of attack that are known to the validation team.
- **Tool-based validation**
 - Various security tools such as password checkers are used to analyse the system in operation.
- **Tiger teams**
 - A team is established whose goal is to breach the security of the system by simulating attacks on the system.
- **Formal verification**
 - The system is verified against a formal security specification.

Security checklist

1. Do all files that are created in the application have appropriate access permissions? The wrong access permissions may lead to these files being accessed by unauthorised users.
2. Does the system automatically terminate user sessions after a period of inactivity? Sessions that are left active may allow unauthorised access through an unattended computer.
3. If the system is written in a programming language without array bound checking, are there situations where buffer overflow may be exploited? Buffer overflow may allow attackers to send code strings to the system and then execute them.
4. If passwords are set, does the system check that password are strong? Strong passwords consist of mixed letters, numbers and punctuation and are not normal dictionary entries. They are more difficult to break than simple passwords.

Safety and dependability cases

- Safety and dependability cases are structured documents that set out detailed arguments and evidence that a required level of safety or dependability has been achieved.
- They are normally required by regulators before a system can be certified for operational use.

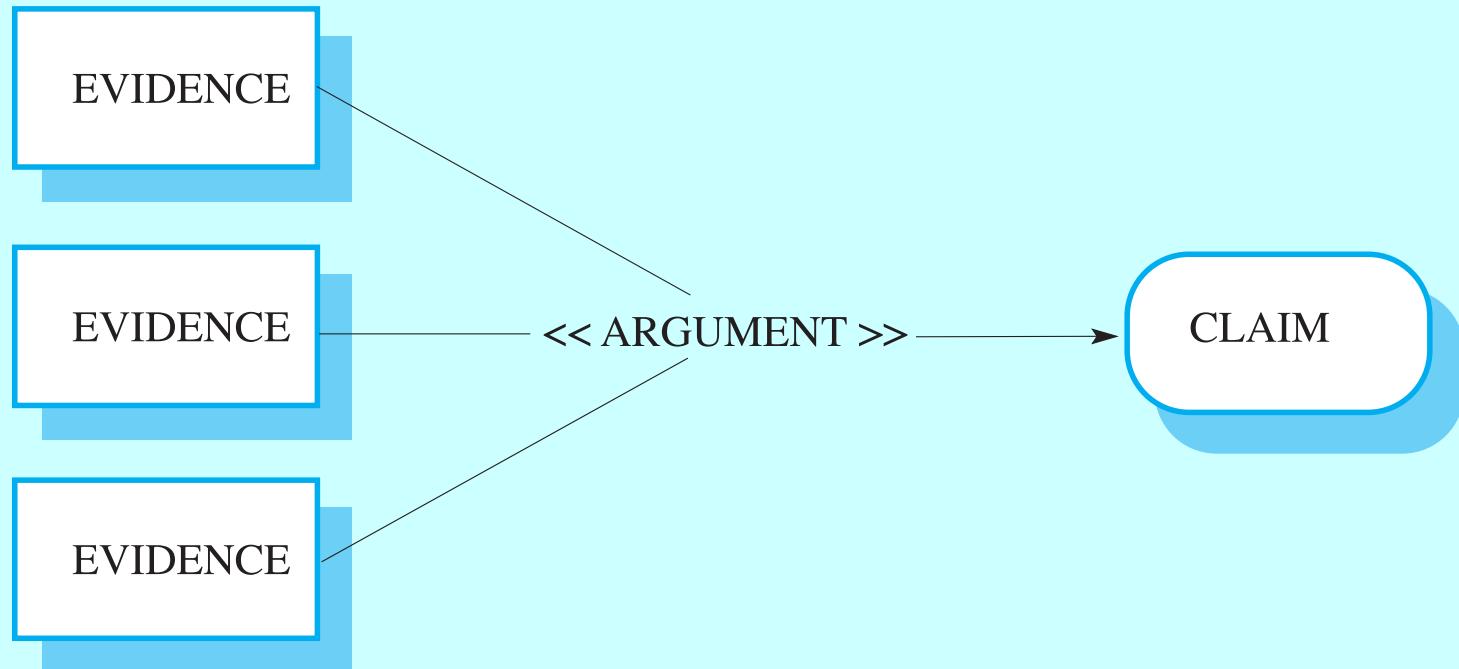
The system safety case

- It is now normal practice for a formal safety case to be required for all safety-critical computer-based systems e.g. railway signalling, air traffic control, etc.
- A safety case is:
 - A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment.
 - Arguments in a safety or dependability case can be based on formal proof, design rationale, safety proofs, etc. Process factors may also be included.

Components of a safety case

Component	Description
System description	An overview of the system and a description of its critical components.
Safety requirements	The safety requirements abstracted from the system requirements specification.
Hazard and risk analysis	Documents describing the hazards and risks that have been identified and the measures taken to reduce risk.
Design analysis	A set of structured arguments that justify why the design is safe.
Verification and validation	A description of the V & V procedures used and, where appropriate, the test plans for the system. Results of system V &V.
Review reports	Records of all design and safety reviews.
Team competences	Evidence of the competence of all of the team involved in safety-related systems development and validation.
Process QA	Records of the quality assurance processes carried out during system development.
Change management processes	Records of all changes proposed, actions taken and, where appropriate, justification of the safety of these changes.
Associated safety cases	References to other safety cases that may impact on this safety case.

Argument structure



Insulin pump argument

Claim:

The maximum single dose computed by the insulin pump will not exceed maxDose.

Evidence:

Safety argument for insulin pump as shown in Figure 24.7

Evidence:

Test data sets for insulin pump

Evidence:

Static analysis report for insulin pump software

Argument:

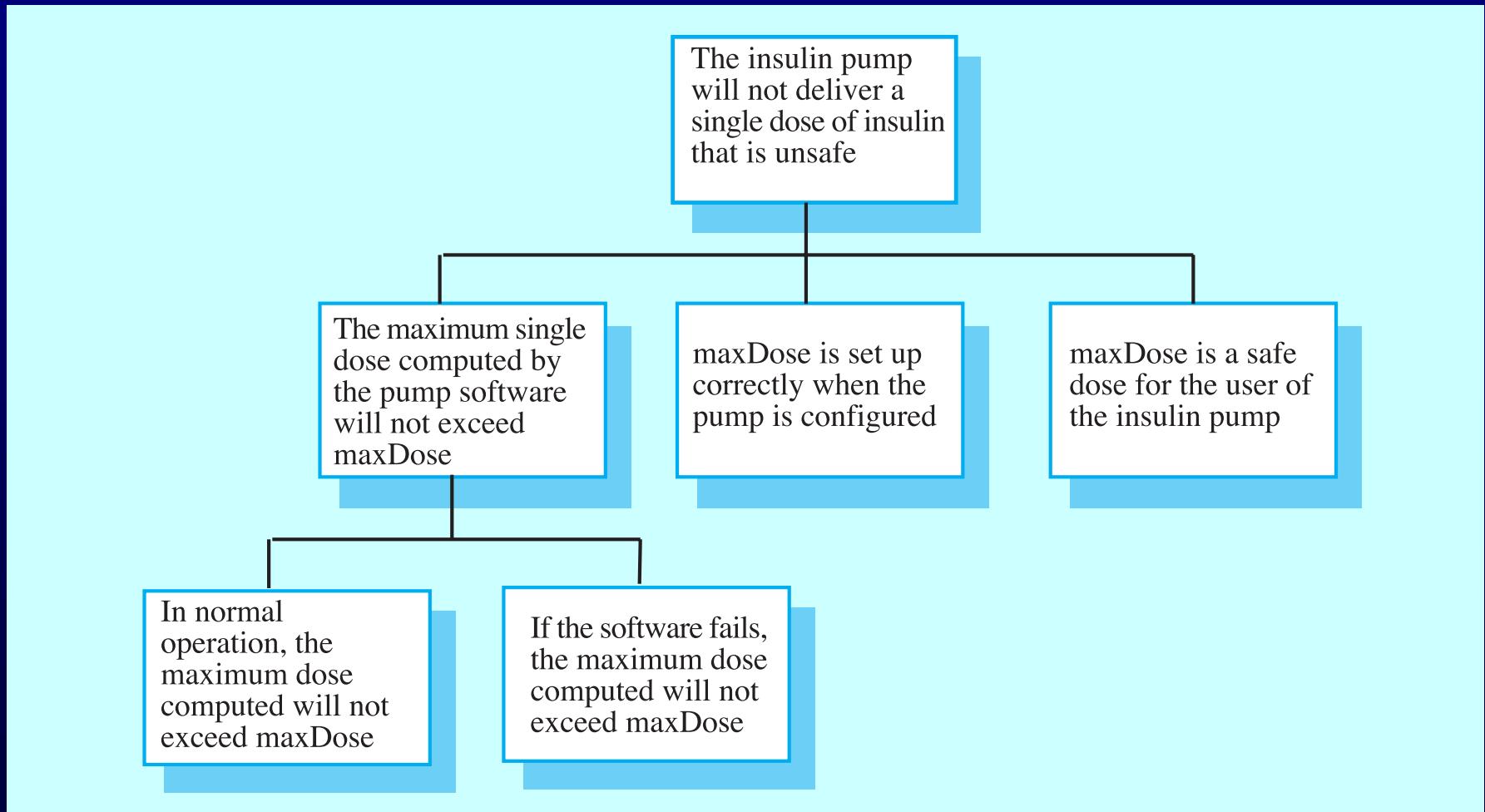
The safety argument presented shows that the maximum dose of insulin that can be computed is equal to maxDose.

In 400 tests, the value of Dose was correctly computed and never exceeded maxDose.

The static analysis of the control software revealed no anomalies.

Overall, it is reasonable to assume that the claim is justified.

Claim hierarchy



Key points

- Reliability measurement relies on exercising the system using an operational profile - a simulated input set which matches the actual usage of the system.
- Reliability growth modelling is concerned with modelling how the reliability of a software system improves as it is tested and faults are removed.
- Safety arguments or proofs are a way of demonstrating that a hazardous condition can never occur.

Key points

- It is important to have a dependable process for safety-critical systems development. The process should include hazard identification and monitoring activities.
- Security validation may involve experience-based analysis, tool-based analysis or the use of ‘tiger teams’ to attack the system.
- Safety cases collect together the evidence that a system is safe.

Security Engineering

Objectives

- To introduce issues that must be considered in the specification and design of secure software
- To discuss security risk management and the derivation of security requirements from a risk analysis
- To describe good design practice for secure systems development.
- To explain the notion of system survivability and to introduce a method of survivability analysis.

Topics covered

- Security concepts
- Security risk management
- Design for security
- System survivability

Security engineering

- Tools, techniques and methods to support the development and maintenance of systems that can resist malicious attacks that are intended to damage a computer-based system or its data.
- A sub-field of the broader field of computer security.

System layers

Application

Reusable components and libraries

Middleware

Database management

Generic, shared applications (Browsers, e-mail, etc)

Operating System

Application/infrastructure security

- Application security is a software engineering problem where the system is designed to resist attacks.
- Infrastructure security is a systems management problem where the infrastructure is configured to resist attacks.
- The focus of this chapter is application security.

Security concepts

Term	Definition
Asset	A system resource that has a value and has to be protected.
Exposure	The possible loss or harm that could result from a successful attack. This can be loss or damage to data or can be a loss of time and effort if recovery is necessary after a security breach.
Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm.
Attack	An exploitation of a system's vulnerability. Generally, this is from outside the system and is a deliberate attempt to cause some damage.
Threats	Circumstances that have potential to cause loss or harm. You can think of these as a system vulnerability that is subjected to an attack.
Control	A protective measure that reduces a system's vulnerability. Encryption would be an example of a control that reduced a vulnerability of a weak access control system.

Examples of security concepts

Term	Definition
Asset	The records of each patient that is receiving or has received treatment.
Exposure	Potential financial loss from future patients who do not seek treatment because they do not trust the clinic to maintain their data. Financial loss from legal action by the sports star. Loss of reputation.
Vulnerability	A weak password system which makes it easy for users to set guessable passwords. User ids that are the same as names.
Attack	An impersonation of an authorised user.
Threat	An unauthorised user will gain access to the system by guessing the credentials (login name and password) of an authorised user.
Control	A password checking system that disallows passwords that are set by users which are proper names or words that are normally included in a dictionary.

Security threats

- Threats to the confidentiality of a system or its data
- Threats to the integrity of a system or its data
- Threats to the availability of a system or its data

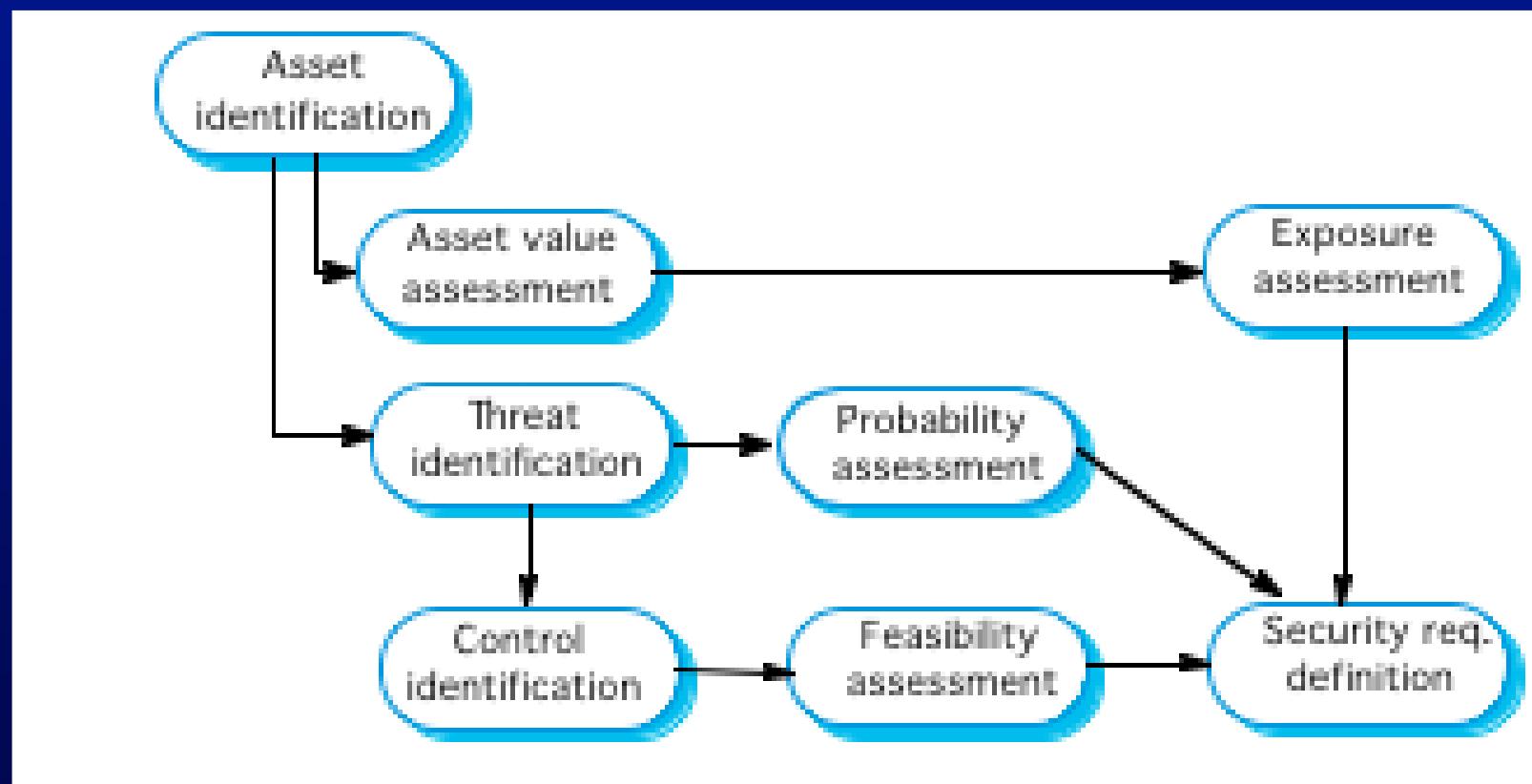
Security controls

- Controls that are intended to ensure that attacks are unsuccessful. This is analogous to fault avoidance.
- Controls that are intended to detect and repel attacks. This is analogous to fault detection and tolerance.
- Controls that are intended to support recovery from problems. This is analogous to fault recovery.

Security risk management

- Risk management is concerned with assessing the possible losses that might ensue from attacks on the system and balancing these losses against the costs of security procedures that may reduce these losses.
- Risk management should be driven by an organisational security policy.
- Risk management involves
 - Preliminary risk assessment
 - Life cycle risk assessment

Preliminary risk assessment



Asset analysis

Asset	Value	Exposure
The information system	High. Required to support all clinical consultations. Potentially safety critical.	High. Financial loss as clinics may have to be cancelled. Costs of restoring system. Possible patient harm if treatment cannot be prescribed.
The patient database	High. Required to support all clinical consultations. Potentially safety critical.	High. Financial loss as clinics may have to be cancelled. Costs of restoring system. Possible patient harm if treatment cannot be prescribed.
An individual patient record	Normally low although may be high for specific high-profile patients	Low direct losses but possible loss of reputation.

Threat and control analysis

Threat	Probability	Control	Feasibility
Unauthorised user gains access as system manager and makes system unavailable	Low	Only allow system management from specific locations which are physically secure.	Low cost of implementation but care must be taken with key distribution and to ensure that keys are available in the event of an emergency.
Unauthorised user gains access as system user and accesses confidential information	High	Require all users to authenticate themselves using biometric mechanism. Log all changes to patient information to track system usage.	Technically feasible but high cost solution. Possible user resistance. Simple and transparent to implement and also supports recovery.

Security requirements

- Patient information must be downloaded at the start of a clinic session to a secure area on the system client that is used by clinical staff.
- Patient information must not be maintained on system clients after a clinic session has finished.
- A log on a separate computer from the database server must be maintained of all changes made to the system database.

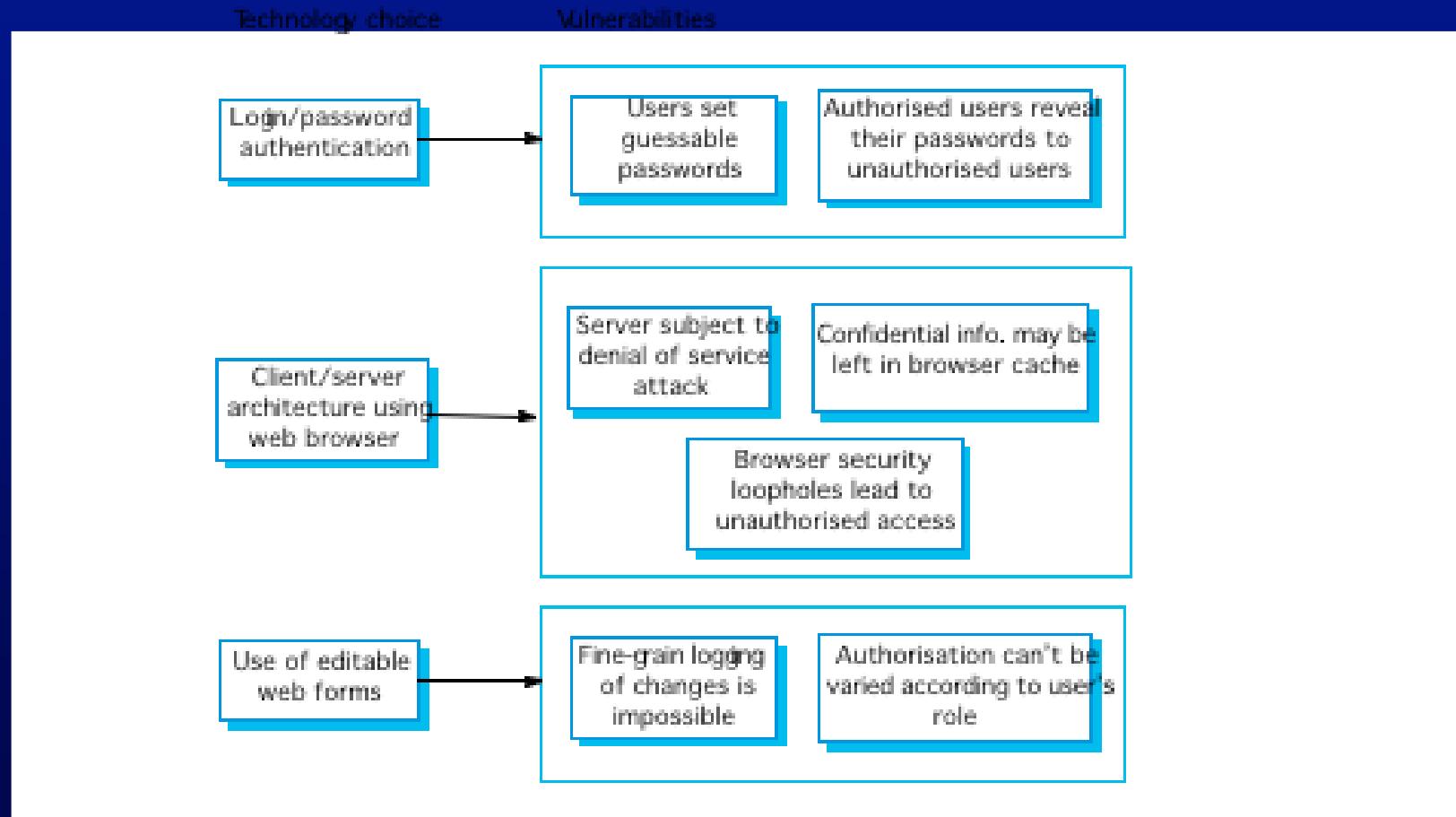
Life cycle risk assessment

- Risk assessment while the system is being developed and after it has been deployed
- More information is available - system platform, middleware and the system architecture and data organisation.
- Vulnerabilities that arise from design choices may therefore be identified.

Examples of design decisions

- System users authenticated using a name/password combination.
- The system architecture is client-server with clients accessing the system through a standard web browser.
- Information is presented as an editable web form.

Technology vulnerabilities



Design for security

- Architectural design - how do architectural design decisions affect the security of a system?
- Good practice - what is accepted good practice when designing secure systems?
- Design for deployment - what support should be designed into a system to avoid the introduction of vulnerabilities when a system is deployed for use?

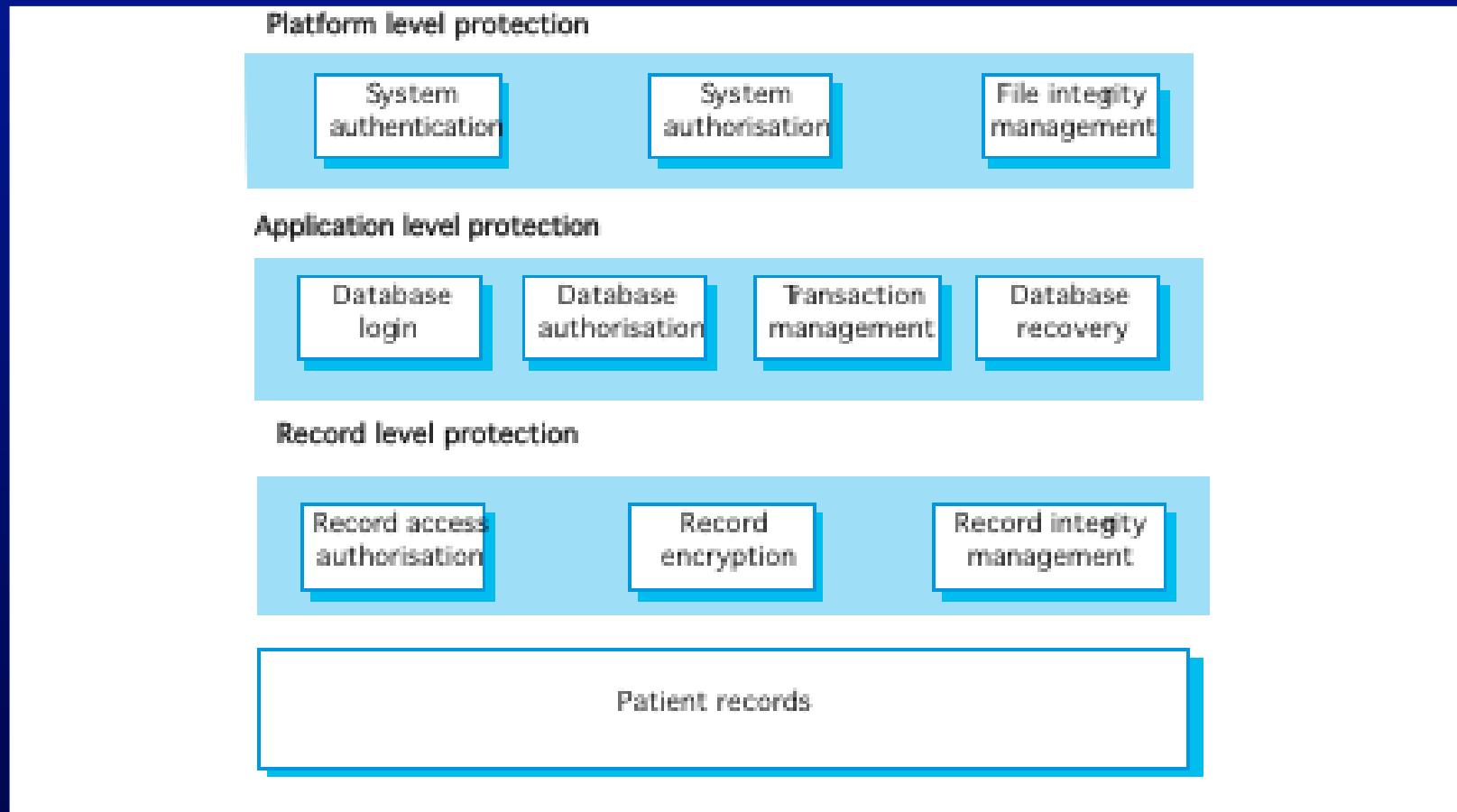
Architectural design

- Protection
 - How should the system be organised so that critical assets can be protected against external attack?
- Distribution
 - How should system assets be distributed so that the effects of a successful attack are minimised?
- Potentially conflicting
 - If assets are distributed, then they are more expensive to protect.

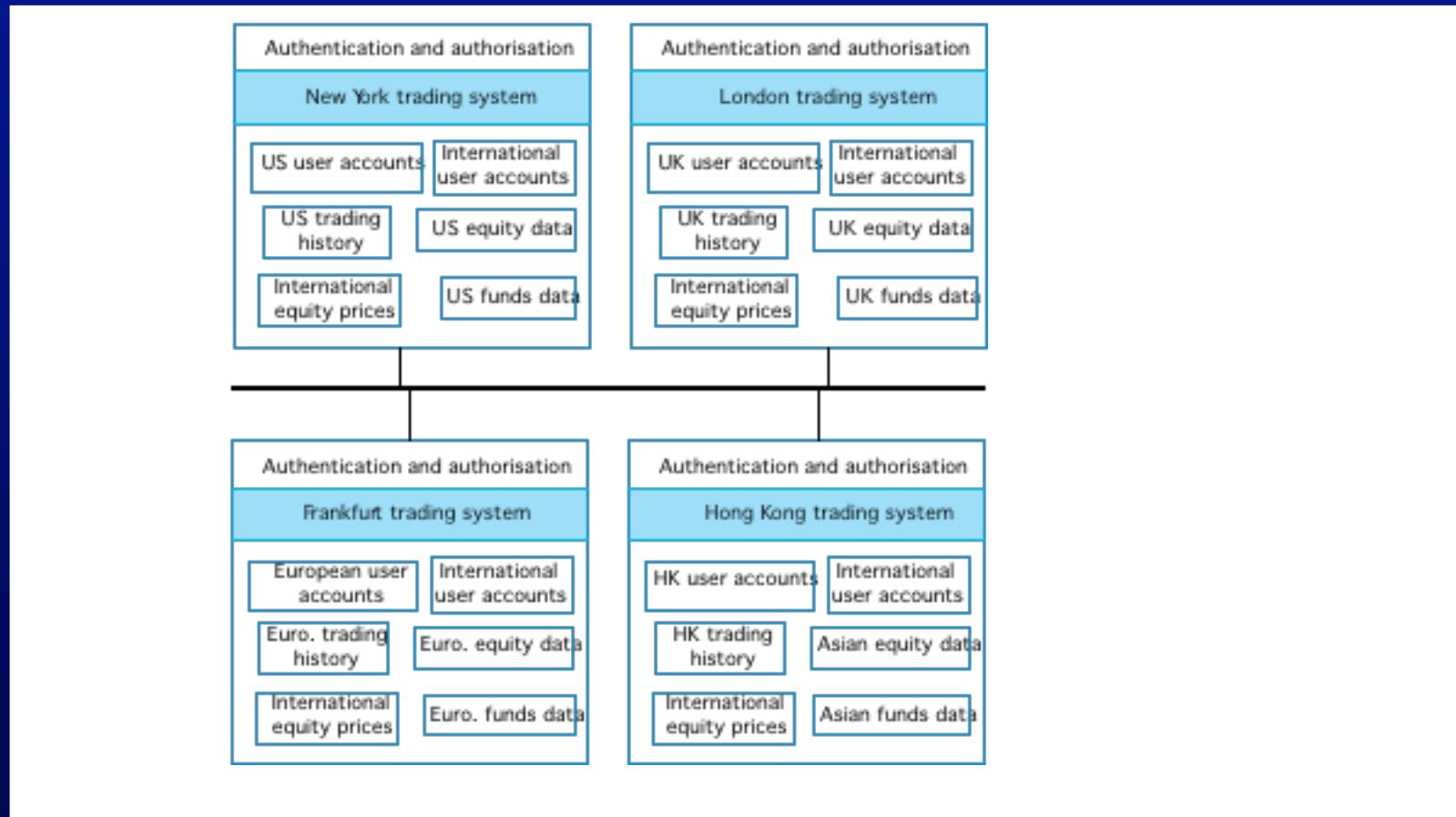
Protection

- Platform-level protection
- Application-level protection
- Record-level protection

Layered protection



A distributed equity system



Design guidelines

- Design guidelines encapsulate good practice in secure systems design
- Design guidelines serve two purposes:
 - They raise awareness of security issues in a software engineering team.
 - They can be used as the basis of a review checklist that is applied during the system validation process.

Design guidelines 1

- Base security decisions on an explicit security policy
- Avoid a single point of failure
- Fail securely
- Balance security and usability
- Be aware of the possibilities of social engineering

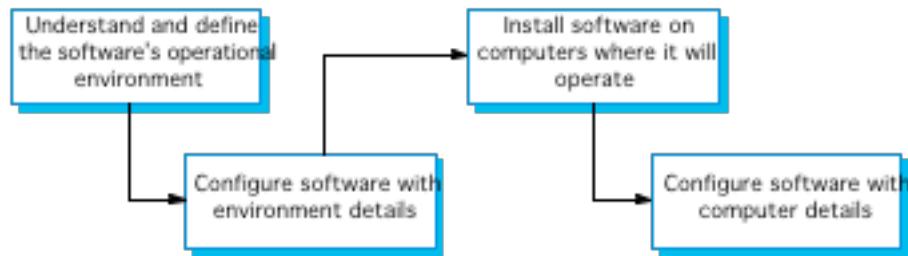
Design guidelines 2

- Use redundancy and diversity to reduce risk
- Validate all inputs
- Compartmentalise your assets
- Design for deployment
- Design for recoverability

Design for deployment

- Deployment involves configuring software to operate in its working environment, installing the system and configuring it for the operational platform.
- Vulnerabilities may be introduced at this stage as a result of configuration mistakes.
- Designing deployment support into the system can reduce the probability that vulnerabilities will be introduced.

System deployment



Deployment support

- Include support for viewing and analysing configurations
- Minimise default privileges and thus limit the damage that might be caused
- Localise configuration settings
- Provide easy ways to fix security vulnerabilities

System survivability

- Survivability is an emergent system property that reflects the systems ability to deliver essential services whilst it is under attack or after part of the system has been damaged
- Survivability analysis and design should be part of the security engineering process

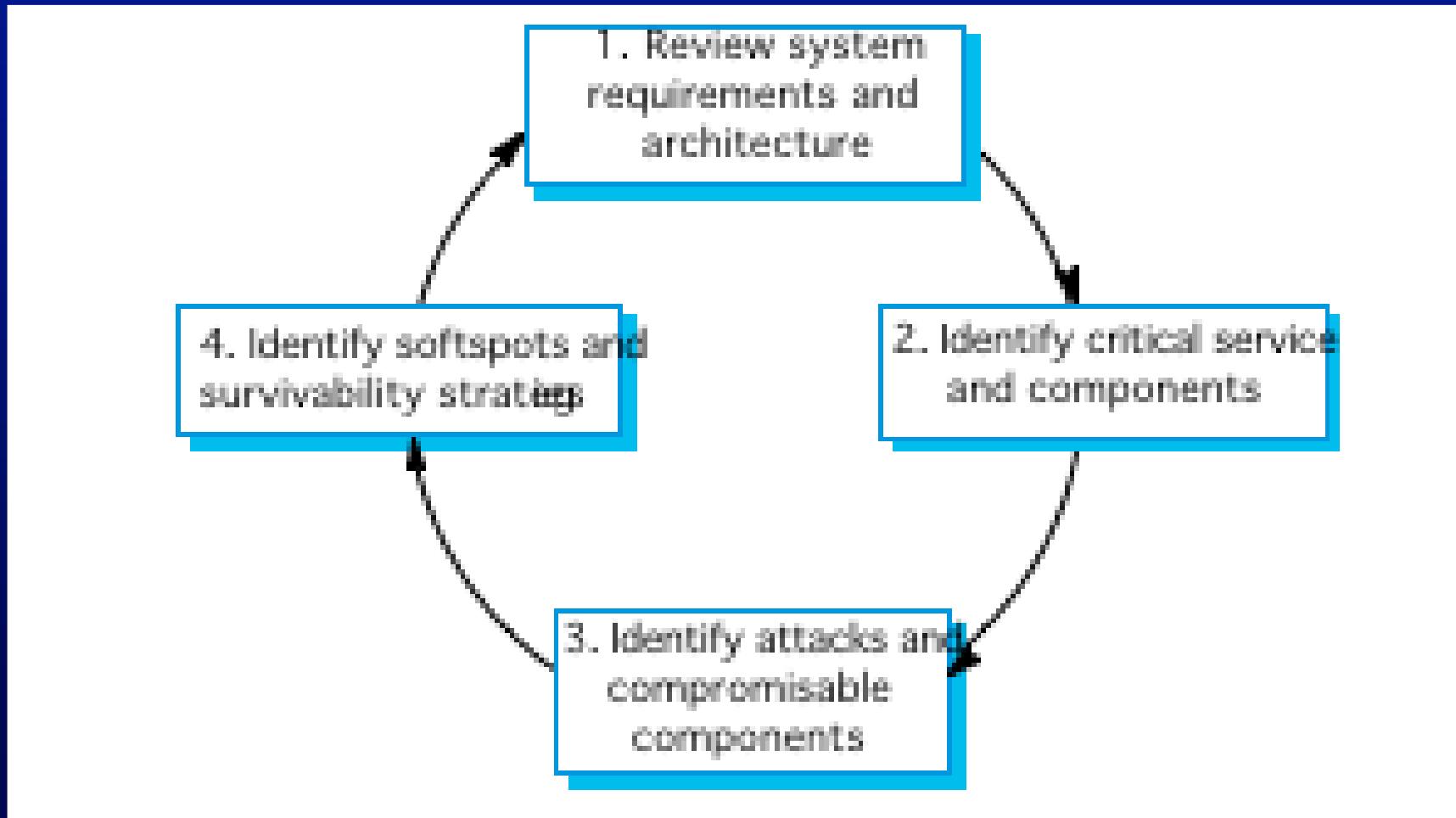
Service availability

- Which system services are the most critical for a business?
- How might these services be compromised?
- What is the minimal quality of service that must be maintained?
- How can these services be protected?
- If a service becomes unavailable, how quickly can it be recovered?

Survivability strategies

- Resistance
 - Avoiding problems by building capabilities into the system to resist attacks
- Recognition
 - Detecting problems by building capabilities into the system to detect attacks and failures and assess the resultant damage
- Recovery
 - Tolerating problems by building capabilities into the system to deliver services whilst under attack

System survivability method



Key activities

- System understanding
 - Review golas, requirements and architecture
- Critical service identification
 - Identify services that must be maintained
- Attack simulation
 - Devise attack scenarios and identify components affected
- Survivability analysis
 - Identify survivability strategies to be applied

Trading system survivability

- User accounts and equity prices replicated across servers so some provision for survivability made
- Key capability to be maintained is the ability to place orders for stock
- Orders must be accurate and reflect the actual sales/purchases made by a trader

Survivability analysis

Attack	Resistance	Recognition	Recovery
Unauthorised user places malicious orders	Require dealing password to place orders that is different from login password.	Send copy of order by email to authorised user with contact phone number (so that they can detect malicious orders) Maintain user's order history and check for unusual trading patterns.	Provide mechanism to automatically 'undo' trades and restore user accounts. Refund users for losses that are due to malicious trading. Insure against consequential losses.
Corruption of transactions database	Require privileged users to be authorised using a stronger authentication mechanism, such as digital certificates.	Maintain read-only copies of transactions for an office on an international server. Periodically compare transactions to check for corruption. Maintain cryptographic checksum with all transaction records to detect corruption.	Recover database from backup copies. Provide a mechanism to replay trades from a specified time to recreate transactions database.

Key points

- Security engineering is concerned with how to develop systems that can resist malicious attacks
- Security threats can be threats to confidentiality, integrity or availability of a system or its data
- Security risk management is concerned with assessing possible losses from attacks and deriving security requirements to minimise losses
- Design for security involves architectural design, following good design practice and minimising the introduction of system vulnerabilities

Key points

- Key issues when designing a secure architecture include organising the structure to protect assets and distributing assets to minimise losses
- General security guidelines sensitise designers to security issues and serve as review checklists
- Configuration visualisation, setting localisation, and minimisation of default privileges help reduce deployment errors
- System survivability reflects the ability of a system to deliver services whilst under attack or after part of the system has been damaged.