

Introduction

- Getting started with software engineering

Objectives

- To introduce software engineering and to explain its importance
- To set out the answers to key questions about software engineering
- To introduce ethical and professional issues and to explain why they are of concern to software engineers

Topics covered

- FAQs about software engineering
- Professional and ethical responsibility

Software engineering

- The economies of ALL developed nations are dependent on software
- More and more systems are software controlled
- Software engineering is concerned with theories, methods and tools for professional software development
- Software engineering expenditure represents a significant fraction of GNP in all developed countries

Software costs

- Software costs often dominate system costs. The costs of software on a PC are often greater than the hardware cost
- Software costs more to maintain than it does to develop. For systems with a long life, maintenance costs may be several times development costs
- Software engineering is concerned with cost-effective software development

FAQs about software engineering

- What is software?
- What is software engineering?
- What is the difference between software engineering and computer science?
- What is the difference between software engineering and system engineering?
- What is a software process?
- What is a software process model?

FAQs about software engineering

- What are the costs of software engineering?
- What are software engineering methods?
- What is CASE (Computer-Aided Software Engineering)
- What are the attributes of good software?
- What are the key challenges facing software engineering?

What is software?

- Computer programs and associated documentation
- 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
 - Bespoke (custom) - developed for a single customer according to their specification

What is software engineering?

- Software engineering is an engineering discipline which 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 currently insufficient to act as a complete underpinning for software 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
- 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
 - Evolutionary development
 - Formal transformation
 - Integration from reusable components

What are the costs of software engineering?

- Roughly 60% of costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs
- Costs vary depending on the type of system being developed and the requirements of system attributes such as performance and system reliability
- Distribution of costs depends on the development model that is used

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 which 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 usable
- Maintainability
 - Software must evolve to meet changing needs
- Dependability
 - Software must be trustworthy
- Efficiency
 - Software should not make wasteful use of system resources
- Usability
 - Software must be usable by the users for which it was designed

What are the key challenges facing software engineering?

- Coping with legacy systems, coping with increasing diversity and coping with demands for reduced delivery times
- Legacy systems
 - Old, valuable systems must be maintained and updated
- Heterogeneity
 - Systems are distributed and include a mix of hardware and software
- Delivery
 - There is increasing pressure for faster delivery of software

Professional and ethical responsibility

- Software engineering involves wider responsibilities than simply the application of technical skills
- Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals
- Ethical behaviour is more than simply upholding the law.

Issues of professional responsibility

- *Confidentiality*

- Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.

- *Competence*

- Engineers should not misrepresent their level of competence. They should not knowingly accept work which is outwith their competence.

Issues of professional responsibility

- *Intellectual property rights*
 - Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.
- *Computer misuse*
 - Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

ACM/IEEE Code of Ethics

- The professional societies in the US have cooperated to produce a code of ethical practice.
- Members of these organisations sign up to the code of practice when they join.
- The Code contains eight Principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession.

Code of ethics - preamble

- **Preamble**

- The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.
- Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

Code of ethics - principles

- 1. PUBLIC
 - Software engineers shall act consistently with the public interest.
- 2. CLIENT AND EMPLOYER
 - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
- 3. PRODUCT
 - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

Code of ethics - principles

- JUDGMENT

- Software engineers shall maintain integrity and independence in their professional judgment.

- 5. MANAGEMENT

- Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

- 6. PROFESSION

- Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

Code of ethics - principles

- 7. COLLEAGUES

- Software engineers shall be fair to and supportive of their colleagues.

- 8. SELF

- Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Ethical dilemmas

- Disagreement in principle with the policies of senior management
- Your employer acts in an unethical way and releases a safety-critical system without finishing the testing of the system
- Participation in the development of military weapons systems or nuclear systems

Key points

- Software engineering is an engineering discipline which 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 which 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.

Systems Engineering

- Designing, implementing, deploying and operating systems which include hardware, software and people

Objectives

- To explain why system software is affected by broader system engineering issues
- To introduce the concept of emergent system properties such as reliability and security
- To explain why the systems environment must be considered in the system design process
- To explain system engineering and system procurement processes

Topics covered

- Emergent system properties
- Systems and their environment
- System modelling
- The system engineering process
- System procurement

What is a system?

- A purposeful collection of inter-related components working together towards some common objective.
- A system may include software, mechanical, electrical and electronic hardware and be operated by people.
- System components are dependent on other system components
- The properties and behaviour of system components are inextricably inter-mingled

Problems of systems engineering

- Large systems are usually designed to solve 'wicked' problems
- Systems engineering requires a great deal of co-ordination across disciplines
 - Almost infinite possibilities for design trade-offs across components
 - Mutual distrust and lack of understanding across engineering disciplines
- Systems must be designed to last many years in a changing environment

Software and systems engineering

- The proportion of software in systems is increasing. Software-driven general purpose electronics is replacing special-purpose systems
- Problems of systems engineering are similar to problems of software engineering
- Software is (unfortunately) seen as a problem in systems engineering. Many large system projects have been delayed because of software problems

Emergent properties

- Properties of the system as a whole rather than properties that can be derived from the properties of components of a system
- Emergent properties are a consequence of the relationships between system components
- They can therefore only be assessed and measured once the components have been integrated into a system

Examples of emergent properties

- *The overall weight of the system*
 - This is an example of an emergent property that can be computed from individual component properties.
- *The reliability of the system*
 - This depends on the reliability of system components and the relationships between the components.
- *The usability of a system*
 - This is a complex property which is not simply dependent on the system hardware and software but also depends on the system operators and the environment where it is used.

Types of emergent property

- Functional properties
 - These appear when all the parts of a system work together to achieve some objective. For example, a bicycle has the functional property of being a transportation device once it has been assembled from its components.
- Non-functional emergent properties
 - Examples are reliability, performance, safety, and security. These relate to the behaviour of the system in its operational environment. They are often critical for computer-based systems as failure to achieve some minimal defined level in these properties may make the system unusable.

System reliability engineering

- Because of component inter-dependencies, faults can be propagated through the system
- System failures often occur because of unforeseen inter-relationships between components
- It is probably impossible to anticipate all possible component relationships
- Software reliability measures may give a false picture of the system reliability

Influences on reliability

- *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 failure is usually distinct from hardware failure in that software does not wear out.
- *Operator reliability*
 - How likely is it that the operator of a system will make an error?

Reliability relationships

- Hardware failure can generate spurious signals that are outside the range of inputs expected by the software
- Software errors can cause alarms to be activated which cause operator stress and lead to operator errors
- The environment in which a system is installed can affect its reliability

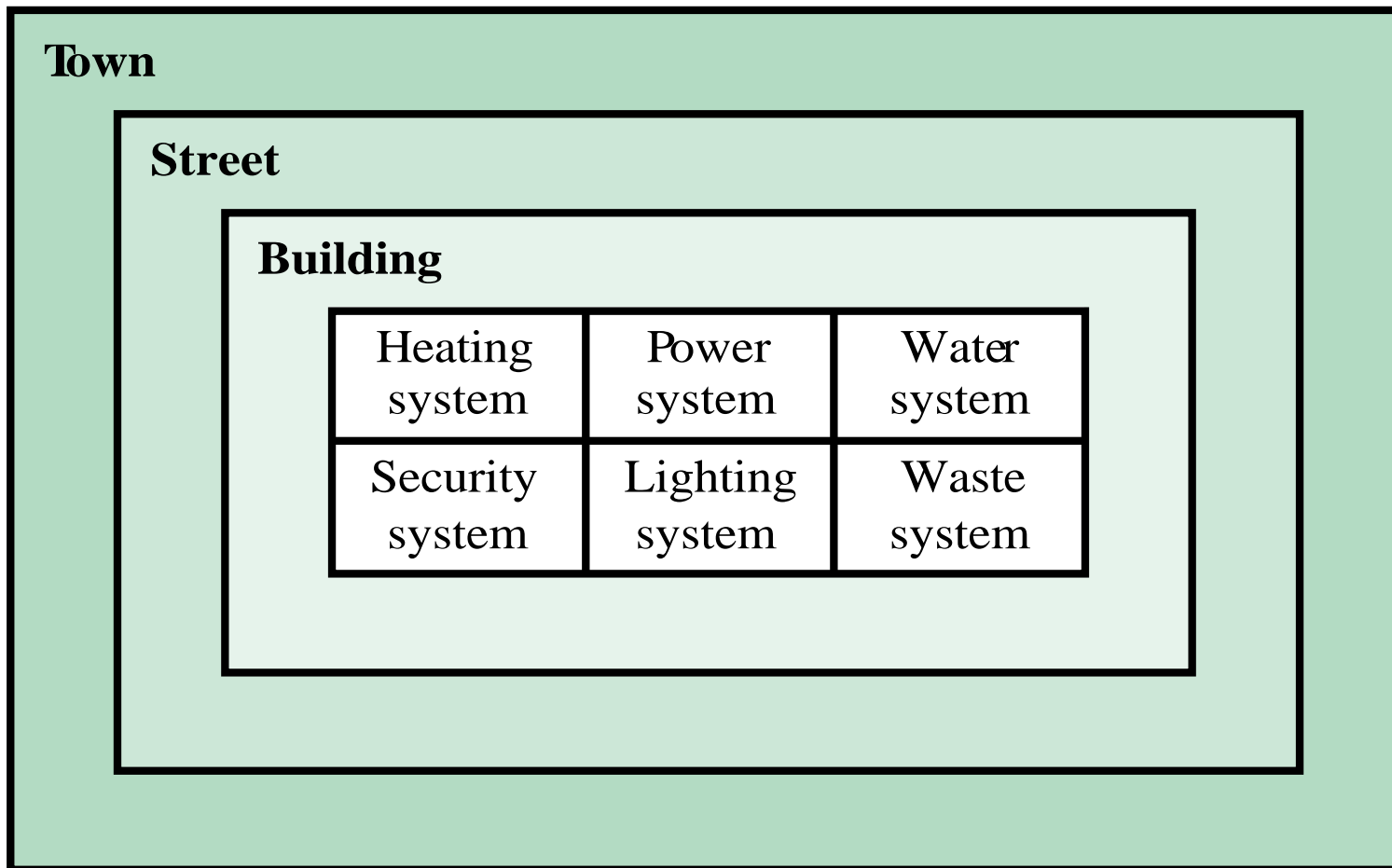
The ‘shall-not’ properties

- Properties such as performance and reliability can be measured
- However, some properties are properties that the system should not exhibit
 - Safety - the system should not behave in an unsafe way
 - Security - the system should not permit unauthorised use
- Measuring or assessing these properties is very hard

Systems and their environment

- Systems are not independent but exist in an environment
- System's function may be to change its environment
- Environment affects the functioning of the system
e.g. system may require electrical supply from its environment
- The organizational as well as the physical environment may be important

System hierarchies



Human and organisational factors

- *Process changes*

- Does the system require changes to the work processes in the environment?

- *Job changes*

- Does the system de-skill the users in an environment or cause them to change the way they work?

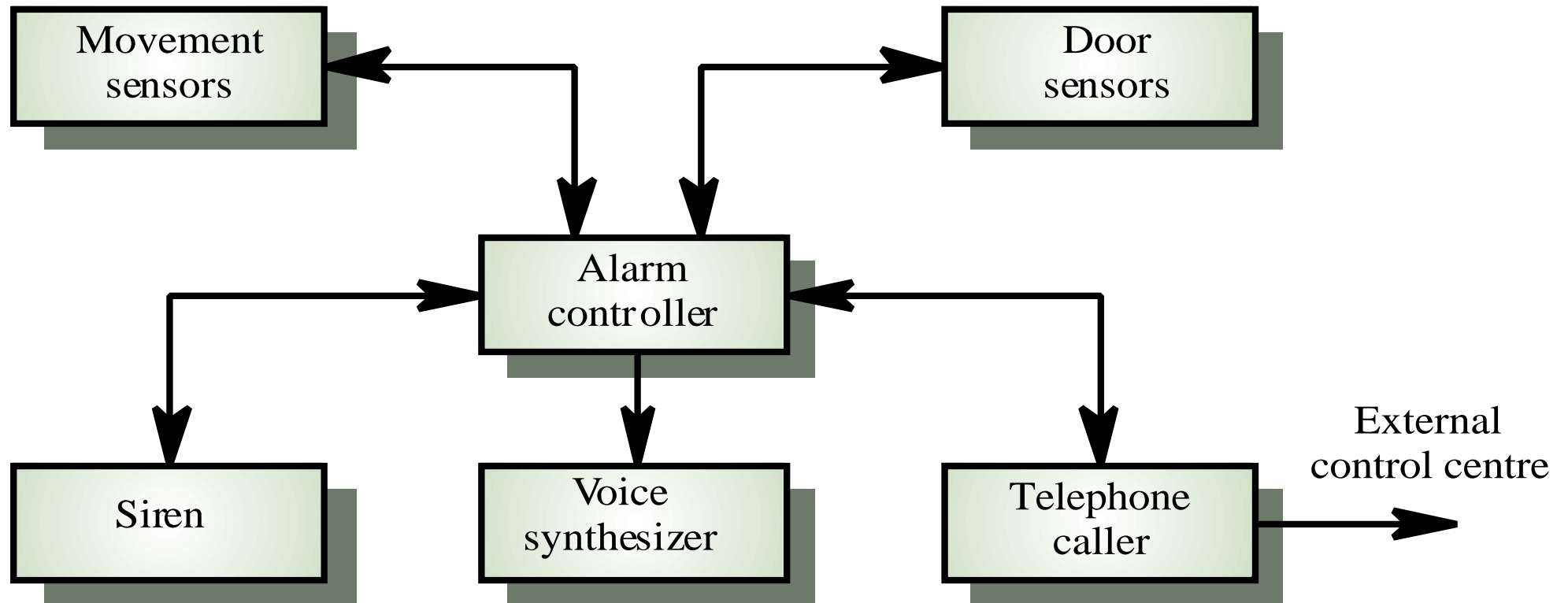
- *Organisational changes*

- Does the system change the political power structure in an organisation?

System architecture modelling

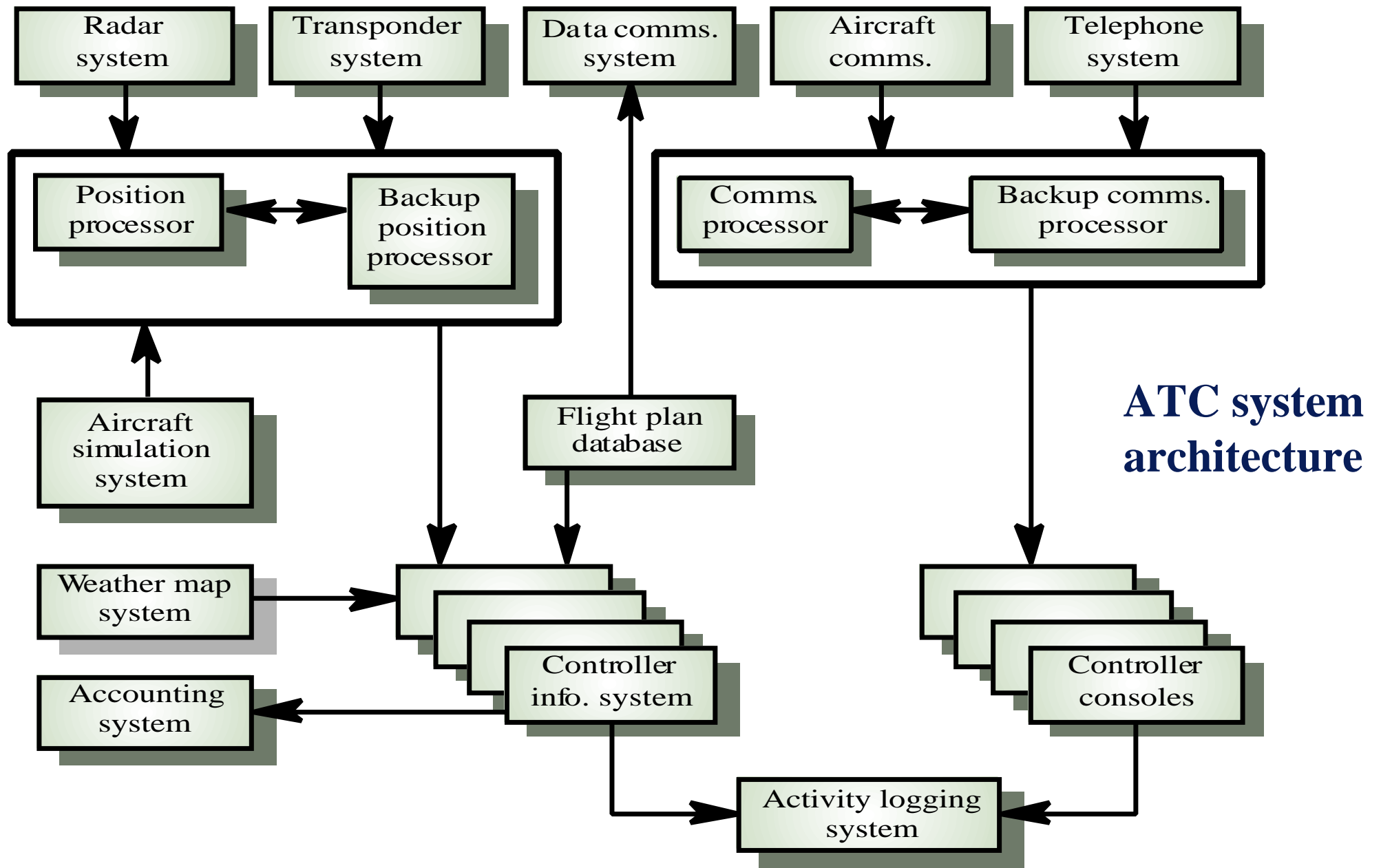
- An architectural model presents an abstract view of the sub-systems making up a system
- May include major information flows between sub-systems
- Usually presented as a block diagram
- May identify different types of functional component in the model

Intruder alarm system



Component types in alarm system

- Sensor
 - Movement sensor, door sensor
- Actuator
 - Siren
- Communication
 - Telephone caller
- Co-ordination
 - Alarm controller
- Interface
 - Voice synthesizer



Functional system components

- Sensor components
- Actuator components
- Computation components
- Communication components
- Co-ordination components
- Interface components

System components

- Sensor components
 - Collect information from the system's environment e.g. radars in an air traffic control system
- Actuator components
 - Cause some change in the system's environment e.g. valves in a process control system which increase or decrease material flow in a pipe
- Computation components
 - Carry out some computations on an input to produce an output e.g. a floating point processor in a computer system

System components

- Communication components
 - Allow system components to communicate with each other e.g. network linking distributed computers
- Co-ordination components
 - Co-ordinate the interactions of other system components e.g. scheduler in a real-time system
- Interface components
 - Facilitate the interactions of other system components e.g. operator interface
- All components are now usually software controlled

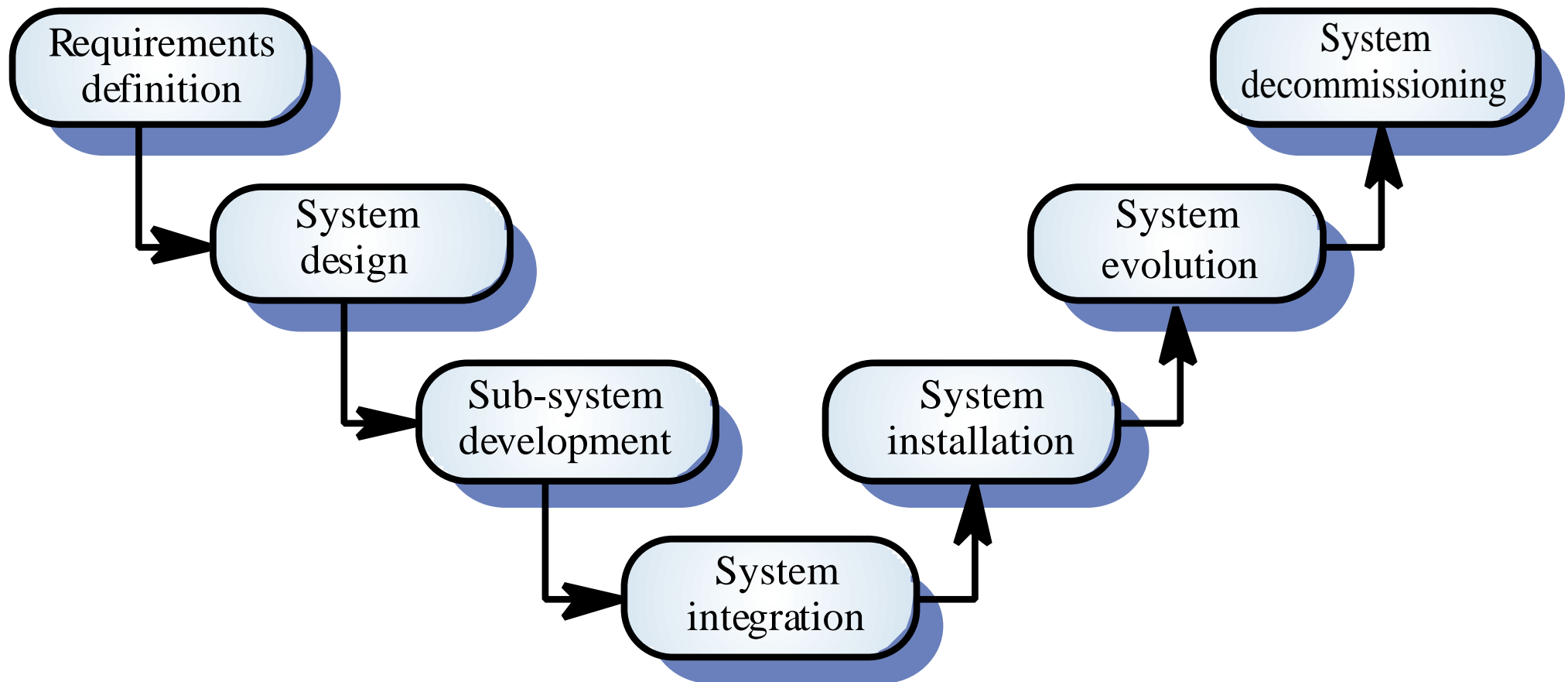
Component types in alarm system

- Sensor
 - Movement sensor, Door sensor
- Actuator
 - Siren
- Communication
 - Telephone caller
- Coordination
 - Alarm controller
- Interface
 - Voice synthesizer

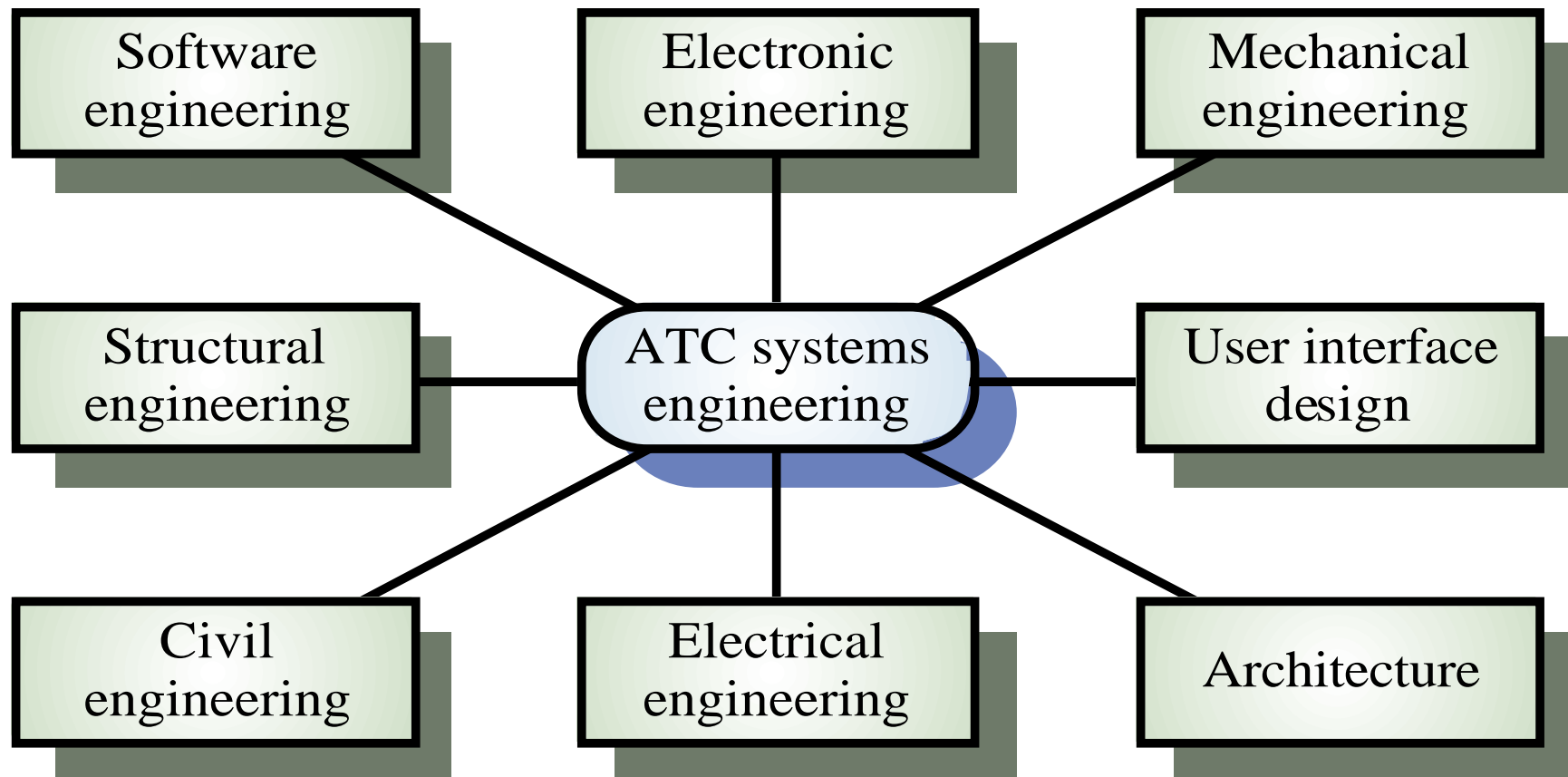
The system engineering process

- Usually follows a ‘waterfall’ model because of the need for parallel development of different parts of the system
 - Little scope for iteration between phases because hardware changes are very expensive. Software may have to compensate for hardware problems
- Inevitably involves engineers from different disciplines who must work together
 - Much scope for misunderstanding here. Different disciplines use a different vocabulary and much negotiation is required. Engineers may have personal agendas to fulfil

The system engineering process



Inter-disciplinary involvement



System requirements definition

- Three types of requirement defined at this stage
 - Abstract functional requirements. System functions are defined in an abstract way
 - System properties. Non-functional requirements for the system in general are defined
 - Undesirable characteristics. Unacceptable system behaviour is specified
- Should also define overall organisational objectives for the system

System objectives

- Functional objectives
 - To provide a fire and intruder alarm system for the building which will provide internal and external warning of fire or unauthorized intrusion
- Organisational objectives
 - To ensure that the normal functioning of work carried out in the building is not seriously disrupted by events such as fire and unauthorized intrusion

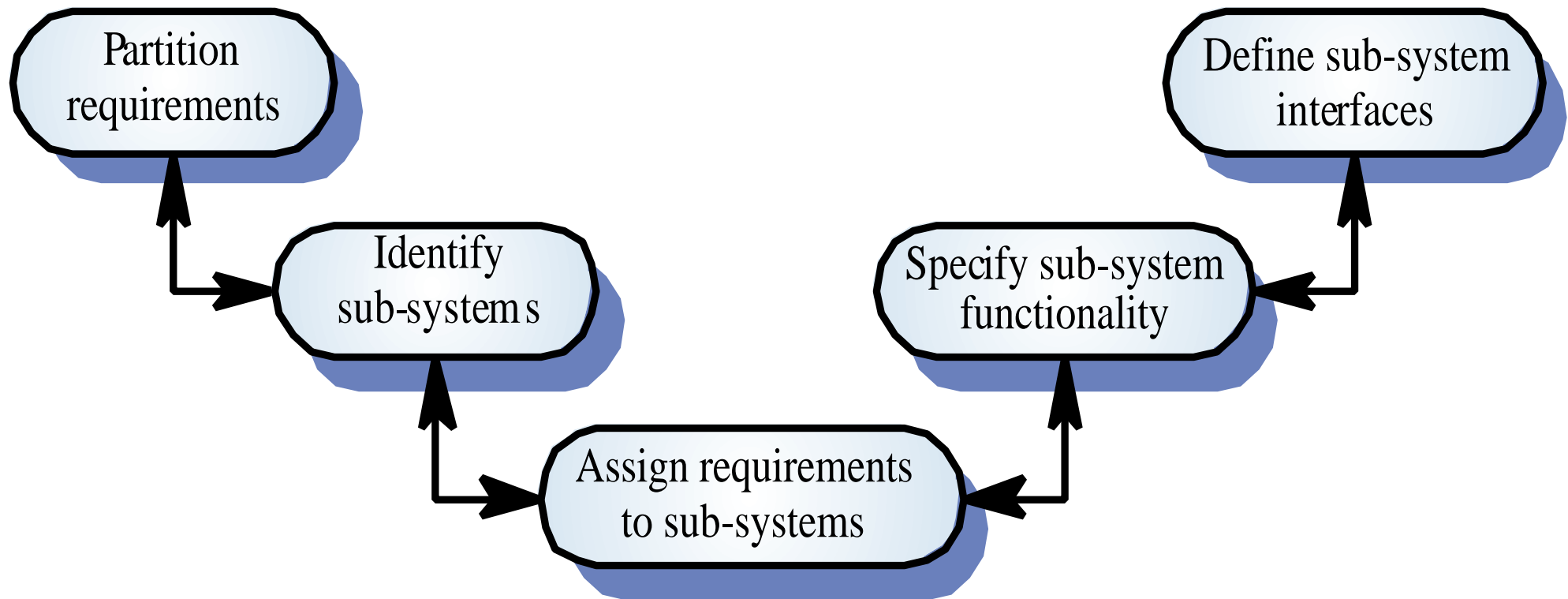
System requirements problems

- Changing as the system is being specified
- Must anticipate hardware/communications developments over the lifetime of the system
- Hard to define non-functional requirements (particularly) without an impression of component structure of the system.

The system design process

- Partition requirements
 - Organise requirements into related groups
- Identify sub-systems
 - Identify a set of sub-systems which collectively can meet the system requirements
- Assign requirements to sub-systems
 - Causes particular problems when COTS are integrated
- Specify sub-system functionality
- Define sub-system interfaces
 - Critical activity for parallel sub-system development

The system design process



System design problems

- Requirements partitioning to hardware, software and human components may involve a lot of negotiation
- Difficult design problems are often assumed to be readily solved using software
- Hardware platforms may be inappropriate for software requirements so software must compensate for this

Sub-system development

- Typically parallel projects developing the hardware, software and communications
- May involve some COTS (Commercial Off-the-Shelf) systems procurement
- Lack of communication across implementation teams
- Bureaucratic and slow mechanism for proposing system changes means that the development schedule may be extended because of the need for rework

System integration

- The process of putting hardware, software and people together to make a system
- Should be tackled incrementally so that sub-systems are integrated one at a time
- Interface problems between sub-systems are usually found at this stage
- May be problems with uncoordinated deliveries of system components

System installation

- Environmental assumptions may be incorrect
- May be human resistance to the introduction of a new system
- System may have to coexist with alternative systems for some time
- May be physical installation problems (e.g. cabling problems)
- Operator training has to be identified

System operation

- Will bring unforeseen requirements to light
- Users may use the system in a way which is not anticipated by system designers
- May reveal problems in the interaction with other systems
 - Physical problems of incompatibility
 - Data conversion problems
 - Increased operator error rate because of inconsistent interfaces

System evolution

- Large systems have a long lifetime. They must evolve to meet changing requirements
- Evolution is inherently costly
 - Changes must be analysed from a technical and business perspective
 - Sub-systems interact so unanticipated problems can arise
 - There is rarely a rationale for original design decisions
 - System structure is corrupted as changes are made to it
- Existing systems which must be maintained are sometimes called **legacy systems**

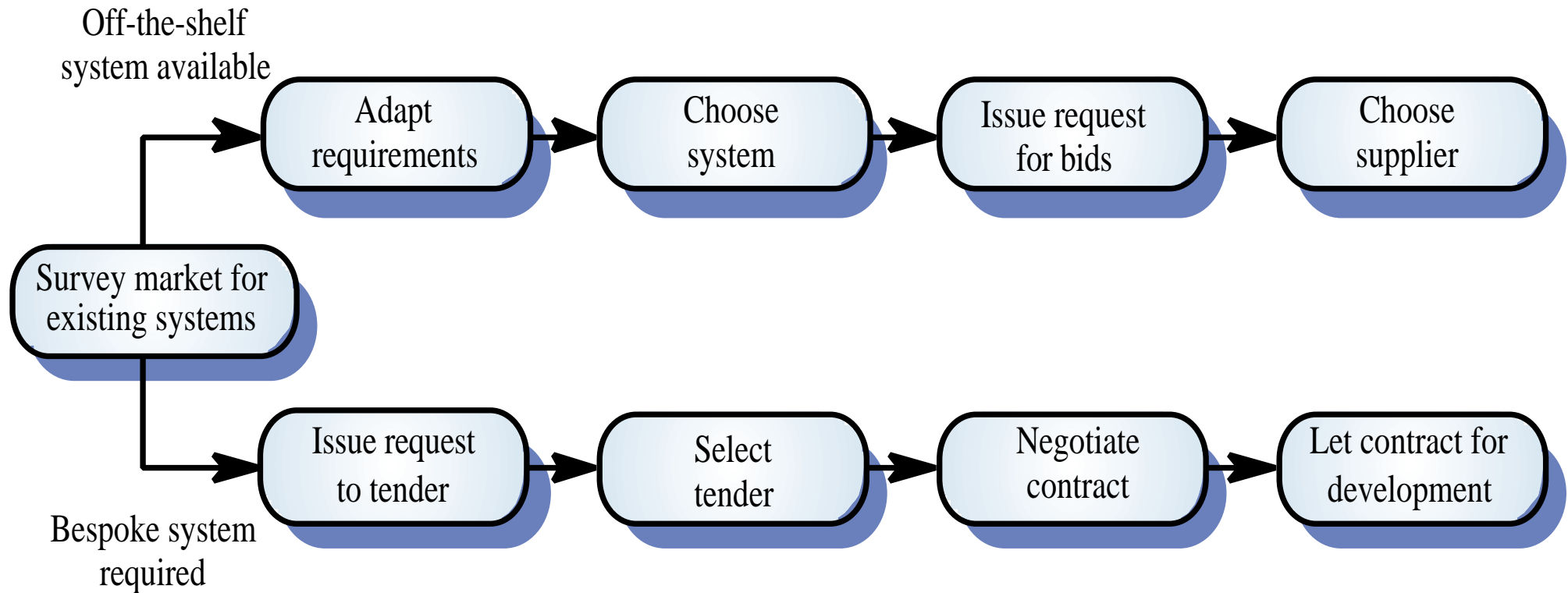
System decommissioning

- Taking the system out of service after its useful lifetime
- May require removal of materials (e.g. dangerous chemicals) which pollute the environment
 - Should be planned for in the system design by encapsulation
- May require data to be restructured and converted to be used in some other system

System procurement

- Acquiring a system for an organization to meet some need
- Some system specification and architectural design is usually necessary before procurement
 - You need a specification to let a contract for system development
 - The specification may allow you to buy a commercial off-the-shelf (COTS) system. Almost always cheaper than developing a system from scratch

The system procurement process



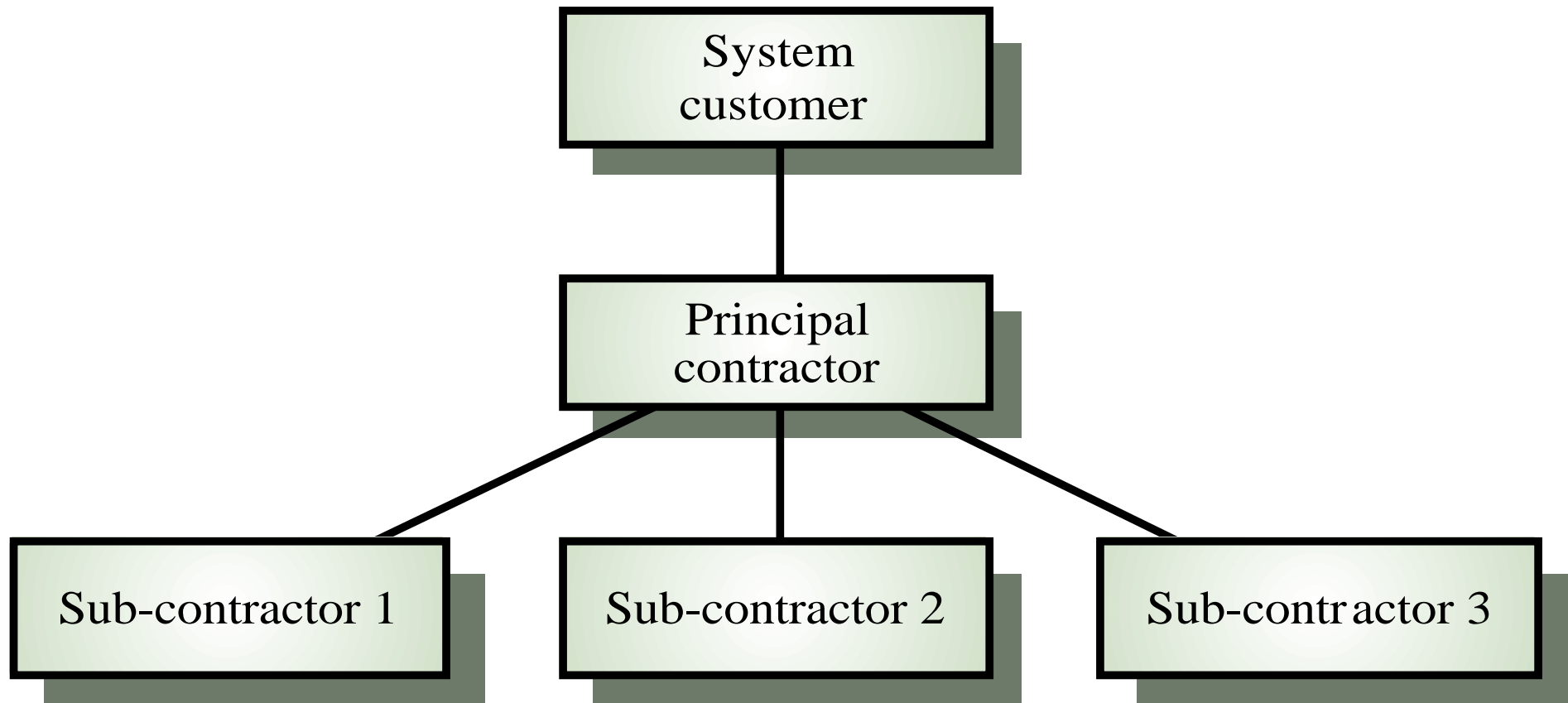
Procurement issues

- Requirements may have to be modified to match the capabilities of off-the-shelf components
- The requirements specification may be part of the contract for the development of the system
- There is usually a contract negotiation period to agree changes after the contractor to build a system has been selected

Contractors and sub-contractors

- The procurement of large hardware/software systems is usually based around some principal contractor
- Sub-contracts are issued to other suppliers to supply parts of the system
- Customer liases with the principal contractor and does not deal directly with sub-contractors

Contractor/Sub-contractor model



Key points

- System engineering involves input from a range of disciplines
- Emergent properties are properties that are characteristic of the system as a whole and not its component parts
- System architectural models show major sub-systems and inter-connections. They are usually described using block diagrams

Key points

- System component types are sensor, actuator, computation, co-ordination, communication and interface
- The systems engineering process is usually a waterfall model and includes specification, design, development and integration.
- System procurement is concerned with deciding which system to buy and who to buy it from

Conclusion

- Systems engineering is hard! There will never be an easy answer to the problems of complex system development
- Software engineers do not have all the answers but may be better at taking a systems viewpoint
- Disciplines need to recognise each others strengths and actively rather than reluctantly cooperate in the systems engineering process

Software Processes

- Coherent sets of activities for specifying, designing, implementing and testing software systems

Objectives

- To introduce software process models
- To describe a number of different process models and when they may be used
- To describe outline process models for requirements engineering, software development, testing and evolution
- To introduce CASE technology to support software process activities

Topics covered

- Software process models
- Process iteration
- Software specification
- Software design and implementation
- Software validation
- Software evolution
- Automated process support

The software process

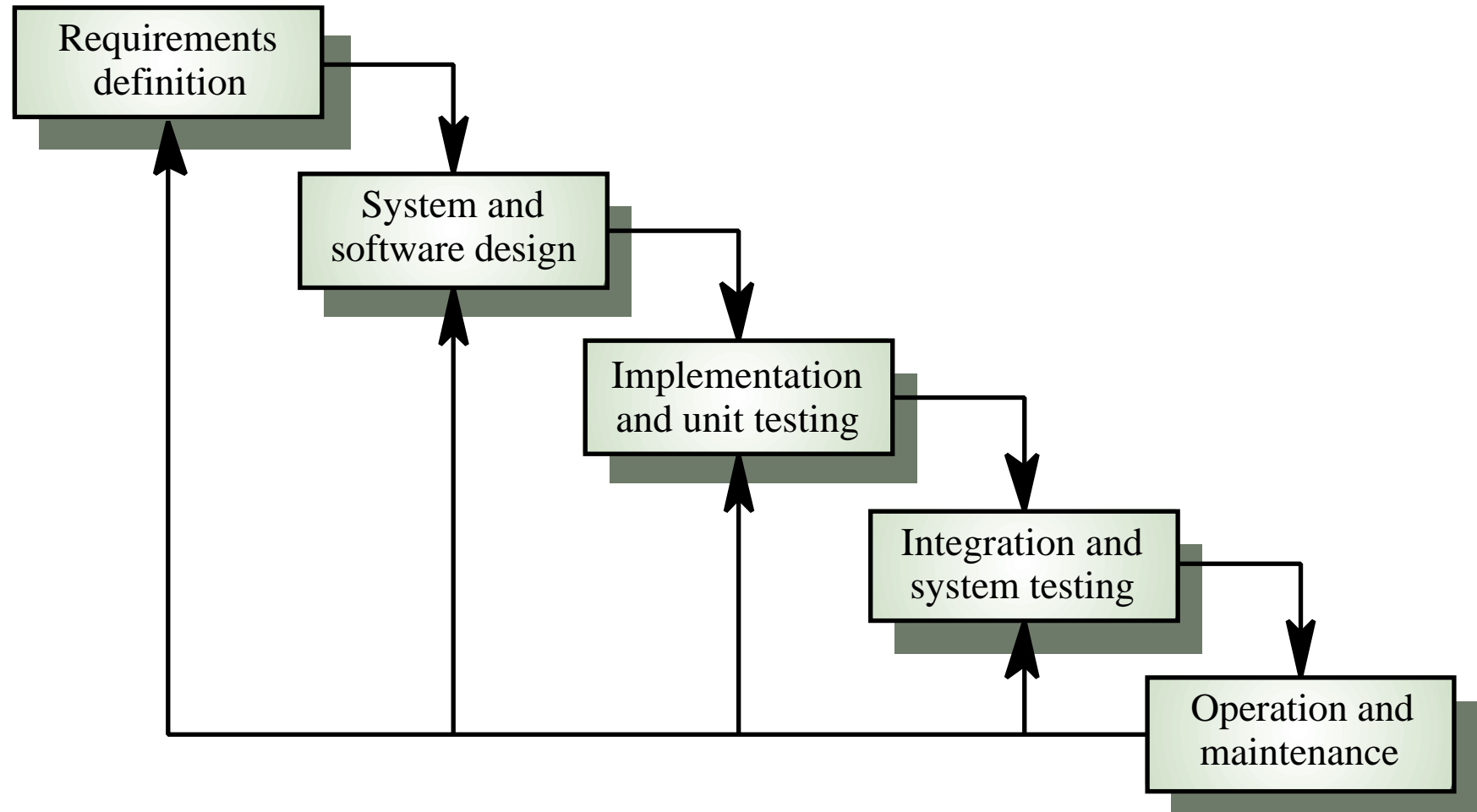
- A structured set of activities required to develop a software system
 - Specification
 - Design
 - Validation
 - Evolution
- A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective

Software process models

Generic software process models

- The waterfall model
 - Separate and distinct phases of specification and development
- Evolutionary development
 - Specification and development are interleaved
- Formal systems development
 - A mathematical system model is formally transformed to an implementation
- Reuse-based development
 - The system is assembled from existing components

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 drawback of the waterfall model is the difficulty of accommodating change after the process is underway

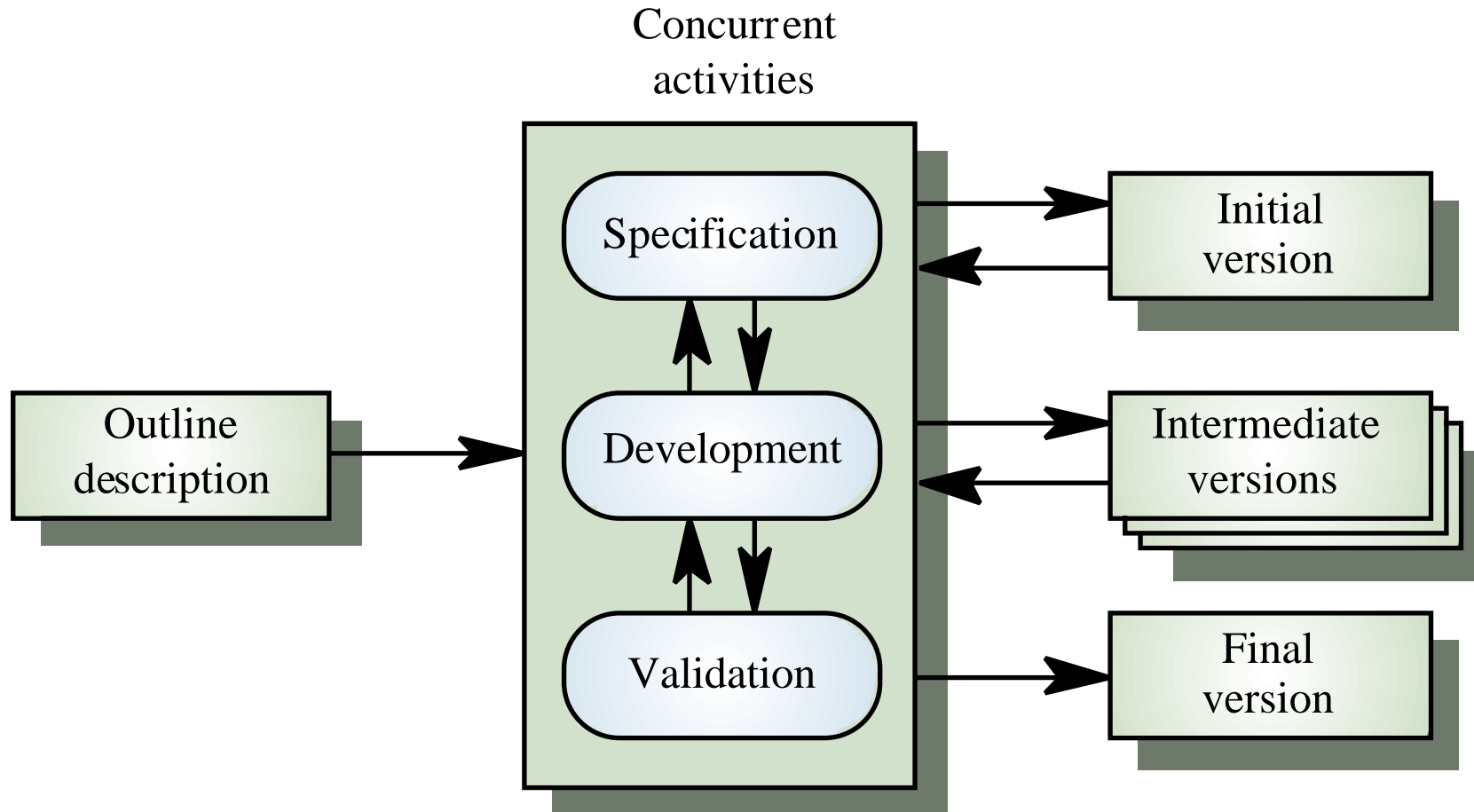
Waterfall model problems

- Inflexible partitioning of the project into distinct stages
- This makes it difficult to respond to changing customer requirements
- Therefore, this model is only appropriate when the requirements are well-understood

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
- Throw-away prototyping
 - Objective is to understand the system requirements. Should start with poorly understood requirements

Evolutionary development



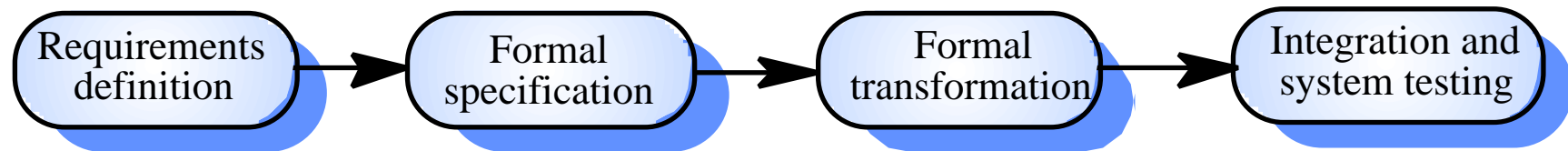
Evolutionary development

- 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

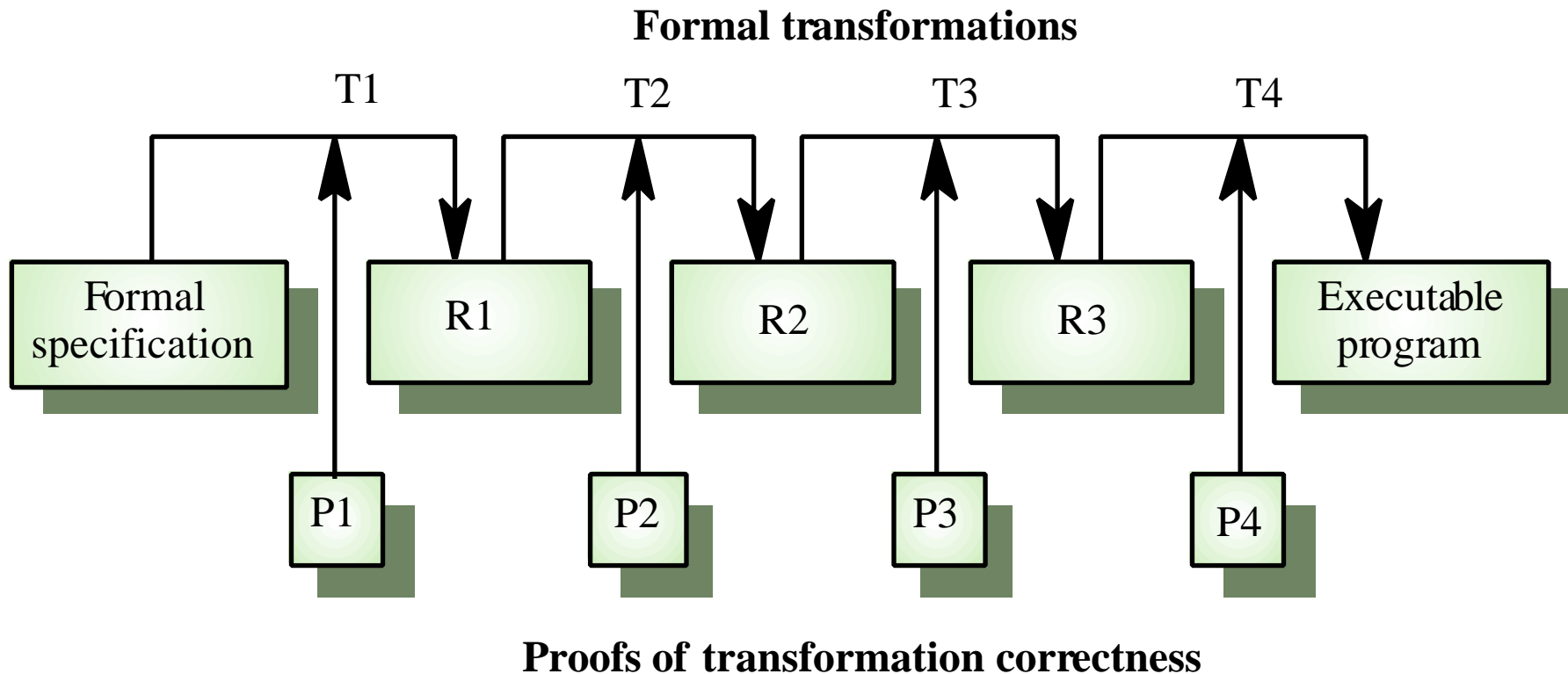
Formal systems development

- Based on the transformation of a mathematical specification through different representations to an executable program
- Transformations are ‘correctness-preserving’ so it is straightforward to show that the program conforms to its specification
- Embodied in the ‘Cleanroom’ approach to software development

Formal systems development



Formal transformations



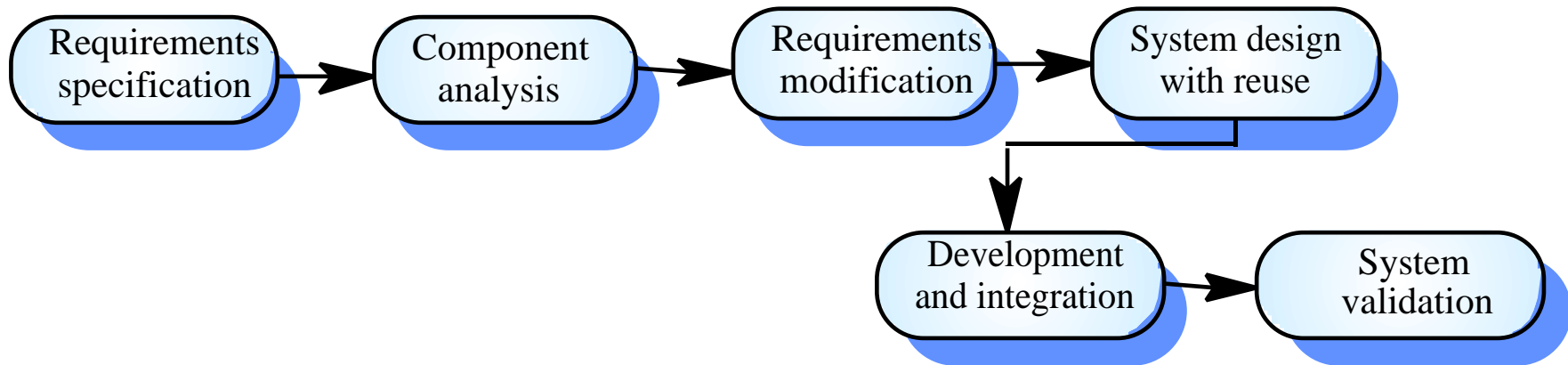
Formal systems development

- Problems
 - Need for specialised skills and training to apply the technique
 - Difficult to formally specify some aspects of the system such as the user interface
- Applicability
 - Critical systems especially those where a safety or security case must be made before the system is put into operation

Reuse-oriented development

- 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 more important but still limited experience with it

Reuse-oriented development



Process iteration

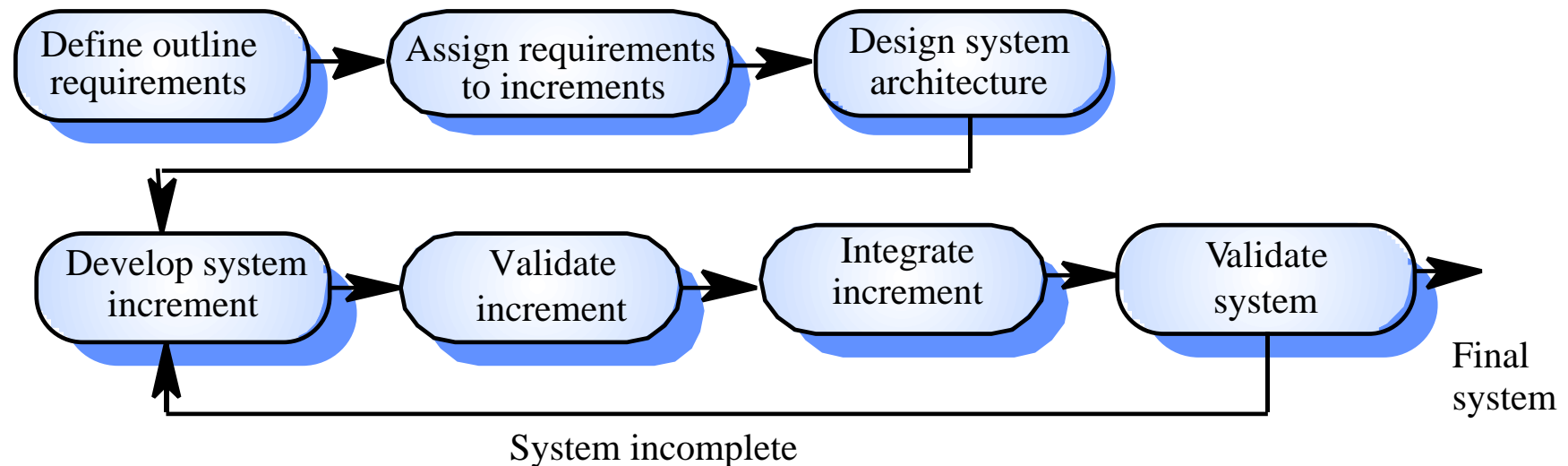
Process iteration

- 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 development
 - Spiral development

Incremental development

- 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

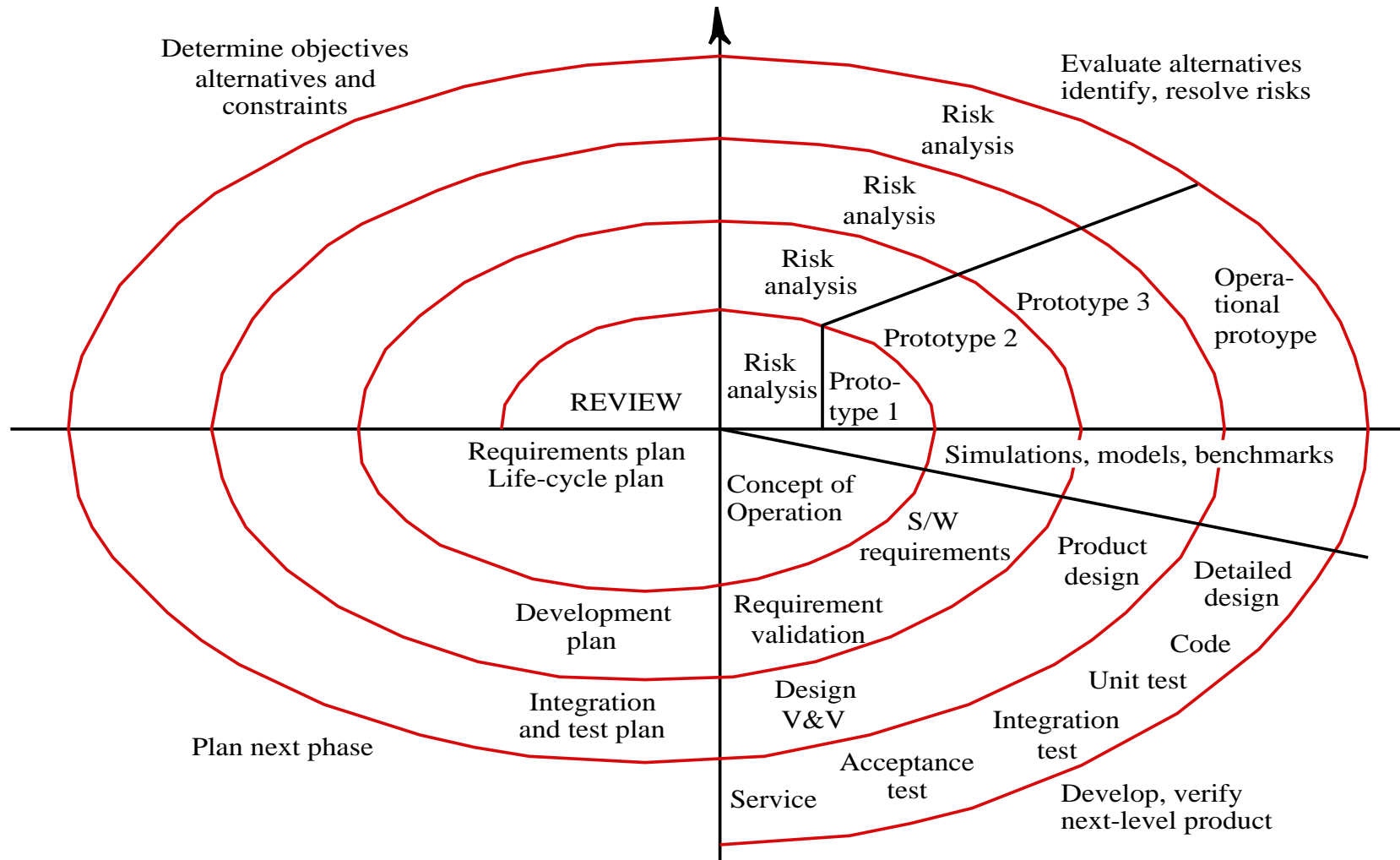
Extreme programming

- New 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



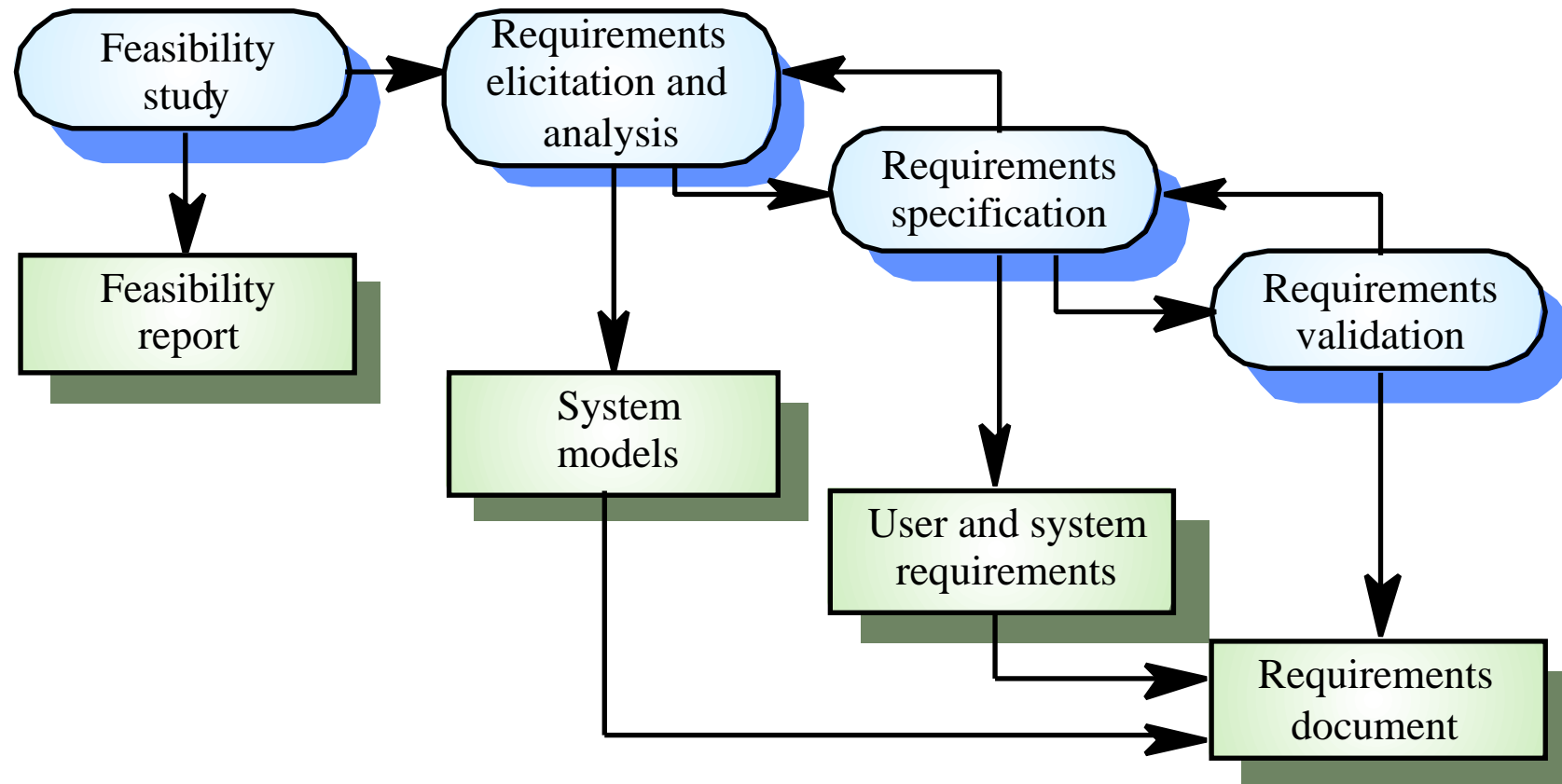
Spiral model sectors

- 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

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



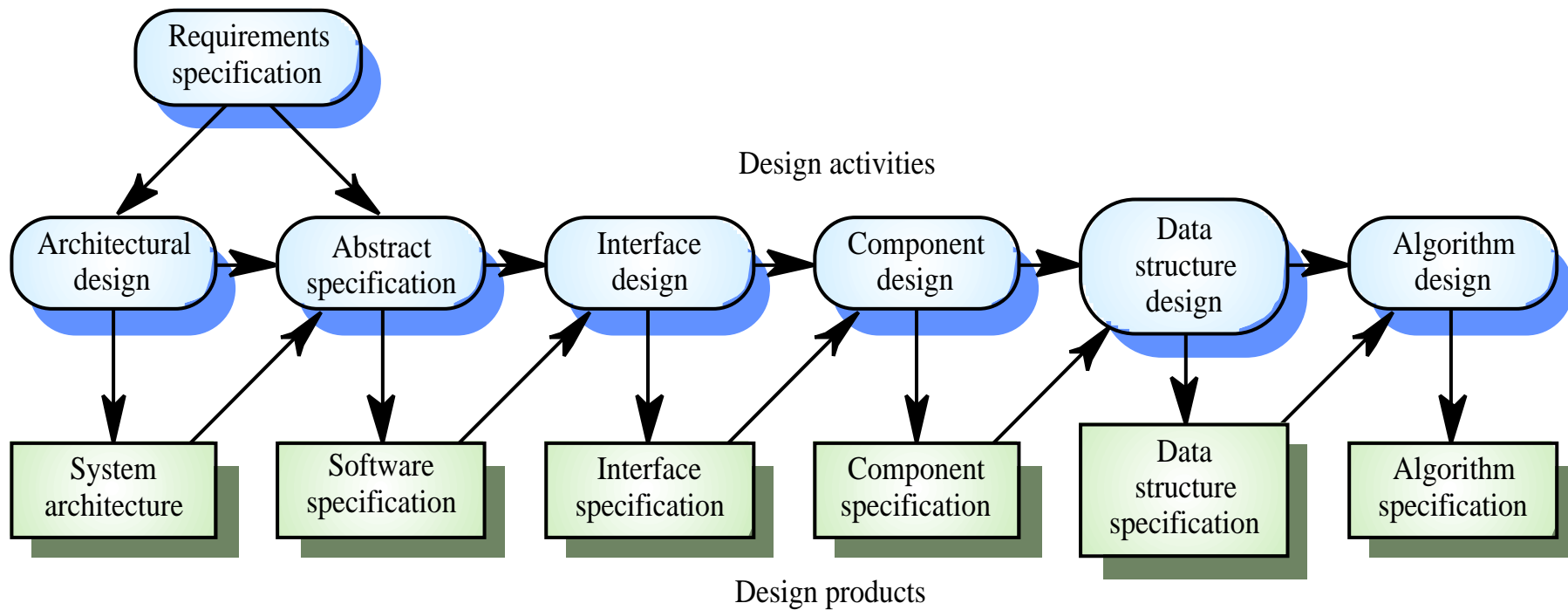
Software design and implementation

- 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



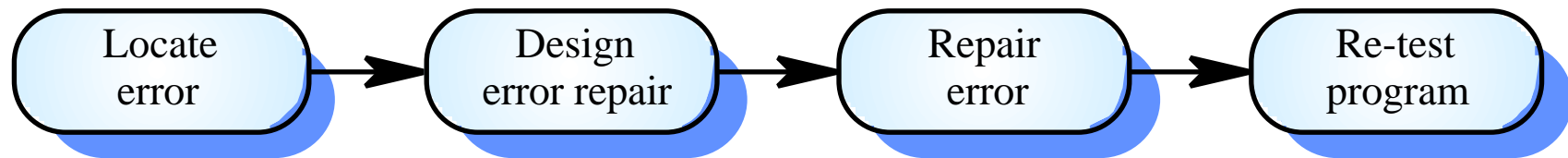
Design methods

- Systematic approaches to developing a software design
- The design is usually documented as a set of graphical models
- Possible models
 - Data-flow model
 - Entity-relation-attribute model
 - Structural model
 - Object models

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

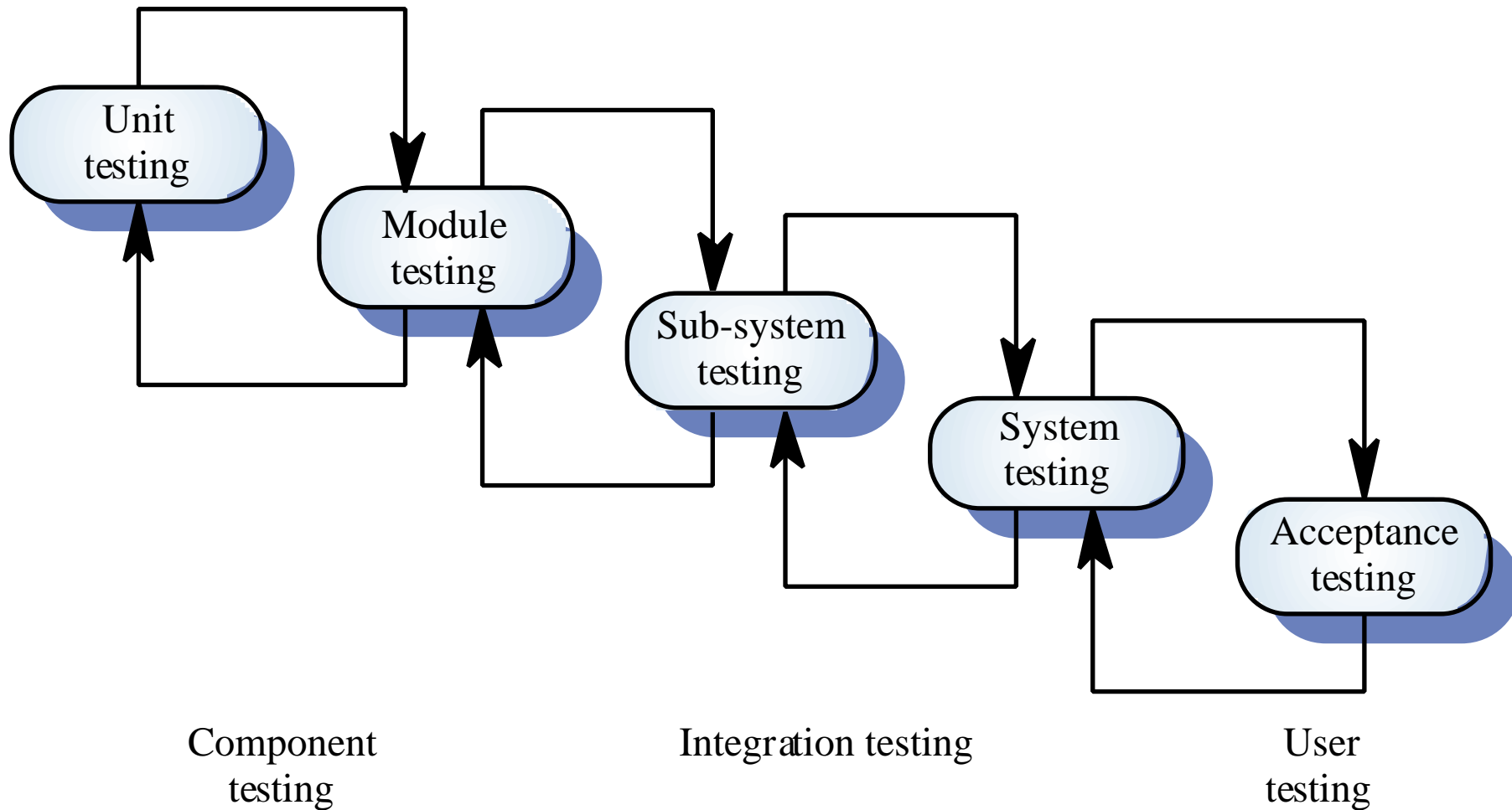
The debugging process



Software validation

- Verification and validation 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

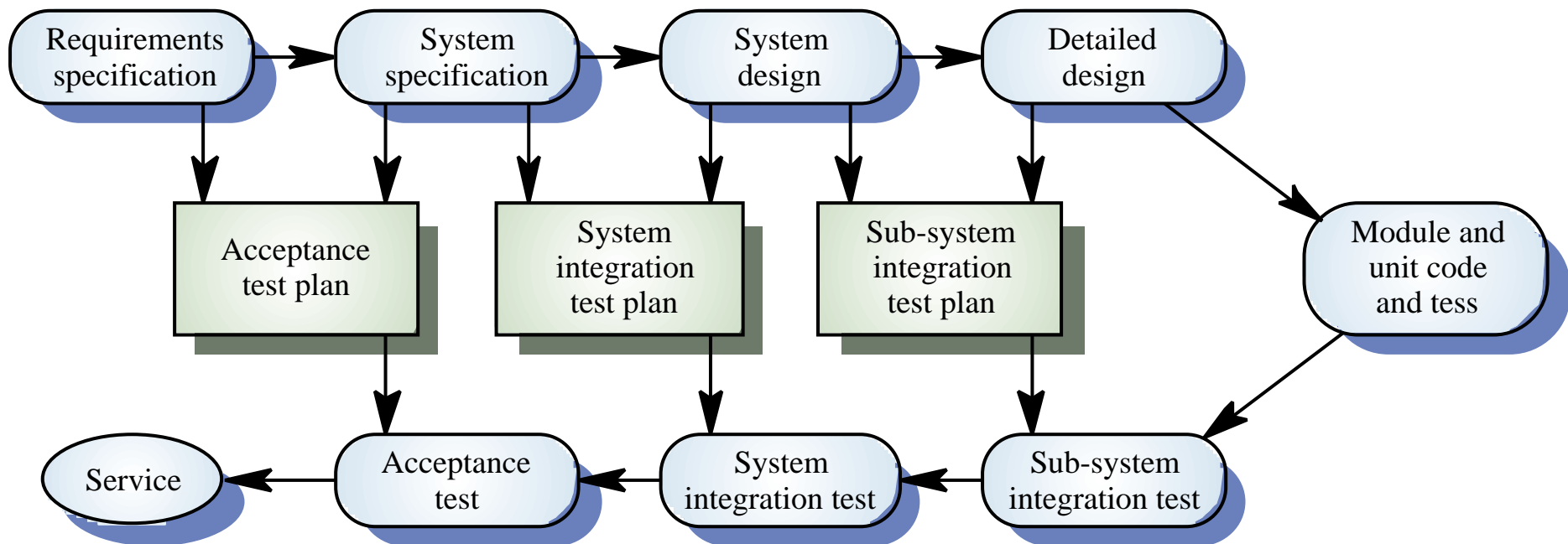
The testing process



Testing stages

- Unit testing
 - Individual components are tested
- Module testing
 - Related collections of dependent components are tested
- Sub-system testing
 - Modules are integrated into sub-systems and tested. The focus here should be on interface testing
- System testing
 - Testing of the system as a whole. Testing of emergent properties
- Acceptance testing
 - Testing with customer data to check that it is acceptable

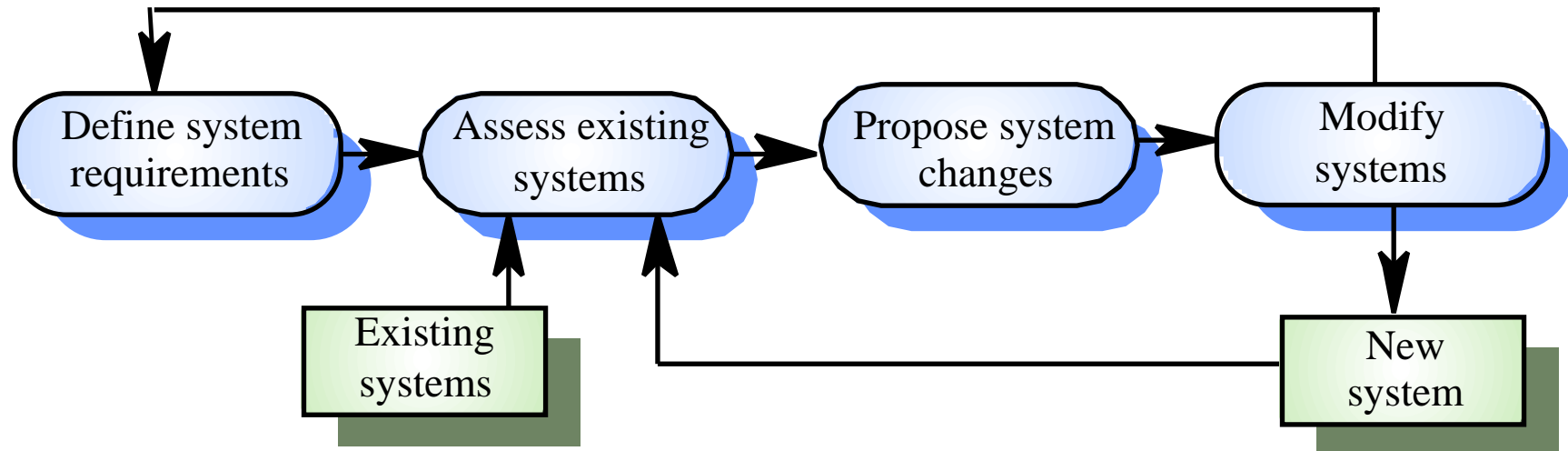
Testing phases



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



Automated process support

CASE

- 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

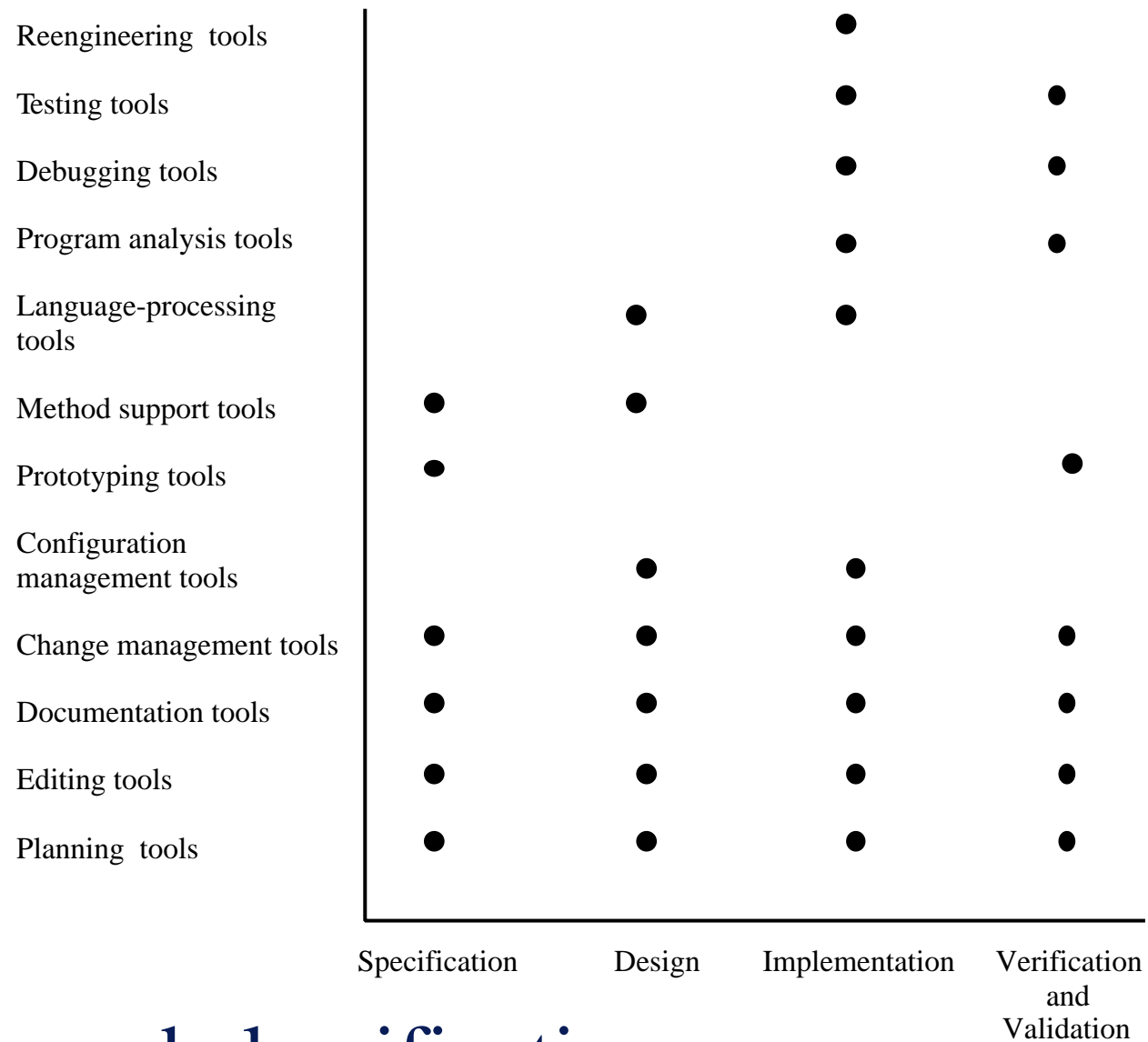
- Case technology has led to significant improvements in the software process though not the order of magnitude improvements that were once predicted
 - Software engineering requires creative thought - this is not readily automatable
 - Software engineering is a team activity and, for large projects, 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

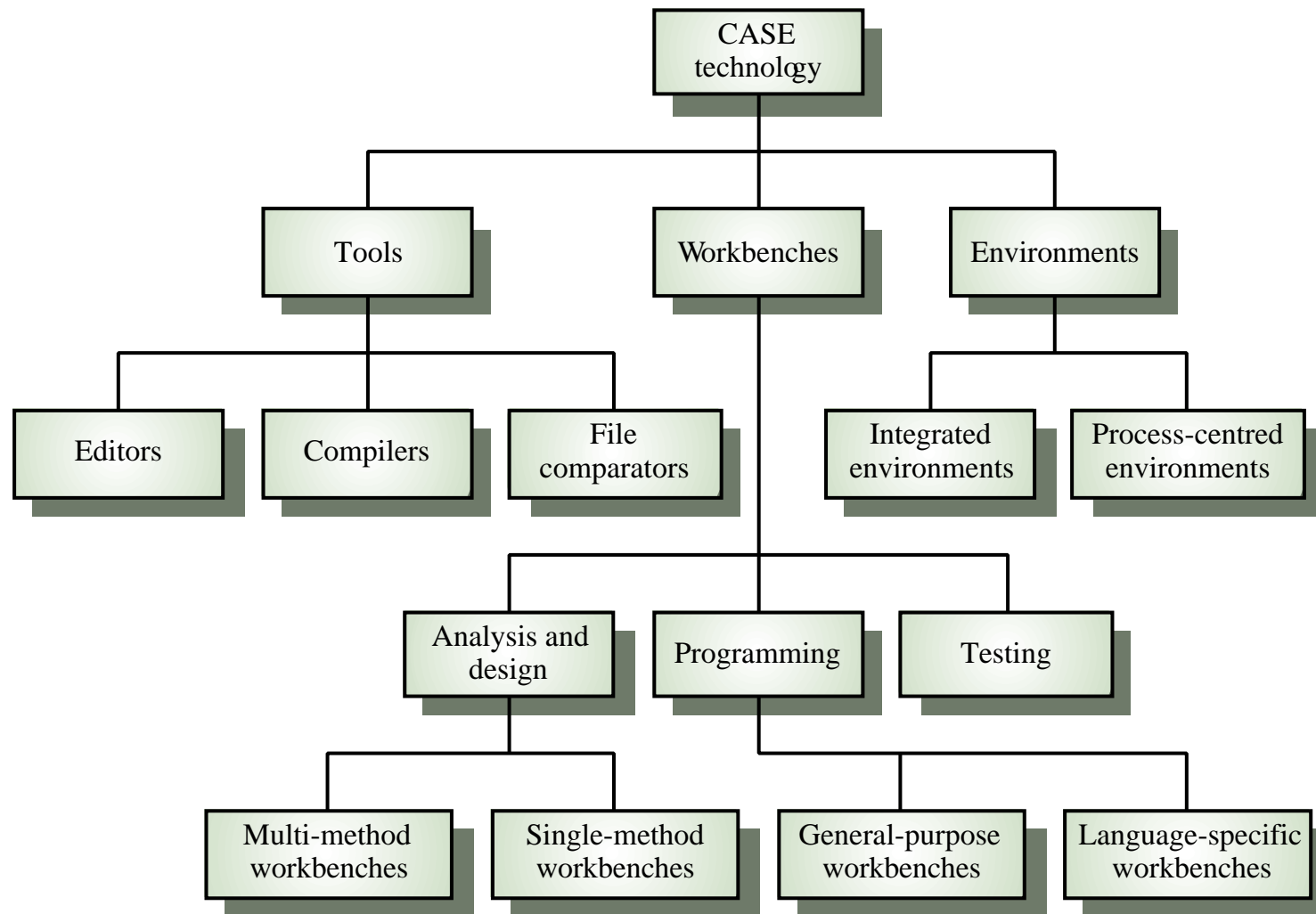


Activity-based classification

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. They are represented in a software process model
- General activities are specification, design and implementation, validation and evolution
- Generic process models describe the organisation of software processes
- 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
- CASE technology supports software process activities

Project management

- Organising, planning and scheduling software projects

Objectives

- To introduce software project management and to describe its distinctive characteristics
- To discuss project planning and the planning process
- To show how graphical schedule representations are used by project management
- To discuss the notion of risks and the risk management process

Topics covered

- Management activities
- Project planning
- Project scheduling
- Risk management

Software project management

- Concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organisations developing and procuring the software
- Project management is needed because software development is always subject to budget and schedule constraints that are set by the organisation developing the software

Software management distinctions

- The product is intangible
- The product is uniquely flexible
- Software engineering is not recognized as an engineering discipline with the same status as mechanical, electrical engineering, etc.
- The software development process is not standardised
- Many software projects are 'one-off' projects

Management activities

Management activities

- Proposal writing
- Project planning and scheduling
- Project costing
- Project monitoring and reviews
- Personnel selection and evaluation
- Report writing and presentations

Management commonalities

- These activities are not peculiar to software management
- Many techniques of engineering project management are equally applicable to software project management
- Technically complex engineering systems tend to suffer from the same problems as software systems

Project staffing

- May not be possible to appoint the ideal people to work on a project
 - Project budget may not allow for the use of highly-paid staff
 - Staff with the appropriate experience may not be available
 - An organisation may wish to develop employee skills on a software project
- Managers have to work within these constraints especially when (as is currently the case) there is an international shortage of skilled IT staff

Project planning

Project planning

- Probably the most time-consuming project management activity
- Continuous activity from initial concept through to system delivery. Plans must be regularly revised as new information becomes available
- Various different types of plan may be developed to support the main software project plan that is concerned with schedule and budget

Types of project plan

Plan	Description
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources and schedule used for system validation.
Configuration management plan	Describes the configuration management procedures and structures to be used.
Maintenance plan	Predicts the maintenance requirements of the system, maintenance costs and effort required.
Staff development plan.	Describes how the skills and experience of the project team members will be developed.

Project planning process

```
Establish the project constraints
Make initial assessments of the project parameters
Define project milestones and deliverables
while project has not been completed or cancelled loop
    Draw up project schedule
    Initiate activities according to schedule
    Wait ( for a while )
    Review project progress
    Revise estimates of project parameters
    Update the project schedule
    Re-negotiate project constraints and deliverables
    if ( problems arise ) then
        Initiate technical review and possible revision
    end if
end loop
```

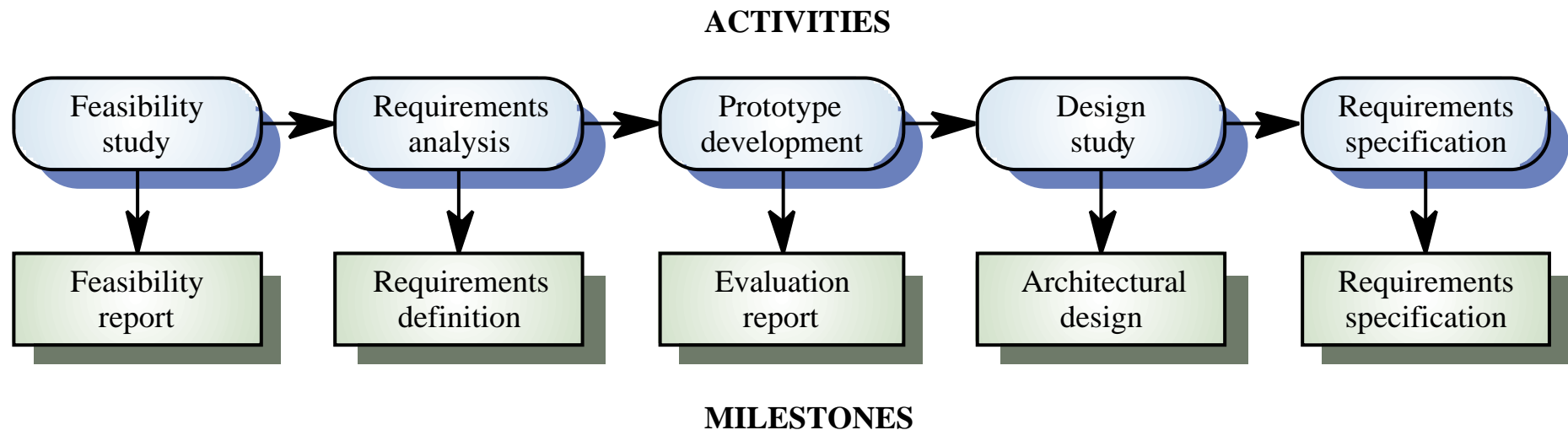
Project plan structure

- Introduction
- Project organisation
- Risk analysis
- Hardware and software resource requirements
- Work breakdown
- Project schedule
- Monitoring and reporting mechanisms

Activity organization

- Activities in a project should be organised to produce tangible outputs for management to judge progress
- *Milestones* are the end-point of a process activity
- *Deliverables* are project results delivered to customers
- The waterfall process allows for the straightforward definition of progress milestones

Milestones in the RE process

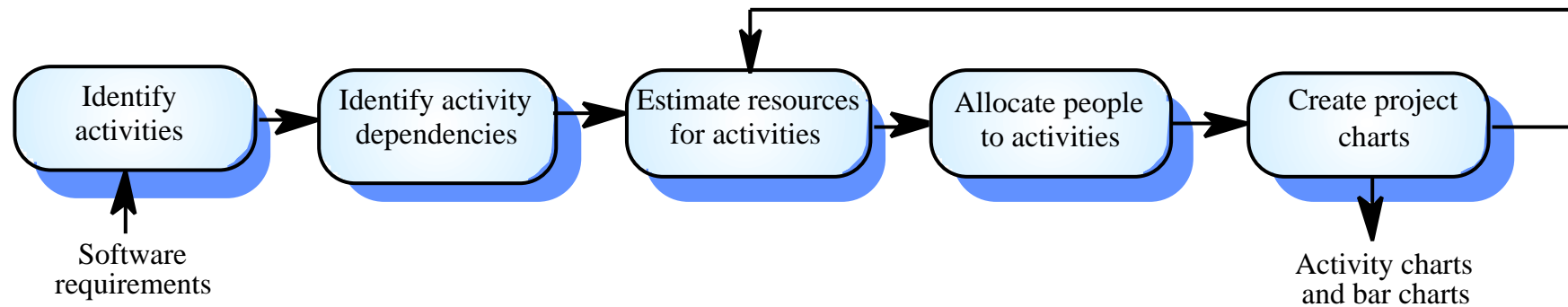


Project scheduling

Project scheduling

- Split project into tasks and estimate time and resources required to complete each task
- Organize tasks concurrently to make optimal use of workforce
- Minimize task dependencies to avoid delays caused by one task waiting for another to complete
- Dependent on project managers intuition and experience

The project scheduling process



Scheduling problems

- Estimating the difficulty of problems and hence the cost of developing a solution is hard
- Productivity is not proportional to the number of people working on a task
- Adding people to a late project makes it later because of communication overheads
- The unexpected always happens. Always allow contingency in planning

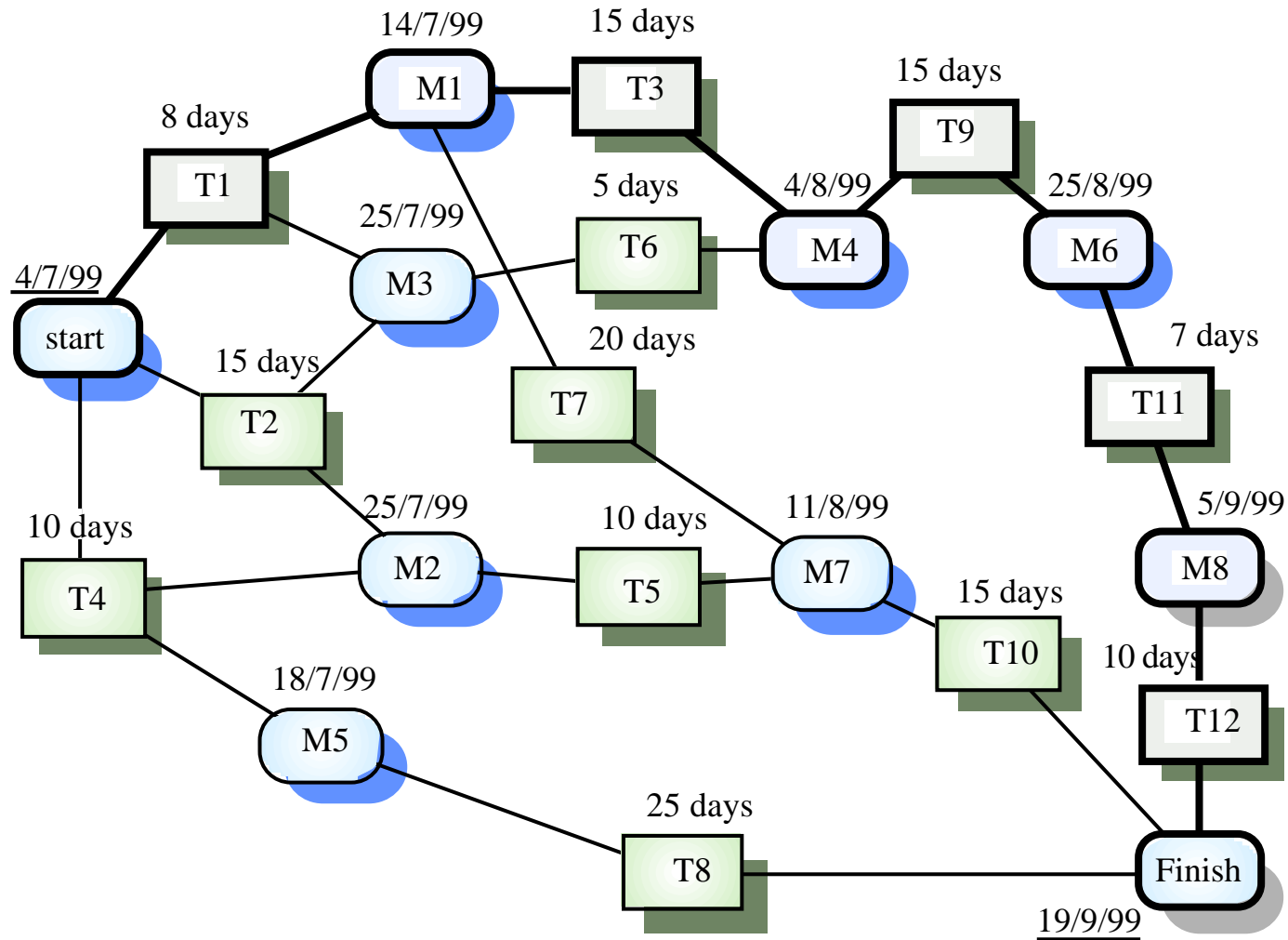
Bar charts and activity networks

- Graphical notations used to illustrate the project schedule
- Show project breakdown into tasks. Tasks should not be too small. They should take about a week or two
- Activity charts show task dependencies and the critical path
- Bar charts show schedule against calendar time

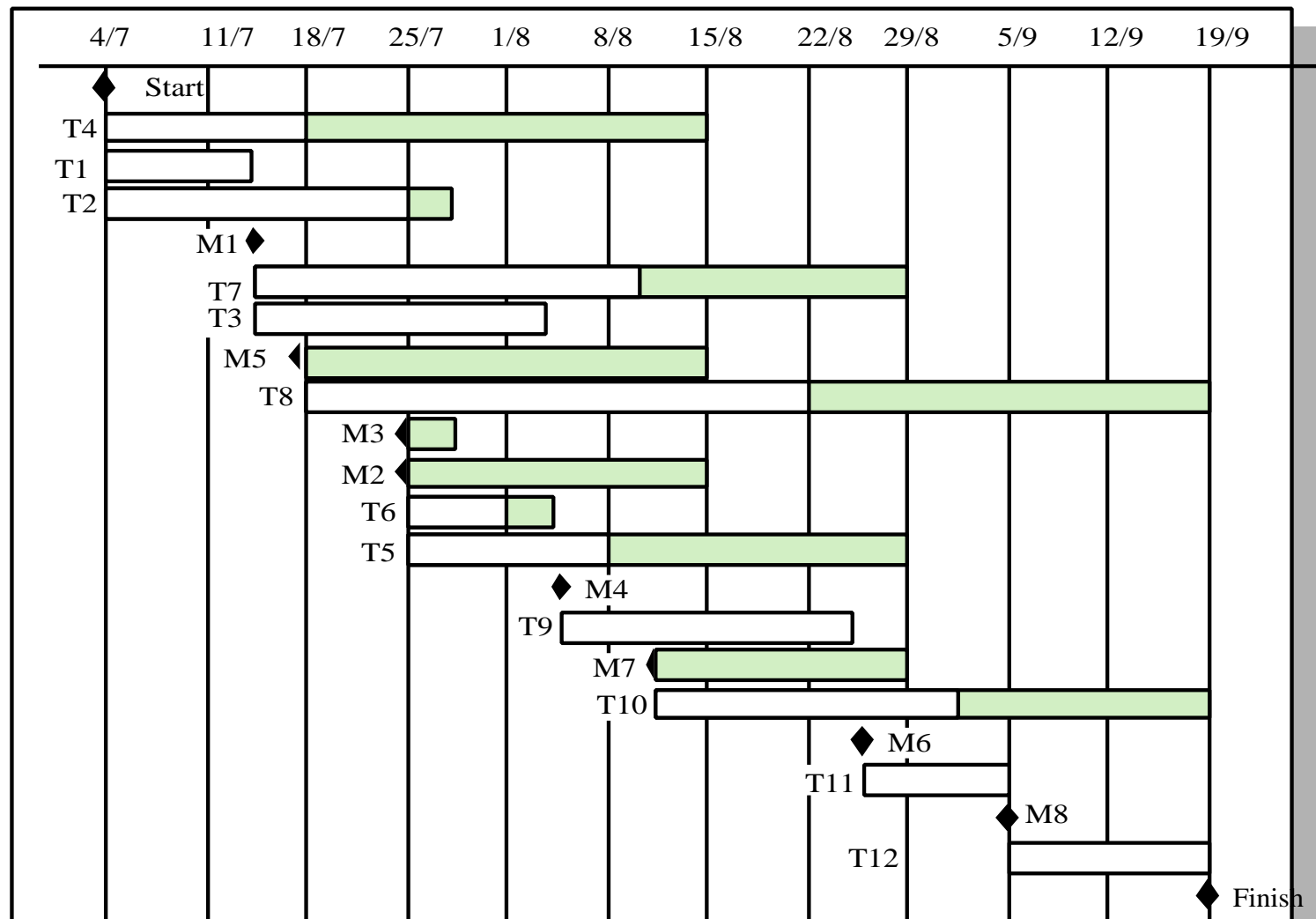
Task durations and dependencies

Task	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

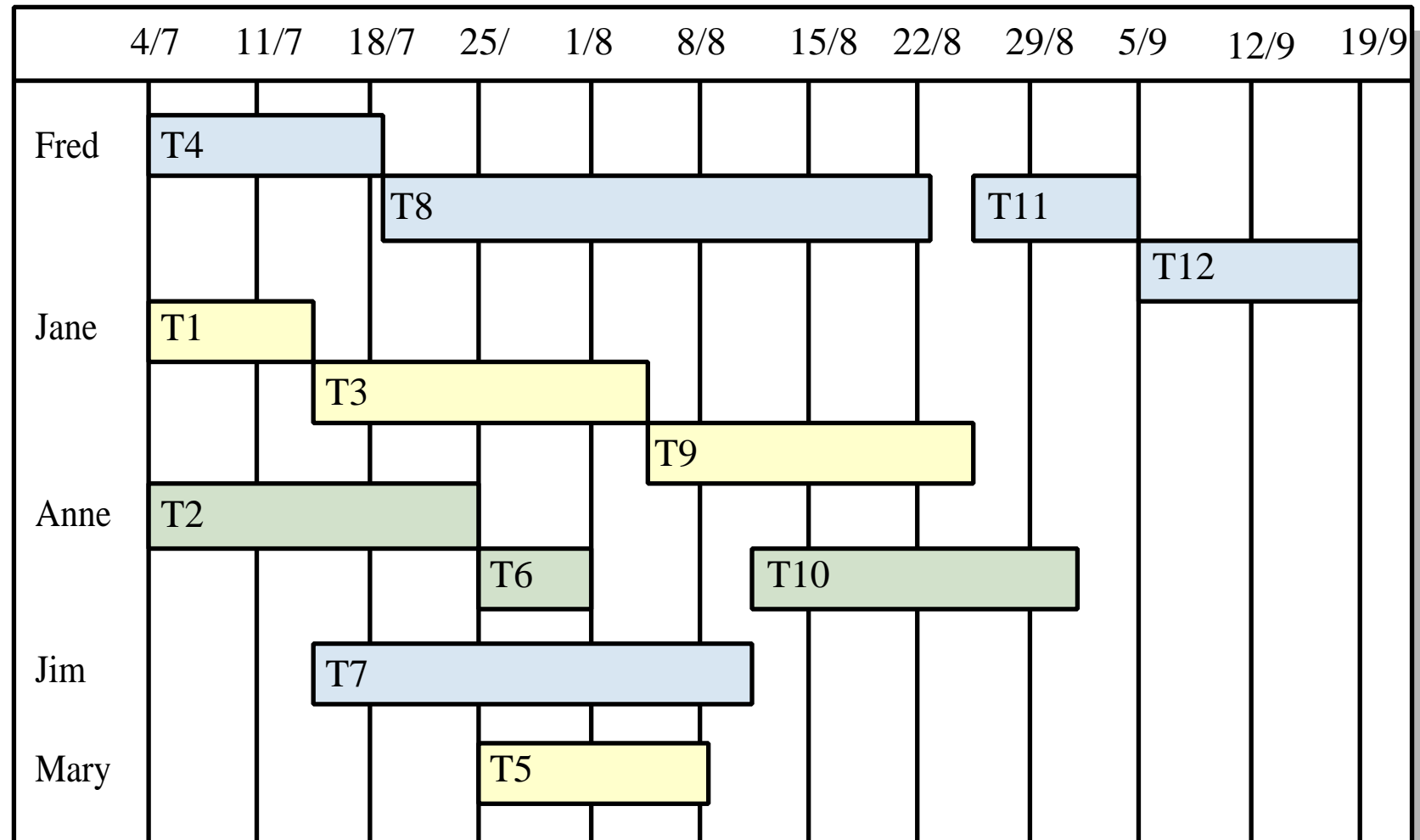
Activity network



Activity timeline



Staff allocation



Risk management

Risk management

- Risk management is concerned with identifying risks and drawing up plans to minimise their effect on a project.
- A risk is a probability that some adverse circumstance will occur.
 - Project risks affect schedule or resources
 - Product risks affect the quality or performance of the software being developed
 - Business risks affect the organisation developing or procuring the software

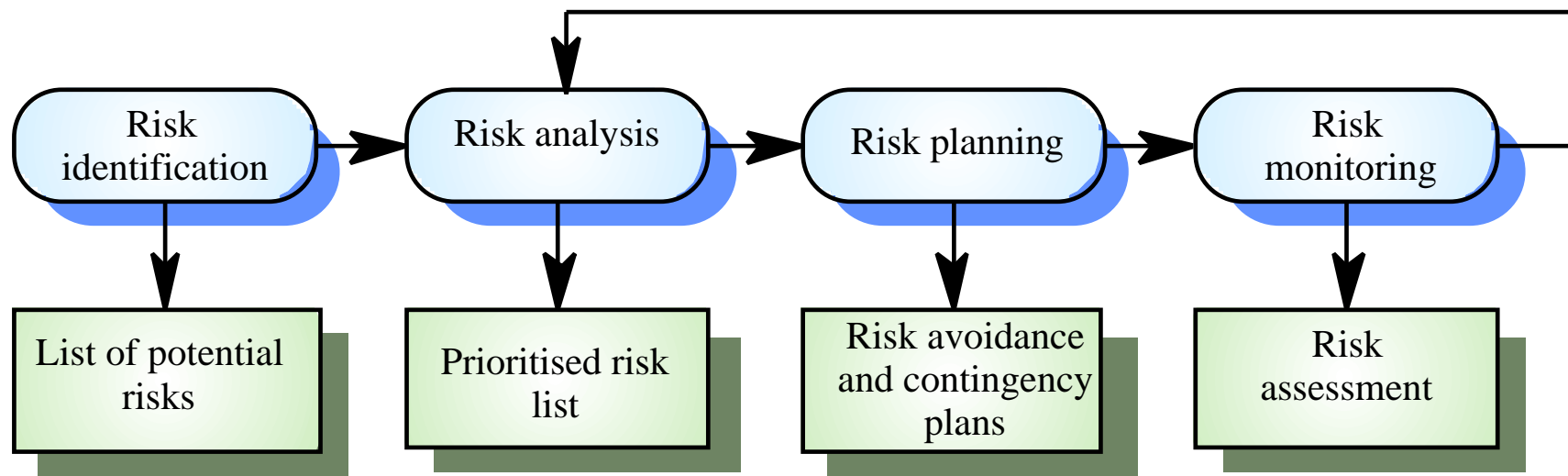
Software risks

Risk	Risk type	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organisational management with different priorities.
Hardware unavailability	Project	Hardware which is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool under-performance	Product	CASE tools which support the project do not perform as anticipated
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

The risk management process

- Risk identification
 - Identify project, product and business risks
- Risk analysis
 - Assess the likelihood and consequences of these risks
- Risk planning
 - Draw up plans to avoid or minimise the effects of the risk
- Risk monitoring
 - Monitor the risks throughout the project

The risk management process



Risk identification

- Technology risks
- People risks
- Organisational risks
- Requirements risks
- Estimation risks

Risks and risk types

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. Software components which should be reused contain defects which limit their functionality.
People	It is impossible to recruit staff with the skills required. Key staff are ill and unavailable at critical times. Required training for staff is not available.
Organisational	The organisation is restructured so that different management are responsible for the project. Organisational financial problems force reductions in the project budget.
Tools	The code generated by CASE tools is inefficient. CASE tools cannot be integrated.
Requirements	Changes to requirements which require major design rework are proposed. Customers fail to understand the impact of requirements changes.
Estimation	The time required to develop the software is underestimated. The rate of defect repair is underestimated. The size of the software is underestimated.

Risk analysis

- Assess probability and seriousness of each risk
- Probability may be very low, low, moderate, high or very high
- Risk effects might be catastrophic, serious, tolerable or insignificant

Risk analysis

Risk	Probability	Effects
Organisational financial problems force reductions in the project budget.	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project.	High	Catastrophic
Key staff are ill at critical times in the project.	Moderate	Serious
Software components which should be reused contain defects which limit their functionality.	Moderate	Serious
Changes to requirements which require major design rework are proposed.	Moderate	Serious
The organisation is restructured so that different management are responsible for the project.	High	Serious
The database used in the system cannot process as many transactions per second as expected.	Moderate	Serious
The time required to develop the software is underestimated.	High	Serious
CASE tools cannot be integrated.	High	Tolerable
Customers fail to understand the impact of requirements changes.	Moderate	Tolerable
Required training for staff is not available.	Moderate	Tolerable
The rate of defect repair is underestimated.	Moderate	Tolerable
The size of the software is underestimated.	High	Tolerable
The code generated by CASE tools is inefficient.	Moderate	Insignificant

Risk planning

- Consider each risk and develop a strategy to manage that risk
- Avoidance strategies
 - The probability that the risk will arise is reduced
- Minimisation strategies
 - The impact of the risk on the project or product will be reduced
- Contingency plans
 - If the risk arises, contingency plans are plans to deal with that risk

Risk management strategies

Risk	Strategy
Organisational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Recruitment problems	Alert customer of potential difficulties and the possibility of delays, investigate buying-in components.
Staff illness	Reorganise team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact, maximise information hiding in the design.
Organisational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying in components, investigate use of a program generator.

Risk monitoring

- Assess each identified risks regularly to decide whether or not it is becoming less or more probable
- Also assess whether the effects of the risk have changed
- Each key risk should be discussed at management progress meetings

Risk factors

Risk type	Potential indicators
Technology	Late delivery of hardware or support software, many reported technology problems
People	Poor staff morale, poor relationships amongst team member, job availability
Organisational	organisational gossip, lack of action by senior management
Tools	reluctance by team members to use tools, complaints about CASE tools, demands for higher-powered workstations
Requirements	many requirements change requests, customer complaints
Estimation	failure to meet agreed schedule, failure to clear reported defects

Key points

- Good project management is essential for project success
- The intangible nature of software causes problems for management
- Managers have diverse roles but their most significant activities are planning, estimating and scheduling
- Planning and estimating are iterative processes which continue throughout the course of a project

Key points

- A project milestone is a predictable state where some formal report of progress is presented to management.
- Risks may be project risks, product risks or business risks
- Risk management is concerned with identifying risks which may affect the project and planning to ensure that these risks do not develop into major threats

Software Requirements

- Descriptions and specifications of a system

Objectives

- To introduce the concepts of user and system requirements
- To describe functional and non-functional requirements
- To explain two techniques for describing system requirements
- To explain how software requirements may be organised in a requirements document

Topics covered

- Functional and non-functional requirements
- User requirements
- System requirements
- The software requirements document

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

Requirements abstraction (Davis)

“If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organisation’s needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the *requirements document* for the system.”

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 services. Written as a contract between client and contractor
- Software specification
 - A detailed software description which can serve as a basis for a design or implementation. Written for developers

Definitions and specifications

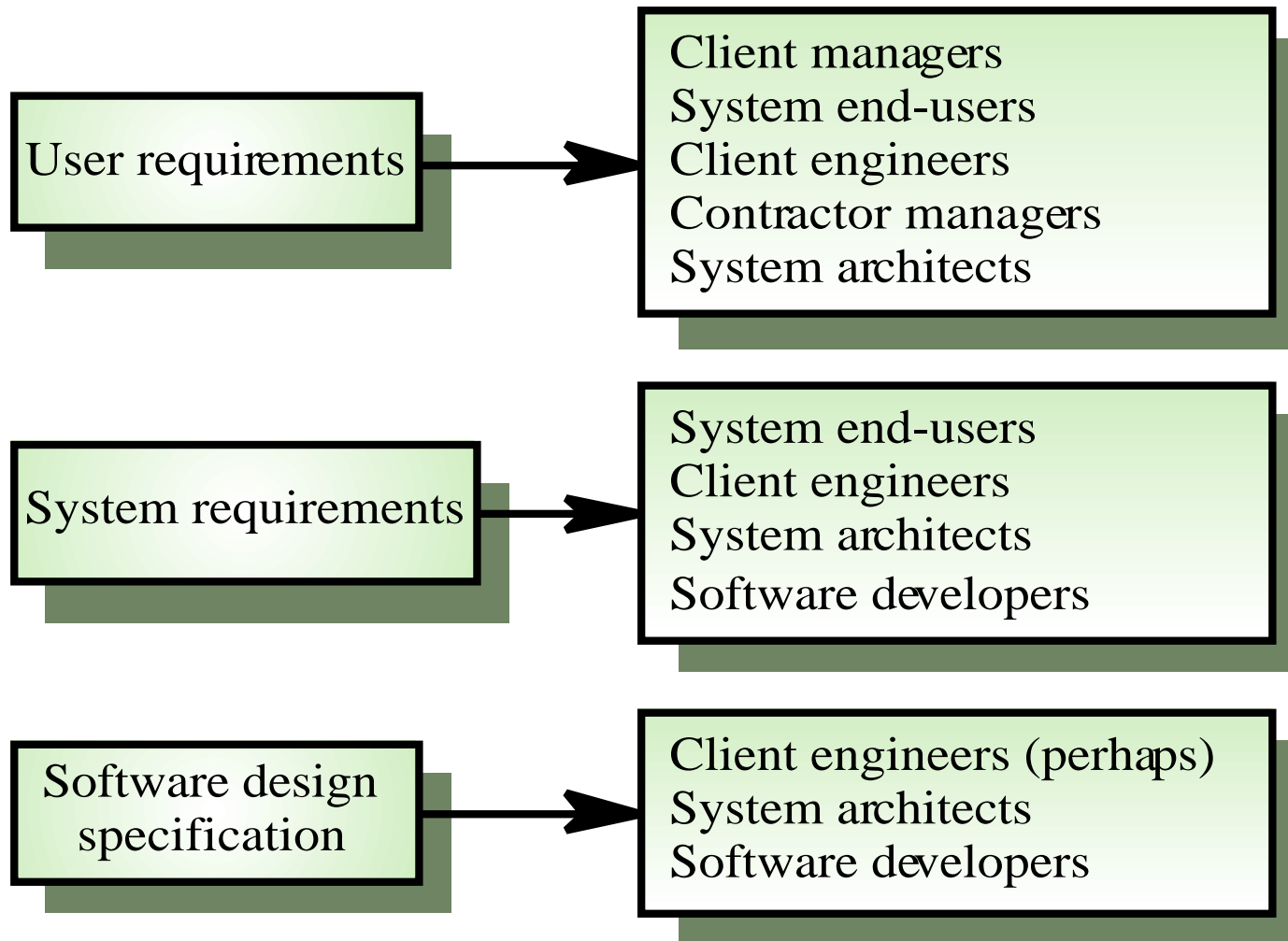
Requirements definition

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

Requirements specification

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

Functional requirements

- Describe functionality or system services
- Depend on the type of software, expected users and the type of system where the software is used
- Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail

Examples of functional 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) which the user shall be able to copy to the account's permanent storage area.

Requirements imprecision

- Problems arise when requirements are not precisely stated
- Ambiguous requirements may be interpreted in different ways by developers and users
- Consider the term ‘appropriate viewers’
 - User intention - special purpose viewer for each different document type
 - Developer interpretation - Provide a text viewer that shows the contents of the document

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

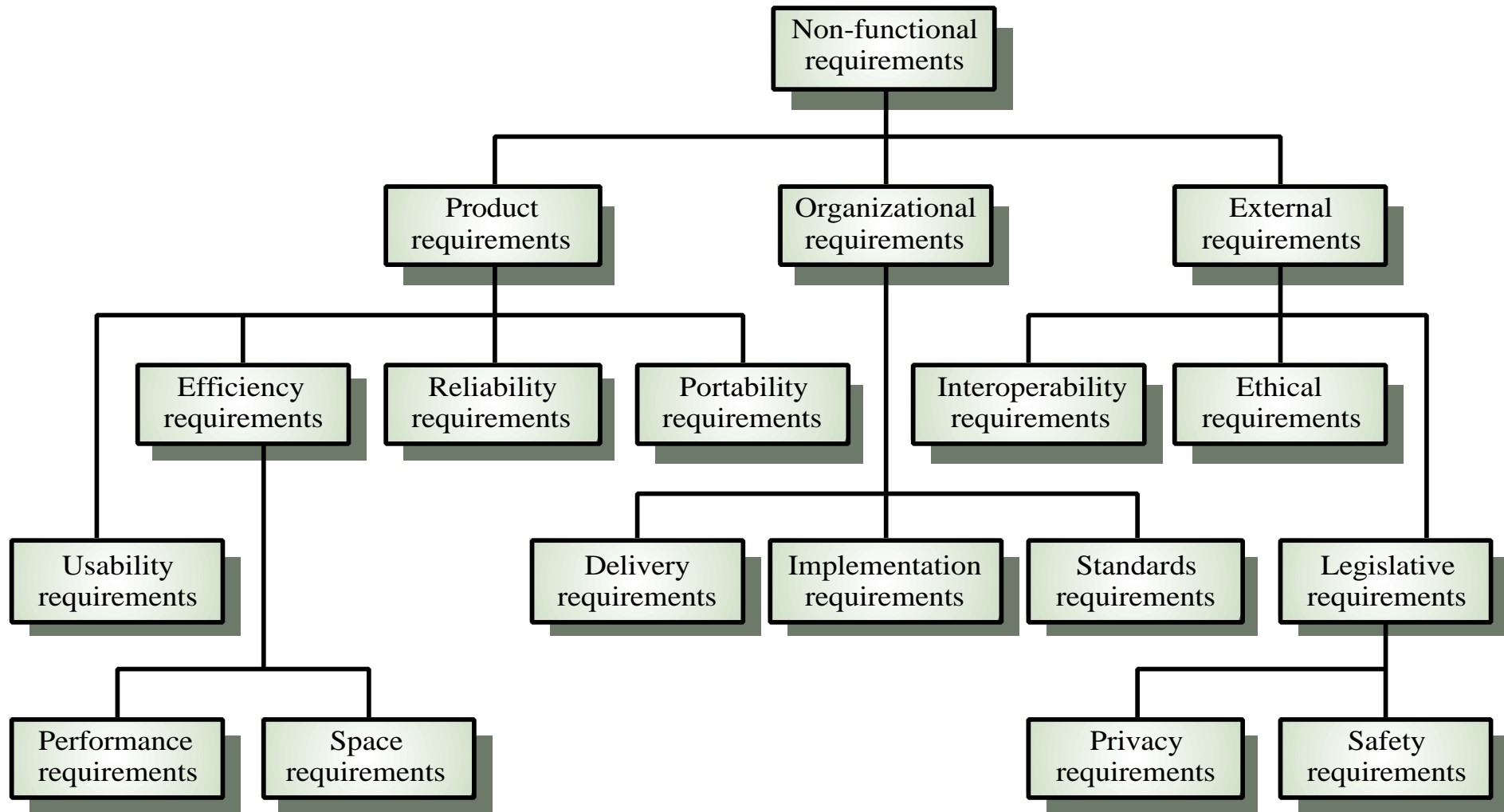
Non-functional requirements

- Define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular CASE system, programming language or development method
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless

Non-functional classifications

- Product requirements
 - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- Organisational requirements
 - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements
 - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Non-functional requirement types



Non-functional requirements examples

- Product requirement
 - 4.C.8 It shall be possible for all necessary communication between the APSE and the user to be expressed in the standard Ada character set
- Organisational requirement
 - 9.3.2 The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95
- External requirement
 - 7.6.5 The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system

Goals and requirements

- Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- Goal
 - A general intention of the user such as ease of use
- Verifiable non-functional requirement
 - A statement using some measure that can be objectively tested
- Goals are helpful to developers as they convey the intentions of the system users

Examples

- **A system goal**
 - The system should be easy to use by experienced controllers and should be organised in such a way that user errors are minimised.
- **A verifiable non-functional requirement**
 - Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day.

Requirements measures

Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	K Bytes Number of RAM 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

Requirements interaction

- Conflicts between different non-functional requirements are common in complex systems
- Spacecraft system
 - To minimise weight, the number of separate chips in the system should be minimised
 - To minimise power consumption, lower power chips should be used
 - However, using low power chips may mean that more chips have to be used. Which is the most critical requirement?

Domain requirements

- Derived from the application domain and describe system characteristics and features that reflect the domain
- May be new functional requirements, constraints on existing requirements or define specific computations
- If domain requirements are not satisfied, the system may be unworkable

Library system domain requirements

- There shall be a standard user interface to all databases which shall be based on the Z39.50 standard.
- Because of copyright restrictions, some documents must be deleted immediately on arrival. Depending on the user's requirements, these documents will either be printed locally on the system server for manually forwarding to the user or routed to a network printer.

Train protection system

- The deceleration of the train shall be computed as:

- $D_{\text{train}} = D_{\text{control}} + D_{\text{gradient}}$

where D_{gradient} is 9.81ms^2 * compensated gradient/alpha and where the values of 9.81ms^2 /alpha are known for different types of train.

Domain requirements problems

- Understandability
 - Requirements are expressed in the language of the application domain
 - This is often not understood by software engineers developing the system
- Implicitness
 - Domain specialists understand the area so well that they do not think of making the domain requirements explicit

User requirements

- Should describe functional and non-functional requirements so that they are understandable by system users who don't have detailed technical knowledge
- User requirements are defined using natural language, tables and diagrams

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

Database requirement

4.A.5 The database shall support the generation and control of configuration objects; that is, objects which are themselves groupings of other objects in the database. The configuration control facilities shall allow access to the objects in a version group by the use of an incomplete name.

Editor grid requirement

2.6 Grid facilities To assist in the positioning of entities on a diagram, the user may turn on a grid in either centimetres or inches, via an option on the control panel. Initially, the grid is off. The grid may be turned on and off at any time during an editing session and can be toggled between inches and centimetres at any time. A grid option will be provided on the reduce-to-fit view but the number of grid lines shown will be reduced to avoid filling the smaller diagram with grid lines.

Requirement problems

- Database requirements includes both conceptual and detailed information
 - Describes the concept of configuration control facilities
 - Includes the detail that objects may be accessed using an incomplete name
- Grid requirement mixes three different kinds of requirement
 - Conceptual functional requirement (the need for a grid)
 - Non-functional requirement (grid units)
 - Non-functional UI requirement (grid switching)

Structured presentation

2.6 Grid facilities

2.6.1 The editor shall provide a grid facility where a matrix of horizontal and vertical lines provide a background to the editor window. This grid shall be a passive grid where the alignment of entities is the user's responsibility.

Rationale: A grid helps the user to create a tidy diagram with well-spaced entities. Although an active grid, where entities 'snap-to' grid lines can be useful, the positioning is imprecise. The user is the best person to decide where entities should be positioned.

Specification: ECLIPSE/WS/Tools/DE/FS Section 5.6

Detailed user requirement

3.5.1 Adding nodes to a design

3.5.1.1 The editor shall provide a facility for users to add nodes of a specified type to their design.

3.5.1.2 The sequence of actions to add a node should be as follows:

1. The user should select the type of node to be added.
2. The user should move the cursor to the approximate node position in the diagram and indicate that the node symbol should be added at that point.
3. The user should then drag the node symbol to its final position.

Rationale: The user is the best person to decide where to position a node on the diagram. This approach gives the user direct control over node type selection and positioning.

Specification: ECLIPSE/WS/Tools/DE/FS. Section 3.5.1

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

System requirements

- More detailed specifications of user requirements
- Serve as a basis for designing the system
- May be used as part of the system contract
- System requirements may be expressed using system models discussed in Chapter 7

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

Problems with NL specification

- Ambiguity
 - The readers and writers of the requirement must interpret the same words in the same way. NL is naturally ambiguous so this is very difficult
- Over-flexibility
 - The same thing may be said in a number of different ways in the specification
- Lack of modularisation
 - NL structures are inadequate to structure system requirements

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.
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 (Ross, 1977; Schoman and Ross, 1977). More recently, use-case descriptions (Jacobsen, Christerson et al., 1993) have been used. I discuss these in the following chapter.
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. I discuss formal specification in Chapter 9.

Structured language specifications

- A limited form of natural language may be used to express requirements
- This removes some of the problems resulting from ambiguity and flexibility and imposes a degree of uniformity on a specification
- Often best supported using a forms-based approach

Form-based specifications

- Definition of the function or entity
- Description of inputs and where they come from
- Description of outputs and where they go to
- Indication of other entities required
- Pre and post conditions (if appropriate)
- The side effects (if any)

Form-based node specification

ECLIPSE/Workstation/Tools/DE/FS/3.5.1

Function Add node

Description Adds a node to an existing design. The user selects the type of node, and its position. When added to the design, the node becomes the current selection. The user chooses the node position by moving the cursor to the area where the node is added.

Inputs Node type, Node position, Design identifier.

Source Node type and Node position are input by the user, Design identifier from the database.

Outputs Design identifier.

Destination The design database. The design is committed to the database on completion of the operation.

Requires Design graph rooted at input design identifier.

Pre-condition The design is open and displayed on the user's screen.

Post-condition The design is unchanged apart from the addition of a node of the specified type at the given position.

Side-effects None

Definition: ECLIPSE/Workstation/Tools/DE/RD/3.5.1

PDL-based requirements definition

- Requirements may be defined operationally using a language like a programming language but with more flexibility of expression
- Most appropriate in two situations
 - Where an operation is specified as a sequence of actions and the order is important
 - When hardware and software interfaces have to be specified
- Disadvantages are
 - The PDL may not be sufficiently expressive to define domain concepts
 - The specification will be taken as a design rather than a specification

Part of an ATM specification

```
class ATM {  
    // declarations here  
    public static void main (String args[]) throws InvalidCard {  
        try {  
            thisCard.read () ; // may throw InvalidCard exception  
            pin = KeyPad.readPin () ; attempts = 1 ;  
            while ( !thisCard.pin.equals (pin) & attempts < 4 )  
                {  
                    pin = KeyPad.readPin () ; attempts = attempts + 1 ;  
                }  
            if (!thisCard.pin.equals (pin))  
                throw new InvalidCard ("Bad PIN");  
            thisBalance = thisCard.getBalance () ;  
            do { Screen.prompt (" Please select a service ") ;  
                service = Screen.touchKey () ;  
                switch (service) {  
                    case Services.withdrawalWithReceipt:  
                        receiptRequired = true ;  
                }  
            } while (service != Services.exit) ;  
        }  
    }  
}
```

PDL disadvantages

- PDL may not be sufficiently expressive to express the system functionality in an understandable way
- Notation is only understandable to people with programming language knowledge
- The requirement may be taken as a design specification rather than a model to help understand the system

Interface specification

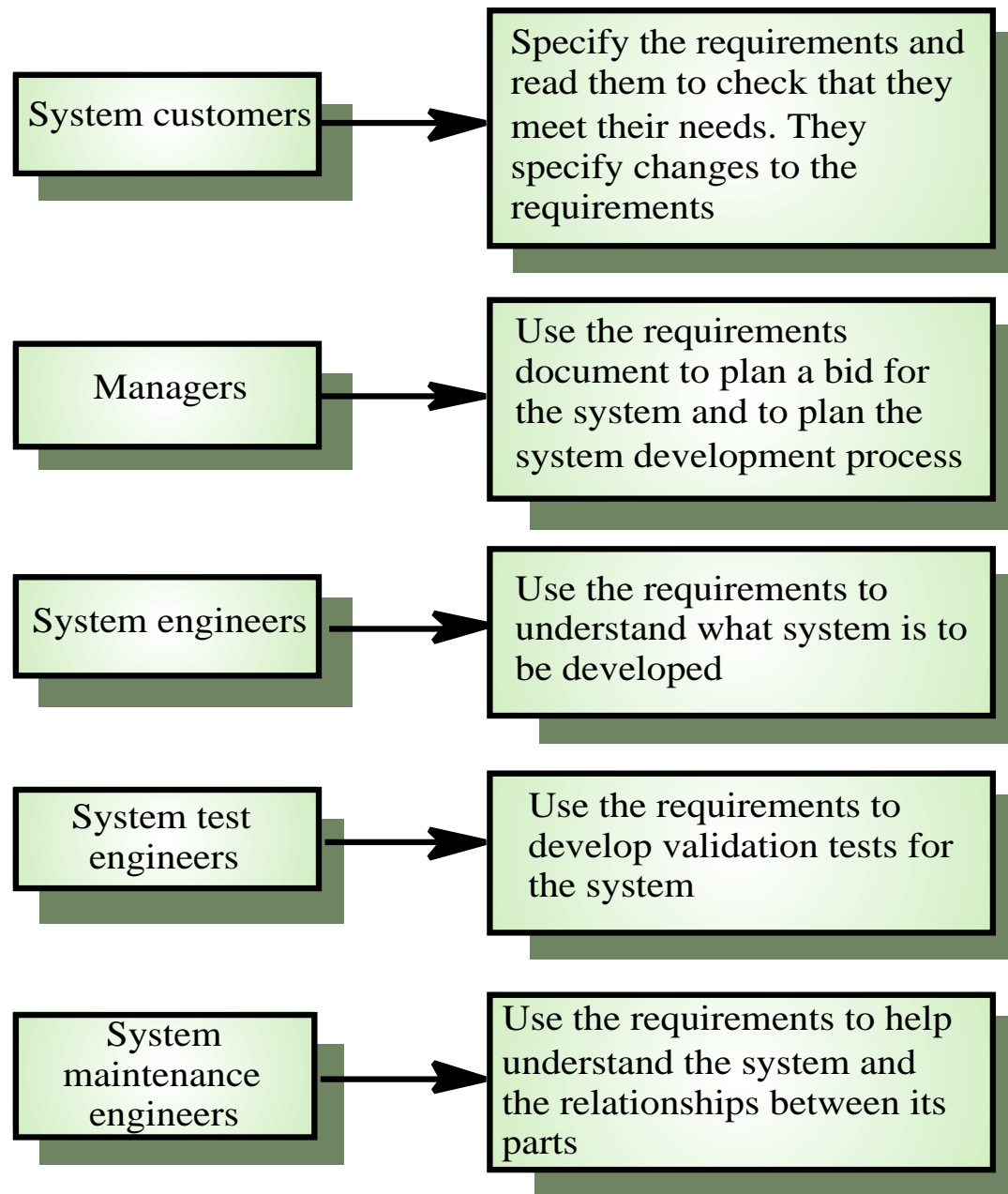
- Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements
- Three types of interface may have to be defined
 - Procedural interfaces
 - Data structures that are exchanged
 - Data representations
- Formal notations are an effective technique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

Requirements document requirements

- Specify external system behaviour
- Specify implementation constraints
- Easy to change
- Serve as reference tool for maintenance
- Record forethought about the life cycle of the system
i.e. predict changes
- Characterise responses to unexpected events

IEEE requirements standard

- Introduction
- General description
- Specific requirements
- Appendices
- Index
- This is a generic structure that must be instantiated for specific systems

Requirements document structure

- 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

Key points

- User requirements should be written in natural language, tables and diagrams
- System requirements are intended to communicate the functions that the system should provide
- System requirements may be written in structured natural language, a PDL or in a formal language
- A software requirements document is an agreed statement of the system requirements

Requirements Engineering Processes

- Processes used to discover, analyse and validate system requirements

Objectives

- To describe the principal requirements engineering activities
- To introduce techniques for requirements elicitation and analysis
- To describe requirements validation
- To discuss the role of requirements management in support of other requirements engineering processes

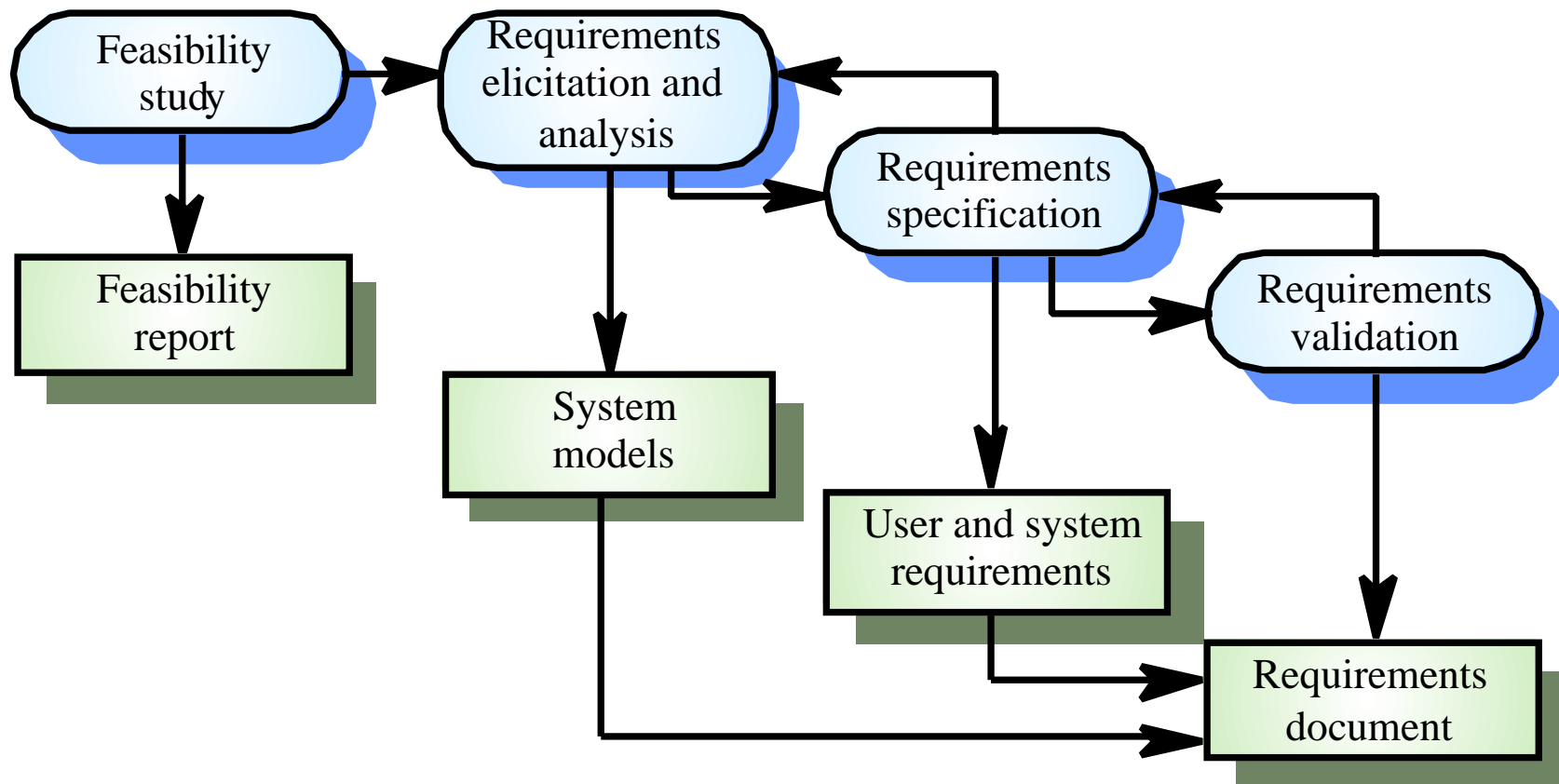
Topics covered

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

Requirements engineering processes

- The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements
- However, there are a number of generic activities common to all processes
 - Requirements elicitation
 - Requirements 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

Feasibility study implementation

- Based on information assessment (what is required), information collection and report writing
- Questions for people in the organisation
 - What if the system wasn't implemented?
 - What are current process problems?
 - How will the proposed system help?
 - What will be the integration problems?
 - Is new technology needed? What skills?
 - What facilities must be supported by the proposed system?

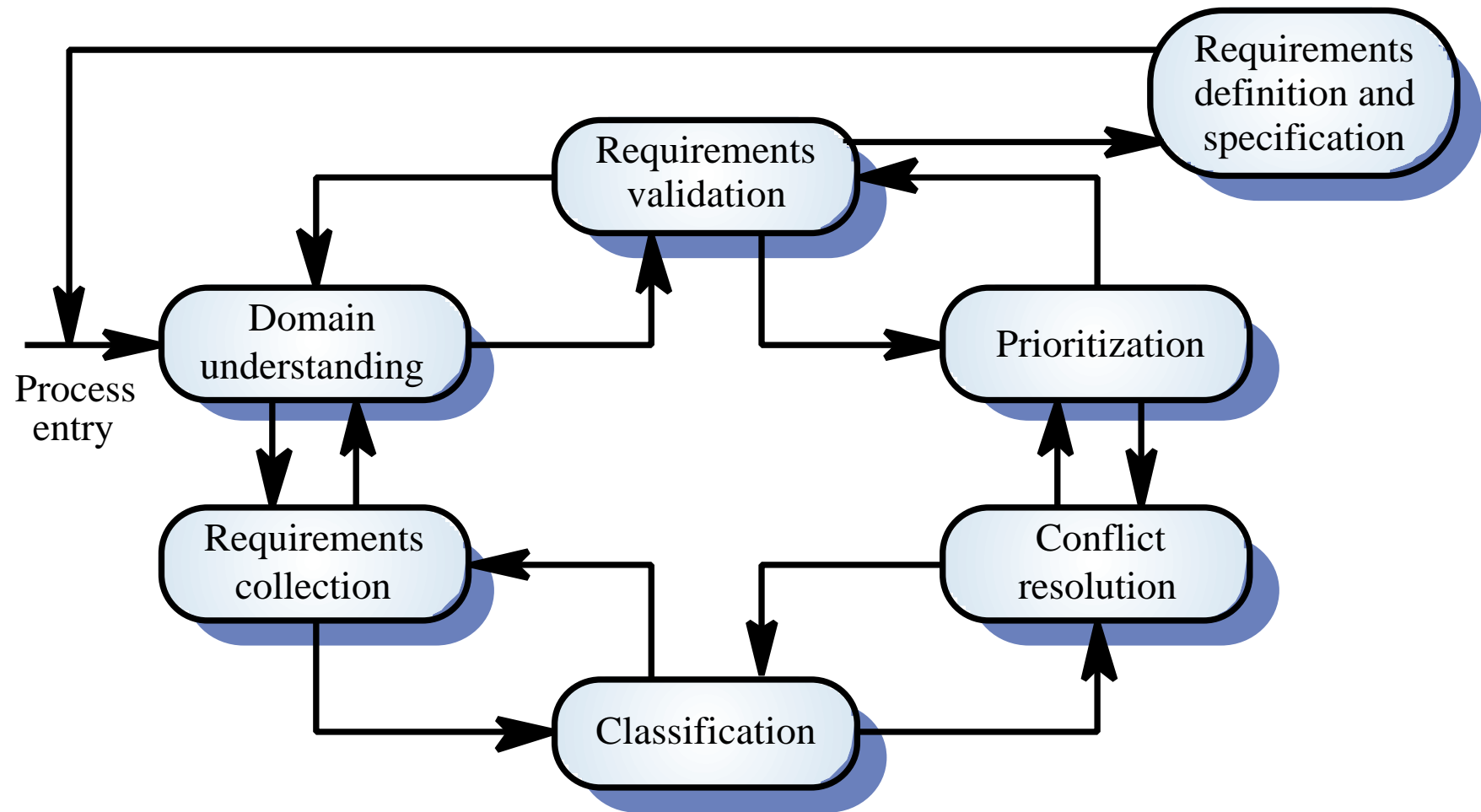
Elicitation and analysis

- 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

The requirements analysis process



Process activities

- Domain understanding
- Requirements collection
- Classification
- Conflict resolution
- Prioritisation
- Requirements checking

System models

- Different models may be produced during the requirements analysis activity
- Requirements analysis may involve three structuring activities which result in these different models
 - Partitioning. Identifies the structural (part-of) relationships between entities
 - Abstraction. Identifies generalities among entities
 - Projection. Identifies different ways of looking at a problem
- System models covered in Chapter 7

Viewpoint-oriented elicitation

- Stakeholders represent different ways of looking at a problem or problem viewpoints
- This multi-perspective analysis is important as there is no single correct way to analyse system requirements

Banking ATM system

- The example used here is an auto-teller system which provides some automated banking services
- I use a very simplified system which offers some services to customers of the bank who own the system and a narrower range of services to other customers
- Services include cash withdrawal, message passing (send a message to request a service), ordering a statement and transferring funds

Autoteller viewpoints

- Bank customers
- Representatives of other banks
- Hardware and software maintenance engineers
- Marketing department
- Bank managers and counter staff
- Database administrators and security staff
- Communications engineers
- Personnel department

Types of viewpoint

- Data sources or sinks
 - Viewpoints are responsible for producing or consuming data. Analysis involves checking that data is produced and consumed and that assumptions about the source and sink of data are valid
- Representation frameworks
 - Viewpoints represent particular types of system model. These may be compared to discover requirements that would be missed using a single representation. Particularly suitable for real-time systems
- Receivers of services
 - Viewpoints are external to the system and receive services from it. Most suited to interactive systems

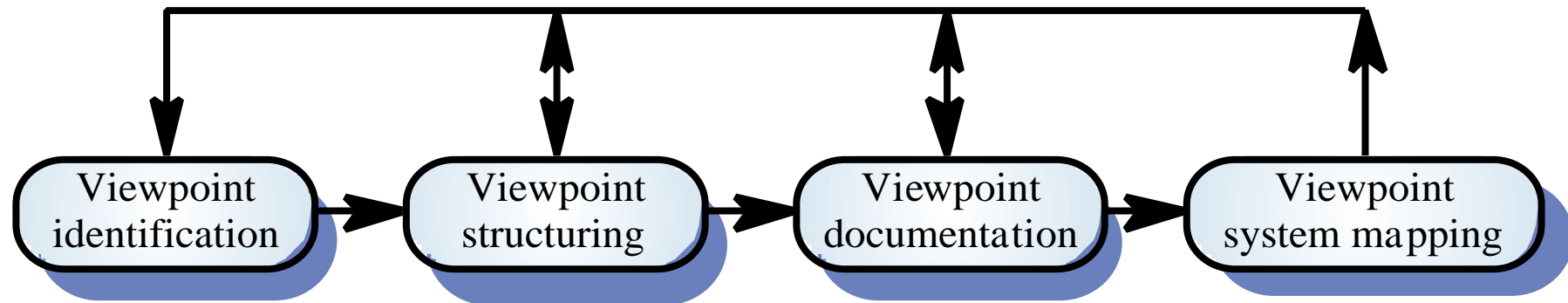
External viewpoints

- Natural to think of end-users as receivers of system services
- Viewpoints are a natural way to structure requirements elicitation
- It is relatively easy to decide if a viewpoint is valid
- Viewpoints and services may be used to structure non-functional requirements

Method-based analysis

- Widely used approach to requirements analysis. Depends on the application of a structured method to understand the system
- Methods have different emphases. Some are designed for requirements elicitation, others are close to design methods
- A viewpoint-oriented method (VORD) is used as an example here. It also illustrates the use of viewpoints

The VORD method



VORD process model

- Viewpoint identification
 - Discover viewpoints which receive system services and identify the services provided to each viewpoint
- Viewpoint structuring
 - Group related viewpoints into a hierarchy. Common services are provided at higher-levels in the hierarchy
- Viewpoint documentation
 - Refine the description of the identified viewpoints and services
- Viewpoint-system mapping
 - Transform the analysis to an object-oriented design

VORD standard forms

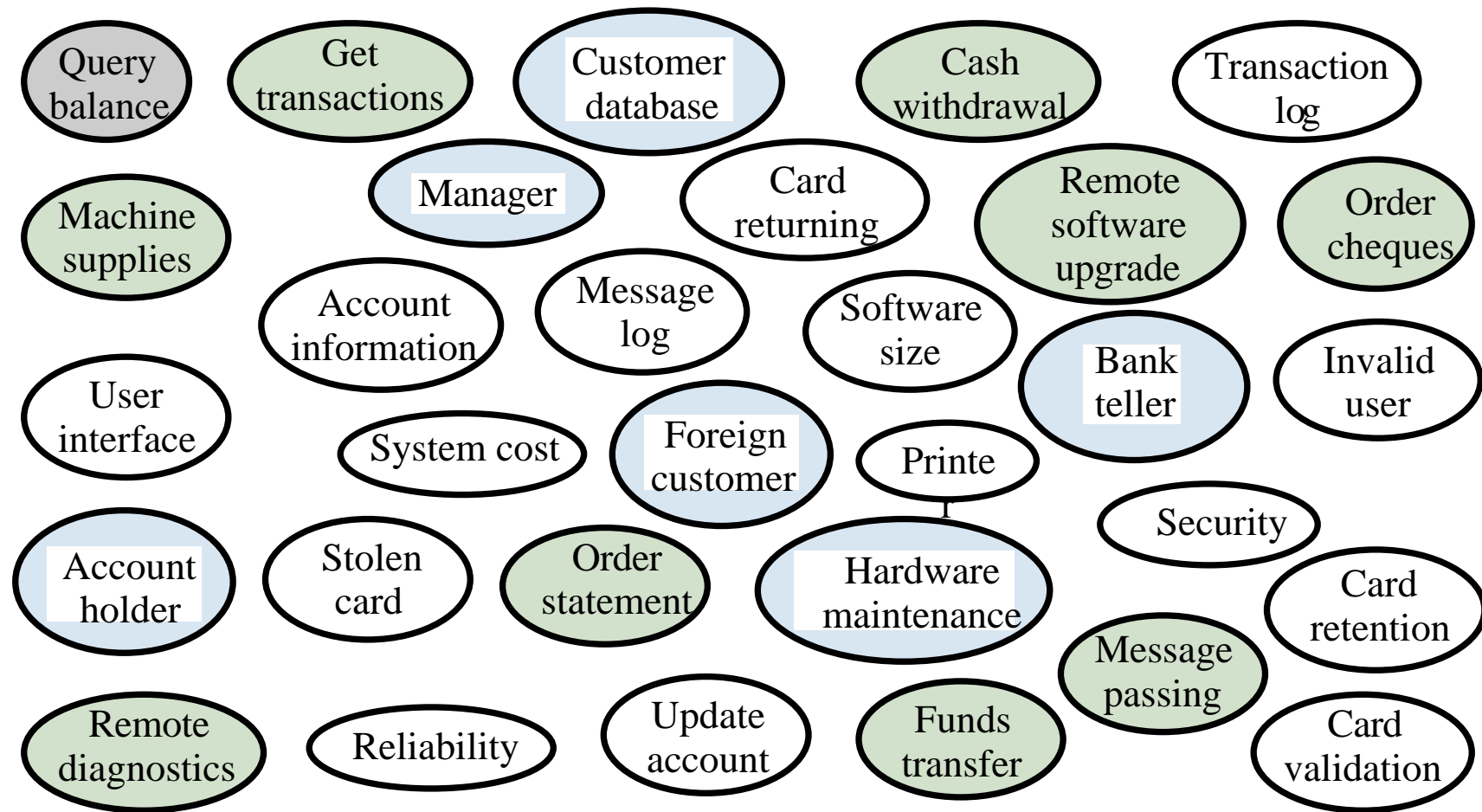
Viewpoint template

Reference:	The viewpoint name.
Attributes:	Attributes providing viewpoint information.
Events:	A reference to a set of event scenarios describing how the system reacts to viewpoint events.
Services	A reference to a set of service descriptions.
Sub-VPs:	The names of sub-viewpoints.

Service template

Reference:	The service name.
Rationale:	Reason why the service is provided.
Specification:	Reference to a list of service specifications. These may be expressed in different notations.
Viewpoints:	List of viewpoint names receiving the service.
Non-functional requirements:	Reference to a set of non-functional requirements which constrain the service.
Provider:	Reference to a list of system objects which provide the service.

Viewpoint identification



Viewpoint service information

ACCOUNT HOLDER

Service list

Withdraw cash
Query balance
Order cheques
Send message
Transaction list
Order statement
Transfer funds

FOREIGN CUSTOMER

Service list

Withdraw cash
Query balance

BANK TELLER

Service list

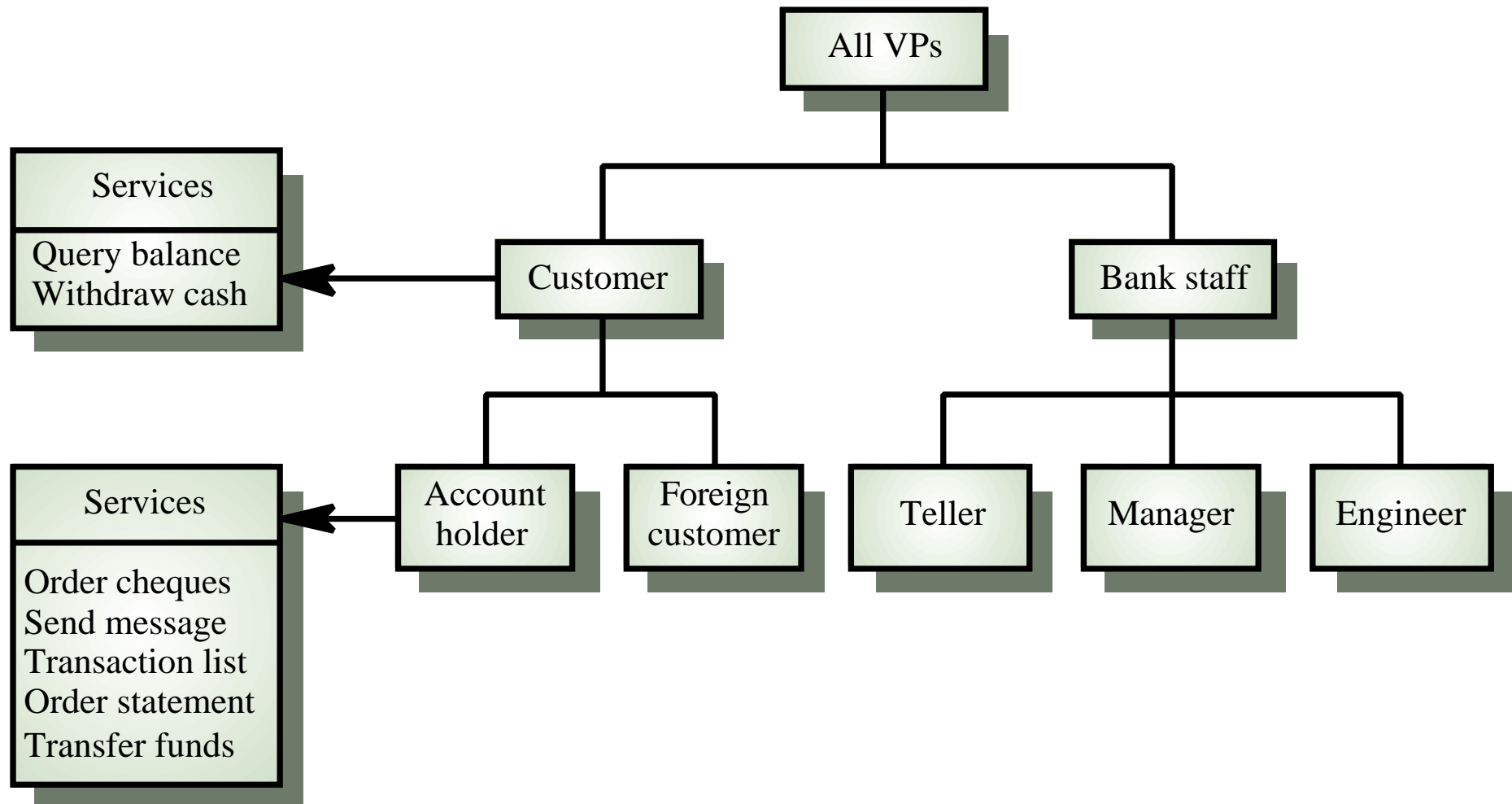
Run diagnostics
Add cash
Add paper
Send message

Viewpoint data/control

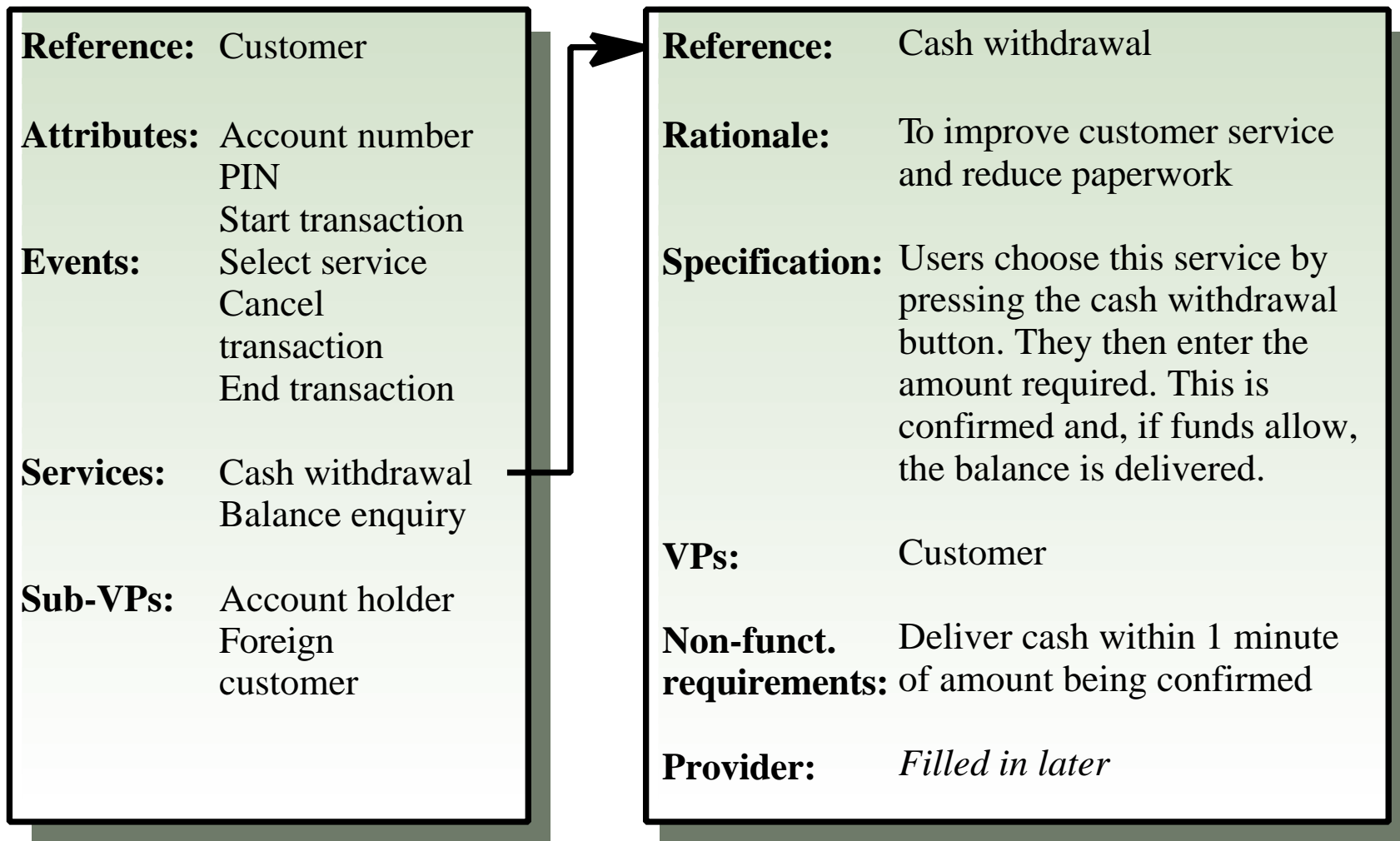
ACCOUNT
HOLDER

Control input	Data input
Start transaction Cancel transaction End transaction Select service	Card details PIN Amount required Message

Viewpoint hierarchy



Customer/cash withdrawal templates



Scenarios

- Scenarios are descriptions of how a system is used in practice
- They are helpful in requirements elicitation as people can relate to these more readily than abstract statement of what they require from a system
- Scenarios are particularly useful for adding detail to an outline requirements description

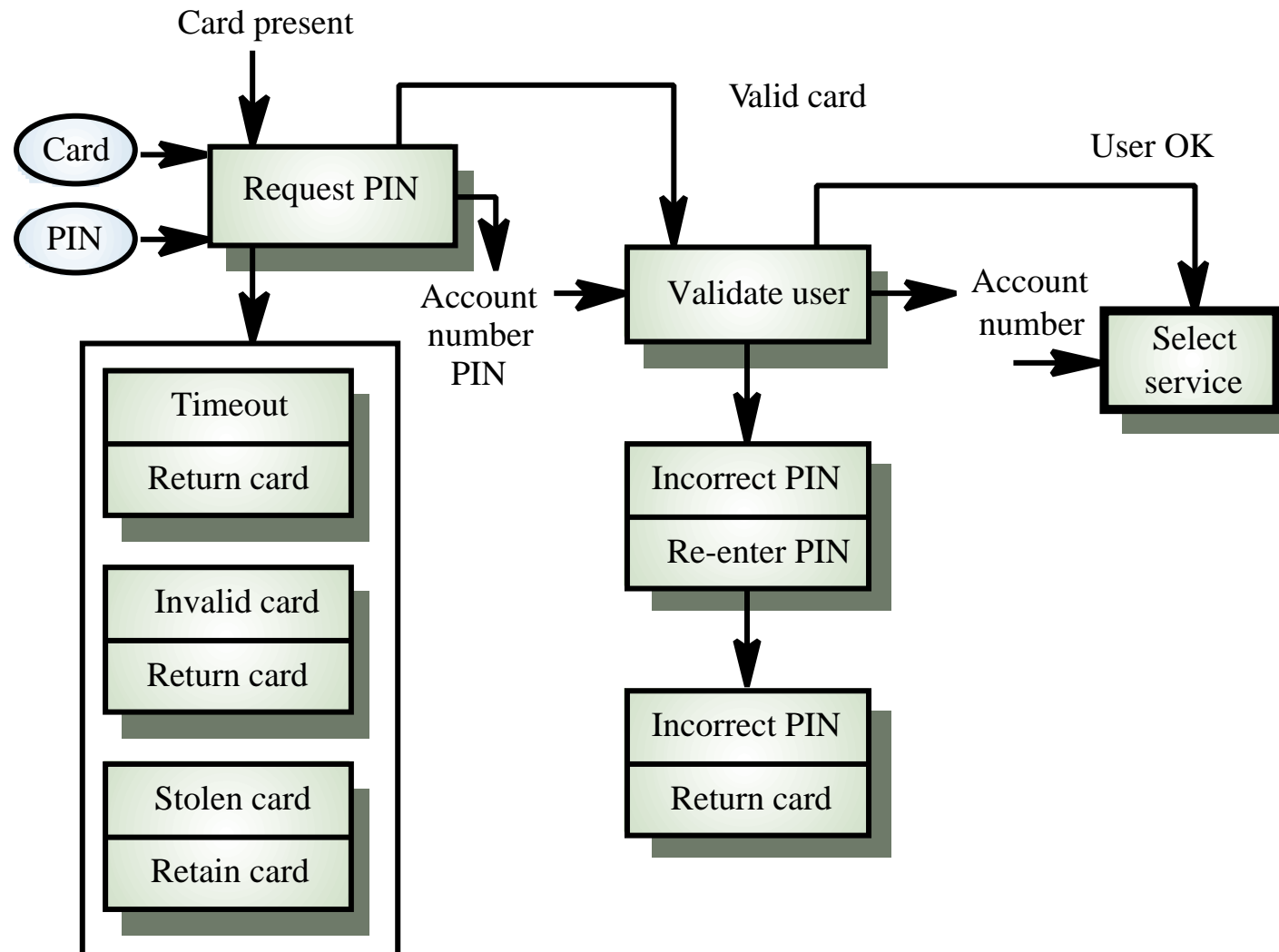
Scenario descriptions

- System state at the beginning of the scenario
- Normal flow of events in the scenario
- What can go wrong and how this is handled
- Other concurrent activities
- System state on completion of the scenario

Event scenarios

- Event scenarios may be used to describe how a system responds to the occurrence of some particular event such as ‘start transaction’
- VORD includes a diagrammatic convention for event scenarios.
 - Data provided and delivered
 - Control information
 - Exception processing
 - The next expected event

Event scenario - start transaction



Notation for data and control analysis

- Ellipses. data provided from or delivered to a viewpoint
- Control information enters and leaves at the top of each box
- Data leaves from the right of each box
- Exceptions are shown at the bottom of each box
- Name of next event is in box with thick edges

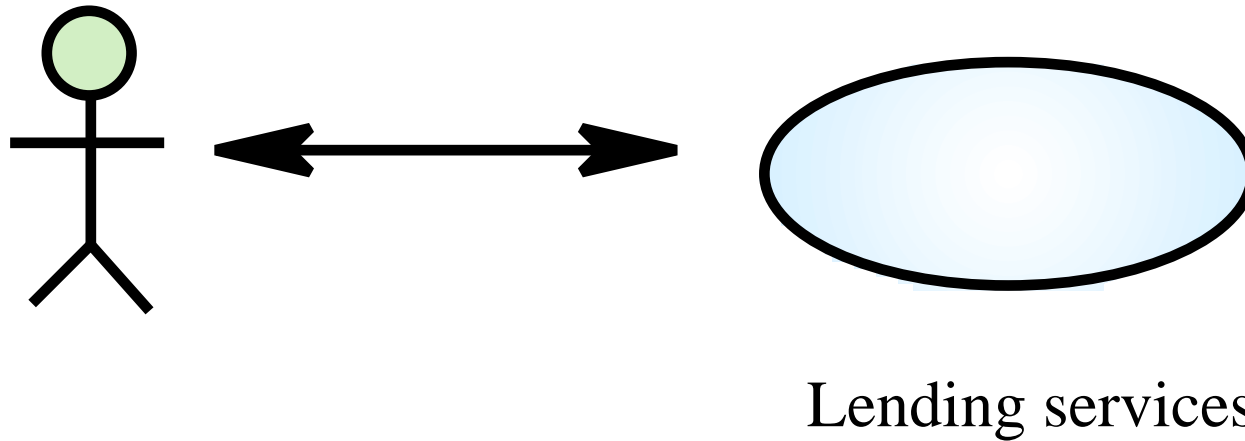
Exception description

- Most methods do not include facilities for describing exceptions
- In this example, exceptions are
 - Timeout. Customer fails to enter a PIN within the allowed time limit
 - Invalid card. The card is not recognised and is returned
 - Stolen card. The card has been registered as stolen and is retained by the machine

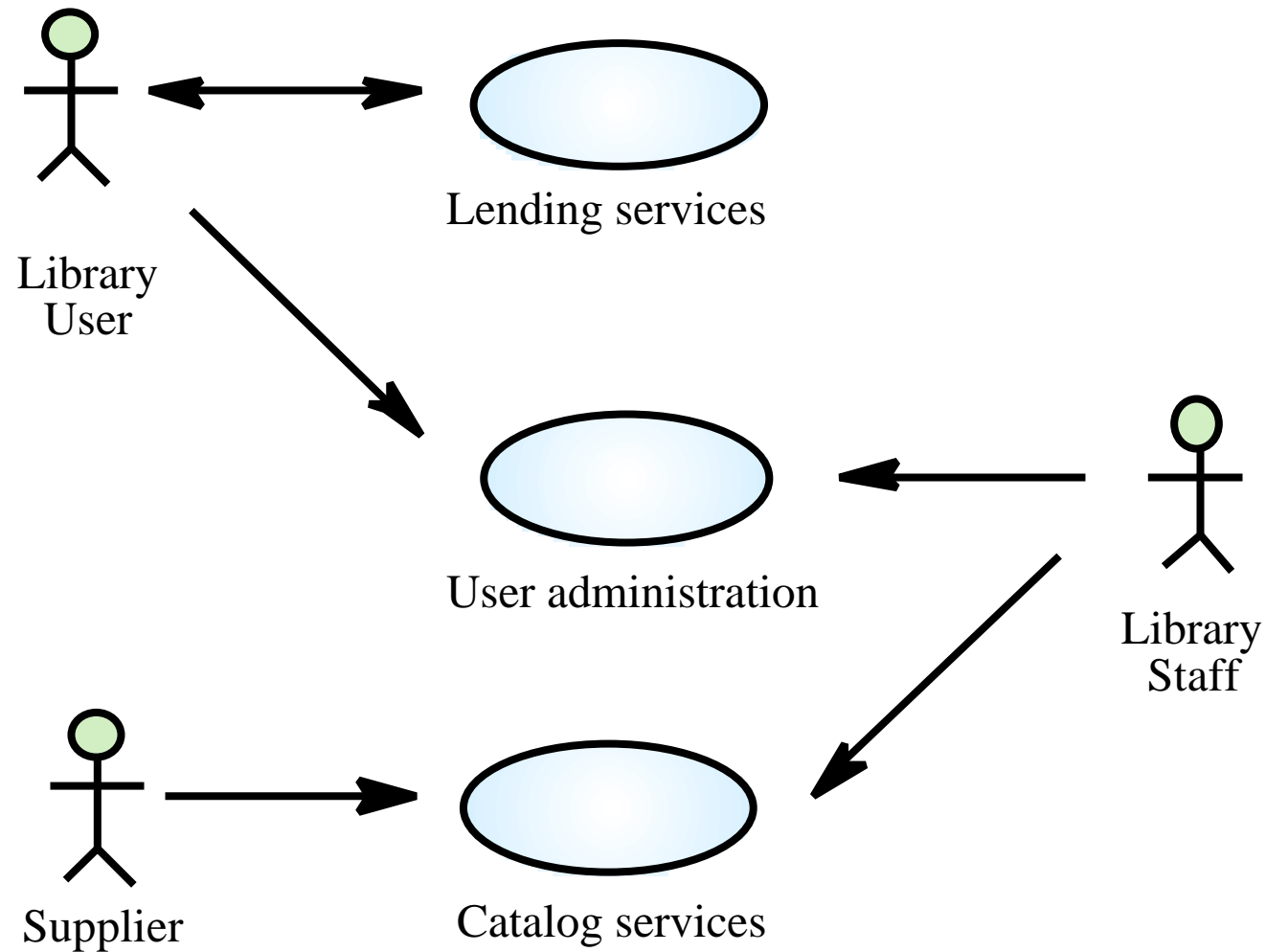
Use cases

- 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

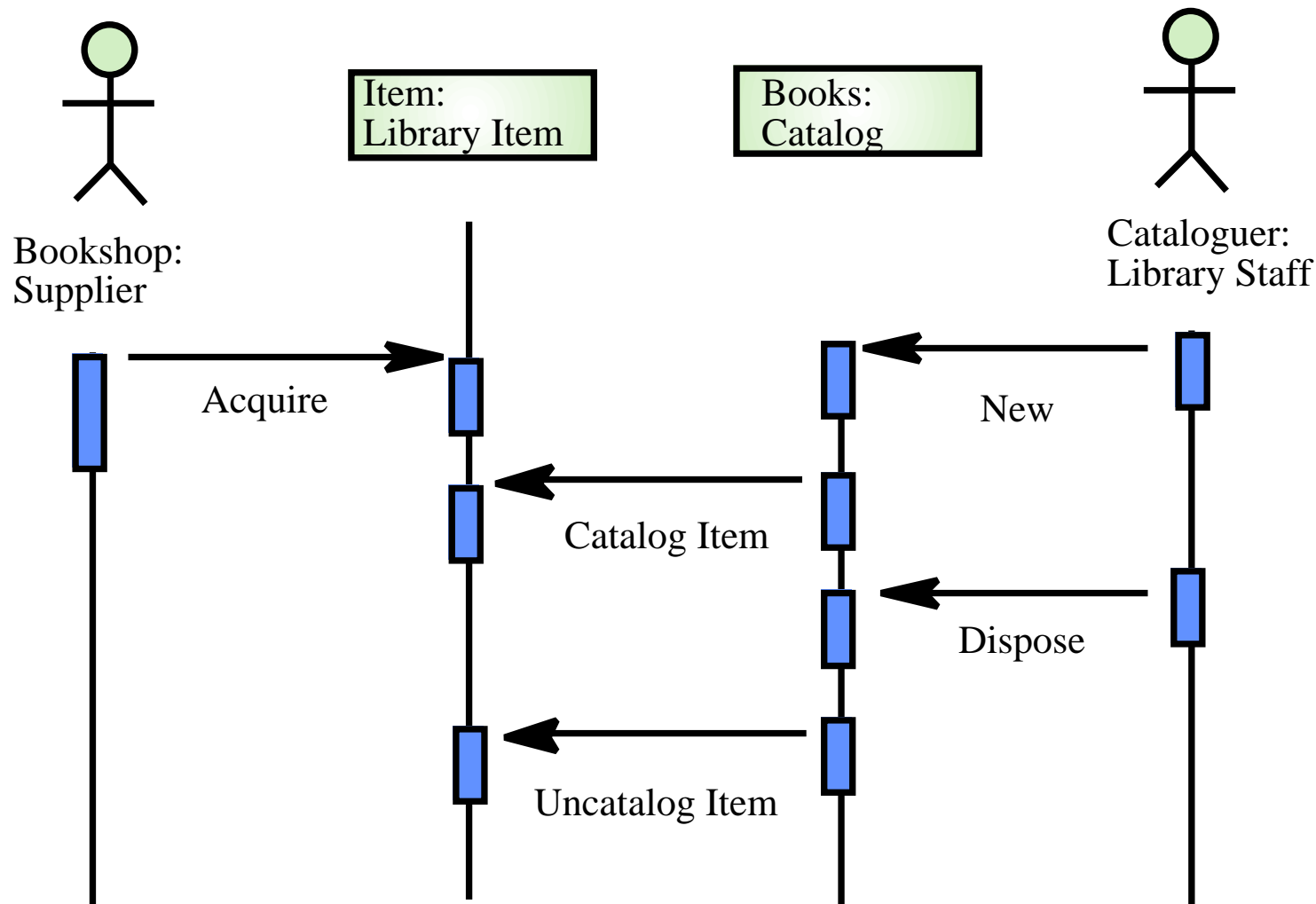
Lending use-case



Library use-cases



Catalogue management



Social and organisational factors

- Software systems are used in a social and organisational context. This can influence or even dominate the system requirements
- Social and organisational factors are not a single viewpoint but are influences on all viewpoints
- Good analysts must be sensitive to these factors but currently no systematic way to tackle their analysis

Example

- Consider a system which allows senior management to access information without going through middle managers
 - Managerial status. Senior managers may feel that they are too important to use a keyboard. This may limit the type of system interface used
 - Managerial responsibilities. Managers may have no uninterrupted time where they can learn to use the system
 - Organisational resistance. Middle managers who will be made redundant may deliberately provide misleading or incomplete information so that the system will fail

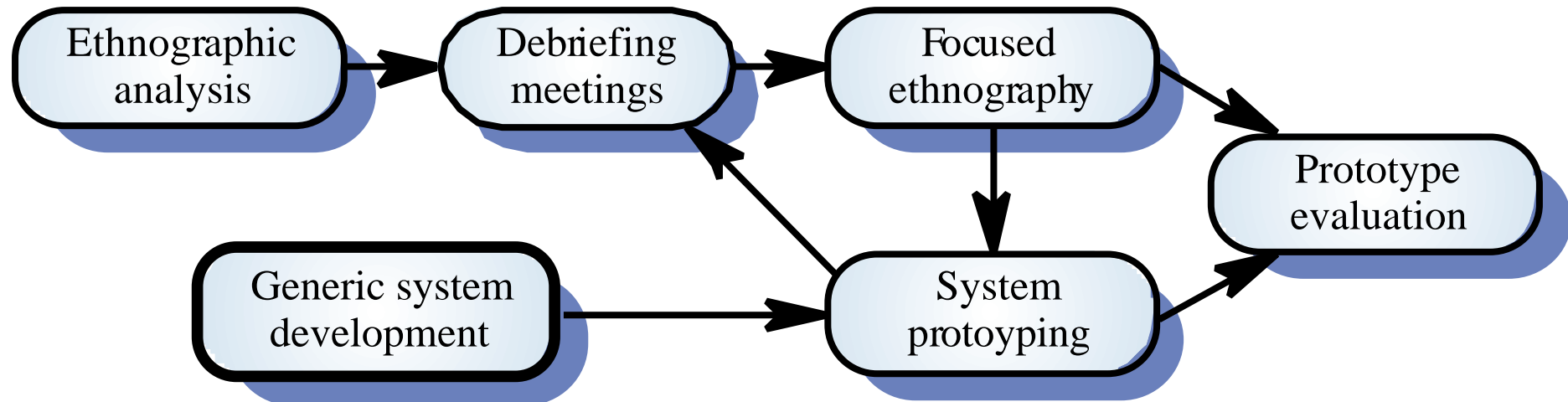
Ethnography

- A social scientists 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

Focused ethnography

- Developed in a project studying the air traffic control process
- Combines ethnography with prototyping
- Prototype development results in unanswered questions which focus the ethnographic analysis
- Problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant

Ethnography and prototyping



Scope of ethnography

- Requirements that are derived from the way that people actually work rather than the way I which process definitions suggest that they ought to work
- Requirements that are derived from cooperation and awareness of other people's activities

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 8
- Test-case generation
 - Developing tests for requirements to check testability
- Automated consistency analysis
 - Checking the consistency of a structured requirements description

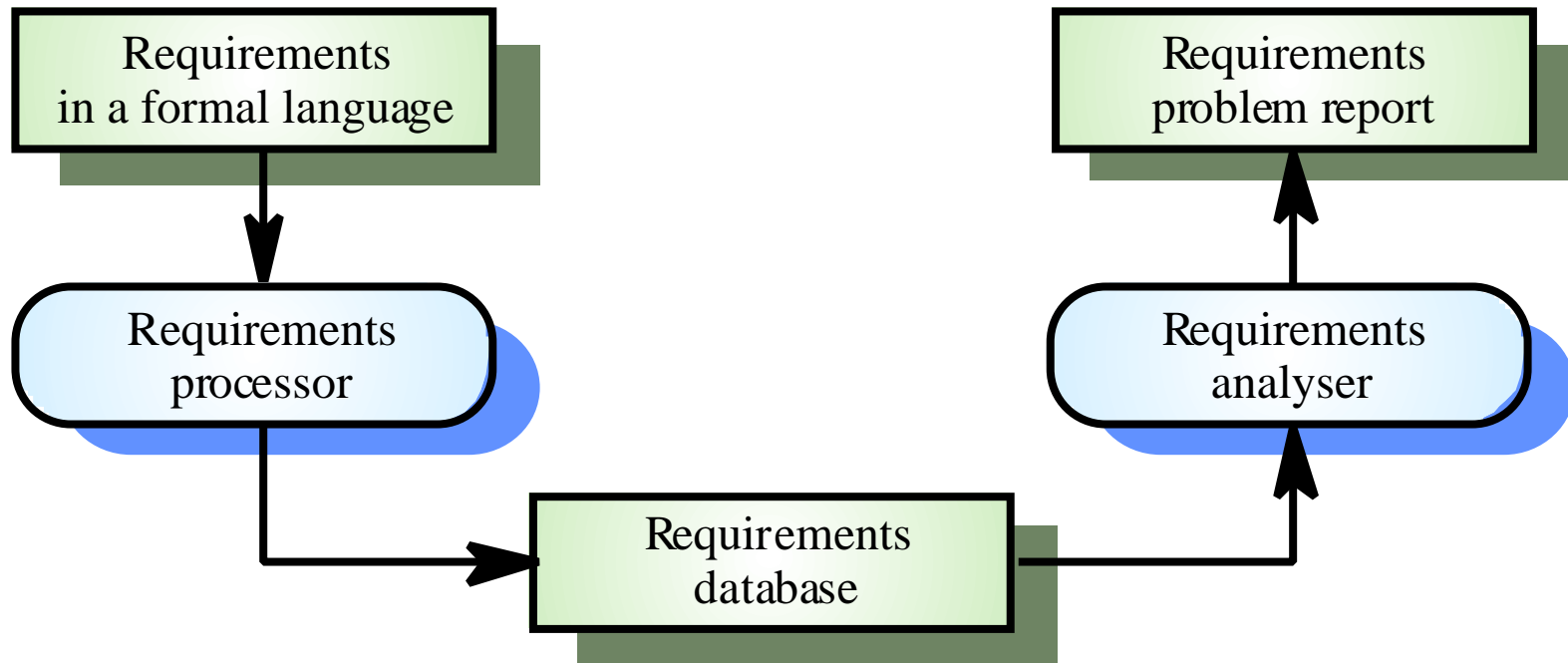
Requirements reviews

- Regular reviews should be held while the requirements definition is being formulated
- Both client and contractor staff should be involved in reviews
- Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage

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?

Automated consistency checking



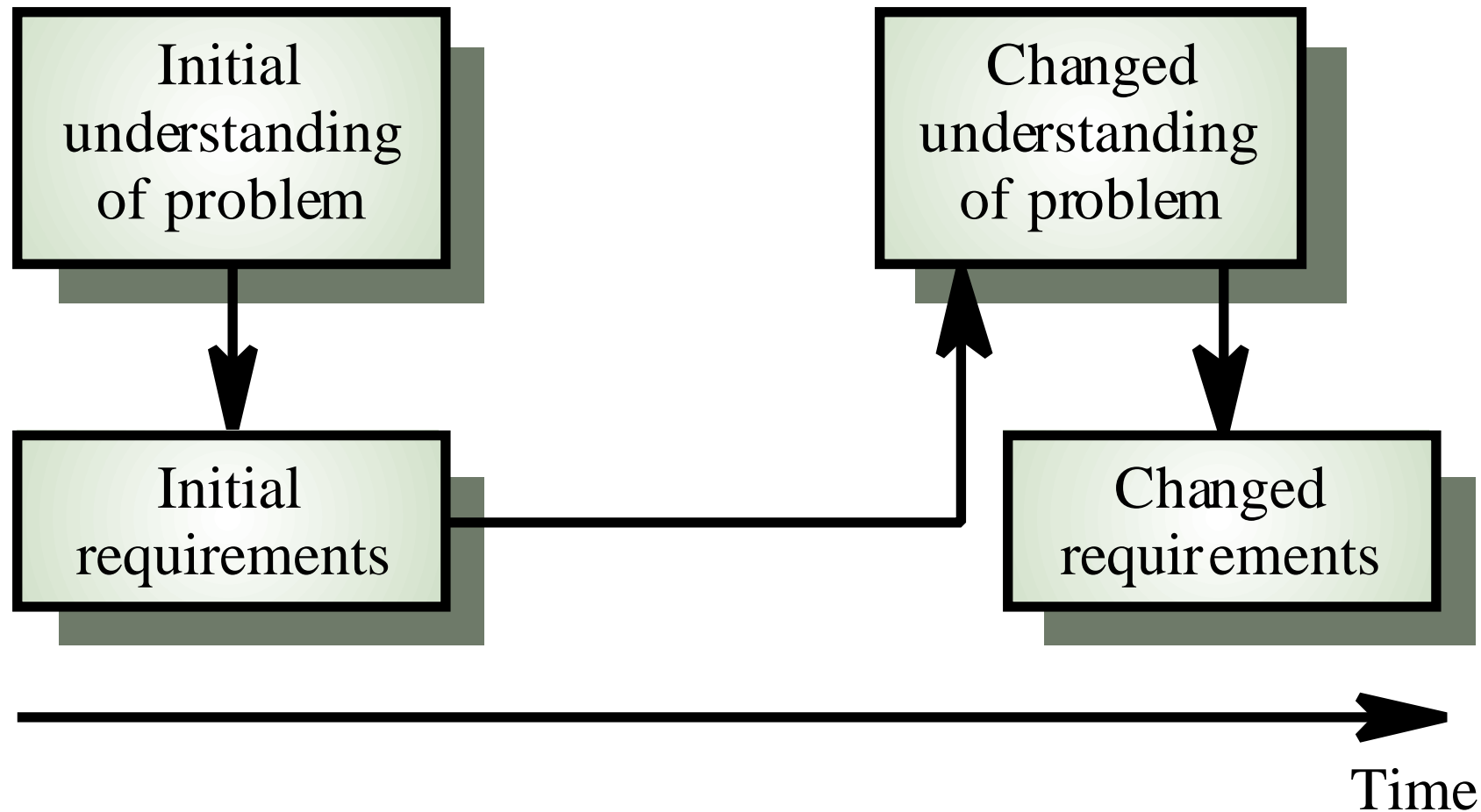
Requirements management

- 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 change

- The priority of requirements from different viewpoints changes during the development process
- System customers may specify requirements from a business perspective that conflict with end-user requirements
- The business and technical environment of the system changes during its development

Requirements evolution



Enduring and volatile requirements

- Enduring requirements. Stable requirements derived from the core activity of the customer organisation. E.g. a hospital will always have doctors, nurses, etc. May be derived from domain models
- Volatile requirements. Requirements which change during development or when the system is in use. In a hospital, requirements derived from health-care policy

Classification of requirements

- Mutable requirements
 - Requirements that change due to the system's environment
- Emergent requirements
 - Requirements that emerge as understanding of the system develops
- Consequential requirements
 - Requirements that result from the introduction of the computer system
- Compatibility requirements
 - Requirements that depend on other systems or organisational processes

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		U	R					
1.2			U			R		U
1.3	R			R				
2.1			R		U			U
2.2								U
2.3		R		U				
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

Requirements change management

- Should apply to all proposed changes to the requirements
- Principal stages
 - Problem analysis. Discuss requirements problem and propose change
 - Change analysis and costing. Assess effects of change on other requirements
 - Change implementation. Modify requirements document and other documents to reflect change

Requirements change management



Key points

- The requirements engineering process includes a feasibility study, requirements elicitation and analysis, requirements specification and requirements management
- Requirements 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

System models

- Abstract descriptions of systems whose requirements are being analysed

Objectives

- To explain why the context of a system should be modelled as part of the RE process
- To describe behavioural modelling, data modelling and object modelling
- To introduce some of the notations used in the Unified Modeling Language (UML)
- To show how CASE workbenches support system modelling

Topics covered

- Context models
- Behavioural models
- Data models
- Object models
- CASE workbenches

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

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

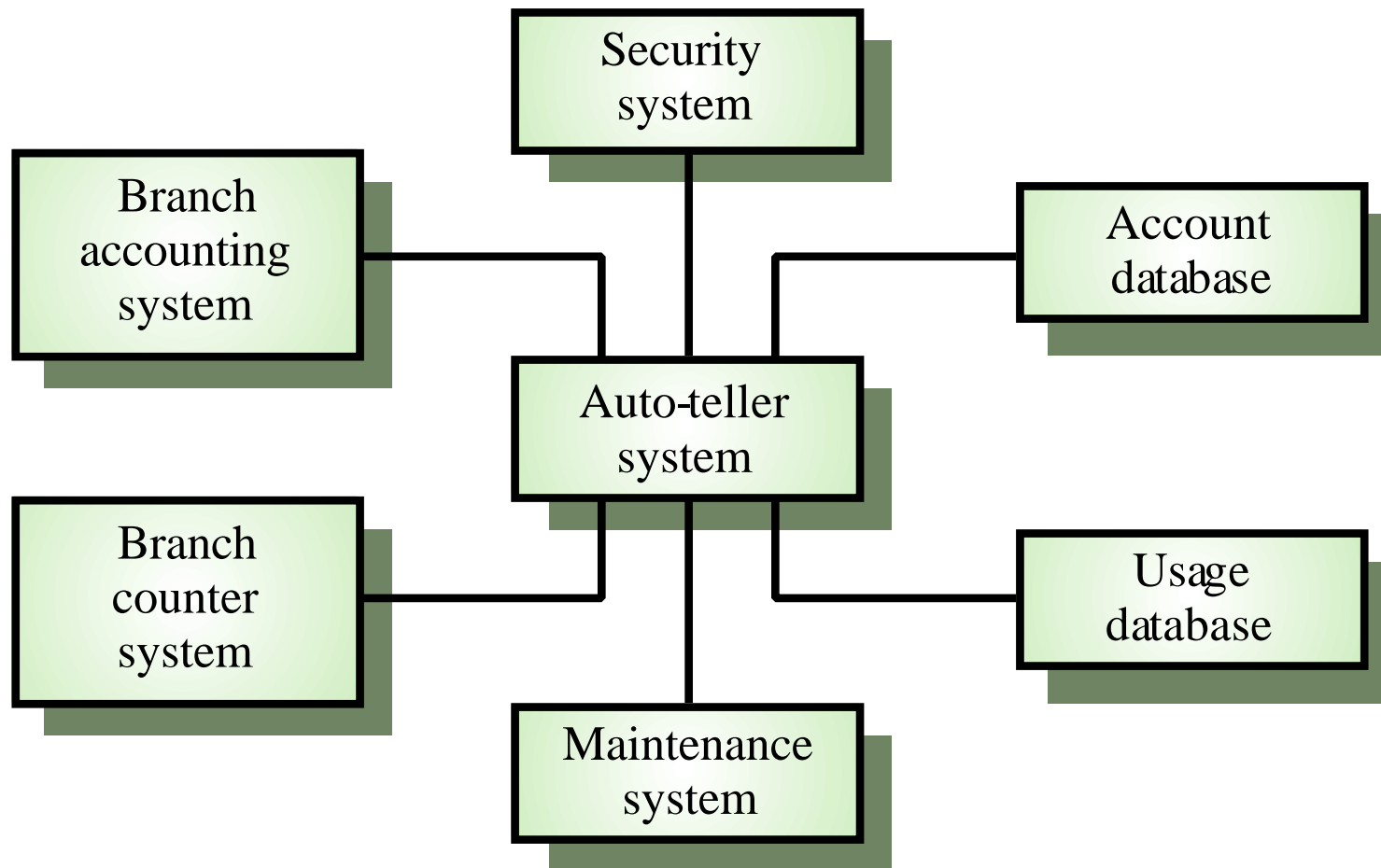
Model types

- Data processing model showing how the data is processed at different stages
- Composition model showing how entities are composed of other entities
- Architectural model showing principal sub-systems
- Classification model showing how entities have common characteristics
- Stimulus/response model showing the system's reaction to events

Context models

- Context models are used to illustrate the boundaries of a system
- Social and organisational concerns may affect the decision on where to position system boundaries
- Architectural models show the a system and its relationship with other systems

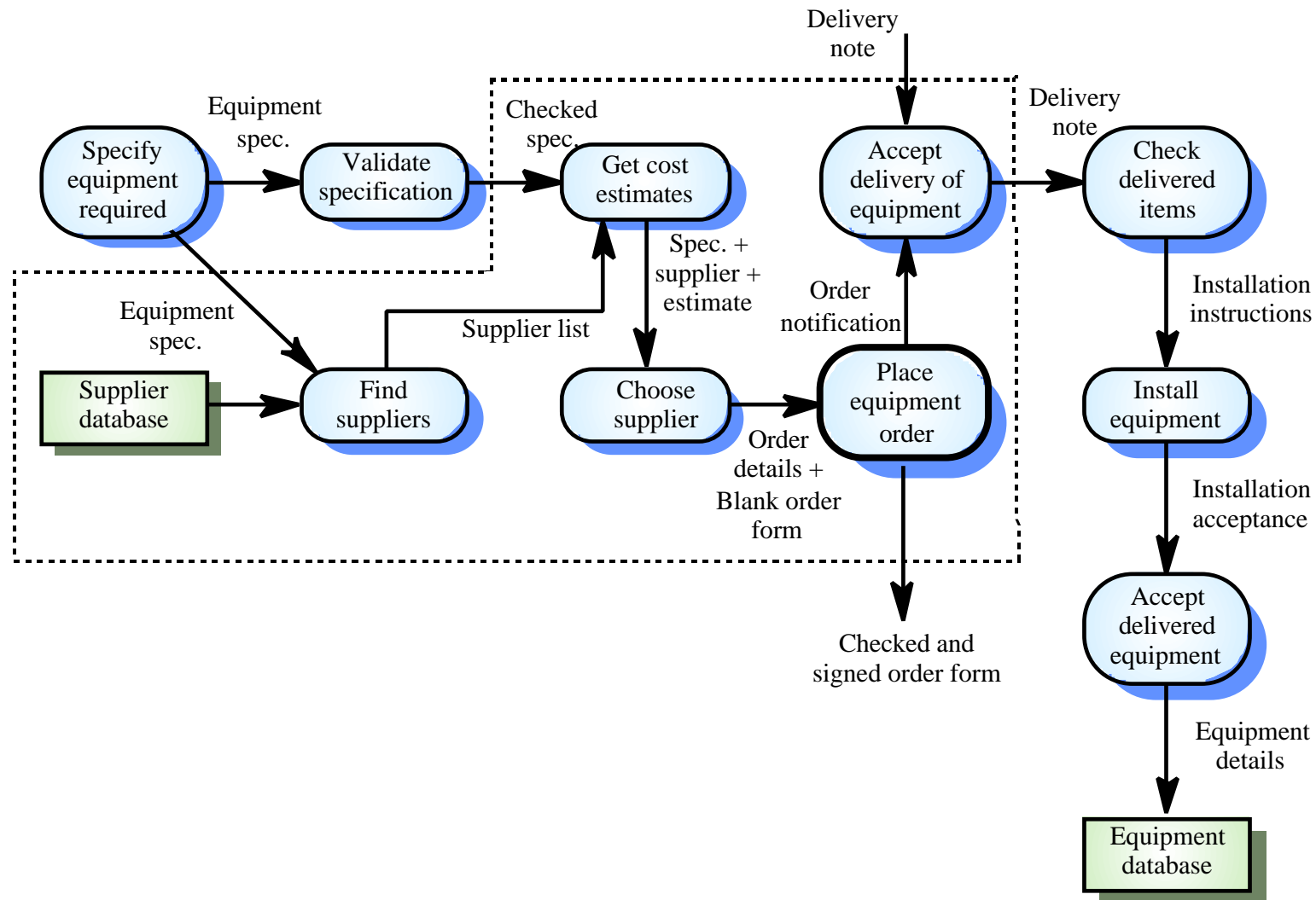
The context of an ATM system



Process models

- Process models show the overall process and the processes that are supported by the system
- Data flow models may be used to show the processes and the flow of information from one process to another

Equipment procurement process



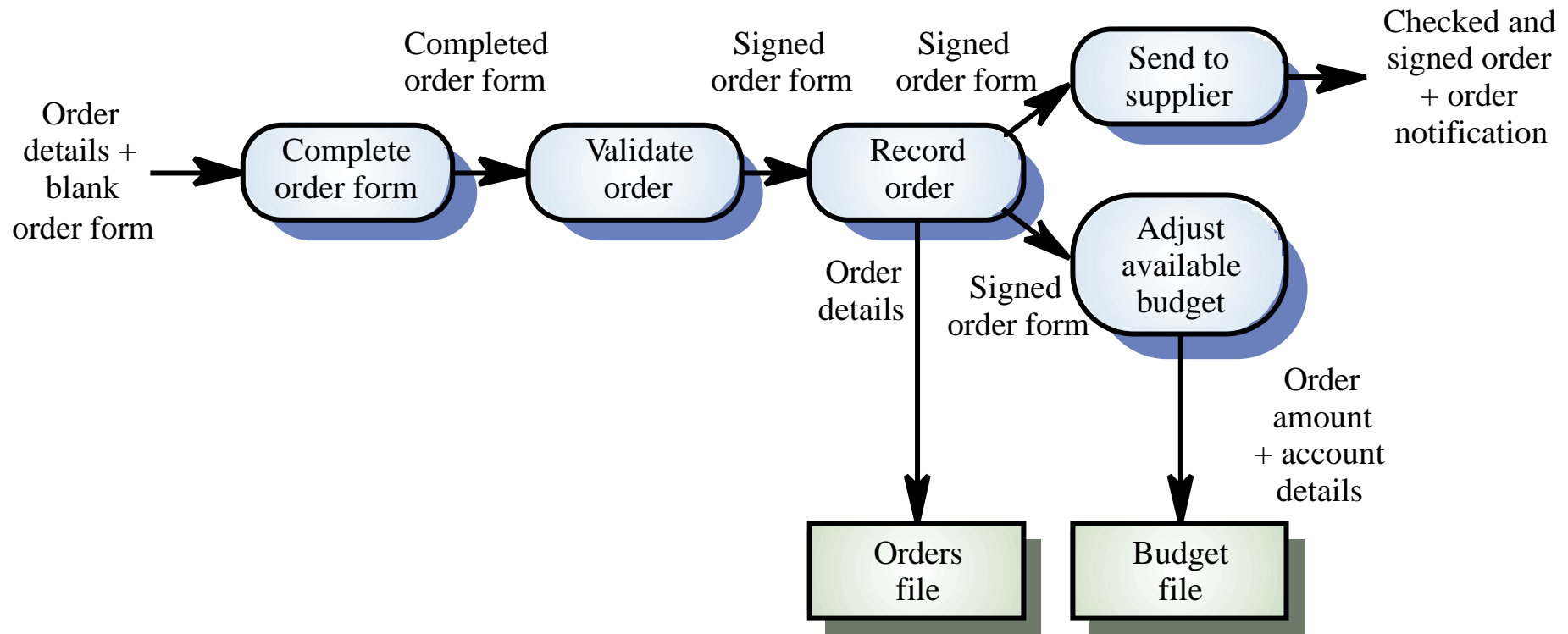
Behavioural models

- Behavioural models are used to describe the overall behaviour of a system
- Two types of behavioural model are shown here
 - 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
- Both of these models are required for a description of the system's behaviour

Data-processing models

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

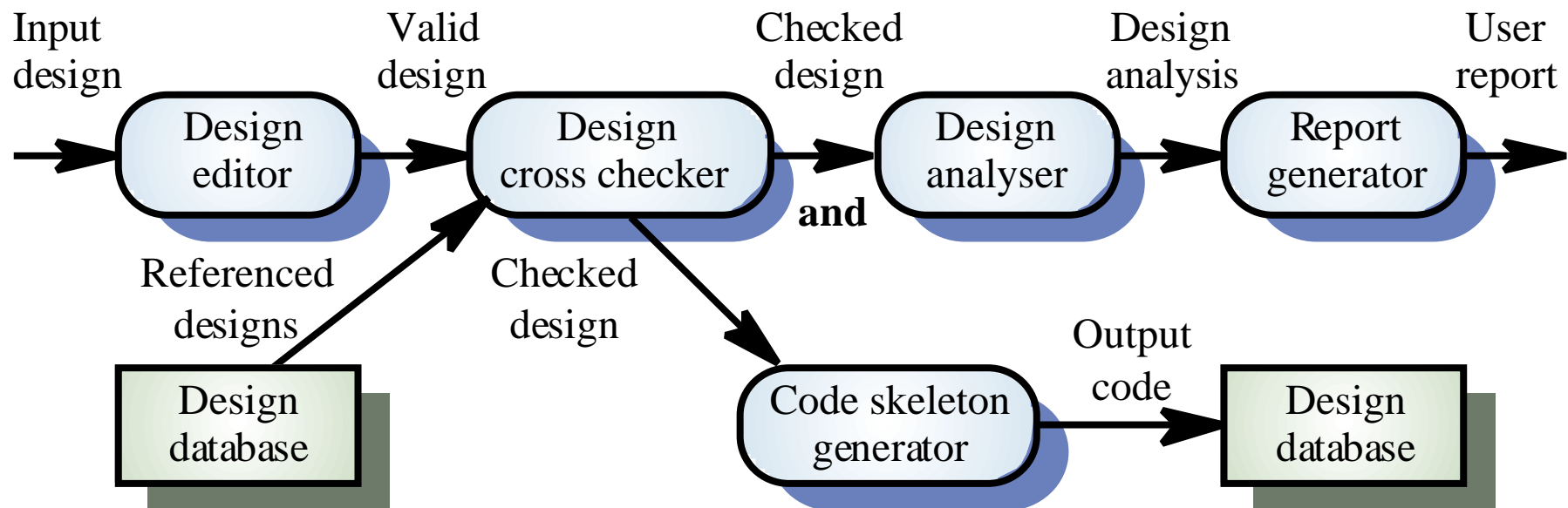
Order processing DFD



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

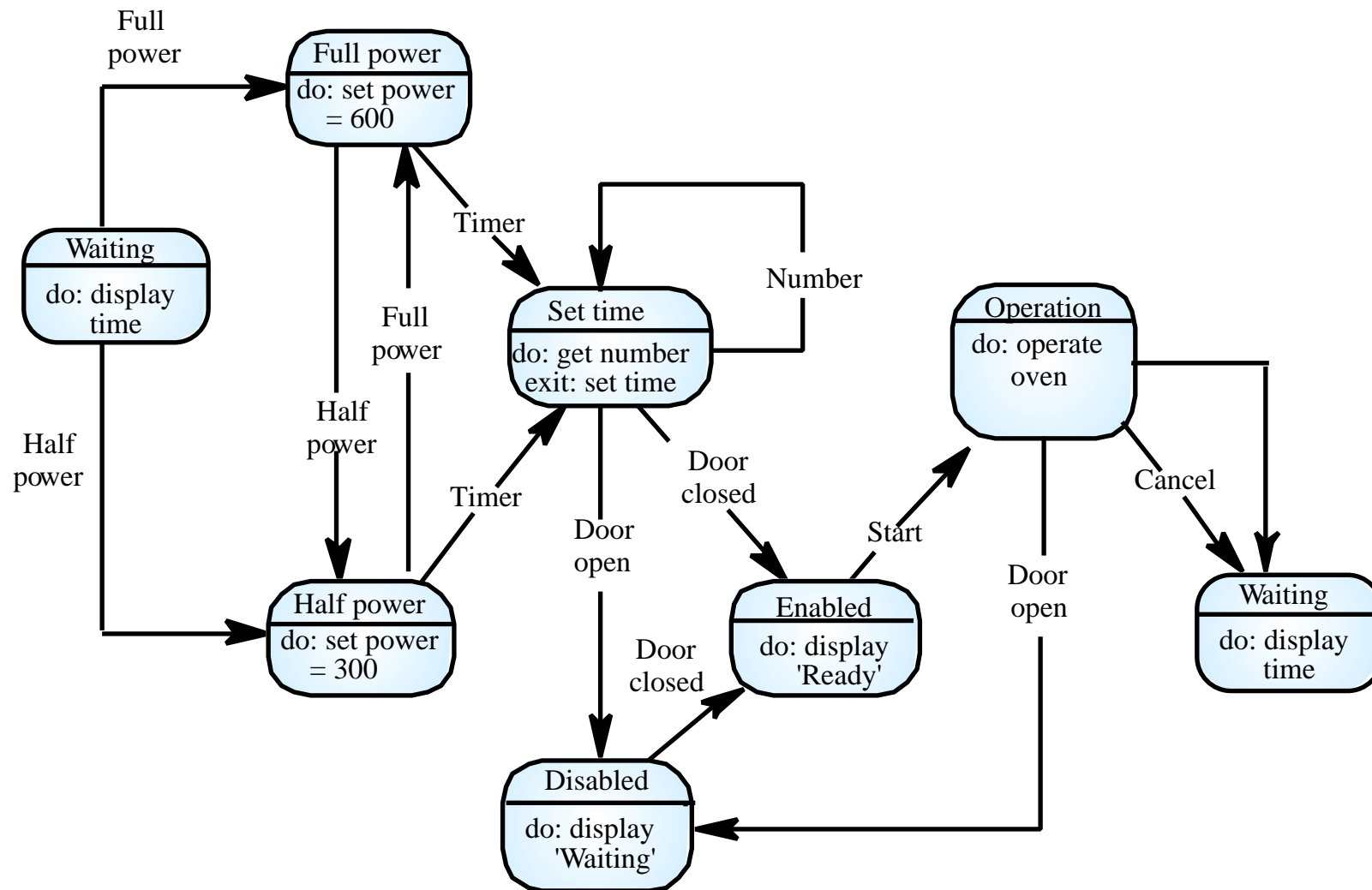
CASE toolset DFD



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

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.

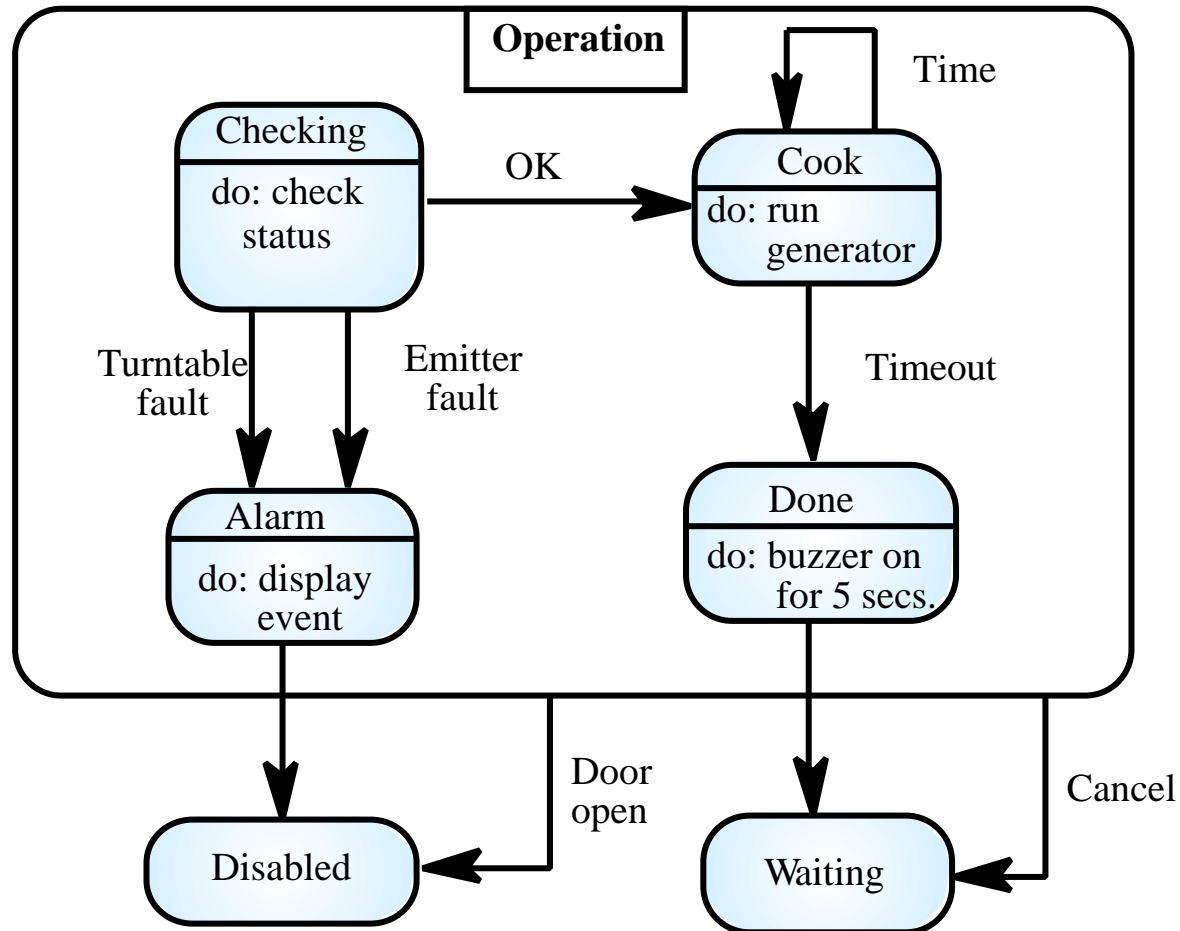
Microwave oven stimuli

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

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

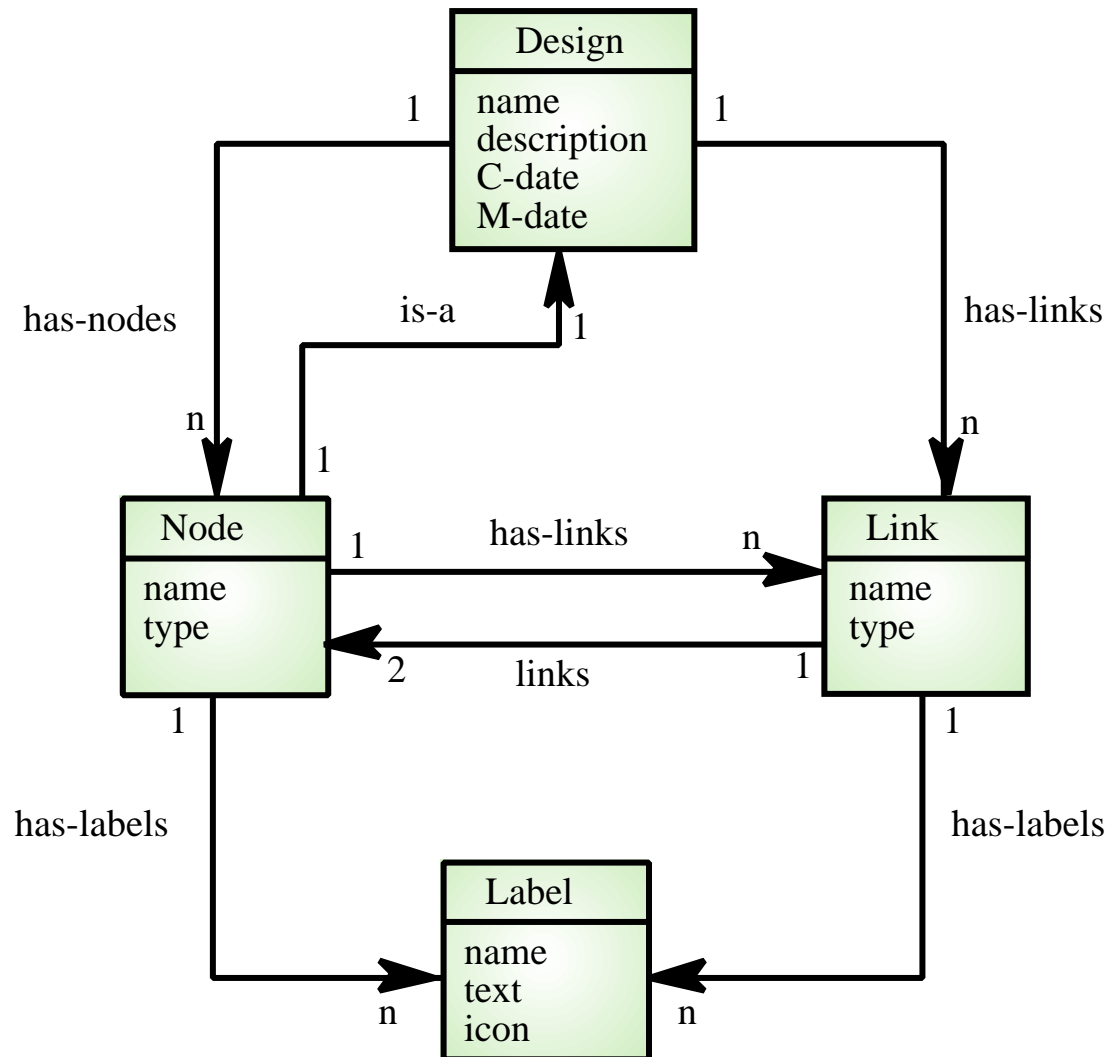
Microwave oven operation



Semantic data models

- Used to describe the logical structure of data processed by the system
- 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

Software design 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
has-labels	1:N relation between entities of type Node or Link and entities of type Label.	Relation	5.10.1998
Label	Holds structured or unstructured information about nodes or links. Labels are represented by an icon (which can be a transparent box) and associated text.	Entity	8.12.1998
Link	A 1:1 relation between design entities represented as nodes. Links are typed and may be named.	Relation	8.12.1998
name (label)	Each label has a name which identifies the type of label. The name must be unique within the set of label types used in a design.	Attribute	8.12.1998
name (node)	Each node has a name which must be unique within a design. The name may be up to 64 characters long.	Attribute	15.11.1998

Object models

- Object models describe the system in terms of object classes
- 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

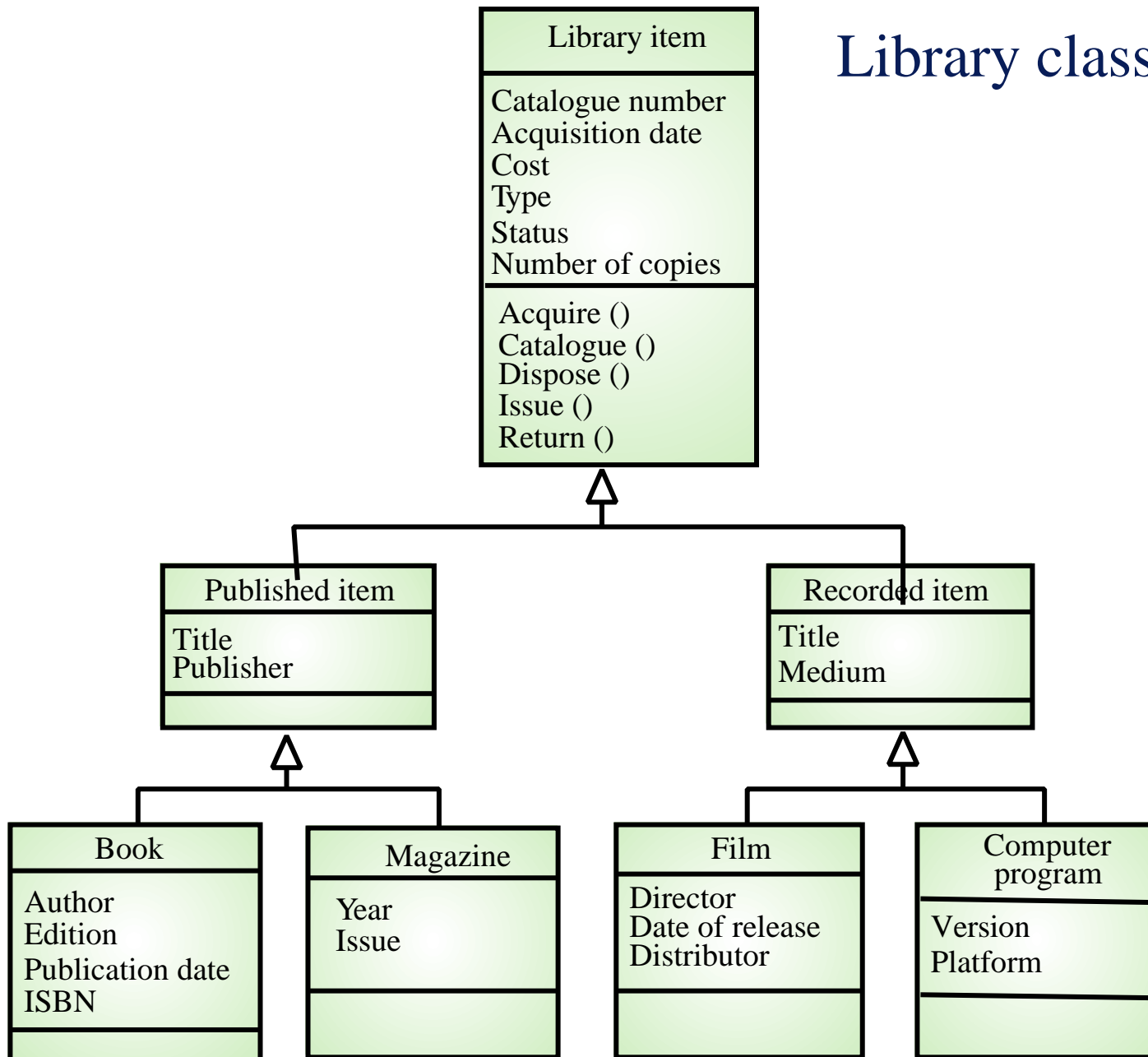
Inheritance models

- 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 is a difficult process if duplication in different branches is to be avoided

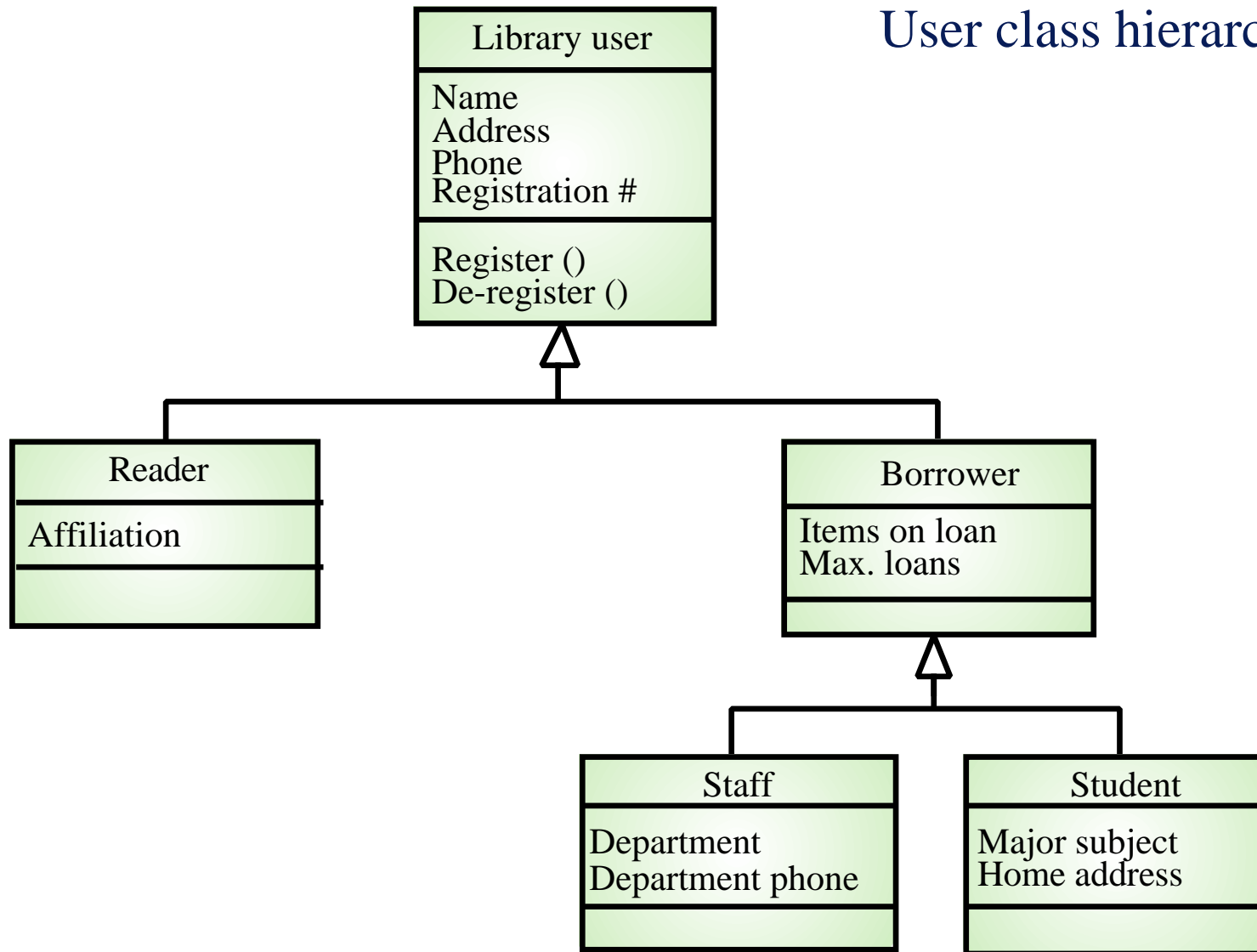
The Unified Modeling Language

- Devised by the developers of widely used object-oriented analysis and design methods
- 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



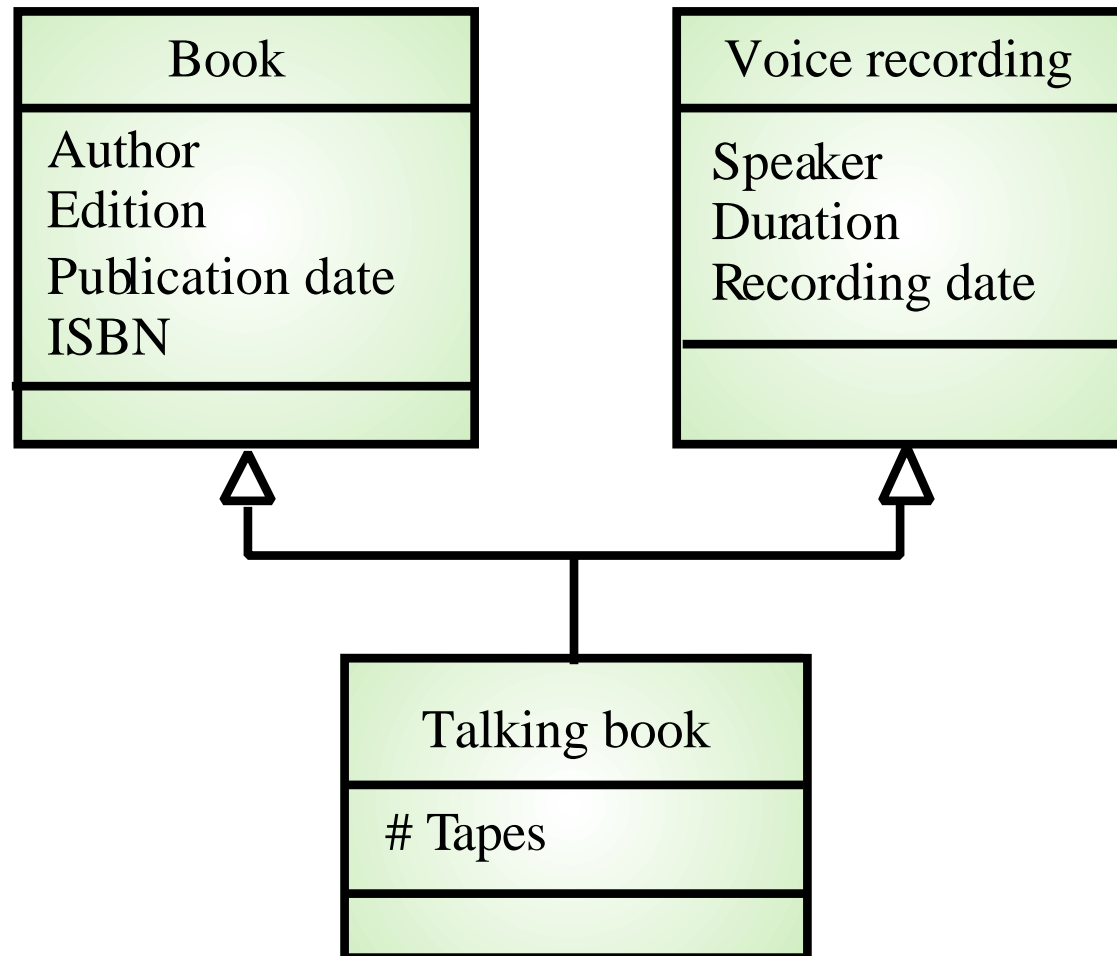
User class hierarchy



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
- Can lead to semantic conflicts where attributes/services with the same name in different super-classes have different semantics
- Makes class hierarchy reorganisation more complex

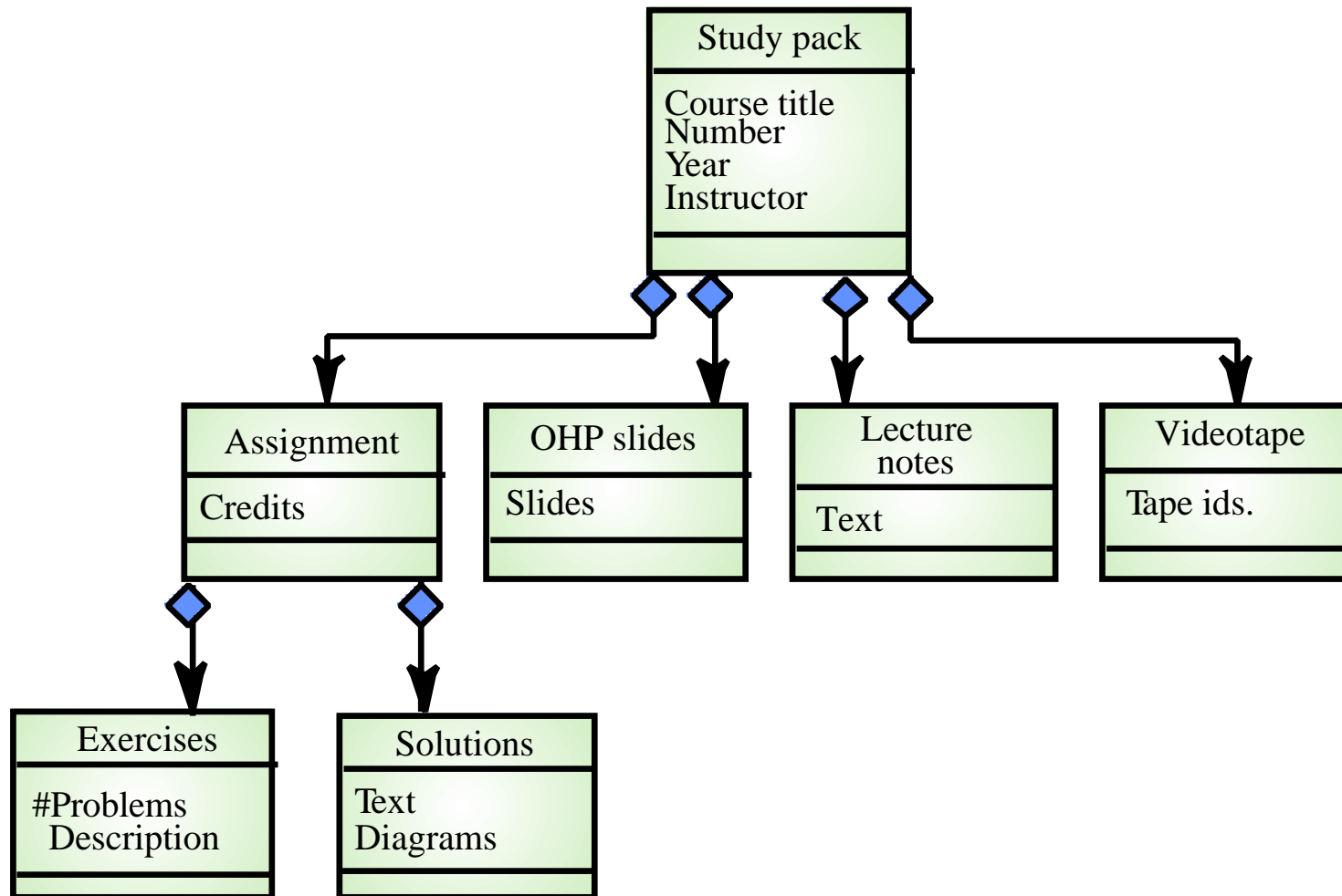
Multiple inheritance



Object aggregation

- Aggregation model shows how classes which are collections are composed of other classes
- Similar to the part-of relationship in semantic data models

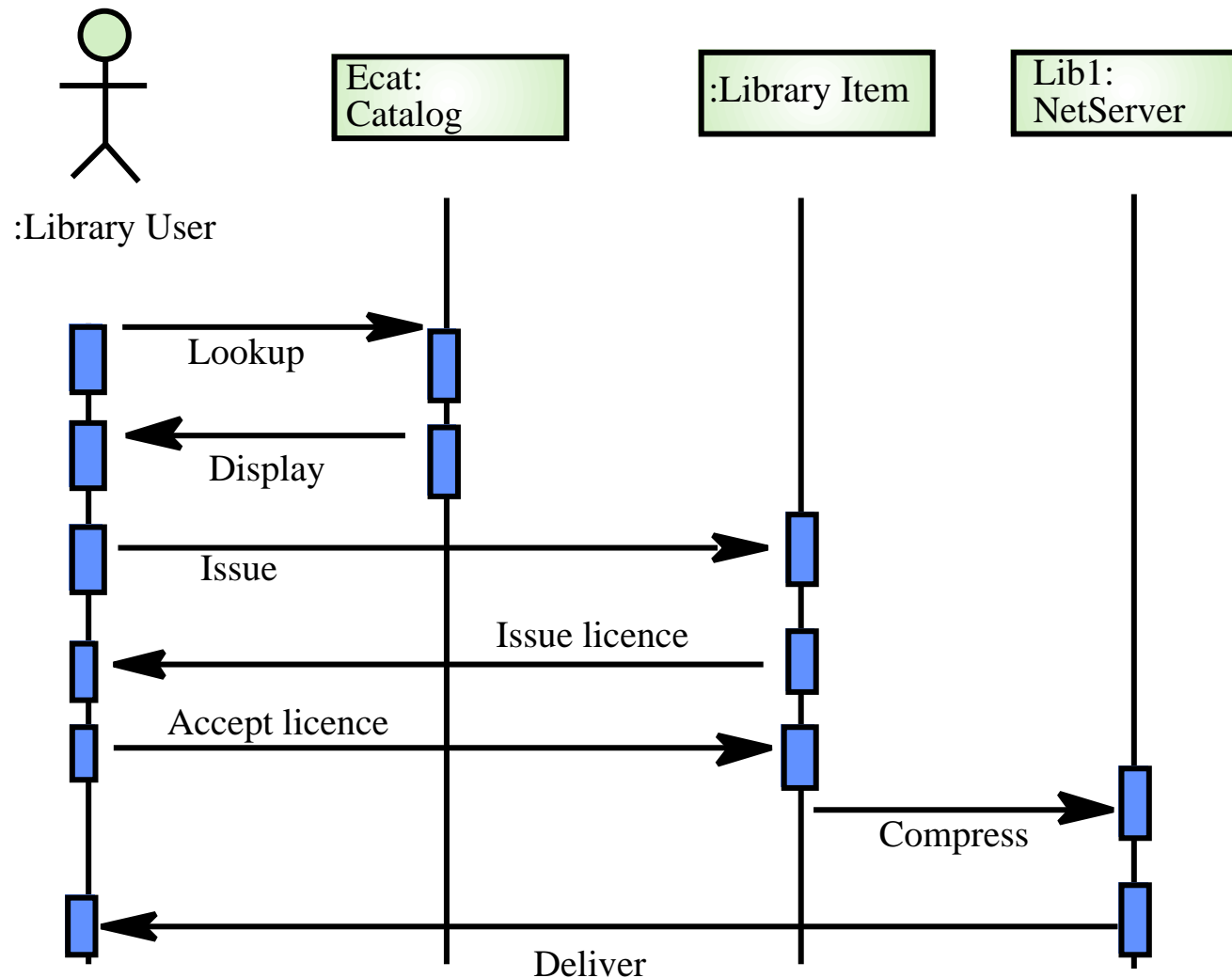
Object aggregation



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

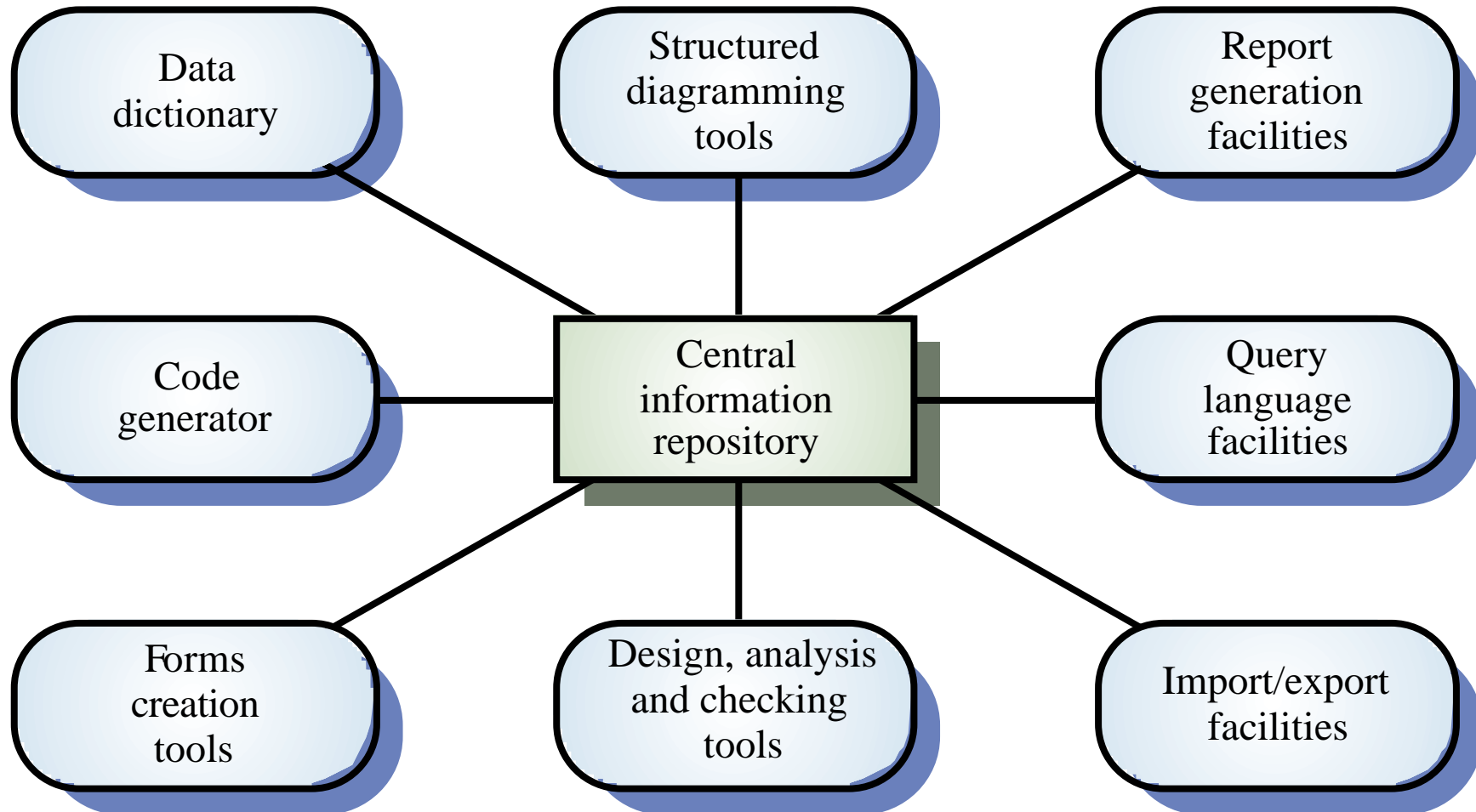
Issue of electronic items



CASE workbenches

- A coherent set of tools that is designed to support related software process activities such as analysis, design or testing
- Analysis and design workbenches support system modelling during both requirements engineering and system design
- These workbenches may support a specific design method or may provide support for a creating several different types of system model

An analysis and design workbench



Analysis workbench components

- Diagram editors
- Model analysis and checking tools
- Repository and associated query language
- Data dictionary
- Report definition and generation tools
- Forms definition tools
- Import/export translators
- Code generation tools

Key points

- A model is an abstract system view.
Complementary types of model provide different system information
- Context models show the position of a system in its environment with other systems and processes
- Data flow models may be used to model the data processing in a system
- State machine models model the system's behaviour in response to internal or external events

Key points

- Semantic data models describe the logical structure of data which is imported to or exported by the systems
- Object models describe logical system entities, their classification and aggregation
- Object models describe the logical system entities and their classification and aggregation
- CASE workbenches support the development of system models

Software Prototyping

- Rapid software development to validate requirements

Objectives

- To describe the use of prototypes in different types of development project
- To discuss evolutionary and throw-away prototyping
- To introduce three rapid prototyping techniques - high-level language development, database programming and component reuse
- To explain the need for user interface prototyping

Topics covered

- Prototyping in the software process
- Prototyping techniques
- User interface prototyping

System prototyping

- Prototyping is the rapid development of a system
- In the past, the developed system was normally thought of as inferior in some way to the required system so further development was required
- Now, the boundary between prototyping and normal system development is blurred and many systems are developed using an evolutionary approach

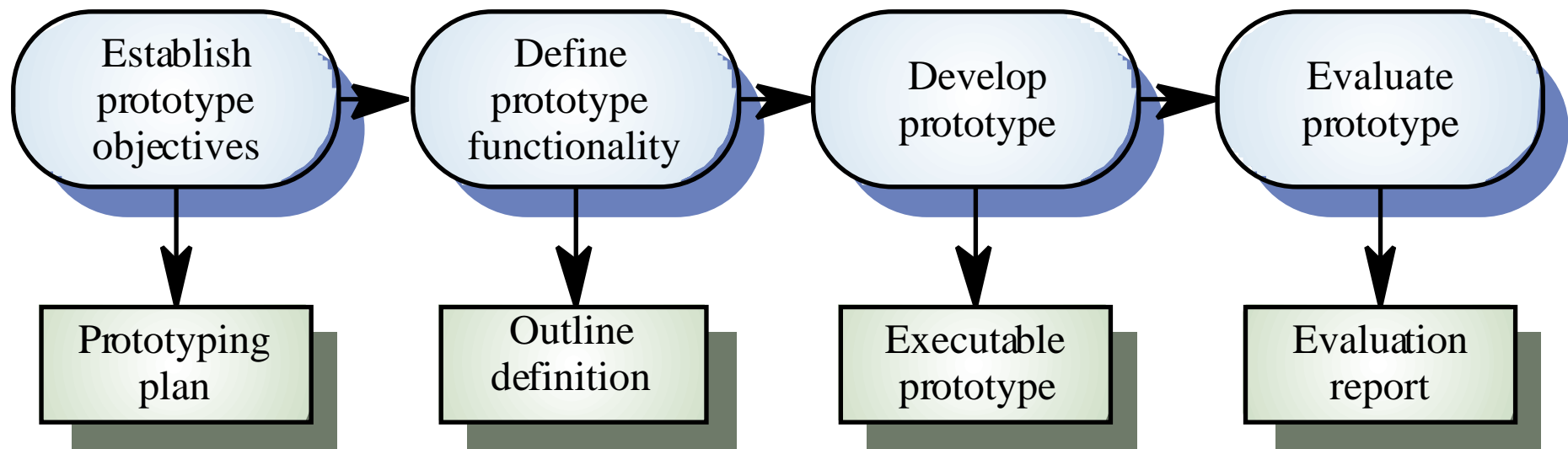
Uses of system prototypes

- The principal use is to help customers and developers understand the requirements for the system
 - Requirements elicitation. Users can experiment with a prototype to see how the system supports their work
 - Requirements validation. The prototype can reveal errors and omissions in the requirements
- Prototyping can be considered as a risk reduction activity which reduces requirements risks

Prototyping benefits

- Misunderstandings between software users and developers are exposed
- Missing services may be detected and confusing services may be identified
- A working system is available early in the process
- The prototype may serve as a basis for deriving a system specification
- The system can support user training and system testing

Prototyping process



Prototyping benefits

- Improved system usability
- Closer match to the system needed
- Improved design quality
- Improved maintainability
- Reduced overall development effort

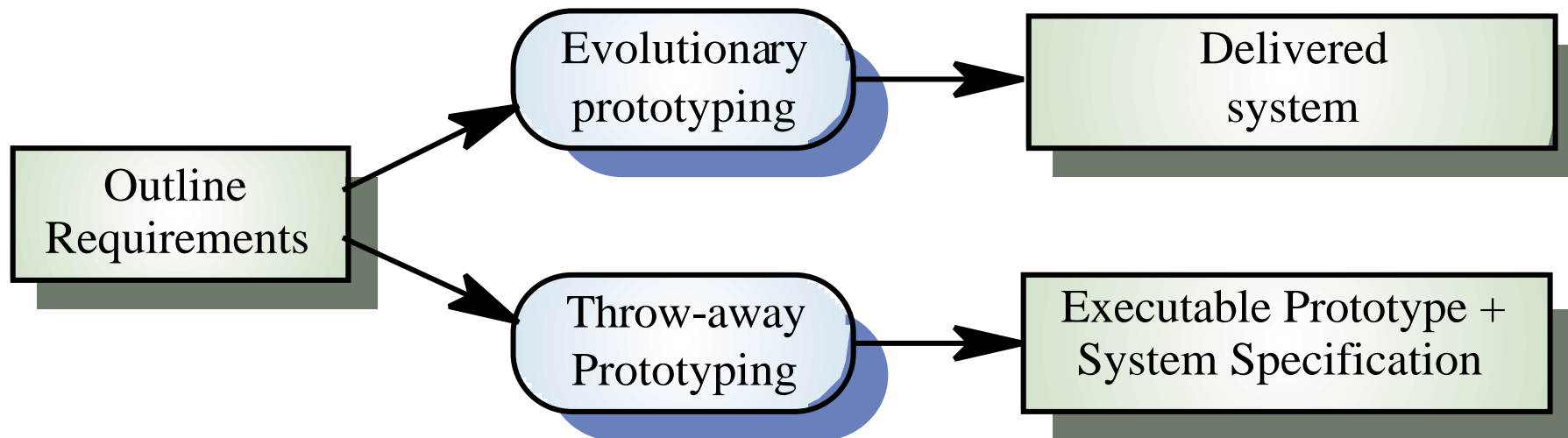
Prototyping in the software process

- Evolutionary prototyping
 - An approach to system development where an initial prototype is produced and refined through a number of stages to the final system
- Throw-away prototyping
 - A prototype which is usually a practical implementation of the system is produced to help discover requirements problems and then discarded. The system is then developed using some other development process

Prototyping objectives

- The objective of *evolutionary prototyping* is to deliver a working system to end-users. The development starts with those requirements which are best understood.
- The objective of *throw-away prototyping* is to validate or derive the system requirements. The prototyping process starts with those requirements which are poorly understood

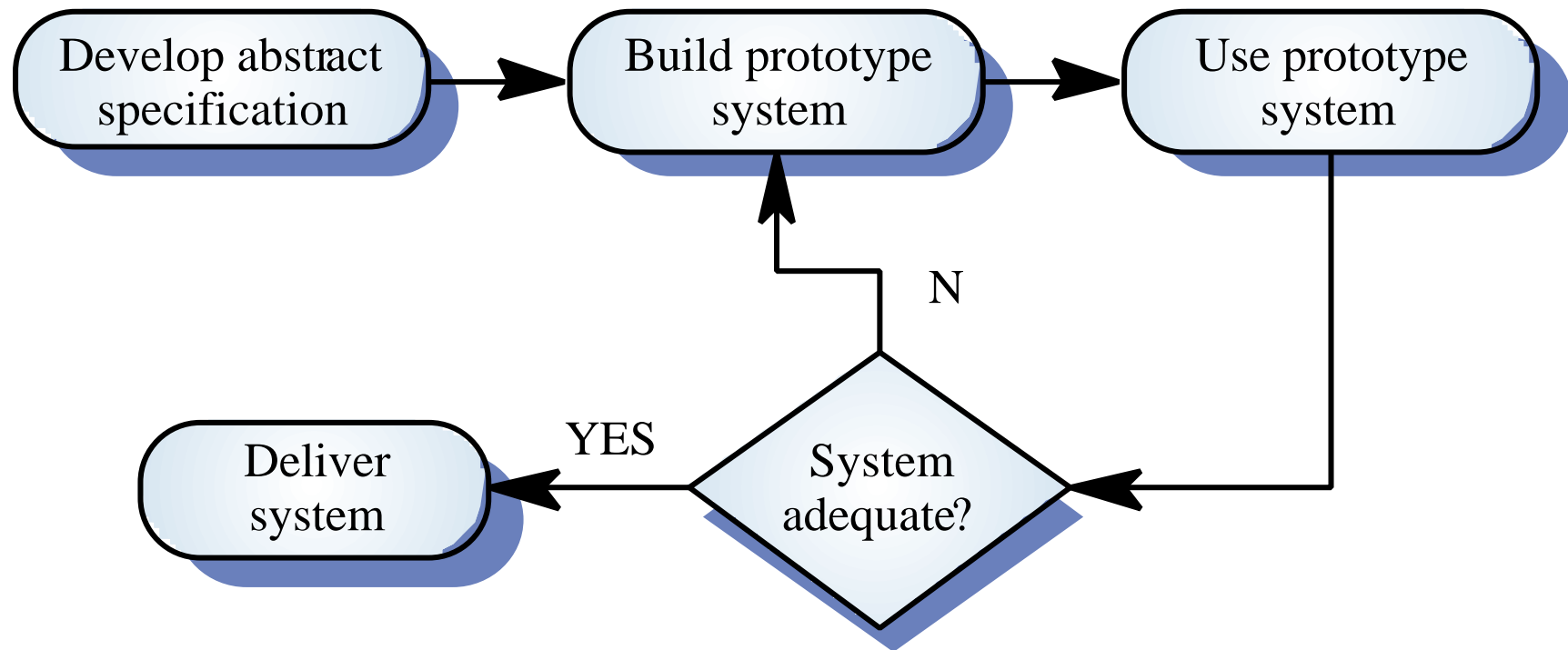
Approaches to prototyping



Evolutionary prototyping

- Must be used for systems where the specification cannot be developed in advance e.g. AI systems and user interface systems
- Based on techniques which allow rapid system iterations
- Verification is impossible as there is no specification. Validation means demonstrating the adequacy of the system

Evolutionary prototyping



Evolutionary prototyping advantages

- Accelerated delivery of the system
 - Rapid delivery and deployment are sometimes more important than functionality or long-term software maintainability
- User engagement with the system
 - Not only is the system more likely to meet user requirements, they are more likely to commit to the use of the system

Evolutionary prototyping

- Specification, design and implementation are inter-twined
- The system is developed as a series of increments that are delivered to the customer
- Techniques for rapid system development are used such as CASE tools and 4GLs
- User interfaces are usually developed using a GUI development toolkit

Evolutionary prototyping problems

- Management problems
 - Existing management processes assume a waterfall model of development
 - Specialist skills are required which may not be available in all development teams
- Maintenance problems
 - Continual change tends to corrupt system structure so long-term maintenance is expensive
- Contractual problems

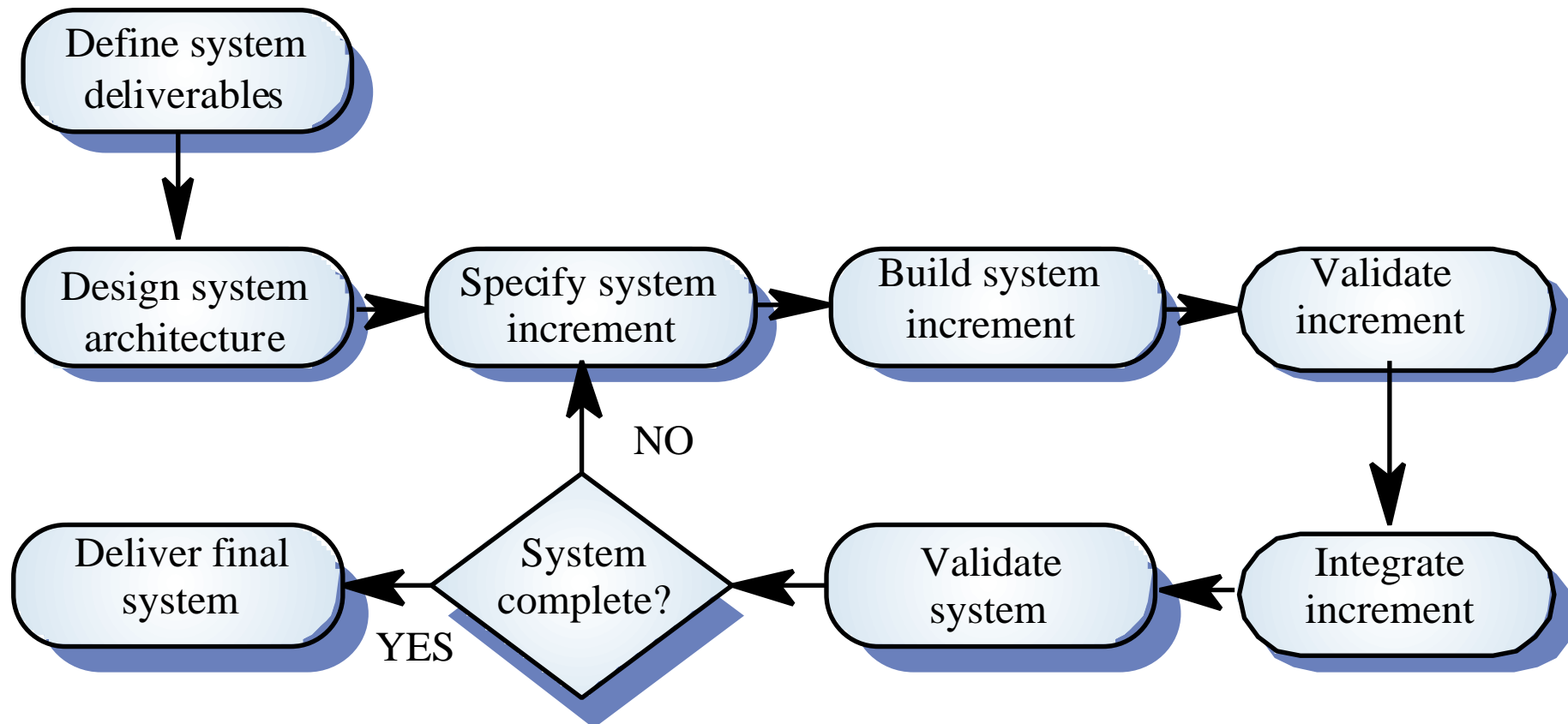
Prototypes as specifications

- Some parts of the requirements (e.g. safety-critical functions) may be impossible to prototype and so don't appear in the specification
- An implementation has no legal standing as a contract
- Non-functional requirements cannot be adequately tested in a system prototype

Incremental development

- System is developed and delivered in increments after establishing an overall architecture
- Requirements and specifications for each increment may be developed
- Users may experiment with delivered increments while others are being developed. therefore, these serve as a form of prototype system
- Intended to combine some of the advantages of prototyping but with a more manageable process and better system structure

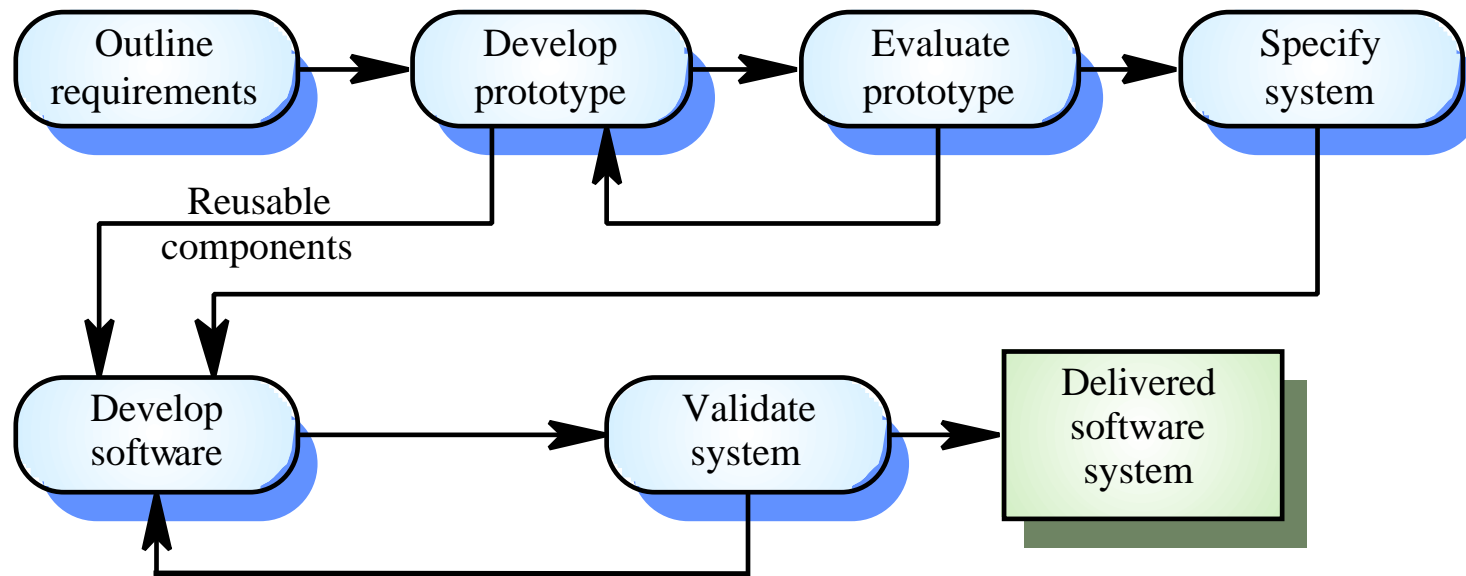
Incremental development process



Throw-away prototyping

- Used to reduce requirements risk
- The prototype is developed from an initial specification, delivered for experiment then discarded
- The throw-away prototype should NOT be considered as a final system
 - Some system characteristics may have been left out
 - There is no specification for long-term maintenance
 - The system will be poorly structured and difficult to maintain

Throw-away prototyping



Prototype delivery

- Developers may be pressurised to deliver a throw-away prototype as a final system
- This is not recommended
 - It may be impossible to tune the prototype to meet non-functional requirements
 - The prototype is inevitably undocumented
 - The system structure will be degraded through changes made during development
 - Normal organisational quality standards may not have been applied

Rapid prototyping techniques

- Various techniques may be used for rapid development
 - Dynamic high-level language development
 - Database programming
 - Component and application assembly
- These are not exclusive techniques - they are often used together
- Visual programming is an inherent part of most prototype development systems

Dynamic high-level languages

- Languages which include powerful data management facilities
- Need a large run-time support system. Not normally used for large system development
- Some languages offer excellent UI development facilities
- Some languages have an integrated support environment whose facilities may be used in the prototype

Prototyping languages

Language	Type	Application domain
Smalltalk	Object-oriented	Interactive systems
Java	Object-oriented	Interactive systems
Prolog	Logic	Symbolic processing
Lisp	List-based	Symbolic processing

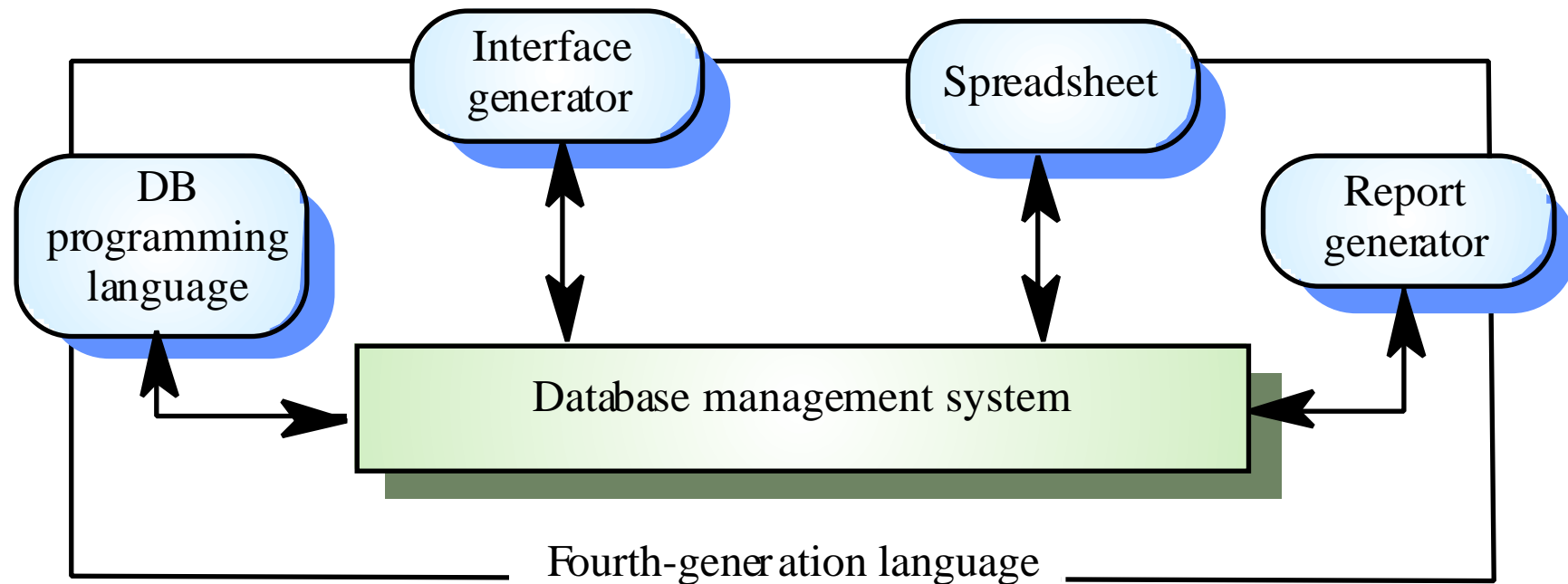
Choice of prototyping language

- What is the application domain of the problem?
- What user interaction is required?
- What support environment comes with the language?
- Different parts of the system may be programmed in different languages. However, there may be problems with language communications

Database programming languages

- Domain specific languages for business systems based around a database management system
- Normally include a database query language, a screen generator, a report generator and a spreadsheet.
- May be integrated with a CASE toolset
- The language + environment is sometimes known as a fourth-generation language (4GL)
- Cost-effective for small to medium sized business systems

Database programming



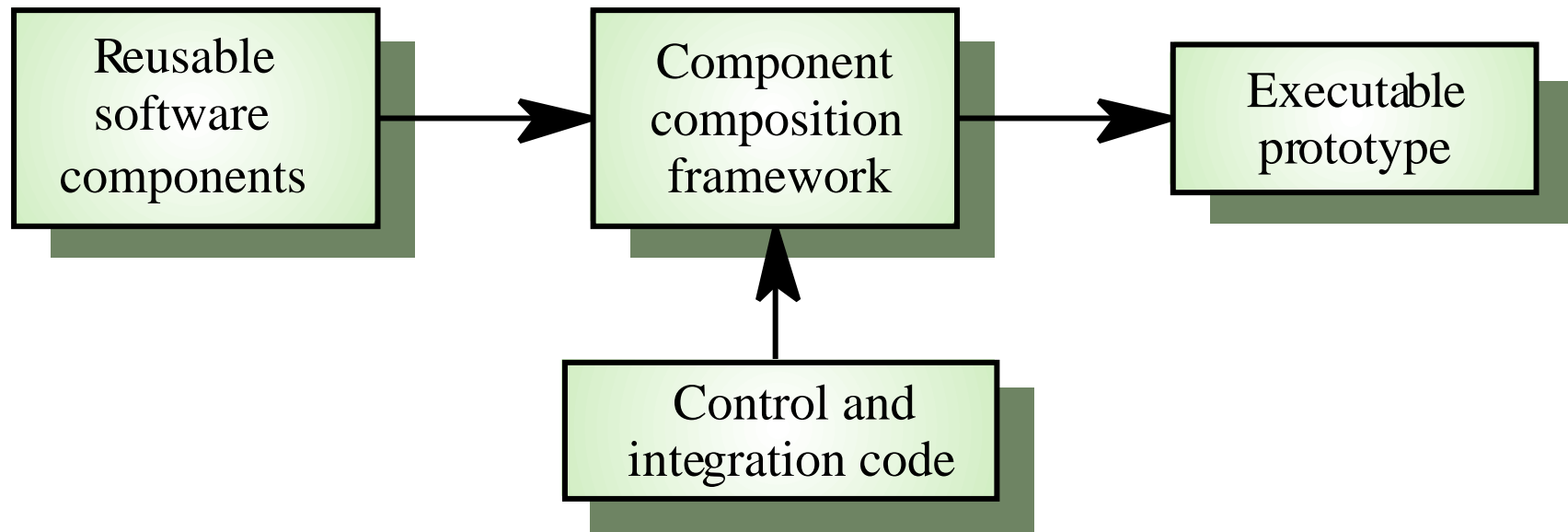
Component and application assembly

- Prototypes can be created quickly from a set of reusable components plus some mechanism to ‘glue’ these component together
- The composition mechanism must include control facilities and a mechanism for component communication
- The system specification must take into account the availability and functionality of existing components

Prototyping with reuse

- Application level development
 - Entire application systems are integrated with the prototype so that their functionality can be shared
 - For example, if text preparation is required, a standard word processor can be used
- Component level development
 - Individual components are integrated within a standard framework to implement the system
 - Framework can be a scripting language or an integration framework such as CORBA

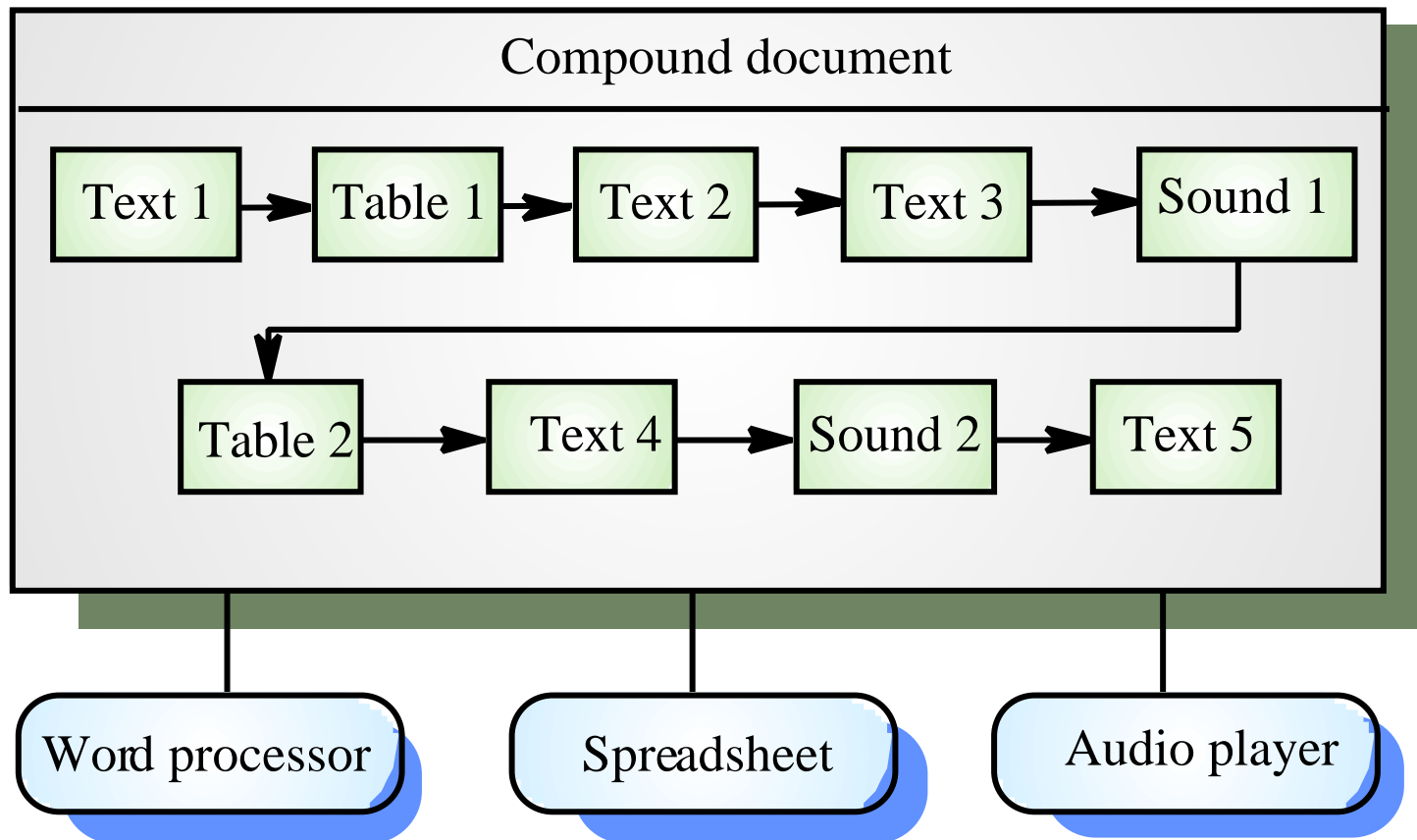
Reusable component composition



Compound documents

- For some applications, a prototype can be created by developing a compound document
- This is a document with active elements (such as a spreadsheet) that allow user computations
- Each active element has an associated application which is invoked when that element is selected
- The document itself is the integrator for the different applications

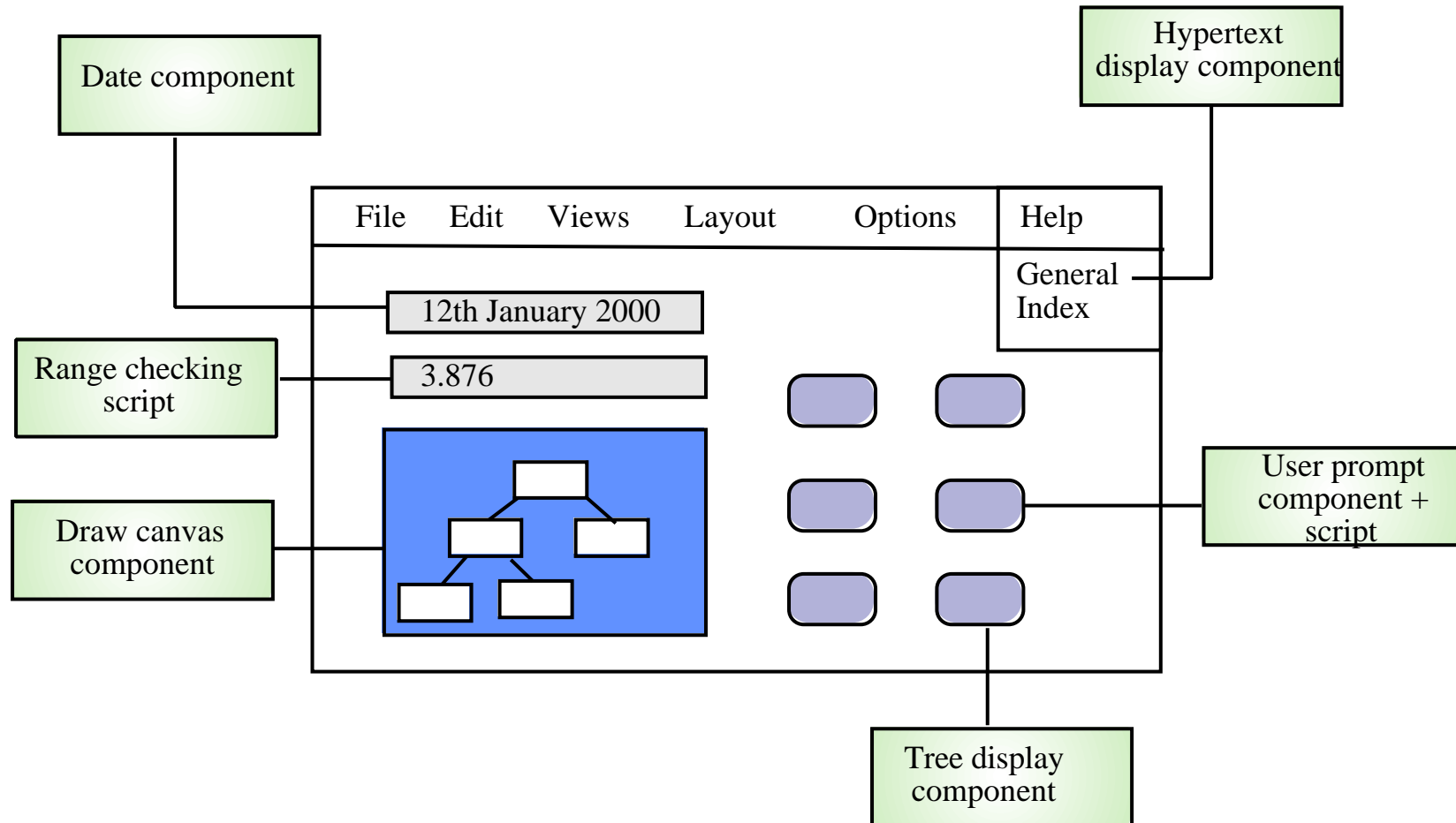
Application linking in compound documents



Visual programming

- Scripting languages such as Visual Basic support visual programming where the prototype is developed by creating a user interface from standard items and associating components with these items
- A large library of components exists to support this type of development
- These may be tailored to suit the specific application requirements

Visual programming with reuse



Problems with visual development

- Difficult to coordinate team-based development
- No explicit system architecture
- Complex dependencies between parts of the program can cause maintainability problems

User interface prototyping

- It is impossible to pre-specify the look and feel of a user interface in an effective way. prototyping is essential
- UI development consumes an increasing part of overall system development costs
- User interface generators may be used to ‘draw’ the interface and simulate its functionality with components associated with interface entities
- Web interfaces may be prototyped using a web site editor

Key points

- A prototype can be used to give end-users a concrete impression of the system's capabilities
- Prototyping is becoming increasingly used for system development where rapid development is essential
- Throw-away prototyping is used to understand the system requirements
- In evolutionary prototyping, the system is developed by evolving an initial version to the final version

Key points

- Rapid development of prototypes is essential. This may require leaving out functionality or relaxing non-functional constraints
- Prototyping techniques include the use of very high-level languages, database programming and prototype construction from reusable components
- Prototyping is essential for parts of the system such as the user interface which cannot be effectively pre-specified. Users must be involved in prototype evaluation

Formal Specification

- Techniques for the unambiguous specification of software

Objectives

- To explain why formal specification techniques help discover problems in system requirements
- To describe the use of algebraic techniques for interface specification
- To describe the use of model-based techniques for behavioural specification

Topics covered

- Formal specification in the software process
- Interface specification
- Behavioural specification

Formal methods

- Formal specification is part of a more general collection of techniques that are known as ‘formal methods’
- These are all based on mathematical representation and analysis of software
- Formal methods include
 - Formal specification
 - Specification analysis and proof
 - Transformational development
 - Program verification

Acceptance of formal methods

- Formal methods have not become mainstream software development techniques as was once predicted
 - Other software engineering techniques have been successful at increasing system quality. Hence the need for formal methods has been reduced
 - Market changes have made time-to-market rather than software with a low error count the key factor. Formal methods do not reduce time to market
 - The scope of formal methods is limited. They are not well-suited to specifying and analysing user interfaces and user interaction
 - Formal methods are hard to scale up to large systems

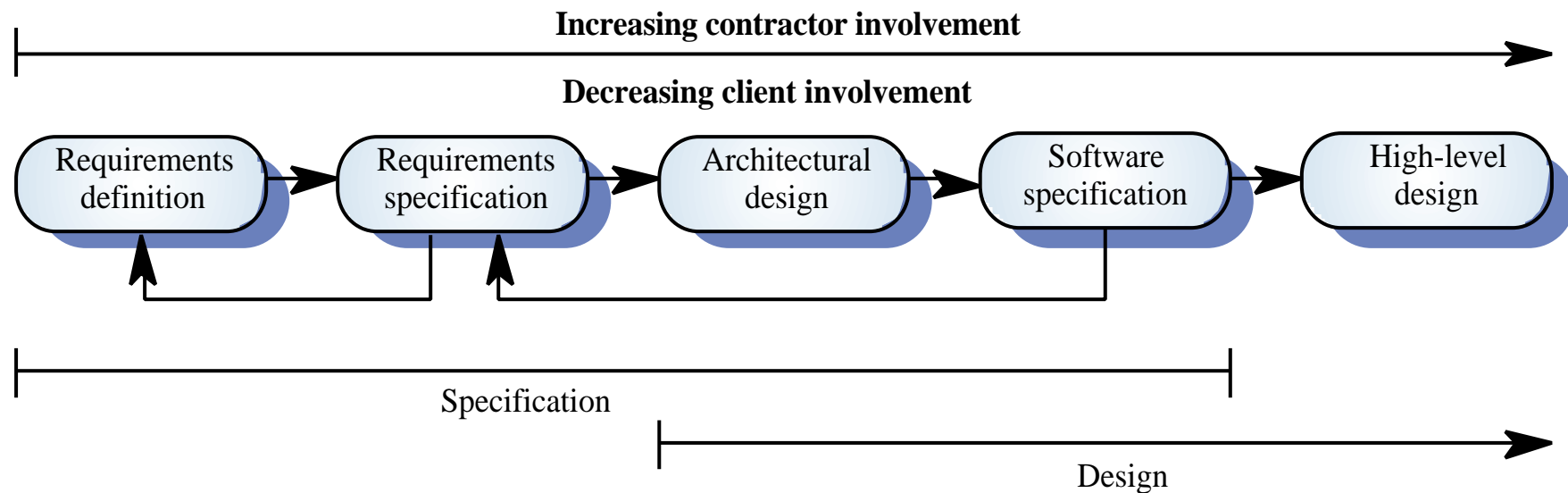
Use of formal methods

- Formal methods have limited practical applicability
- Their principal benefits are in reducing the number of errors in systems so their main area of applicability is critical systems
- In this area, the use of formal methods is most likely to be cost-effective

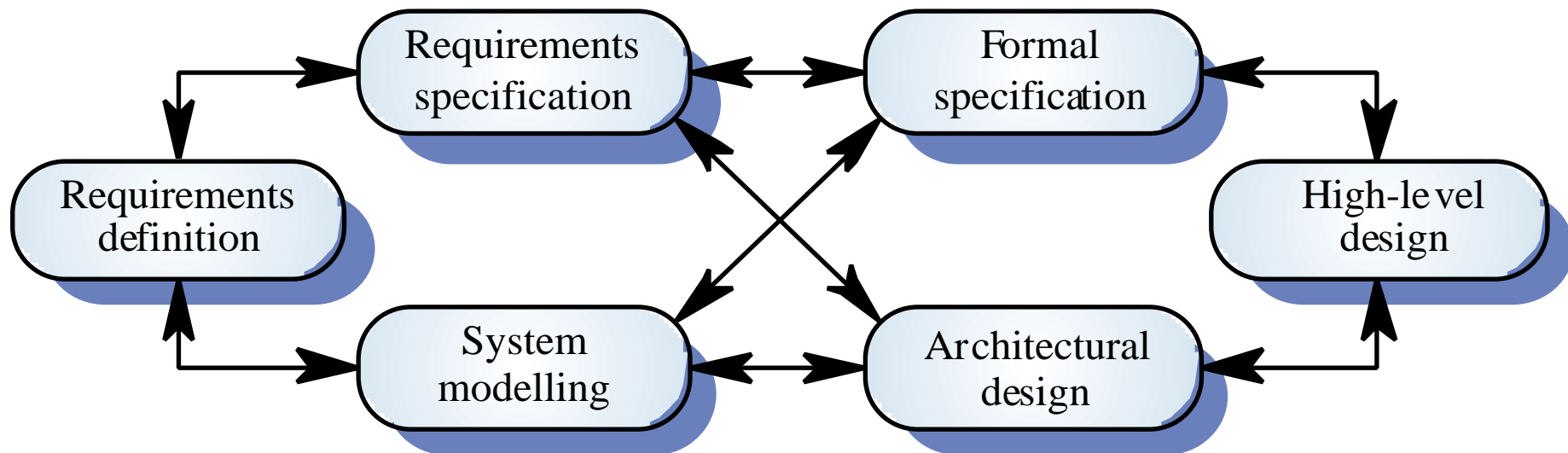
Specification in the software process

- Specification and design are inextricably intermingled.
- Architectural design is essential to structure a specification.
- Formal specifications are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics.

Specification and design



Specification in the software process



Specification techniques

- Algebraic approach
 - The system is specified in terms of its operations and their relationships
- Model-based approach
 - The system is specified in terms of a state model that is constructed using mathematical constructs such as sets and sequences. Operations are defined by modifications to the system's state

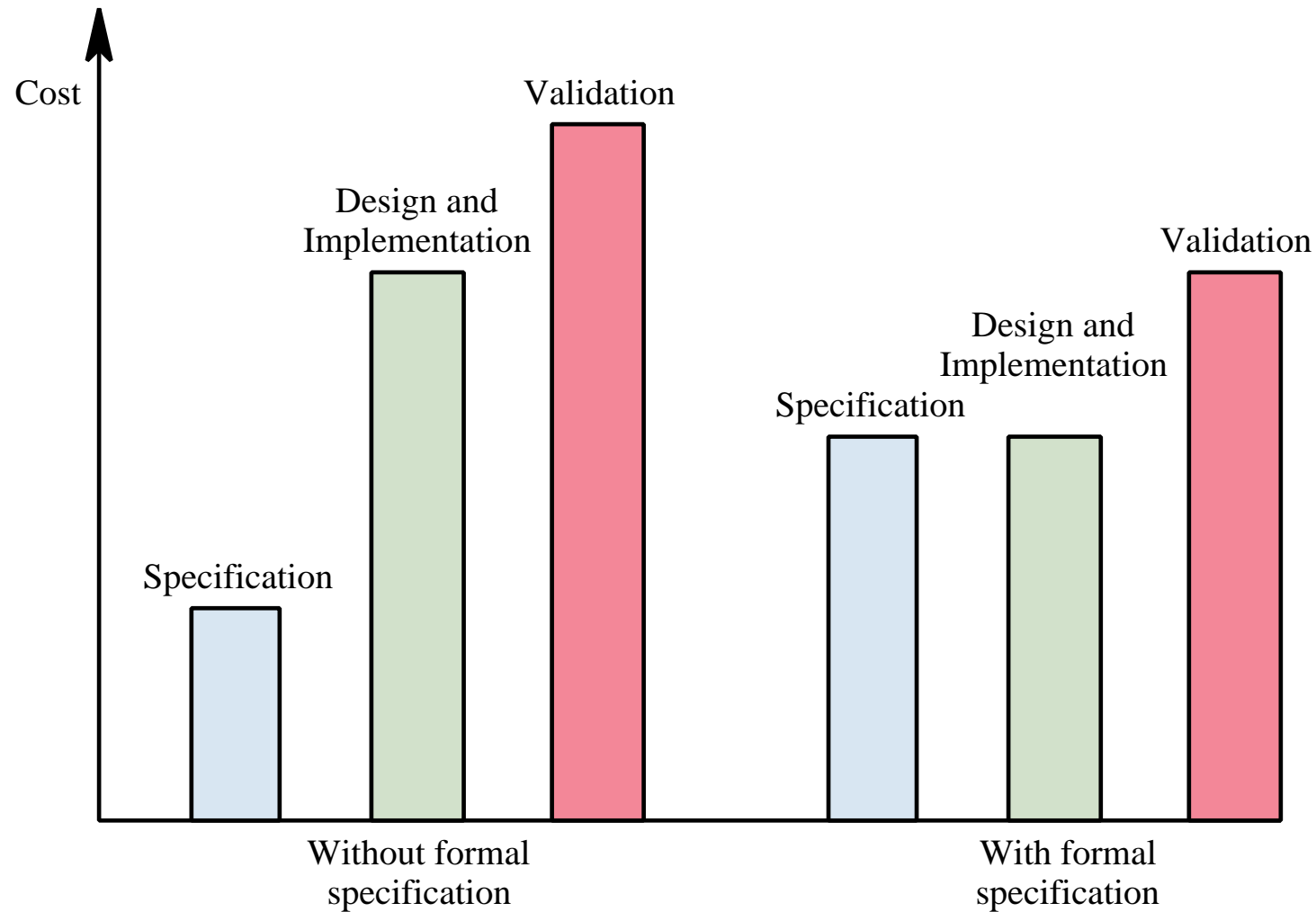
Formal specification languages

	Sequential	Concurrent
Algebraic	Larch (Guttag, Horning et al., 1985; Guttag, Horning et al., 1993), OBJ (Futatsugi, Goguen et al., 1985)	Lotos (Bolognesi and Brinksma, 1987),
Model-based	Z (Spivey, 1992) VDM (Jones, 1980) B (Wordsworth, 1996)	CSP (Hoare, 1985) Petri Nets (Peterson, 1981)

Use of formal specification

- Formal specification involves investing more effort in the early phases of software development
- This reduces requirements errors as it forces a detailed analysis of the requirements
- Incompleteness and inconsistencies can be discovered and resolved
- Hence, savings as made as the amount of rework due to requirements problems is reduced

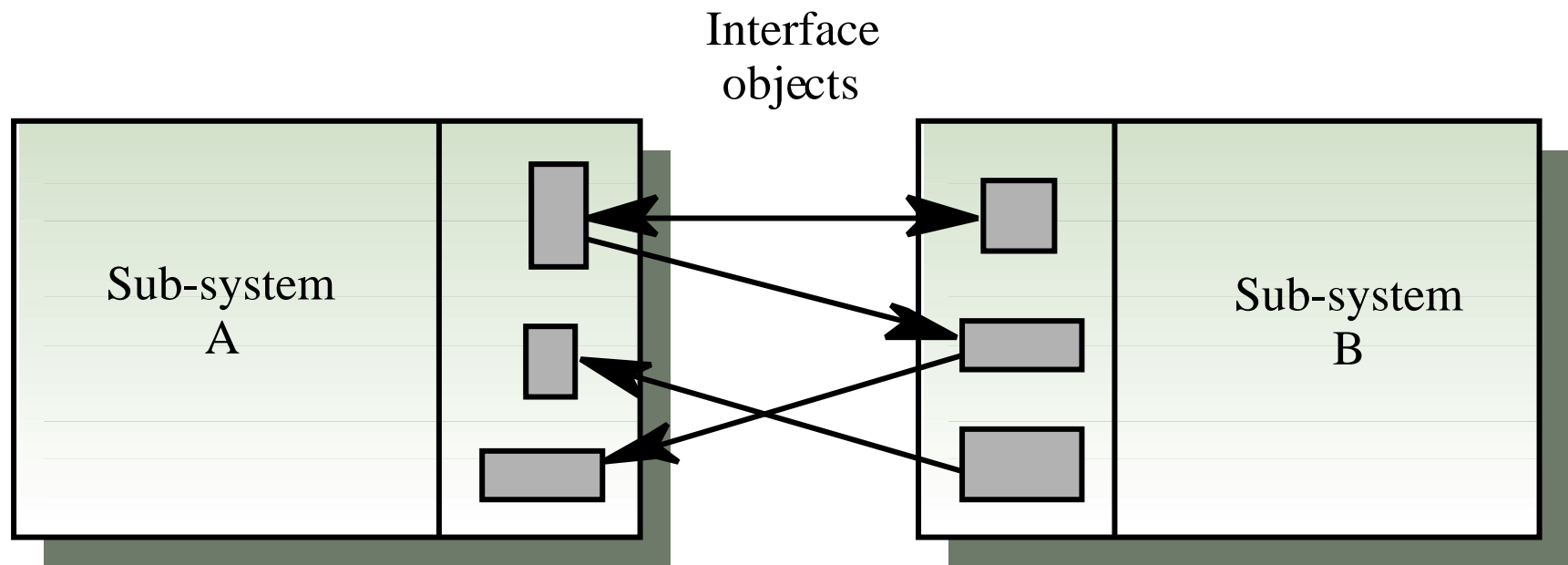
Development costs with formal specification



Interface specification

- Large systems are decomposed into subsystems with well-defined interfaces between these subsystems
- Specification of subsystem interfaces allows independent development of the different subsystems
- Interfaces may be defined as abstract data types or object classes
- The algebraic approach to formal specification is particularly well-suited to interface specification

Sub-system interfaces



The structure of an algebraic specification

< SPECIFICATION NAME > (Generic Parameter)

sort < name >

imports < LIST OF SPECIFICATION NAMES >

Informal description of the sort and its operations

Operation signatures setting out the names and the types of the parameters to the operations defined over the sort

Axioms defining the operations over the sort

Specification components

- Introduction
 - Defines the sort (the type name) and declares other specifications that are used
- Description
 - Informally describes the operations on the type
- Signature
 - Defines the syntax of the operations in the interface and their parameters
- Axioms
 - Defines the operation semantics by defining axioms which characterise behaviour

Systematic algebraic specification

- Algebraic specifications of a system may be developed in a systematic way
 - Specification structuring.
 - Specification naming.
 - Operation selection.
 - Informal operation specification
 - Syntax definition
 - Axiom definition

Specification operations

- Constructor operations. Operations which create entities of the type being specified
- Inspection operations. Operations which evaluate entities of the type being specified
- To specify behaviour, define the inspector operations for each constructor operation

Operations on a list ADT

- Constructor operations which evaluate to sort List
 - Create, Cons and Tail
- Inspection operations which take sort list as a parameter and return some other sort
 - Head and Length.
- Tail can be defined using the simpler constructors Create and Cons. No need to define Head and Length with Tail.

List specification

LIST (Elem)

sort List
imports INTEGER

Defines a list where elements are added at the end and removed from the front. The operations are Create, which brings an empty list into existence, Cons, which creates a new list with an added member, Length, which evaluates the list size, Head, which evaluates the front element of the list, and Tail, which creates a list by removing the head from its input list. Undefined represents an undefined value of type Elem.

Create \rightarrow List
Cons (List, Elem) \rightarrow List
Head (List) \rightarrow Elem
Length (List) \rightarrow Integer
Tail (List) \rightarrow List

Head (Create) = Undefined **exception** (empty list)
Head (Cons (L, v)) = **if** L = Create **then** v **else** Head (L)
Length (Create) = 0
Length (Cons (L, v)) = Length (L) + 1
Tail (Create) = Create
Tail (Cons (L, v)) = **if** L = Create **then** Create **else** Cons (Tail (L), v)

Recursion in specifications

- Operations are often specified recursively
- $\text{Tail}(\text{Cons}(L, v)) = \text{if } L = \text{Create} \text{ then Create}$
 $\text{else Cons}(\text{Tail}(L), v)$
 - $\text{Cons}([5, 7], 9) = [5, 7, 9]$
 - $\text{Tail}([5, 7, 9]) = \text{Tail}(\text{Cons}([5, 7], 9)) =$
 - $\text{Cons}(\text{Tail}([5, 7]), 9) = \text{Cons}(\text{Tail}(\text{Cons}([5], 7)), 9) =$
 - $\text{Cons}(\text{Cons}(\text{Tail}([5]), 7), 9) =$
 - $\text{Cons}(\text{Cons}(\text{Tail}(\text{Cons}([], 5)), 7), 9) =$
 - $\text{Cons}(\text{Cons}([\text{Create}], 7), 9) = \text{Cons}([7], 9) = [7, 9]$

Interface specification in critical systems

- Consider an air traffic control system where aircraft fly through managed sectors of airspace
- Each sector may include a number of aircraft but, for safety reasons, these must be separated
- In this example, a simple vertical separation of 300m is proposed
- The system should warn the controller if aircraft are instructed to move so that the separation rule is breached

A sector object

- Critical operations on an object representing a controlled sector are
 - Enter. Add an aircraft to the controlled airspace
 - Leave. Remove an aircraft from the controlled airspace
 - Move. Move an aircraft from one height to another
 - Lookup. Given an aircraft identifier, return its current height

Primitive operations

- It is sometimes necessary to introduce additional operations to simplify the specification
- The other operations can then be defined using these more primitive operations
- Primitive operations
 - Create. Bring an instance of a sector into existence
 - Put. Add an aircraft without safety checks
 - In-space. Determine if a given aircraft is in the sector
 - Occupied. Given a height, determine if there is an aircraft within 300m of that height

Sector specification

SECTOR

sort Sector
imports INTEGER, BOOLEAN

Enter - adds an aircraft to the sector if safety conditions are satisfied
Leave - removes an aircraft from the sector
Move - moves an aircraft from one height to another if safe to do so
Lookup - Finds the height of an aircraft in the sector

Create - creates an empty sector
Put - adds an aircraft to a sector with no constraint checks
In-space - checks if an aircraft is already in a sector
Occupied - checks if a specified height is available

Enter (Sector, Call-sign, Height) → Sector
Leave (Sector, Call-sign) → Sector
Move (Sector, Call-sign, Height) → Sector
Lookup (Sector, Call-sign) → Height

Create → Sector
Put (Sector, Call-sign, Height) → Sector
In-space (Sector, Call-sign) → Boolean
Occupied (Sector, Height) → Boolean

```

Enter (S, CS, H) =
  if In-space (S, CS) then S exception (Aircraft already in sector)
  elseif Occupied (S, H) then S exception (Height conflict)
  else Put (S, CS, H)

Leave (Create, CS) = Create exception (Aircraft not in sector)
Leave (Put (S, CS1, H1), CS) =
  if CS = CS1 then S else Put (Leave (S, CS), CS1, H1)

Move (S, CS, H) =
  if S = Create then Create exception (No aircraft in sector)
  elseif not In-space (S, CS) then S exception (Aircraft not in sector)
  elseif Occupied (S, H) then S exception (Height conflict)
  else Put (Leave (S, CS), CS, H)

-- NO-HEIGHT is a constant indicating that a valid height cannot be returned

Lookup (Create, CS) = NO-HEIGHT exception (Aircraft not in sector)
Lookup (Put (S, CS1, H1), CS) =
  if CS = CS1 then H1 else Lookup (S, CS)

Occupied (Create, H) = false
Occupied (Put (S, CS1, H1), H) =
  if (H1 > H and H1 - H ≤ 300) or (H > H1 and H - H1 ≤ 300) then true
  else Occupied (S, H)

In-space (Create, CS) = false
In-space (Put (S, CS1, H1), CS) =
  if CS = CS1 then true else In-space (S, CS)
  
```

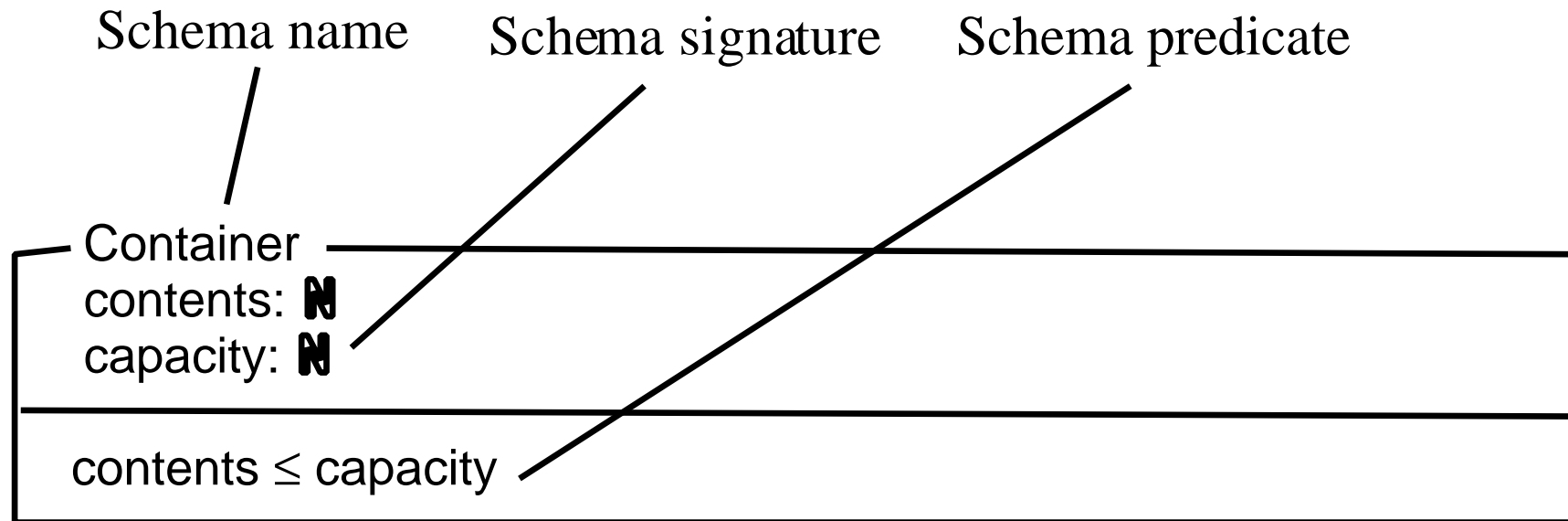
Specification commentary

- Use the basic constructors Create and Put to specify other operations
- Define Occupied and In-space using Create and Put and use them to make checks in other operation definitions
- All operations that result in changes to the sector must check that the safety criterion holds

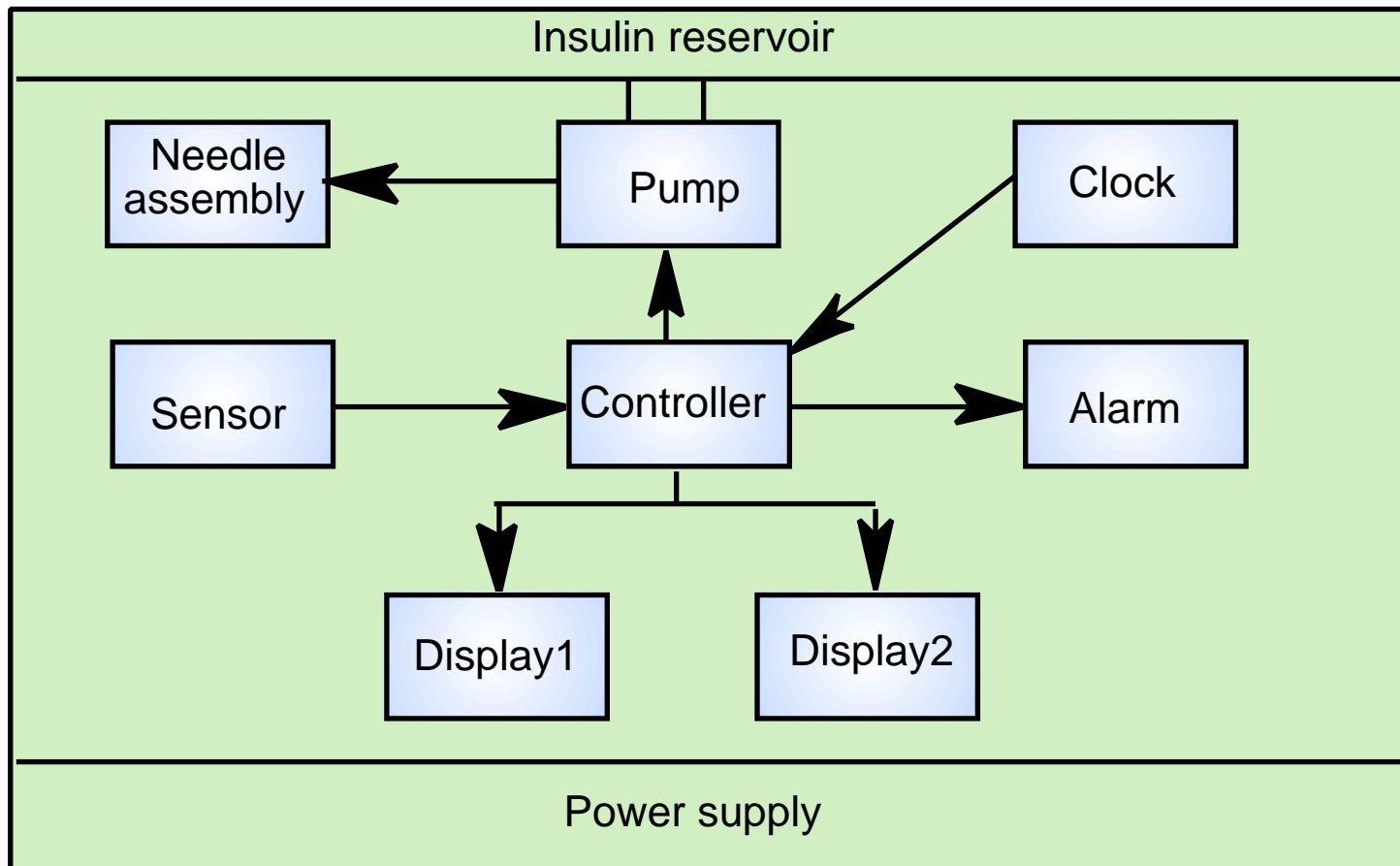
Behavioural specification

- Algebraic specification can be cumbersome when the object operations are not independent of the object state
- Model-based specification exposes the system state and defines the operations in terms of changes to that state
- The Z notation is a mature technique for model-based specification. It combines formal and informal description and uses graphical highlighting when presenting specifications

The structure of a Z schema



An insulin pump



Modelling the insulin pump

- The schema models the insulin pump as a number of state variables
 - reading?
 - dose, cumulative_dose
 - r0, r1, r2
 - capacity
 - alarm!
 - pump!
 - display1!, display2!
- Names followed by a ? are inputs, names followed by a ! are outputs

Schema invariant

- Each Z schema has an invariant part which defines conditions that are always true
- For the insulin pump schema it is always true that
 - The dose must be less than or equal to the capacity of the insulin reservoir
 - No single dose may be more than 5 units of insulin and the total dose delivered in a time period must not exceed 50 units of insulin. This is a safety constraint (see Chapters 16 and 17)
 - `display1!` shows the status of the insulin reservoir.

Insulin pump schema

Insulin_pump

reading? : \mathbb{N}

dose, cumulative_dose: \mathbb{N}

r0, r1, r2: \mathbb{N} // used to record the last 3 readings taken

capacity: \mathbb{N}

alarm!: {off, on}

pump!: \mathbb{N}

display1!, display2!: STRING

$\text{dose} \leq \text{capacity} \wedge \text{dose} \leq 5 \wedge \text{cumulative_dose} \leq 50$

$\text{capacity} \geq 40 \Rightarrow \text{display1!} = " "$

$\text{capacity} \leq 39 \wedge \text{capacity} \geq 10 \Rightarrow \text{display1!} = \text{"Insulin low"}$

$\text{capacity} \leq 9 \Rightarrow \text{alarm!} = \text{on} \wedge \text{display1!} = \text{"Insulin very low"}$

$\text{r2} = \text{reading?}$

The dosage computation

- The insulin pump computes the amount of insulin required by comparing the current reading with two previous readings
- If these suggest that blood glucose is rising then insulin is delivered
- Information about the total dose delivered is maintained to allow the safety check invariant to be applied
- Note that this invariant always applies - there is no need to repeat it in the dosage computation

DOSAGE schema

DOSAGE
 Δ Insulin_Pump

```
(
dose = 0 ∧
(
(( r1 ≥ r0) ∧ ( r2 = r1)) ∨
(( r1 > r0) ∧ ( r2 ≤ r1)) ∨
(( r1 < r0) ∧ ((r1-r2) > (r0-r1)))
) ∨
dose = 4 ∧
(
(( r1 ≤ r0) ∧ (r2=r1)) ∨
(( r1 < r0) ∧ ((r1-r2) ≤ (r0-r1)))
) ∨
dose =(r2 -r1) * 4 ∧
(
(( r1 ≤ r0) ∧ ( r2 > r1)) ∨
(( r1 > r0) ∧ ((r2 - r1) ≥ (r1 - r0)))
)
)
capacity' = capacity - dose
cumulative_dose' = cumulative_dose + dose
r0' = r1 ∧ r1' = r2
```

Output schemas

- The output schemas model the system displays and the alarm that indicates some potentially dangerous condition
- The output displays show the dose computed and a warning message
- The alarm is activated if blood sugar is very low - this indicates that the user should eat something to increase their blood sugar level

Output schemas

DISPLAY
 Δ Insulin_Pump

$\text{display2!} = \text{Nat_to_string}(\text{dose}) \wedge$
 $(\text{reading?} < 3 \Rightarrow \text{display1!} = \text{"Sugar low"}) \vee$
 $\text{reading?} > 30 \Rightarrow \text{display1!} = \text{"Sugar high"} \vee$
 $\text{reading?} \geq 3 \text{ and } \text{reading?} \leq 30 \Rightarrow \text{display1!} = \text{"OK"})$

ALARM
 Δ Insulin_Pump

$(\text{reading?} < 3 \vee \text{reading?} > 30) \Rightarrow \text{alarm!} = \text{on} \vee$
 $(\text{reading?} \geq 3 \wedge \text{reading?} \leq 30) \Rightarrow \text{alarm!} = \text{off}$

Schema consistency

- It is important that schemas are consistent. Inconsistency suggests a problem with the system requirements
- The INSULIN_PUMP schema and the DISPLAY are inconsistent
 - display1! shows a warning message about the insulin reservoir (INSULIN_PUMP)
 - display1! Shows the state of the blood sugar (DISPLAY)
- This must be resolved before implementation of the system

Key points

- Formal system specification complements informal specification techniques
- Formal specifications are precise and unambiguous. They remove areas of doubt in a specification
- Formal specification forces an analysis of the system requirements at an early stage. Correcting errors at this stage is cheaper than modifying a delivered system

Key points

- Formal specification techniques are most applicable in the development of critical systems and standards.
- Algebraic techniques are suited to interface specification where the interface is defined as a set of object classes
- Model-based techniques model the system using sets and functions. This simplifies some types of behavioural specification

Architectural Design

- Establishing the overall structure of a software system

Objectives

- To introduce architectural design and to discuss its importance
- To explain why multiple models are required to document a software architecture
- To describe types of architectural model that may be used
- To discuss how domain-specific reference models may be used as a basis for product-lines and to compare software architectures

Topics covered

- System structuring
- Control models
- Modular decomposition
- Domain-specific architectures

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*

Architectural design

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

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

Architectural design process

- System structuring
 - The system is decomposed into several principal sub-systems and communications between these sub-systems are identified
- Control modelling
 - A model of the control relationships between the different parts of the system is established
- Modular decomposition
 - The identified sub-systems are decomposed into modules

Sub-systems and modules

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

Architectural models

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

Architectural models

- 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

Architectural styles

- The architectural model of a system may conform to a generic architectural model or style
- An awareness of these styles can simplify the problem of defining system architectures
- However, most large systems are heterogeneous and do not follow a single architectural style

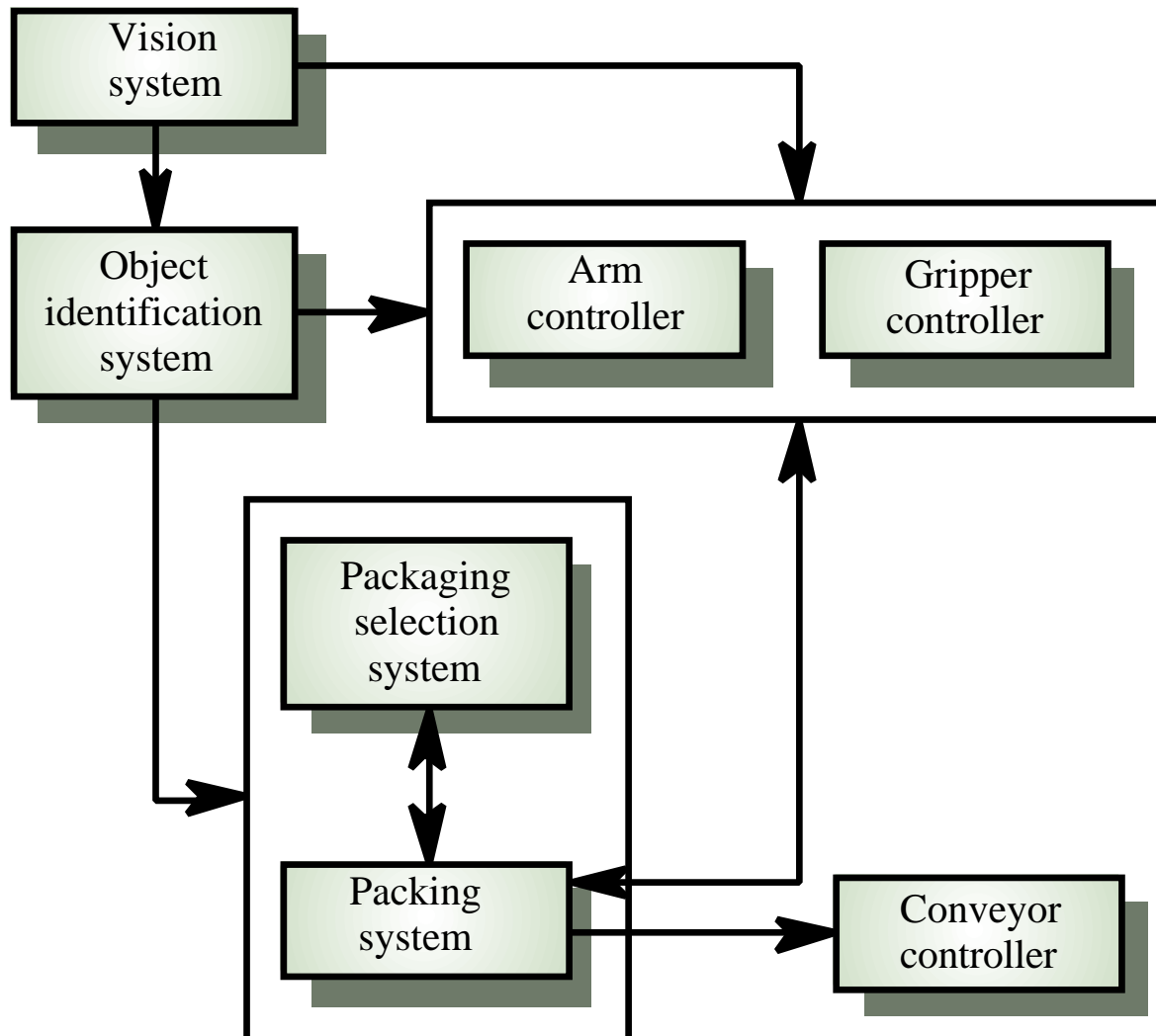
Architecture attributes

- Performance
 - Localise operations to minimise sub-system communication
- Security
 - Use a layered architecture with critical assets in inner layers
- Safety
 - Isolate safety-critical components
- Availability
 - Include redundant components in the architecture
- Maintainability
 - Use fine-grain, self-contained components

System structuring

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

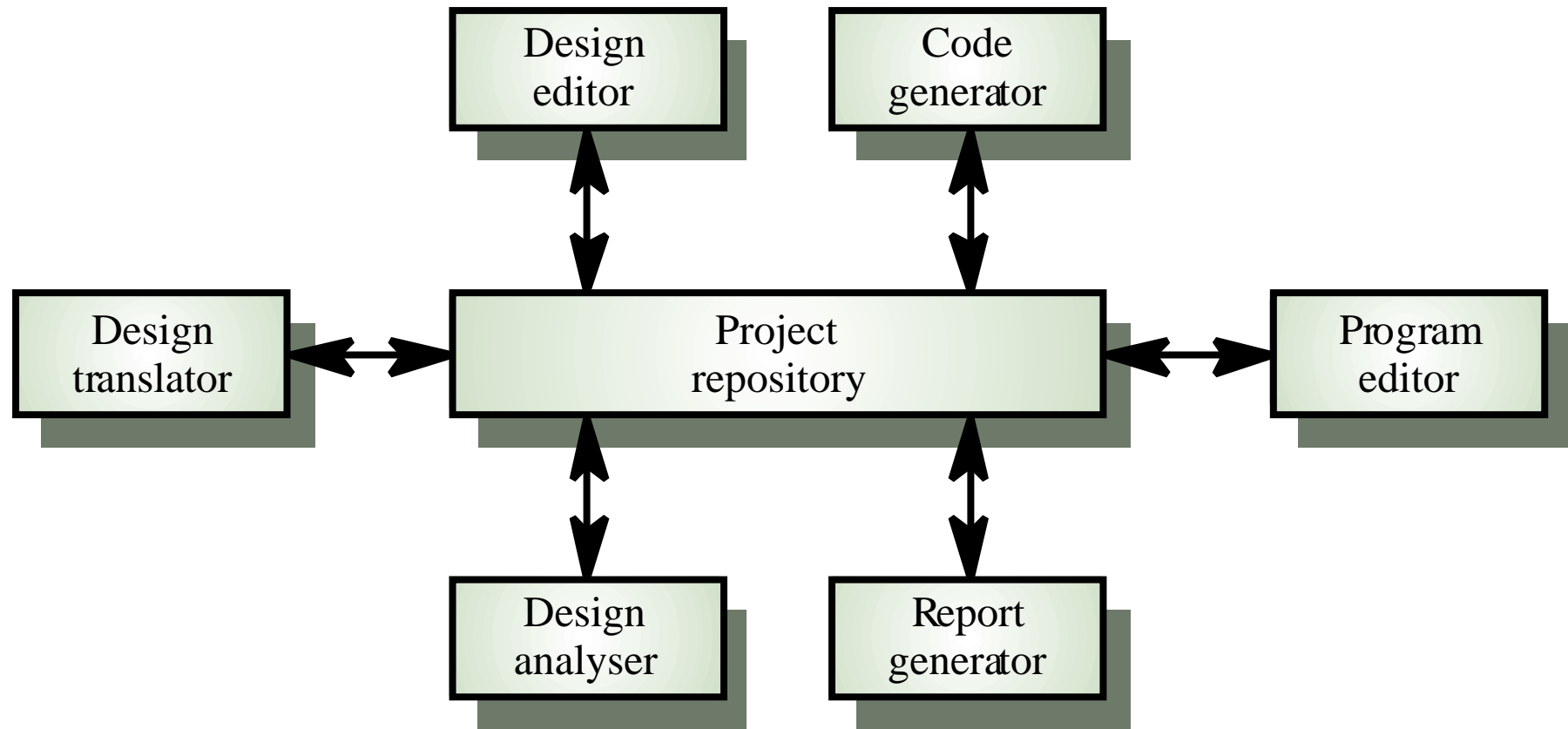
Packing robot control system



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

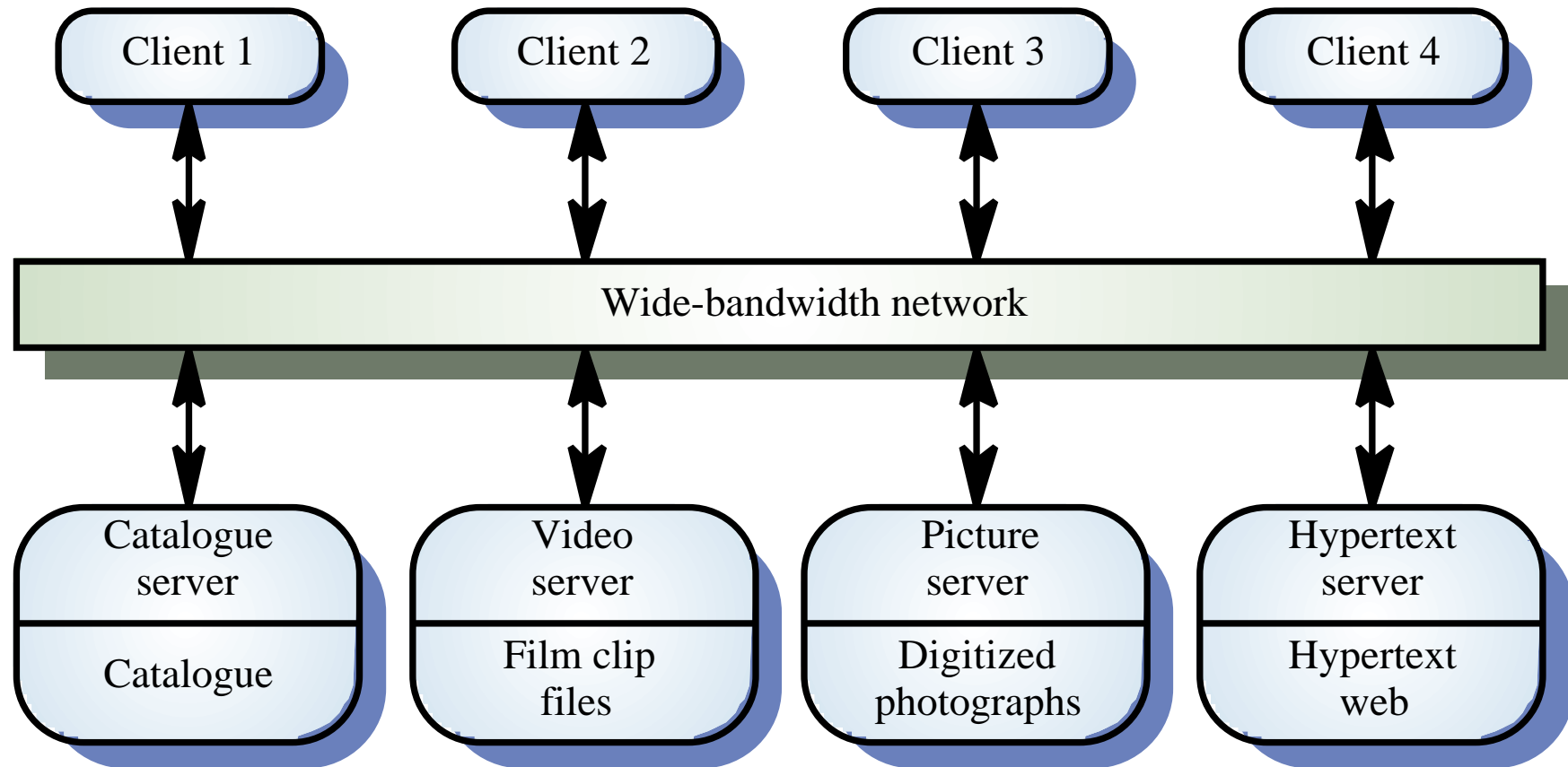
- Disadvantages

- 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 architecture

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

Film and picture library



Client-server characteristics

- Advantages

- Distribution of data is straightforward
- Makes effective use of networked systems. May require cheaper hardware
- Easy to add new servers or upgrade existing servers

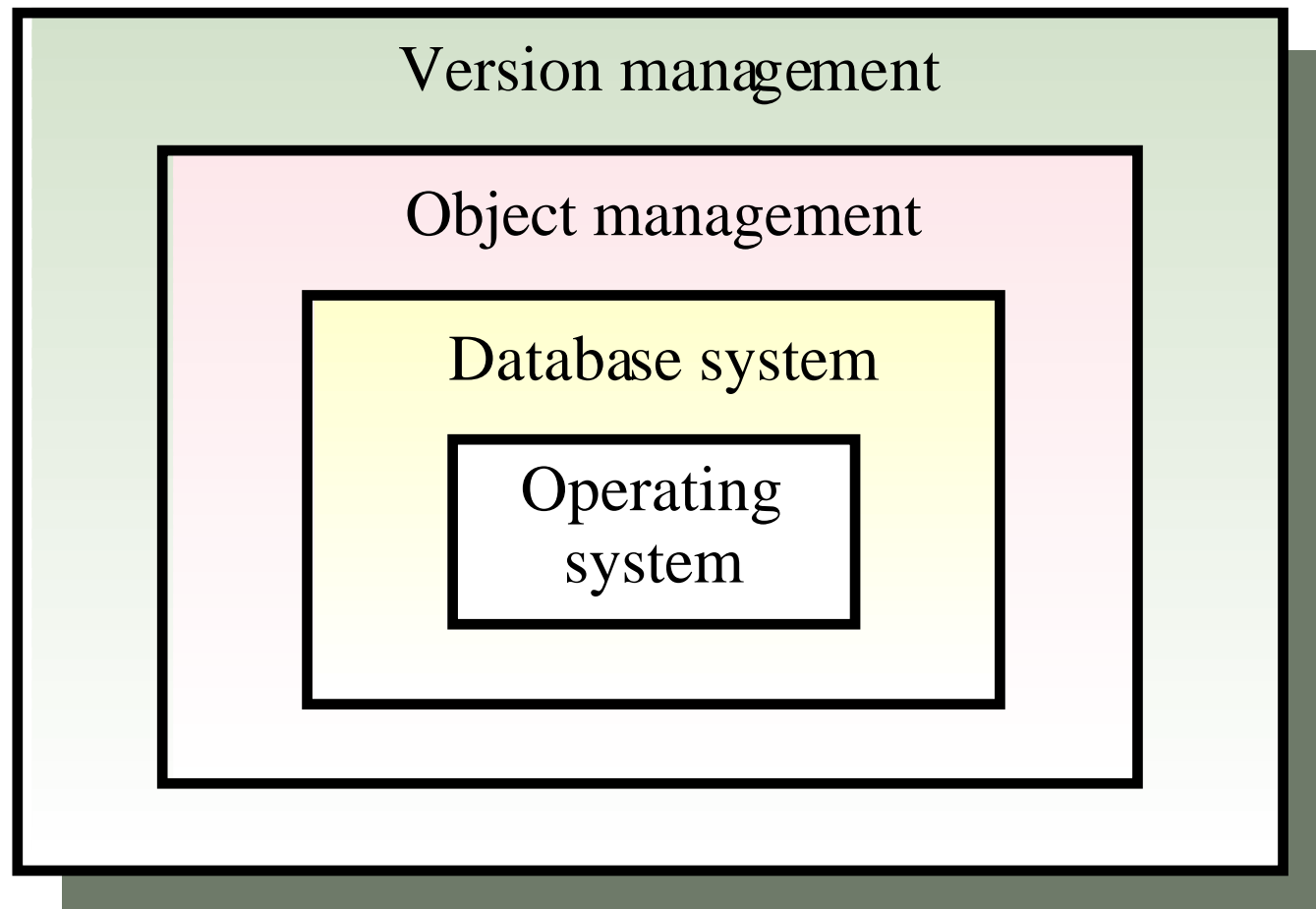
- Disadvantages

- 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 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 difficult to structure systems in this way

Version management system



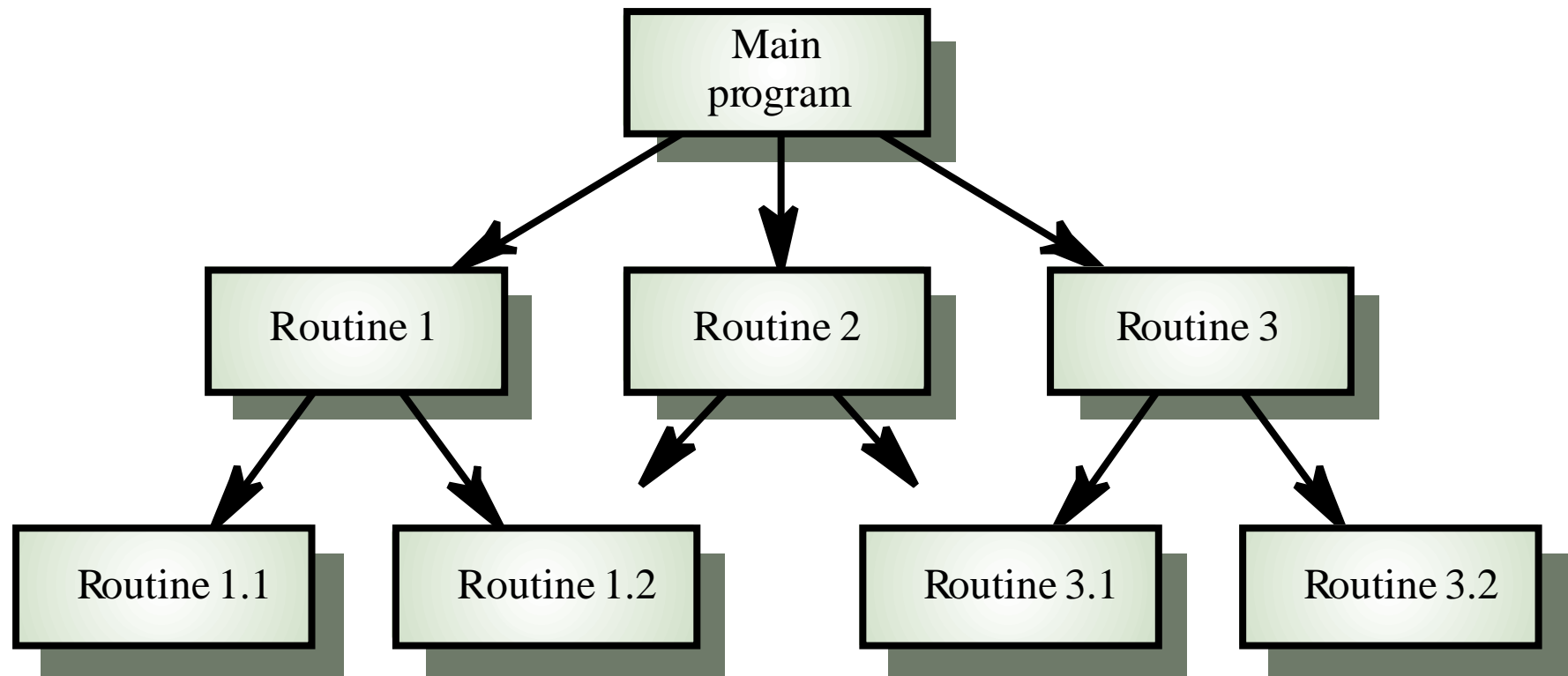
Control models

- 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

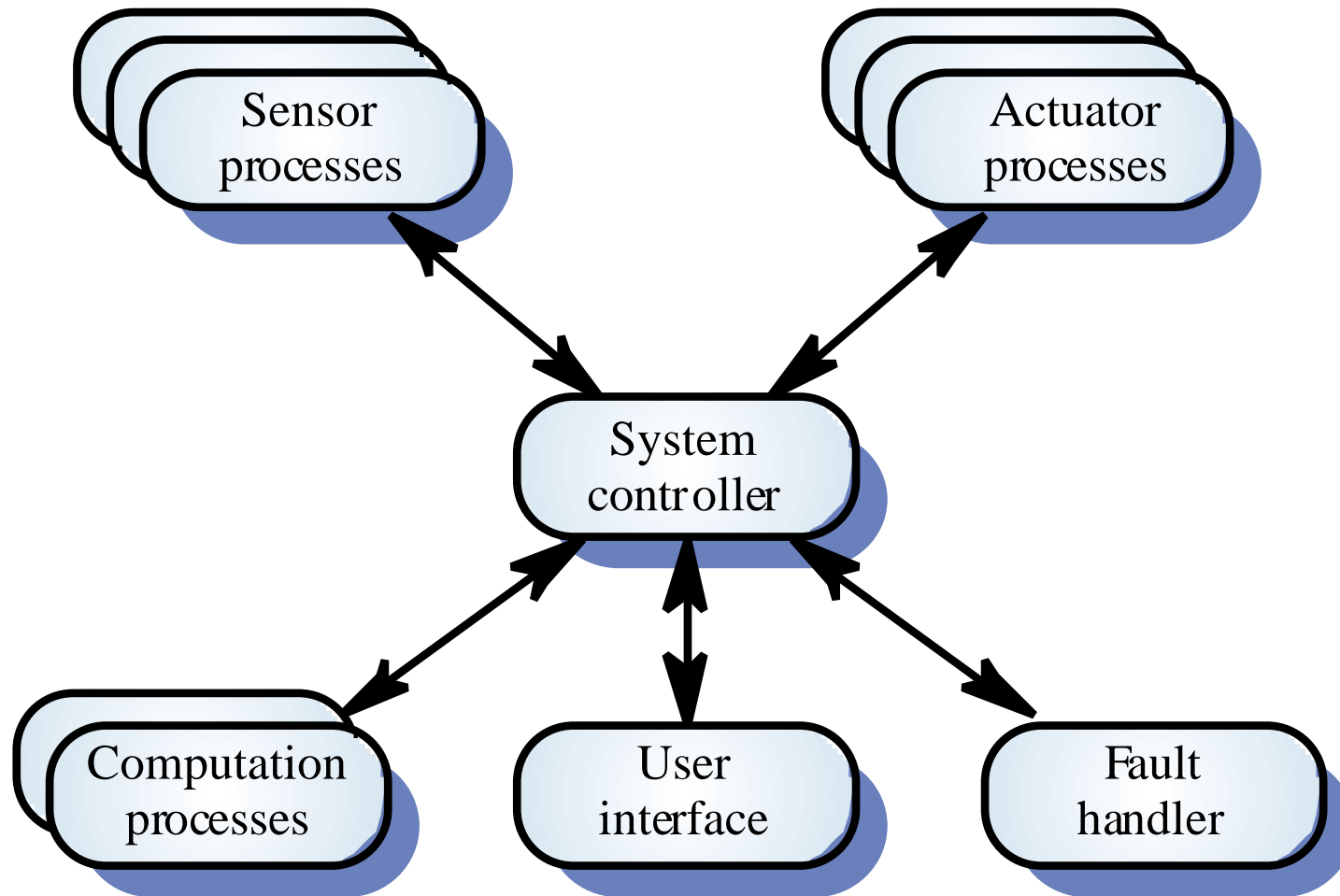
Centralised control

- 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



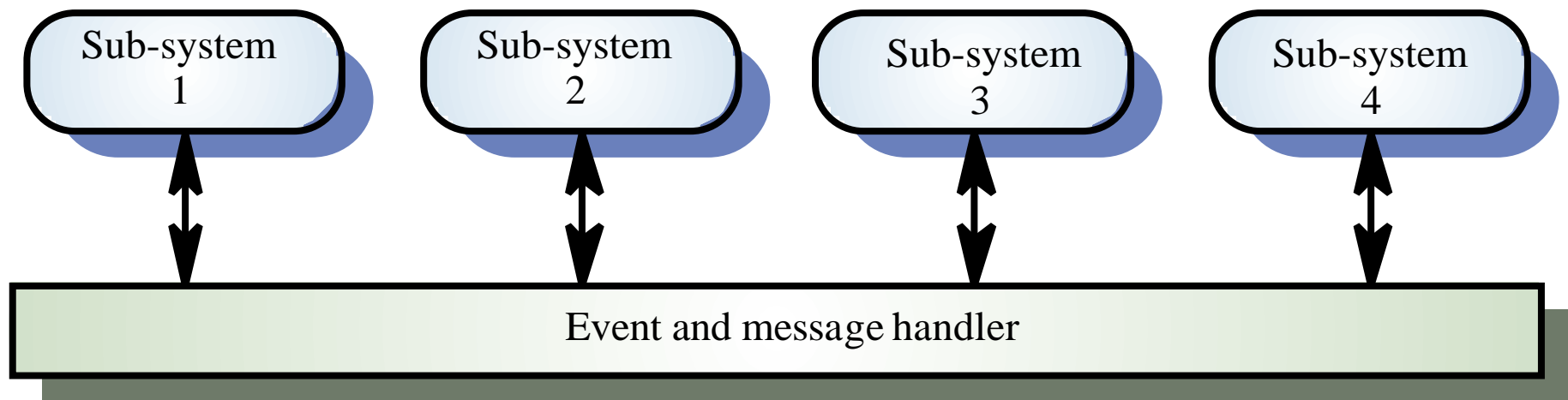
Event-driven systems

- 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

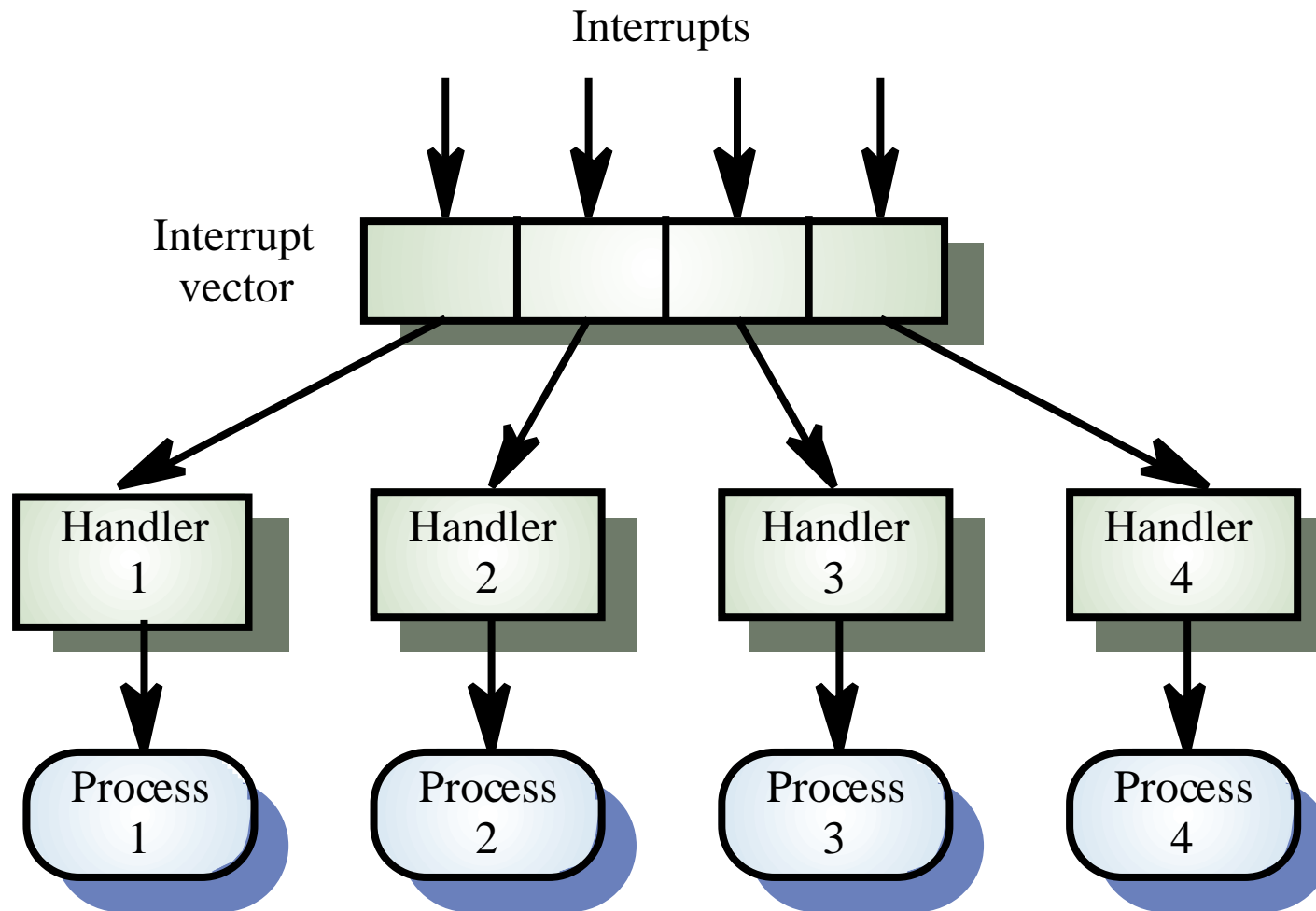
Selective broadcasting



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



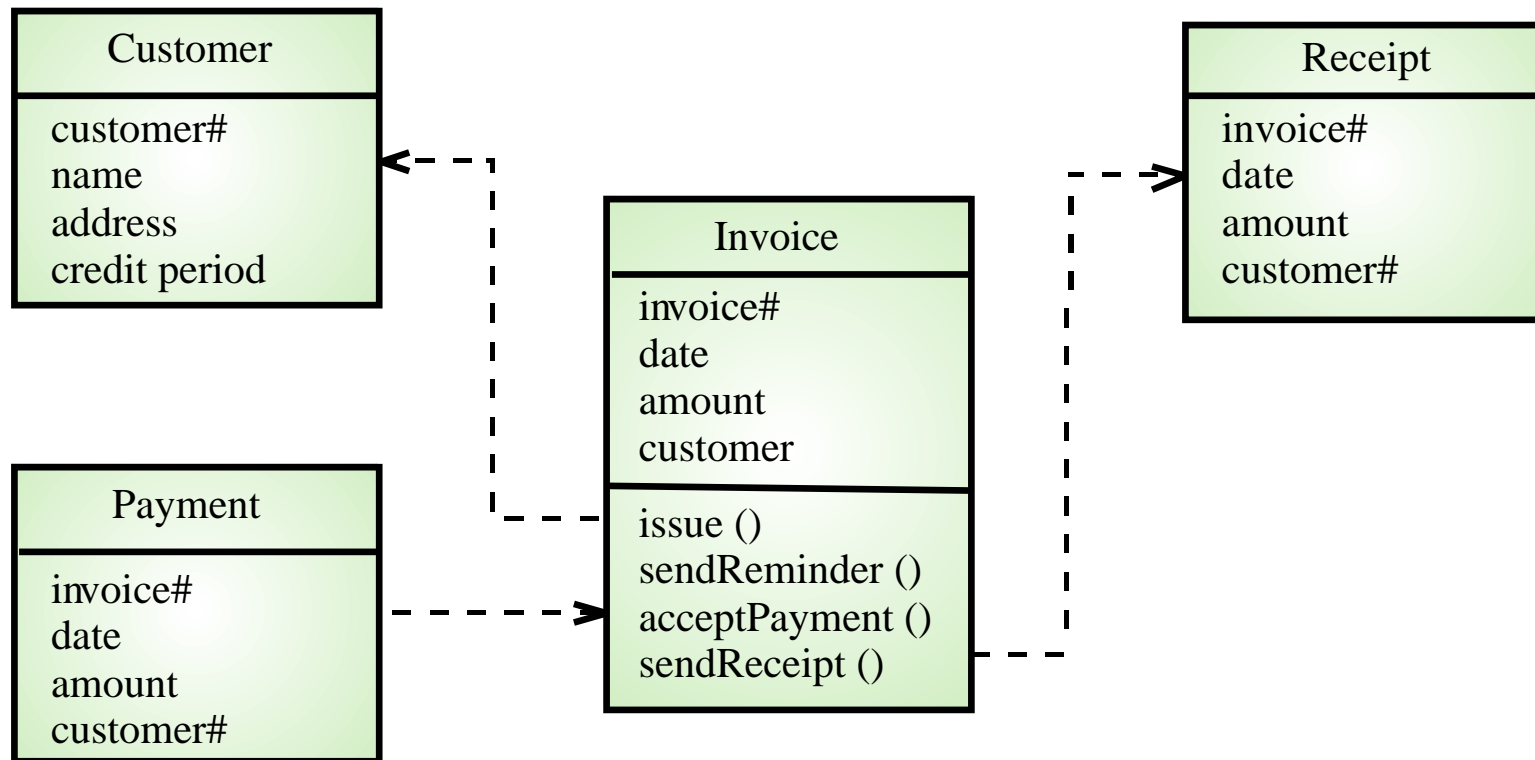
Modular decomposition

- 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 data-flow model where the system is decomposed into functional modules which transform inputs to outputs. Also known as the pipeline model
- 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

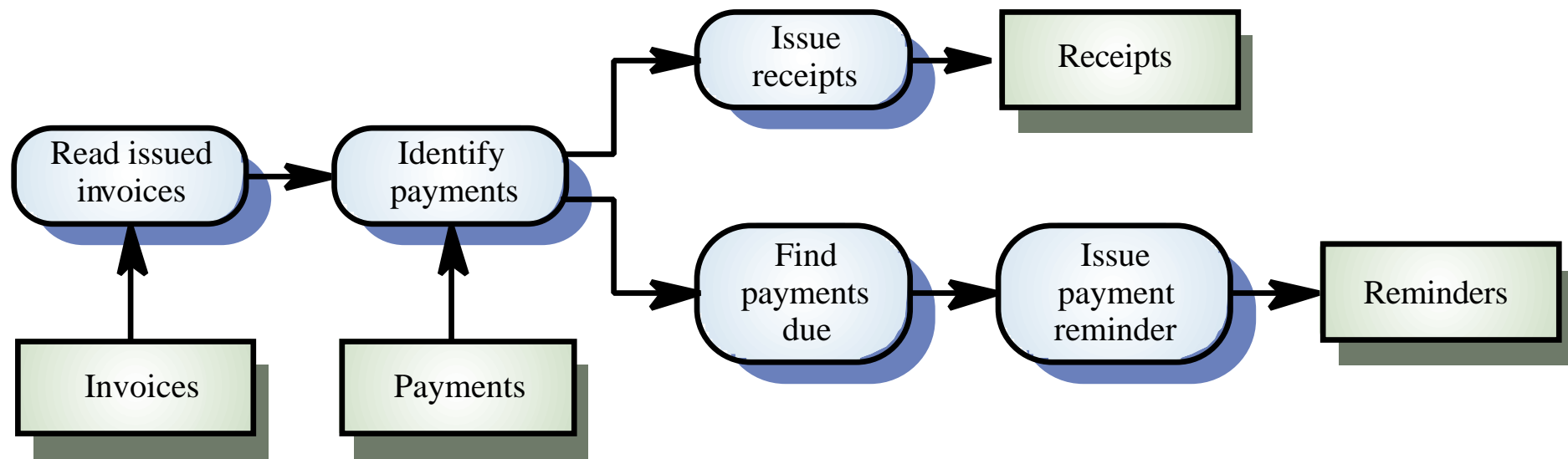
Invoice processing system



Data-flow models

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

Invoice processing system



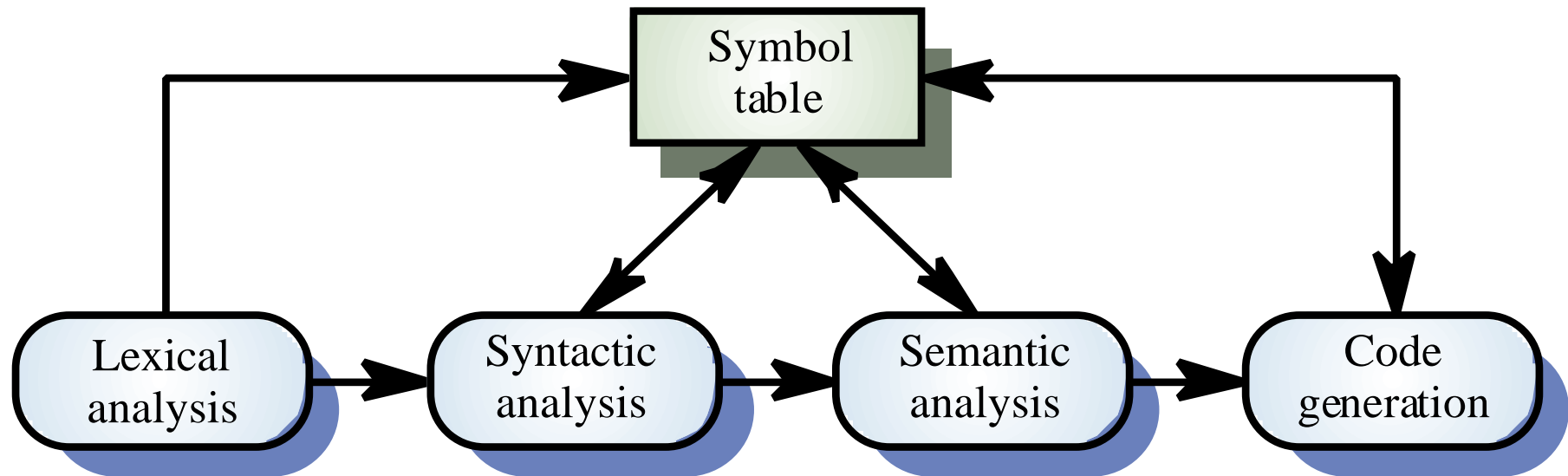
Domain-specific architectures

- Architectural models which are specific to some application domain
- Two types of domain-specific model
 - Generic models which are abstractions from a number of real systems and which encapsulate the principal characteristics of these systems
 - Reference models which are more abstract, 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

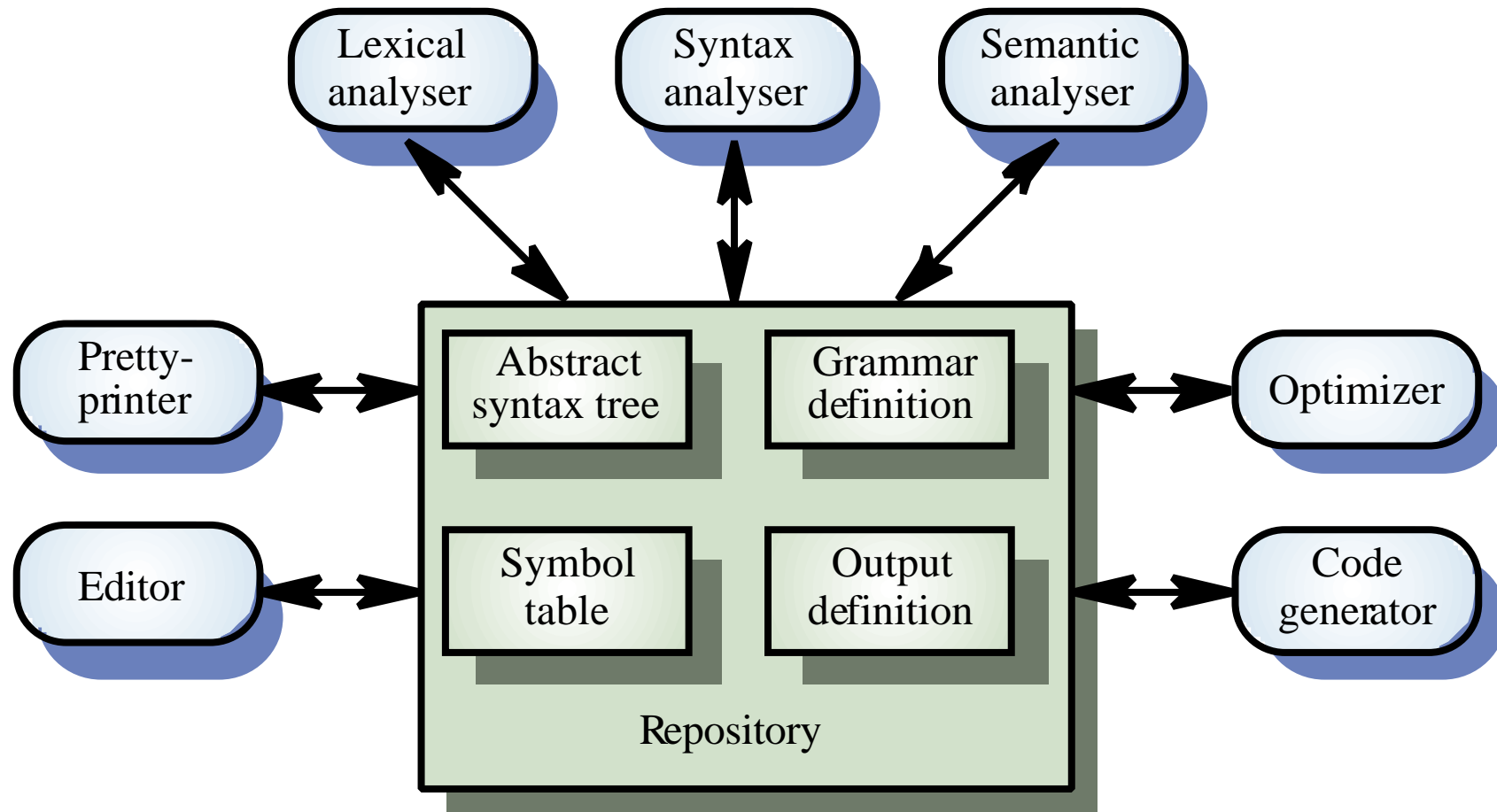
Generic models

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

Compiler model



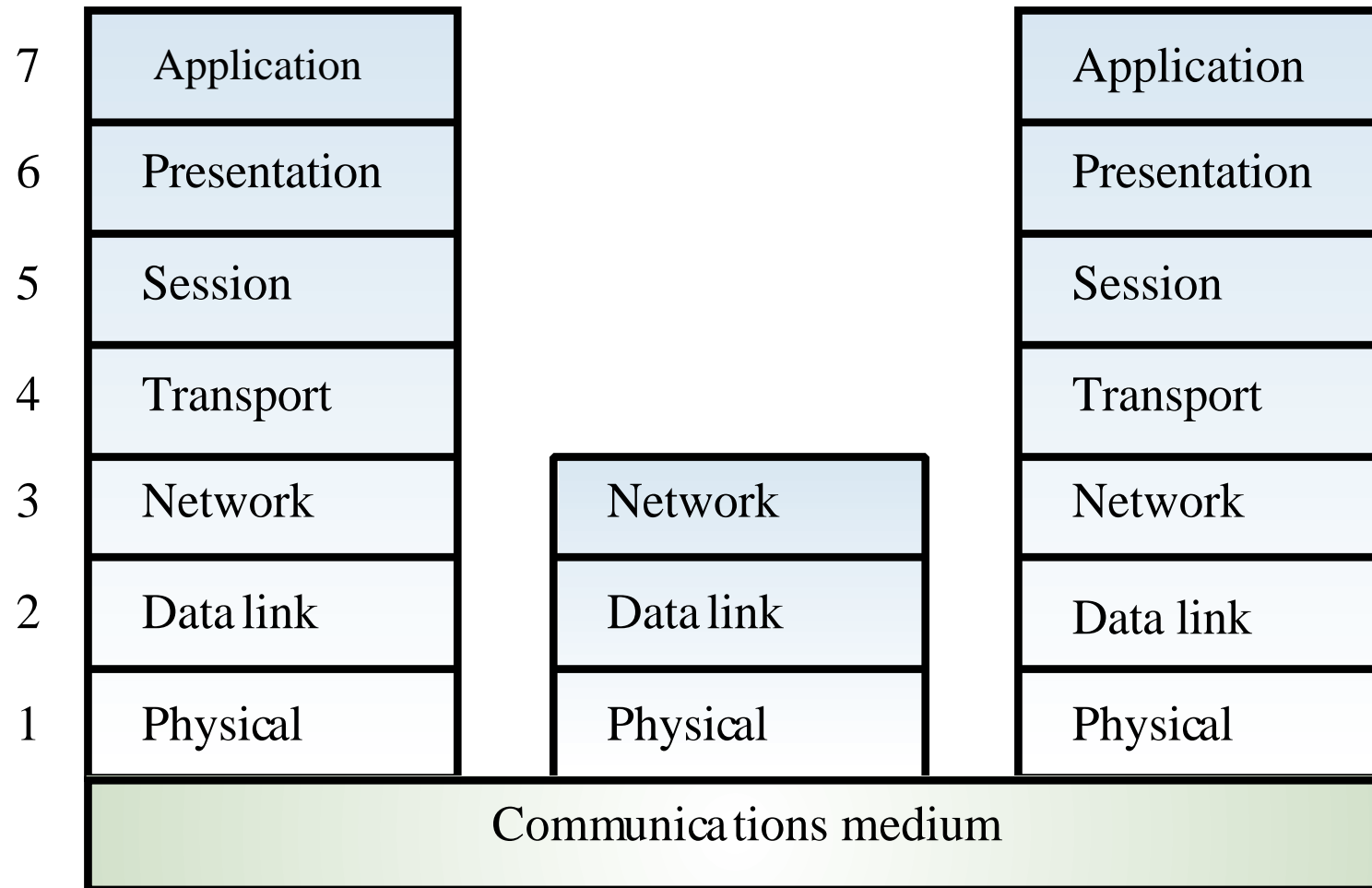
Language processing system



Reference architectures

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

OSI reference model



Key points

- The software architect is responsible for deriving a structural system model, a control model and a sub-system decomposition model
- Large systems rarely conform to a single architectural model
- System decomposition models include repository models, client-server models and abstract machine models
- Control models include centralised control and event-driven models

Key points

- Modular decomposition models include data-flow and object models
- Domain specific architectural models are abstractions over an application domain. They may be constructed by abstracting from existing systems or may be idealised reference models

Distributed Systems Architectures

Architectural design for software
that executes on more than one
processor

Objectives

- To explain the advantages and disadvantages of distributed systems architectures
- To describe different approaches to the development of client-server systems
- To explain the differences between client-server and distributed object architectures
- To describe object request brokers and the principles underlying the CORBA standards

Topics covered

- Multiprocessor architectures
- Client-server architectures
- Distributed object architectures
- CORBA

Distributed systems

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

System types

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

Distributed system characteristics

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

Distributed system disadvantages

- Complexity
- Security
- Manageability
- Unpredictability

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

Issues in distributed system design

Distributed systems 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

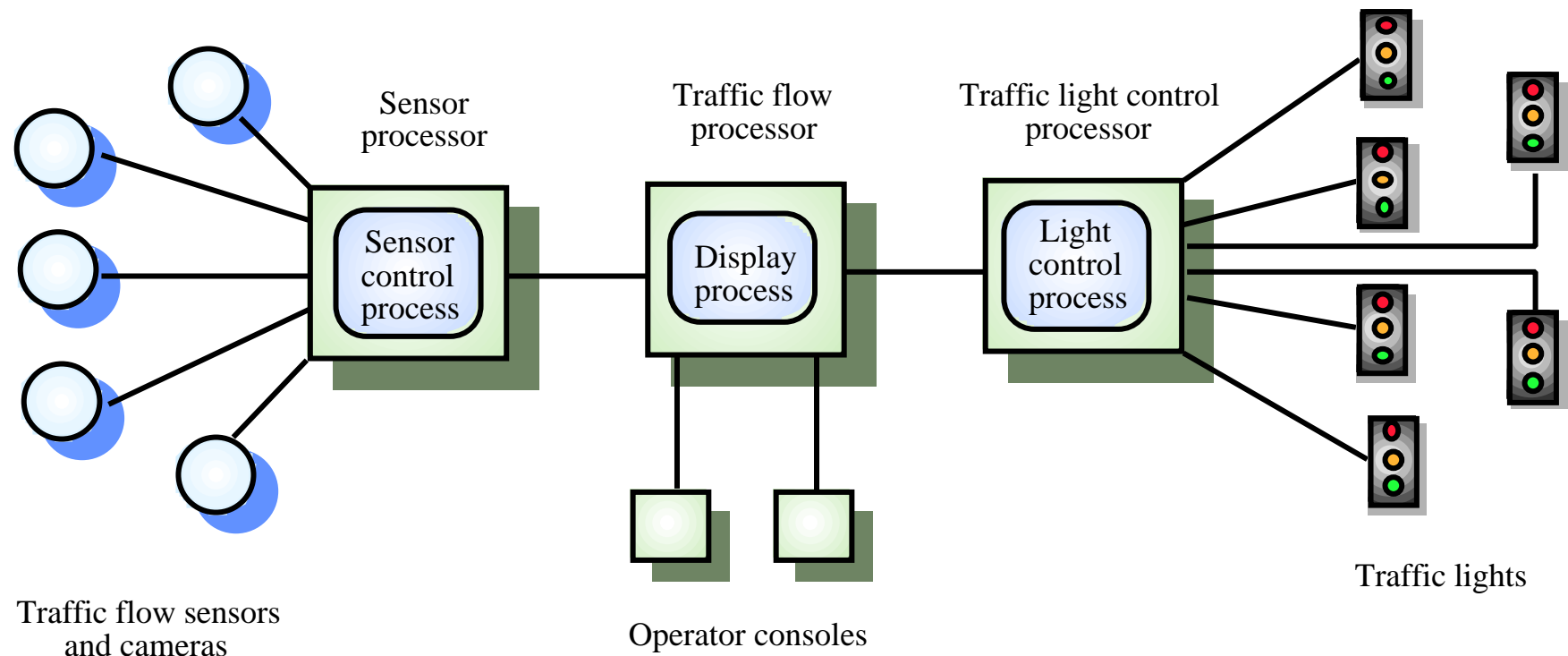
Middleware

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

Multiprocessor architectures

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

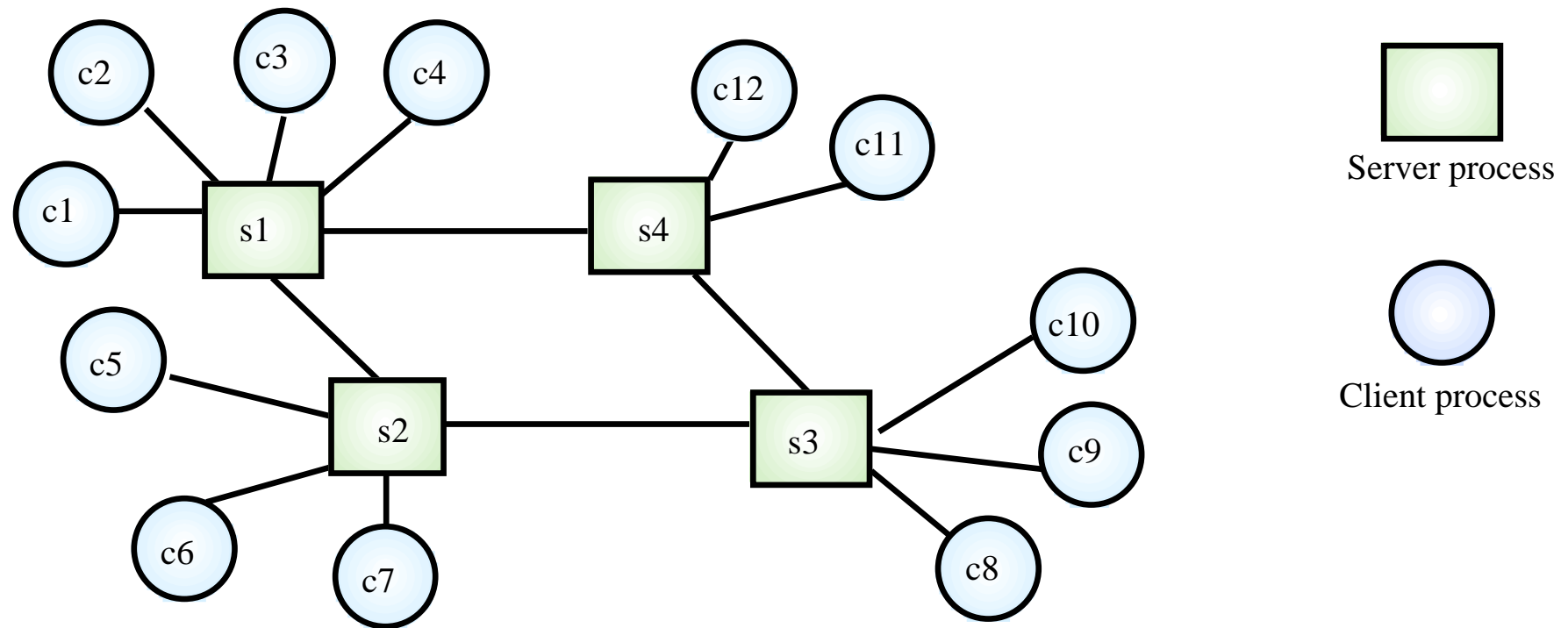
A multiprocessor traffic control system



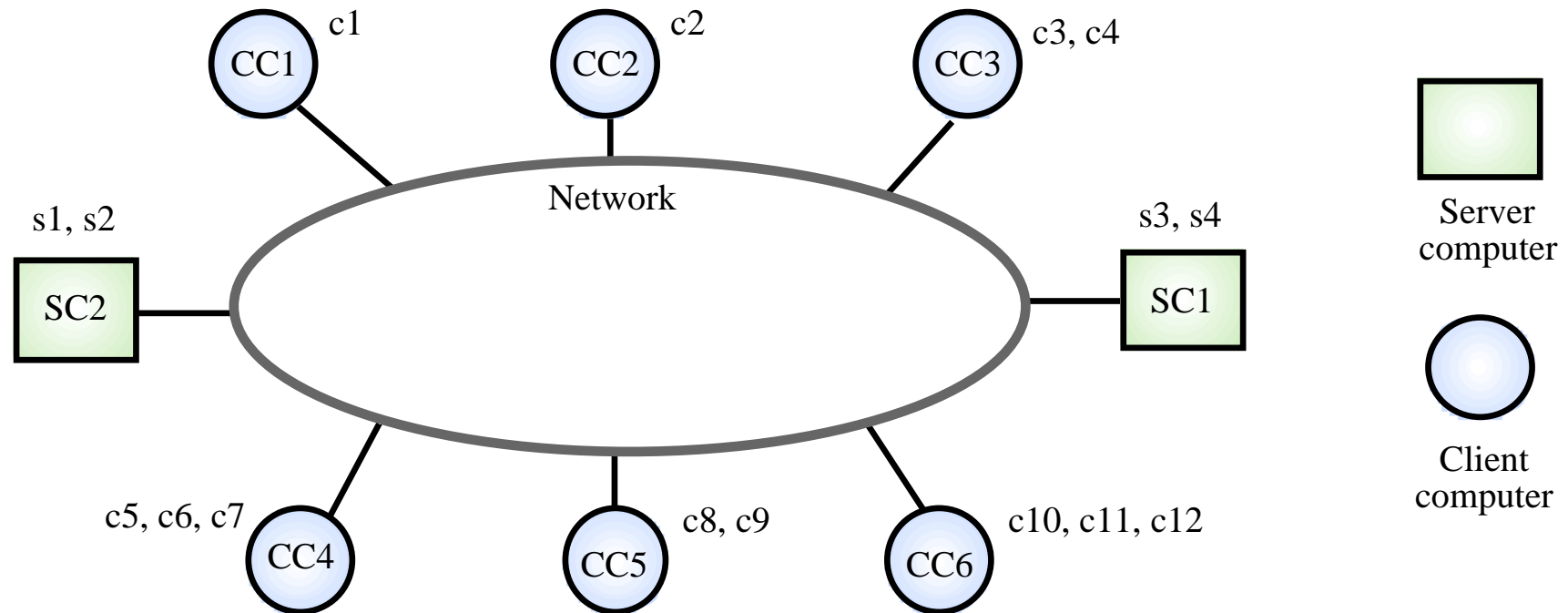
Client-server architectures

- 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



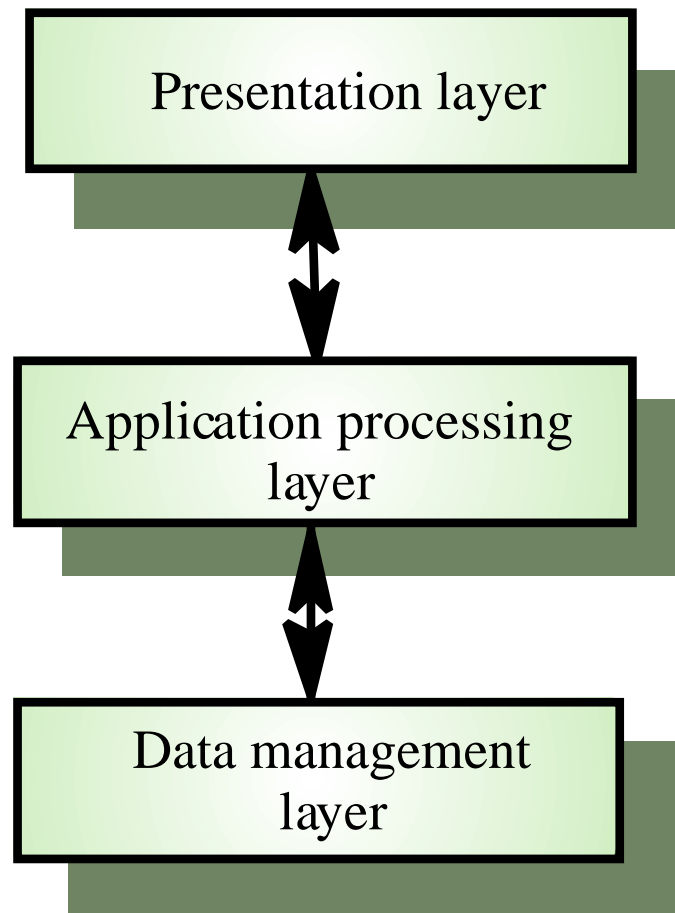
Computers in a C/S network



Layered application architecture

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

Application layers



Thin and fat clients

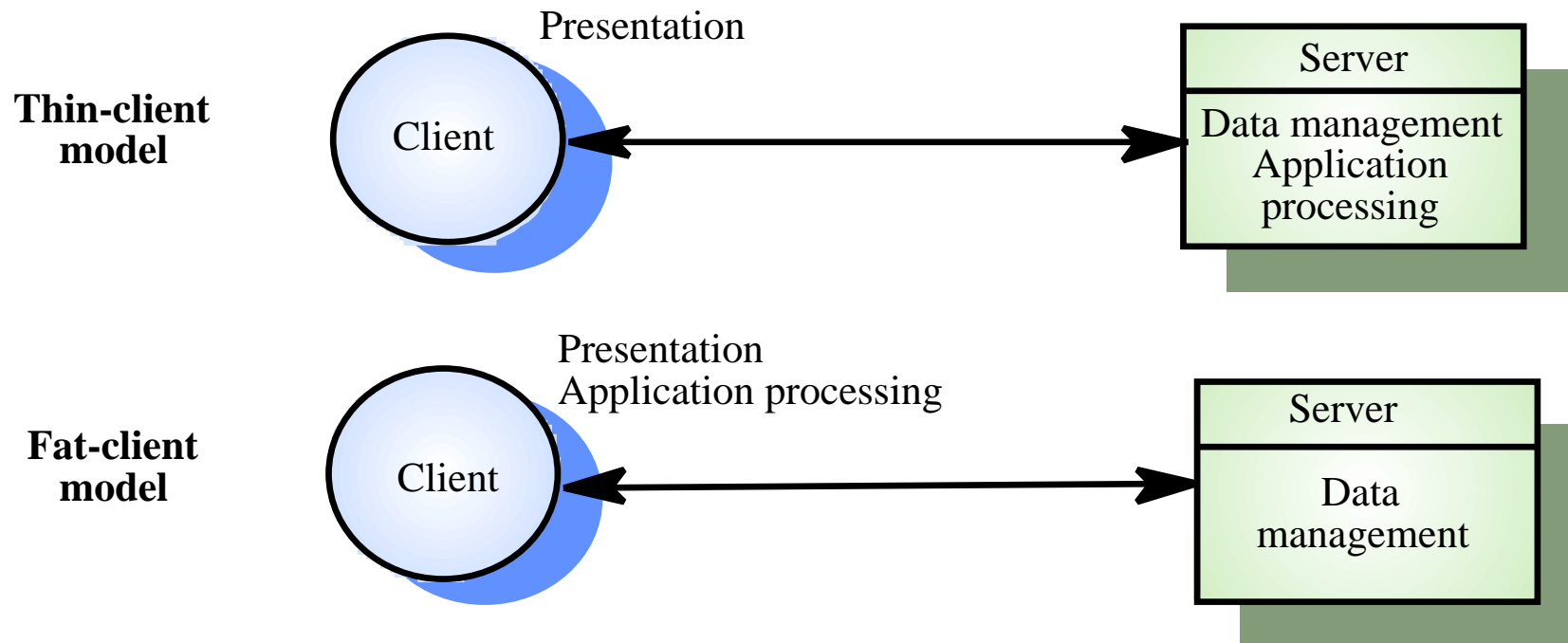
- *Thin-client model*

- In a thin-client model, all of the application processing and data management is carried out on the server. The client is simply responsible for running the presentation software.

- *Fat-client model*

- In this model, the server is only responsible for data management. The software on the client implements the application logic and the interactions with the system user.

Thin and fat clients



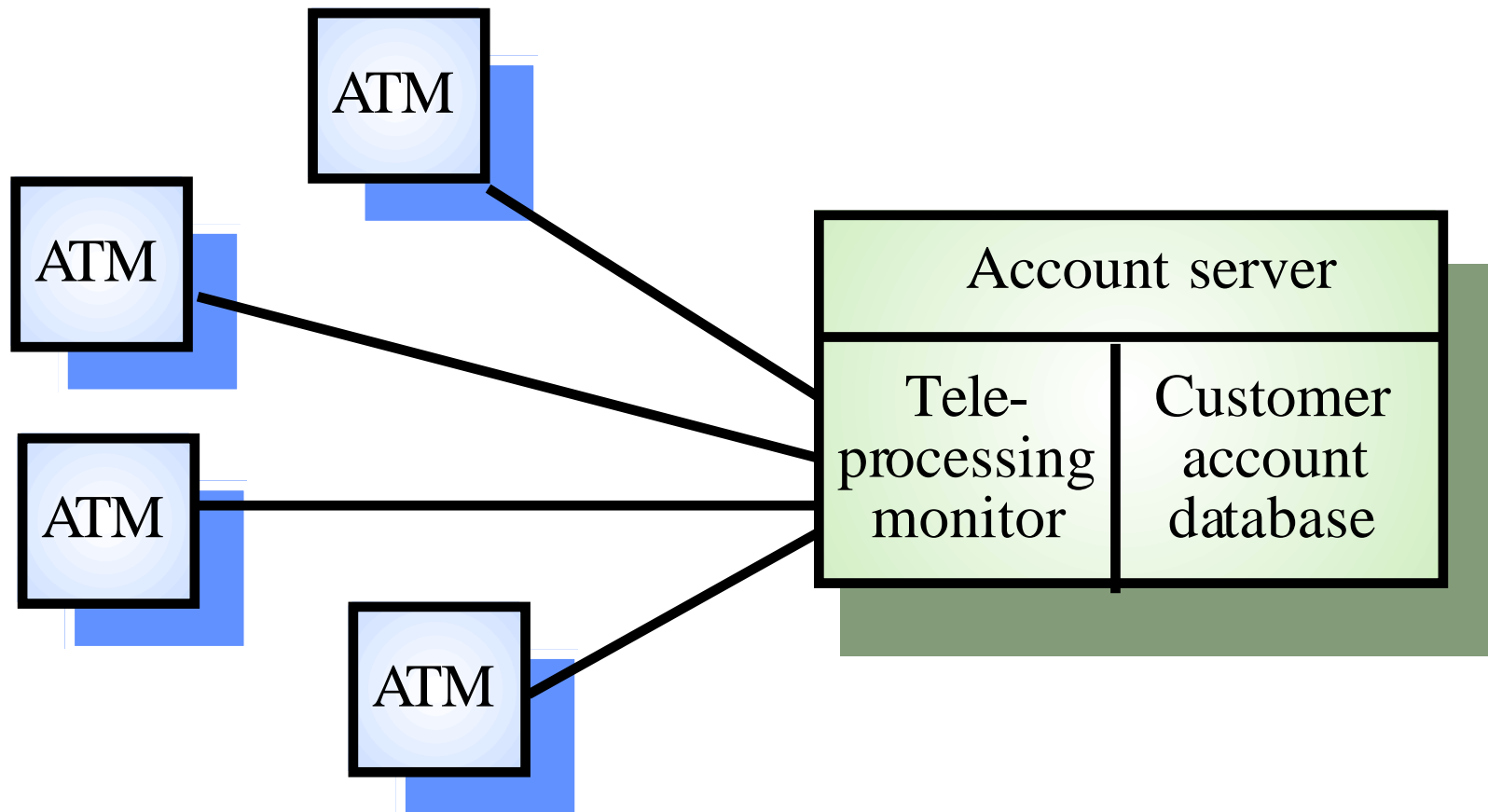
Thin client model

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

Fat client model

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

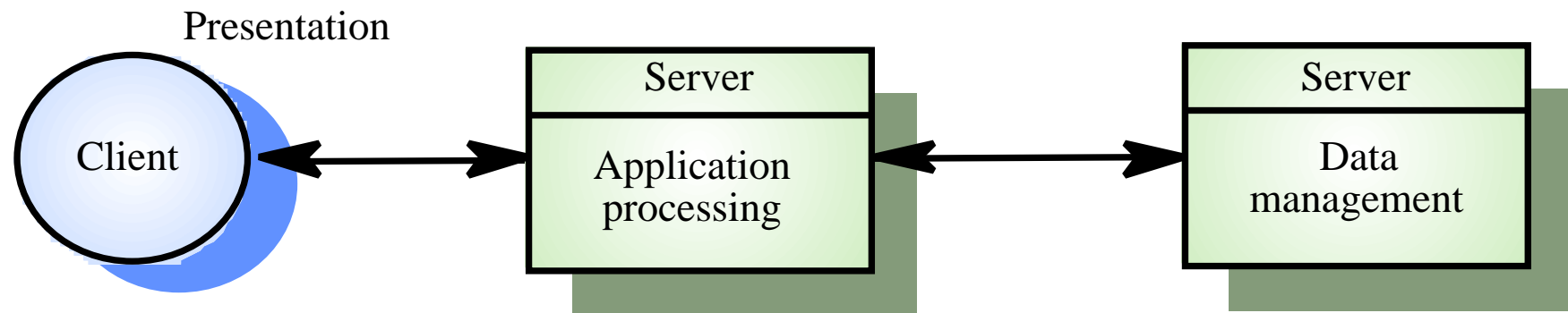
A client-server ATM system



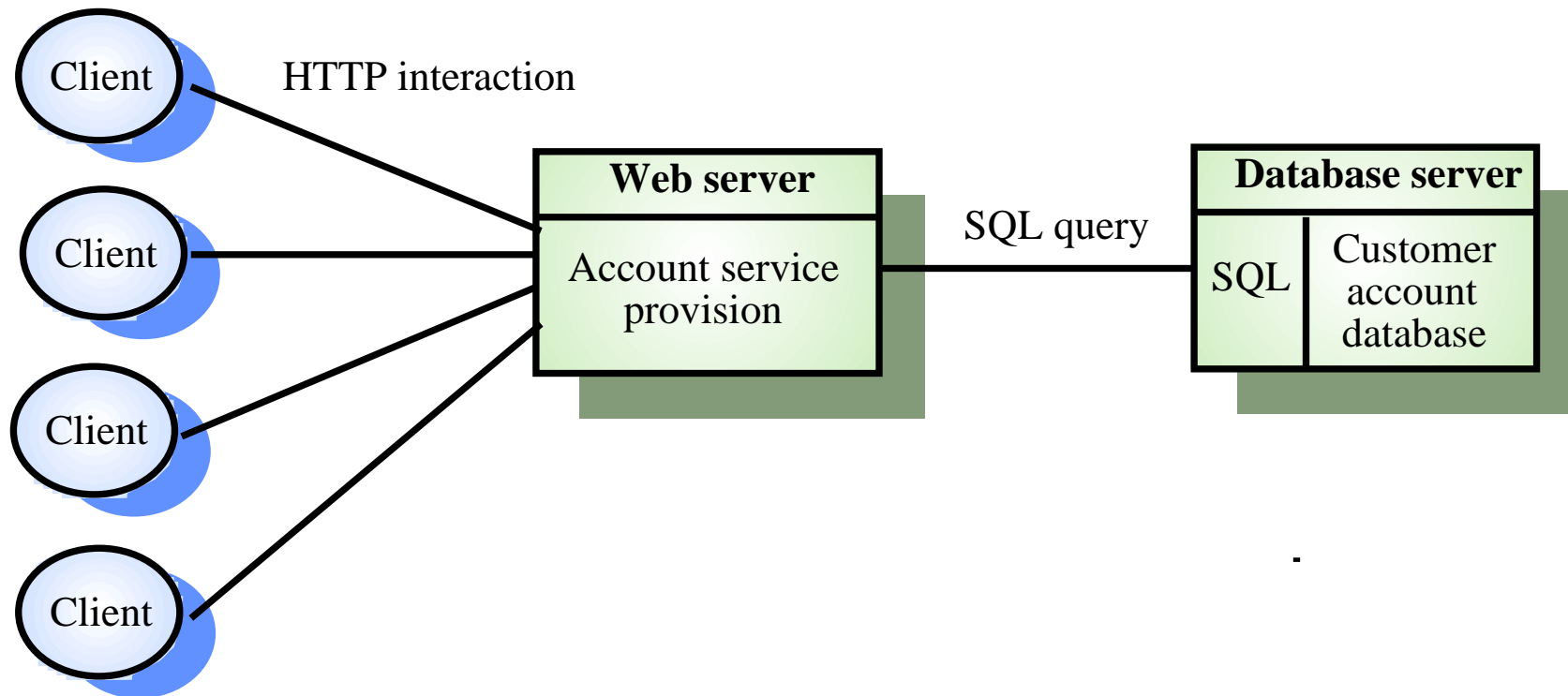
Three-tier architectures

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

A 3-tier C/S architecture



An internet banking system



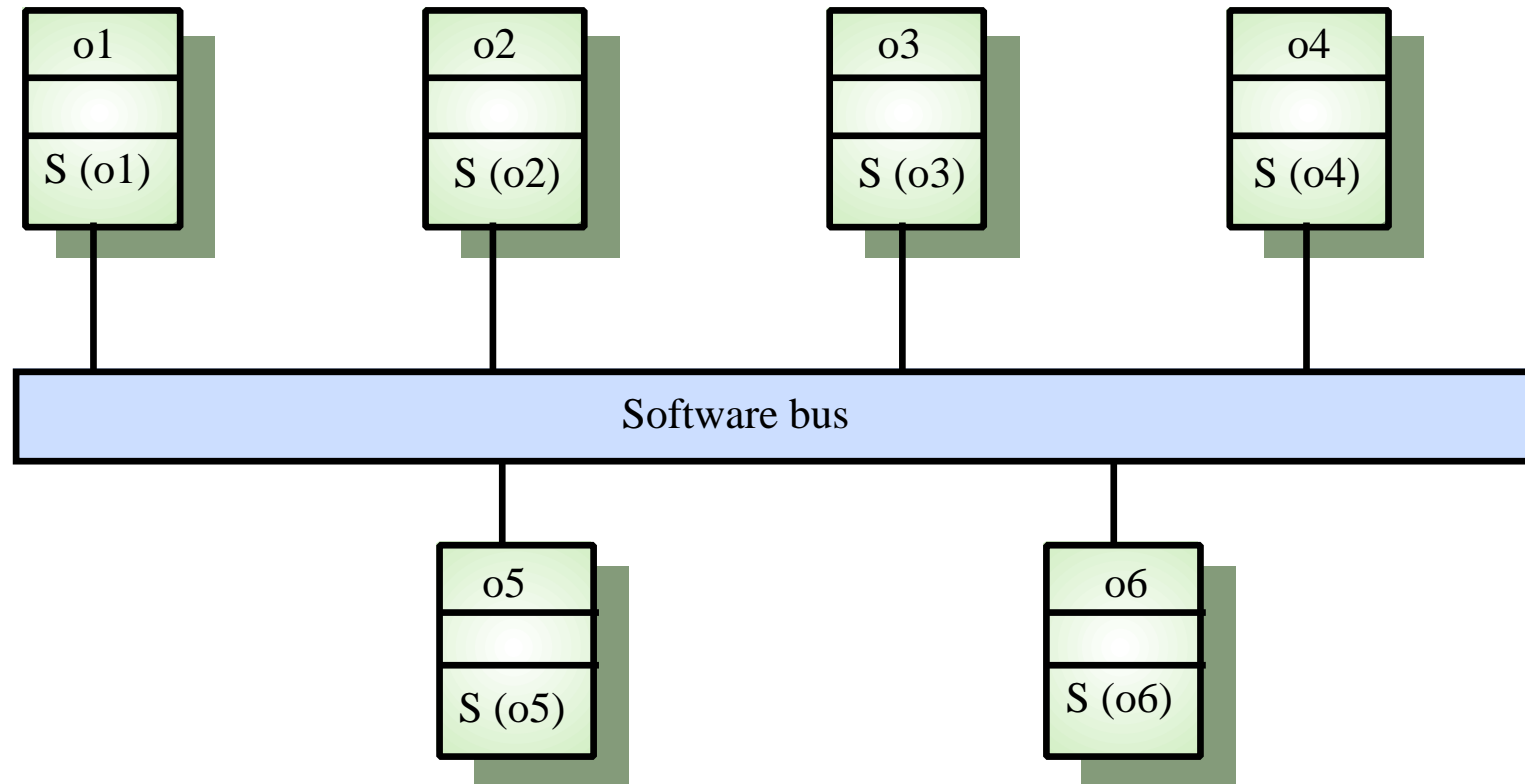
Use of C/S architectures

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

Distributed object architectures

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

Distributed object architecture



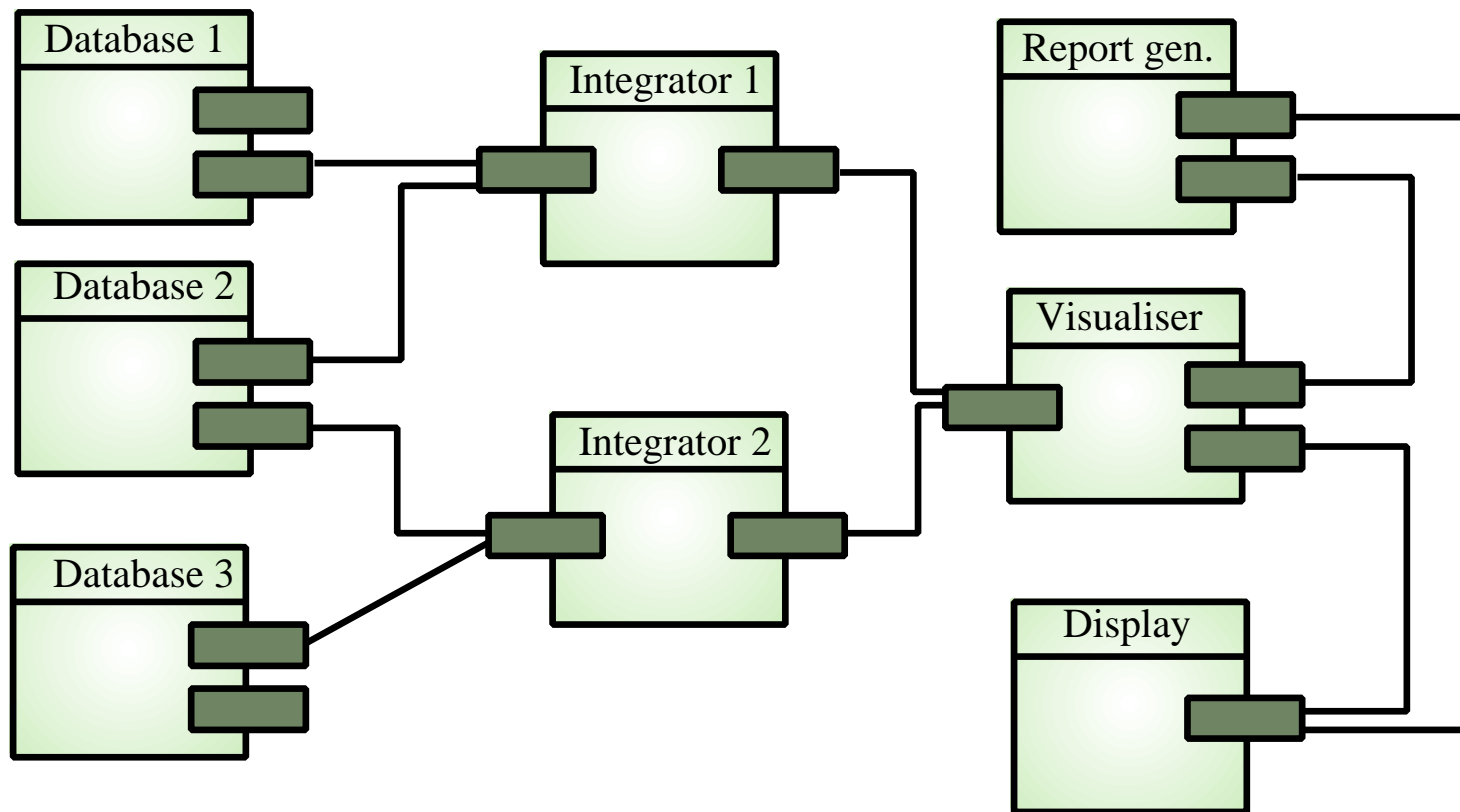
Advantages of distributed object architecture

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

Uses of distributed object architecture

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

A data mining system



Data mining system

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

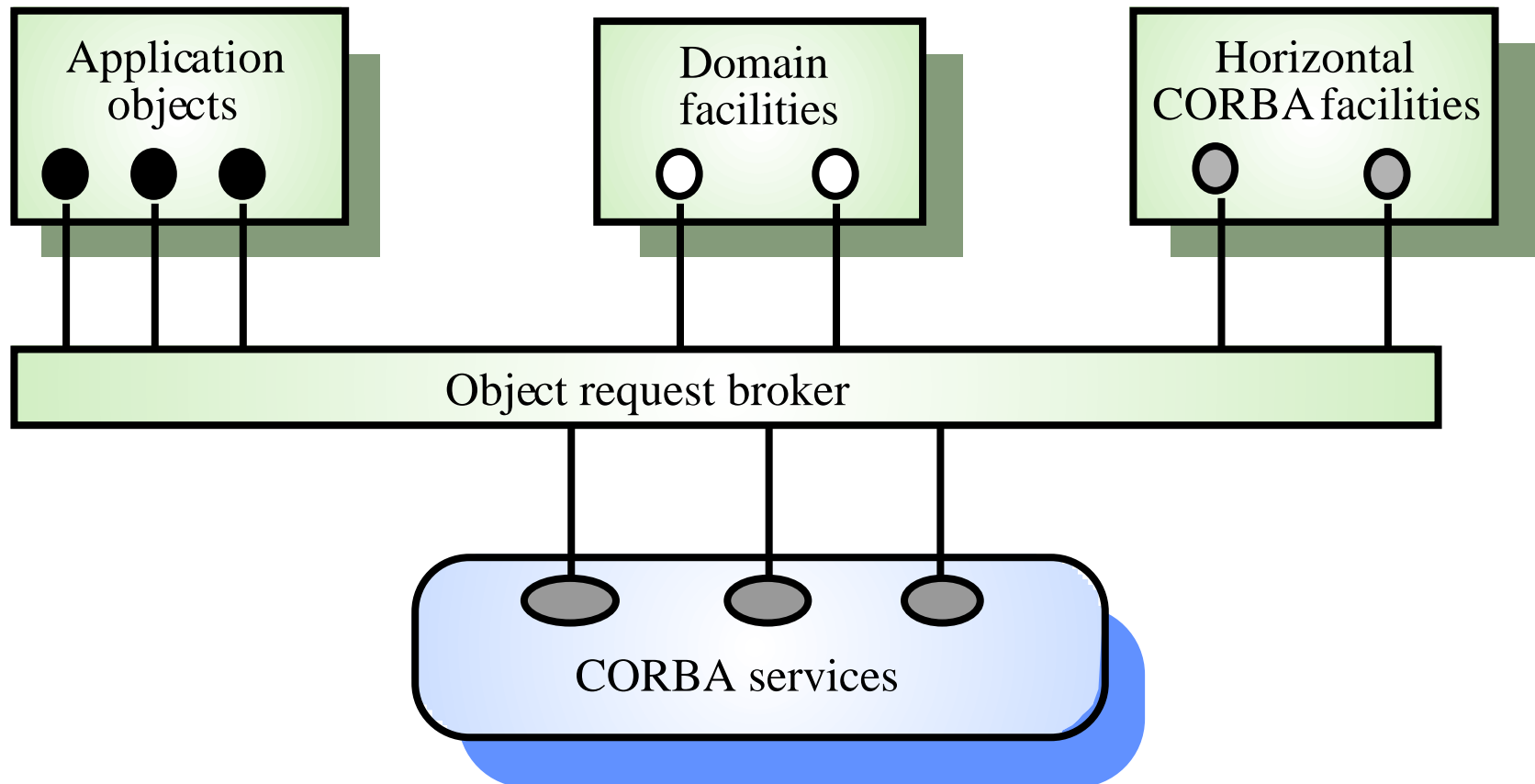
CORBA

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

Application structure

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

CORBA application structure



CORBA standards

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

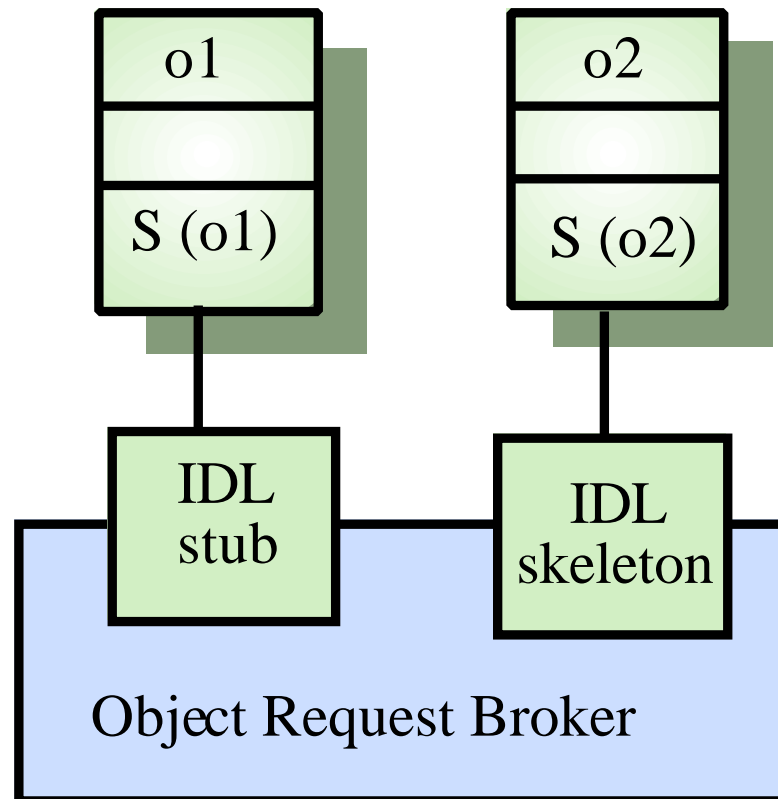
CORBA objects

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

Object request broker (ORB)

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

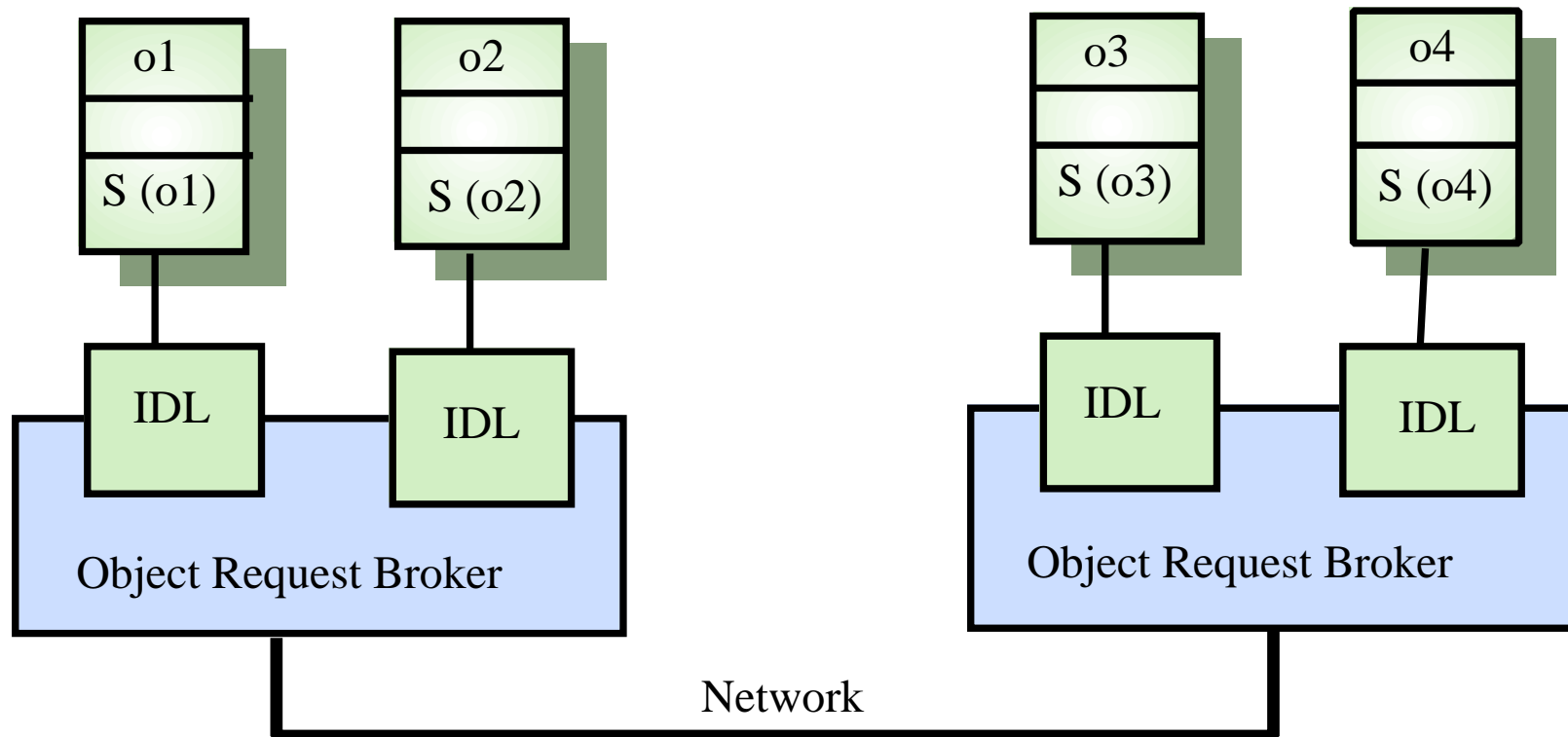
ORB-based object communications



Inter-ORB communications

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

Inter-ORB communications



CORBA services

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

Key points

- Almost all new large systems are distributed systems
- 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

Key points

- In a distributed object architecture, there is no distinction between clients and servers
- Distributed object systems require middleware to handle object communications
- The CORBA standards are a set of middleware standards that support distributed object architectures

Object-oriented Design

Designing systems using self-contained objects and object classes

Objectives

- To explain how a software design may be represented as a set of interacting objects that manage their own state and operations
- To describe the activities in the object-oriented design process
- To introduce various models that describe an object-oriented design
- To show how the UML may be used to represent these models

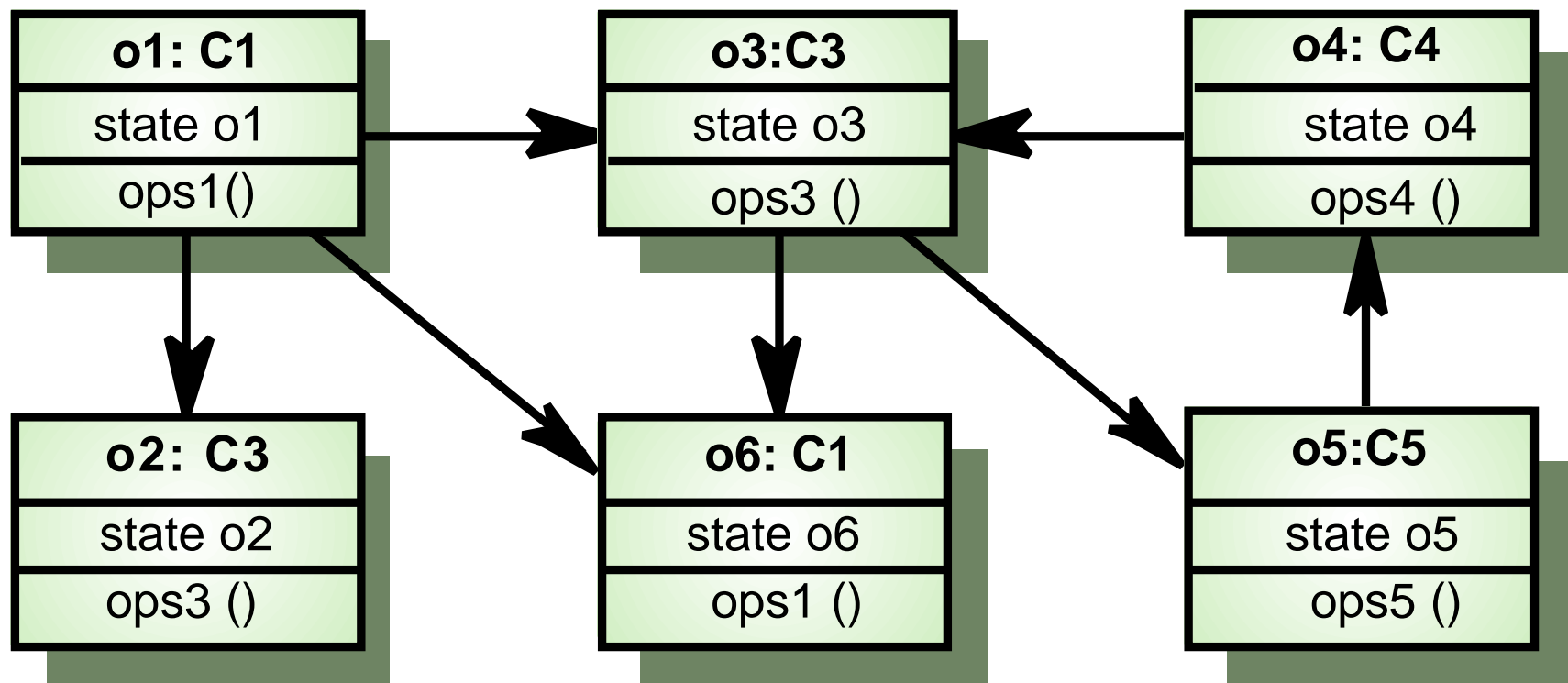
Topics covered

- Objects and object classes
- An object-oriented design process
- Design evolution

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

Interacting objects



Advantages of OOD

- Easier maintenance. Objects may be understood as stand-alone entities
- Objects are appropriate reusable components
- For some systems, there may be an obvious mapping from real world entities to system objects

Object-oriented development

- Object-oriented analysis, design and programming are related but distinct
- OOA is concerned with developing an object model of the application domain
- OOD is concerned with developing an object-oriented system model to implement requirements
- OOP is concerned with realising an OOD using an OO programming language such as Java or C++

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

Objects

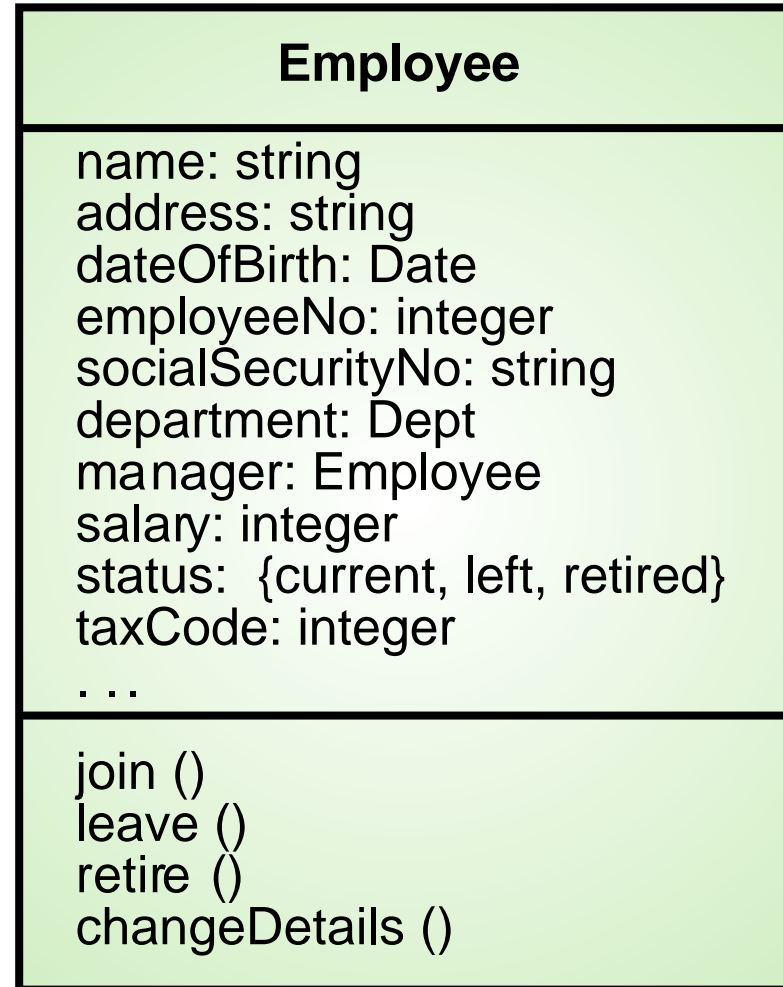
An **object** is an entity which has a state and a defined set of operations which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) which request these services when some computation is required.

Objects are created according to some **object class** definition. An object class definition serves as a template for objects. It includes declarations of all the attributes and services which should be associated with an object of that class.

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

- Conceptually, objects communicate by message passing.
- Messages
 - The name of the service requested by the calling object.
 - Copies of the information required to execute the service and the name of a holder for the result of the service.
- In practice, messages are often implemented by procedure calls
 - Name = procedure name.
 - Information = parameter list.

Message examples

```
// Call a method associated with a buffer  
// object that returns the next value  
// in the buffer
```

```
    v = circularBuffer.Get () ;
```

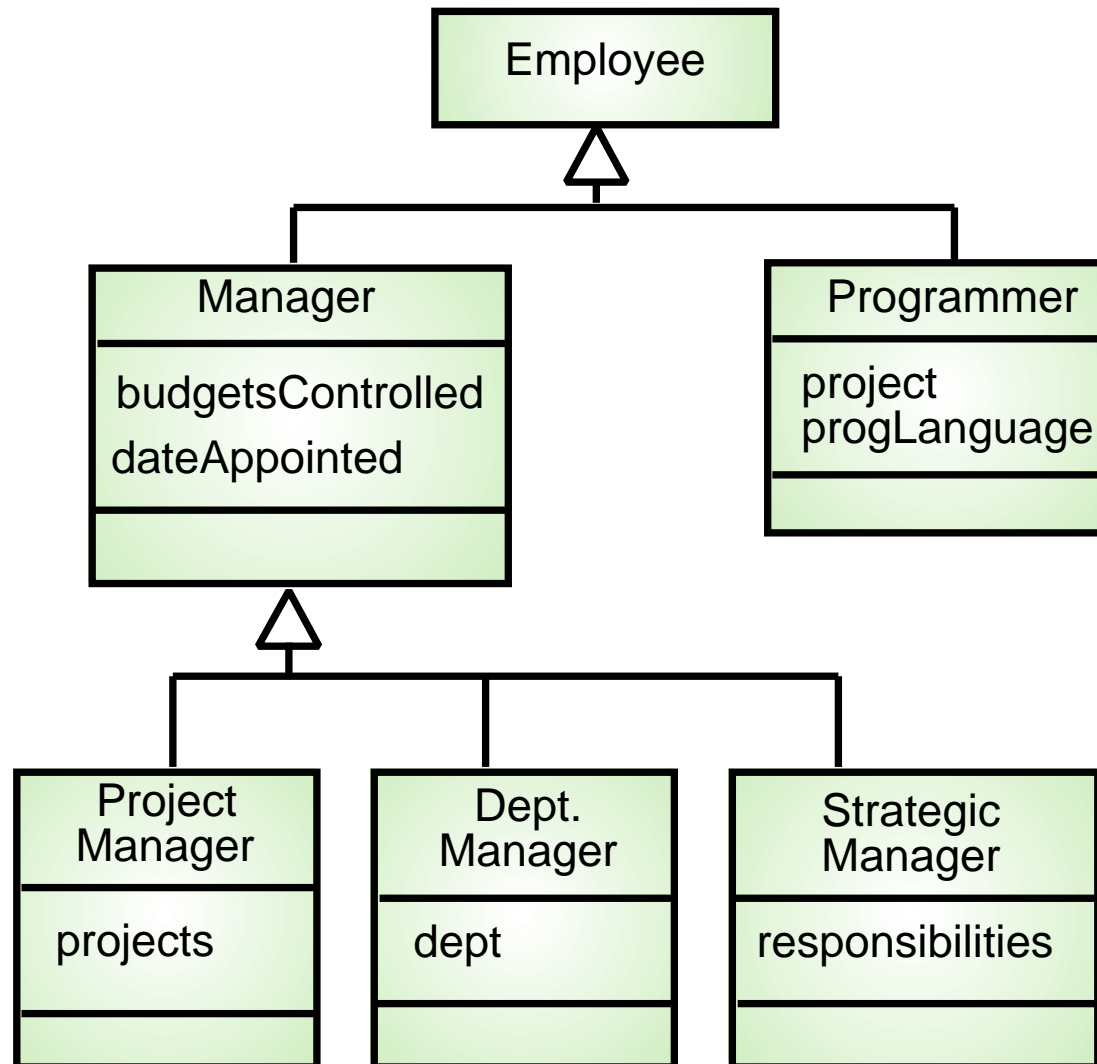
```
// Call the method associated with a  
// thermostat object that sets the  
// temperature to be maintained
```

```
    thermostat.setTemp (20) ;
```


Generalisation and inheritance

- Objects are members of classes which 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



Advantages of inheritance

- It is an abstraction mechanism which may be used to classify entities
- It is a reuse mechanism at both the design and the programming level
- The inheritance graph is a source of organisational knowledge about domains and systems

Problems with inheritance

- Object classes are not self-contained. they cannot be understood without reference to their super-classes
- Designers have a tendency to reuse the inheritance graph created during analysis. Can lead to significant inefficiency
- The inheritance graphs of analysis, design and implementation have different functions and should be separately maintained

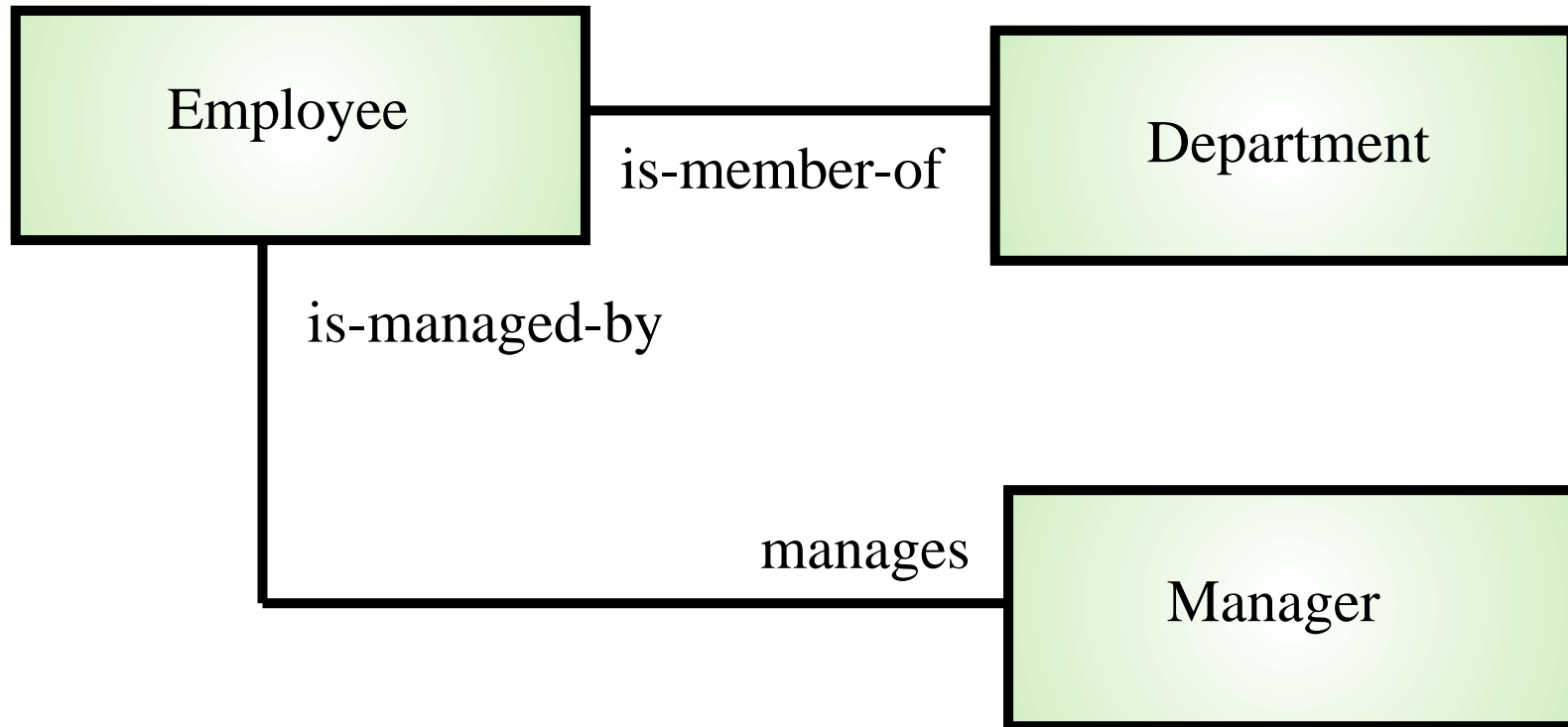
Inheritance and OOD

- There are differing views as to whether inheritance is fundamental to OOD.
 - View 1. Identifying the inheritance hierarchy or network is a fundamental part of object-oriented design. Obviously this can only be implemented using an OOPL.
 - View 2. Inheritance is a useful implementation concept which allows reuse of attribute and operation definitions. Identifying an inheritance hierarchy at the design stage places unnecessary restrictions on the implementation
- Inheritance introduces complexity and this is undesirable, especially in critical systems

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

Active transponder object

- Active objects may have their attributes modified by operations but may also update them autonomously using internal operations
- Transponder object broadcasts an aircraft's position. The position may be updated using a satellite positioning system. The object periodically update the position by triangulation from satellites

An active transponder object

```
class Transponder extends Thread {  
  
    Position currentPosition ;  
    Coords c1, c2 ;  
    Satellite sat1, sat2 ;  
    Navigator theNavigator ;  
  
    public Position givePosition ()  
    {  
        return currentPosition ;  
    }  
  
    public void run ()  
    {  
        while (true)  
        {  
            c1 = sat1.position () ;  
            c2 = sat2.position () ;  
            currentPosition = theNavigator.compute (c1, c2) ;  
        }  
    }  
  
} //Transponder
```

Java threads

- Threads in Java are a simple construct for implementing concurrent objects
- Threads must include a method called `run()` and this is started up by the Java run-time system
- Active objects typically include an infinite loop so that they are always carrying out the computation

An object-oriented design process

- Define the context and modes of use of the system
- Design the system architecture
- Identify the principal system objects
- Develop design models
- Specify object interfaces

Weather system description

A weather data collection 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.

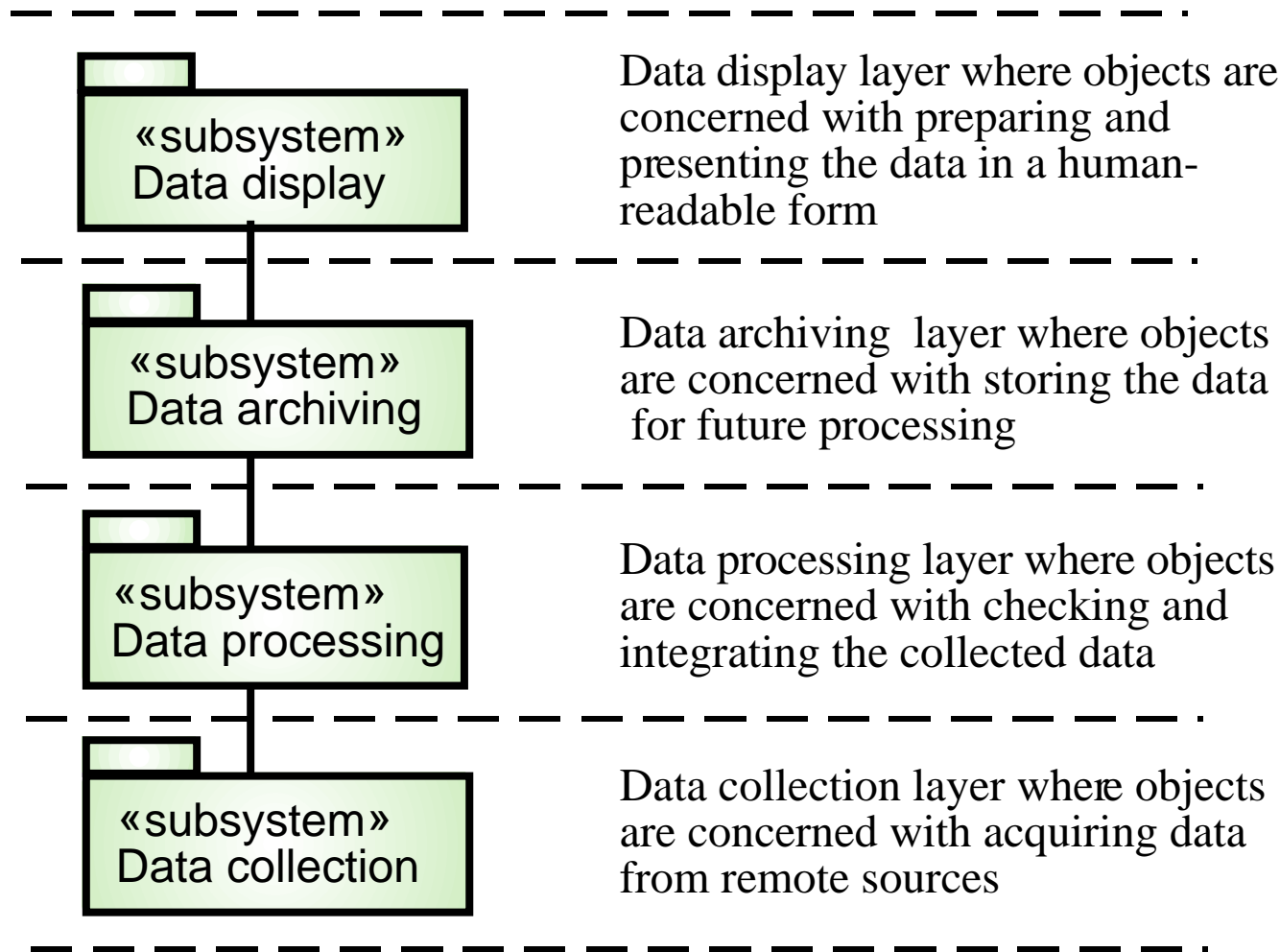
The area computer validates the collected data and integrates it with the data from different sources. The integrated data is archived and, using data from this archive and a digitised map database a set of local weather maps is created. Maps may be printed for distribution on a special-purpose map printer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

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 every five minutes.

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.

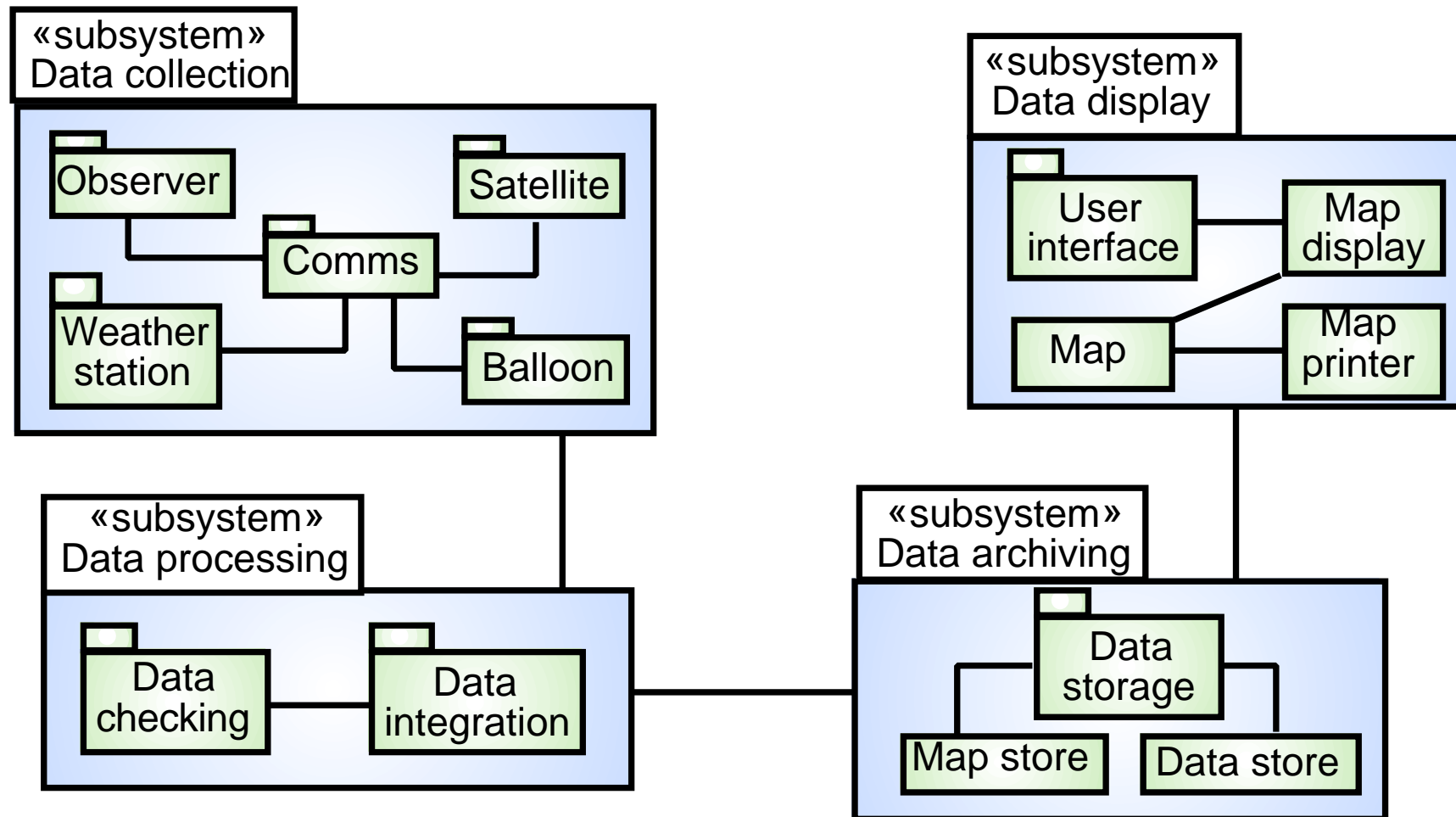
Layered architecture



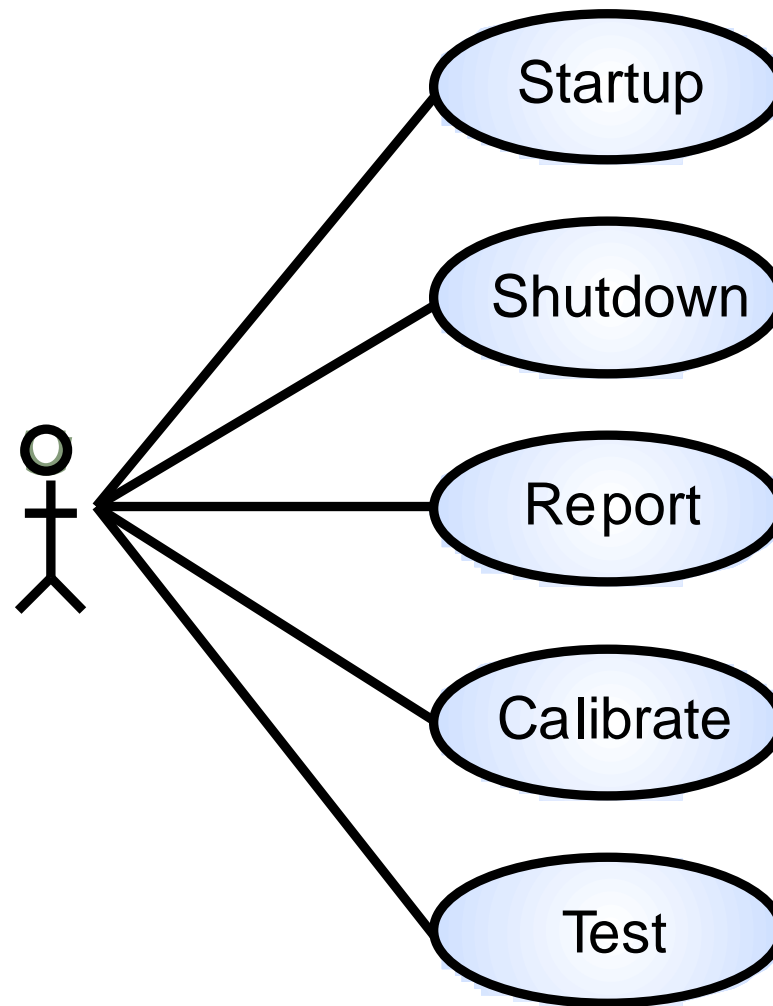
System context and models of use

- 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

Subsystems in the weather mapping system



Use-cases for the weather station



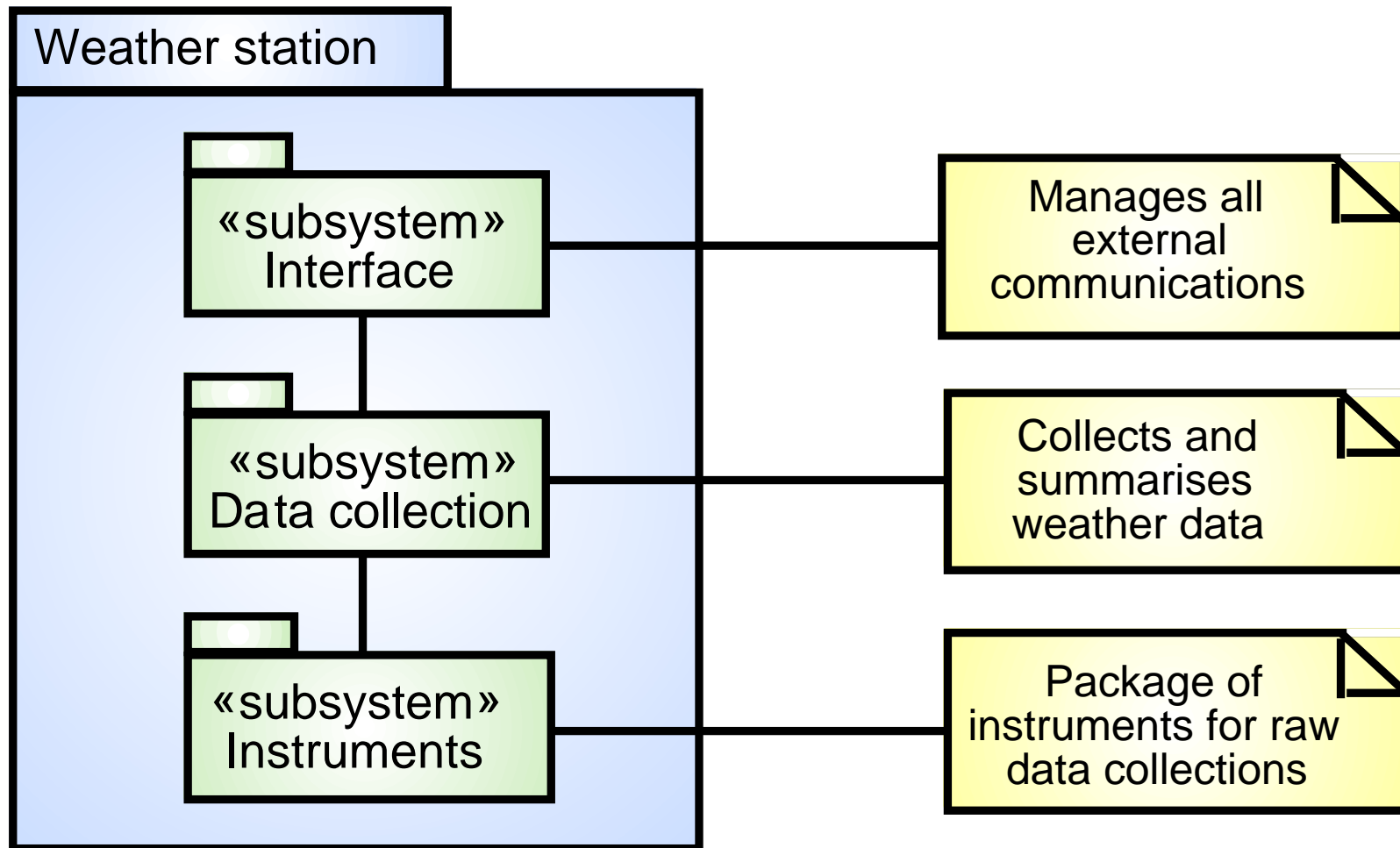
Use-case description

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

Architectural design

- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture
- Layered architecture is appropriate for the weather station
 - Interface layer for handling communications
 - Data collection layer for managing instruments
 - Instruments layer for collecting data
- There should 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 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 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.

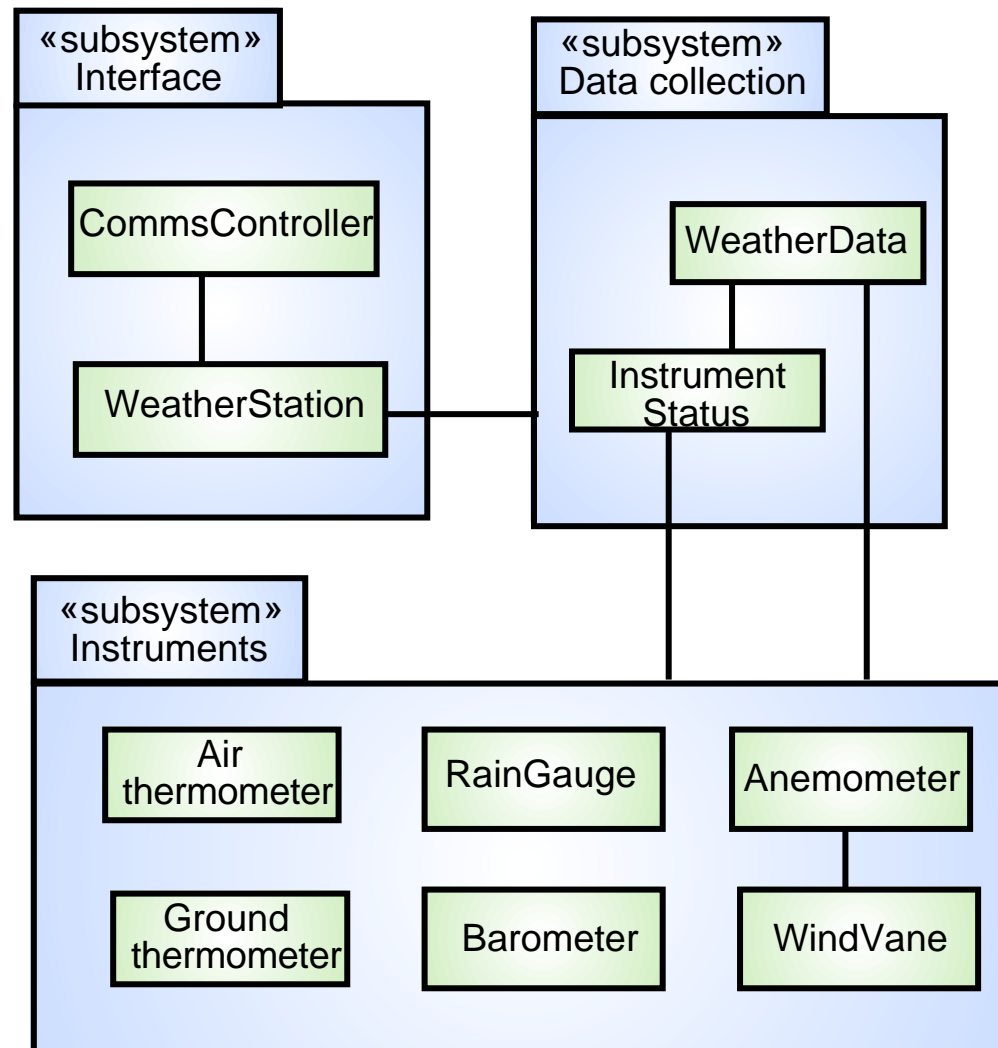
Examples of design models

- 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.

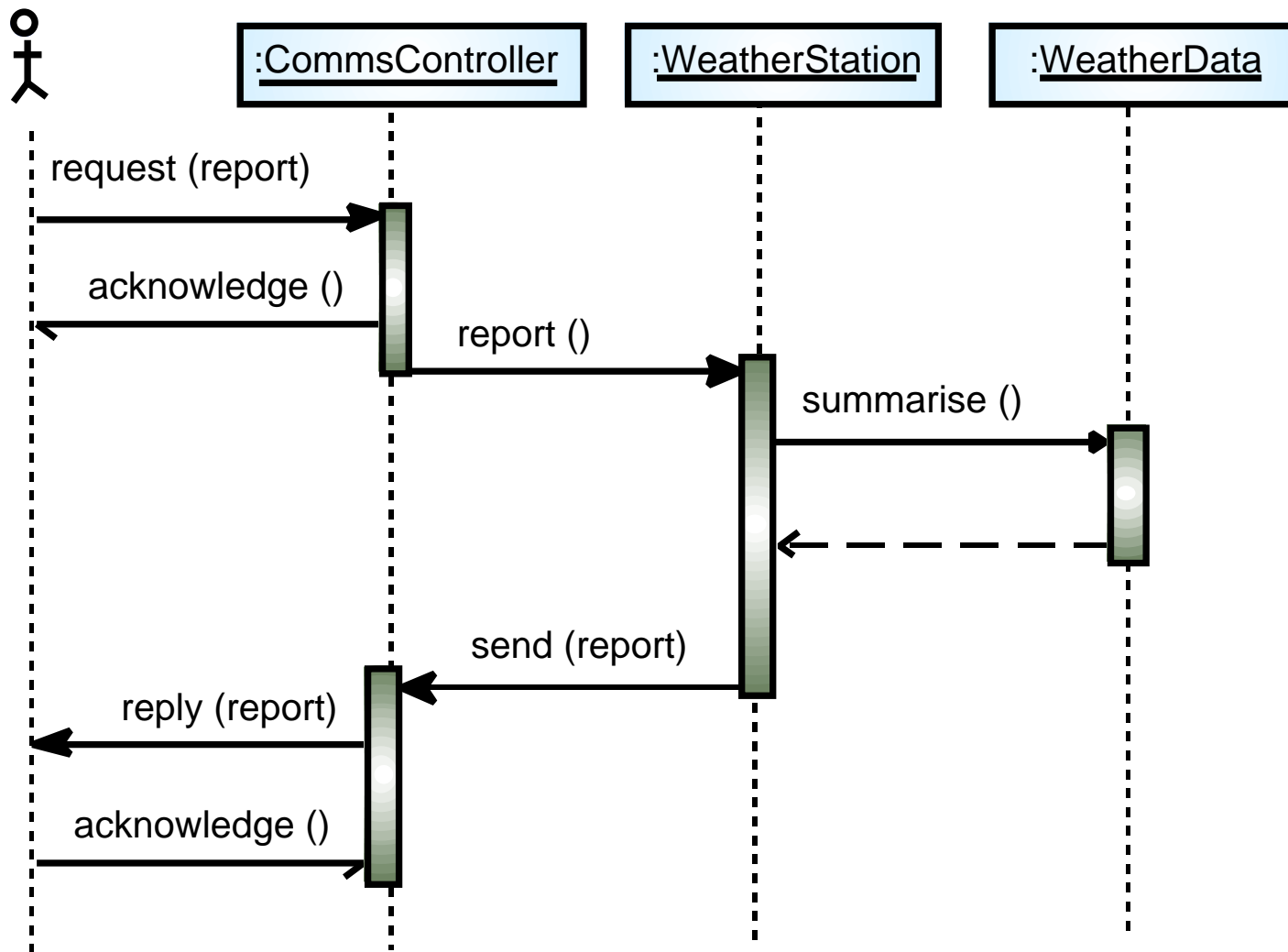
Weather station subsystems



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

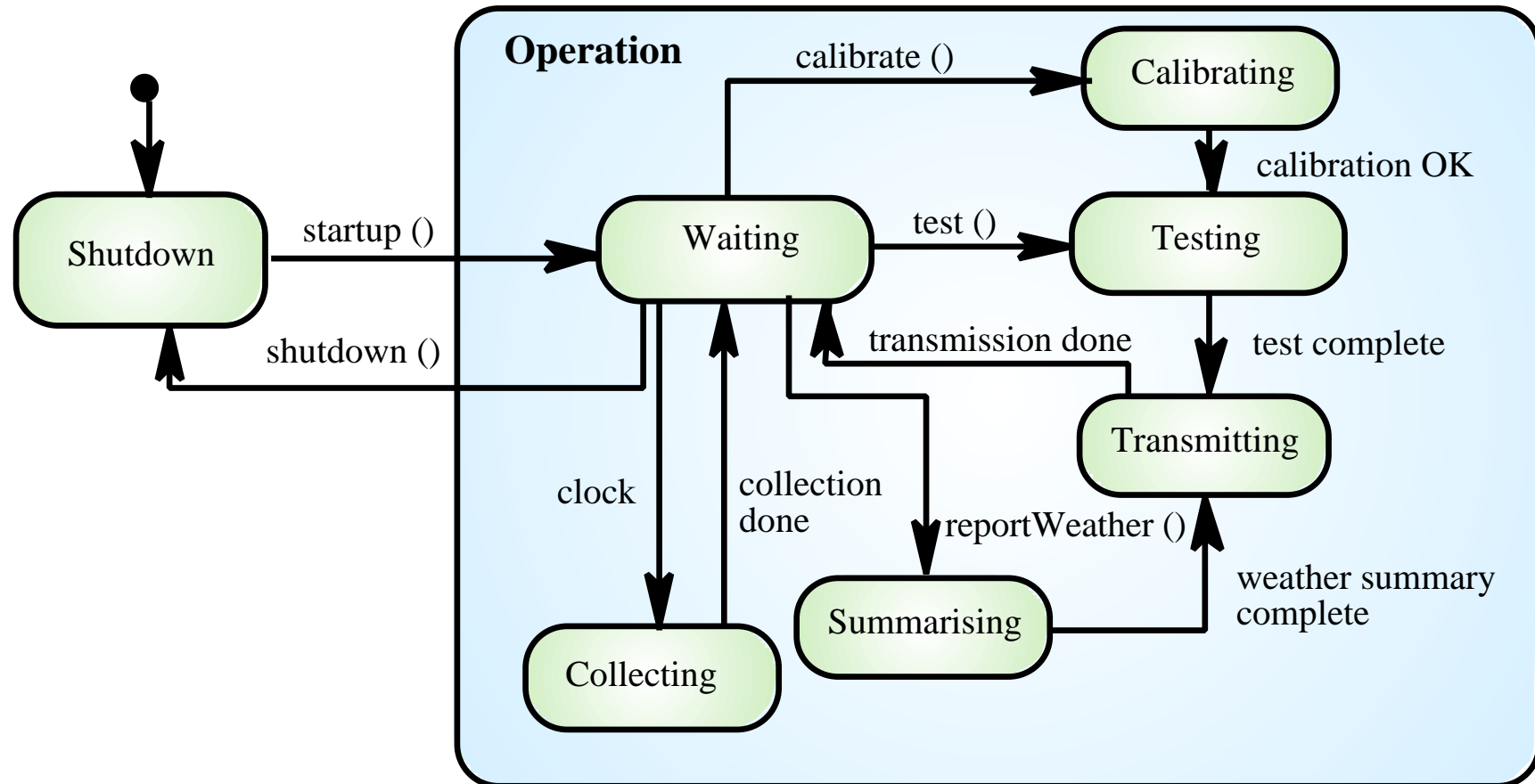
Data collection sequence



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
```

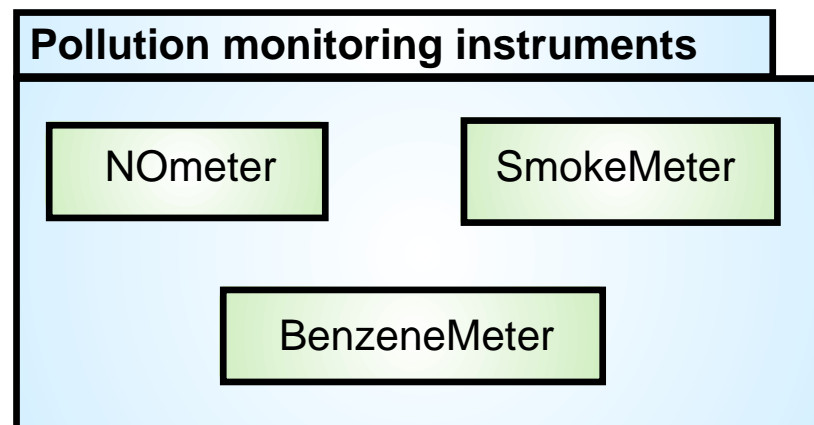
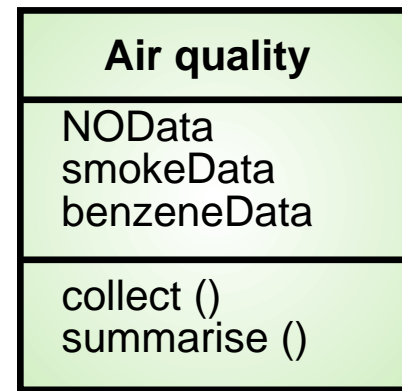
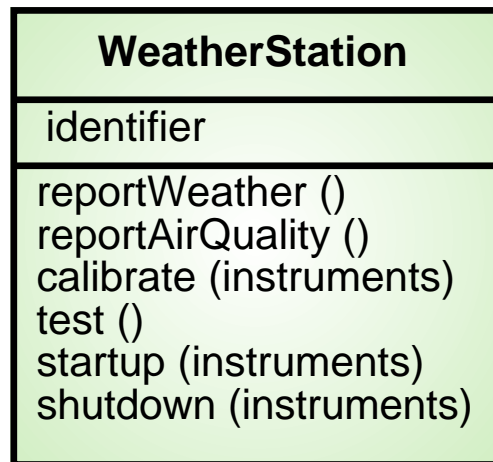
Design evolution

- 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



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 simplifies system evolution

Real-time Software Design

- Designing embedded software systems whose behaviour is subject to timing constraints

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 executive
- To introduce generic architectures for monitoring and control and data acquisition systems

Topics covered

- Systems design
- Real-time executives
- 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

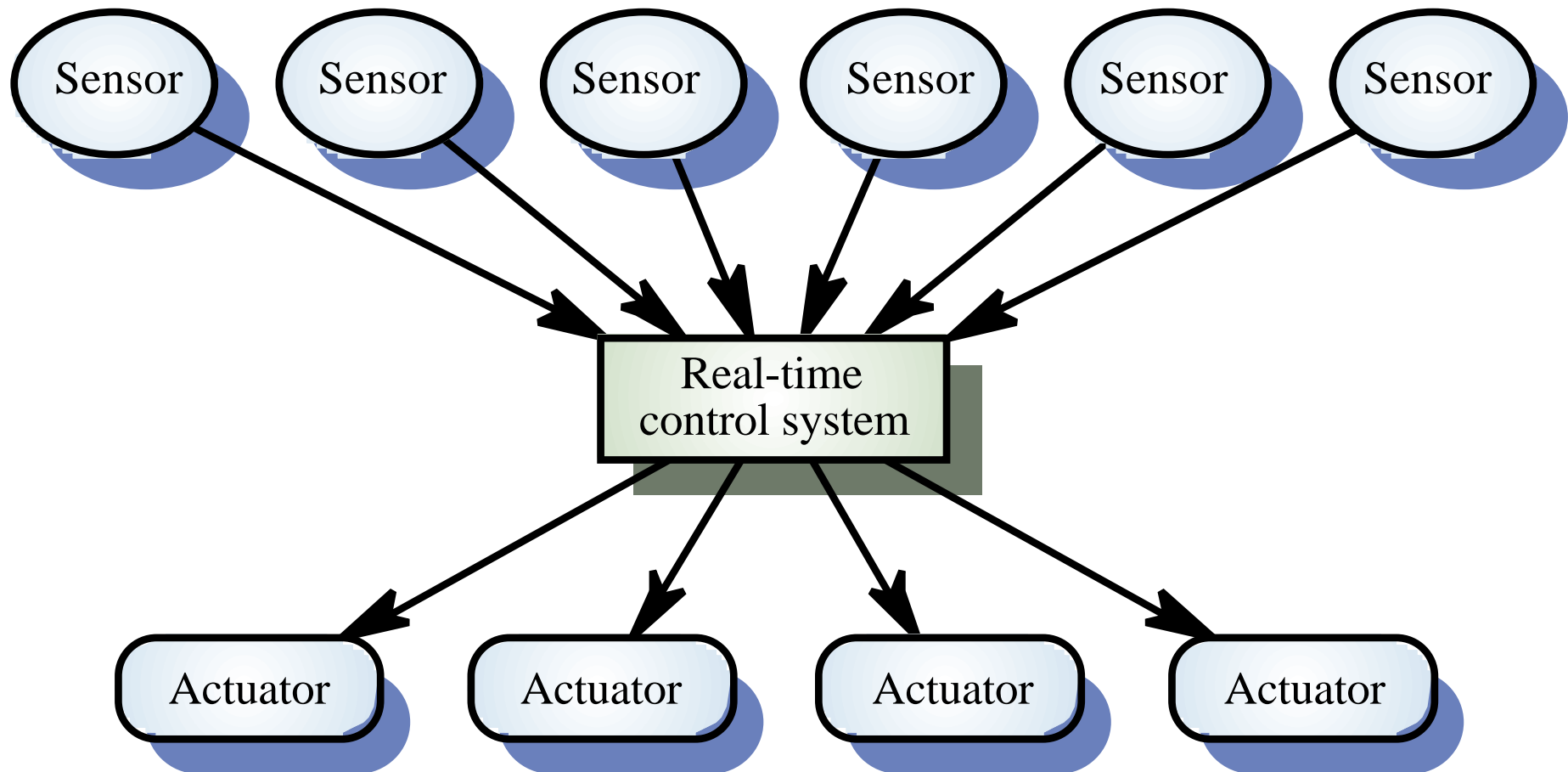
Stimulus/Response Systems

- 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 usually designed as cooperating processes with a real-time executive controlling these processes

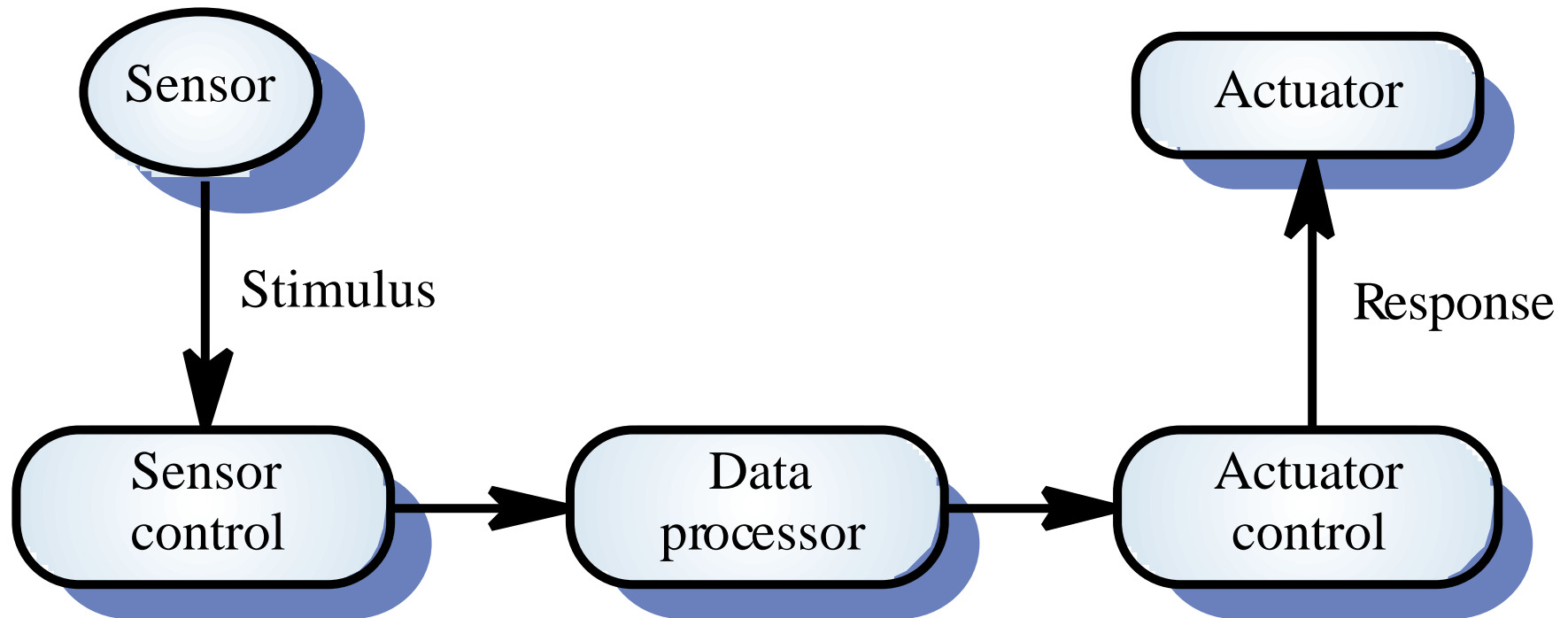
A real-time system model



System elements

- Sensors 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
 - Generates control signals for the actuator

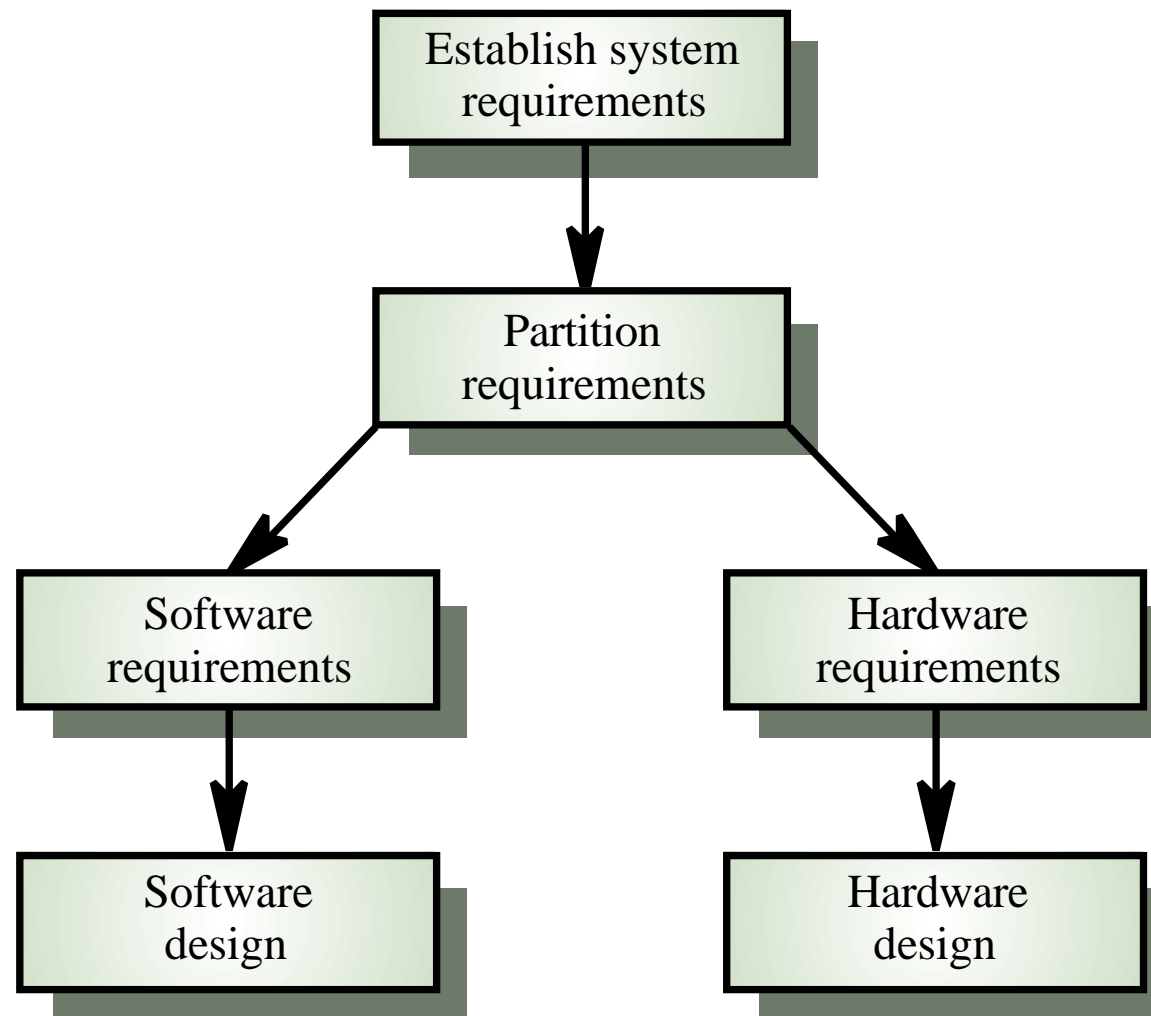
Sensor/actuator processes



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

Hardware and software design



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 executive or operating system

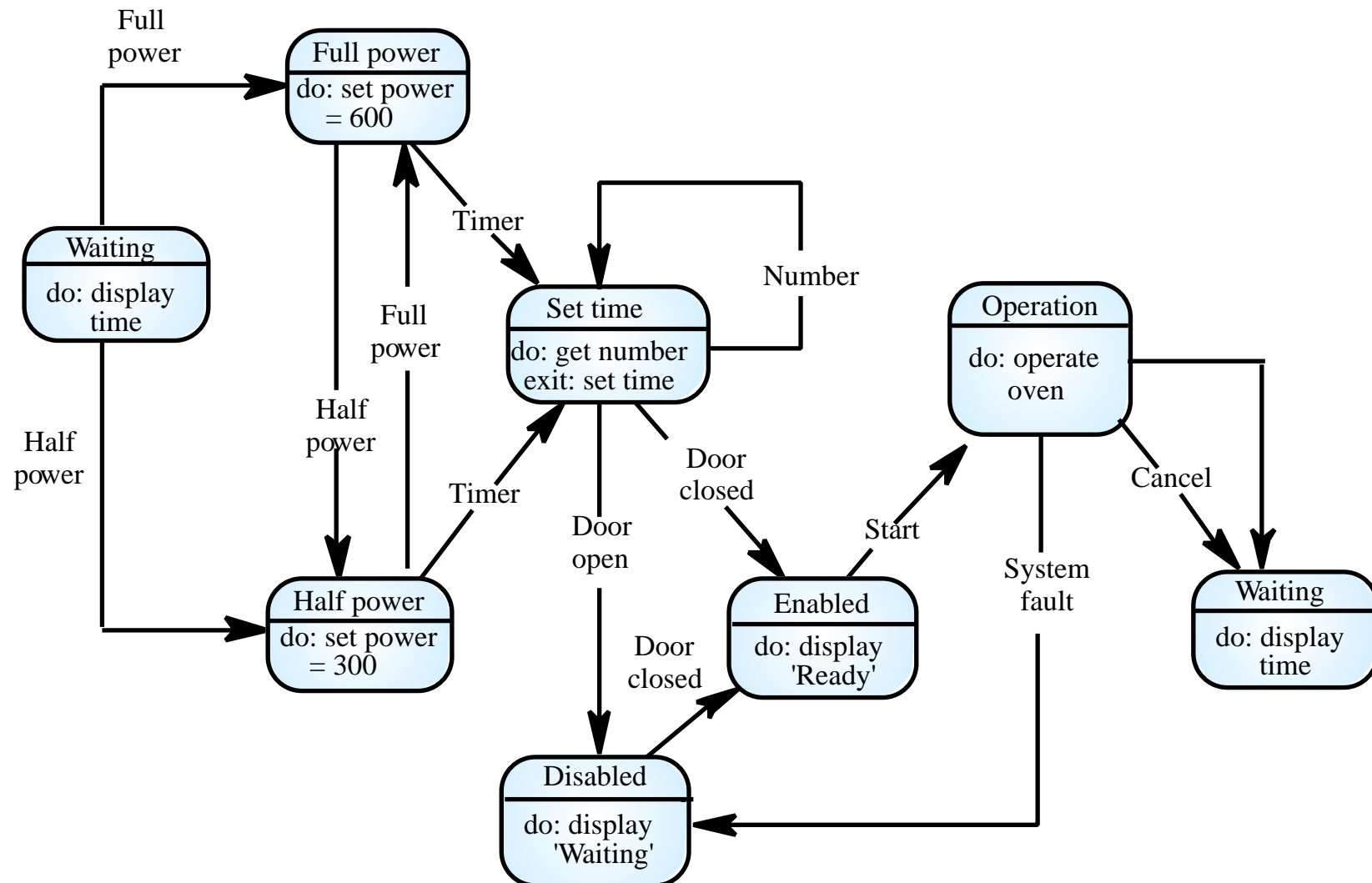
Timing constraints

- 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

State machine 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 also Chapter 7.

Microwave oven state machine



Real-time programming

- Hard-real time systems may have to be 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
- Ada as a language designed to support real-time systems design so includes a general purpose concurrency mechanism

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 or programming where precise control of timing is required
 - Not possible to specify thread execution time
 - Uncontrollable garbage collection
 - Not possible to discover queue sizes for shared resources
 - Variable virtual machine implementation
 - Not possible to do space or timing analysis

Real-time executives

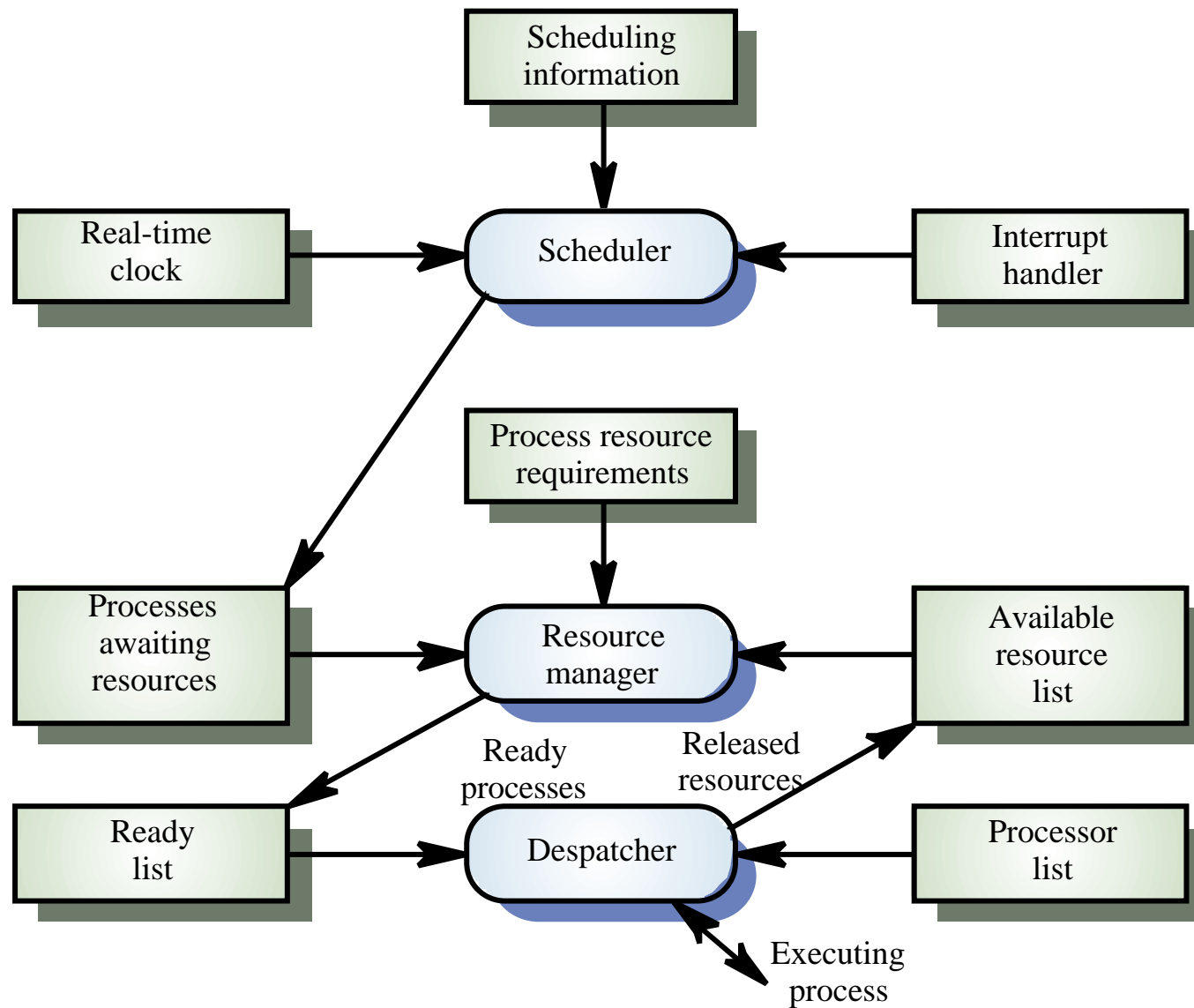
- Real-time executives are specialised operating systems which manage the processes in the RTS
- Responsible for process management and resource (processor and memory) allocation
- May be based on a standard RTE kernel which is used unchanged or modified for a particular application
- Does not include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

Executive 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.
- Despatcher
 - 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 executive 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

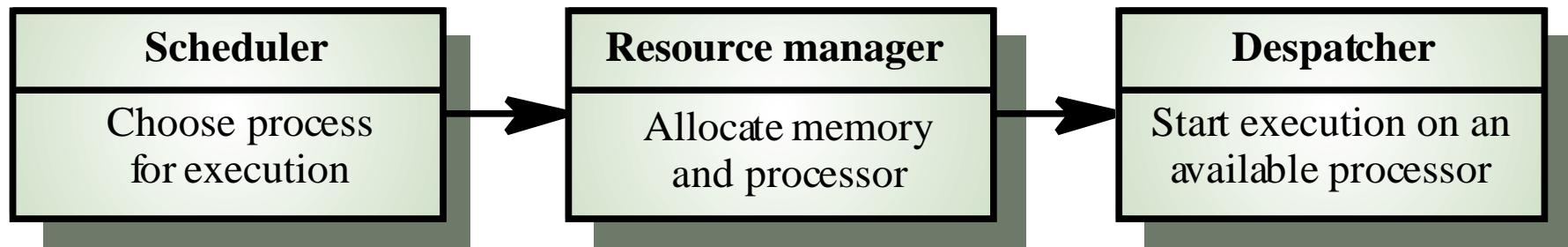
Periodic process servicing

- 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 executive uses the real-time clock to determine when to execute a process
- 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

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

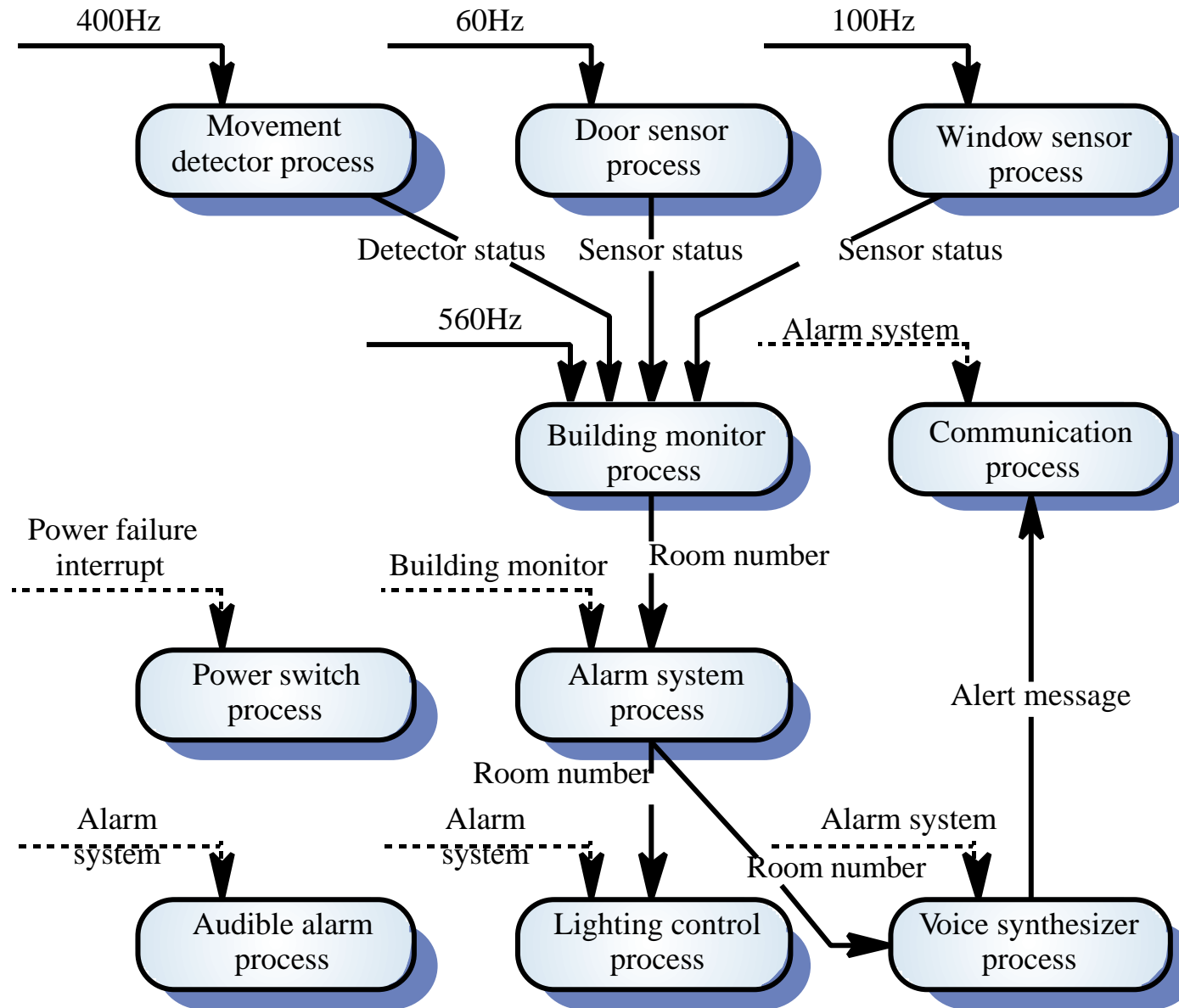
- Identify stimuli and associated responses
- Define the timing constraints associated with each stimulus and response
- Allocate system functions to concurrent processes
- Design algorithms for stimulus processing and response generation
- Design a scheduling system which ensures that processes will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



Process architecture

// See <http://www.software-engin.com/> for links to the complete Java code for this
// example

```
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 1


```

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 ;
        }
    }
    lights.shutdown () ; siren.shutdown () ; synthesizer.shutdown () ;
    windows.shutdown () ; doors.shutdown () ; movements.shutdown () ;

} // run
} //BuildingMonitor

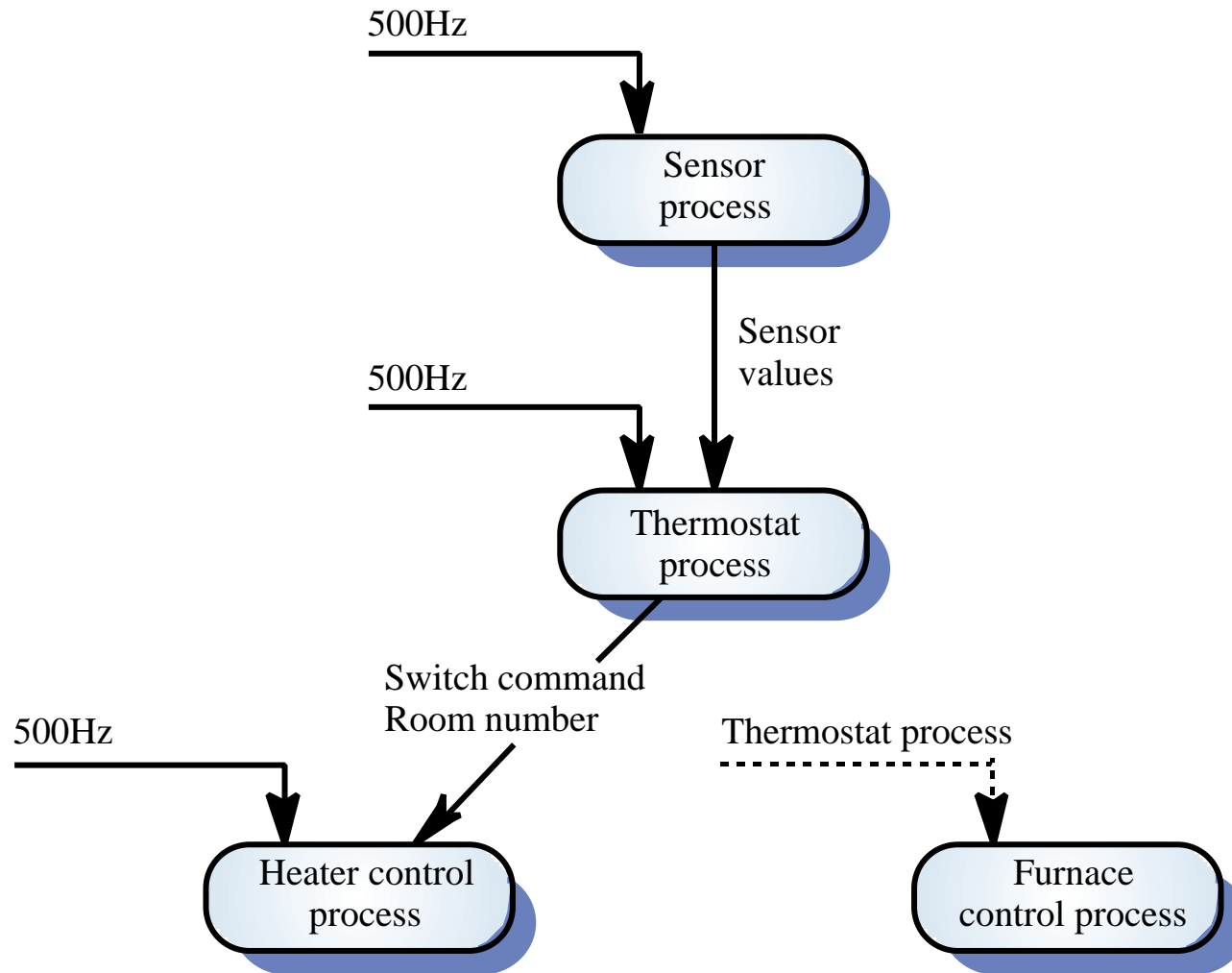
```

Building_monitor process 2

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 which monitors temperature and switches heaters on and off

A temperature control system



Data acquisition systems

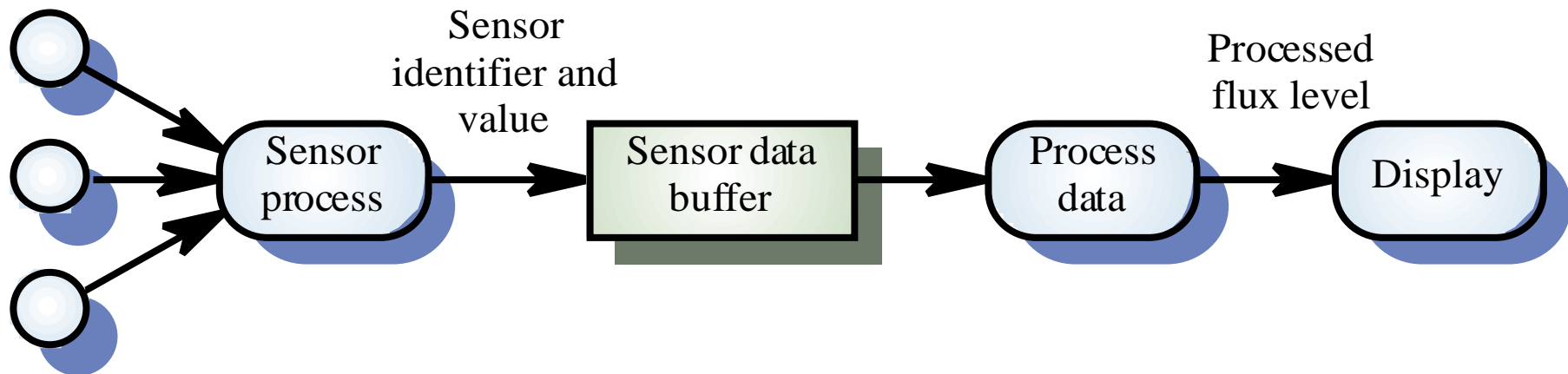
- Collect data from sensors for subsequent processing and analysis.
- Data collection processes and processing processes may have different periods and deadlines.
- Data collection may be faster than processing e.g. collecting information about an explosion.
- Circular or ring buffers are a mechanism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

Reactor data collection

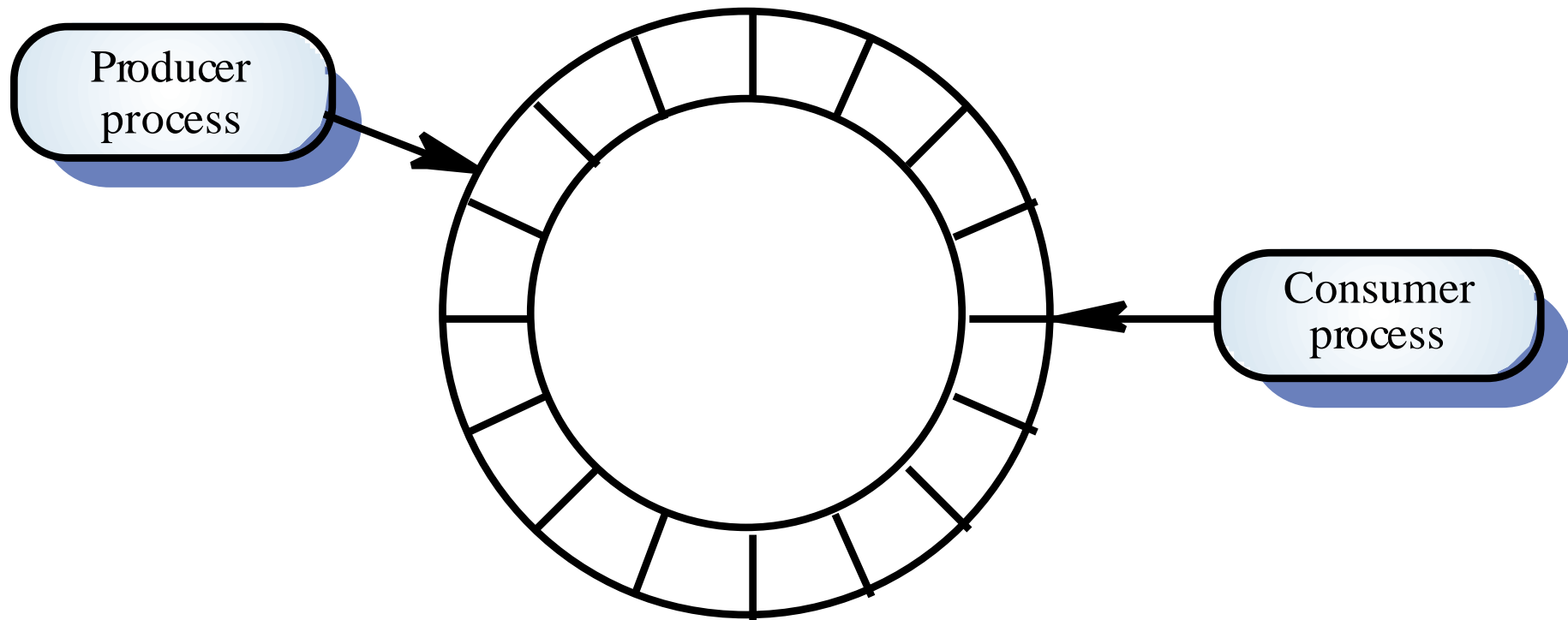
- A system collects data from a set of sensors monitoring the neutron flux from a nuclear reactor.
- Flux data is placed in a ring buffer for later processing.
- The ring buffer is itself implemented as a concurrent process so that the collection and processing processes may be synchronized.

Reactor flux monitoring

Sensors (each data flow is a sensor value)



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.


```

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

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

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

```

Java implementation of a ring buffer 1

```
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
```

Java implementation of a ring buffer 2

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 executives 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
- Java has facilities for supporting concurrency but is not suitable for the development of time-critical systems

Design with Reuse

- Building software from reusable components.

Objectives

- To explain the benefits of software reuse and some reuse problems
- To describe different types of reusable component and processes for reuse
- To introduce application families as a route to reuse
- To describe design patterns as high-level abstractions that promote reuse

Topics covered

- Component-based development
- Application families
- Design patterns

Software reuse

- In most engineering disciplines, systems are designed by composing existing components that have been used in other systems
- Software engineering has been more focused on original development but it is now recognised that to achieve better software, more quickly and at lower cost, we need to adopt a design process that is based on *systematic reuse*

Reuse-based software engineering

- Application system reuse
 - The whole of an application system may be reused either by incorporating it without change into other systems (COTS reuse) or by developing application families
- Component reuse
 - Components of an application from sub-systems to single objects may be reused
- Function reuse
 - Software components that implement a single well-defined function may be reused

Reuse practice

- Application system reuse
 - Widely practised as software systems are implemented as application families. COTS reuse is becoming increasingly common
- Component reuse
 - Now seen as the key to effective and widespread reuse through component-based software engineering. However, it is still relatively immature
- Function reuse
 - Common in some application domains (e.g. engineering) where domain-specific libraries of reusable functions have been established

Benefits of reuse

- Increased reliability
 - Components exercised in working systems
- Reduced process risk
 - Less uncertainty in development costs
- Effective use of specialists
 - Reuse components instead of people
- Standards compliance
 - Embed standards in reusable components
- Accelerated development
 - Avoid original development and hence speed-up production

Requirements for design with reuse

- It must be possible to find appropriate reusable components
- The reuser of the component must be confident that the components will be reliable and will behave as specified
- The components must be documented so that they can be understood and, where appropriate, modified

Reuse problems

- Increased maintenance costs
- Lack of tool support
- Not-invented-here syndrome
- Maintaining a component library
- Finding and adapting reusable components

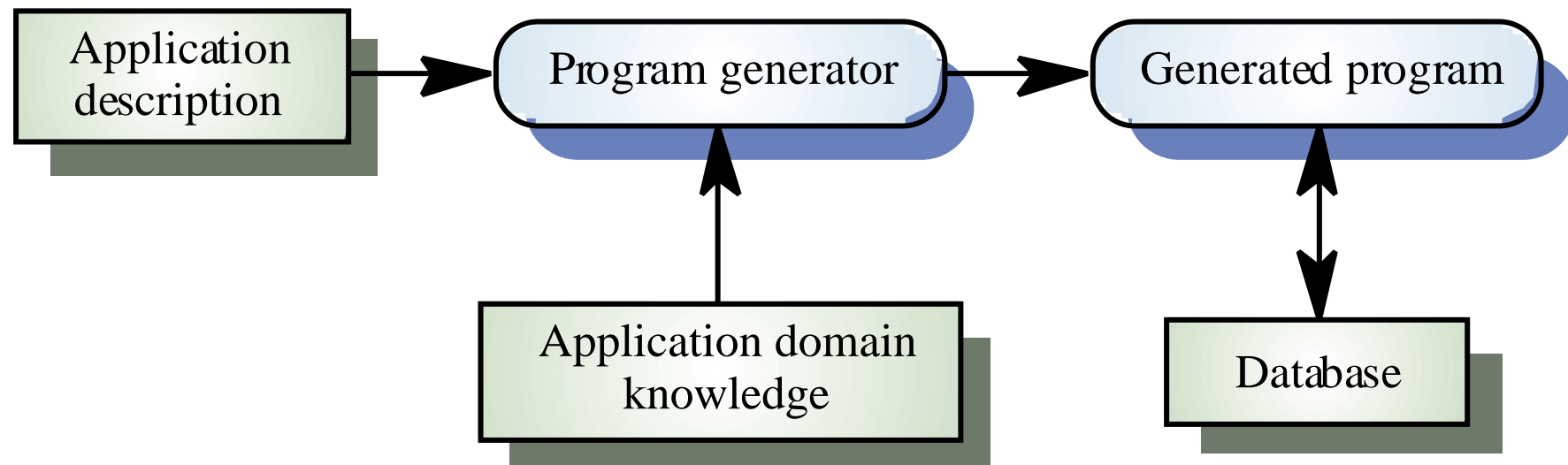
Generator-based reuse

- Program generators involve the reuse of standard patterns and algorithms
- These are embedded in the generator and parameterised by user commands. A program is then automatically generated
- Generator-based reuse is possible when domain abstractions and their mapping to executable code can be identified
- A domain specific language is used to compose and control these abstractions

Types of program generator

- Types of program generator
 - Application generators for business data processing
 - Parser and lexical analyser generators for language processing
 - Code generators in CASE tools
- Generator-based reuse is very cost-effective but its applicability is limited to a relatively small number of application domains
- It is easier for end-users to develop programs using generators compared to other component-based approaches to reuse

Reuse through program generation



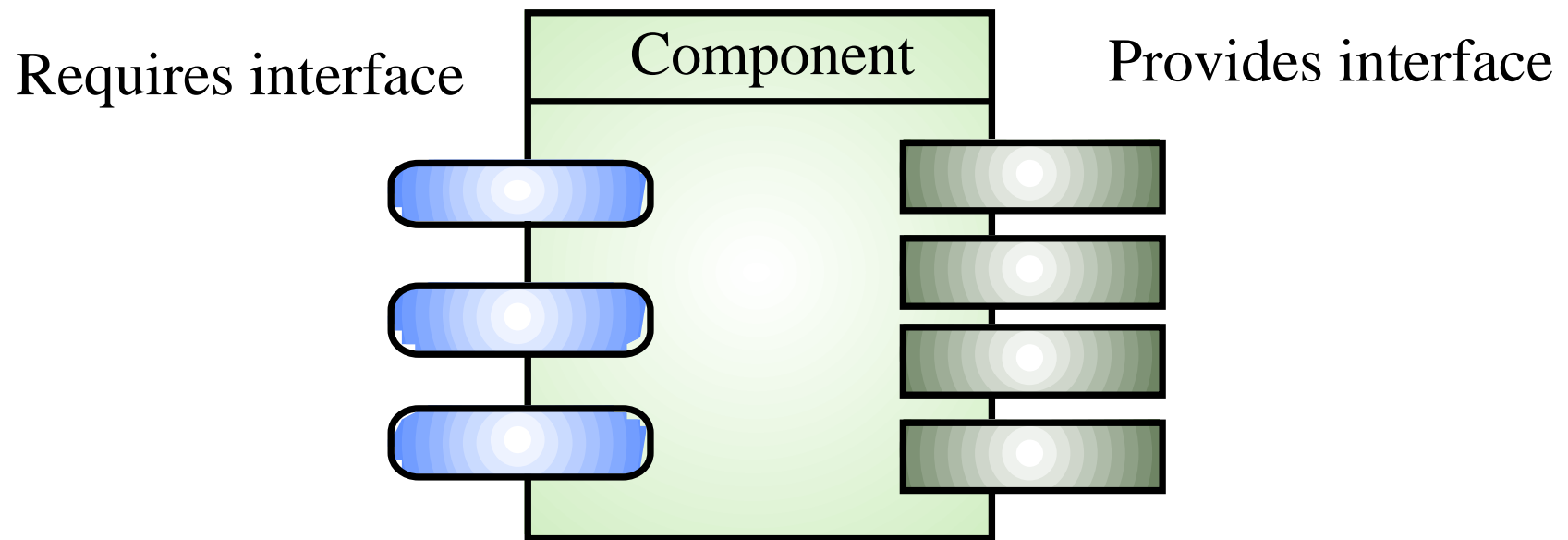
Component-based development

- Component-based software engineering (CBSE) is an approach to software development that relies on reuse
- It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific
- Components are more abstract than object classes and can be considered to be stand-alone service providers

Components

- Components provide a service without regard to where the component is executing or its programming language
 - A component is an independent executable entity that can be made up of one or more executable objects
 - The component interface is published and all interactions are through the published interface
- Components can range in size from simple functions to entire application systems

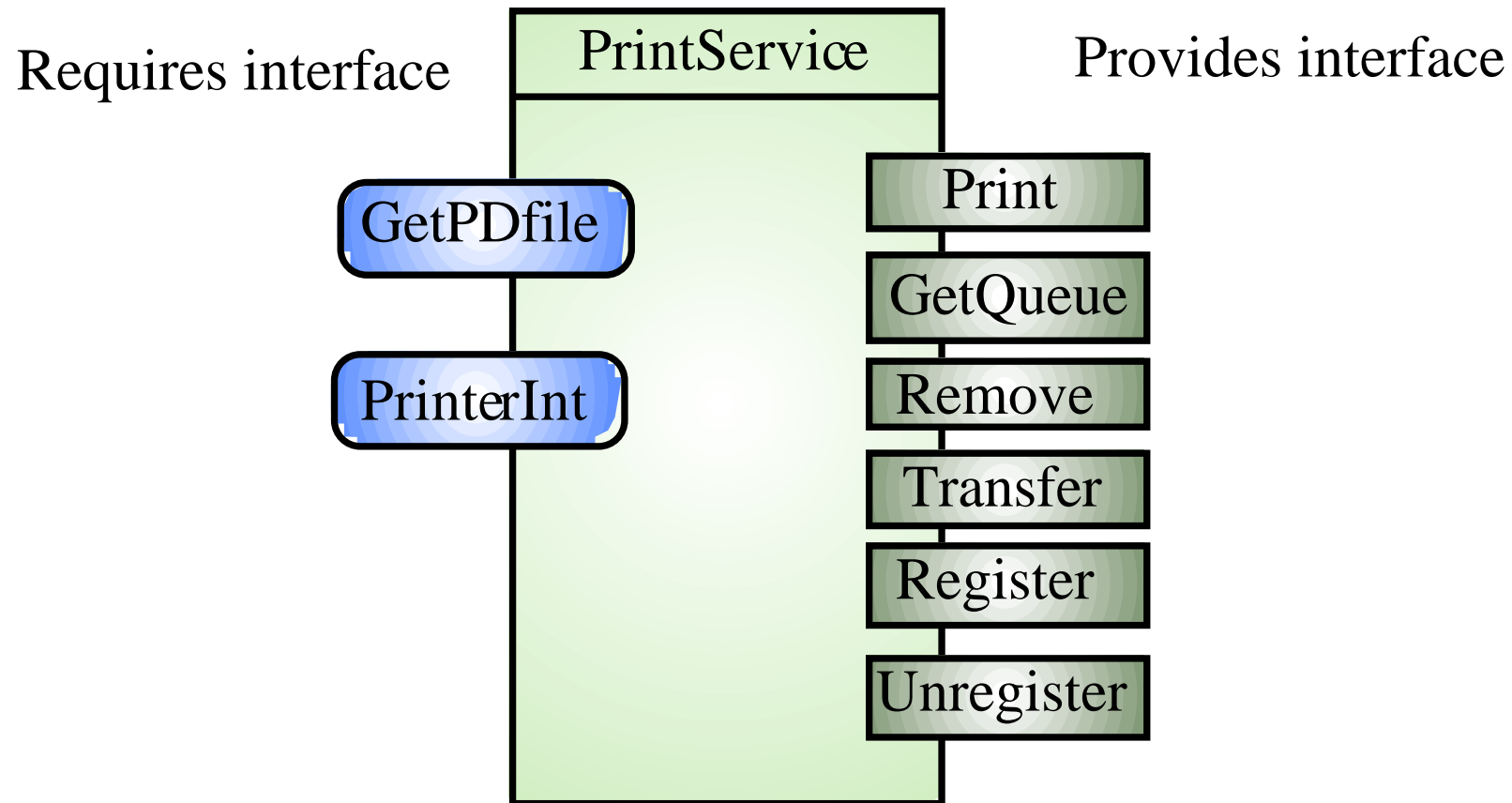
Component interfaces



Component interfaces

- Provides interface
 - Defines the services that are provided by the component to other components
- Requires interface
 - Defines the services that specifies what services must be made available for the component to execute as specified

Printing services component



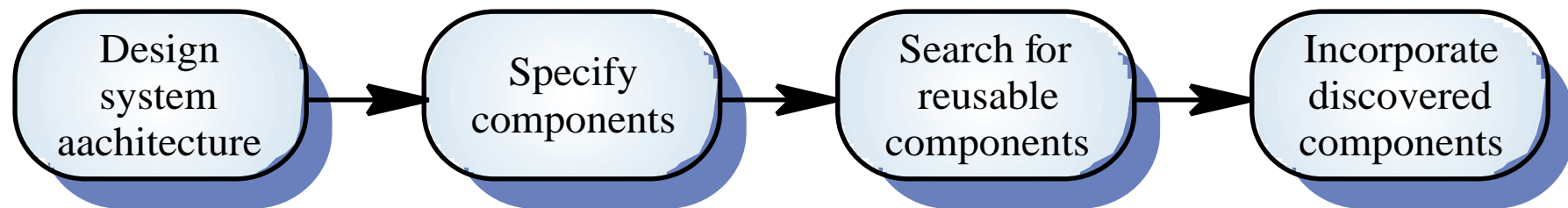
Component abstractions

- *Functional abstraction*
 - The component implements a single function such as a mathematical function
- *Casual groupings*
 - The component is a collection of loosely related entities that might be data declarations, functions, etc.
- *Data abstractions*
 - The component represents a data abstraction or class in an object-oriented language
- *Cluster abstractions*
 - The component is a group of related classes that work together
- *System abstraction*
 - The component is an entire self-contained system

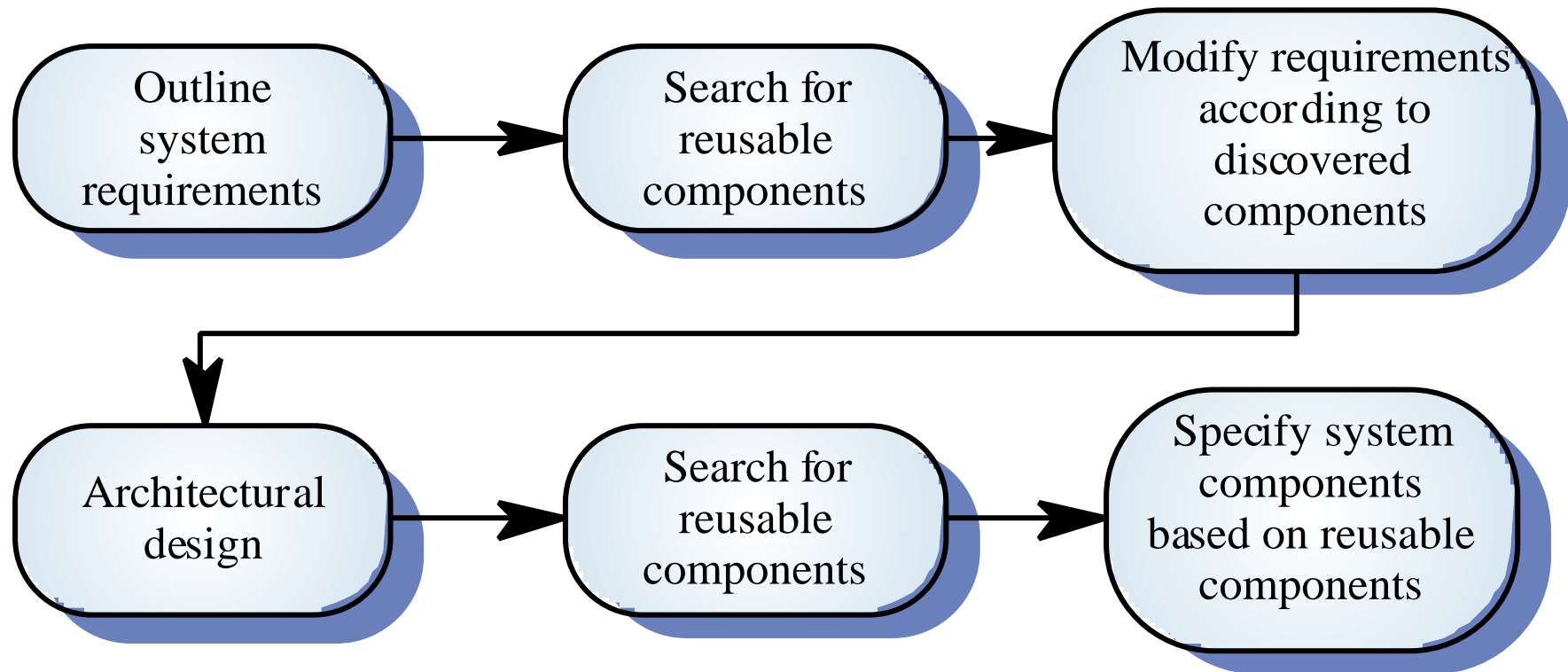
CBSE processes

- Component-based development can be integrated into a standard software process by incorporating a reuse activity in the process
- However, in reuse-driven development, the system requirements are modified to reflect the components that are available
- CBSE usually involves a prototyping or an incremental development process with components being ‘glued together’ using a scripting language

An opportunistic reuse process



Development with reuse



CBSE problems

- Component incompatibilities may mean that cost and schedule savings are less than expected
- Finding and understanding components
- Managing evolution as requirements change in situations where it may be impossible to change the system components

Application frameworks

- Frameworks are a sub-system design made up of a collection of abstract and concrete classes and the interfaces between them
- The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework
- Frameworks are moderately large entities that can be reused

Framework classes

- System infrastructure frameworks
 - Support the development of system infrastructures such as communications, user interfaces and compilers
- Middleware integration frameworks
 - Standards and classes that support component communication and information exchange
- Enterprise application frameworks
 - Support the development of specific types of application such as telecommunications or financial systems

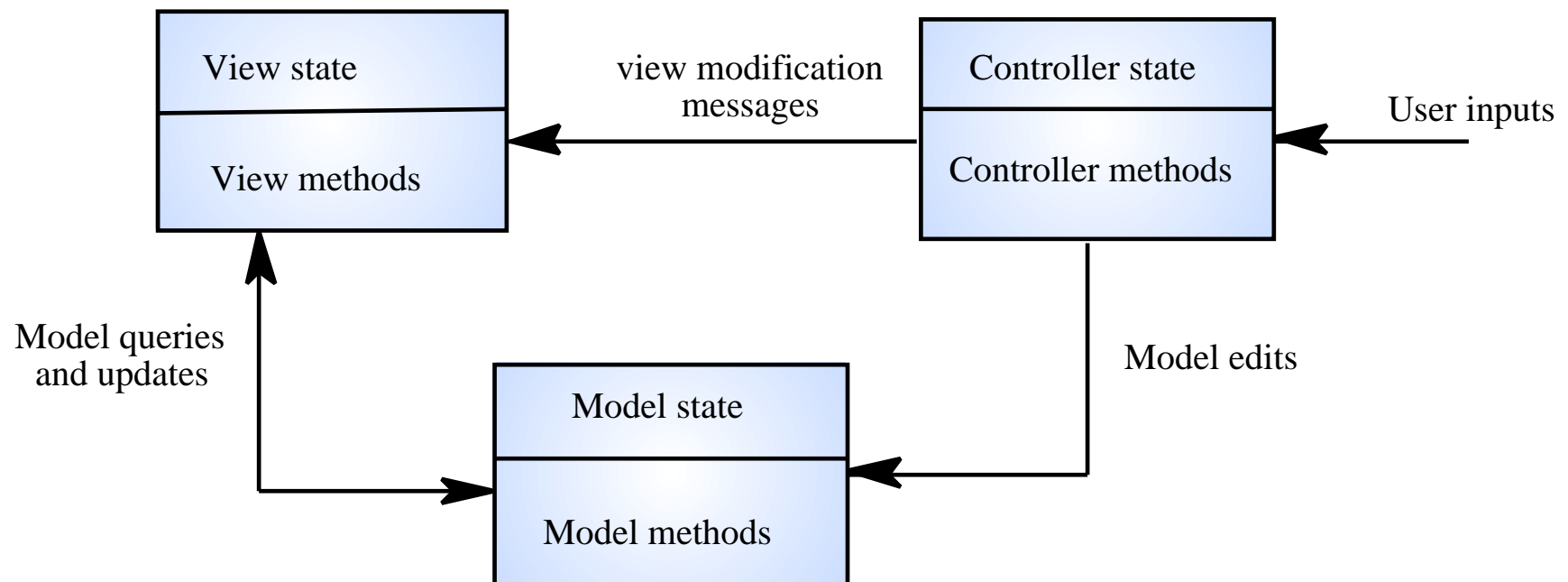
Extending frameworks

- Frameworks are generic and are extended to create a more specific application or sub-system
- Extending the framework involves
 - Adding concrete classes that inherit operations from abstract classes in the framework
 - Adding methods that are called in response to events that are recognised by the framework
- Problem with frameworks is their complexity and the time it takes to use them effectively

Model-view controller

- System infrastructure framework for GUI design
- Allows for multiple presentations of an object and separate interactions with these presentations
- MVC framework involves the instantiation of a number of patterns (discussed later)

Model-view controller



COTS product reuse

- COTS - Commercial Off-The-Shelf systems
- COTS systems are usually complete application systems that offer an API (Application Programming Interface)
- Building large systems by integrating COTS systems is now a viable development strategy for some types of system such as E-commerce systems

COTS system integration problems

- Lack of control over functionality and performance
 - COTS systems may be less effective than they appear
- Problems with COTS system inter-operability
 - Different COTS systems may make different assumptions that means integration is difficult
- No control over system evolution
 - COTS vendors not system users control evolution
- Support from COTS vendors
 - COTS vendors may not offer support over the lifetime of the product

Component development for reuse

- Components for reuse may be specially constructed by generalising existing components
- Component reusability
 - Should reflect stable domain abstractions
 - Should hide state representation
 - Should be as independent as possible
 - Should publish exceptions through the component interface
- There is a trade-off between reusability and usability.
 - The more general the interface, the greater the reusability but it is then more complex and hence less usable

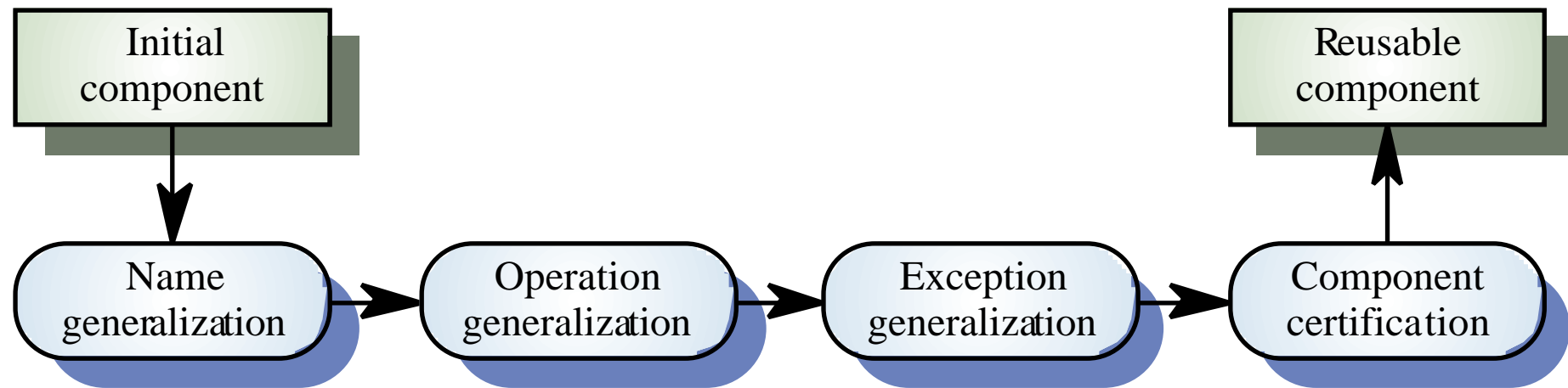
Reusable components

- The development cost of reusable components is higher than the cost of specific equivalents. This extra reusability enhancement cost should be an organization rather than a project cost
- Generic components may be less space-efficient and may have longer execution times than their specific equivalents

Reusability enhancement

- Name generalisation
 - Names in a component may be modified so that they are not a direct reflection of a specific application entity
- Operation generalisation
 - Operations may be added to provide extra functionality and application specific operations may be removed
- Exception generalisation
 - Application specific exceptions are removed and exception management added to increase the robustness of the component
- Component certification
 - Component is certified as reusable

Reusability enhancement process



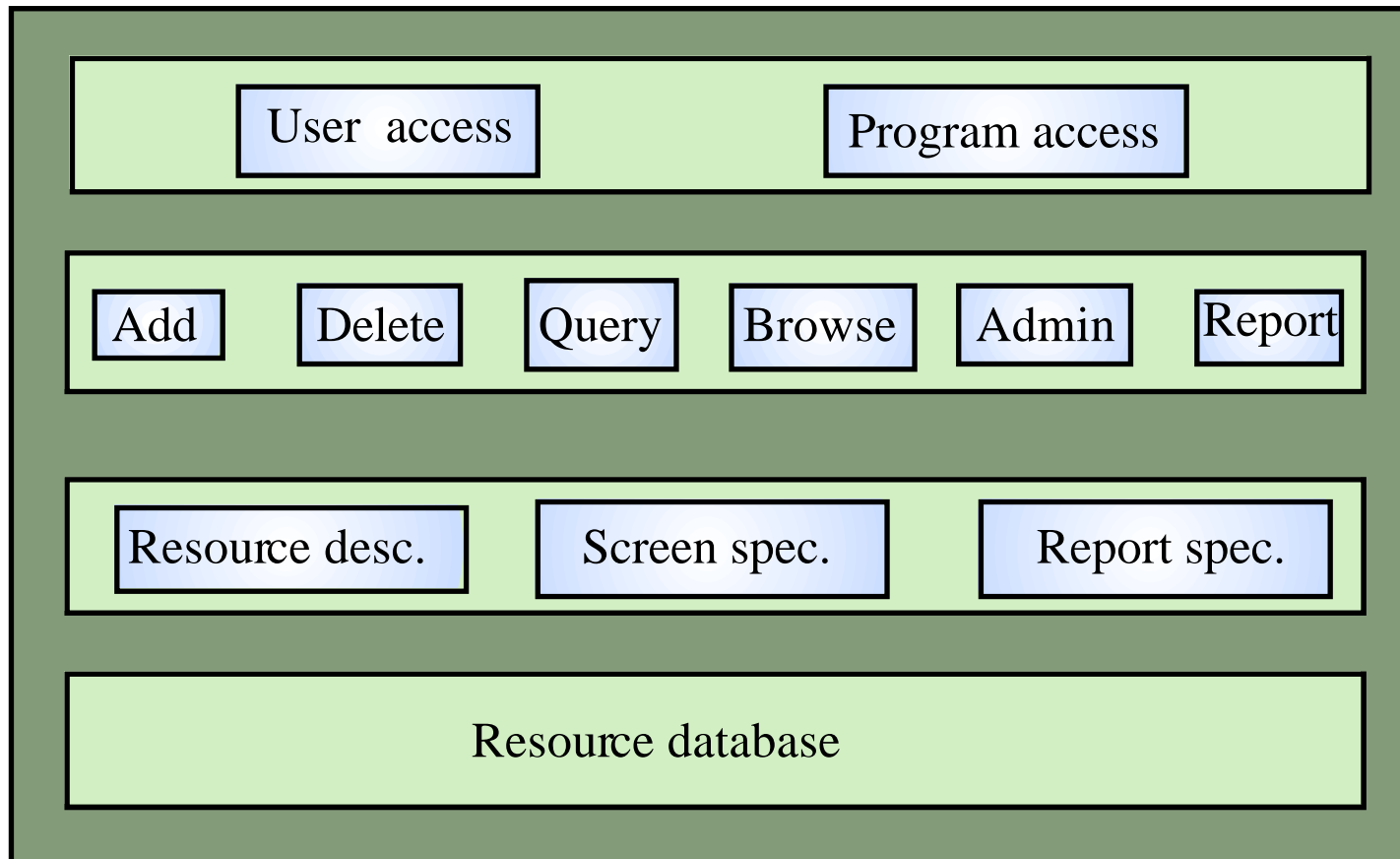
Application families

- An application family or product line is a related set of applications that has a common, domain-specific architecture
- The common core of the application family is reused each time a new application is required
- Each specific application is specialised in some way

Application family specialisation

- Platform specialisation
 - Different versions of the application are developed for different platforms
- Configuration specialisation
 - Different versions of the application are created to handle different peripheral devices
- Functional specialisation
 - Different versions of the application are created for customers with different requirements

A resource management system



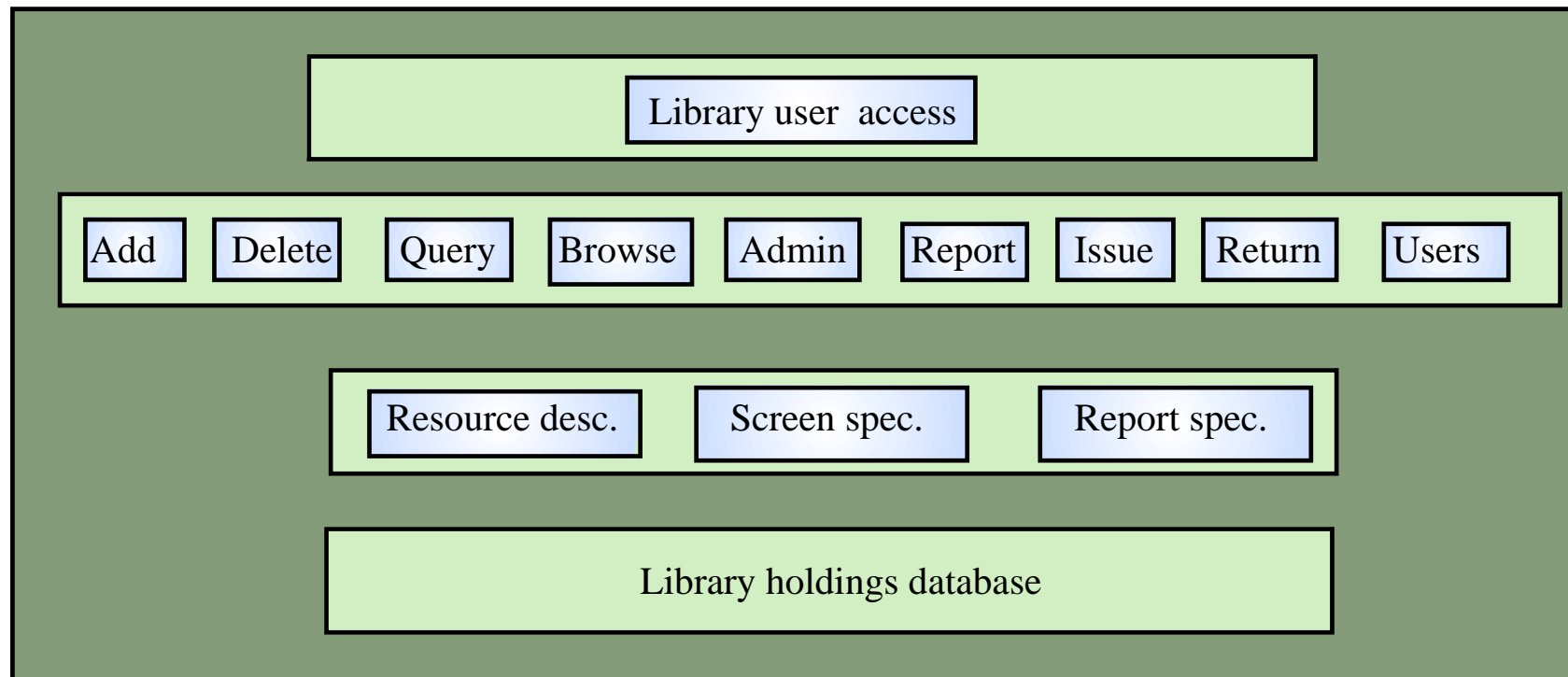
Inventory management systems

- Resource database
 - Maintains details of the things that are being managed
- I/O descriptions
 - Describes the structures in the resource database and input and output formats that are used
- Query level
 - Provides functions implementing queries over the resources
- Access interfaces
 - A user interface and an application programming interface

Application family architectures

- Architectures must be structured in such a way to separate different sub-systems and to allow them to be modified
- The architecture should also separate entities and their descriptions and the higher levels in the system access entities through descriptions rather than directly

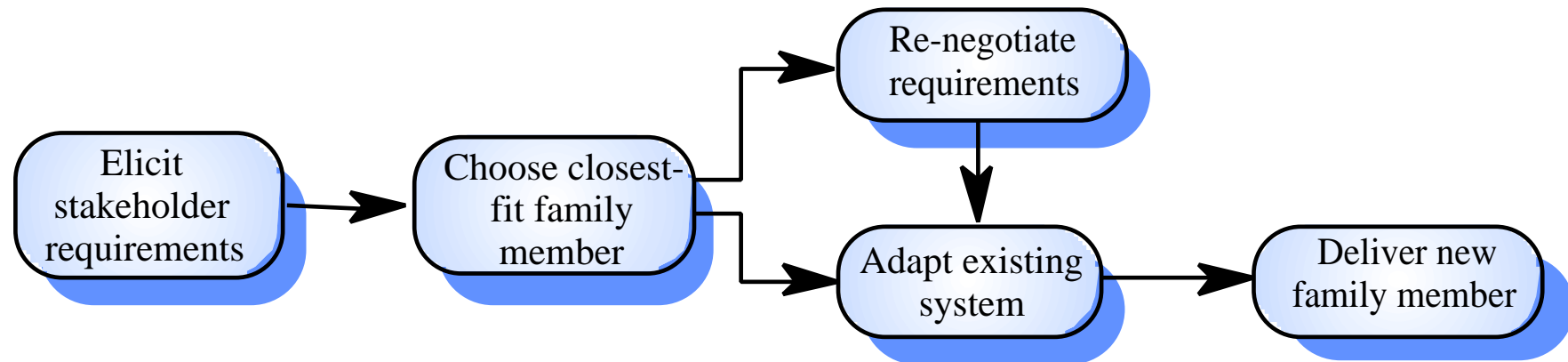
A library system



Library system

- The resources being managed are the books in the library
- Additional domain-specific functionality (issue, borrow, etc.) must be added for this application

Family member development



Family member development

- Elicit stakeholder requirements
 - Use existing family member as a prototype
- Choose closest-fit family member
 - Find the family member that best meets the requirements
- Re-negotiate requirements
 - Adapt requirements as necessary to capabilities of the software
- Adapt existing system
 - Develop new modules and make changes for family member
- Deliver new family member
 - Document key features for further member development

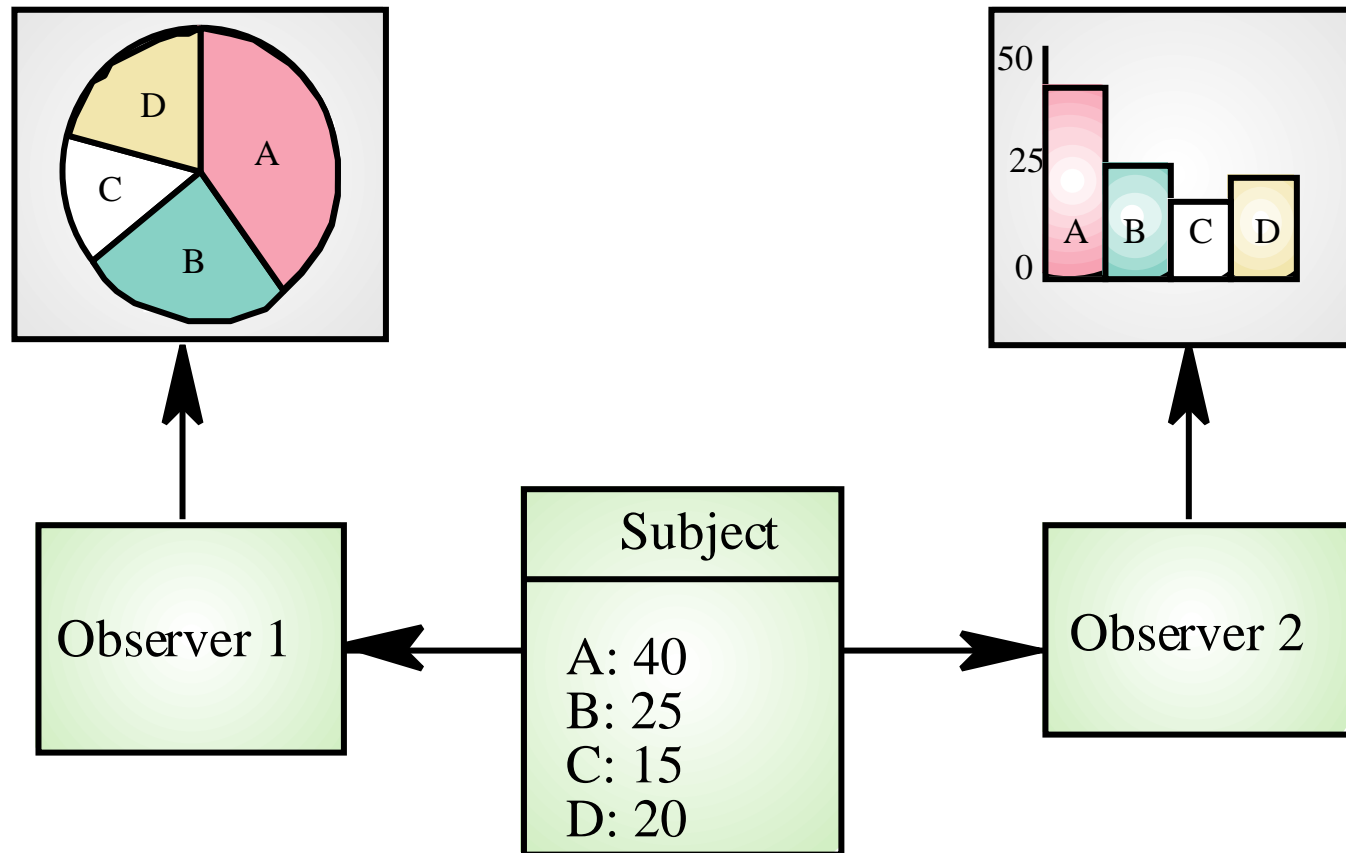
Design patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution
- A pattern is a description of the problem and the essence of its solution
- It should be sufficiently abstract to be reused in different settings
- Patterns often rely on object characteristics such as inheritance and polymorphism

Pattern elements

- Name
 - A meaningful pattern identifier
- Problem description
- Solution description
 - Not a concrete design but a template for a design solution that can be instantiated in different ways
- Consequences
 - The results and trade-offs of applying the pattern

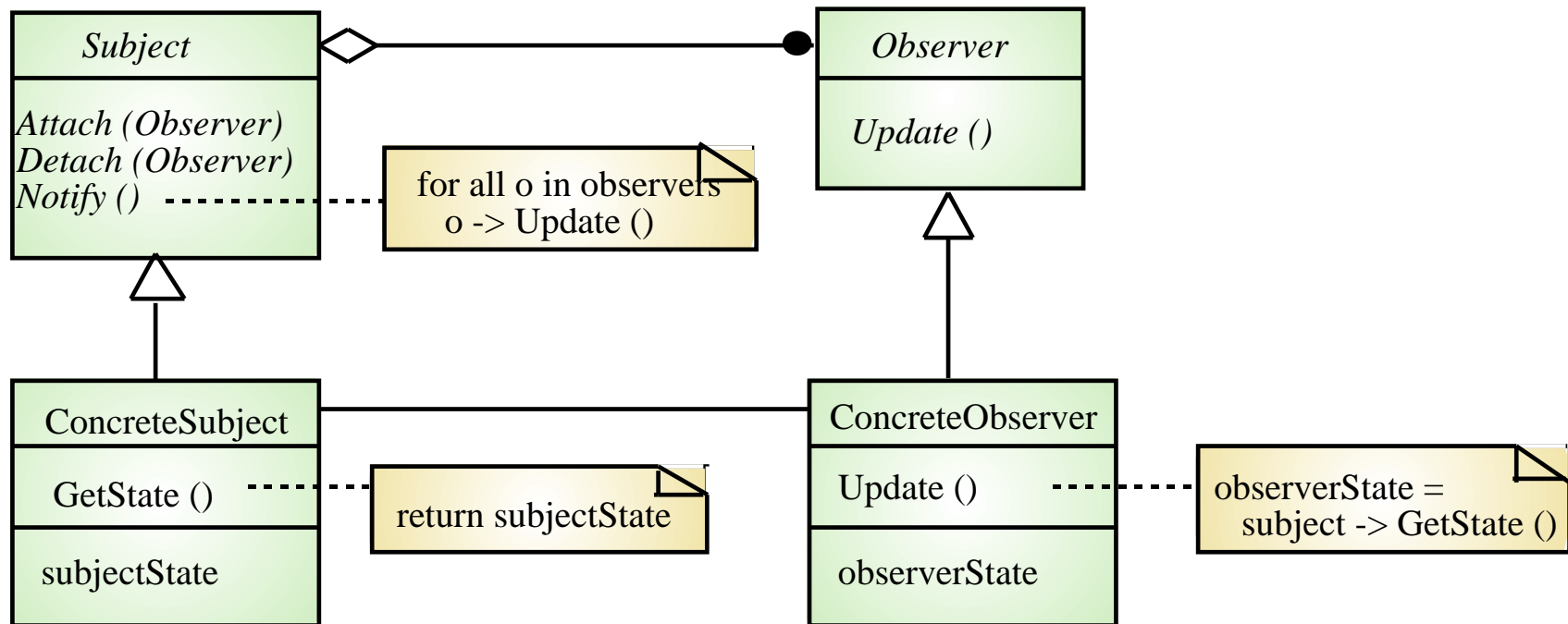
Multiple displays



The Observer pattern

- Name
 - Observer
- Description
 - Separates the display of object state from the object itself
- Problem description
 - Used when multiple displays of state are needed
- Solution description
 - See slide with UML description
- Consequences
 - Optimisations to enhance display performance are impractical

The Observer pattern



Key points

- Design with reuse involves designing software around good design and existing components
- Advantages are lower costs, faster software development and lower risks
- Component-based software engineering relies on black-box components with defined requires and provides interfaces
- COTS product reuse is concerned with the reuse of large, off-the-shelf systems

Key points

- Software components for reuse should be independent, should reflect stable domain abstractions and should provide access to state through interface operations
- Application families are related applications developed around a common core
- Design patterns are high-level abstractions that document successful design solutions

User interface design

- Designing effective interfaces for software systems

Objectives

- To suggest some general design principles for user interface design
- To explain different interaction styles
- To introduce styles of information presentation
- To describe the user support which should be built-in to user interfaces
- To introduce usability attributes and system approaches to system evaluation

Topics covered

- User interface design principles
- User interaction
- Information presentation
- User support
- Interface evaluation

The user interface

- 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

Graphical user interfaces

- Most users of business systems interact with these systems through graphical interfaces although, in some cases, legacy text-based interfaces are still used

GUI characteristics

Characteristic	Description
Windows	Multiple windows allow different information to be displayed simultaneously on the user's screen.
Icons	Icons different types of information. On some systems, icons represent files; on others, icons represent processes.
Menus	Commands are selected from a menu rather than typed in a command language.
Pointing	A pointing device such as a mouse is used for selecting choices from a menu or indicating items of interest in a window.
Graphics	Graphical elements can be mixed with text on the same display.

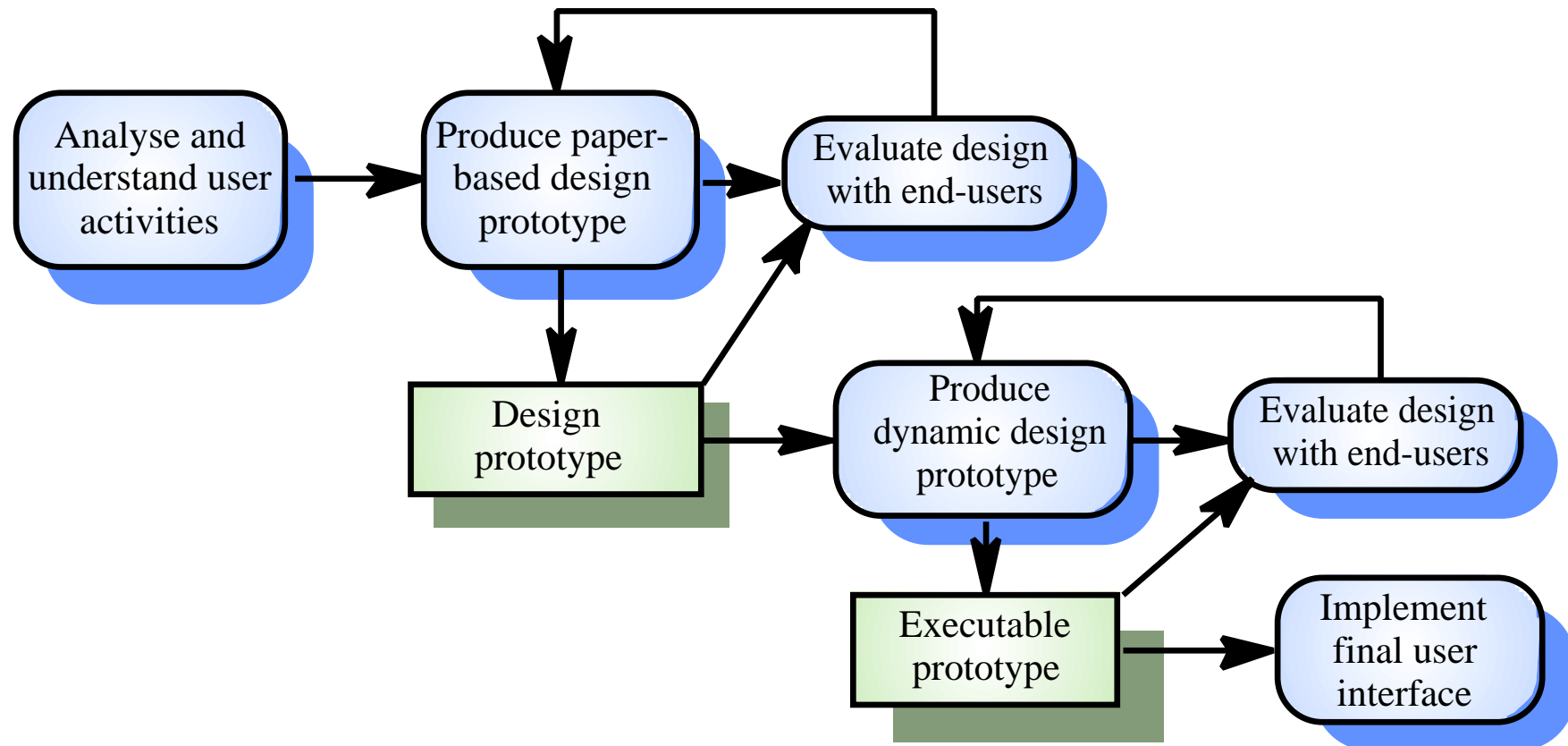
GUI advantages

- They are easy to learn and use.
 - Users without experience can learn to use the system quickly.
- The user may switch quickly from one task to another and can interact with several different applications.
 - Information remains visible in its own window when attention is switched.
- Fast, full-screen interaction is possible with immediate access to anywhere on the screen

User-centred design

- The aim of this chapter is to sensitise software engineers to key issues underlying the design rather than the implementation of user interfaces
- User-centred design is an approach to UI design where the needs of the user are paramount and where the user is involved in the design process
- UI design always involves the development of prototype interfaces

User interface design process



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

User-system interaction

- 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 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	Takes up a lot of screen space	Stock control, Personal loan processing
Command language	Powerful and flexible	Hard to learn Poor error management	Operating systems, Library information retrieval systems
Natural language	Accessible to casual users Easily extended	Requires more typing Natural language understanding systems are unreliable	Timetable systems WWW information retrieval systems

Advantages and disadvantages

Direct manipulation advantages

- Users feel in control of the computer and are less likely to be intimidated by it
- User learning time is relatively short
- Users get immediate feedback on their actions so mistakes can be quickly detected and corrected

Direct manipulation problems

- The derivation of an appropriate information space model can be very difficult
- Given that users have a large information space, what facilities for navigating around that space should be provided?
- Direct manipulation interfaces can be complex to program and make heavy demands on the computer system

Control panel interface

Title	JSD. example	<input type="checkbox"/>	Grid	Busy
Method	JSD			
Type	Network	Units	cm ▶	QUIT
Selection	Process	Reduce	Full ▶	PRINT
NODE LINKS FONT LABEL EDIT				

Menu systems

- Users make a selection from a list of possibilities presented to them by the system
- The selection may be made by pointing and clicking with a mouse, using cursor keys or by typing the name of the selection
- May make use of simple-to-use terminals such as touchscreens

Advantages of menu systems

- Users need not remember command names as they are always presented with a list of valid commands
- Typing effort is minimal
- User errors are trapped by the interface
- Context-dependent help can be provided. The user's context is indicated by the current menu selection

Problems with menu systems

- Actions which involve logical conjunction (and) or disjunction (or) are awkward to represent
- Menu systems are best suited to presenting a small number of choices. If there are many choices, some menu structuring facility must be used
- Experienced users find menus slower than command language

Form-based interface

NEW BOOK			
Title	<input type="text"/>	ISBN	<input type="text"/>
Author	<input type="text"/>	Price	<input type="text"/>
Publisher	<input type="text"/>	Publication date	<input type="text"/>
Edition	<input type="text"/>	Number of copies	<input type="text"/>
Classification	<input type="text"/>	Loan status	<input type="text"/>
Date of purchase	<input type="text"/>	Order status	<input type="text"/>

Command interfaces

- User types commands to give instructions to the system e.g. UNIX
- May be implemented using cheap terminals.
- Easy to process using compiler techniques
- Commands of arbitrary complexity can be created by command combination
- Concise interfaces requiring minimal typing can be created

Problems with command interfaces

- Users have to learn and remember a command language. Command interfaces are therefore unsuitable for occasional users
- Users make errors in command. An error detection and recovery system is required
- System interaction is through a keyboard so typing ability is required

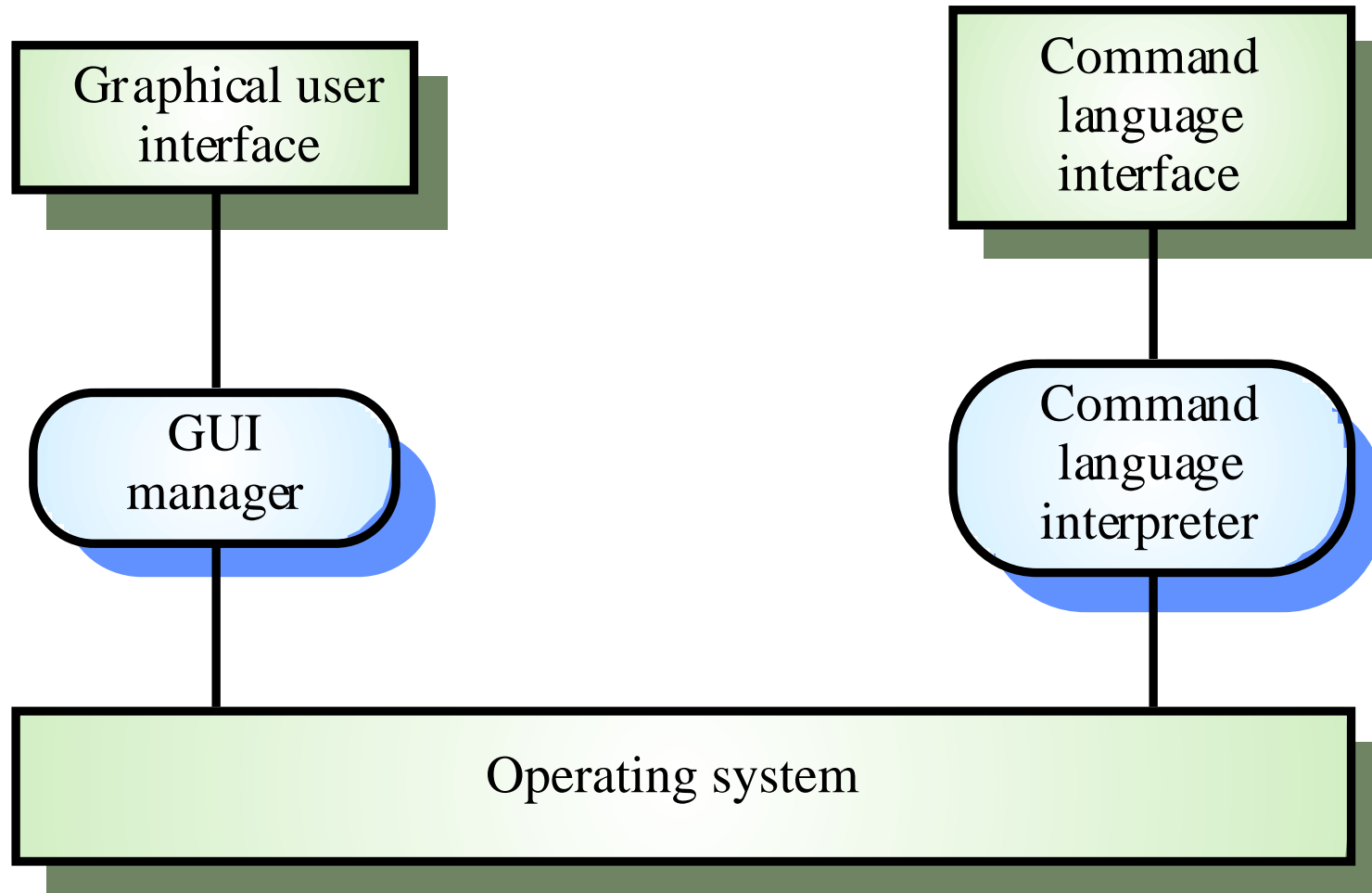
Command languages

- Often preferred by experienced users because they allow for faster interaction with the system
- Not suitable for casual or inexperienced users
- May be provided as an alternative to menu commands (keyboard shortcuts). In some cases, a command language interface and a menu-based interface are supported at the same time

Natural language interfaces

- The user types a command in a natural language. Generally, the vocabulary is limited and these systems are confined to specific application domains (e.g. timetable enquiries)
- NL processing technology is now good enough to make these interfaces effective for casual users but experienced users find that they require too much typing

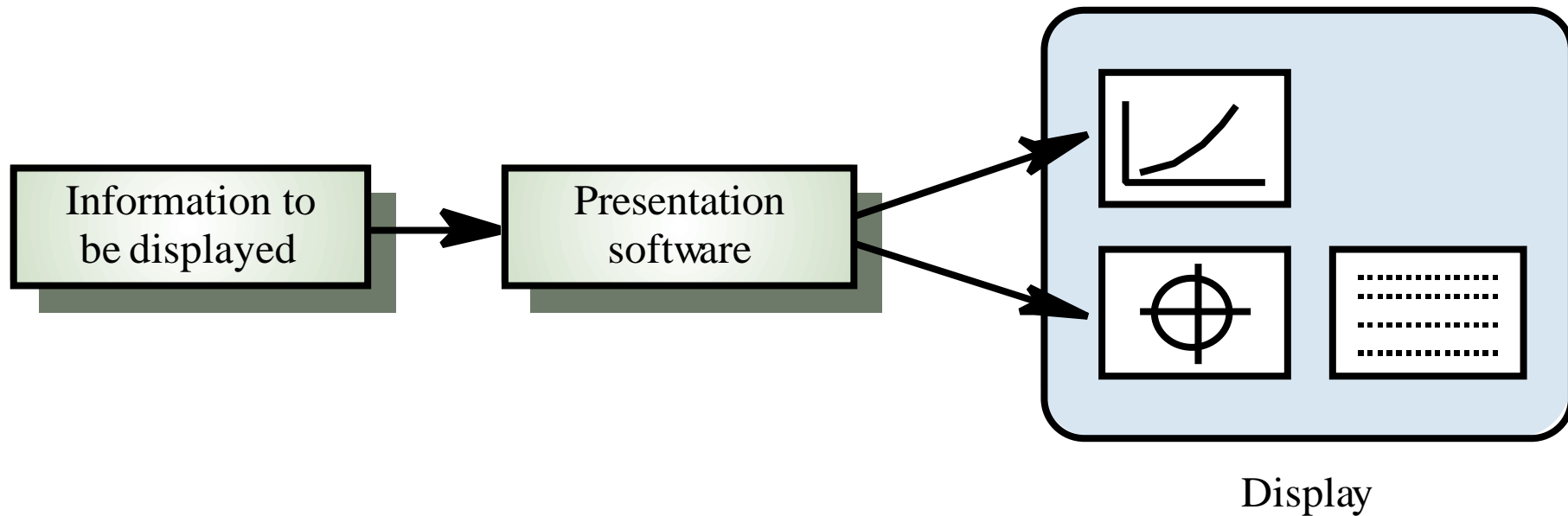
Multiple user interfaces



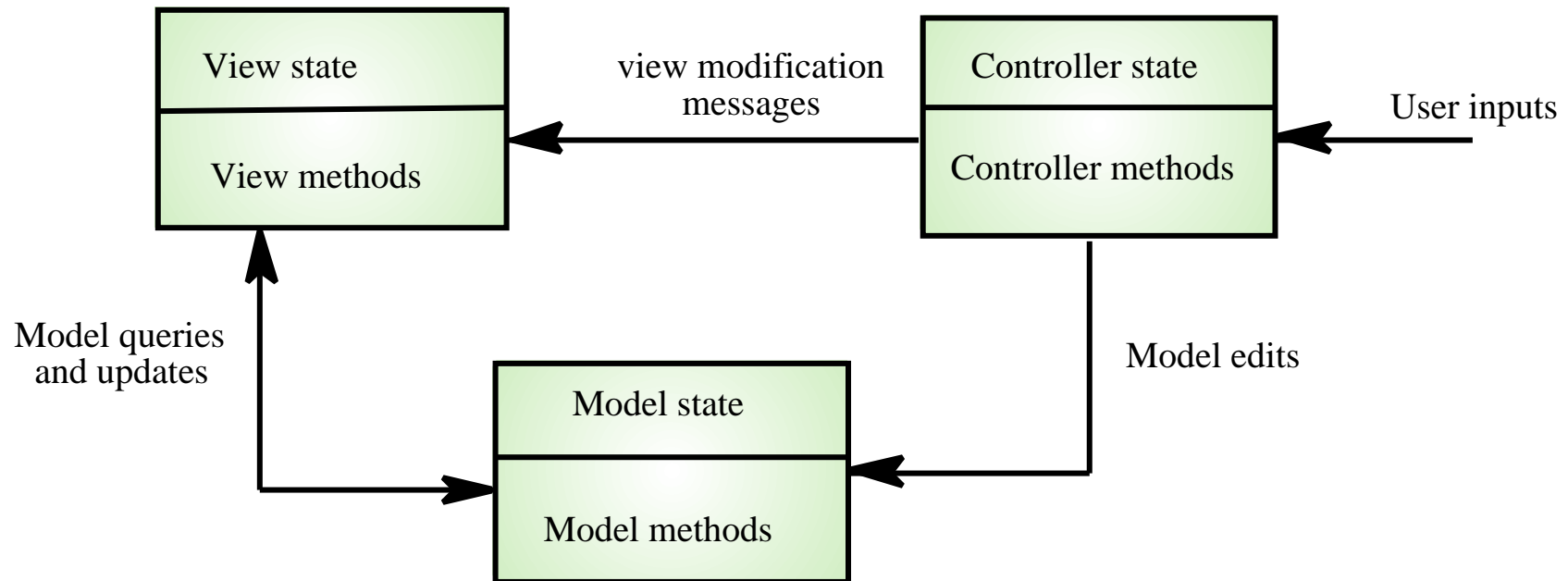
Information presentation

- 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

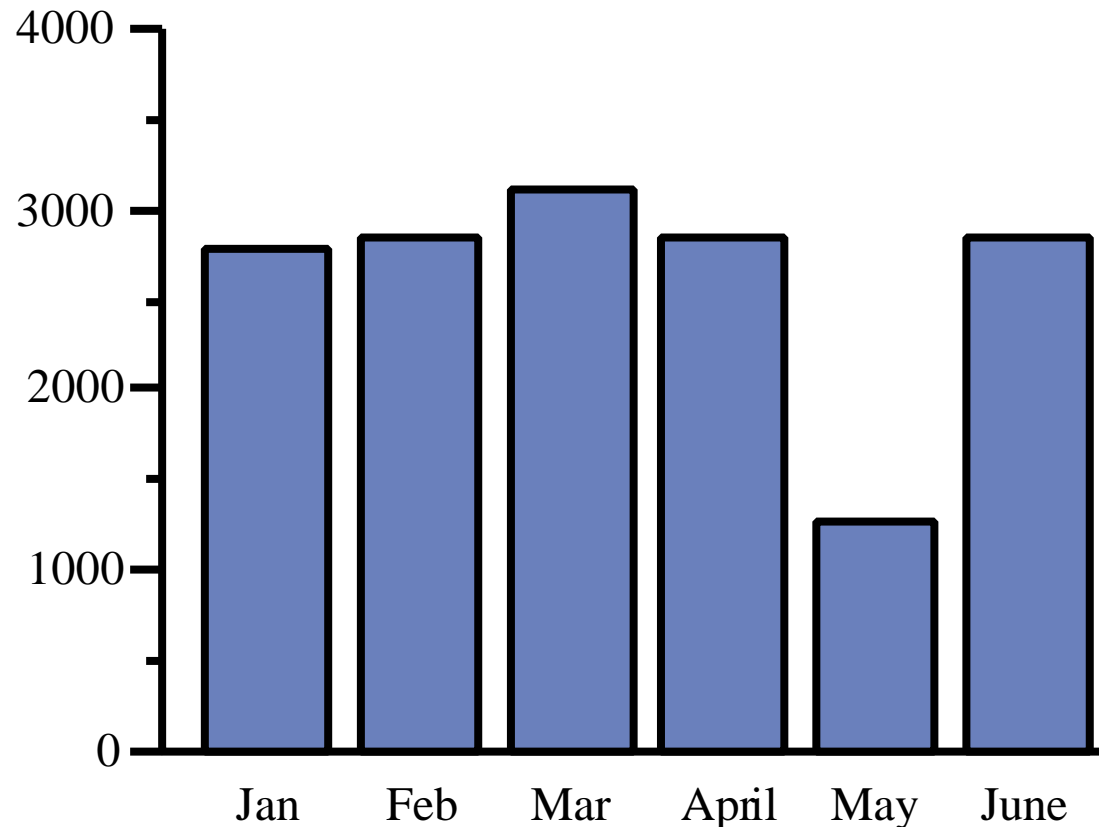
- Static information
 - Initialised at the beginning of a session. It does not change during the session
 - May be either numeric or textual
- Dynamic information
 - Changes during a session and the changes must be communicated to the system user
 - May be either numeric or textual

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

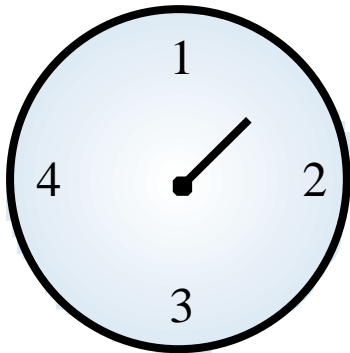
Jan	Feb	Mar	April	May	June
2842	2851	3164	2789	1273	2835



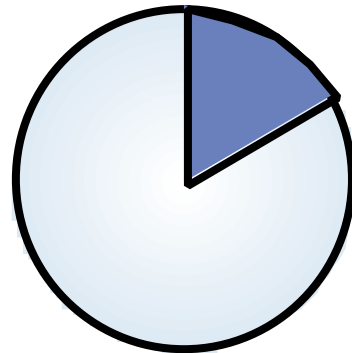
Analogue vs. digital presentation

- 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

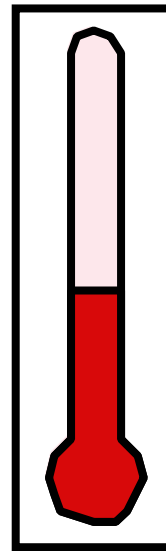
Dynamic information display



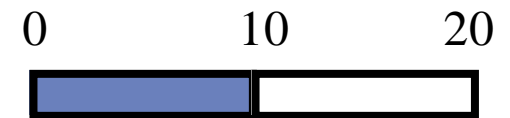
Dial with needle



Pie chart

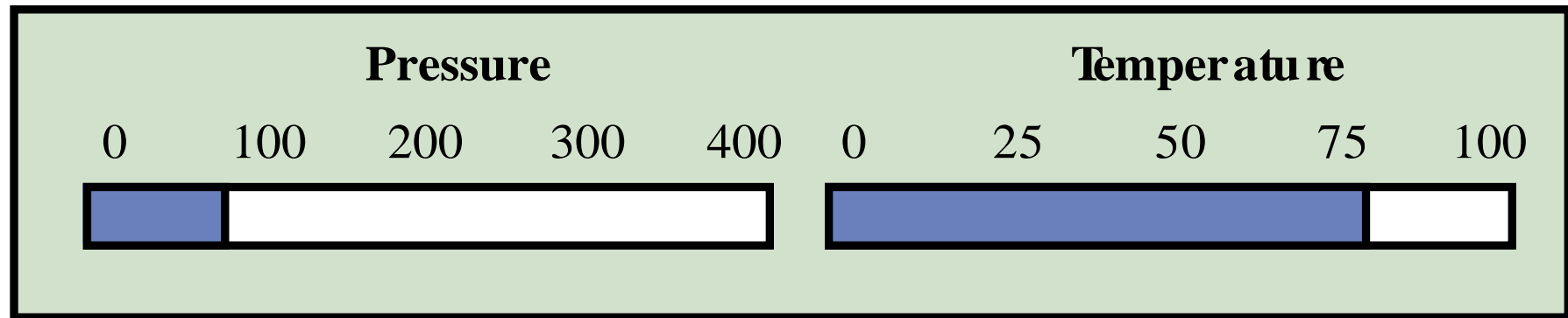


Thermometer

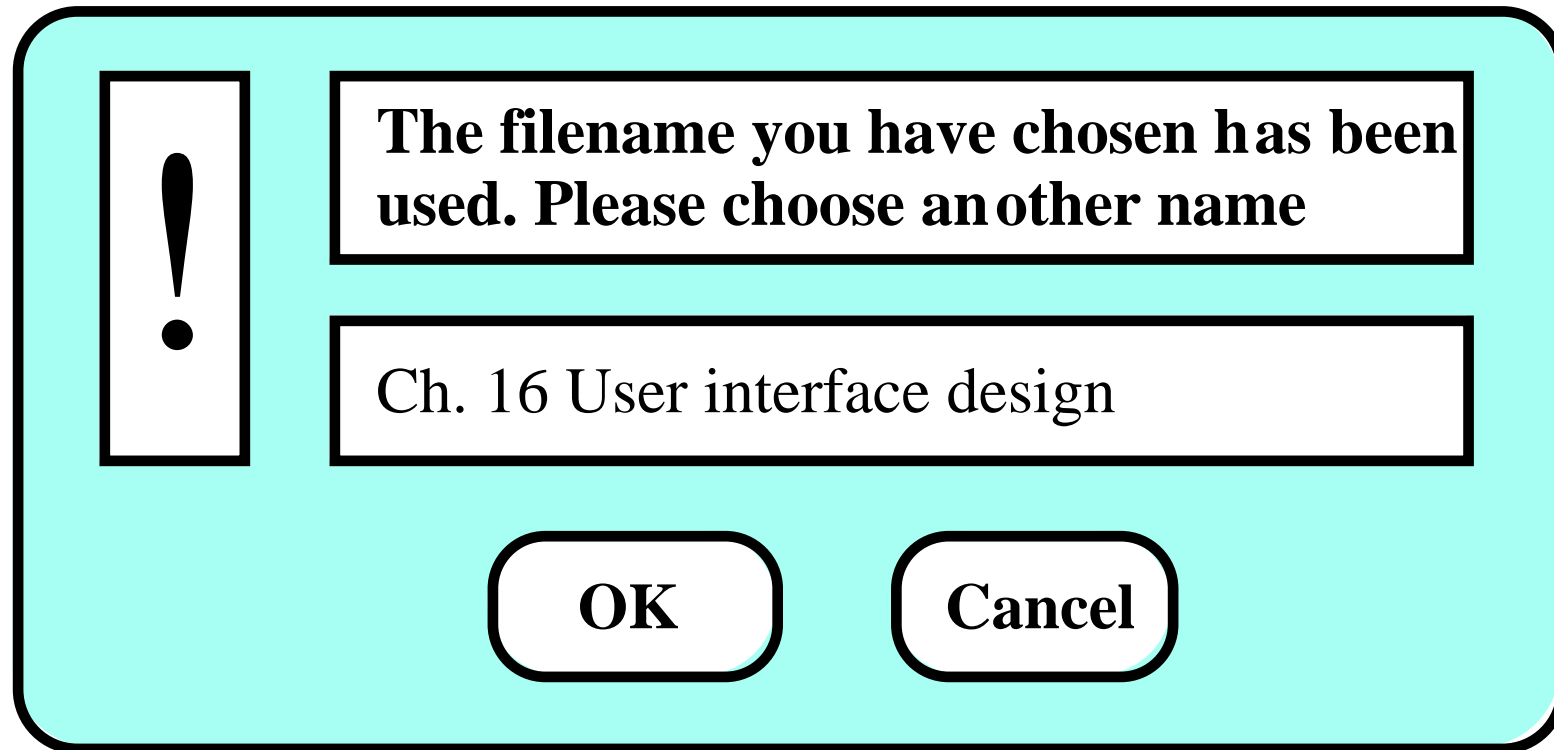


Horizontal bar

Displaying relative values



Textual highlighting



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
- 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
 - Over-use of colour in the display

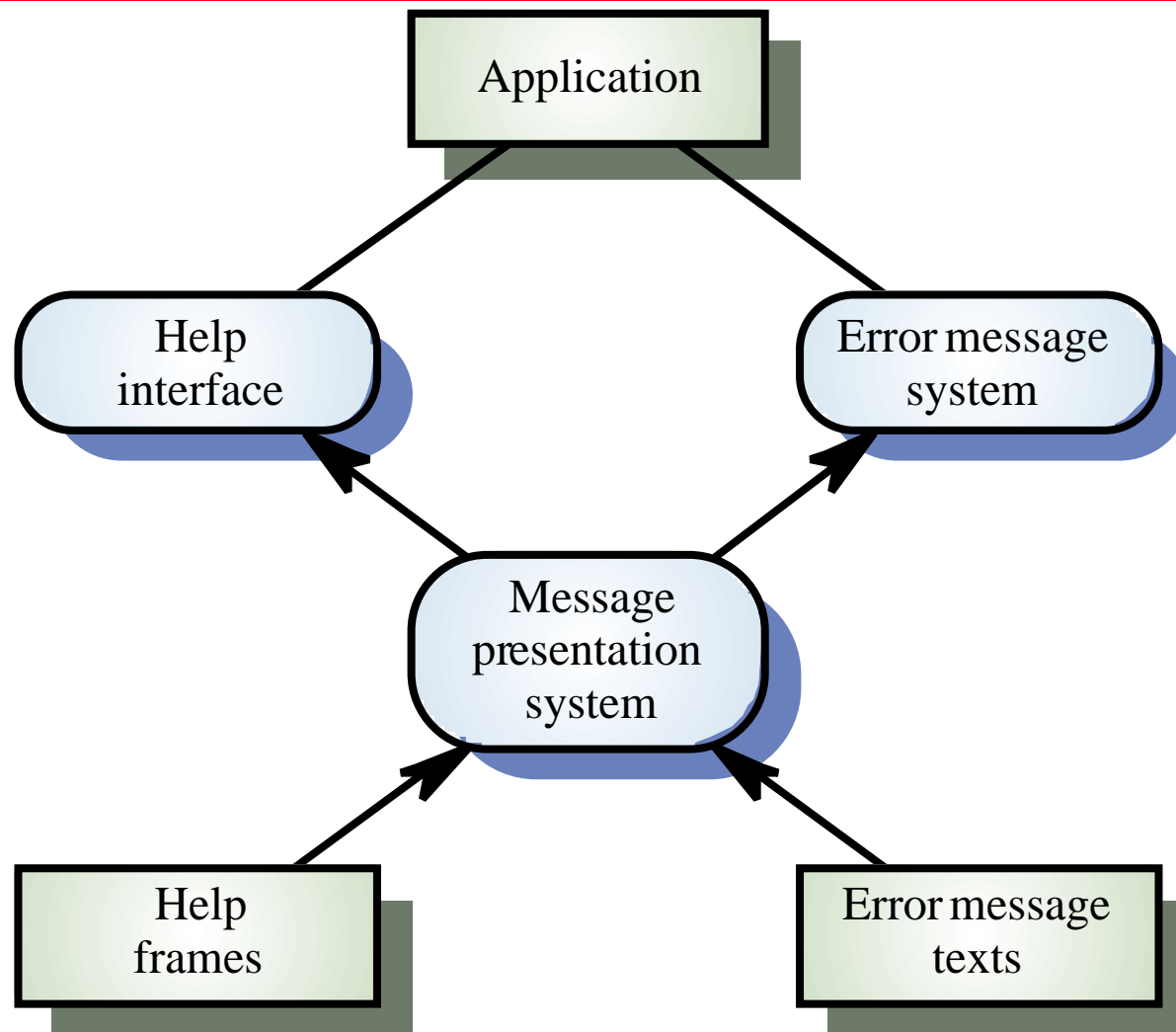
Colour use guidelines

- Don't use too many colours
- Use colour coding to support use tasks
- Allow users to control colour coding
- Design for monochrome then add colour
- Use colour coding consistently
- Avoid colour pairings which clash
- Use colour change to show status change
- Be aware that colour displays are usually lower resolution

User support

- User guidance covers all system facilities to support users including on-line help, error messages, manuals etc.
- The user guidance system should be integrated with the user interface to help users when they need information about the system or when they make some kind of error
- The help and message system should, if possible, be integrated

Help and message system



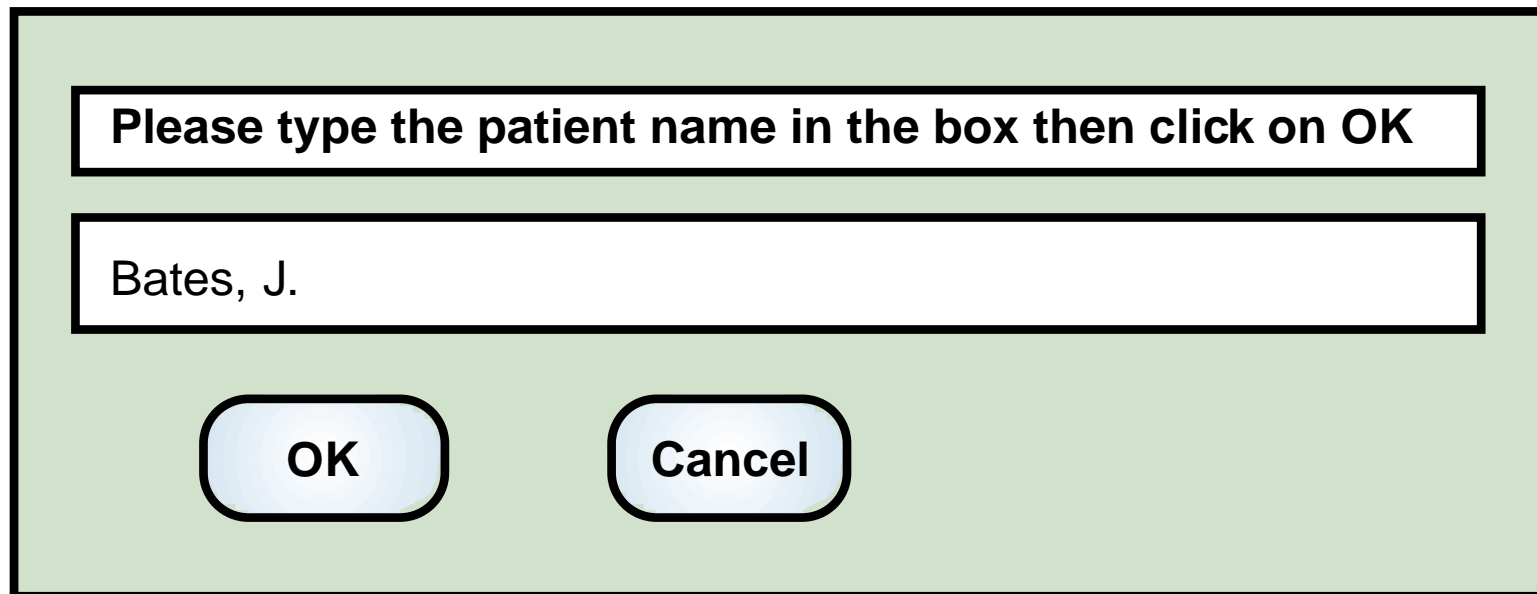
Error messages

- 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

Context	The user guidance system should be aware of what the user is doing and should adjust the output message to the current context.
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 the problem. The user guidance system 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 which 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.

Nurse input of a patient's name



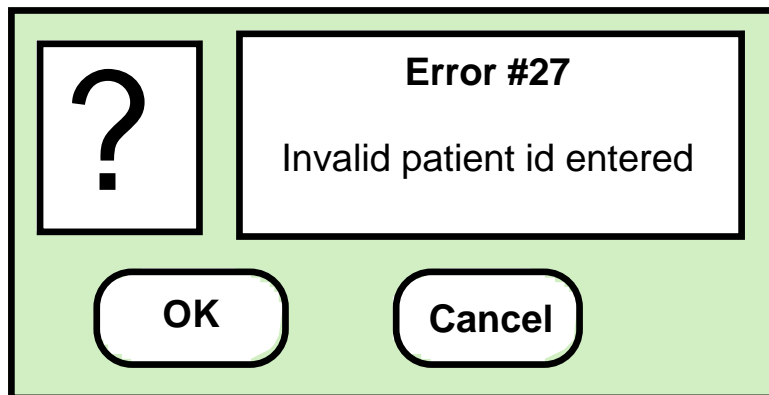
Please type the patient name in the box then click on OK

Bates, J.

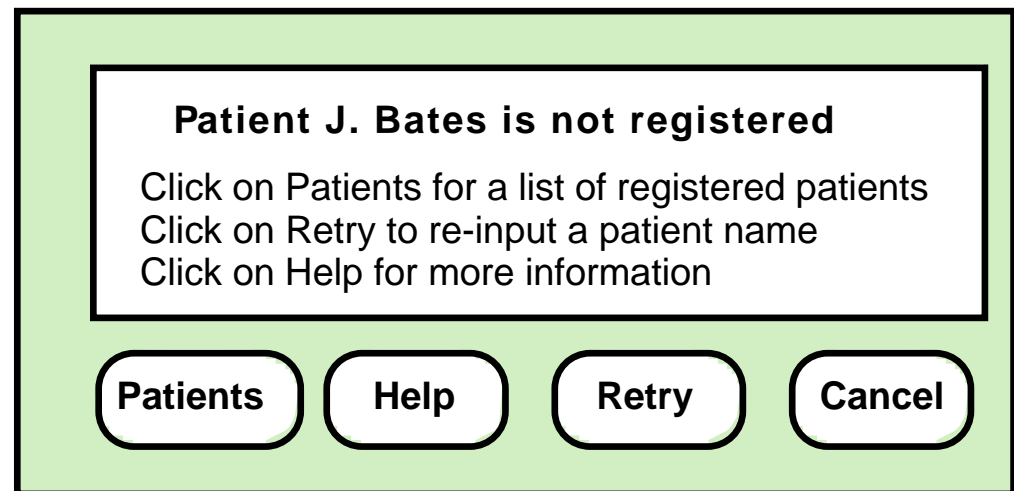
OK Cancel

System and user-oriented error messages

System-oriented error message



User-oriented error message



Help system design

- *Help?* means ‘help I want information’
- *Help!* means “HELP. I'm in trouble”
- Both of these requirements have to be taken into account in help system design
- Different facilities in the help system may be required

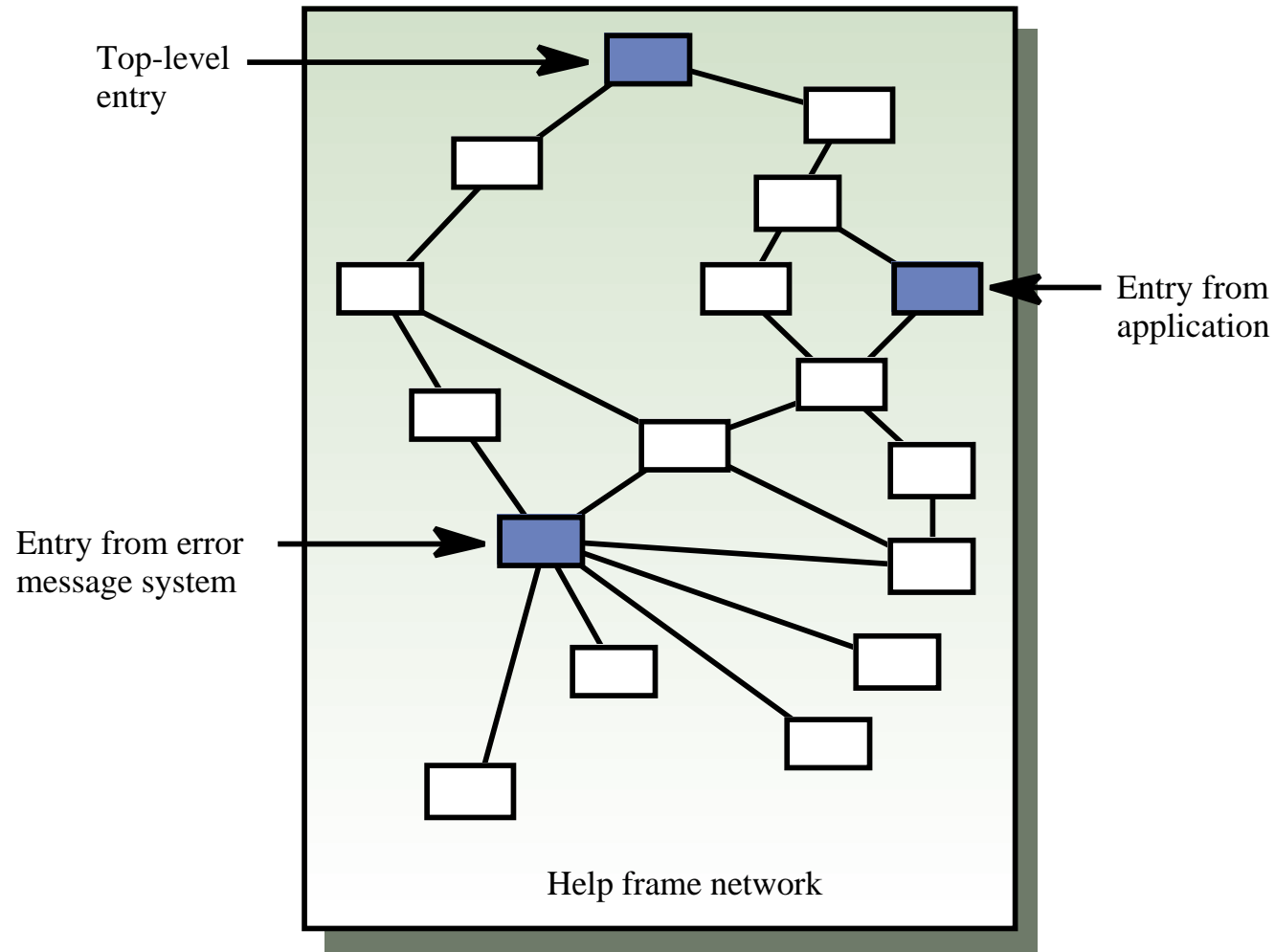
Help information

- Should not simply be an on-line manual
- Screens or windows don't map well onto paper pages.
- The dynamic characteristics of the display can improve information presentation.
- People are not so good at reading screen as they are text.

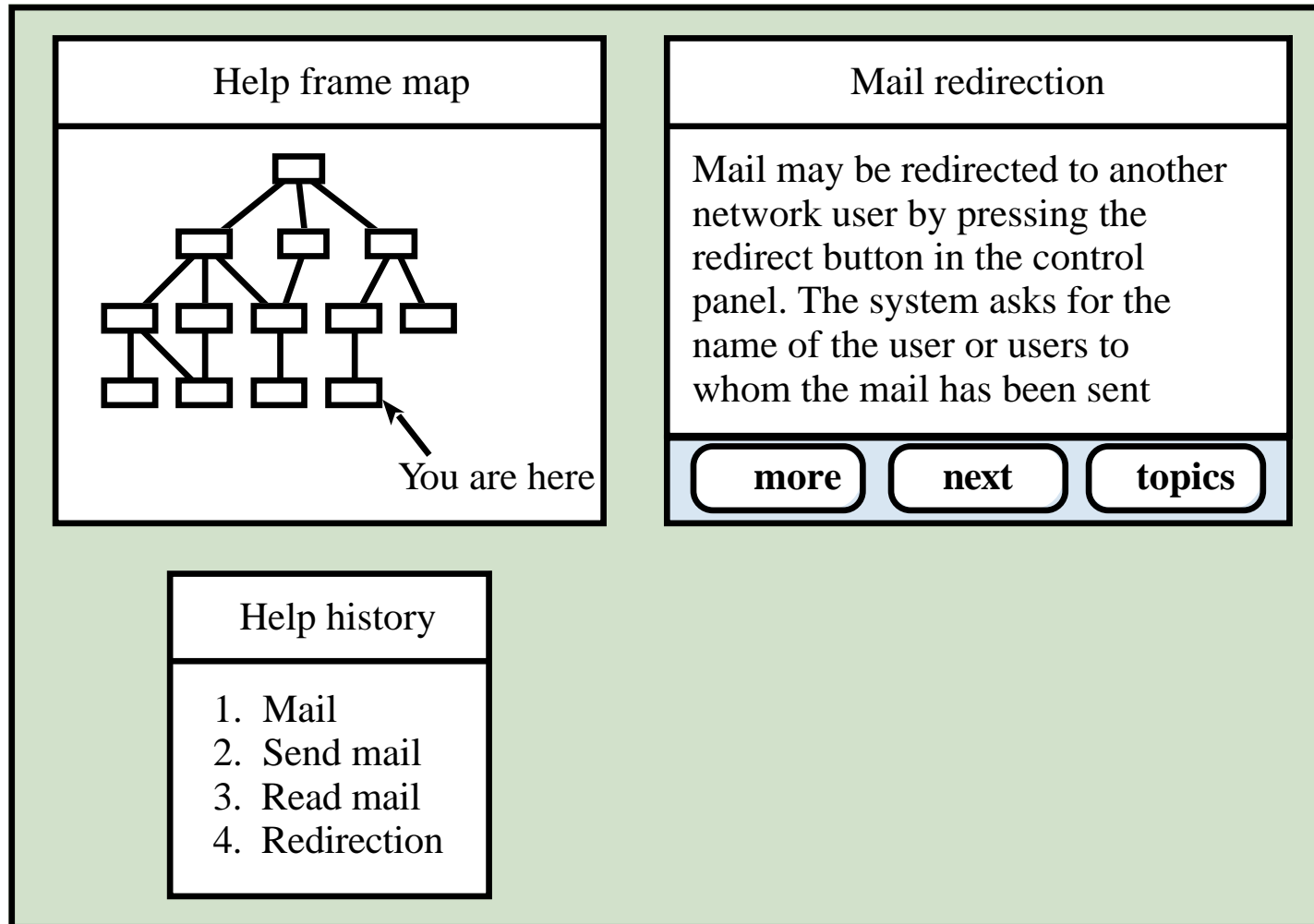
Help system use

- Multiple entry points should be provided so that the user can get into the help system from different places.
- Some indication of where the user is positioned in the help system is valuable.
- Facilities should be provided to allow the user to navigate and traverse the help system.

Entry points to a help system



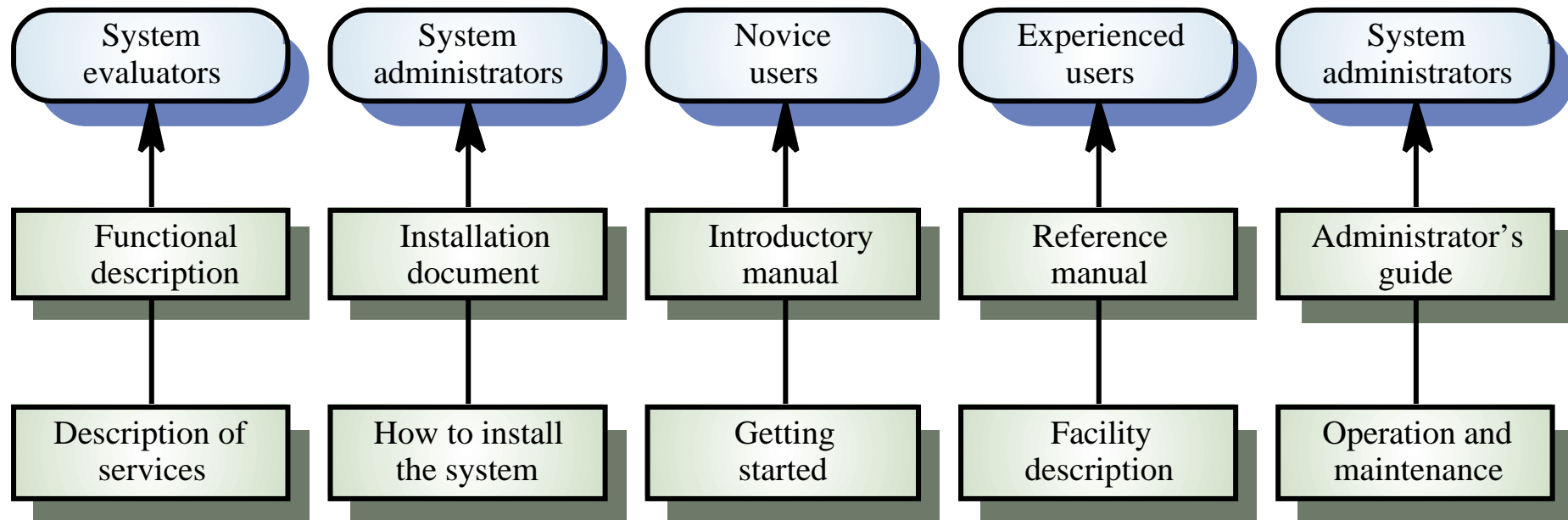
Help system windows



User documentation

- As well as on-line information, paper documentation should be supplied with a system
- Documentation should be designed for a range of users from inexperienced to experienced
- As well as manuals, other easy-to-use documentation such as a quick reference card may be provided

User document types



Document types

- Functional description
 - Brief description of what the system can do
- Introductory manual
 - Presents an informal introduction to the system
- System reference manual
 - Describes all system facilities in detail
- System installation manual
 - Describes how to install the system
- System administrator's manual
 - Describes how to manage the system when it is in use

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 a grip button for on-line user feedback.

Key points

- Interface design should be user-centred. An interface should be logical and consistent and help users recover from errors
- 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

- Systems should provide on-line help. This should include “help, I’m in trouble” and “help, I want information”
- Error messages should be positive rather than negative.
- A range of different types of user documents should be provided
- Ideally, a user interface should be evaluated against a usability specification

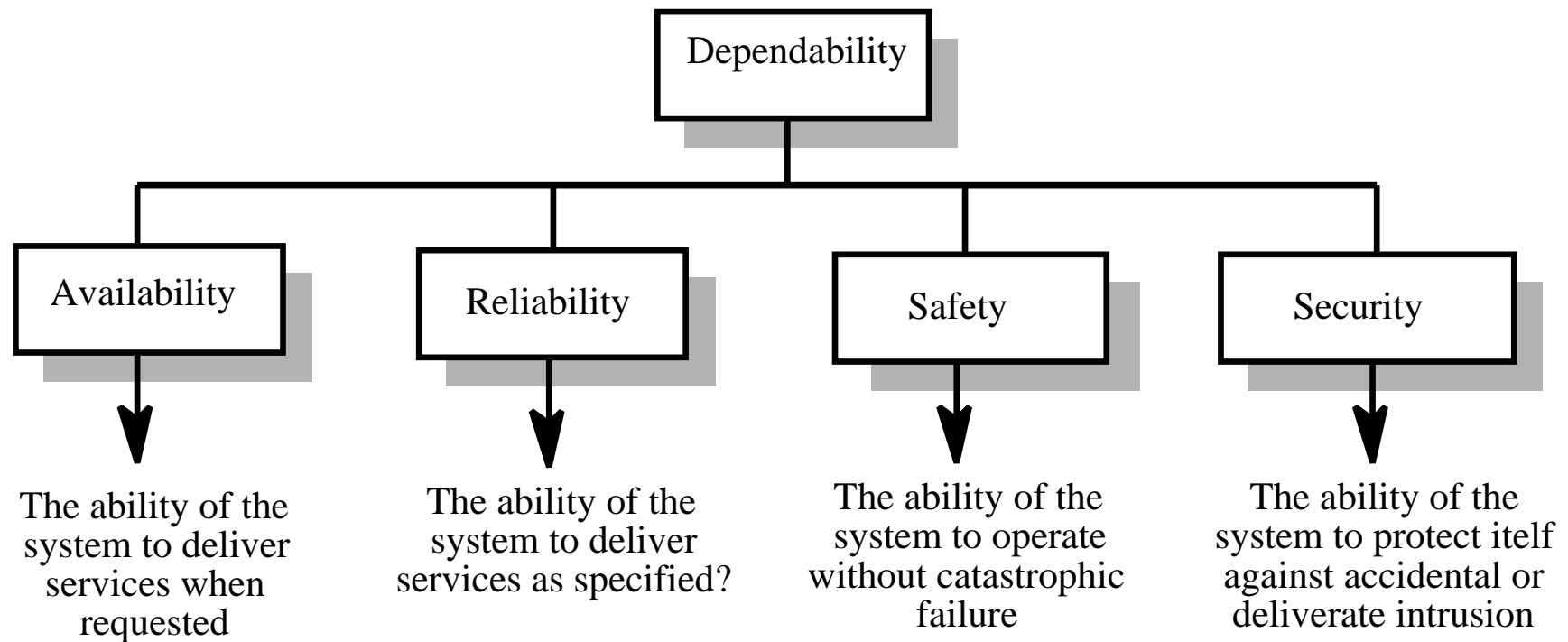
Dependability

- The extent to which a critical system is trusted by its users

The concept of dependability

- For critical systems, it is usually the case that the most important system property is the dependability of the system
- The dependability of a system reflects the user's degree of trust in that system. It reflects the extent of the user's confidence that it will operate as users expect and that it will not 'fail' in normal use
- Usefulness and trustworthiness are not the same thing. A system does not have to be trusted to be useful

Dimensions of dependability



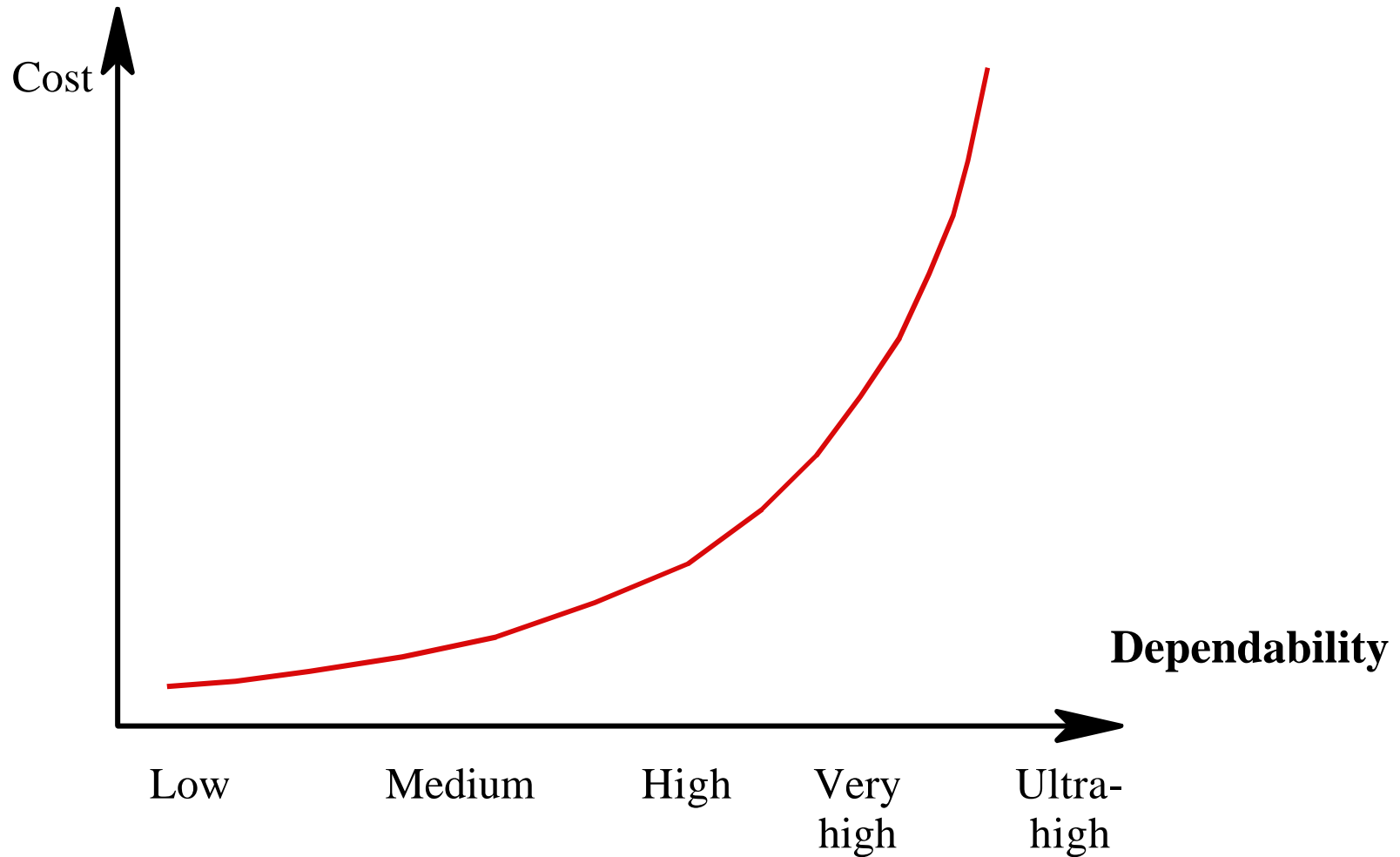
Maintainability

- A system attribute which is concerned with the ease of repairing the system after a failure has been discovered or changing the system to include new features
- Very important for critical systems as faults are often introduced into a system because of maintenance problems
- Maintainability is distinct from other dimensions of dependability because it is a static and not a dynamic system attribute. I do not cover it in this course.

Survivability

- The ability of a system to continue to deliver its services to users in the face of deliberate or accidental attack
- This is an increasingly important attribute for distributed systems whose security can be compromised
- Survivability subsumes the notion of resilience - the ability of a system to continue in operation in spite of component failures

Costs of increasing dependability



Dependability costs

- Dependability costs tend to increase exponentially as increasing levels of dependability are required
- There are two reasons for this
 - The use of more expensive development techniques and hardware that are required to achieve the higher levels of dependability
 - The increased testing and system validation that is required to convince the system client that the required levels of dependability have been achieved

Dependability vs performance

- Untrustworthy systems may be rejected by their users
- System failure costs may be very high
- It is very difficult to tune systems to make them more dependable
- It may be possible to compensate for poor performance
- Untrustworthy systems may cause loss of valuable information

Dependability economics

- Because of very high costs of dependability achievement, it may be more cost effective to accept untrustworthy systems and pay for failure costs
- However, this depends on social and political factors. A reputation for products that can't be trusted may lose future business
- Depends on system type - for business systems in particular, modest levels of dependability may be adequate

Availability and reliability

- Reliability
 - The probability of failure-free system operation over a specified time in a given environment for a given purpose
- Availability
 - The probability that a system, at a point in time, will be operational and able to deliver the requested services
- Both of these attributes can be expressed quantitatively

Availability and reliability

- It is sometimes possible to subsume system availability under system reliability
 - Obviously if a system is unavailable it is not delivering the specified system services
- However, it is possible to have systems with low reliability that must be available. So long as system failures can be repaired quickly and do not damage data, low reliability may not be a problem
- Availability takes repair time into account

Reliability terminology

Term	Description
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users
System error	Erroneous system behaviour where the behaviour of the system does not conform to its specification.
System fault	An incorrect system state i.e. a system state that is unexpected by the designers of the system.
Human error or mistake	Human behaviour that results in the introduction of faults into a system.

Faults and failures

- Failures are usually a result of system errors that are derived from faults in the system
- However, faults do not necessarily result in system errors
 - The faulty system state may be transient and ‘corrected’ before an error arises
- Errors do not necessarily lead to system failures
 - The error can be corrected by built-in error detection and recovery
 - The failure can be protected against by built-in protection facilities. These may, for example, protect system resources from system errors

Perceptions of reliability

- The formal definition of reliability does not always reflect the user's perception of a system's reliability
 - The assumptions that are made about the environment where a system will be used may be incorrect
 - Usage of a system in an office environment is likely to be quite different from usage of the same system in a university environment
 - The consequences of system failures affects the perception of reliability
 - Unreliable windscreen wipers in a car may be irrelevant in a dry climate
 - Failures that have serious consequences (such as an engine breakdown in a car) are given greater weight by users than failures that are inconvenient

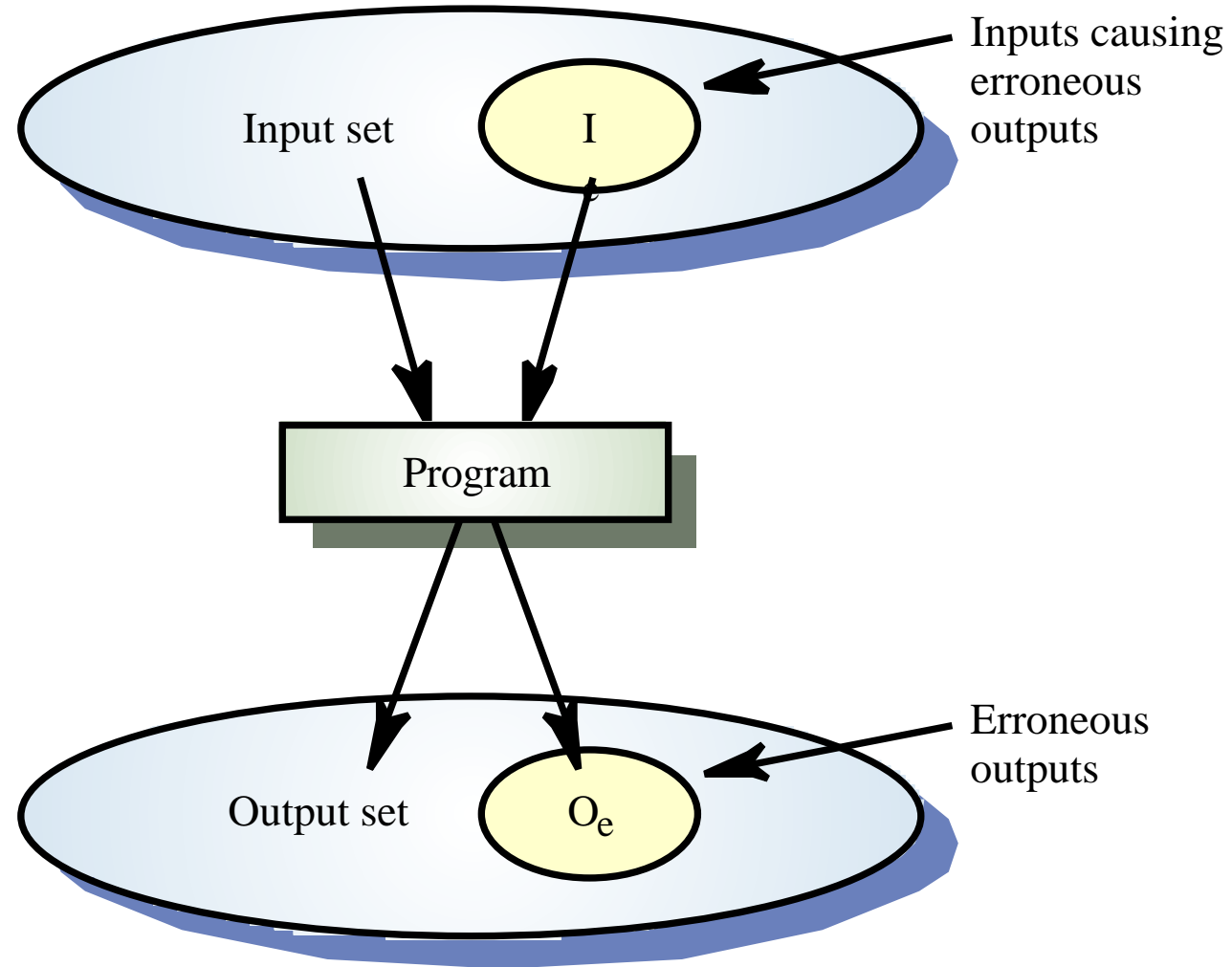
Reliability achievement

- Fault avoidance
 - Development techniques are used that either minimise the possibility of mistakes or trap mistakes before they result in the introduction of system faults
- Fault detection and removal
 - Verification and validation techniques that increase the probability of detecting and correcting errors before the system goes into service are used
- Fault tolerance
 - Run-time techniques are used to ensure that system faults do not result in system errors and/or that system errors do not lead to system failures

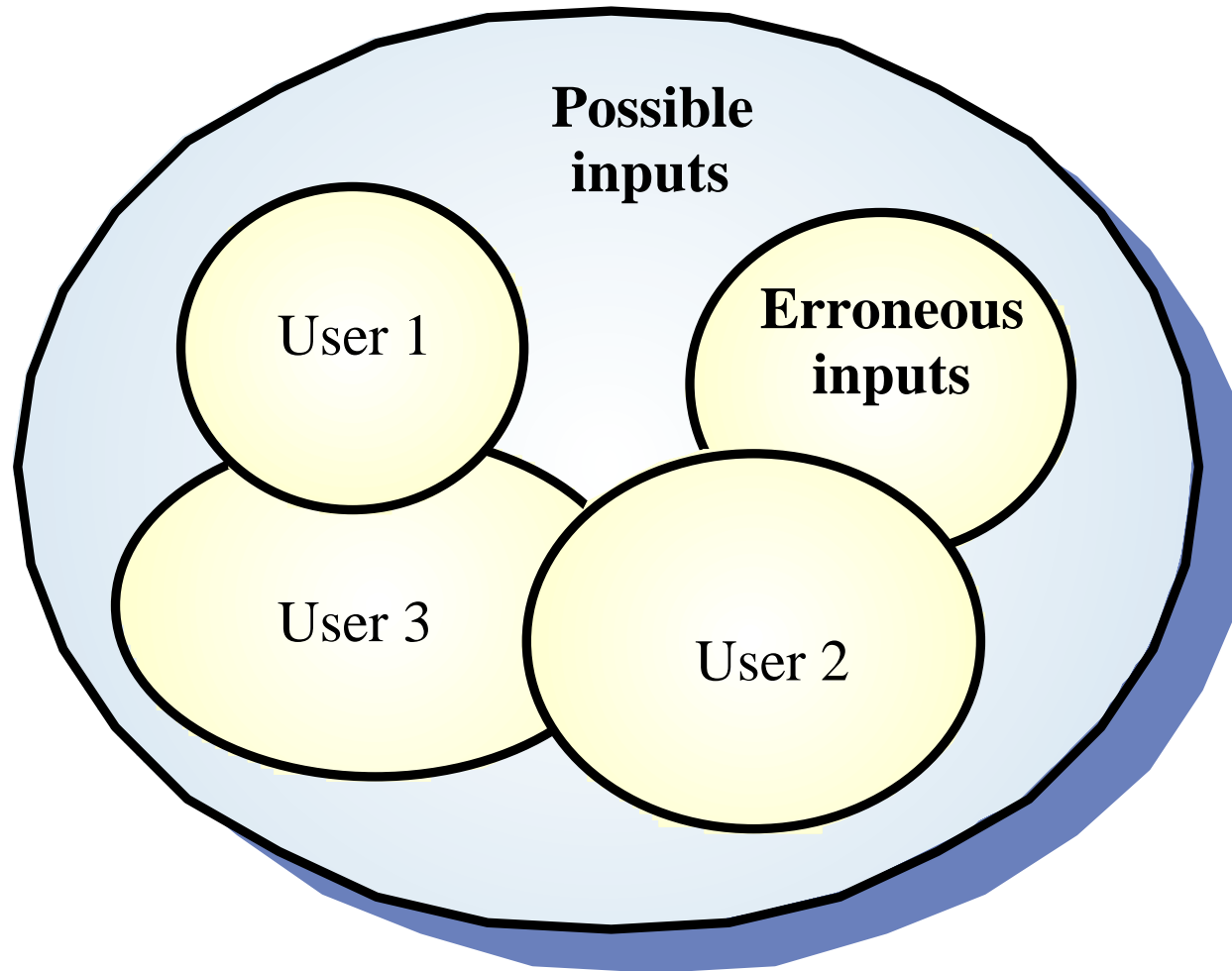
Reliability modelling

- You can model a system as an input-output mapping where some inputs will result in erroneous outputs
- The reliability of the system is the probability that a particular input will lie in the set of inputs that cause erroneous outputs
- Different people will use the system in different ways so this probability is not a static system attribute but depends on the system's environment

Input/output mapping



Reliability perception



Reliability improvement

- Removing $X\%$ of the faults in a system will not necessarily improve the reliability by $X\%$. A study at IBM showed that removing 60% of product defects resulted in a 3% improvement in reliability
- Program defects may be in rarely executed sections of the code so may never be encountered by users. Removing these does not affect the perceived reliability
- A program with known faults may therefore still be seen as reliable by its users

Safety

- Safety is a property of a system that reflects the system's ability to operate, normally or abnormally, without danger of causing human injury or death and without damage to the system's environment
- It is increasingly important to consider software safety as more and more devices incorporate software-based control systems
- Safety requirements are exclusive requirements i.e. they exclude undesirable situations rather than specify required system services

Safety criticality

- Primary safety-critical systems
 - Embedded software systems whose failure can cause the associated hardware to fail and directly threaten people.
- Secondary safety-critical systems
 - Systems whose failure results in faults in other systems which can threaten people
- Discussion here focuses on primary safety-critical systems
 - Secondary safety-critical systems can only be considered on a one-off basis

Safety and reliability

- Safety and reliability are related but distinct
 - In general, reliability and availability are necessary but not sufficient conditions for system safety
- Reliability is concerned with conformance to a given specification and delivery of service
- Safety is concerned with ensuring system cannot cause damage irrespective of whether or not it conforms to its specification

Unsafe reliable systems

- Specification errors
 - If the system specification is incorrect then the system can behave as specified but still cause an accident
- Hardware failures generating spurious inputs
 - Hard to anticipate in the specification
- Context-sensitive commands i.e. issuing the right command at the wrong time
 - Often the result of operator error

Safety terminology

Term	Definition
Accident (or mishap)	An unplanned event or sequence of events which results in human death or injury, damage to property or to the environment. A computer-controlled machine injuring its operator is an example of an accident.
Hazard	A condition with the potential for causing or contributing to an accident. A failure of the sensor which detects an obstacle in front of a machine is an example of a hazard.
Damage	A measure of the loss resulting from a mishap. Damage can range from many people killed as a result of an accident to minor injury or property damage.
Hazard severity	An assessment of the worst possible damage which could result from a particular hazard. Hazard severity can range from catastrophic where many people are killed to minor where only minor damage results
Hazard probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from <i>probable</i> (say 1/100 chance of a hazard occurring) to implausible (no conceivable situations are likely where the hazard could occur).
Risk	This is a measure of the probability that the system will cause an accident. The risk is assessed by considering the hazard probability, the hazard severity and the probability that a hazard will result in an accident.

Safety achievement

- Hazard avoidance
 - The system is designed so that some classes of hazard simply cannot arise.
- Hazard detection and removal
 - The system is designed so that hazards are detected and removed before they result in an accident
- Damage limitation
 - The system includes protection features that minimise the damage that may result from an accident

Normal accidents

- Accidents in complex systems rarely have a single cause as these systems are designed to be resilient to a single point of failure
 - Designing systems so that a single point of failure does not cause an accident is a fundamental principle of safe systems design
- Almost all accidents are a result of combinations of malfunctions
- It is probably the case that anticipating all problem combinations, especially, in software controlled systems is impossible so achieving complete safety is impossible

Security

- The security of a system is a system property that reflects the system's ability to protect itself from accidental or deliberate external attack
- Security is becoming increasingly important as systems are networked so that external access to the system through the Internet is possible
- Security is an essential pre-requisite for availability, reliability and safety

Fundamental security

- If a system is a networked system and is insecure then statements about its reliability and its safety are unreliable
- These statements depend on the executing system and the developed system being the same. However, intrusion can change the executing system and/or its data
- Therefore, the reliability and safety assurance is no longer valid

Security terminology

Term	Definition
Exposure	Possible loss or harm in a computing system
Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm
Attack	An exploitation of a system vulnerability
Threats	Circumstances that have potential to cause loss or harm
Control	A protective measure that reduces a system vulnerability

Damage from insecurity

- Denial of service
 - The system is forced into a state where normal services are unavailable or where service provision is significantly degraded
- Corruption of programs or data
 - The programs or data in the system may be modified in an unauthorised way
- Disclosure of confidential information
 - Information that is managed by the system may be exposed to people who are not authorised to read or use that information

Security assurance

- Vulnerability avoidance
 - The system is designed so that vulnerabilities do not occur. For example, if there is no external network connection then external attack is impossible
- Attack detection and elimination
 - The system is designed so that attacks on vulnerabilities are detected and neutralised before they result in an exposure. For example, virus checkers find and remove viruses before they infect a system
- Exposure limitation
 - The system is designed so that the adverse consequences of a successful attack are minimised. For example, a backup policy allows damaged information to be restored

Key points

- The dependability in a system reflects the user's trust in that system
- The availability of a system is the probability that it will be available to deliver services when requested
- The reliability of a system is the probability that system services will be delivered as specified
- Reliability and availability are generally seen as necessary but not sufficient conditions for safety and security

Key points

- Reliability is related to the probability of an error occurring in operational use. A system with known faults may be reliable
- Safety is a system attribute that reflects the system's ability to operate without threatening people or the environment
- Security is a system attribute that reflects the system's ability to protect itself from external attack

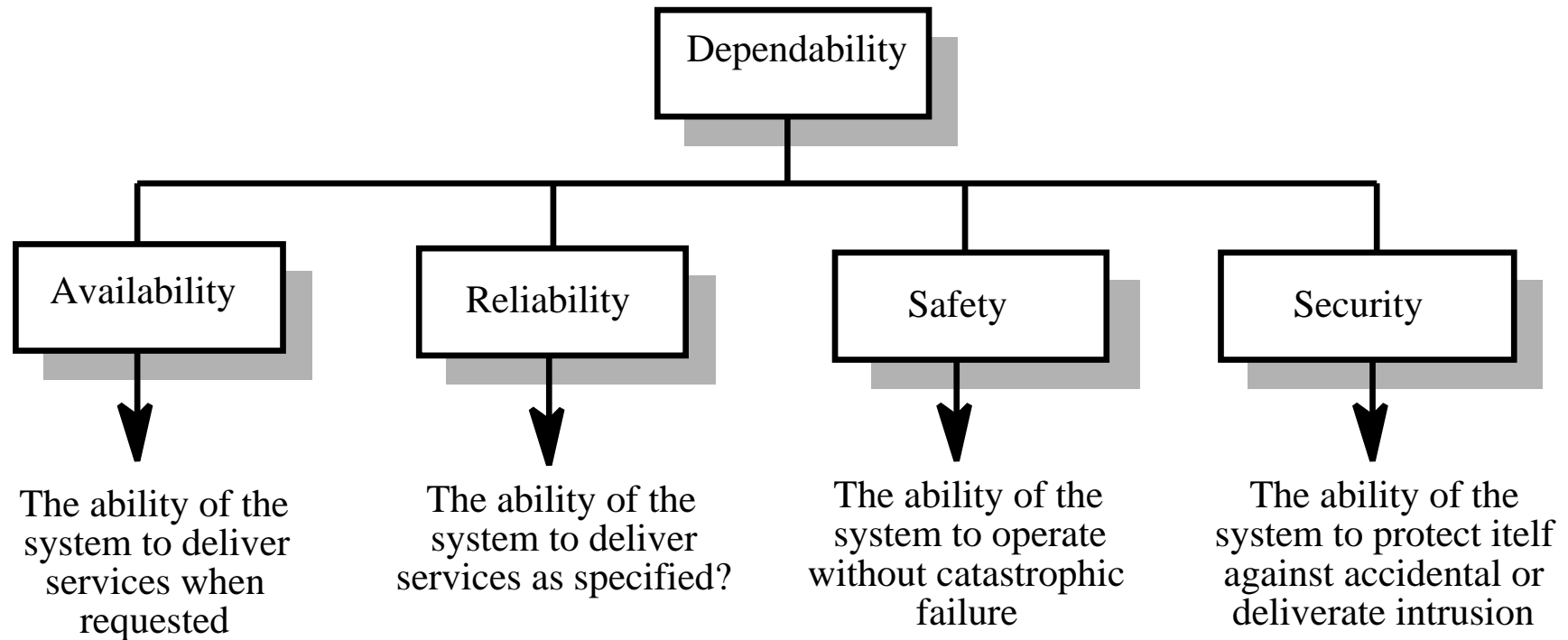
Dependability

- The extent to which a critical system is trusted by its users

The concept of dependability

- For critical systems, it is usually the case that the most important system property is the dependability of the system
- The dependability of a system reflects the user's degree of trust in that system. It reflects the extent of the user's confidence that it will operate as users expect and that it will not 'fail' in normal use
- Usefulness and trustworthiness are not the same thing. A system does not have to be trusted to be useful

Dimensions of dependability



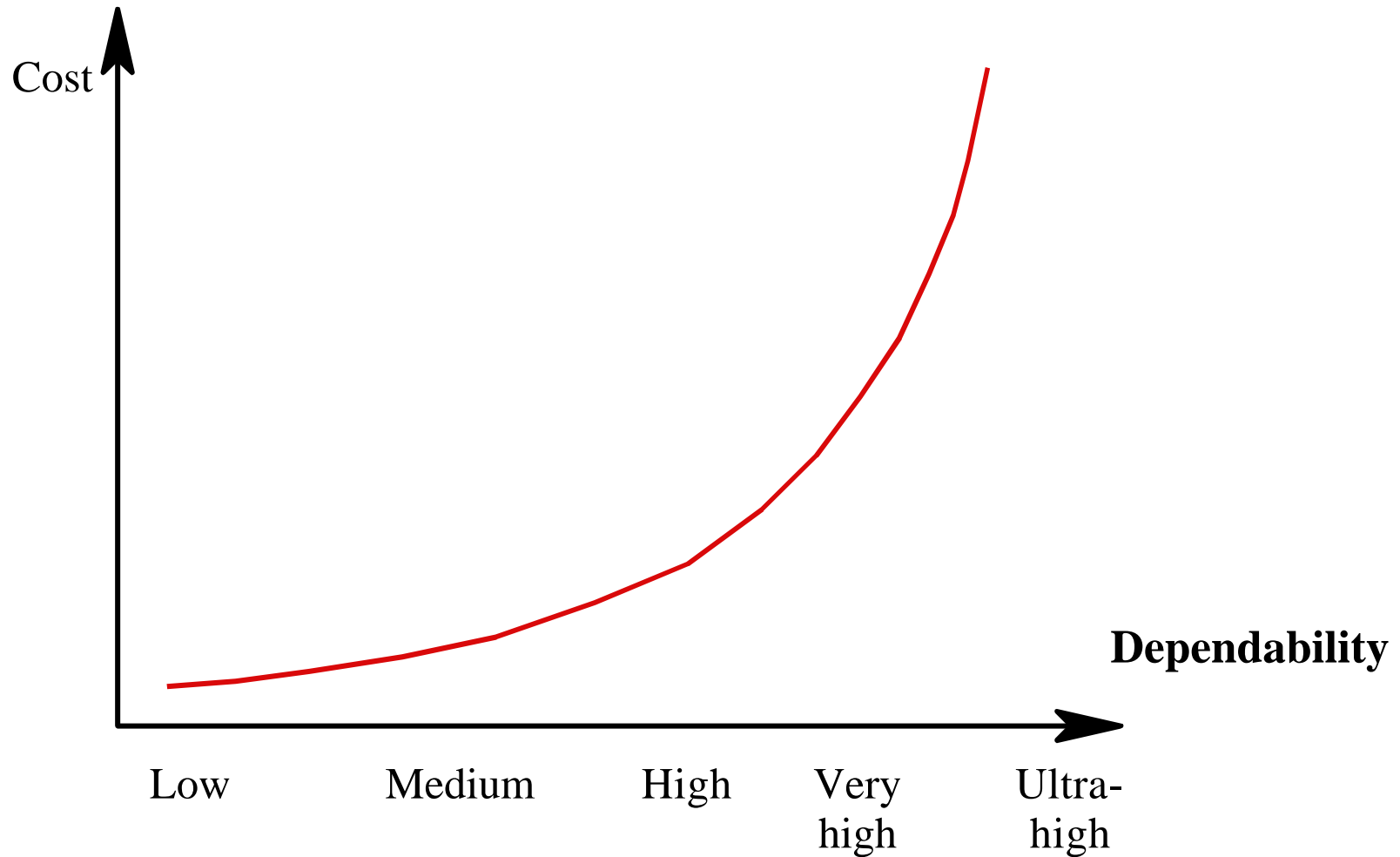
Maintainability

- A system attribute which is concerned with the ease of repairing the system after a failure has been discovered or changing the system to include new features
- Very important for critical systems as faults are often introduced into a system because of maintenance problems
- Maintainability is distinct from other dimensions of dependability because it is a static and not a dynamic system attribute. I do not cover it in this course.

Survivability

- The ability of a system to continue to deliver its services to users in the face of deliberate or accidental attack
- This is an increasingly important attribute for distributed systems whose security can be compromised
- Survivability subsumes the notion of resilience - the ability of a system to continue in operation in spite of component failures

Costs of increasing dependability



Dependability costs

- Dependability costs tend to increase exponentially as increasing levels of dependability are required
- There are two reasons for this
 - The use of more expensive development techniques and hardware that are required to achieve the higher levels of dependability
 - The increased testing and system validation that is required to convince the system client that the required levels of dependability have been achieved

Dependability vs performance

- Untrustworthy systems may be rejected by their users
- System failure costs may be very high
- It is very difficult to tune systems to make them more dependable
- It may be possible to compensate for poor performance
- Untrustworthy systems may cause loss of valuable information

Dependability economics

- Because of very high costs of dependability achievement, it may be more cost effective to accept untrustworthy systems and pay for failure costs
- However, this depends on social and political factors. A reputation for products that can't be trusted may lose future business
- Depends on system type - for business systems in particular, modest levels of dependability may be adequate

Availability and reliability

- Reliability
 - The probability of failure-free system operation over a specified time in a given environment for a given purpose
- Availability
 - The probability that a system, at a point in time, will be operational and able to deliver the requested services
- Both of these attributes can be expressed quantitatively

Availability and reliability

- It is sometimes possible to subsume system availability under system reliability
 - Obviously if a system is unavailable it is not delivering the specified system services
- However, it is possible to have systems with low reliability that must be available. So long as system failures can be repaired quickly and do not damage data, low reliability may not be a problem
- Availability takes repair time into account

Reliability terminology

Term	Description
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users
System error	Erroneous system behaviour where the behaviour of the system does not conform to its specification.
System fault	An incorrect system state i.e. a system state that is unexpected by the designers of the system.
Human error or mistake	Human behaviour that results in the introduction of faults into a system.

Faults and failures

- Failures are usually a result of system errors that are derived from faults in the system
- However, faults do not necessarily result in system errors
 - The faulty system state may be transient and ‘corrected’ before an error arises
- Errors do not necessarily lead to system failures
 - The error can be corrected by built-in error detection and recovery
 - The failure can be protected against by built-in protection facilities. These may, for example, protect system resources from system errors

Perceptions of reliability

- The formal definition of reliability does not always reflect the user's perception of a system's reliability
 - The assumptions that are made about the environment where a system will be used may be incorrect
 - Usage of a system in an office environment is likely to be quite different from usage of the same system in a university environment
 - The consequences of system failures affects the perception of reliability
 - Unreliable windscreen wipers in a car may be irrelevant in a dry climate
 - Failures that have serious consequences (such as an engine breakdown in a car) are given greater weight by users than failures that are inconvenient

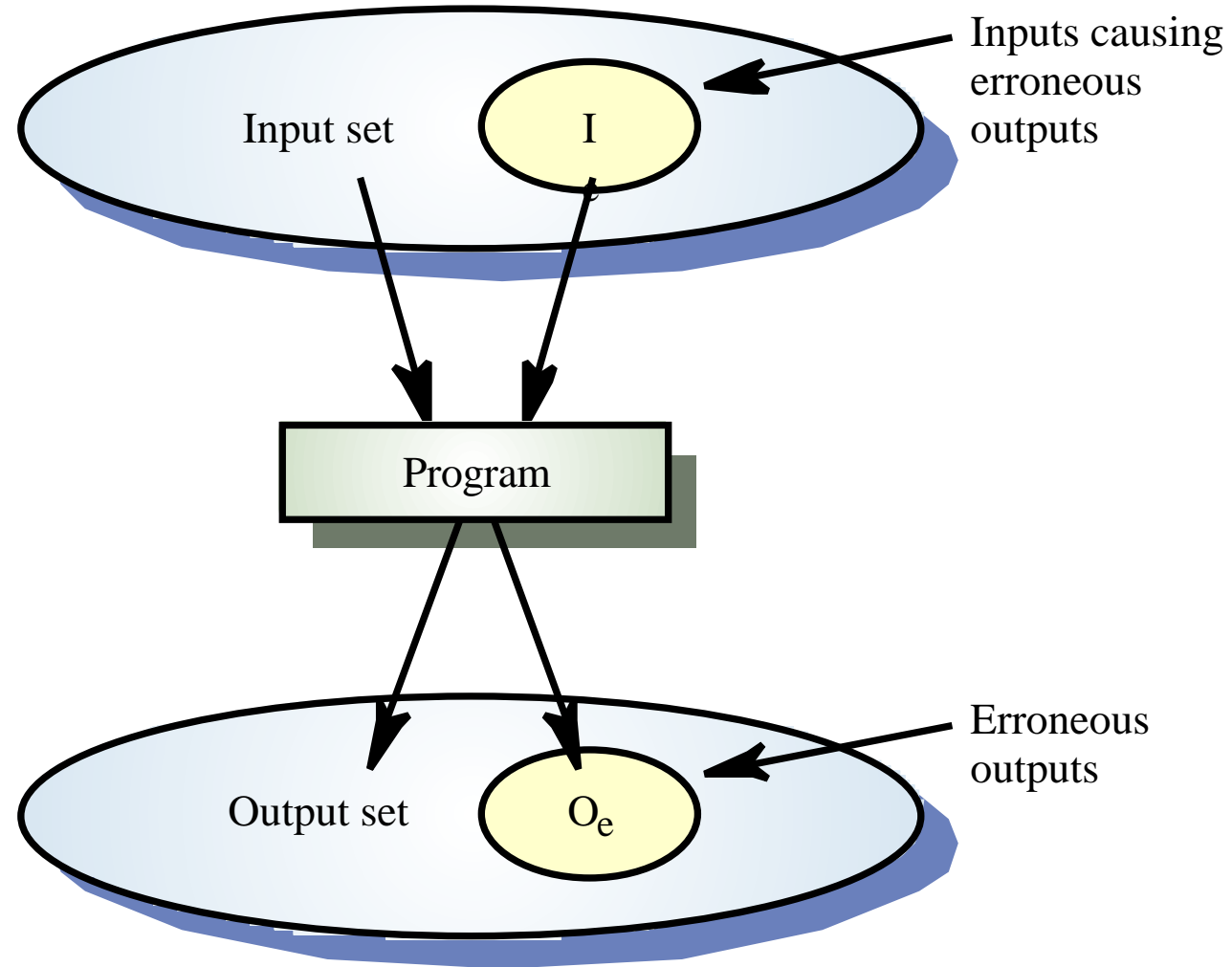
Reliability achievement

- Fault avoidance
 - Development techniques are used that either minimise the possibility of mistakes or trap mistakes before they result in the introduction of system faults
- Fault detection and removal
 - Verification and validation techniques that increase the probability of detecting and correcting errors before the system goes into service are used
- Fault tolerance
 - Run-time techniques are used to ensure that system faults do not result in system errors and/or that system errors do not lead to system failures

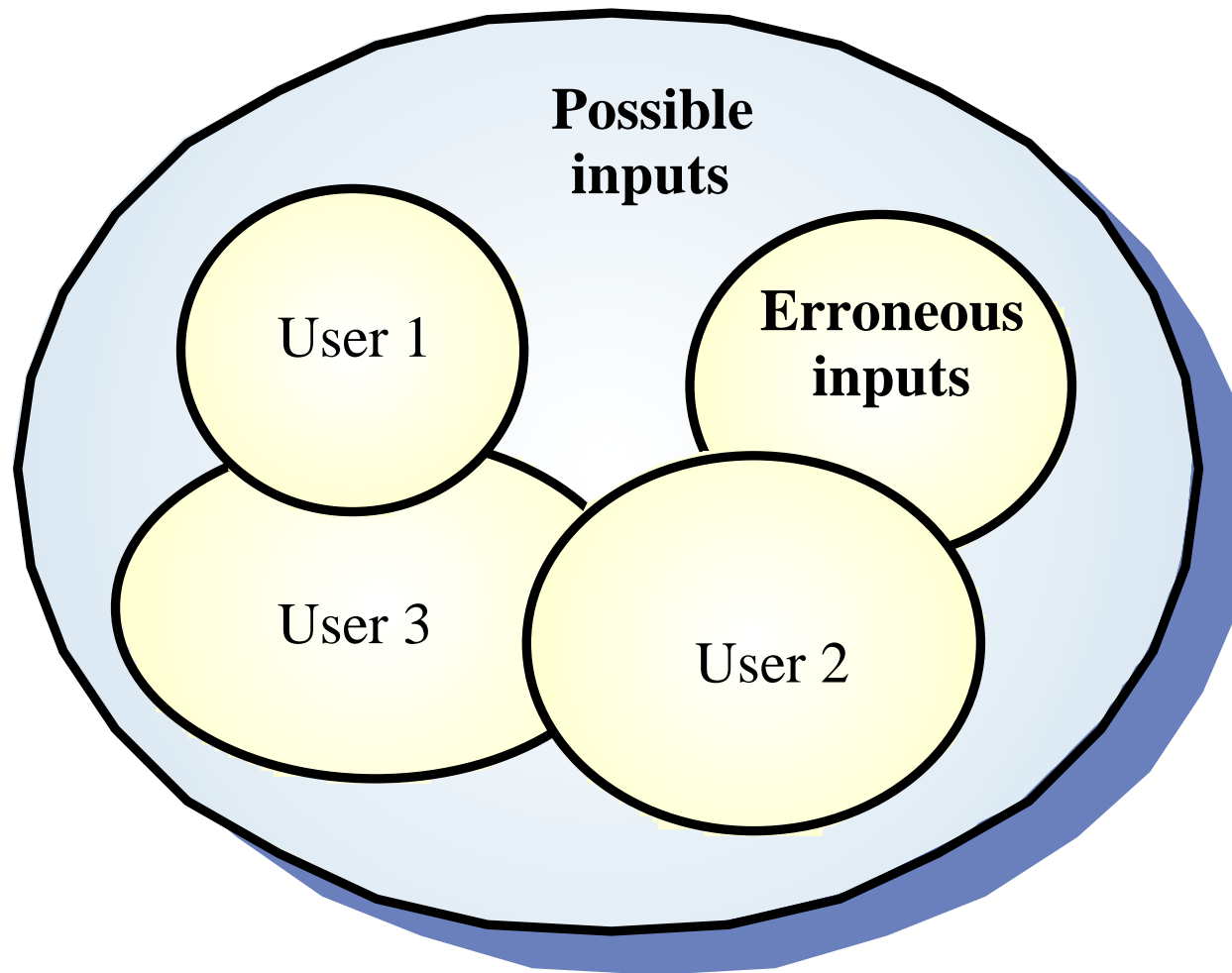
Reliability modelling

- You can model a system as an input-output mapping where some inputs will result in erroneous outputs
- The reliability of the system is the probability that a particular input will lie in the set of inputs that cause erroneous outputs
- Different people will use the system in different ways so this probability is not a static system attribute but depends on the system's environment

Input/output mapping



Reliability perception



Reliability improvement

- Removing $X\%$ of the faults in a system will not necessarily improve the reliability by $X\%$. A study at IBM showed that removing 60% of product defects resulted in a 3% improvement in reliability
- Program defects may be in rarely executed sections of the code so may never be encountered by users. Removing these does not affect the perceived reliability
- A program with known faults may therefore still be seen as reliable by its users

Safety

- Safety is a property of a system that reflects the system's ability to operate, normally or abnormally, without danger of causing human injury or death and without damage to the system's environment
- It is increasingly important to consider software safety as more and more devices incorporate software-based control systems
- Safety requirements are exclusive requirements i.e. they exclude undesirable situations rather than specify required system services

Safety criticality

- Primary safety-critical systems
 - Embedded software systems whose failure can cause the associated hardware to fail and directly threaten people.
- Secondary safety-critical systems
 - Systems whose failure results in faults in other systems which can threaten people
- Discussion here focuses on primary safety-critical systems
 - Secondary safety-critical systems can only be considered on a one-off basis

Safety and reliability

- Safety and reliability are related but distinct
 - In general, reliability and availability are necessary but not sufficient conditions for system safety
- Reliability is concerned with conformance to a given specification and delivery of service
- Safety is concerned with ensuring system cannot cause damage irrespective of whether or not it conforms to its specification

Unsafe reliable systems

- Specification errors
 - If the system specification is incorrect then the system can behave as specified but still cause an accident
- Hardware failures generating spurious inputs
 - Hard to anticipate in the specification
- Context-sensitive commands i.e. issuing the right command at the wrong time
 - Often the result of operator error

Safety terminology

Term	Definition
Accident (or mishap)	An unplanned event or sequence of events which results in human death or injury, damage to property or to the environment. A computer-controlled machine injuring its operator is an example of an accident.
Hazard	A condition with the potential for causing or contributing to an accident. A failure of the sensor which detects an obstacle in front of a machine is an example of a hazard.
Damage	A measure of the loss resulting from a mishap. Damage can range from many people killed as a result of an accident to minor injury or property damage.
Hazard severity	An assessment of the worst possible damage which could result from a particular hazard. Hazard severity can range from catastrophic where many people are killed to minor where only minor damage results
Hazard probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from <i>probable</i> (say 1/100 chance of a hazard occurring) to implausible (no conceivable situations are likely where the hazard could occur).
Risk	This is a measure of the probability that the system will cause an accident. The risk is assessed by considering the hazard probability, the hazard severity and the probability that a hazard will result in an accident.

Safety achievement

- Hazard avoidance
 - The system is designed so that some classes of hazard simply cannot arise.
- Hazard detection and removal
 - The system is designed so that hazards are detected and removed before they result in an accident
- Damage limitation
 - The system includes protection features that minimise the damage that may result from an accident

Normal accidents

- Accidents in complex systems rarely have a single cause as these systems are designed to be resilient to a single point of failure
 - Designing systems so that a single point of failure does not cause an accident is a fundamental principle of safe systems design
- Almost all accidents are a result of combinations of malfunctions
- It is probably the case that anticipating all problem combinations, especially, in software controlled systems is impossible so achieving complete safety is impossible

Security

- The security of a system is a system property that reflects the system's ability to protect itself from accidental or deliberate external attack
- Security is becoming increasingly important as systems are networked so that external access to the system through the Internet is possible
- Security is an essential pre-requisite for availability, reliability and safety

Fundamental security

- If a system is a networked system and is insecure then statements about its reliability and its safety are unreliable
- These statements depend on the executing system and the developed system being the same. However, intrusion can change the executing system and/or its data
- Therefore, the reliability and safety assurance is no longer valid

Security terminology

Term	Definition
Exposure	Possible loss or harm in a computing system
Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm
Attack	An exploitation of a system vulnerability
Threats	Circumstances that have potential to cause loss or harm
Control	A protective measure that reduces a system vulnerability

Damage from insecurity

- Denial of service
 - The system is forced into a state where normal services are unavailable or where service provision is significantly degraded
- Corruption of programs or data
 - The programs or data in the system may be modified in an unauthorised way
- Disclosure of confidential information
 - Information that is managed by the system may be exposed to people who are not authorised to read or use that information

Security assurance

- Vulnerability avoidance
 - The system is designed so that vulnerabilities do not occur. For example, if there is no external network connection then external attack is impossible
- Attack detection and elimination
 - The system is designed so that attacks on vulnerabilities are detected and neutralised before they result in an exposure. For example, virus checkers find and remove viruses before they infect a system
- Exposure limitation
 - The system is designed so that the adverse consequences of a successful attack are minimised. For example, a backup policy allows damaged information to be restored

Key points

- The dependability in a system reflects the user's trust in that system
- The availability of a system is the probability that it will be available to deliver services when requested
- The reliability of a system is the probability that system services will be delivered as specified
- Reliability and availability are generally seen as necessary but not sufficient conditions for safety and security

Key points

- Reliability is related to the probability of an error occurring in operational use. A system with known faults may be reliable
- Safety is a system attribute that reflects the system's ability to operate without threatening people or the environment
- Security is a system attribute that reflects the system's ability to protect itself from external attack

Dependable Systems Specification

- Processes and techniques for developing a specification for system availability, reliability, safety and security

Functional and non-functional requirements

- System functional requirements may be generated to define error checking and recovery facilities and features that provide protection against system failures.
- Non-functional requirements may be generated to specify the required reliability and availability of the system.

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?

System reliability engineering

- Sub-discipline of systems engineering that is concerned with making judgements on system reliability
- It takes into account the probabilities of failure of different components in the system and their combinations
 - Consider a system with 2 components A and B where the probability of failure of A is $P(A)$ and the probability of failure of B is $P(B)$.

Failure probabilities

- If there are 2 components and the operation of the system depends on both of them then the probability of system failure is
 - $P(S) = P(A) + P(B)$
- Therefore, as the number of components increase then the probability of system failure increases
- If components are replicated then the probability of failure is
 - $P(S) = P(A)^n$ (all components must fail)

Functional reliability requirements

- A predefined range for all values that are input by the operator shall be defined and the system shall check that all operator inputs fall within this predefined range.
- The system shall check all disks for bad blocks when it is initialised.
- The system must use N-version programming to implement the braking control system.
- The system must be implemented in a safe subset of Ada and checked using static analysis

Non-functional reliability specification

- The required level of system reliability required should be expressed in quantitatively
- Reliability is a dynamic system attribute-
reliability specifications related to the source code are meaningless.
 - No more than N faults/1000 lines.
 - This is only useful for a post-delivery process analysis where you are trying to assess how good your development techniques are.
- An appropriate reliability metric should be chosen to specify the overall system reliability

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. For example, 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. For example, 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. For example, an MTTF of 500 means that 1 failure can be expected every 500 time units.
MTTR Mean time to repair	The average time between a system failure and the return of that system to service.
AVAIL Availability	The probability that the system is available for use at a given time. For example, an availability of 0.998 means that in every 1000 time units, the system is likely to be available for 998 of these.

Availability

- Measure of the fraction of the time that the system is available for use
- Takes repair and restart time into account
- Availability of 0.998 means software is available for 998 out of 1000 time units
- Relevant for non-stop, continuously running systems
 - telephone switching systems, railway signalling systems

Probability of failure on demand

- This is the probability that the system will fail when a service request is made. Useful when demands for service are intermittent and relatively infrequent
- Appropriate for protection systems where services are demanded occasionally and where there are serious consequence if the service is not delivered
- Relevant for many safety-critical systems with exception management components
 - Emergency shutdown system in a chemical plant

Rate of fault occurrence (ROCOF)

- Reflects the rate of occurrence of failure in the system
- ROCOF of 0.002 means 2 failures are likely in each 1000 operational time units e.g. 2 failures per 1000 hours of operation
- Relevant for operating systems, transaction processing systems where the system has to process a large number of similar requests that are relatively frequent
 - Credit card processing system, airline booking system

Mean time to failure

- Measure of the time between observed failures of the system. Is the reciprocal of RCOF for stable systems
- MTTF of 500 means that the mean time between failures is 500 time units
- Relevant for systems with long transactions i.e. where system processing takes a long time.
MTTF should be longer than transaction length
 - Computer-aided design systems where a designer will work on a design for several hours, word processor systems

Failure consequences

- Reliability measurements do NOT take the consequences of failure into account
- Transient faults may have no real consequences but other faults may cause data loss or corruption and loss of system service
- May be necessary to identify different failure classes and use different metrics for each of these. The reliability specification must be structured.

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

Examples of a reliability spec.

Failure class	Example	Reliability metric
Permanent, non-corrupting.	The system fails to operate with any card which 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 which is input.	POFOD 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

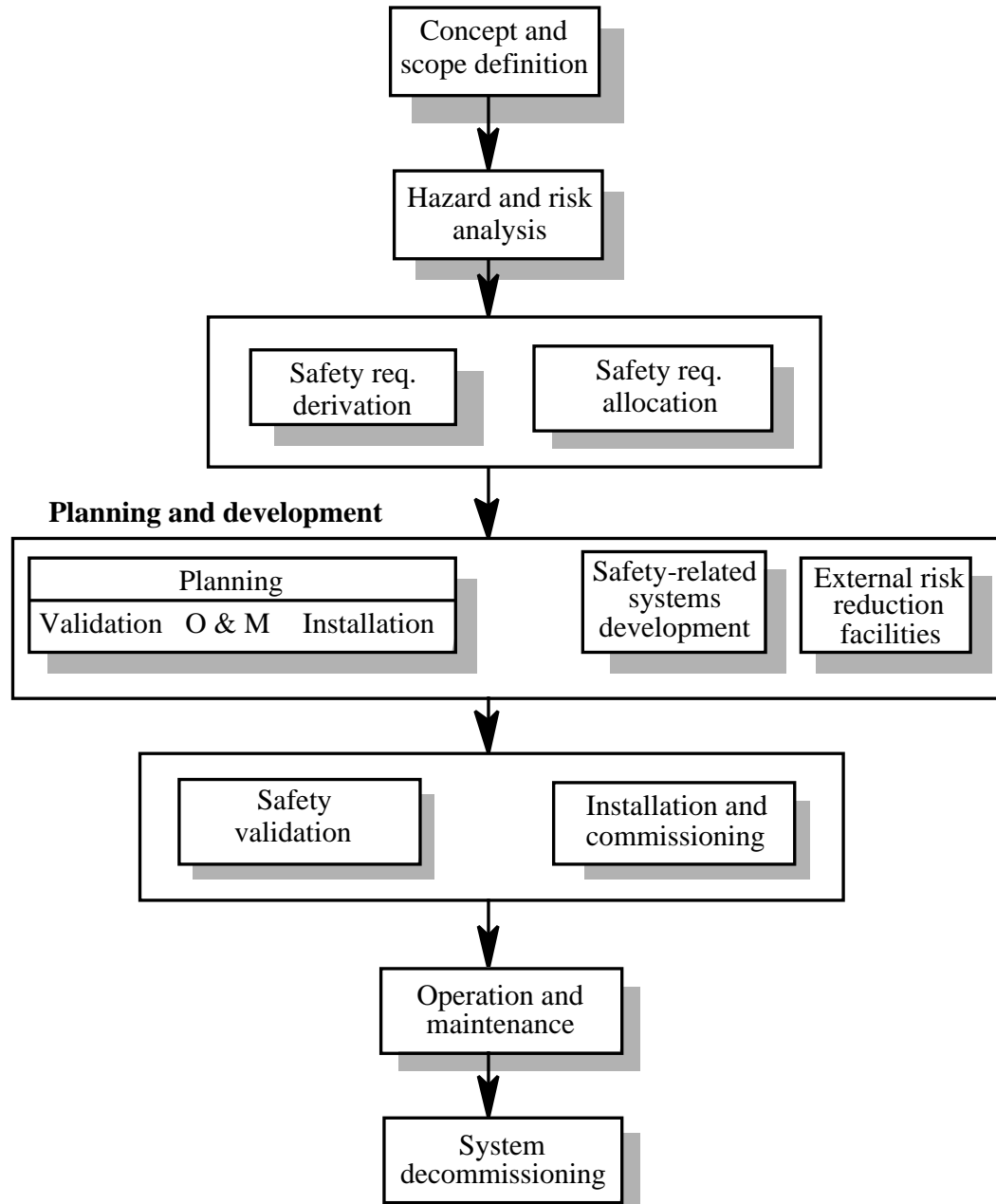
- It is impossible to empirically validate very high reliability specifications
- No database corruptions means POFOD of less than 1 in 200 million
- If a transaction takes 1 second, then simulating one day's transactions takes 3.5 days
- It would take longer than the system's lifetime to test it for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

Key points

- There are both functional and non-functional dependability requirements
- Non-functional availability and reliability requirements should be specified quantitatively
- Metrics that may be used are AVAIL, POFOD, ROCOF and MTTF
- When deriving a reliability specification, the consequences of different types of fault should be taken into account

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
- 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

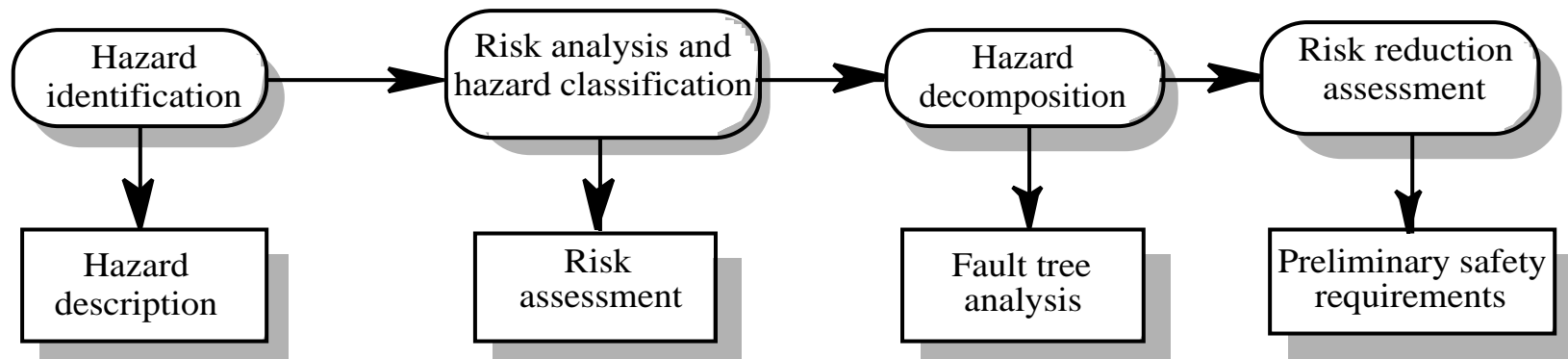


The safety life-cycle

Safety processes

- Hazard and risk analysis
 - Assess the hazards and the risks of damage associated with the system
- Safety requirements specification
 - Specify a set of safety requirements which apply to the system
- Designation of safety-critical systems
 - Identify the sub-systems whose incorrect operation may compromise system safety. Ideally, these should be as small a part as possible of the whole system.
- Safety validation
 - Check the overall system safety

Hazard and risk analysis



Hazard and risk analysis

- Identification of hazards which can arise which compromise the safety of the system and assessing the risks associated with these hazards
- Structured into various classes of hazard analysis and carried out throughout software process from specification to implementation
- A risk analysis should be carried out and documented for each identified hazard and actions taken to ensure the most serious/likely hazards do not result in accidents

Hazard analysis stages

- Hazard identification
 - Identify potential hazards which may arise
- Risk analysis and hazard classification
 - Assess the risk associated with each hazard
- Hazard decomposition
 - Decompose hazards to discover their potential root causes
- Risk reduction assessment
 - Define how each hazard must be taken into account when the system is designed

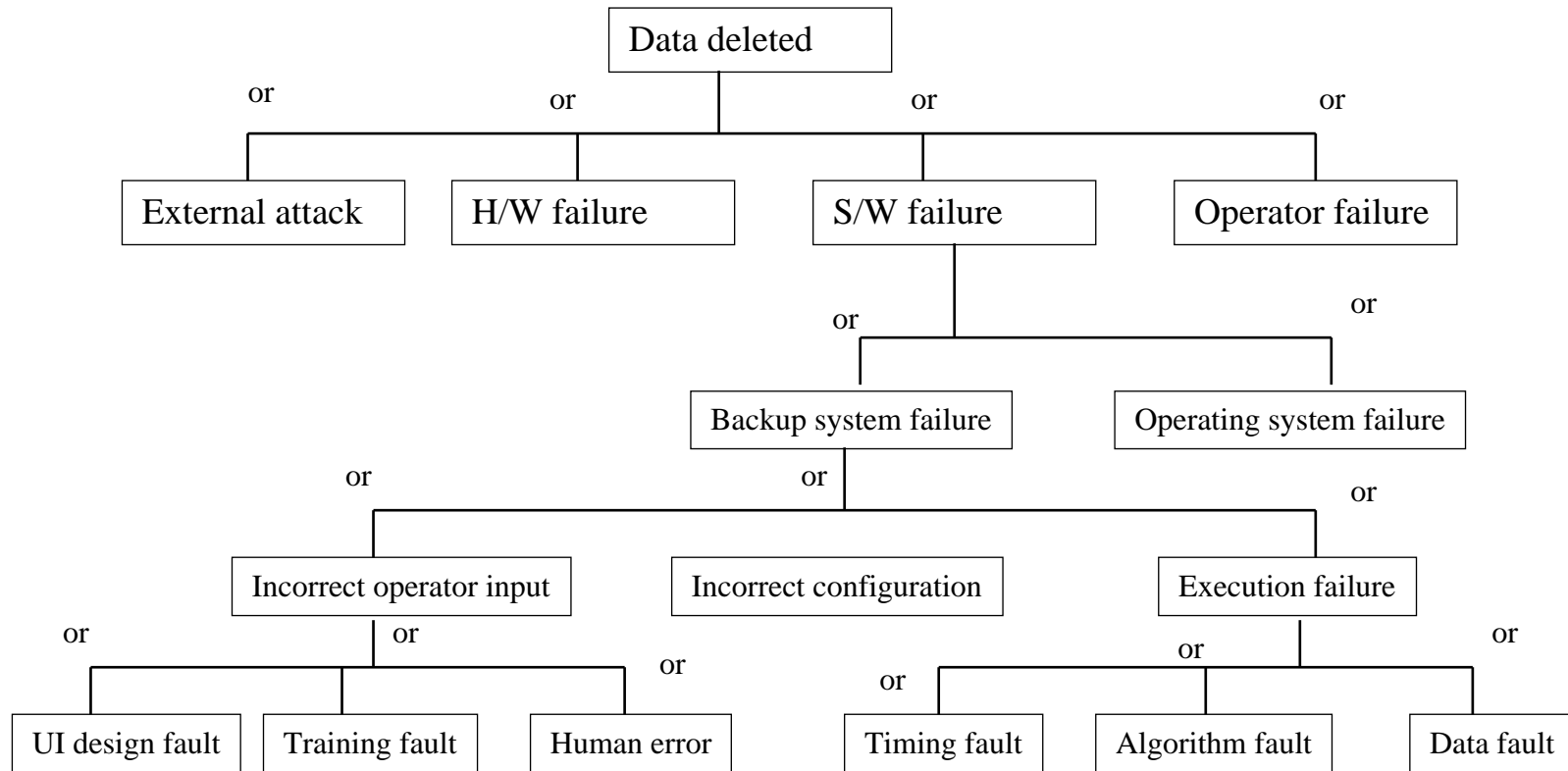
Fault-tree analysis

- Method of hazard analysis which starts with an identified fault and works backward to the causes of the fault.
- Can be used at all stages of hazard analysis from preliminary analysis through to detailed software checking
- Top-down hazard analysis method. May be combined with bottom-up methods which start with system failures and lead to hazards

Fault- tree analysis

- Identify hazard
- Identify potential causes of the hazard. Usually there will be a number of alternative causes. Link these on the fault-tree with 'or' or 'and' symbols
- Continue process until root causes are identified
- Consider the following example which considers how data might be lost in some system where a backup process is running

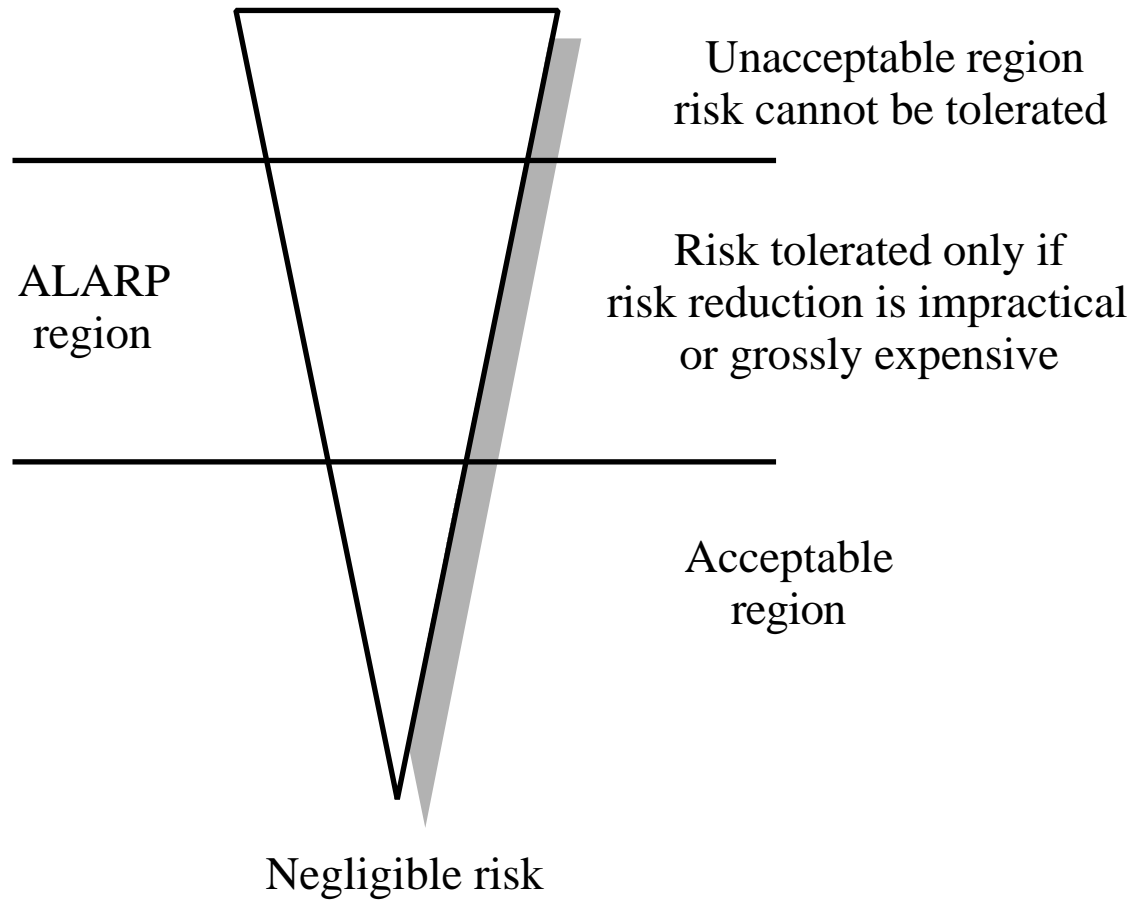
Fault tree



Risk assessment

- Assesses hazard severity, hazard probability and accident probability
- Outcome of risk assessment is a statement of acceptability
 - Intolerable. Must never arise or result in an accident
 - As low as reasonably practical(ALARP) Must minimise possibility of hazard given cost and schedule constraints
 - Acceptable. Consequences of hazard are acceptable and no extra costs should be incurred to reduce hazard probability

Levels of risk



Risk acceptability

- The acceptability of a risk is determined by human, social and political considerations
- In most societies, the boundaries between the regions are pushed upwards with time i.e. society is less willing to accept risk
 - For example, the costs of cleaning up pollution may be less than the costs of preventing it but this may not be socially acceptable
- Risk assessment is subjective
 - Risks are identified as probable, unlikely, etc. This depends on who is making the assessment

Risk reduction

- System should be specified so that hazards do not arise or result in an accident
- Hazard avoidance
 - The system should be designed so that the hazard can never arise during correct system operation
- Hazard detection and removal
 - The system should be designed so that hazards are detected and neutralised before they result in an accident
- Damage limitation
 - The system is designed in such a way that the consequences of an accident are minimised

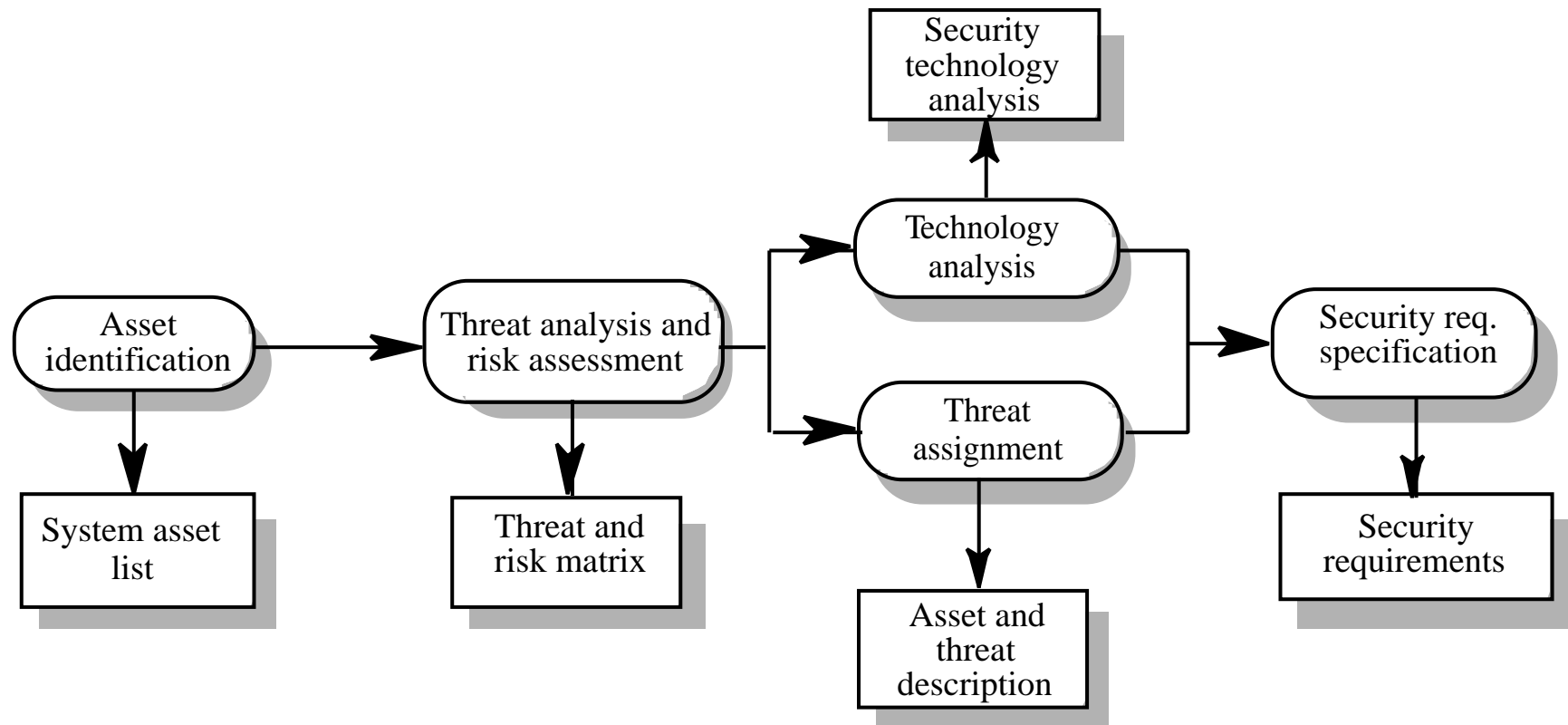
Specifying forbidden behaviour

- The system shall not allow users to modify access permissions on any files that they have not created (security)
- The system shall not allow reverse thrust mode to be selected when the aircraft is in flight (safety)
- The system shall not allow the simultaneous activation of more than three alarm signals (safety)

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
 - Generic threats rather than system specific hazards
 - Mature security technology (encryption, etc.). However, there are problems in transferring this into general use

The security specification process



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 identified the security technologies that may be used to protect against different threats to the system.

Key points

- Hazard analysis is a key activity in the safety specification process.
- Fault-tree analysis is a technique which can be used in the hazard analysis process.
- Risk analysis is the process of assessing the likelihood that a hazard will result in an accident. Risk analysis identifies critical hazards and classifies risks according to their seriousness.
- To specify security requirements, you should identify the assets that are to be protected and define how security techniques should be used to protect them.

Dependable software development

- Programming techniques for building dependable software systems.

Software dependability

- In general, software customers expect all software to be dependable. However, for non-critical applications, they may be willing to accept some system failures
- Some applications, however, have very high dependability requirements and special programming techniques must be used to achieve this

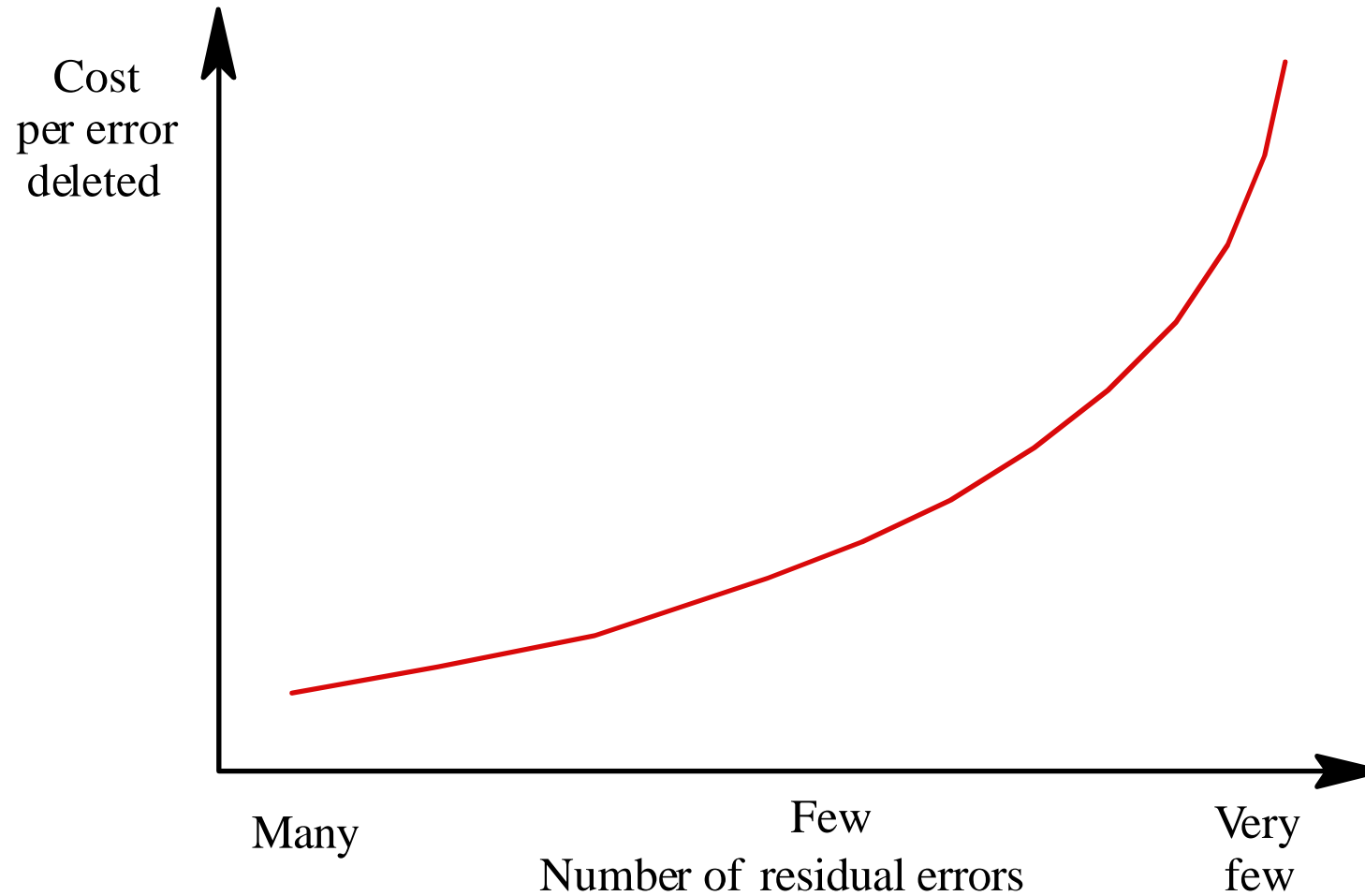
Dependability achievement

- Fault avoidance
 - The software is developed in such a way that human error is avoided and thus system faults are minimised
 - The development process is organised so that faults in the software are detected and repaired before delivery to the customer
- Fault tolerance
 - The software is designed so that faults in the delivered software do not result in system failure

Fault minimisation

- Current methods of software engineering now allow for the production of fault-free software.
- Fault-free software means software which conforms to its specification. It does NOT mean software which will always perform correctly as there may be specification errors.
- The cost of producing fault free software is very high. It is only cost-effective in exceptional situations. May be cheaper to accept software faults

Fault removal costs



Fault-free software development

- Needs a precise (preferably formal) specification.
- Requires an organizational commitment to quality.
- Information hiding and encapsulation in software design is essential
- A programming language with strict typing and run-time checking should be used
- Error-prone constructs should be avoided
- Dependable and repeatable development process

Structured programming

- First discussed in the 1970's
- Programming without gotos
- While loops and if statements as the only control statements.
- Top-down design.
- Important because it promoted thought and discussion about programming
- Leads to programs that are easier to read and understand

Error-prone constructs

- Floating-point numbers
 - Inherently imprecise. The imprecision may lead to invalid comparisons
- Pointers
 - Pointers referring to the wrong memory areas can corrupt data. Aliasing can make programs difficult to understand and change
- Dynamic memory allocation
 - Run-time allocation can cause memory overflow
- Parallelism
 - Can result in subtle timing errors because of unforeseen interaction between parallel processes

Error-prone constructs

- Recursion
 - Errors in recursion can cause memory overflow
- Interrupts
 - Interrupts can cause a critical operation to be terminated and make a program difficult to understand. they are comparable to goto statements.
- Inheritance
 - Code is not localised. This can result in unexpected behaviour when changes are made and problems of understanding
- These constructs don't have to be avoided but they must be used with great care.

Information hiding

- Information should only be exposed to those parts of the program which need to access it. This involves the creation of objects or abstract data types which maintain state and operations on that state
- This avoids faults for three reasons:
 - the probability of accidental corruption of information
 - the information is surrounded by ‘firewalls’ so that problems are less likely to spread to other parts of the program
 - as all information is localised, the programmer is less likely to make errors and reviewers are more likely to find errors

A queue specification in Java

```
interface Queue {  
  
    public void put (Object o) ;  
    public void remove (Object o) ;  
    public int size () ;  
  
} //Queue
```

Signal declaration in Java

```
class Signal {  
  
    public final int red = 1 ;  
    public final int amber = 2 ;  
    public final int green = 3 ;  
  
    ... other declarations here ...  
}
```

Reliable software processes

- To ensure a minimal number of software faults, it is important to have a well-defined, repeatable software process
- A well-defined repeatable process is one that does not depend entirely on individual skills; rather can be enacted by different people
- For fault minimisation, it is clear that the process activities should include significant verification and validation

Process validation activities

- Requirements inspections
- Requirements management
- Model checking
- Design and code inspection
- Static analysis
- Test planning and management
- Configuration management is also essential

Fault tolerance

- In critical situations, software systems must be fault tolerant. Fault tolerance is required where there are high availability requirements or where system failure costs are very high..
- Fault tolerance means that the system can continue in operation in spite of software failure
- Even if the system seems to be fault-free, it must also be fault tolerant as there may be specification errors or the validation may be incorrect

Fault tolerance actions

- Fault detection
 - The system must detect that a fault (an incorrect system state) has occurred.
- Damage assessment
 - The parts of the system state affected by the fault must be detected.
- Fault recovery
 - The system must restore its state to a known safe state.
- Fault repair
 - The system may be modified to prevent recurrence of the fault. As many software faults are transitory, this is often unnecessary.

Approaches to fault tolerance

- Defensive programming
 - Programmers assume that there may be faults in the code of the system and incorporate redundant code to check the state after modifications to ensure that it is consistent.
 - Fault-tolerant architectures
 - Hardware and software system architectures that support hardware and software redundancy and a fault tolerance controller that detects problems and supports fault recovery
 - These are complementary rather than opposing techniques

Exception management

- A program exception is an error or some unexpected event such as a power failure.
- Exception handling constructs allow for such events to be handled without the need for continual status checking to detect exceptions.
- Using normal control constructs to detect exceptions in a sequence of nested procedure calls needs many additional statements to be added to the program and adds a significant timing overhead.

Exceptions in Java

```
class SensorFailureException extends Exception {
    SensorFailureException (String msg) {
        super (msg) ;
        Alarm.activate (msg) ;
    }
} // SensorFailureException

class Sensor {
    int readVal () throws SensorFailureException {
        try {
            int theValue = DeviceIO.readInteger () ;
            if (theValue < 0)
                throw new SensorFailureException ("Sensor failure") ;
            return theValue ;
        }
        catch (deviceIOException e)
            { throw new SensorFailureException (" Sensor read error ") ; }
    } // readVal
} // Sensor
```

Programming with exceptions

- Exceptions can be used as a normal programming technique and not just as a way of recovering from faults
- Consider the example of a temperature control system for a refrigeration unit

A temperature controller

- Controls a freezer and keeps temperature within a specified range
- Switches a refrigerant pump on and off
- Sets of an alarm is the maximum allowed temperature is exceeded
- Uses exceptions as a normal programming technique


```

class FreezerController {
    Sensor tempSensor = new Sensor () ;
    Dial tempDial = new Dial () ;
    float freezerTemp = tempSensor.readVal () ;
    final float dangerTemp = (float) -18.0 ;
    final long coolingTime = (long) 200000.0 ;
    public void run ( ) throws InterruptedException {
        try {
            Pump.switchIt (Pump.on) ;
            do { if (freezerTemp > tempDial.setting ())
                if (Pump.status == Pump.off)
                {
                    Pump.switchIt (Pump.on) ;
                    Thread.sleep (coolingTime) ;
                } else
                if (Pump.status == Pump.on)
                    Pump.switchIt (Pump.off) ;
                if (freezerTemp > dangerTemp)
                    throw new FreezerTooHotException () ;
                freezerTemp = tempSensor.readVal () ;
            } while (true) ;
        } // try block
        catch (FreezerTooHotException f)
        {
            Alarm.activate ( ) ; }
        catch (InterruptedException e)
        {
            System.out.println ("Thread exception") ;
            throw new InterruptedException ( ) ;
        }
    } //run
} // FreezerController

```

Freezer controller (Java)

Fault detection

- Languages such as Java and Ada have a strict type system that allows many errors to be trapped at compile-time
- However, some classes of error can only be discovered at run-time
- Fault detection involves detecting an erroneous system state and throwing an exception to manage the detected fault

Fault detection

- Preventative fault detection
 - The fault detection mechanism is initiated before the state change is committed. If an erroneous state is detected, the change is not made
- Retrospective fault detection
 - The fault detection mechanism is initiated after the system state has been changed. Used when an incorrect sequence of correct actions leads to an erroneous state or when preventative fault detection involves too much overhead

Type system extension

- Preventative fault detection really involves extending the type system by including additional constraints as part of the type definition
- These constraints are implemented by defining basic operations within a class definition

```

class PositiveEvenInteger {
    int val = 0 ;

    PositiveEvenInteger (int n) throws NumericException
    {
        if (n < 0 | n%2 == 1)
            throw new NumericException () ;
        else
            val = n ;
    } // PositiveEvenInteger

    public void assign (int n) throws NumericException
    {
        if (n < 0 | n%2 == 1)
            throw new NumericException ();
        else
            val = n ;
    } // assign
    int toInteger ()
    {
        return val ;
    } //to Integer

    boolean equals (PositiveEvenInteger n)
    {
        return (val == n.val) ;
    } // equals
} //PositiveEven

```

PositiveEvenInteger

Damage assessment

- Analyse system state to judge the extent of corruption caused by a system failure
- Must assess what parts of the state space have been affected by the failure
- Generally based on ‘validity functions’ which can be applied to the state elements to assess if their value is within an allowed range

Damage assessment techniques

- Checksums are used for damage assessment in data transmission
- Redundant pointers can be used to check the integrity of data structures
- Watch dog timers can check for non-terminating processes. If no response after a certain time, a problem is assumed

```

class RobustArray {
    // Checks that all the objects in an array of objects
    // conform to some defined constraint
    boolean [] checkState ;
    CheckableObject [] theRobustArray ;

    RobustArray (CheckableObject [] theArray)
    {
        checkState = new boolean [theArray.length] ;
        theRobustArray = theArray ;
    } //RobustArray
    public void assessDamage () throws ArrayDamagedException
    {
        boolean hasBeenDamaged = false ;

        for (int i= 0; i <this.theRobustArray.length ; i ++)
        {
            if (! theRobustArray [i].check ())
            {
                checkState [i] = true ;
                hasBeenDamaged = true ;
            }
            else
                checkState [i] = false ;
        }
        if (hasBeenDamaged)
            throw new ArrayDamagedException () ;
    } //assessDamage
} // RobustArray

```

Java class with damage assessment

Fault recovery

- Forward recovery
 - Apply repairs to a corrupted system state
- Backward recovery
 - Restore the system state to a known safe state
- Forward recovery is usually application specific
 - domain knowledge is required to compute possible state corrections
- Backward error recovery is simpler. Details of a safe state are maintained and this replaces the corrupted system state

Forward recovery

- Corruption of data coding
 - Error coding techniques which add redundancy to coded data can be used for repairing data corrupted during transmission
- Redundant pointers
 - When redundant pointers are included in data structures (e.g. two-way lists), a corrupted list or filestore may be rebuilt if a sufficient number of pointers are uncorrupted
 - Often used for database and filesystem repair

Backward recovery

- Transactions are a frequently used method of backward recovery. Changes are not applied until computation is complete. If an error occurs, the system is left in the state preceding the transaction
- Periodic checkpoints allow system to 'roll-back' to a correct state

Safe sort procedure

- Sort operation monitors its own execution and assesses if the sort has been correctly executed
- Maintains a copy of its input so that if an error occurs, the input is not corrupted
- Based on identifying and handling exceptions
- Possible in this case as ‘valid’ sort is known. However, in many cases it is difficult to write validity checks

```

class SafeSort {
    static void sort ( int [] intarray, int order ) throws SortError
    {
        int [] copy = new int [intarray.length];

        // copy the input array

        for (int i = 0; i < intarray.length ; i++)
            copy [i] = intarray [i] ;
        try {
            Sort.bubblesort (intarray, intarray.length, order) ;
            if (order == Sort.ascending)
                for (int i = 0; i <= intarray.length-2 ; i++)
                    if (intarray [i] > intarray [i+1])
                        throw new SortError () ;
            else
                for (int i = 0; i <= intarray.length-2 ; i++)
                    if (intarray [i+1] > intarray [i])
                        throw new SortError () ;
        } // try block
        catch (SortError e )
        {
            for (int i = 0; i < intarray.length ; i++)
                intarray [i] = copy [i] ;
            throw new SortError ("Array not sorted") ;
        } //catch
    } // sort
} // SafeSort

```

Safe sort procedure (Java)

Key points

- Fault tolerant software can continue in execution in the presence of software faults
- Fault tolerance requires failure detection, damage assessment, recovery and repair
- Defensive programming is an approach to fault tolerance that relies on the inclusion of redundant checks in a program
- Exception handling facilities simplify the process of defensive programming

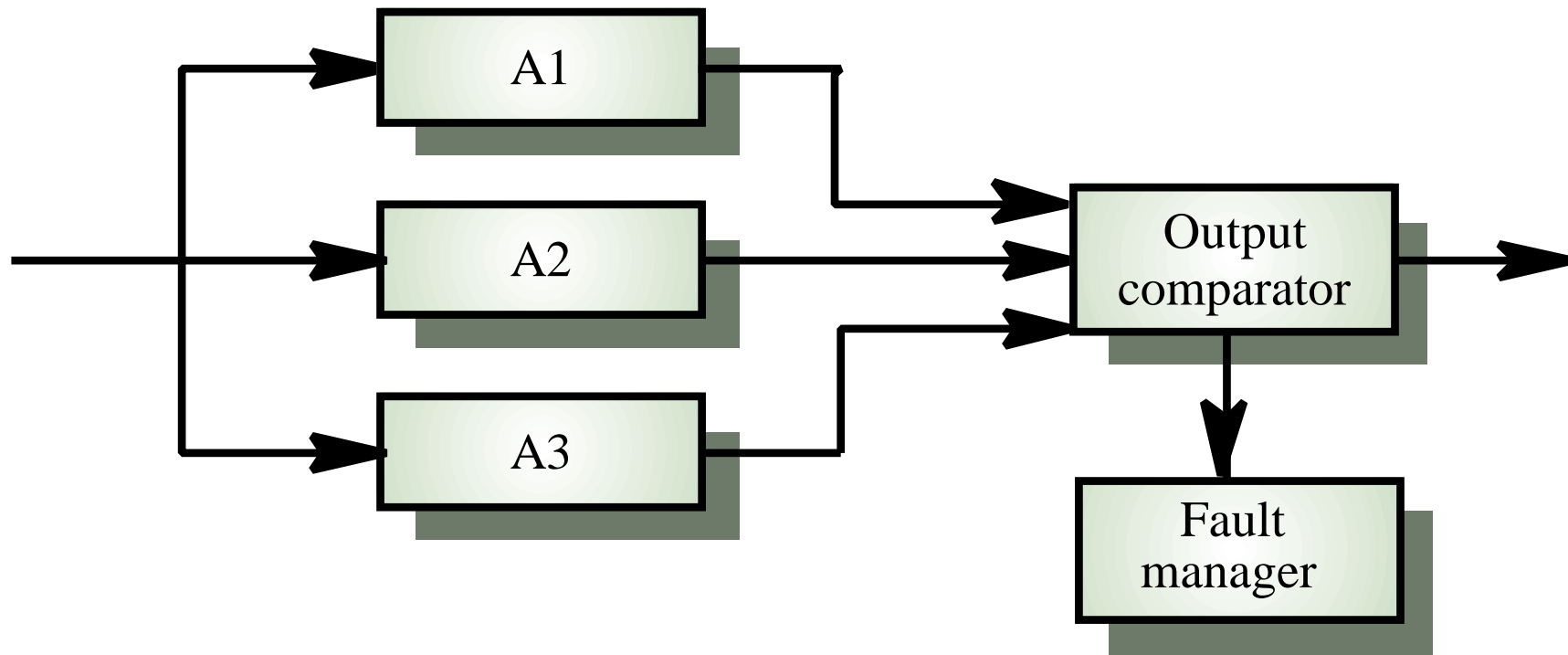
Fault tolerant architecture

- Defensive programming cannot cope with faults that involve interactions between the hardware and the software
- Misunderstandings of the requirements may mean that checks and the associated code are incorrect
- Where systems have high availability requirements, a specific architecture designed to support fault tolerance may be required.
- This must tolerate both hardware and software failure

Hardware fault tolerance

- Depends on triple-modular redundancy (TMR)
- There are three replicated identical components which receive the same input and whose outputs are compared
- If one output is different, it is ignored and component failure is assumed
- Based on most faults resulting from component failures rather than design faults and a low probability of simultaneous component failure

Hardware reliability with TMR



Output selection

- The output comparator is a (relatively) simple hardware unit.
- It compares its input signals and, if one is different from the others, it rejects it. Essentially, selection of the actual output depends on the majority vote.
- The output comparator is connected to a fault management unit that can either try to repair the faulty unit or take it out of service.

Fault tolerant software architectures

- The success of TMR at providing fault tolerance is based on two fundamental assumptions
 - The hardware components do not include common design faults
 - Components fail randomly and there is a low probability of simultaneous component failure
- Neither of these assumptions are true for software
 - It isn't possible simply to replicate the same component as they would have common design faults
 - Simultaneous component failure is therefore virtually inevitable
- Software systems must therefore be diverse

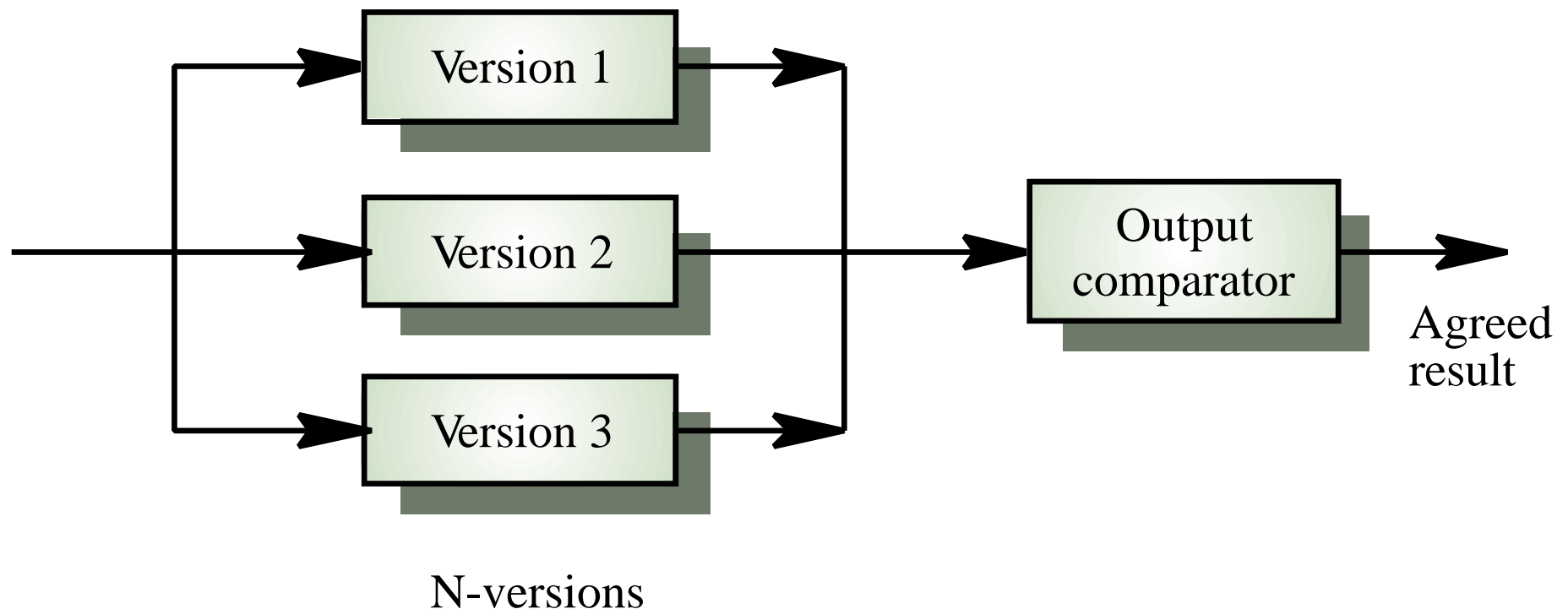
Design diversity

- Different versions of the system are designed and implemented in different ways. They therefore ought to have different failure modes.
- Different approaches to design (e.g object-oriented and function oriented)
 - Implementation in different programming languages
 - Use of different tools and development environments
 - Use of different algorithms in the implementation

Software analogies to TMR

- N-version programming
 - The same specification is implemented in a number of different versions by different teams. All versions compute simultaneously and the majority output is selected using a voting system..
 - This is the most commonly used approach e.g. in Airbus 320.
- Recovery blocks
 - A number of **explicitly** different versions of the same specification are written and executed in sequence
 - An acceptance test is used to select the output to be transmitted.

N-version programming



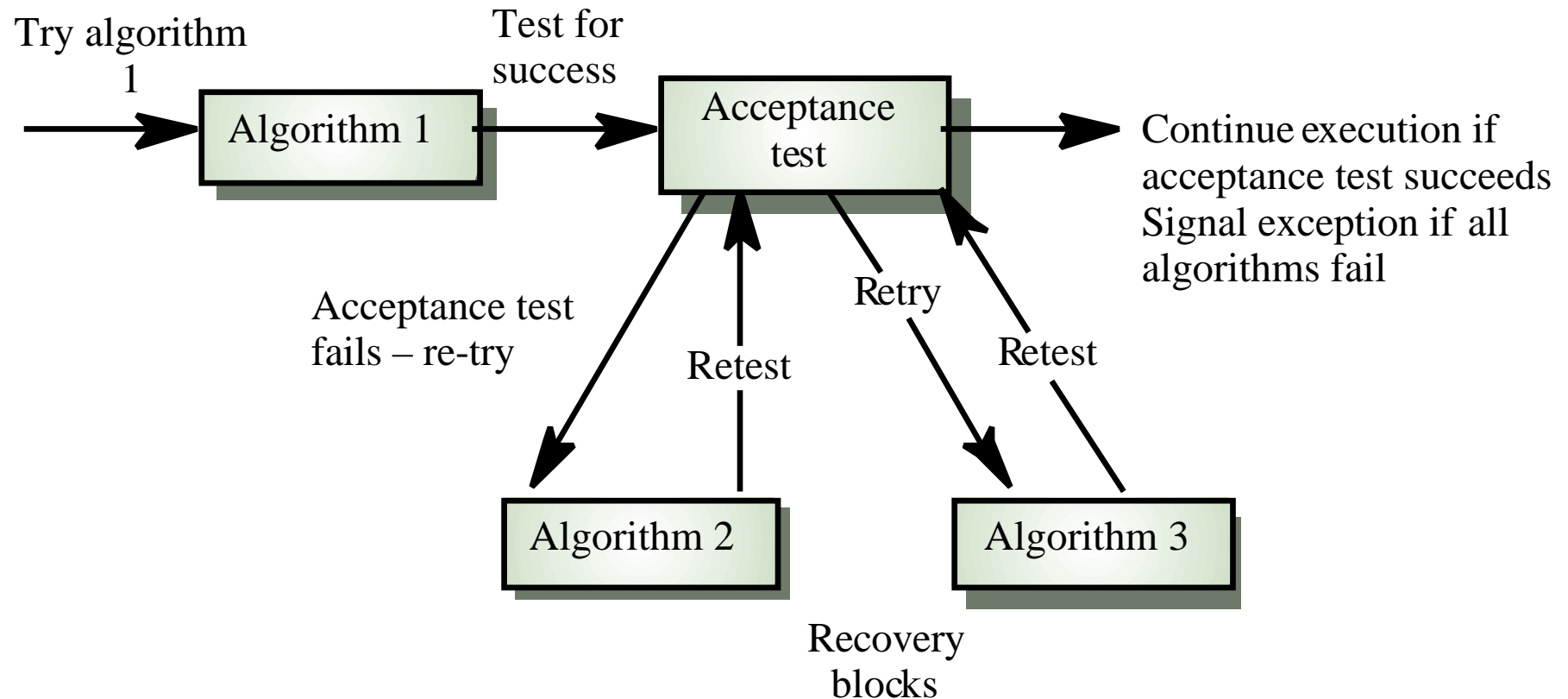
Output comparison

- As in hardware systems, the output comparator is a simple piece of software that uses a voting mechanism to select the output.
- In real-time systems, there may be a requirement that the results from the different versions are all produced within a certain time frame.

N-version programming

- The different system versions are designed and implemented by different teams. It is assumed that there is a low probability that they will make the same mistakes. The algorithms used should but may not be different.
- There is some empirical evidence that teams commonly misinterpret specifications in the same way and chose the same algorithms in their systems.

Recovery blocks



Recovery blocks

- Force a different algorithm to be used for each version so they reduce the probability of common errors
- However, the design of the acceptance test is difficult as it must be independent of the computation used
- There are problems with this approach for real-time systems because of the sequential operation of the redundant versions

Problems with design diversity

- Teams are not culturally diverse so they tend to tackle problems in the same way
- Characteristic errors
 - Different teams make the same mistakes. Some parts of an implementation are more difficult than others so all teams tend to make mistakes in the same place.
 - Specification errors
 - If there is an error in the specification then this is reflected in all implementations
 - This can be addressed to some extent by using multiple specification representations

Specification dependency

- Both approaches to software redundancy are susceptible to specification errors. If the specification is incorrect, the system could fail
- This is also a problem with hardware but software specifications are usually more complex than hardware specifications and harder to validate
- This has been addressed in some cases by developing separate software specifications from the same user specification

Is software redundancy needed?

- Unlike hardware, software faults are not an inevitable consequence of the physical world
- Some people therefore believe that a higher level of reliability and availability can be attained by investing effort in reducing software complexity.
- Redundant software is much more complex so there is scope for a range of additional errors that affect the system reliability but are caused by the existence of the fault-tolerance controllers.

Key points

- Dependability in a system can be achieved through fault avoidance and fault tolerance
- Some programming language constructs such as gotos, recursion and pointers are inherently error-prone
- Data typing allows many potential faults to be trapped at compile time.

Key points

- Fault tolerant architectures rely on replicated hardware and software components
- They include mechanisms to detect a faulty component and to switch it out of the system
- N-version programming and recovery blocks are two different approaches to designing fault-tolerant software architectures
- Design diversity is essential for software redundancy

Verification and Validation

- Assuring that a software system meets a user's needs

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

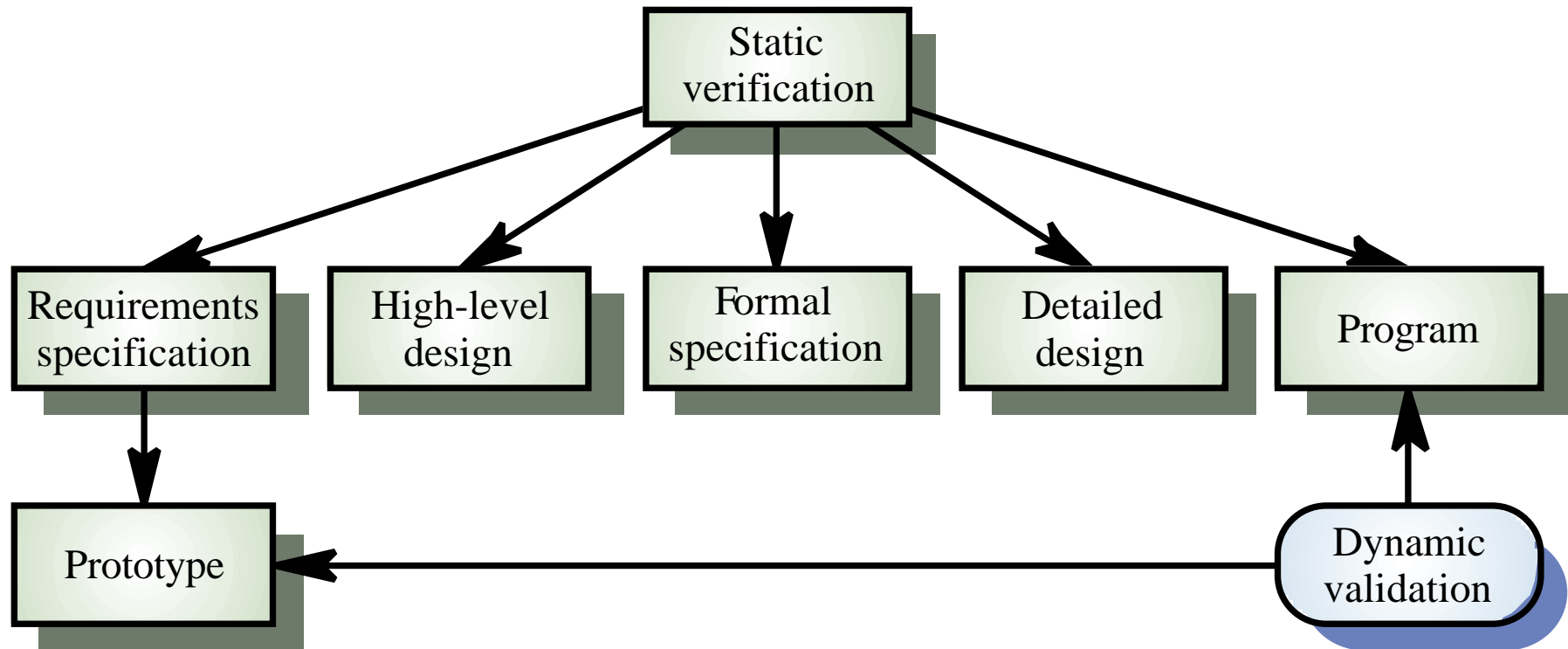
The V & V process

- 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 usable in an operational situation.

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
- A successful test is a test which discovers one or more errors
- The only validation technique for non-functional requirements
- 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 20
- Statistical testing
 - tests designed to reflect the frequency of user inputs. Used for reliability estimation.
 - Covered in Chapter 21

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

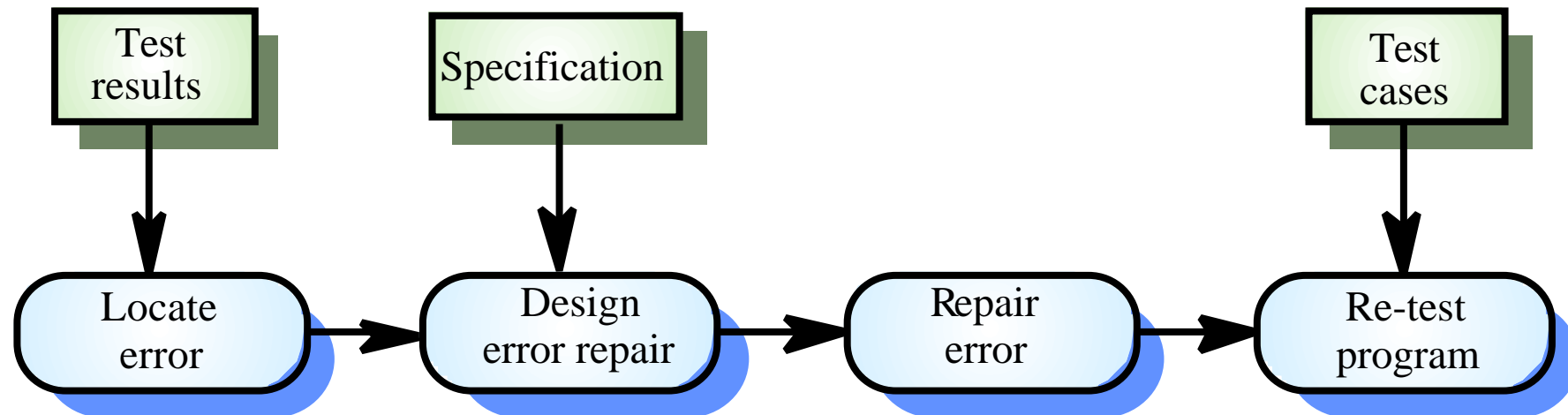
V & V confidence

- 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

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

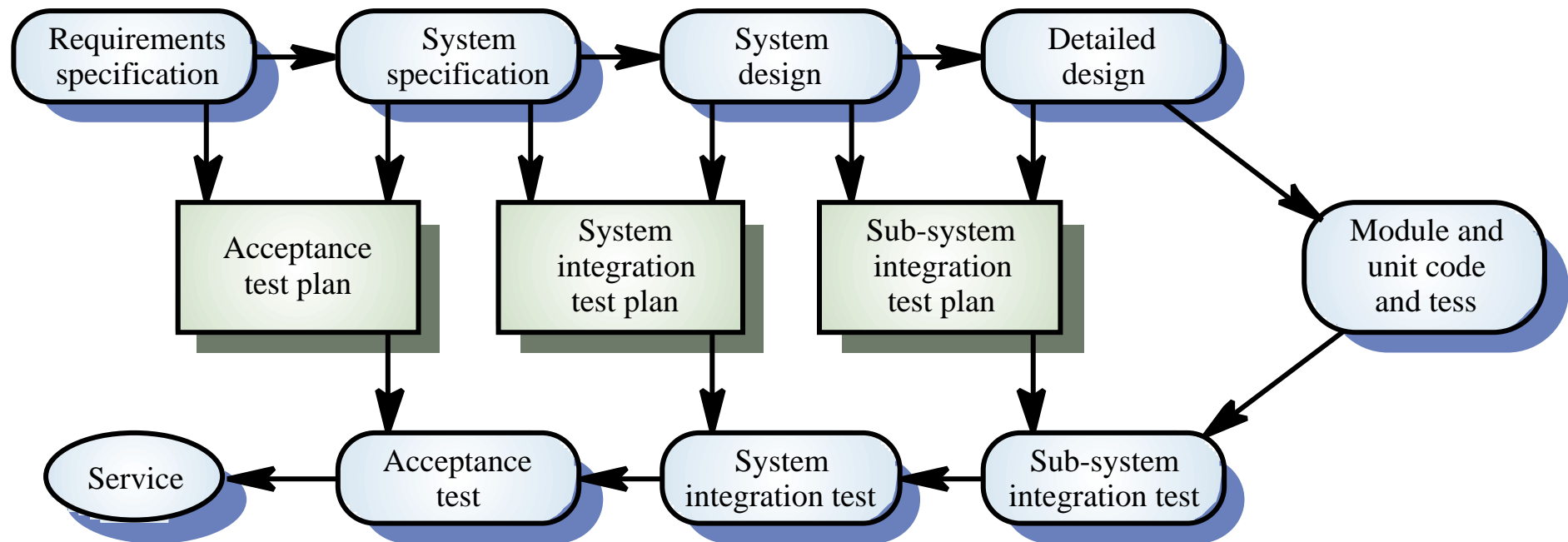
The debugging process



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

Software inspections

- Involve people examining the source representation with the aim of discovering anomalies and defects
- Do not require execution of a system so may be used before implementation
- May be applied to any representation of the system (requirements, design, test data, etc.)
- Very effective technique for discovering 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.

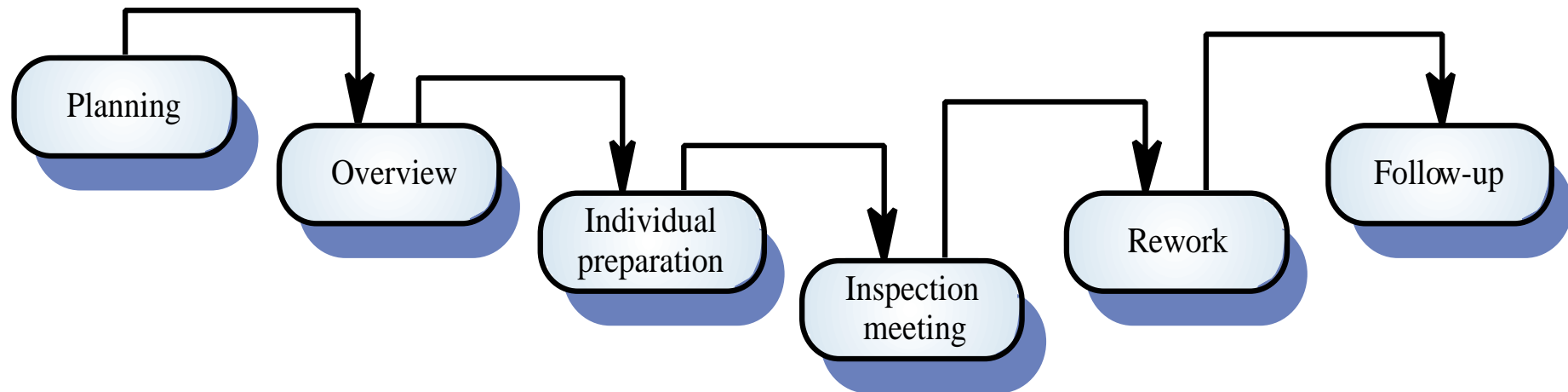
Program inspections

- 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 must be available
- An error checklist should be prepared
- Management must accept that inspection will increase costs early in the software process
- Management must not use inspections for staff appraisal

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 teams

- Made up of at least 4 members
- Author of the code being inspected
- Inspector who finds errors, omissions and inconsistencies
- Reader who reads the code to the team
- Moderator who chairs the meeting and notes discovered errors
- Other roles are Scribe and Chief moderator

Inspection checklists

- Checklist of common errors should be used to drive the inspection
- Error checklist is programming language dependent
- The 'weaker' the type checking, the larger the checklist
- Examples: Initialisation, Constant naming, loop termination, array bounds, etc.

Fault class	Inspection check
Data faults	<p>Are all program variables initialised before their values are used?</p> <p>Have all constants been named?</p> <p>Should the lower bound of arrays be 0, 1, or something else?</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 delimiter explicitly assigned?</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>
Input/output faults	<p>Are all input variables used?</p> <p>Are all output variables assigned a value before they are output?</p>
Interface faults	<p>Do all function and procedure 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 checks

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 = £2800

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
- Very effective as an aid to inspections. 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. Must be used with care.

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

LINT static analysis

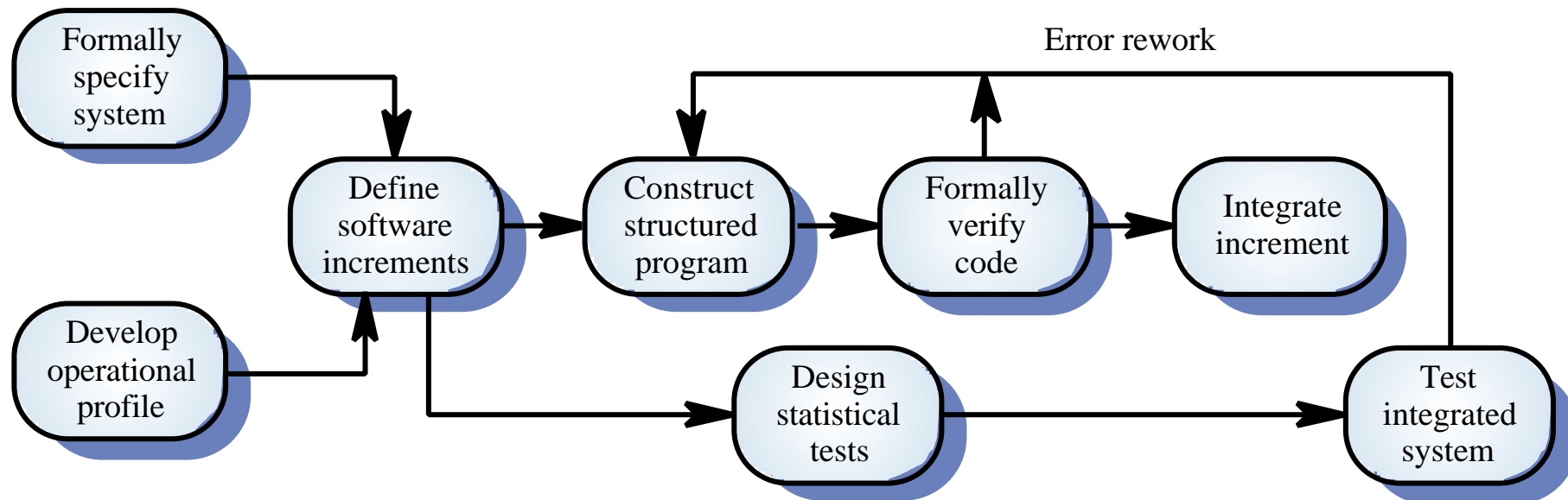
Use of static analysis

- 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

Cleanroom software development

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

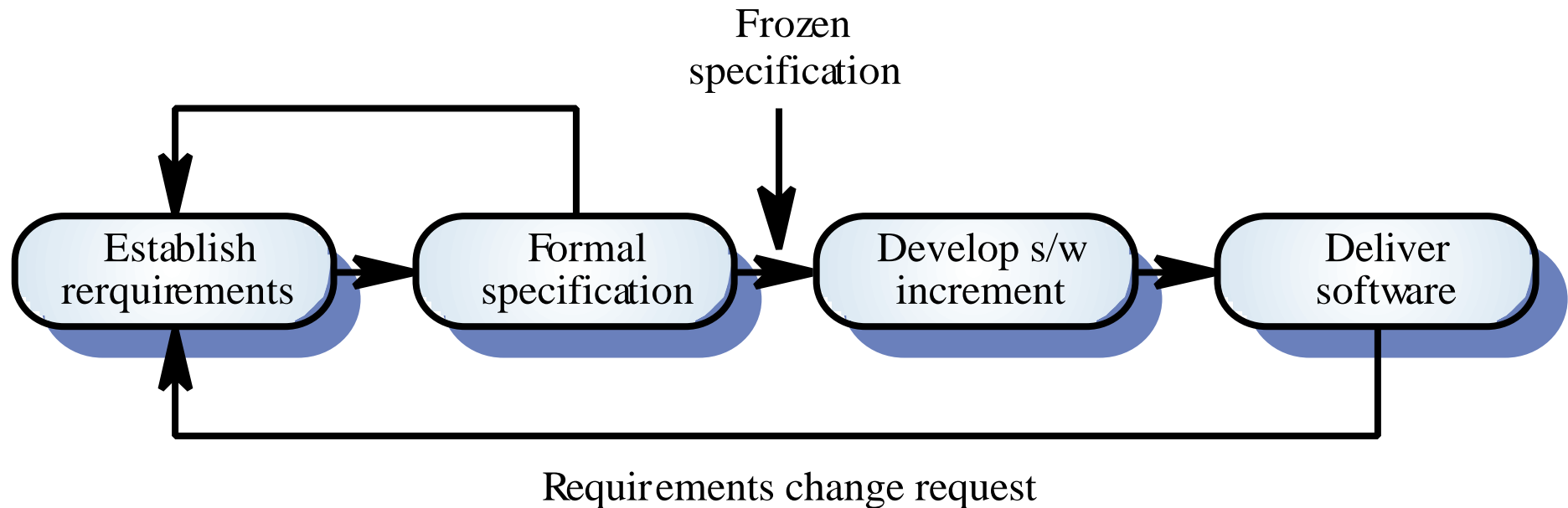
The Cleanroom process



Cleanroom process characteristics

- Formal specification using a state transition model
- Incremental development
- Structured programming - limited control and abstraction constructs are used
- Static verification using rigorous inspections
- Statistical testing of the system (covered in Ch. 21).

Incremental development



Formal specification and inspections

- The state based model is a system specification and the inspection process checks the program against this model
- 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

- Results in IBM have been very impressive with few discovered faults in delivered systems
- Independent assessment shows that the process is no more expensive than other approaches
- Fewer errors than in a 'traditional' development process
- Not clear how this approach can be transferred to an environment with less skilled or less highly motivated 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 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

Defect testing

- Testing programs to establish the presence of system defects

Objectives

- To understand testing techniques that are geared to discover program faults
- To introduce guidelines for interface testing
- To understand specific approaches to object-oriented testing
- To understand the principles of CASE tool support for testing

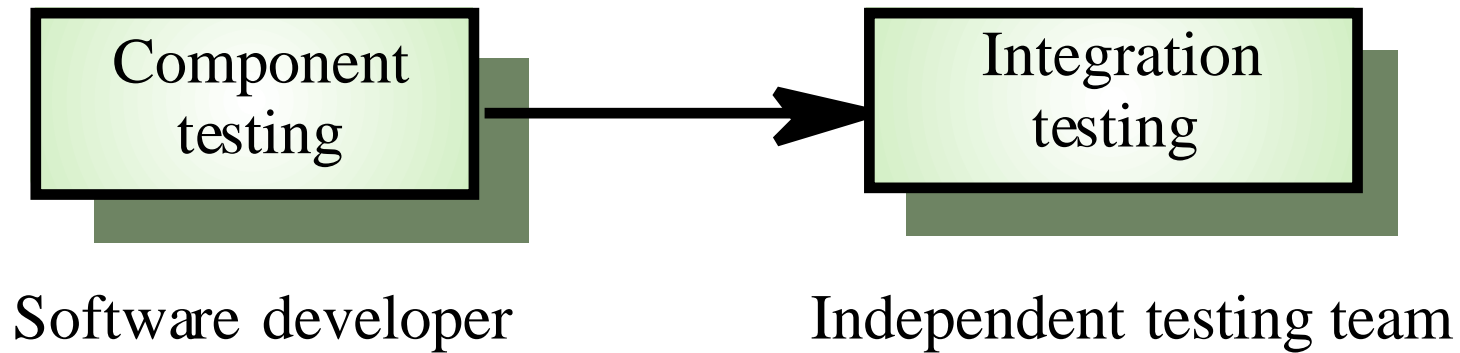
Topics covered

- Defect testing
- Integration testing
- Object-oriented testing
- Testing workbenches

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
- Integration 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

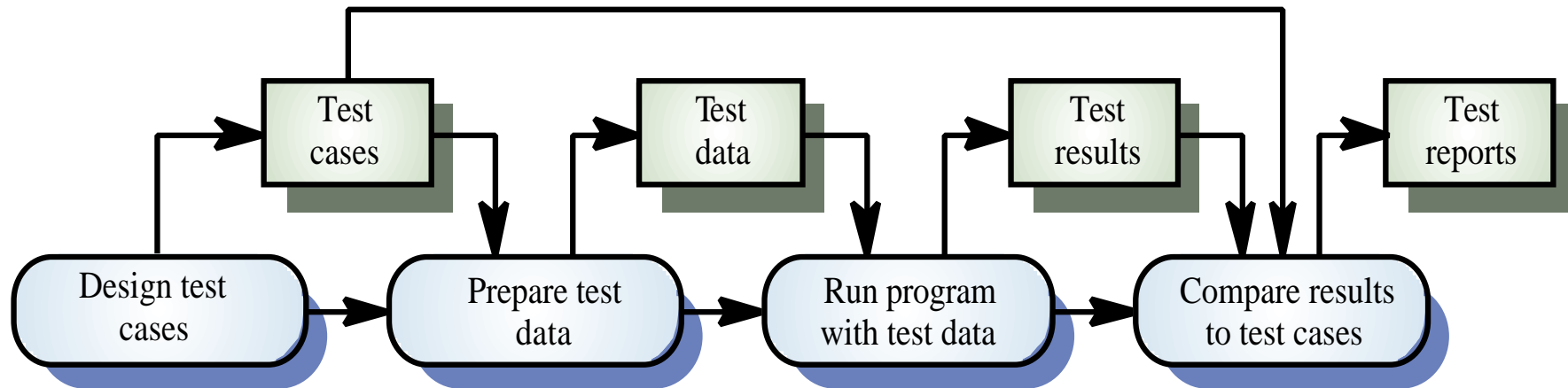
Testing priorities

- Only exhaustive testing can show a program is free from defects. However, exhaustive testing is impossible
- Tests should exercise a system's capabilities rather than its components
- Testing old capabilities is more important than testing new capabilities
- Testing typical situations is more important than boundary value cases

Test data and test cases

- *Test data* Inputs which have been devised to test the system
- *Test cases* Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification

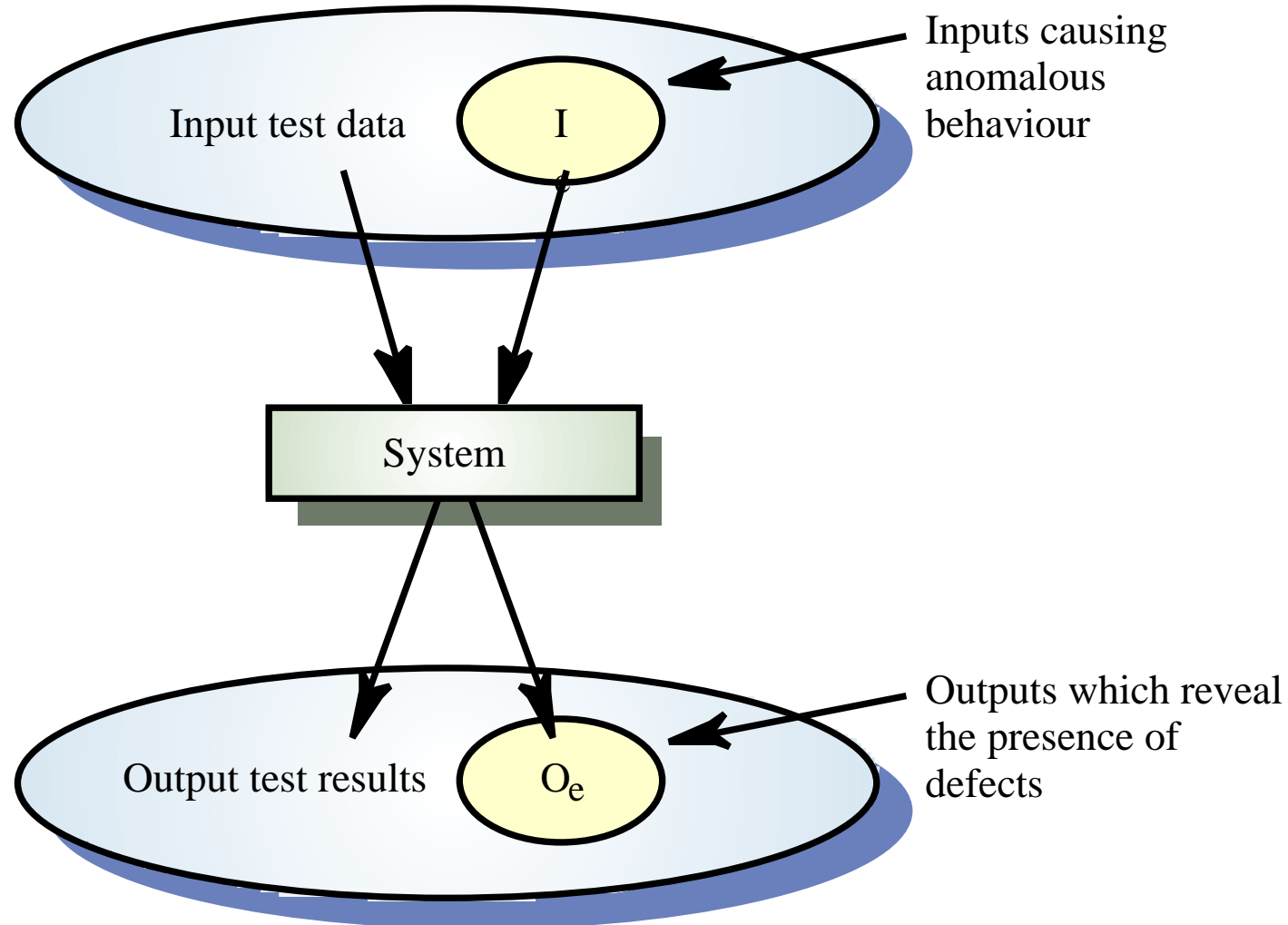
The defect testing process



Black-box testing

- An approach to testing where the program is considered as a ‘black-box’
- The program test cases are based on the system specification
- Test planning can begin early in the software process

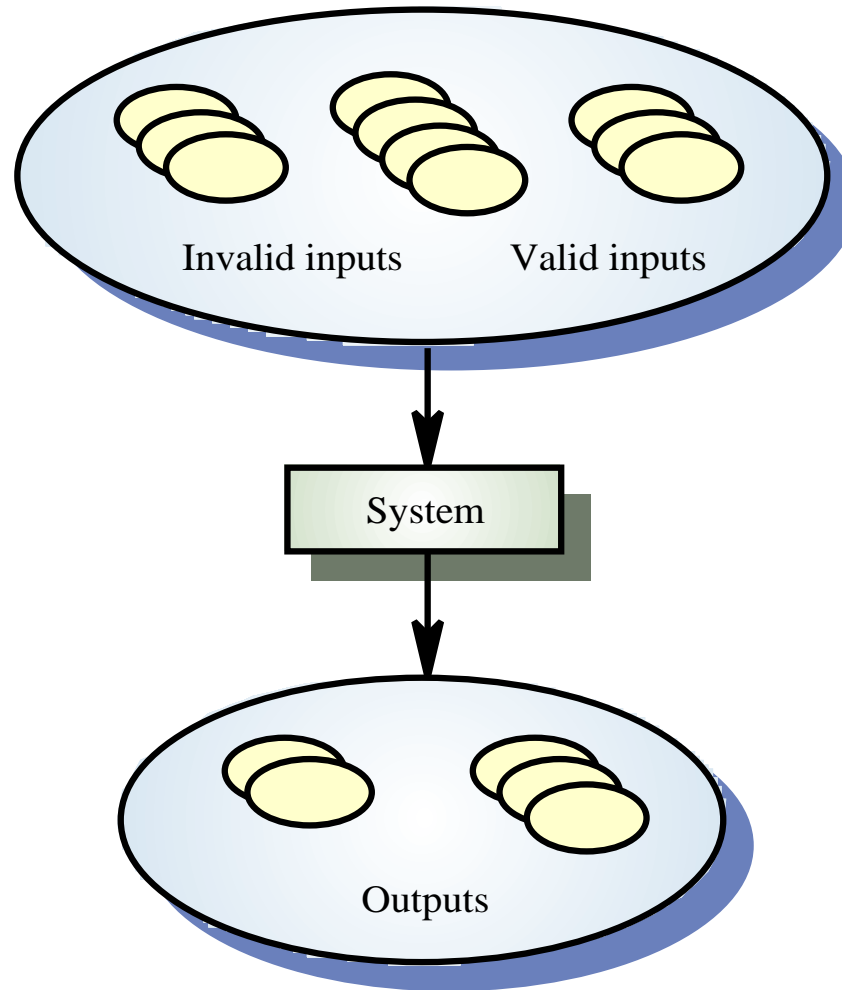
Black-box testing



Equivalence partitioning

- 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 where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition

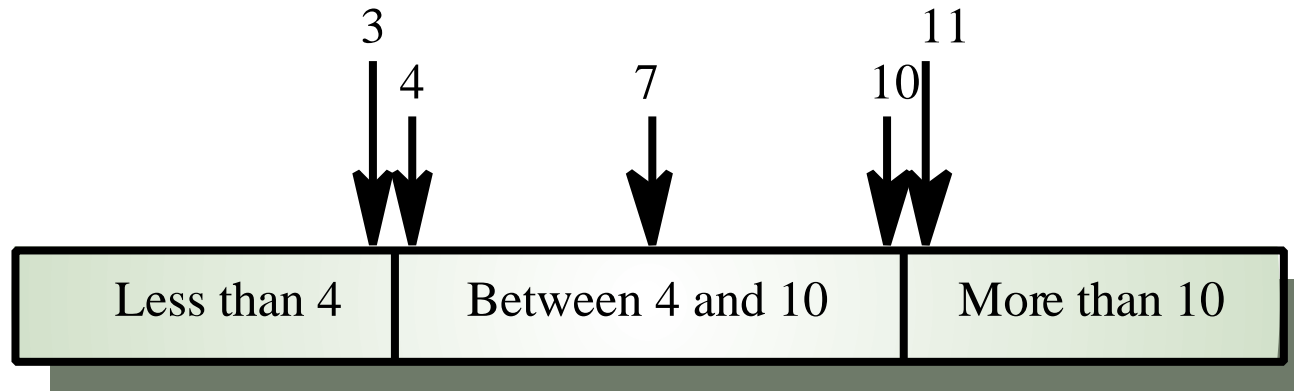
Equivalence partitioning



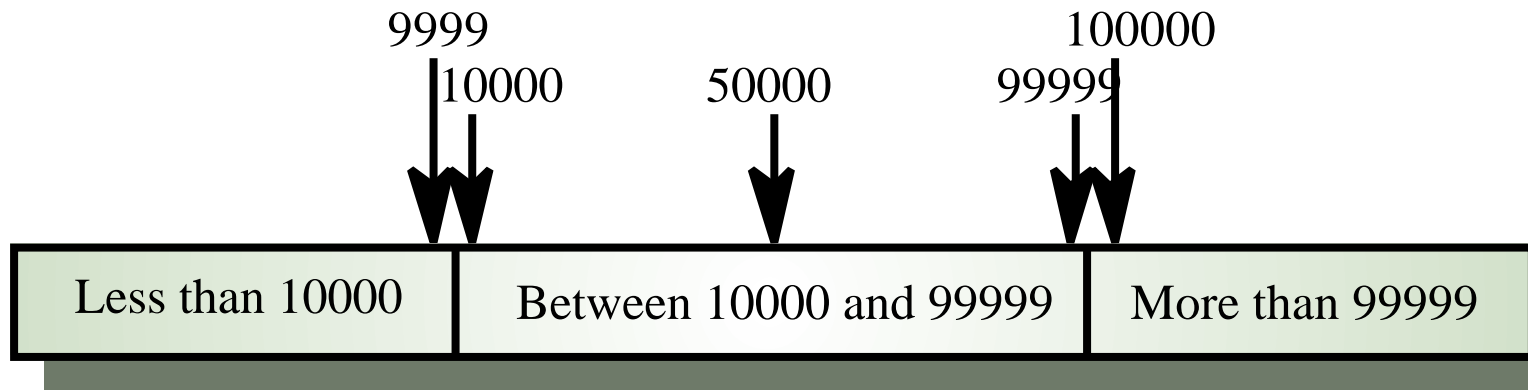
Equivalence partitioning

- Partition system inputs and outputs into ‘equivalence sets’
 - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are $<10,000$, $10,000-99,999$ and $>99,999$
- Choose test cases at the boundary of these sets
 - 00000, 09999, 10000, 99999, 10001

Equivalence partitions



Number of input values



Input values

Search routine specification

procedure Search (Key : ELEM ; T: ELEM_ARRAY;
Found : **in out** BOOLEAN; L: **in out** ELEM_INDEX) ;

Pre-condition

-- the array 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

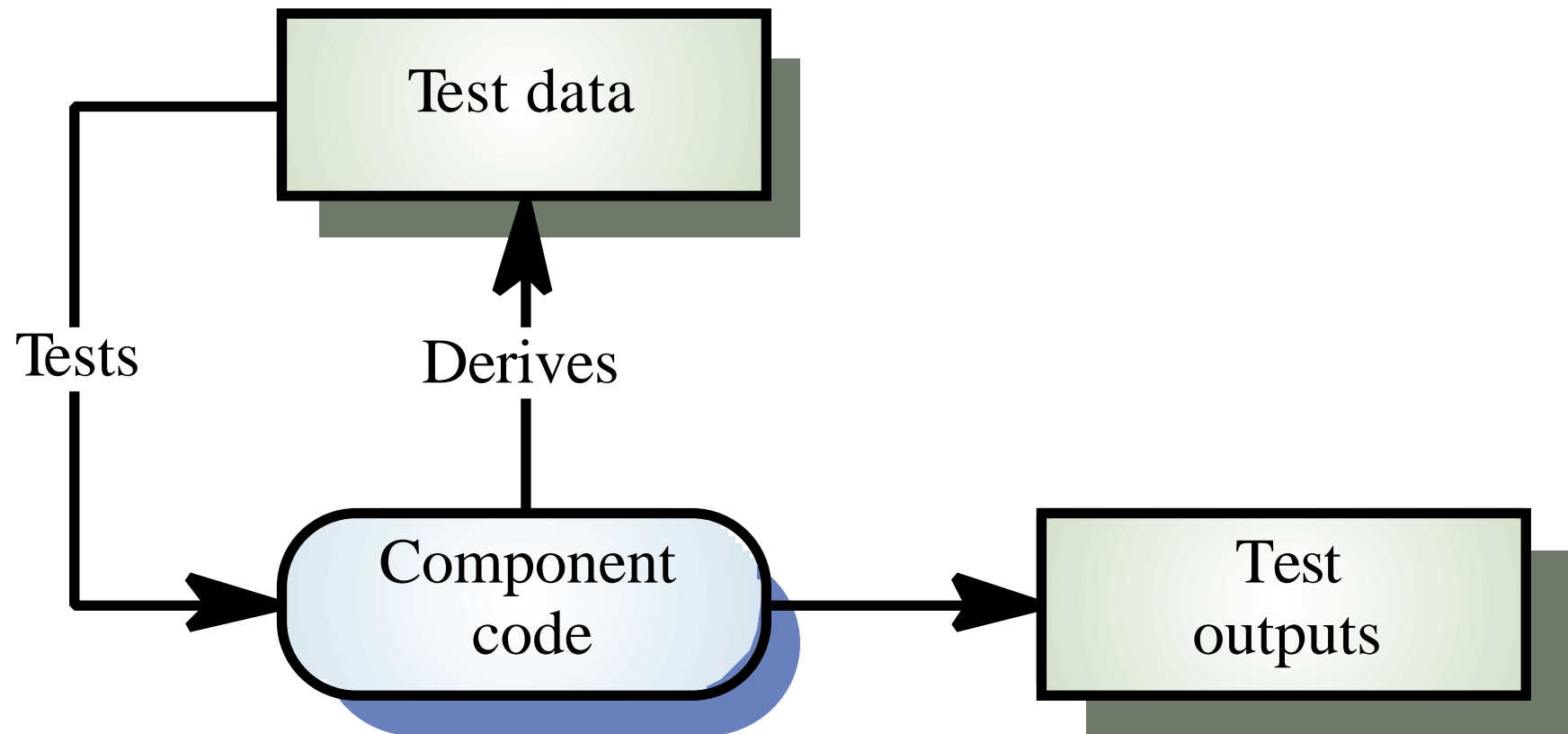
Array	Element
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

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)

White-box testing



```

class BinSearch {

    // This is an encapsulation of a binary search function that takes an array of
    // ordered objects and a key and returns an object with 2 attributes namely
    // index - the value of the array index
    // found - a boolean indicating whether or not the key is in the array
    // An object is returned because it is not possible in Java to pass basic types by
    // reference to a function and so return two values
    // the key is -1 if the element is not found

    public static void search ( int key, int [] elemArray, Result r )
    {
        int bottom = 0 ;
        int top = elemArray.length - 1 ;
        int mid ;
        r.found = false ; r.index = -1 ;
        while ( bottom <= top )
        {
            mid = (top + bottom) / 2 ;
            if (elemArray [mid] == key)
            {
                r.index = mid ;
                r.found = true ;
                return ;
            } // if part
            else
            {
                if (elemArray [mid] < key)
                    bottom = mid + 1 ;
                else
                    top = mid - 1 ;
            }
        } //while loop
    } // search
} //BinSearch

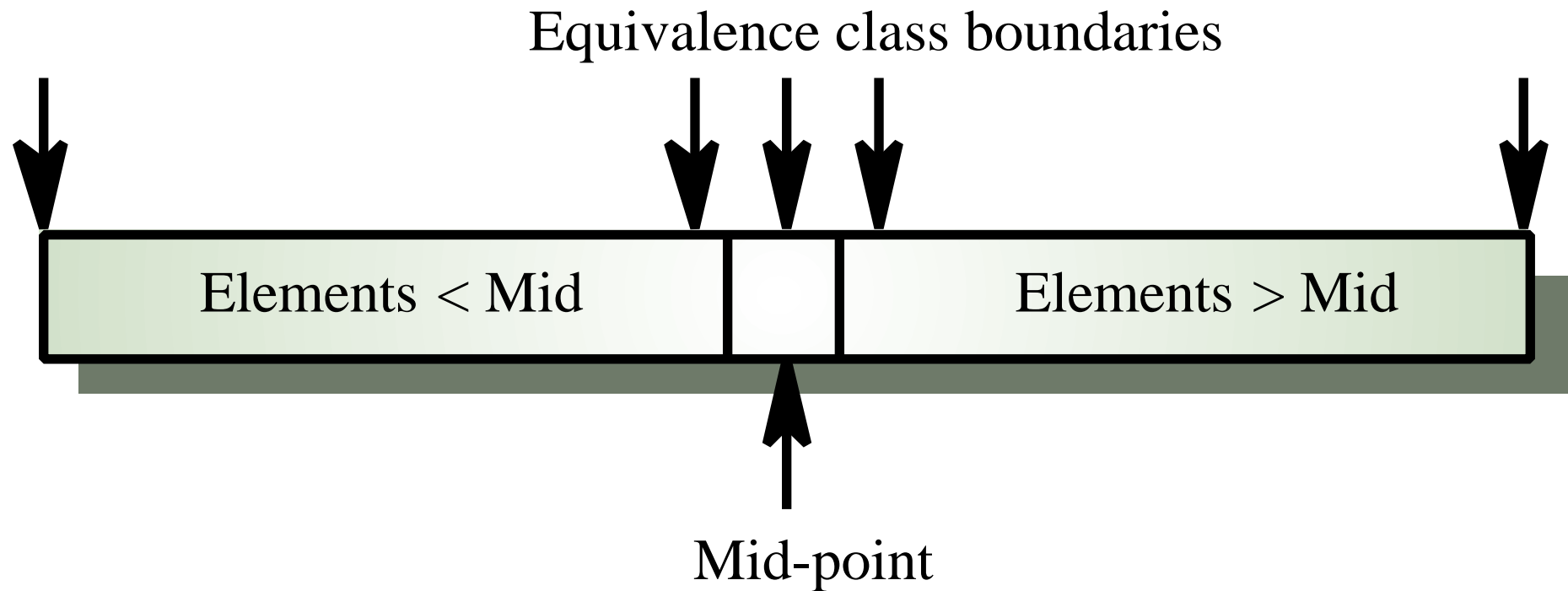
```

Binary search (Java)

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

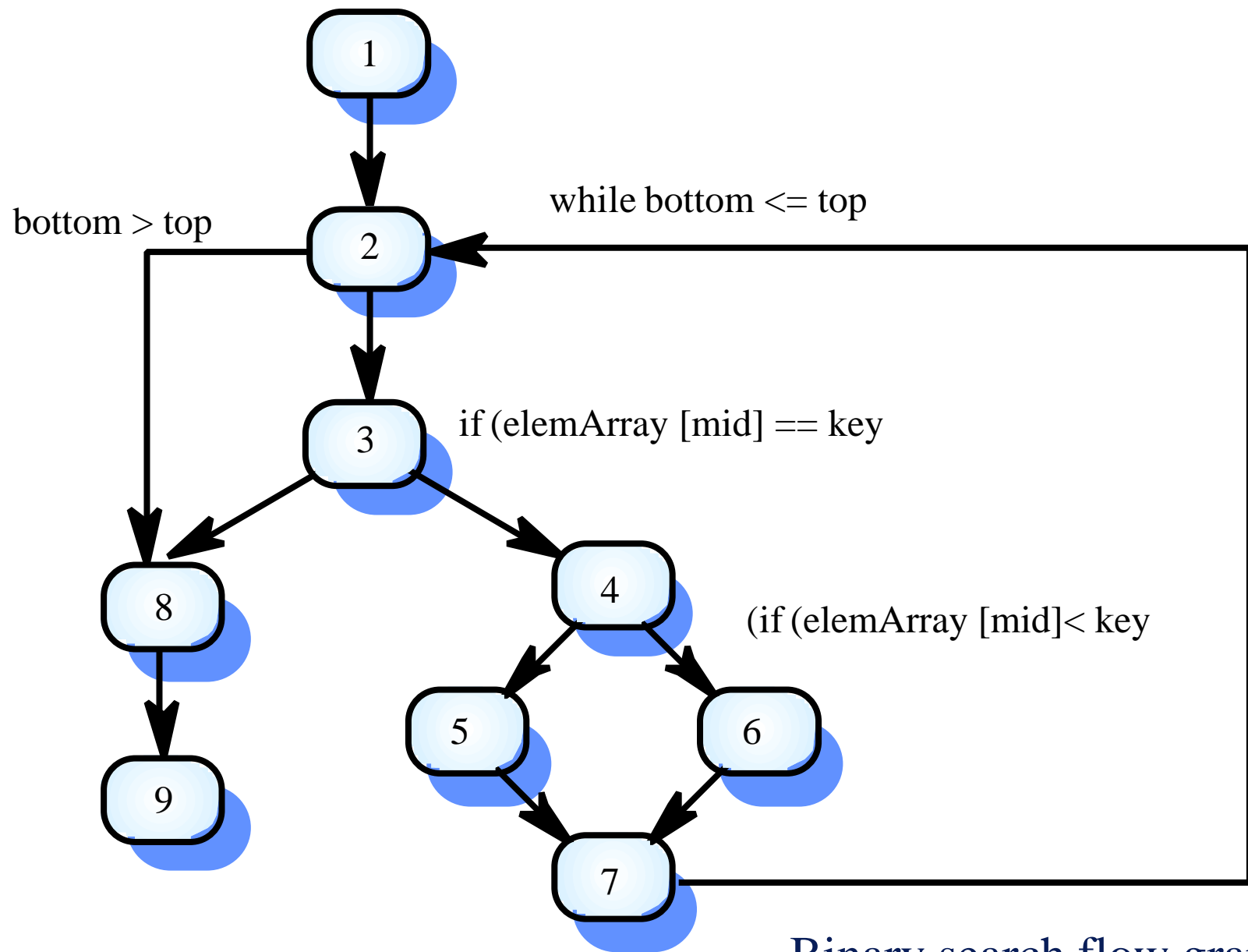
- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once
- The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control
- Statements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

Program flow graphs

- Describes the program control flow. Each branch is shown as a separate path and loops are shown by arrows looping back to the loop condition node
- Used as a basis for computing the cyclomatic complexity
- Cyclomatic complexity = Number of edges - Number of nodes + 2

Cyclomatic complexity

- The number of tests to test all control statements equals the cyclomatic complexity
- Cyclomatic complexity equals number of conditions in a program
- Useful if used with care. Does not imply adequacy of testing.
- Although all paths are executed, all combinations of paths are not executed



Binary search flow graph

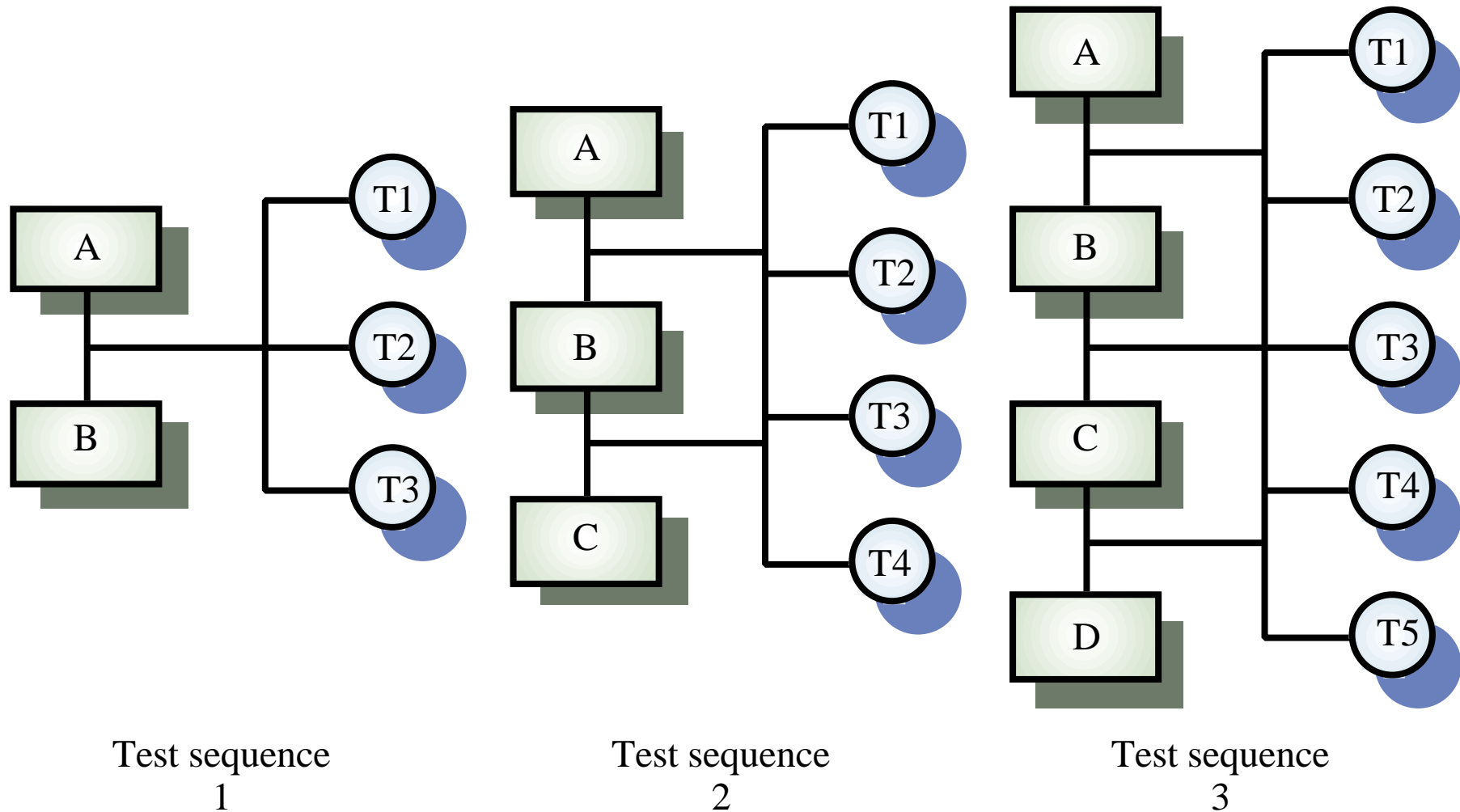
Independent paths

- 1, 2, 3, 8, 9
- 1, 2, 3, 4, 6, 7, 2
- 1, 2, 3, 4, 5, 7, 2
- 1, 2, 3, 4, 6, 7, 2, 8, 9
- 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

Integration testing

- Tests complete systems or subsystems composed of integrated components
- Integration testing should be black-box testing with tests derived from the specification
- Main difficulty is localising errors
- Incremental integration testing reduces this problem

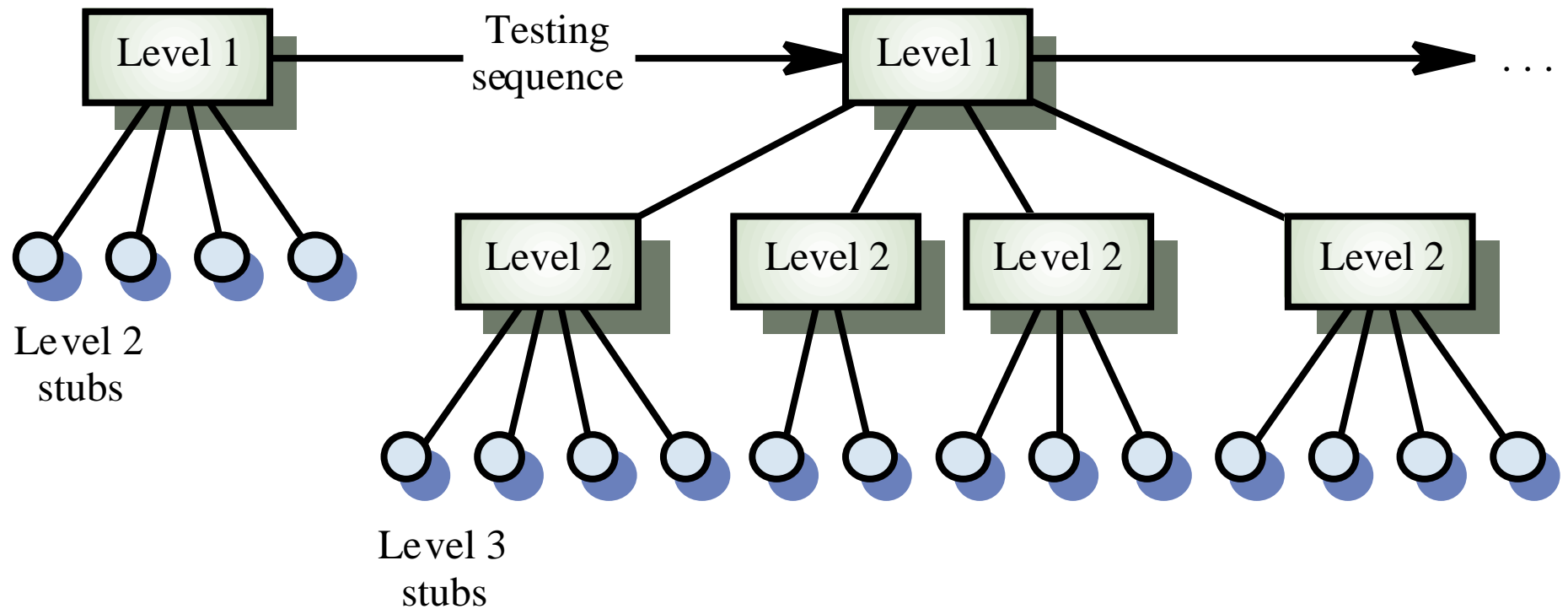
Incremental integration testing



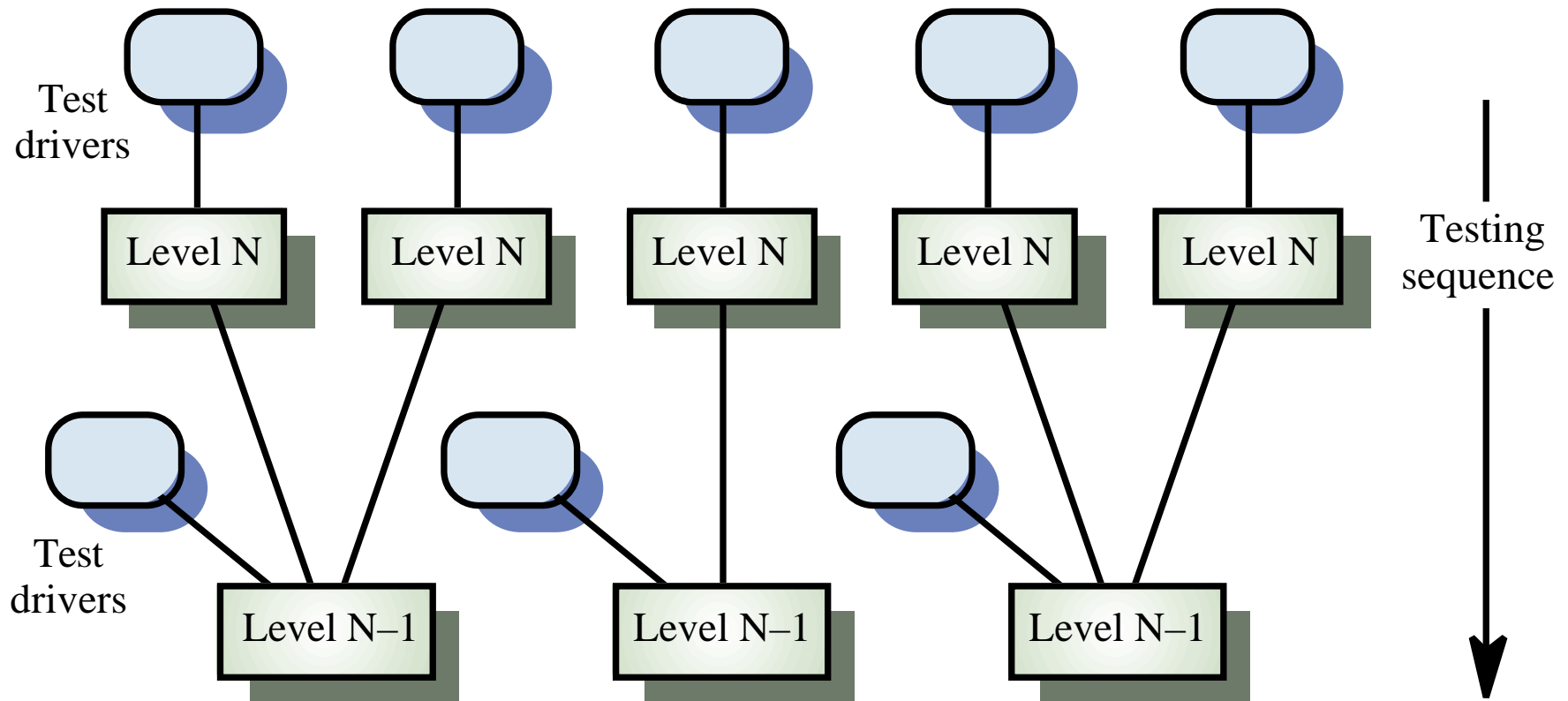
Approaches to integration testing

- Top-down testing
 - Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- Bottom-up testing
 - Integrate individual components in levels until the complete system is created
- In practice, most integration involves a combination of these strategies

Top-down testing



Bottom-up testing



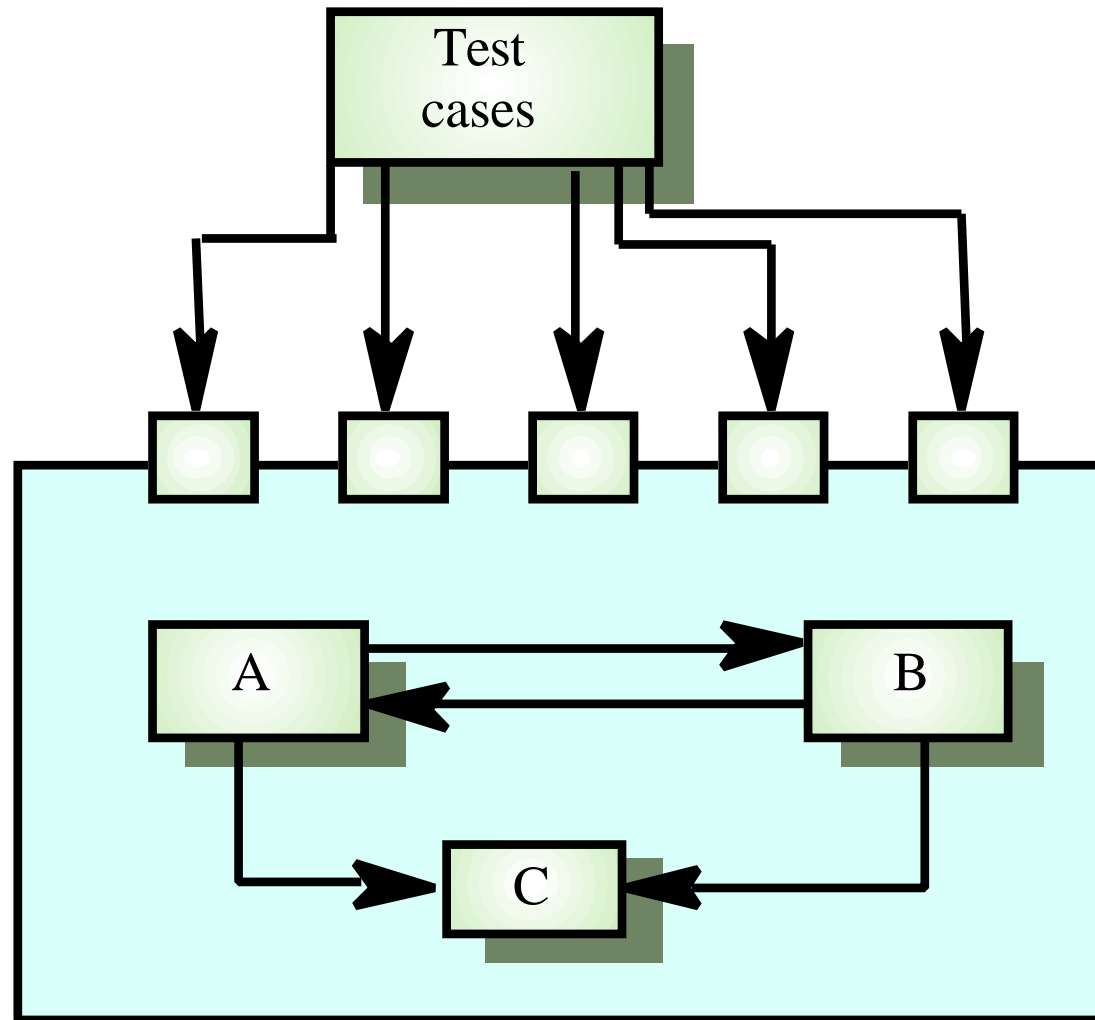
Tetsing approaches

- 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

Interface testing

- Takes place when modules or sub-systems are integrated to create larger systems
- 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



Interfaces types

- Parameter interfaces
 - Data passed from one procedure to another
- Shared memory interfaces
 - Block of memory is shared between procedures
- 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

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
- Particularly relevant to distributed systems which can exhibit severe degradation as a network becomes overloaded

Object-oriented testing

- The components to be tested are object classes that are instantiated as objects
- Larger grain than individual functions so approaches to white-box testing have to be extended
- No obvious ‘top’ to the system for top-down integration and testing

Testing levels

- Testing operations associated with objects
- Testing object classes
- Testing clusters of cooperating objects
- Testing the complete OO system

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)

- Test cases are needed for all operations
- Use a state model to identify state transitions for testing
- Examples of testing sequences
 - Shutdown ♦ Waiting ♦ Shutdown
 - Waiting ♦ Calibrating ♦ Testing ♦ Transmitting ♦ Waiting
 - Waiting ♦ Collecting ♦ Waiting ♦ Summarising ♦ Transmitting ♦ Waiting

Object integration

- Levels of integration are less distinct in object-oriented systems
- Cluster testing is concerned with integrating and testing clusters of cooperating objects
- Identify clusters using knowledge of the operation of objects and the system features that are implemented by these clusters

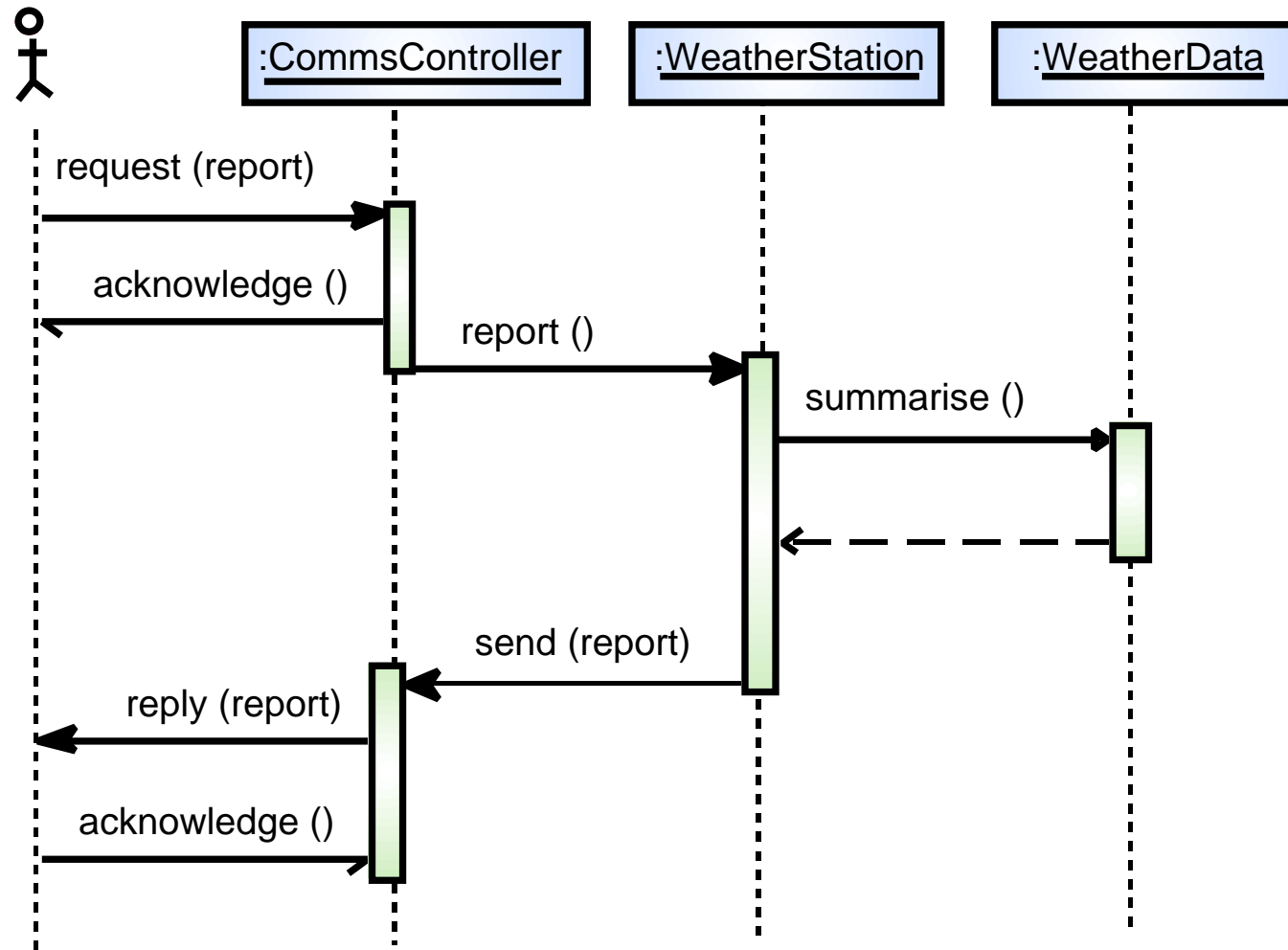
Approaches to cluster testing

- Use-case or scenario testing
 - Testing is based on a user interactions with the system
 - Has the advantage that it tests system features as experienced by users
- Thread testing
 - Tests the systems response to events as processing threads through the system
- Object interaction testing
 - Tests sequences of object interactions that stop when an object operation does not call on services from another object

Scenario-based testing

- Identify scenarios from use-cases and supplement these with interaction diagrams that show the objects involved in the scenario
- Consider the scenario in the weather station system where a report is generated

Collect weather data



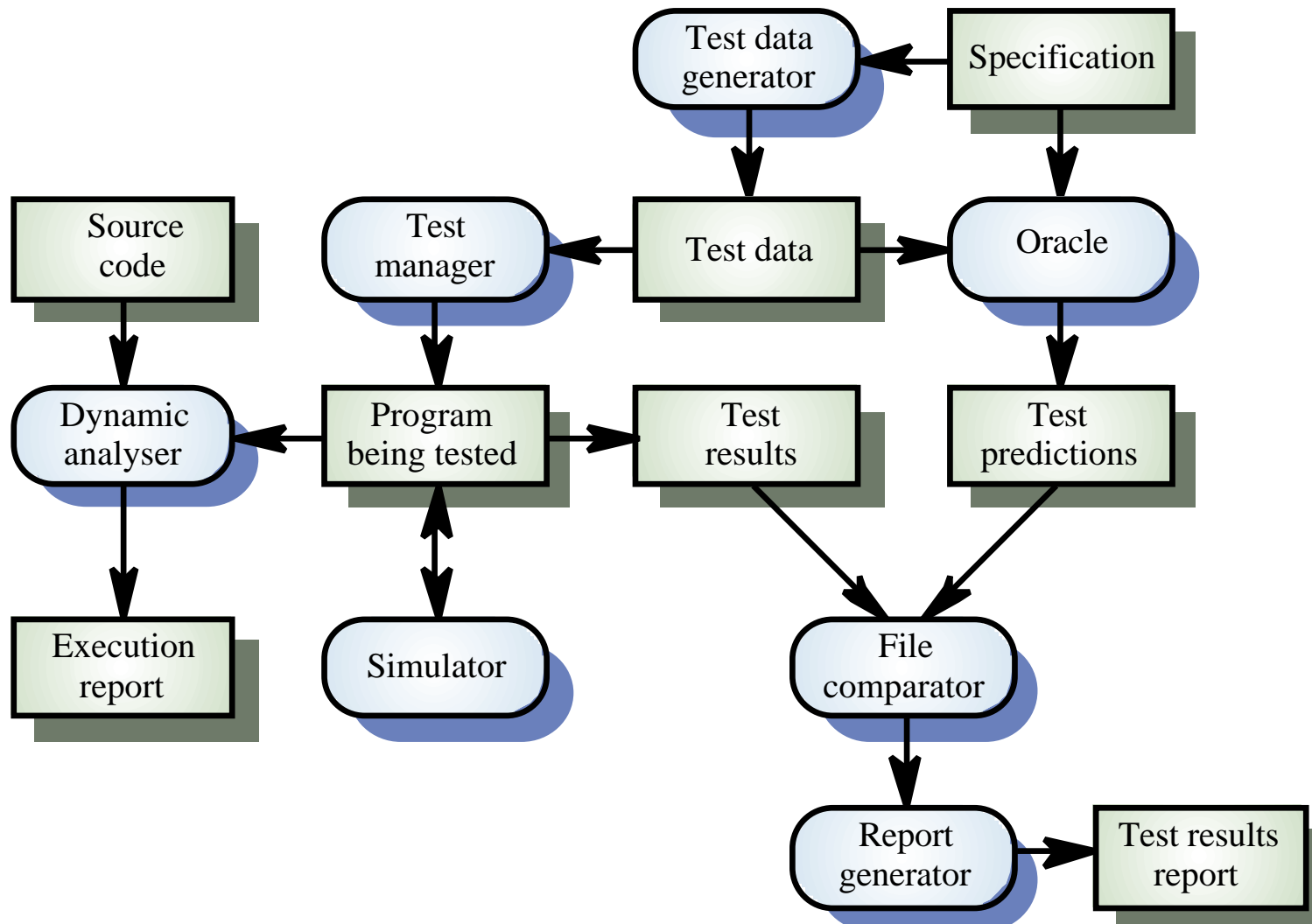
Weather station testing

- Thread of methods executed
 - CommsController:request ♦ WeatherStation:report ♦
WeatherData:summarise
- Inputs and outputs
 - Input of report request with associated acknowledge and a final output of a report
 - Can be tested by creating raw data and ensuring that it is summarised properly
 - Use the same raw data to test the WeatherData object

Testing workbenches

- Testing is an expensive process phase. Testing workbenches provide a range of tools to reduce the time required and total testing costs
- Most testing workbenches are open systems because testing needs are organisation-specific
- Difficult to integrate with closed design and analysis workbenches

A testing workbench



Tetsing 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

- Test parts of a system which are commonly used rather than those which are rarely executed
- Equivalence partitions are sets of test cases where the program should behave in an equivalent way
- Black-box testing is based on the system specification
- Structural testing identifies test cases which cause all paths through the program to be executed

Key points

- Test coverage measures ensure that all statements have been executed at least once.
- Interface defects arise because of specification misreading, misunderstanding, errors or invalid timing assumptions
- To test object classes, test all operations, attributes and states
- Integrate object-oriented systems around clusters of objects

Critical Systems Validation

Validating the reliability, safety
and security of computer-based
systems

Validation perspectives

- Reliability validation
 - Does the measured reliability of the system meet its specification?
 - Is the reliability of the system good enough to satisfy users?
- Safety validation
 - Does the system always operate in such a way that accidents do not occur or that accident consequences are minimised?
- Security validation
 - Is the system and its data secure against external attack?

Validation techniques

- Static techniques
 - Design reviews and program inspections
 - Mathematical arguments and proof
- Dynamic techniques
 - Statistical testing
 - Scenario-based testing
 - Run-time checking
- Process validation
 - Design development processes that minimise the chances of process errors that might compromise the dependability of the system

Static validation techniques

- Static validation is concerned with analyses of the system documentation (requirements, design, code, test data).
- It is concerned with finding errors in the system and identifying potential problems that may arise during system execution.
- Documents may be prepared (structured arguments, mathematical proofs, etc.) to support the static validation

Static techniques for safety validation

- Demonstrating safety by testing is difficult because testing is intended to demonstrate what the system does in a particular situation. Testing all possible operational situations is impossible
- Normal reviews for correctness may be supplemented by specific techniques that are intended to focus on checking that unsafe situations never arise

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

Hazard-driven analysis

- Effective safety assurance relies on hazard identification (covered in previous lectures)
- Safety can be assured by
 - Hazard avoidance
 - Accident avoidance
 - Protection systems
- Safety reviews should demonstrate that one or more of these techniques have been applied to all identified hazards

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 presents a list of arguments, based on identified hazards, why there is an acceptably low probability that these hazards will not result in an accident
- Arguments can be based on formal proof, design rationale, safety proofs, etc. Process factors may also be included

Formal methods and critical systems

- The development of critical systems is one of the ‘success’ stories for formal methods
- Formal methods are mandated in Britain for the development of some types of safety-critical software for defence applications
- There is not currently general agreement on the value of formal methods in critical systems development

Formal methods and validation

- Specification validation
 - Developing a formal model of a system requirements specification forces a detailed analysis of that specification and this usually reveals errors and omissions
 - Mathematical analysis of the formal specification is possible and this also discovers specification problems
- Formal verification
 - Mathematical arguments (at varying degrees of rigour) are used to demonstrate that a program or a design is consistent with its formal specification

Problems with formal validation

- The formal model of the specification is not understandable by domain experts
 - It is difficult or impossible to check if the formal model is an accurate representation of the specification for most systems
 - A consistently wrong specification is not useful!
- Verification does not scale-up
 - Verification is complex, error-prone and requires the use of systems such as theorem provers. The cost of verification increases exponentially as the system size increases.

Formal methods conclusion

- Formal specification and checking of critical system components is, in my view, useful
 - While formality does not provide any guarantees, it helps to increase confidence in the system by demonstrating that some classes of error are not present
- Formal verification is only likely to be used for very small, critical, system components
 - About 5-6000 lines of code seems to be the upper limit for practical verification

Safety proofs

- Safety proofs are intended to show that the system cannot reach in unsafe state
- Weaker than correctness proofs which must show that the system code conforms to its specification
- Generally based on proof by contradiction
 - Assume that an unsafe state can be reached
 - Show that this is contradicted by the program code
- May be displayed graphically

Construction of a safety proof

- 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

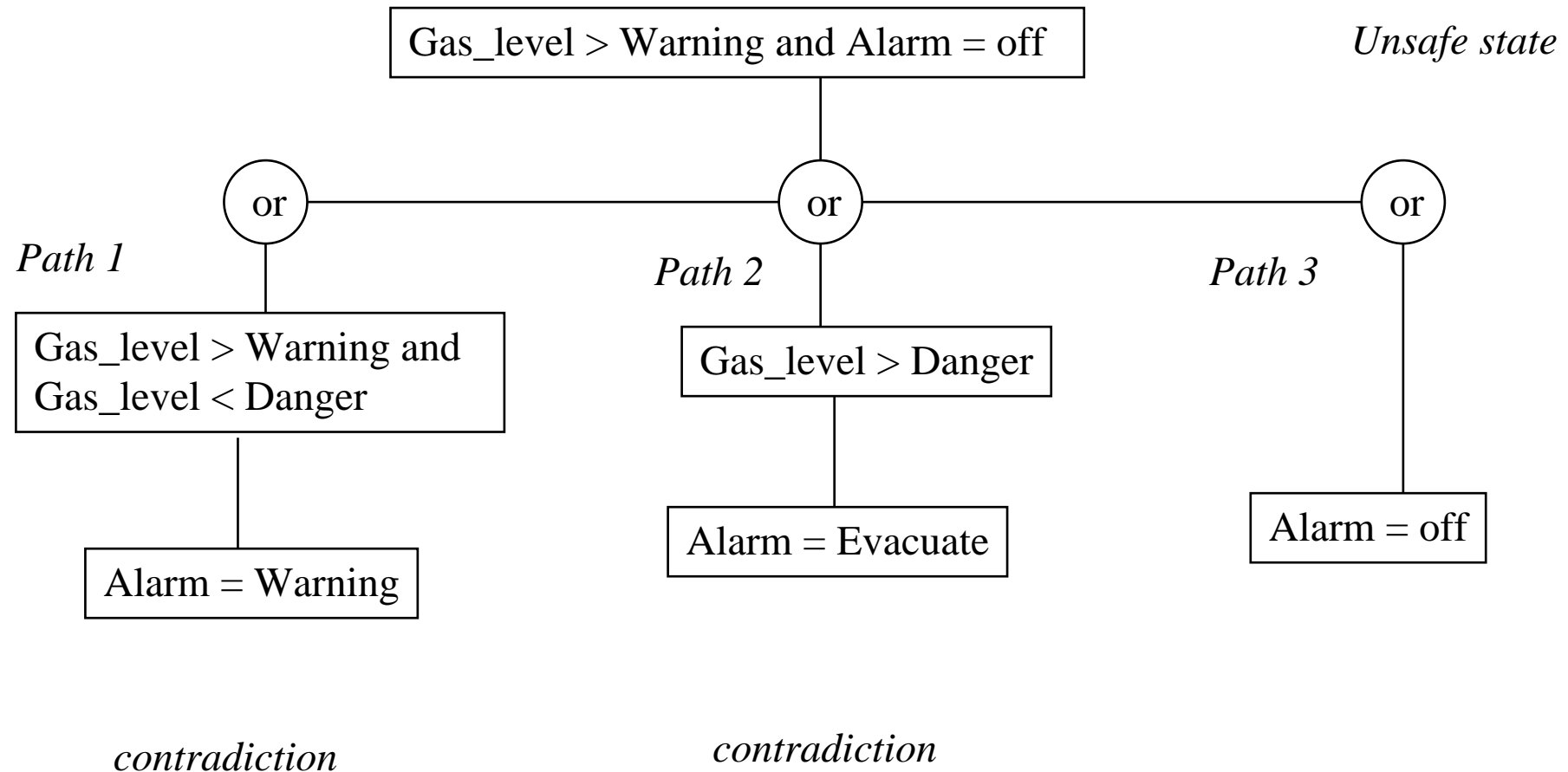
Gas warning system

- System to warn of poisonous gas. Consists of a sensor, a controller and an alarm
- Two levels of gas are hazardous
 - Warning level - no immediate danger but take action to reduce level
 - Evacuate level - immediate danger. Evacuate the area
- The controller takes air samples, computes the gas level and then decides whether or not the alarm should be activated

Gas sensor control

```
Gas_level: GL_TYPE ;
loop
  -- Take 100 samples of air
  Gas_level := 0.000 ;
  for i in 1..100 loop
    Gas_level := Gas_level + Gas_sensor.Read ;
  end loop ;
  Gas_level := Gas_level / 100 ;
  if Gas_level > Warning and Gas_level < Danger then
    Alarm := Warning ; Wait_for_reset ;
  elsif Gas_level > Danger then
    Alarm := Evacuate ; Wait_for_reset ;
  else
    Alarm := off ;
  end if ;
end loop ;
```

Graphical argument



Condition checking

Gas_level < Warning	Path 3	Alarm = off (Contradiction)
Gas_level = Warning	Path 3	Alarm = off (Contradiction)
Gas_level > Warning and Gas_level < Danger	Path 1	Alarm = Warning (Contradiction)
Gas_level = Danger	Path 3	Alarm = off
Gas_level > Danger	Path 2	Alarm = Evacuate (Contradiction)

Code is incorrect.

Gas_level = Danger does not cause the alarm to be on

Key points

- Safety-related systems should be developed to be as simple as possible using ‘safe’ development techniques
- Safety assurance may depend on ‘trusted’ development processes and specific development techniques such as the use of formal methods and safety proofs
- Safety proofs are easier than proofs of consistency or correctness. They must demonstrate that the system cannot reach an unsafe state. Usually proofs by contradiction

Dynamic validation techniques

- These are techniques that are concerned with validating the system in execution
 - Testing techniques - analysing the system outside of its operational environment
 - Run-time checking - checking during execution that the system is operating within a dependability ‘envelope’

Reliability validation

- Reliability validation involves exercising the program to assess whether or not it has reached the required level of reliability
- Cannot be included as part of a normal defect testing process because data for defect testing is (usually) atypical of actual usage data
- Statistical testing must be used where a statistically significant data sample based on simulated usage is used to assess the reliability

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

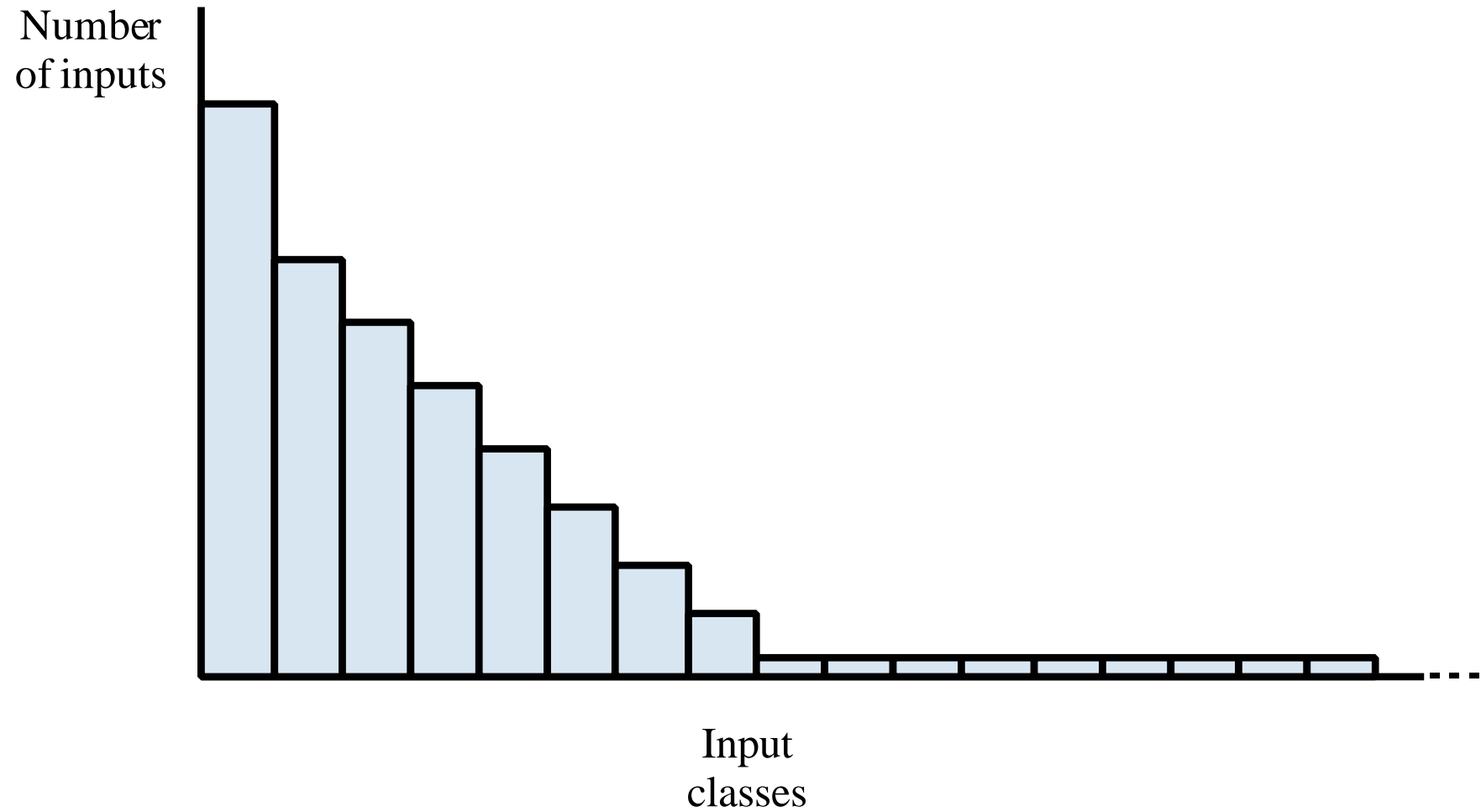
Reliability validation process

- 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

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
- 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



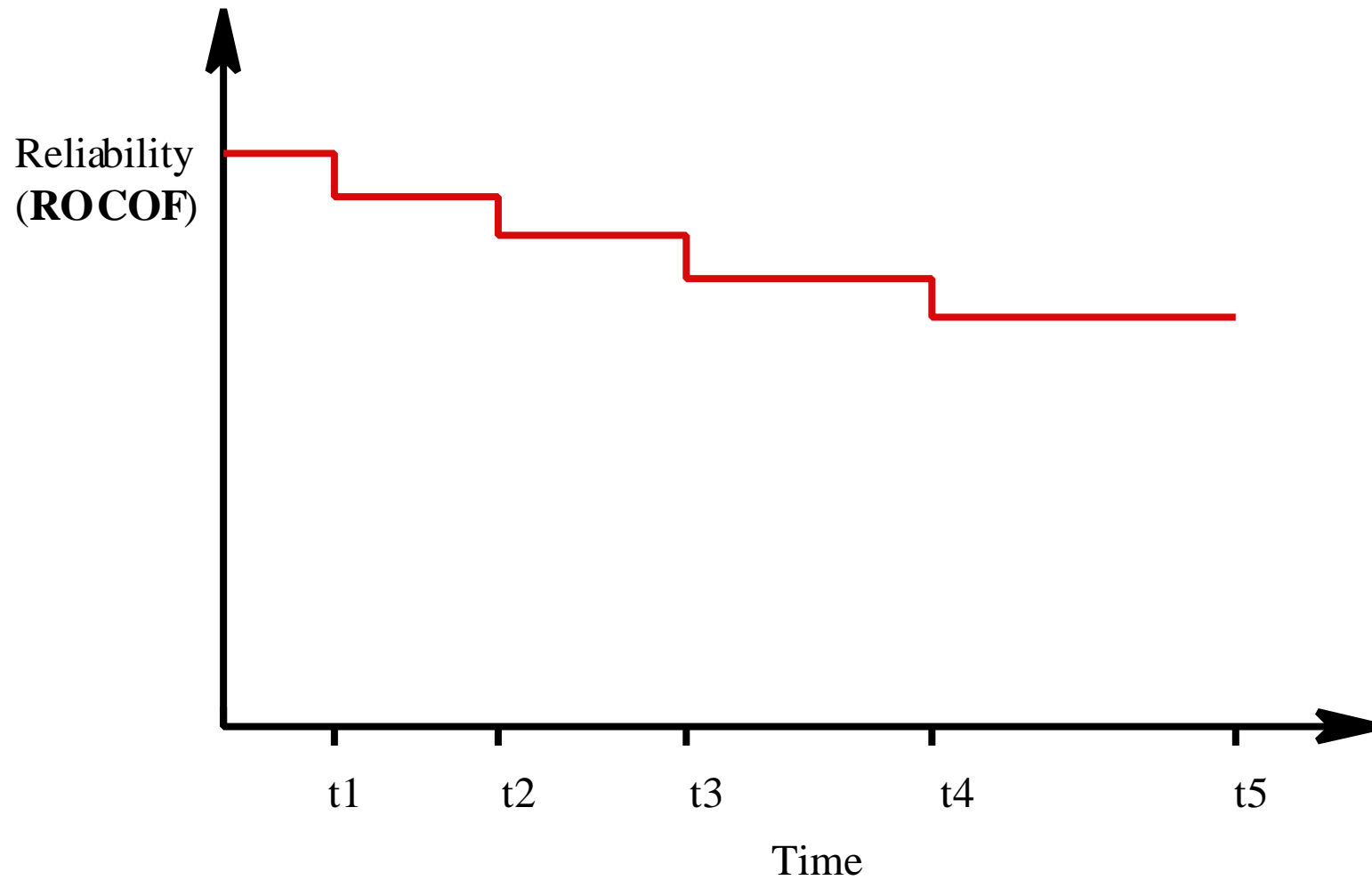
Operational profile generation

- 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 modelling

- A reliability growth model is a mathematical model of the system reliability change as it is tested and faults are removed
- Used as a means of reliability prediction by extrapolating from current data
 - Simplifies test planning and customer negotiations
- Depends on the use of statistical testing to measure the reliability of a system version

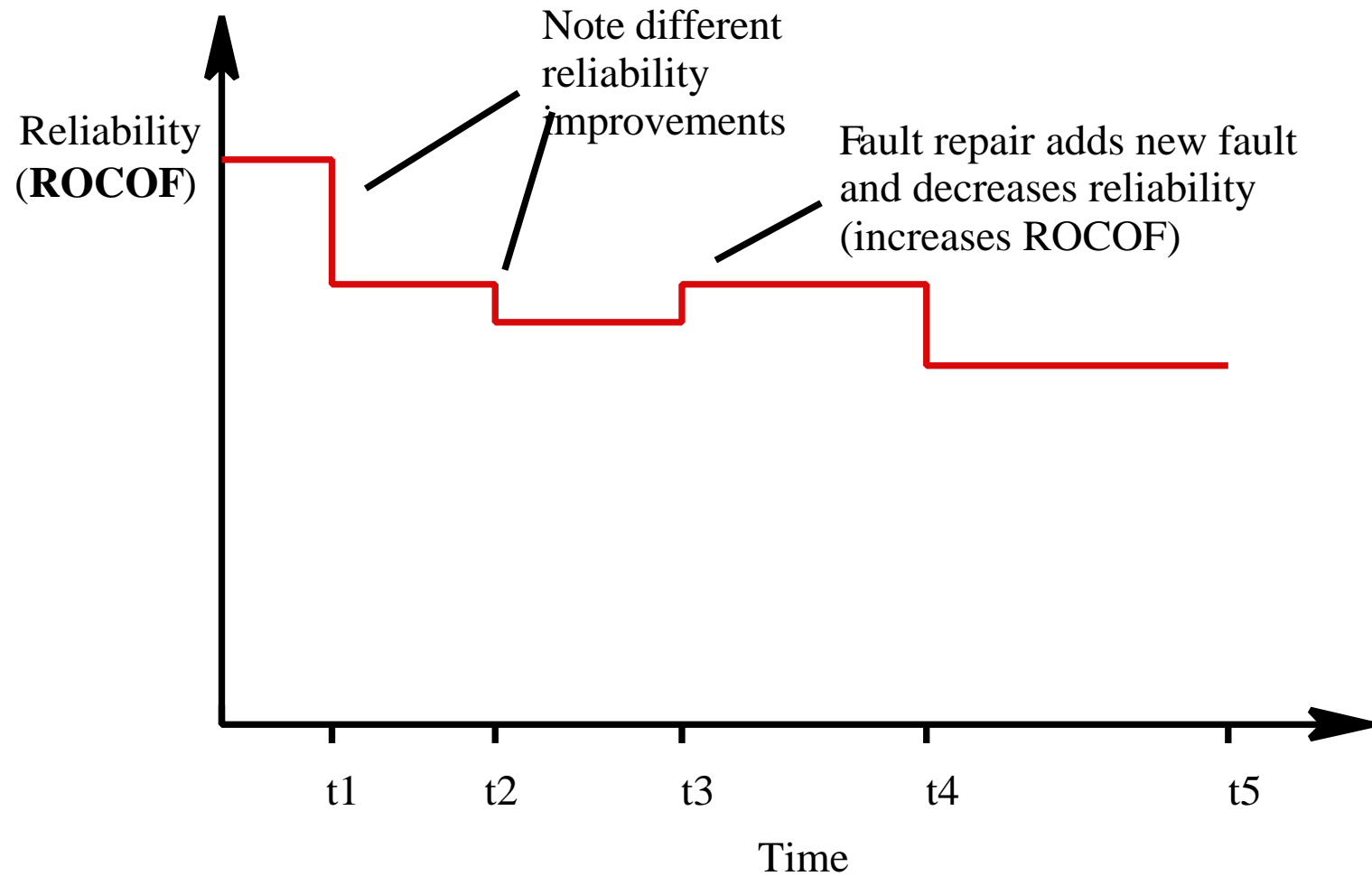
Equal-step reliability growth



Observed reliability growth

- Simple equal-step model but does not 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 may be more accurate

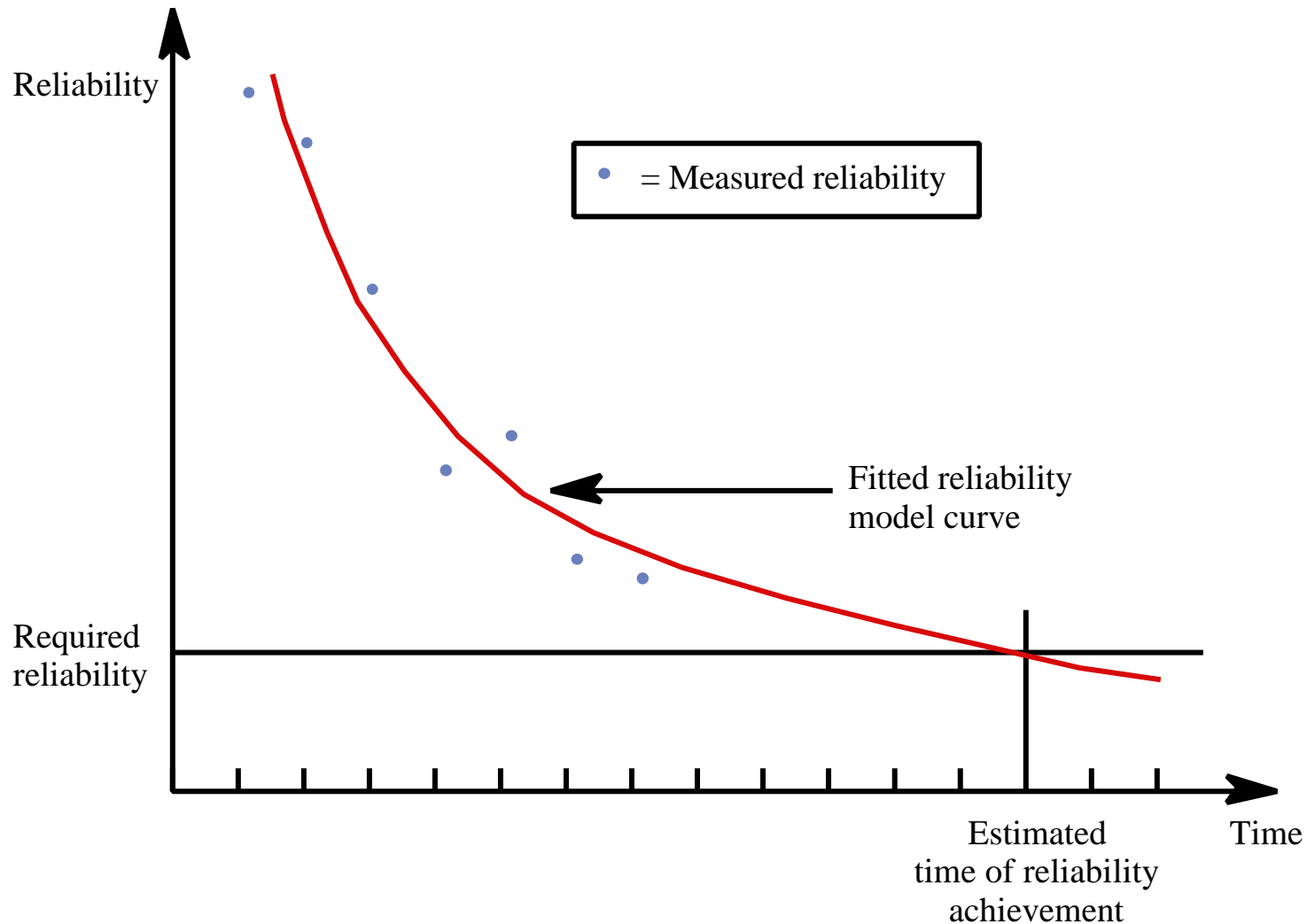
Random-step reliability growth



Growth model selection

- Many different reliability growth models have been proposed
- No universally applicable growth model
- Reliability should be measured and observed data should be fitted to several models
- Best-fit model should be used for reliability prediction

Reliability prediction



Reliability validation problems

- Operational profile uncertainty
 - Is the operational profile an accurate reflection of the real use of the system
- High costs of test data generation
 - Very expensive to generate and check the large number of test cases that are required
- Statistical uncertainty for high-reliability systems
 - It may be impossible to generate enough failures to draw statistically valid conclusions

Security validation

- Security validation has something in common with safety validation
- 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; Safety problems are 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.

Key points

- Statistical testing supplements the defect testing process and is intended to measure the reliability of a system
- Reliability validation 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

The portable insulin pump

Validating the safety of the insulin pump system

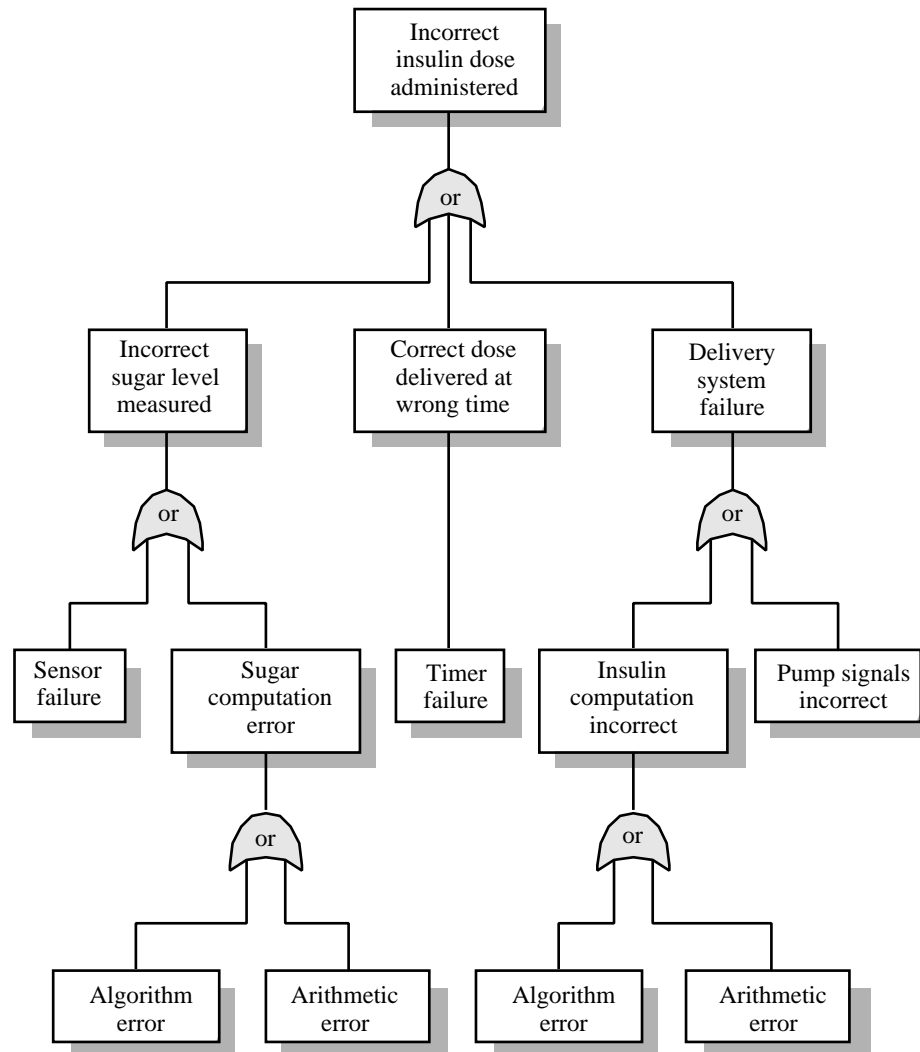
Safety validation

- Design validation
 - Checking the design to ensure that hazards do not arise or that they can be handled without causing an accident.
- Code validation
 - Testing the system to check the conformance of the code to its specification and to check that the code is a true implementation of the design.
- Run-time validation
 - Designing safety checks while the system is in operation to ensure that it does not reach an unsafe state.

Insulin system hazards

- insulin overdose or underdose (biological)
- power failure (electrical)
- machine interferes electrically with other medical equipment such as a heart pacemaker (electrical)
- parts of machine break off in patient's body(physical)
- infection caused by introduction of machine (biol.)
- allergic reaction to the materials or insulin used in the machine (biol.).

Fault tree for software hazards



Safety proofs

- Safety proofs are intended to show that the system cannot reach in unsafe state
- Weaker than correctness proofs which must show that the system code conforms to its specification
- Generally based on proof by contradiction
 - Assume that an unsafe state can be reached
 - Show that this is contradicted by the program code

Insulin delivery system

- Safe state is a shutdown state where no insulin is delivered
 - If hazard arises, shutting down the system will prevent an accident
- Software may be included to detect and prevent hazards such as power failure
- Consider only hazards arising from software failure
 - Arithmetic error The insulin dose is computed incorrectly because of some failure of the computer arithmetic
 - Algorithmic error The dose computation algorithm is incorrect

Arithmetic errors

- Use language exception handling mechanisms to trap errors as they arise
- Use explicit error checks for all errors which are identified
- Avoid error-prone arithmetic operations (multiply and divide). Replace with add and subtract
- Never use floating-point numbers
- Shut down system if exception detected (safe state)

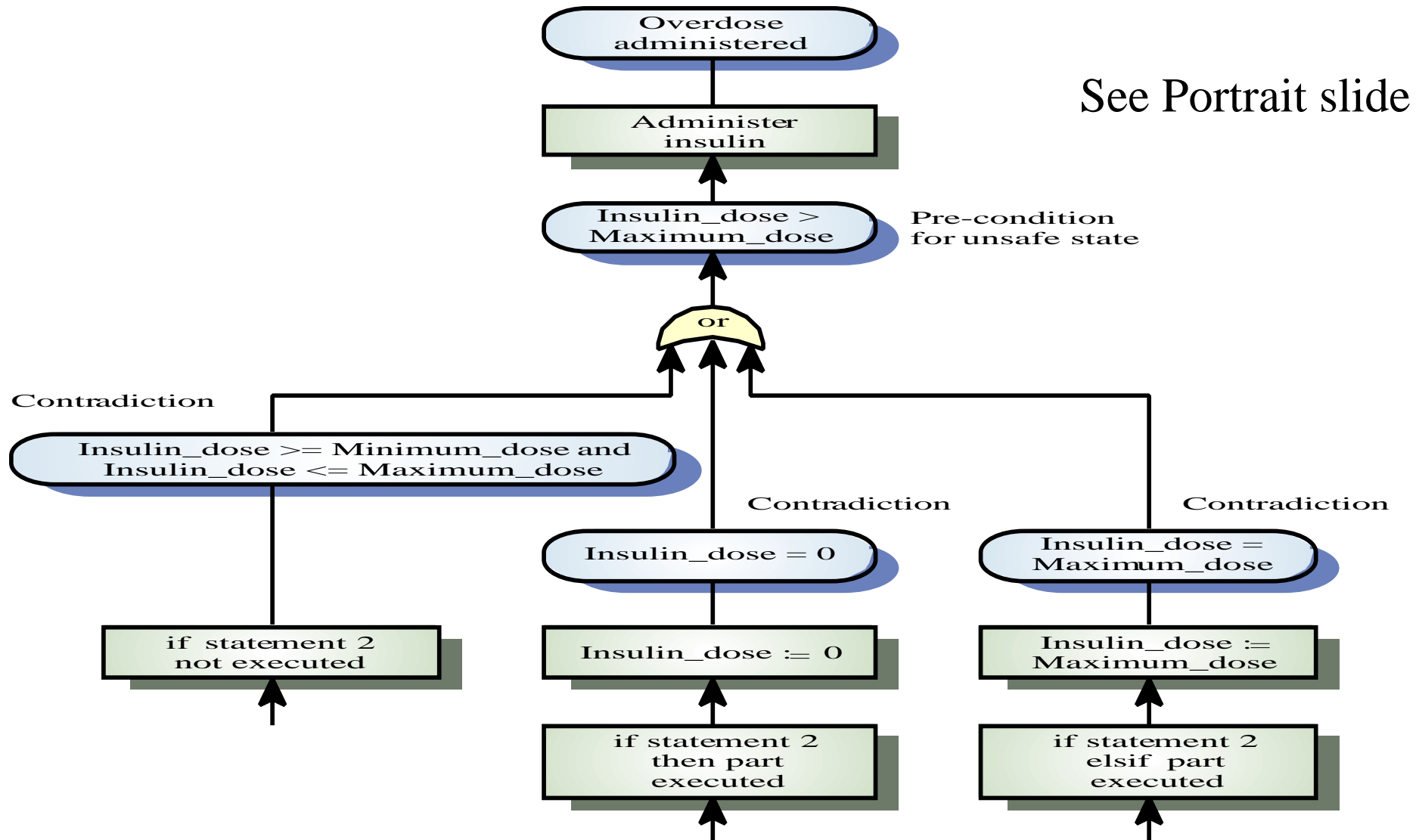
Algorithmic errors

- Harder to detect than arithmetic errors. System should always err on the side of safety
- Use reasonableness checks for the dose delivered based on previous dose and rate of dose change
- Set maximum delivery level in any specified time period
- If computed dose is very high, medical intervention may be necessary anyway because the patient may be ill

Insulin delivery code

```
// The insulin dose to be delivered is a function of blood sugar level, the previous dose
// delivered and the time of delivery of the previous dose
currentDose = computeInsulin () ;
// Safety check - adjust currentDose if necessary
if (previousDose == 0)                                // if statement 1
{
    if (currentDose > 16)
        currentDose = 16 ;
}
else
    if (currentDose > (previousDose * 2) )
        currentDose = previousDose * 2 ;
if ( currentDose < minimumDose )                        // if statement 2
    currentDose = 0 ;                                  // then branch
else if ( currentDose > maxDose )                      // else branch
    currentDose = maxDose ;
administerInsulin (currentDose) ;
```

Informal safety proof



System testing

- System testing of the software has to rely on simulators for the sensor and the insulin delivery components.
- Test for normal operation using an operational profile. Can be constructed using data gathered from existing diabetics
- Testing has to include situations where rate of change of glucose is very fast and very slow
- Test for exceptions using the simulator

Safety assertions

- Predicates included in the program indicating conditions which should hold at that point
- May be based on pre-computed limits e.g. number of insulin pump increments in maximum dose
- Used in formal program inspections or may be pre-processed into safety checks that are executed when the system is in operation

Safety 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
```

Managing people

- Managing people working as individuals and in groups

Objectives

- To describe simple models of human cognition and their relevance for software managers
- To explain the key issues that determine the success or otherwise of team working
- To discuss the problems of selecting and retaining technical staff
- To introduce the people capability maturity model (P-CMM)

Topics covered

- Limits to thinking
- Group working
- Choosing and keeping people
- The people capability maturity model

People in the process

- People are an organisation's most important assets
- The tasks of a manager are essentially people oriented. Unless there is some understanding of people, management will be unsuccessful
- Software engineering is primarily a cognitive activity. Cognitive limitations effectively limit the software process

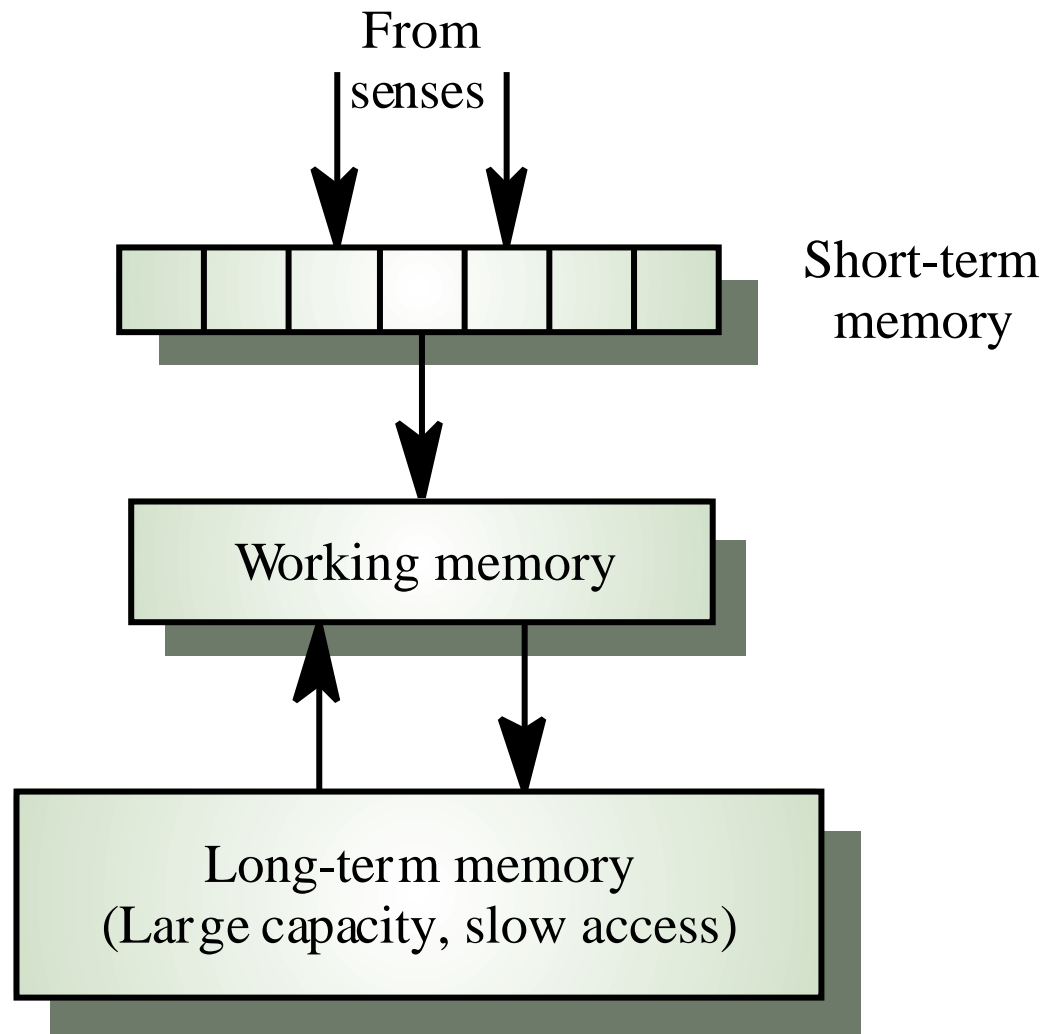
Management activities

- Problem solving (using available people)
- Motivating (people who work on a project)
- Planning (what people are going to do)
- Estimating (how fast people will work)
- Controlling (people's activities)
- Organising (the way in which people work)

Limits to thinking

- People don't all think the same way but everyone is subject to some basic constraints on their thinking due to
 - Memory organisation
 - Knowledge representation
 - Motivation influences
- If we understand these constraints, we can understand how they affect people participating in the software process

Memory organisation



Short-term memory

- Fast access, limited capacity
- 5-7 locations
- Holds 'chunks' of information where the size of a chunk may vary depending on its familiarity
- Fast decay time

Working memory

- Larger capacity, longer access time
- Memory area used to integrate information from short-term memory and long-term memory.
- Relatively fast decay time.

Long-term memory

- Slow access, very large capacity
- Unreliable retrieval mechanism
- Slow but finite decay time - information needs reinforced
- Relatively high threshold - work has to be done to get information into long-term memory.

Information transfer

- Problem solving usually requires transfer between short-term memory and working memory
- Information may be lost or corrupted during this transfer
- Information processing occurs in the transfer from short-term to long-term memory

Cognitive chunking

Loop (process entire array)

Loop (process unsorted part of array)

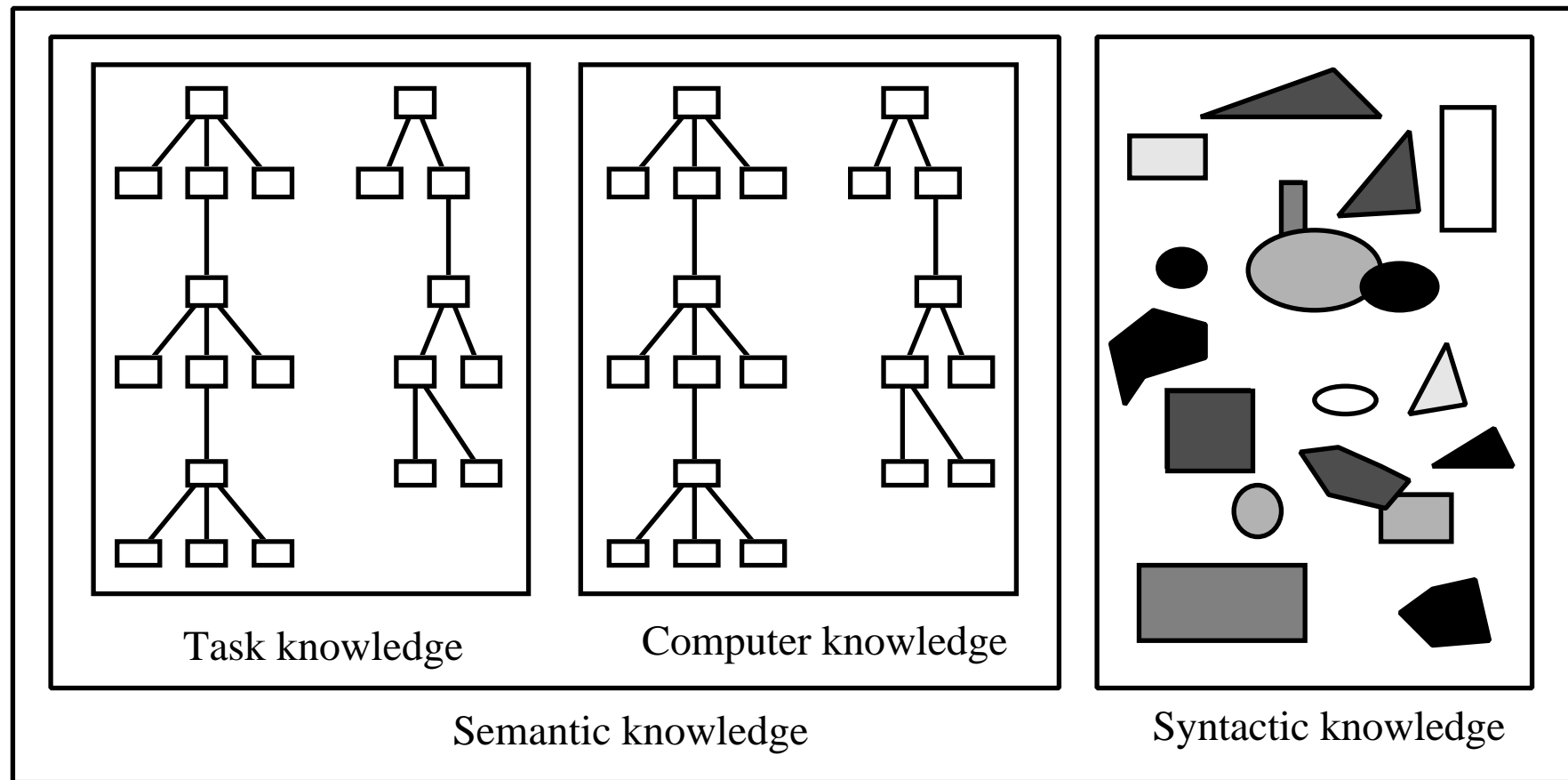
Compare adjacent elements

Swap if necessary so that smaller comes first

Knowledge modelling

- *Semantic knowledge* knowledge of concepts such as the operation of assignment, concept of parameter passing etc.
- *Syntactic knowledge* knowledge of details of a representation e.g. an Ada while loop.
- Semantic knowledge seems to be stored in a structured, representation independent way.

Syntactic/semantic knowledge



Knowledge acquisition

- Semantic knowledge through experience and active learning - the 'ah' factor
- Syntactic knowledge acquired by memorisation.
- New syntactic knowledge can interfere with existing syntactic knowledge.
 - Problems arise for experienced programmers in mixing up syntax of different programming languages

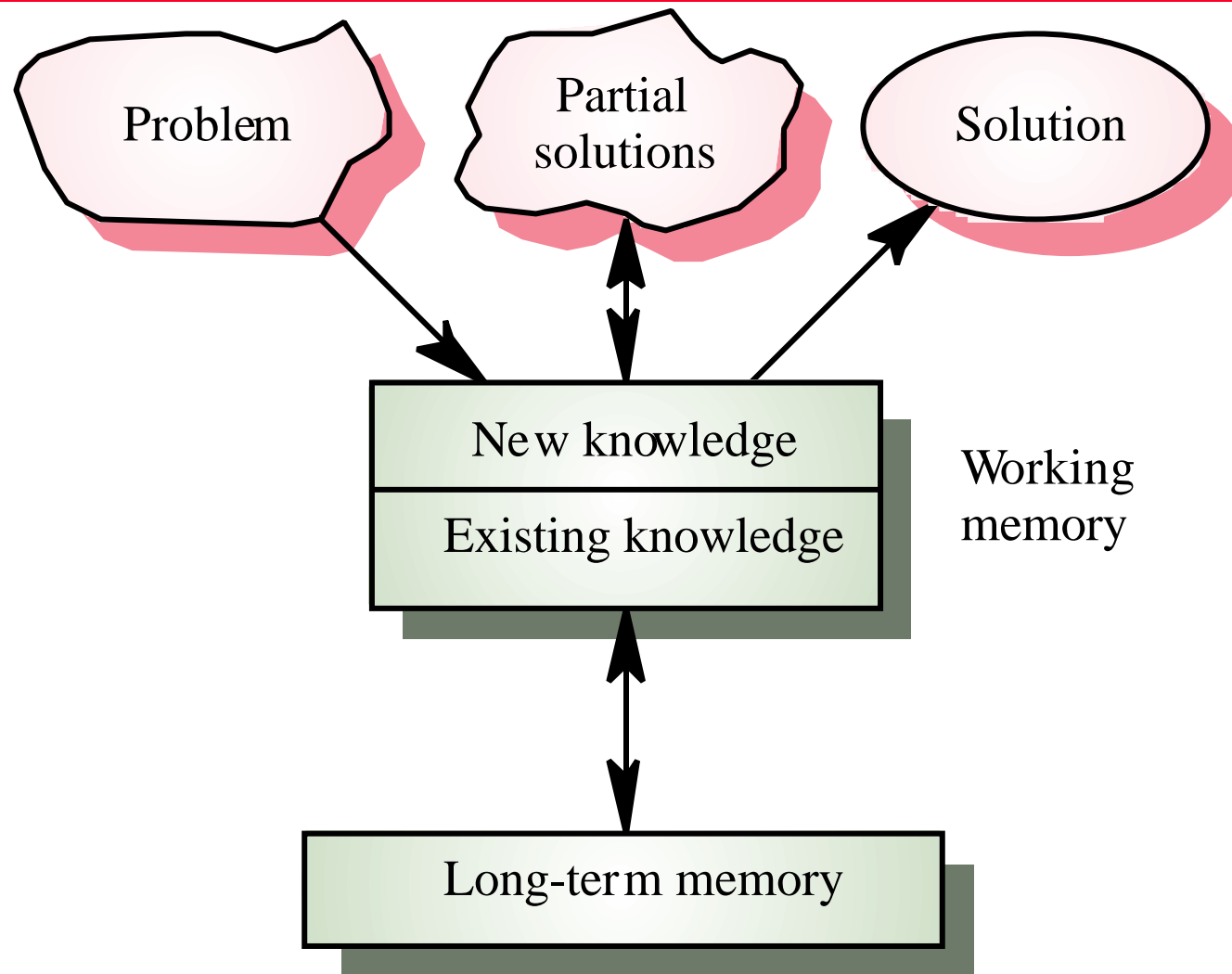
Semantic knowledge

- *Computing concepts* - notion of a writable store, iteration, concept of an object, etc.
- *Task concepts* - principally algorithmic - how to tackle a particular task
- Software development ability is the ability to integrate new knowledge with existing computer and task knowledge and hence derive creative problem solutions
- Thus, problem solving is language independent

Problem solving

- Requires the integration of different types of knowledge (computer, task, domain, organisation)
- Development of a semantic model of the solution and testing of this model against the problem
- Representation of this model in an appropriate notation or programming language

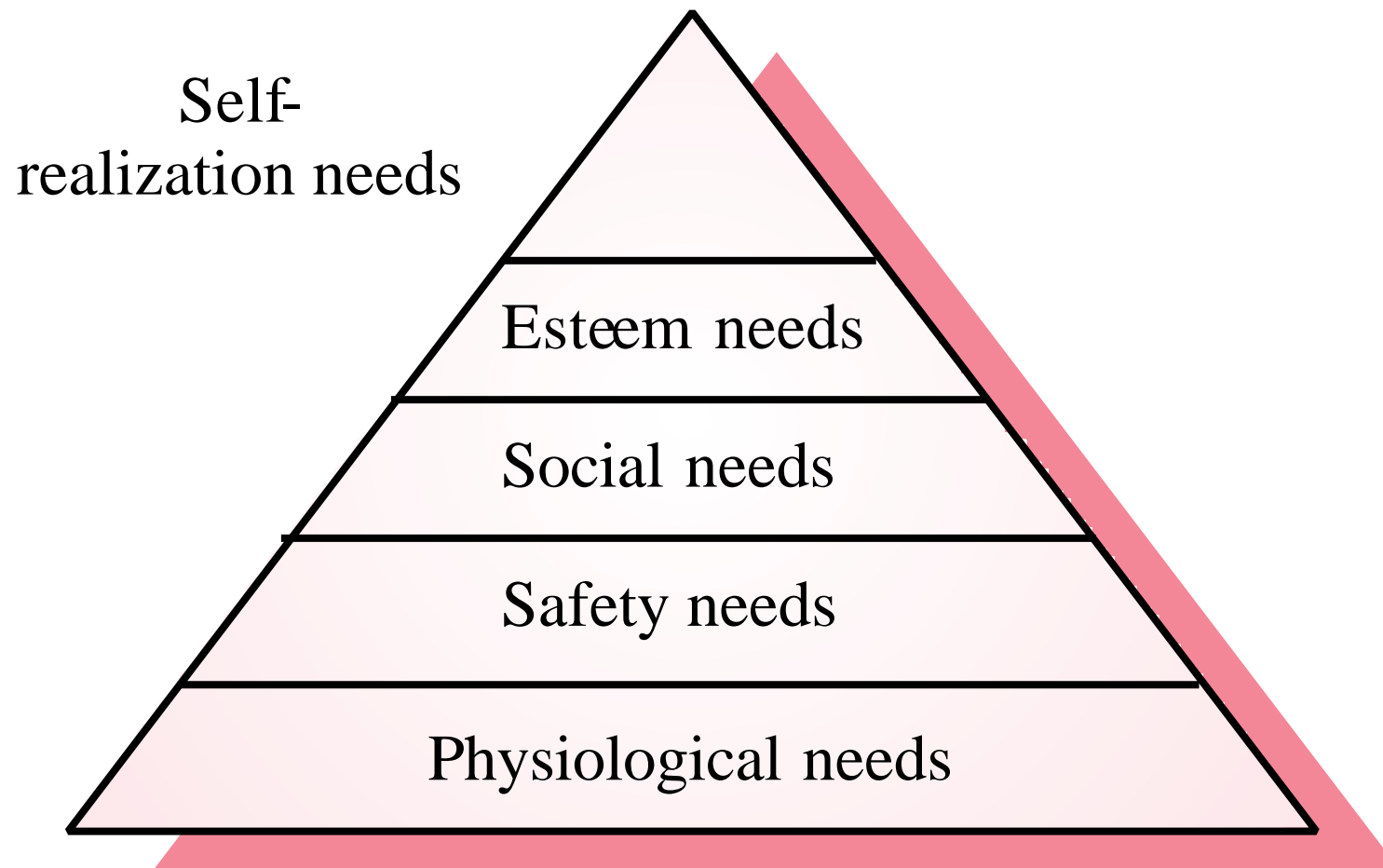
Problem solving



Motivation

- An important role of a manager is to motivate the people working on a project
- Motivation is a complex issue but it appears that there are different types of motivation based on
 - Basic needs (e.g. food, sleep, etc.)
 - Personal needs (e.g. respect, self-esteem)
 - Social needs (e.g. to be accepted as part of a group)

Human needs hierarchy



Motivating people

- Motivations depend on satisfying needs
- It can be assumed that physiological and safety needs are satisfied
- Social, esteem and self-realization needs are most significant from a managerial viewpoint

Need satisfaction

- Social
 - Provide communal facilities
 - Allow informal communications
- Esteem
 - Recognition of achievements
 - Appropriate rewards
- Self-realization
 - Training - people want to learn more
 - Responsibility

Personality types

- The needs hierarchy is almost certainly an oversimplification
- Motivation should also take into account different personality types:
 - Task-oriented
 - Self-oriented
 - Interaction-oriented

Personality types

- Task-oriented.
 - The motivation for doing the work is the work itself
- Self-oriented.
 - The work is a means to an end which is the achievement of individual goals - e.g. to get rich, to play tennis, to travel etc.
- Interaction-oriented
 - The principal motivation is the presence and actions of co-workers. People go to work because they like to go to work

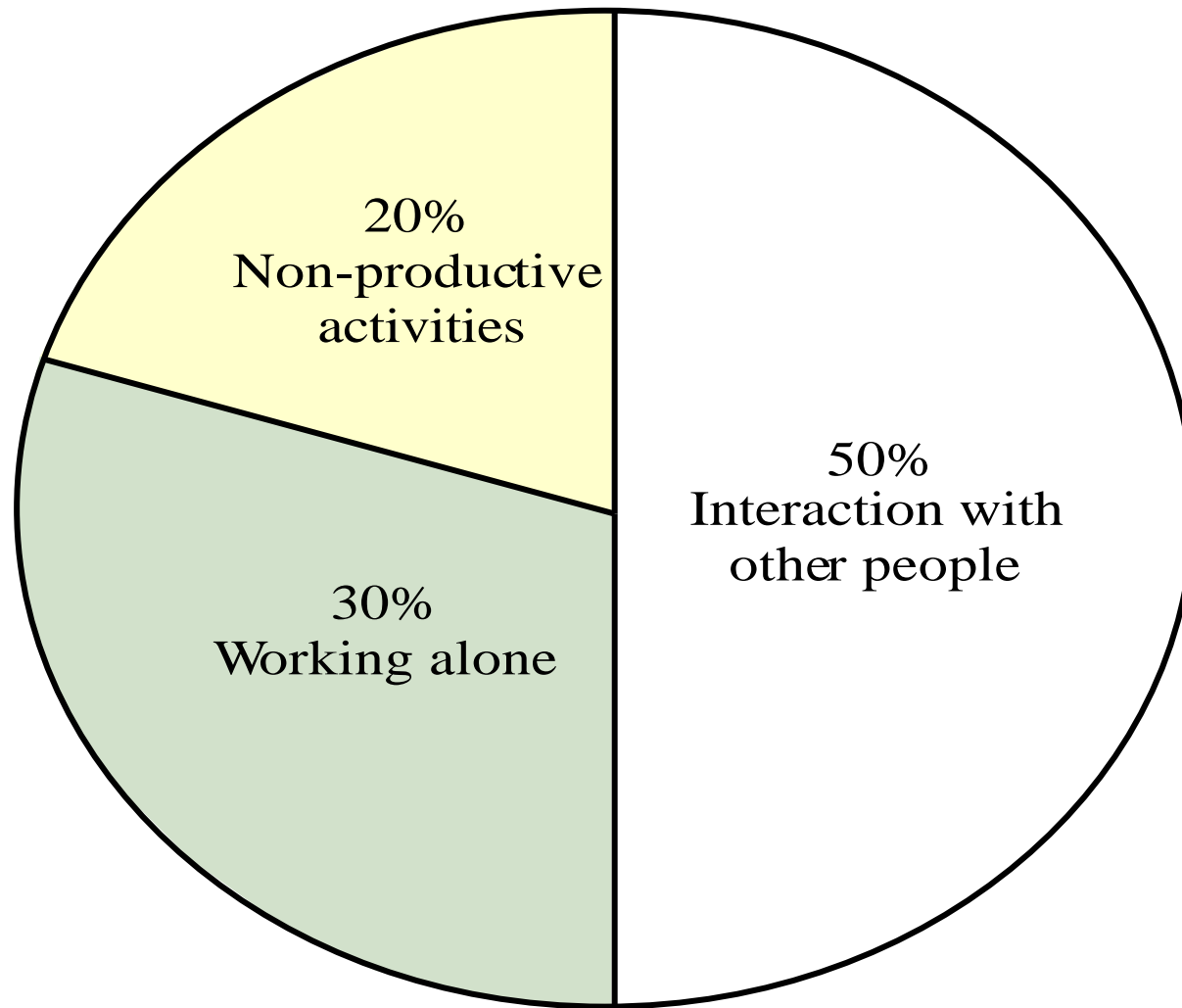
Motivation balance

- Individual motivations are made up of elements of each class
- Balance can change depending on personal circumstances and external events
- However, people are not just motivated by personal factors but also by being part of a group and culture.
- People go to work because they are motivated by the people that they work with

Group working

- Most software engineering is a group activity
 - The development schedule for most non-trivial software projects is such that they cannot be completed by one person working alone
- Group interaction is a key determinant of group performance
- Flexibility in group composition is limited
 - Managers must do the best they can with available people

Time distribution



Group composition

- Group composed of members who share the same motivation can be problematic
 - Task-oriented - everyone wants to do their own thing
 - Self-oriented - everyone wants to be the boss
 - Interaction-oriented - too much chatting, not enough work
- An effective group has a balance of all types
- Can be difficult to achieve because most engineers are task-oriented
- Need for all members to be involved in decisions which affect the group

Group leadership

- Leadership depends on respect not titular status
- There may be both a technical and an administrative leader
- Democratic leadership is more effective than autocratic leadership
- A career path based on technical competence should be supported

Group cohesiveness

- In a cohesive group, members consider the group to be more important than any individual in it
- Advantages of a cohesive group are:
 - Group quality standards can be developed
 - Group members work closely together so inhibitions caused by ignorance are reduced
 - Team members learn from each other and get to know each other's work
 - Egoless programming where members strive to improve each other's programs can be practised

Developing cohesiveness

- Cohesiveness is influenced by factors such as the organisational culture and the personalities in the group
- Cohesiveness can be encouraged through
 - Social events
 - Developing a group identity and territory
 - Explicit team-building activities
- Openness with information is a simple way of ensuring all group members feel part of the group

Group loyalties

- Group members tend to be loyal to cohesive groups
- 'Groupthink' is preservation of group irrespective of technical or organizational considerations
- Management should act positively to avoid groupthink by forcing external involvement with each group

Group communications

- Good communications are essential for effective group working
- Information must be exchanged on the status of work, design decisions and changes to previous decisions
- Good communications also strengthens group cohesion as it promotes understanding

Group communications

- Status of group members
 - Higher status members tend to dominate conversations
- Personalities in groups
 - Too many people of the same personality type can be a problem
- Sexual composition of group
 - Mixed-sex groups tend to communicate better
- Communication channels
 - Communications channelled through a central coordinator tend to be ineffective

Group organisation

- Software engineering group sizes should be relatively small (< 8 members)
- Break big projects down into multiple smaller projects
- Small teams may be organised in an informal, democratic way
- Chief programmer teams try to make the most effective use of skills and experience

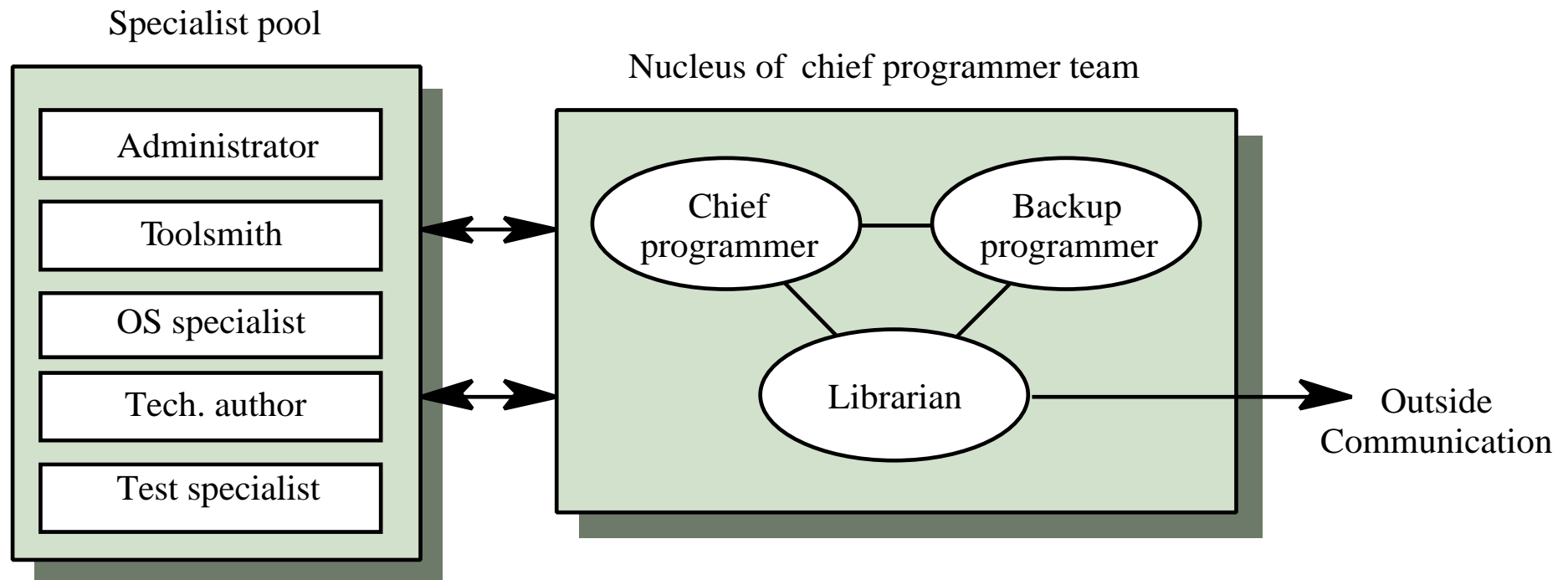
Democratic team organisation

- The group acts as a whole and comes to a consensus on decisions affecting the system
- The group leader serves as the external interface of the group but does not allocate specific work items
- Rather, work is discussed by the group as a whole and tasks are allocated according to ability and experience
- This approach is successful for groups where all members are experienced and competent

Extreme programming groups

- Extreme programming groups are variants of democratic organisation
- In extreme programming groups, some ‘management’ decisions are devolved to group members
- Programmers work in pairs and take a collective responsibility for code that is developed

Chief programmer teams



Chief programmer teams

- Consist of a kernel of specialists helped by others added to the project as required
- The motivation behind their development is the wide difference in ability in different programmers
- Chief programmer teams provide a supporting environment for very able programmers to be responsible for most of the system development

Problems

- This chief programmer approach, in different forms, has undoubtedly been successful
- However, it suffers from a number of problems
 - Talented designers and programmers are hard to find. Without exception people in these roles, the approach will fail
 - Other group members may resent the chief programmer taking the credit for success so may deliberately undermine his/her role
 - High project risk as the project will fail if both the chief and deputy programmer are unavailable
 - Organisational structures and grades may be unable to accommodate this type of group

Choosing and keeping people

- Choosing people to work on a project is a major managerial responsibility
- Appointment decisions are usually based on
 - information provided by the candidate (their resumé or CV)
 - information gained at an interview
 - recommendations from other people who know the candidate
- Some companies use psychological or aptitude tests
 - There is no agreement on whether or not these tests are actually useful

Factor	Explanation
Application domain experience	For a project to develop a successful system, the developers must understand the application domain.
Platform experience	May be significant if low-level programming is involved. Otherwise, not usually a critical attribute.
Programming language experience	Normally only significant for short duration projects where there is insufficient time to learn a new language.
Educational background	May provide an indicator of the basic fundamentals which the candidate should know and of their ability to learn. This factor becomes increasingly irrelevant as engineers gain experience across a range of projects.
Communication ability	Very important because of the need for project staff to communicate orally and in writing with other engineers, managers and customers.
Adaptability	Adaptability may be judged by looking at the different types of experience which candidates have had. This is an important attribute as it indicates an ability to learn.
Attitude	Project staff should have a positive attitude to their work and should be willing to learn new skills. This is an important attribute but often very difficult to assess.
Personality	Again, an important attribute but difficult to assess. Candidates must be reasonably compatible with other team members. No particular type of personality is more or less suited to software engineering.

Staff selection
factors

Working environments

- Physical workplace provision has an important effect on individual productivity and satisfaction
 - Comfort
 - Privacy
 - Facilities
- Health and safety considerations must be taken into account
 - Lighting
 - Heating
 - Furniture

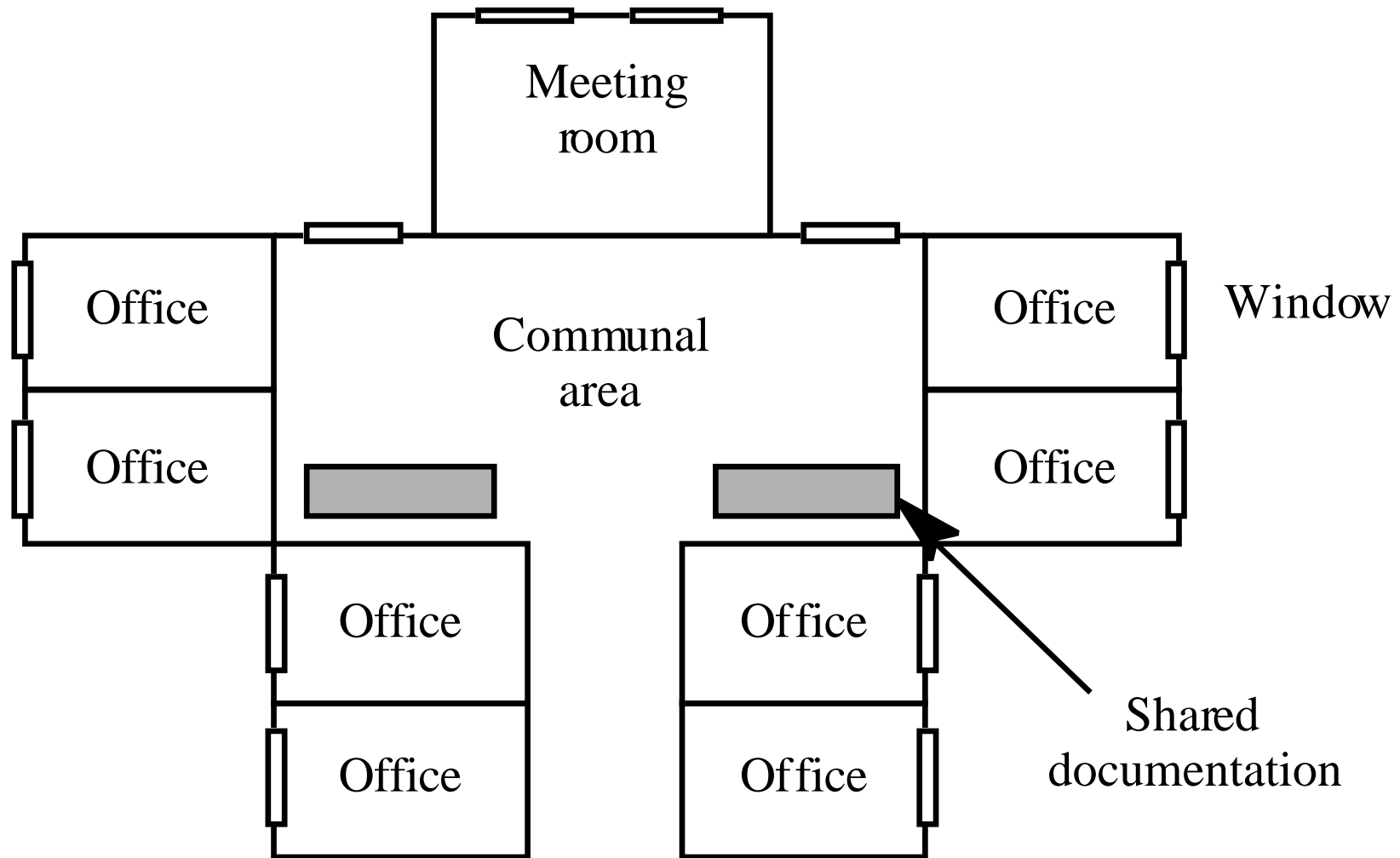
Environmental factors

- Privacy - each engineer requires an area for uninterrupted work
- Outside awareness - people prefer to work in natural light
- Personalization - individuals adopt different working practices and like to organize their environment in different ways

Workspace organisation

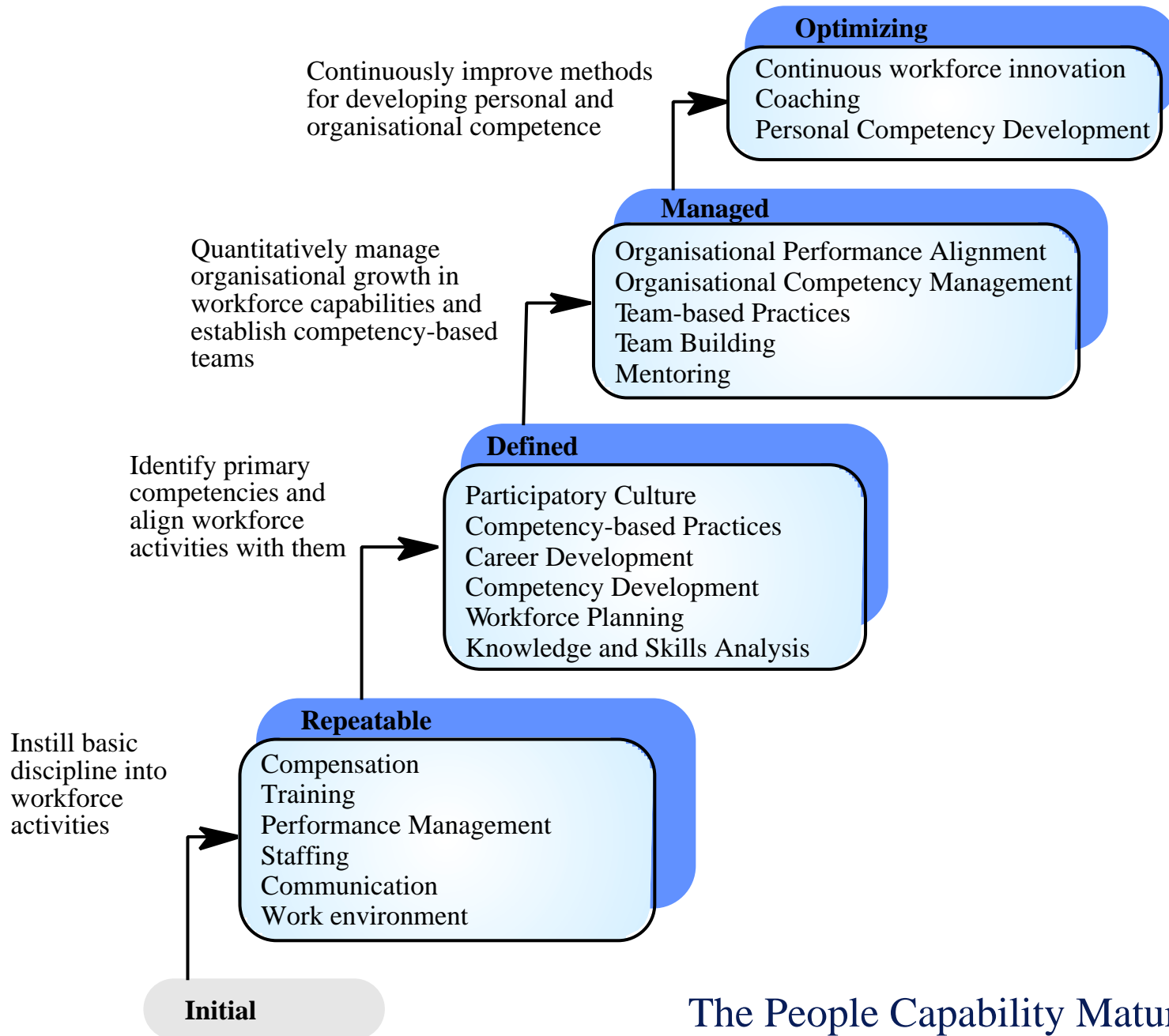
- Workspaces should provide private spaces where people can work without interruption
 - Providing individual offices for staff has been shown to increase productivity
- However, teams working together also require spaces where formal and informal meetings can be held

Office layout



The People Capability Maturity Model

- Intended as a framework for managing the development of people involved in software development
- Five stage model
 - Initial. Ad-hoc people management
 - Repeatable. Policies developed for capability improvement
 - Defined. Standardised people management across the organisation
 - Managed. Quantitative goals for people management in place
 - Optimizing. Continuous focus on improving individual competence and workforce motivation



The People Capability Maturity Model

P-CMM Objectives

- To improve organisational capability by improving workforce capability
- To ensure that software development capability is not reliant on a small number of individuals
- To align the motivation of individuals with that of the organisation
- To help retain people with critical knowledge and skills

Key points

- Managers must have some understanding of human factors to avoid making unrealistic demands on people
- Problem solving involves integrating information from long-term memory with new information from short-term memory
- Staff selection factors include education, domain experience, adaptability and personality

Key points

- Software development groups should be small and cohesive
- Group communications are affected by status, group size, group organisation and the sexual composition of the group
- The working environment has a significant effect on productivity
- The People Capability Maturity Model is a framework for improving the capabilities of staff in an organisation

Software cost estimation

- Predicting the resources required for a software development process

Objectives

- To introduce the fundamentals of software costing and pricing
- To describe three metrics for software productivity assessment
- To explain why different techniques should be used for software estimation
- To describe the COCOMO 2 algorithmic cost estimation model

Topics covered

- Productivity
- Estimation techniques
- Algorithmic cost modelling
- Project duration and staffing

Fundamental estimation questions

- How much effort is required to complete an activity?
- How much calendar time is needed to complete an activity?
- What is the total cost of an activity?
- Project estimation and scheduling and interleaved management activities

Software cost components

- Hardware and software costs
- Travel and training costs
- Effort costs (the dominant factor in most projects)
 - salaries of engineers involved in the project
 - Social and insurance costs
- Effort costs must take overheads into account
 - costs of building, heating, lighting
 - costs of networking and communications
 - costs of shared facilities (e.g library, staff restaurant, etc.)

Costing and pricing

- Estimates are made to discover the cost, to the developer, of producing a software system
- There is not a simple relationship between the development cost and the price charged to the customer
- Broader organisational, economic, political and business considerations influence the price charged

Software pricing factors

Factor	Description
Market opportunity	A development organisation may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the opportunity of more profit later. The experience gained may allow new products to be developed.
Cost estimate uncertainty	If an organisation is unsure of its cost estimate, it may increase its price by some contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organisation may lower its price to win a contract. After the contract is awarded, high prices may be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a small profit or break even than to go out of business.

Programmer productivity

- A measure of the rate at which individual engineers involved in software development produce software and associated documentation
- Not quality-oriented although quality assurance is a factor in productivity assessment
- Essentially, we want to measure useful functionality produced per time unit

Productivity measures

- Size related measures based on some output from the software process. This may be lines of delivered source code, object code instructions, etc.
- Function-related measures based on an estimate of the functionality of the delivered software. Function-points are the best known of this type of measure

Measurement problems

- Estimating the size of the measure
- Estimating the total number of programmer months which have elapsed
- Estimating contractor productivity (e.g. documentation team) and incorporating this estimate in overall estimate

Lines of code

- What's a line of code?
 - The measure was first proposed when programs were typed on cards with one line per card
 - How does this correspond to statements as in Java which can span several lines or where there can be several statements on one line
- What programs should be counted as part of the system?
- Assumes linear relationship between system size and volume of documentation

Productivity comparisons

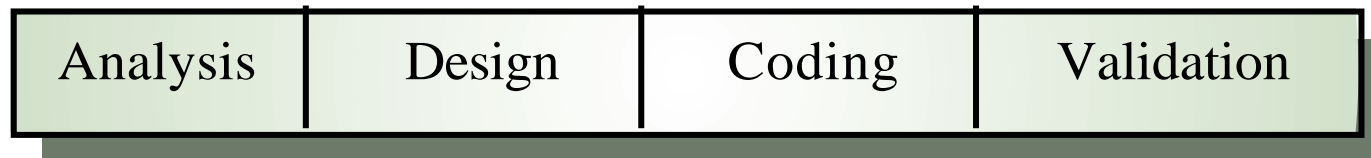
- The lower level the language, the more productive the programmer
 - The same functionality takes more code to implement in a lower-level language than in a high-level language
- The more verbose the programmer, the higher the productivity
 - Measures of productivity based on lines of code suggest that programmers who write verbose code are more productive than programmers who write compact code

High and low level languages

Low-level language



High-level language



System development times

	Analysis	Design	Coding	Testing	Documentation
Assembly code	3 weeks	5 weeks	8 weeks	10 weeks	2 weeks
High-level language	3 weeks	5 weeks	8 weeks	6 weeks	2 weeks
	Size	Effort	Productivity		
Assembly code	5000 lines	28 weeks	714 lines/month		
High-level language	1500 lines	20 weeks	300 lines/month		

Function points

- Based on a combination of program characteristics
 - external inputs and outputs
 - user interactions
 - external interfaces
 - files used by the system
- A weight is associated with each of these
- The function point count is computed by multiplying each raw count by the weight and summing all values

Function points

- Function point count modified by complexity of the project
- FPs can be used to estimate LOC depending on the average number of LOC per FP for a given language
 - $LOC = AVC * \text{number of function points}$
 - AVC is a language-dependent factor varying from 200-300 for assemble language to 2-40 for a 4GL
- FPs are very subjective. They depend on the estimator.
 - Automatic function-point counting is impossible

Object points

- Object points are an alternative function-related measure to function points when 4GLs or similar languages are used for development
- Object points are NOT the same as object classes
- The number of object points in a program is a weighted estimate of
 - The number of separate screens that are displayed
 - The number of reports that are produced by the system
 - The number of 3GL modules that must be developed to supplement the 4GL code

Object point estimation

- Object points are easier to estimate from a specification than function points as they are simply concerned with screens, reports and 3GL modules
- They can therefore be estimated at an early point in the development process. At this stage, it is very difficult to estimate the number of lines of code in a system

Productivity estimates

- Real-time embedded systems, 40-160 LOC/P-month
- Systems programs , 150-400 LOC/P-month
- Commercial applications, 200-800 LOC/P-month
- In object points, productivity has been measured between 4 and 50 object points/month depending on tool support and developer capability

Factors affecting productivity

Factor	Description
Application domain experience	Knowledge of the application domain is essential for effective software development. Engineers who already understand a domain are likely to be the most productive.
Process quality	The development process used can have a significant effect on productivity. This is covered in Chapter 31.
Project size	The larger a project, the more time required for team communications. Less time is available for development so individual productivity is reduced.
Technology support	Good support technology such as CASE tools, supportive configuration management systems, etc. can improve productivity.
Working environment	As discussed in Chapter 28, a quiet working environment with private work areas contributes to improved productivity.

Quality and productivity

- All metrics based on volume/unit time are flawed because they do not take quality into account
- Productivity may generally be increased at the cost of quality
- It is not clear how productivity/quality metrics are related
- If change is constant then an approach based on counting lines of code is not meaningful

Estimation techniques

- There is no simple way to make an accurate estimate of the effort required to develop a software system
 - Initial estimates are based on inadequate information in a user requirements definition
 - The software may run on unfamiliar computers or use new technology
 - The people in the project may be unknown
- Project cost estimates may be self-fulfilling
 - The estimate defines the budget and the product is adjusted to meet the budget

Estimation techniques

- Algorithmic cost modelling
- Expert judgement
- Estimation by analogy
- Parkinson's Law
- Pricing to win

Algorithmic code modelling

- A formulaic approach based on historical cost information and which is generally based on the size of the software
- Discussed later in this chapter

Expert judgement

- One or more experts in both software development and the application domain use their experience to predict software costs. Process iterates until some consensus is reached.
- Advantages: Relatively cheap estimation method. Can be accurate if experts have direct experience of similar systems
- Disadvantages: Very inaccurate if there are no experts!

Estimation by analogy

- The cost of a project is computed by comparing the project to a similar project in the same application domain
- Advantages: Accurate if project data available
- Disadvantages: Impossible if no comparable project has been tackled. Needs systematically maintained cost database

Parkinson's Law

- The project costs whatever resources are available
- Advantages: No overspend
- Disadvantages: System is usually unfinished

Pricing to win

- The project costs whatever the customer has to spend on it
- Advantages: You get the contract
- Disadvantages: The probability that the customer gets the system he or she wants is small. Costs do not accurately reflect the work required

Top-down and bottom-up estimation

- Any of these approaches may be used top-down or bottom-up
- Top-down
 - Start at the system level and assess the overall system functionality and how this is delivered through sub-systems
- Bottom-up
 - Start at the component level and estimate the effort required for each component. Add these efforts to reach a final estimate

Top-down estimation

- Usable without knowledge of the system architecture and the components that might be part of the system
- Takes into account costs such as integration, configuration management and documentation
- Can underestimate the cost of solving difficult low-level technical problems

Bottom-up estimation

- Usable when the architecture of the system is known and components identified
- Accurate method if the system has been designed in detail
- May underestimate costs of system level activities such as integration and documentation

Estimation methods

- Each method has strengths and weaknesses
- Estimation should be based on several methods
- If these do not return approximately the same result, there is insufficient information available
- Some action should be taken to find out more in order to make more accurate estimates
- Pricing to win is sometimes the only applicable method

Experience-based estimates

- Estimating is primarily experience-based
- However, new methods and technologies may make estimating based on experience inaccurate
 - Object oriented rather than function-oriented development
 - Client-server systems rather than mainframe systems
 - Off the shelf components
 - Component-based software engineering
 - CASE tools and program generators

Pricing to win

- This approach may seem unethical and unbusinesslike
- However, when detailed information is lacking it may be the only appropriate strategy
- The project cost is agreed on the basis of an outline proposal and the development is constrained by that cost
- A detailed specification may be negotiated or an evolutionary approach used for system development

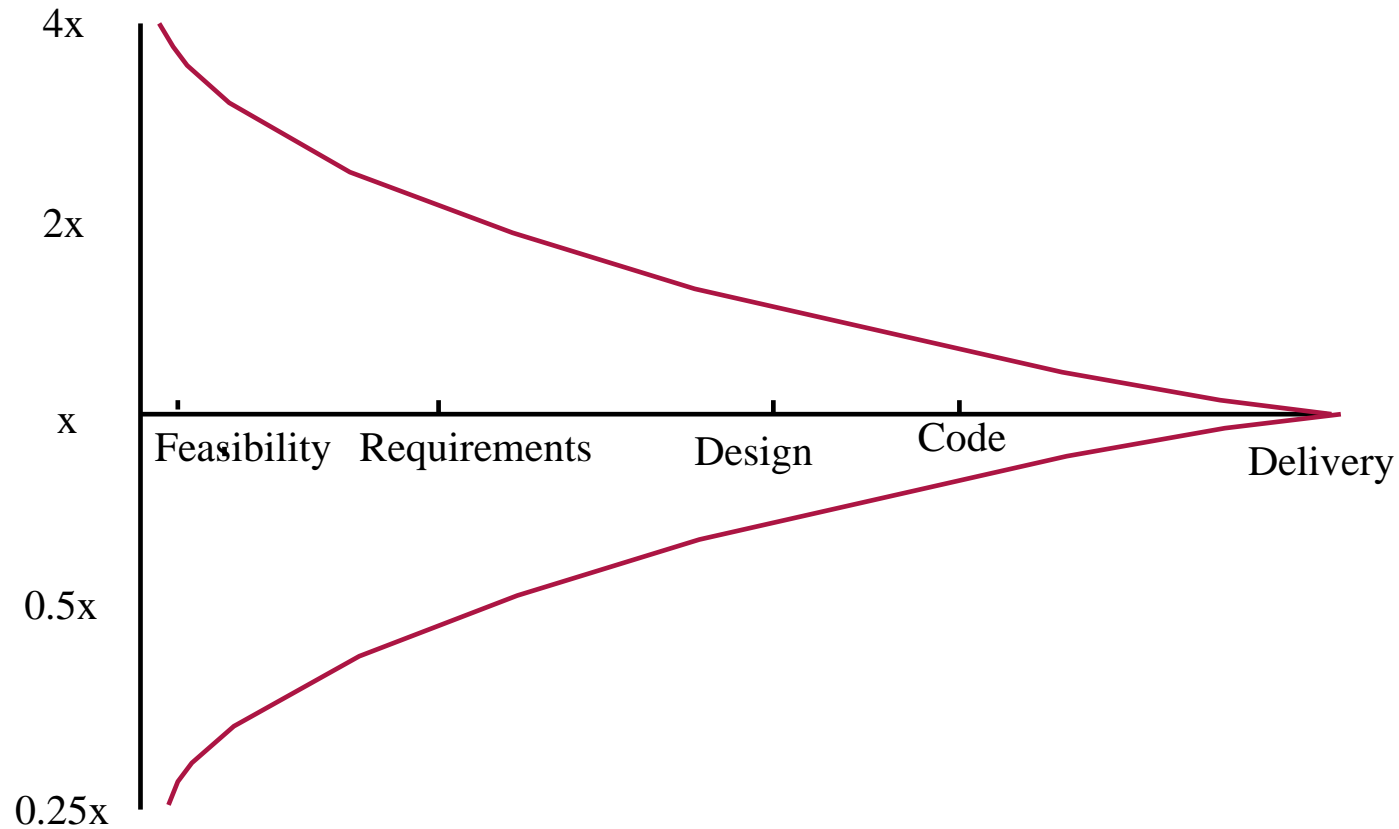
Algorithmic cost modelling

- Cost is estimated as a mathematical function of product, project and process attributes whose values are estimated by project managers
 - $\text{Effort} = A \times \text{Size}^B \times M$
 - A is an organisation-dependent constant, B reflects the disproportionate effort for large projects and M is a multiplier reflecting product, process and people attributes
- Most commonly used product attribute for cost estimation is code size
- Most models are basically similar but with different values for A, B and M

Estimation accuracy

- The size of a software system can only be known accurately when it is finished
- Several factors influence the final size
 - Use of COTS and components
 - Programming language
 - Distribution of system
- As the development process progresses then the size estimate becomes more accurate

Estimate uncertainty



The COCOMO model

- An empirical model based on project experience
- Well-documented, ‘independent’ model which is not tied to a specific software vendor
- Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2
- COCOMO 2 takes into account different approaches to software development, reuse, etc.

COCOMO 81

Project complexity	Formula	Description
Simple	$PM = 2.4 (KDSI)^{1.05} \times M$	Well-understood applications developed by small teams.
Moderate	$PM = 3.0 (KDSI)^{1.12} \times M$	More complex projects where team members may have limited experience of related systems.
Embedded	$PM = 3.6 (KDSI)^{1.20} \times M$	Complex projects where the software is part of a strongly coupled complex of hardware, software, regulations and operational procedures.

COCOMO 2 levels

- COCOMO 2 is a 3 level model that allows increasingly detailed estimates to be prepared as development progresses
- Early prototyping level
 - Estimates based on object points and a simple formula is used for effort estimation
- Early design level
 - Estimates based on function points that are then translated to LOC
- Post-architecture level
 - Estimates based on lines of source code

Early prototyping level

- Supports prototyping projects and projects where there is extensive reuse
- Based on standard estimates of developer productivity in object points/month
- Takes CASE tool use into account
- Formula is
 - $PM = (NOP \times (1 - \%reuse/100)) / PROD$
 - PM is the effort in person-months, NOP is the number of object points and PROD is the productivity

Object point productivity

Developer's experience and capability	Very low	Low	Nominal	High	Very high
ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NOP/month)	4	7	13	25	50

Early design level

- Estimates can be made after the requirements have been agreed
- Based on standard formula for algorithmic models
 - $PM = A \times \text{Size}^B \times M + PM_m$ where
 - $M = \text{PERS} \times \text{RCPX} \times \text{RUSE} \times \text{PDIF} \times \text{PREX} \times \text{FCIL} \times \text{SCED}$
 - $PM_m = (\text{ASLOC} \times (\text{AT}/100)) / \text{ATPROD}$
 - $A = 2.5$ in initial calibration, Size in KLOC, B varies from 1.1 to 1.24 depending on novelty of the project, development flexibility, risk management approaches and the process maturity

Multipliers

- Multipliers reflect the capability of the developers, the non-functional requirements, the familiarity with the development platform, etc.
 - RCPX - product reliability and complexity
 - RUSE - the reuse required
 - PDIF - platform difficulty
 - PREX - personnel experience
 - PERS - personnel capability
 - SCED - required schedule
 - FCIL - the team support facilities
- PM reflects the amount of automatically generated code

Post-architecture level

- Uses same formula as early design estimates
- Estimate of size is adjusted to take into account
 - Requirements volatility. Rework required to support change
 - Extent of possible reuse. Reuse is non-linear and has associated costs so this is not a simple reduction in LOC
 - $ESLOC = ASLOC \times (AA + SU + 0.4DM + 0.3CM + 0.3IM)/100$
 - » ESLOC is equivalent number of lines of new code. ASLOC is the number of lines of reusable code which must be modified, DM is the percentage of design modified, CM is the percentage of the code that is modified, IM is the percentage of the original integration effort required for integrating the reused software.
 - » SU is a factor based on the cost of software understanding, AA is a factor which reflects the initial assessment costs of deciding if software may be reused.

The exponent term

- This depends on 5 scale factors (see next slide). Their sum/100 is added to 1.01
- Example
 - Precedenteness - new project - 4
 - Development flexibility - no client involvement - Very high - 1
 - Architecture/risk resolution - No risk analysis - V. Low - 5
 - Team cohesion - new team - nominal - 3
 - Process maturity - some control - nominal - 3
- Scale factor is therefore 1.17

Exponent scale factors

Scale factor	Explanation
Precedentedness	Reflects the previous experience of the organisation with this type of project. Very low means no previous experience, Extra high means that the organisation is completely familiar with this application domain.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; Extra high means that the client only sets general goals.
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis, Extra high means a complete a thorough risk analysis.
Team cohesion	Reflects how well the development team know each other and work together. Very low means very difficult interactions, Extra high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organisation. The computation of this value depends on the CMM Maturity Questionnaire but an estimate can be achieved by subtracting the CMM process maturity level from 5.

Multipliers

- Product attributes
 - concerned with required characteristics of the software product being developed
- Computer attributes
 - constraints imposed on the software by the hardware platform
- Personnel attributes
 - multipliers that take the experience and capabilities of the people working on the project into account.
- Project attributes
 - concerned with the particular characteristics of the software development project

Project cost drivers

Product attributes			
RELY	Required system reliability	DATA	Size of database used
CPLX	Complexity of system modules	RUSE	Required percentage of reusable components
DOCU	Extent of documentation required		
Computer attributes			
TIME	Execution time constraints	STOR	Memory constraints
PVOL	Volatility of development platform		
Personnel attributes			
ACAP	Capability of project analysts	PCAP	Programmer capability
PCON	Personnel continuity	AEXP	Analyst experience in project domain
PEXP	Programmer experience in project domain	LTEX	Language and tool experience
Project attributes			
TOOL	Use of software tools	SITE	Extent of multi-site working and quality of site communications
SCED	Development schedule compression		

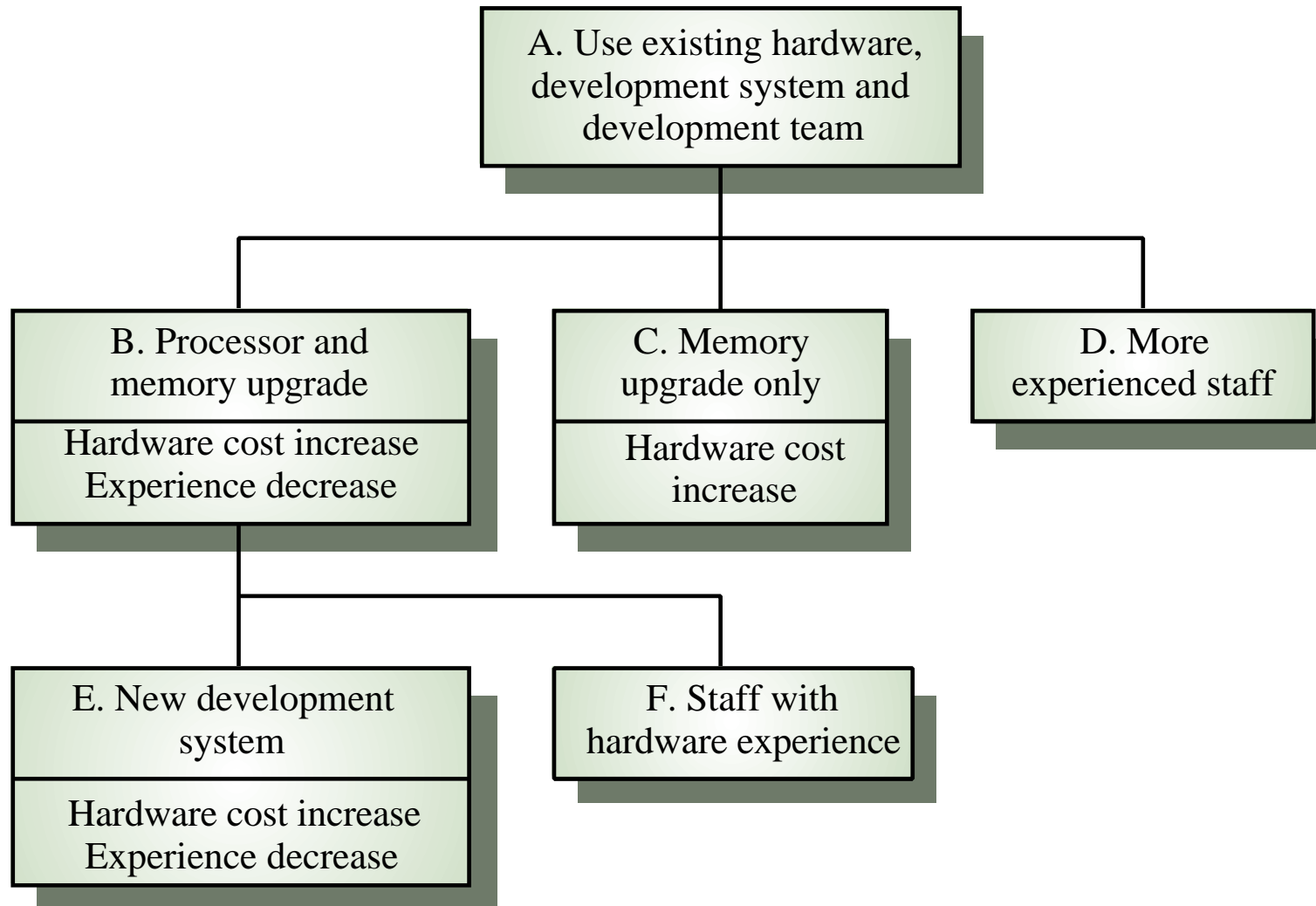
Effects of cost drivers

Exponent value System size (including factors for reuse and requirements volatility) Initial COCOMO estimate without cost drivers	1.17 128, 000 DSI 730 person-months
Reliability Complexity Memory constraint Tool use Schedule Adjusted COCOMO estimate	Very high, multiplier = 1.39 Very high, multiplier = 1.3 High, multiplier = 1.21 Low, multiplier = 1.12 Accelerated, multiplier = 1.29 2306 person-months
Reliability Complexity Memory constraint Tool use Schedule Adjusted COCOMO estimate	Very low, multiplier = 0.75 Very low, multiplier = 0.75 None, multiplier = 1 Very high, multiplier = 0.72 Normal, multiplier = 1 295 person-months

Project planning

- Algorithmic cost models provide a basis for project planning as they allow alternative strategies to be compared
- Embedded spacecraft system
 - Must be reliable
 - Must minimise weight (number of chips)
 - Multipliers on reliability and computer constraints > 1
- Cost components
 - Target hardware
 - Development platform
 - Effort required

Management options



Management options costs

Option	RELY	STOR	TIME	TOOLS	LTEX	Total effort	Software cost	Hardware cost	Total cost
A	1.39	1.06	1.11	0.86	1	63	949393	100000	1049393
B	1.39	1	1	1.12	1.22	88	1313550	120000	1402025
C	1.39	1	1.11	0.86	1	60	895653	105000	1000653
D	<i>1.39</i>	<i>1.06</i>	<i>1.11</i>	<i>0.86</i>	<i>0.84</i>	<i>51</i>	<i>769008</i>	<i>100000</i>	<i>897490</i>
E	1.39	1	1	0.72	1.22	56	844425	220000	1044159
F	1.39	1	1	1.12	0.84	57	851180	120000	1002706

Option choice

- Option D (use more experienced staff) appears to be the best alternative
 - However, it has a high associated risk as experienced staff may be difficult to find
- Option C (upgrade memory) has a lower cost saving but very low risk
- Overall, the model reveals the importance of staff experience in software development

Project duration and staffing

- As well as effort estimation, managers must estimate the calendar time required to complete a project and when staff will be required
- Calendar time can be estimated using a COCOMO 2 formula
 - $TDEV = 3 \times (PM)^{(0.33+0.2*(B-1.01))}$
 - PM is the effort computation and B is the exponent computed as discussed above (B is 1 for the early prototyping model). This computation predicts the nominal schedule for the project
- The time required is independent of the number of people working on the project

Staffing requirements

- Staff required can't be computed by dividing the development time by the required schedule
- The number of people working on a project varies depending on the phase of the project
- The more people who work on the project, the more total effort is usually required
- A very rapid build-up of people often correlates with schedule slippage

Key points

- Factors affecting productivity include individual aptitude, domain experience, the development project, the project size, tool support and the working environment
- Different techniques of cost estimation should be used when estimating costs
- Software may be priced to gain a contract and the functionality adjusted to the price

Key points

- Algorithmic cost estimation is difficult because of the need to estimate attributes of the finished product
- The COCOMO model takes project, product, personnel and hardware attributes into account when predicting effort required
- Algorithmic cost models support quantitative option analysis
- The time to complete a project is not proportional to the number of people working on the project

Quality Management

- Managing the quality of the software process and products

Objectives

- To introduce the quality management process and key quality management activities
- To explain the role of standards in quality management
- To explain the concept of a software metric, predictor metrics and control metrics
- To explain how measurement may be used in assessing software quality

Topics covered

- Quality assurance and standards
- Quality planning
- Quality control
- Software measurement and metrics

Software quality management

- Concerned with ensuring that the required level of quality is achieved in a software product
- Involves defining appropriate quality standards and procedures and ensuring that these are followed
- Should aim to develop a ‘quality culture’ where quality is seen as everyone’s responsibility

What is quality?

- Quality, simplistically, means that a product should meet its specification
- This is problematical for software systems
 - Tension between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.)
 - Some quality requirements are difficult to specify in an unambiguous way
 - Software specifications are usually incomplete and often inconsistent

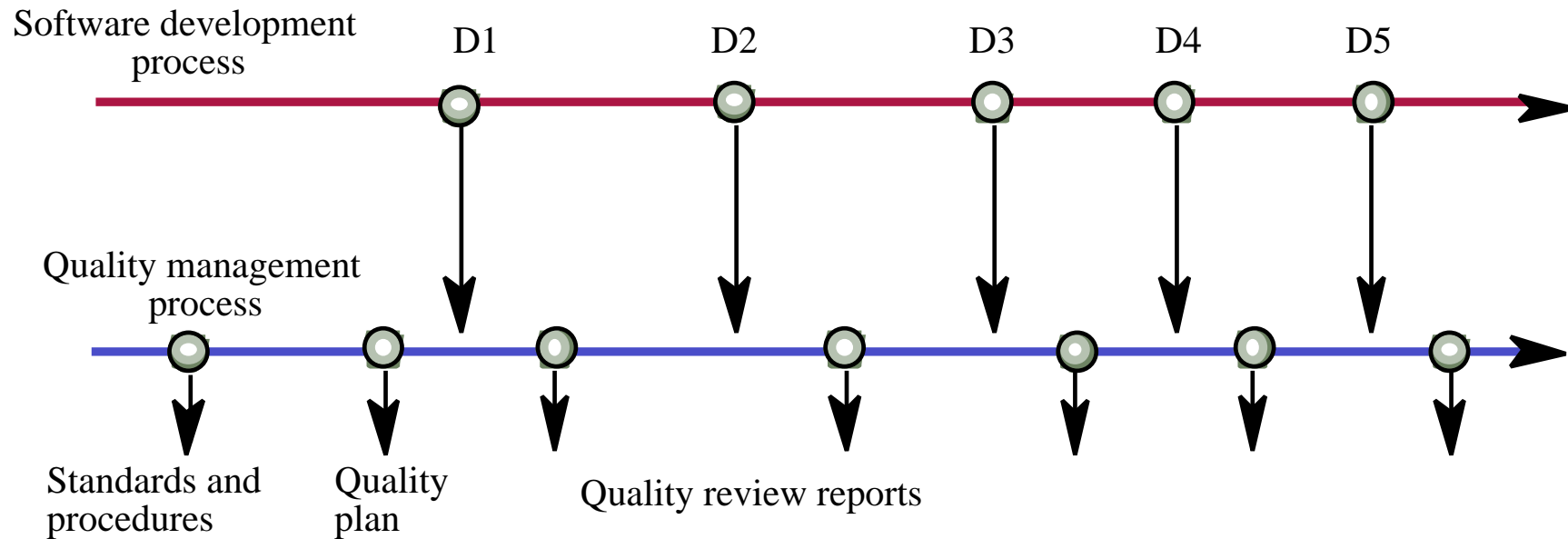
The quality compromise

- We cannot wait for specifications to improve before paying attention to quality management
- Must put procedures into place to improve quality in spite of imperfect specification
- Quality management is therefore not just concerned with reducing defects but also with other product qualities

Quality management activities

- Quality assurance
 - Establish organisational procedures and standards for quality
- Quality planning
 - Select applicable procedures and standards for a particular project and modify these as required
- Quality control
 - Ensure that procedures and standards are followed by the software development team
- Quality management should be separate from project management to ensure independence

Quality management and software development



ISO 9000

- International set of standards for quality management
- Applicable to a range of organisations from manufacturing to service industries
- ISO 9001 applicable to organisations which design, develop and maintain products
- ISO 9001 is a generic model of the quality process Must be instantiated for each organisation

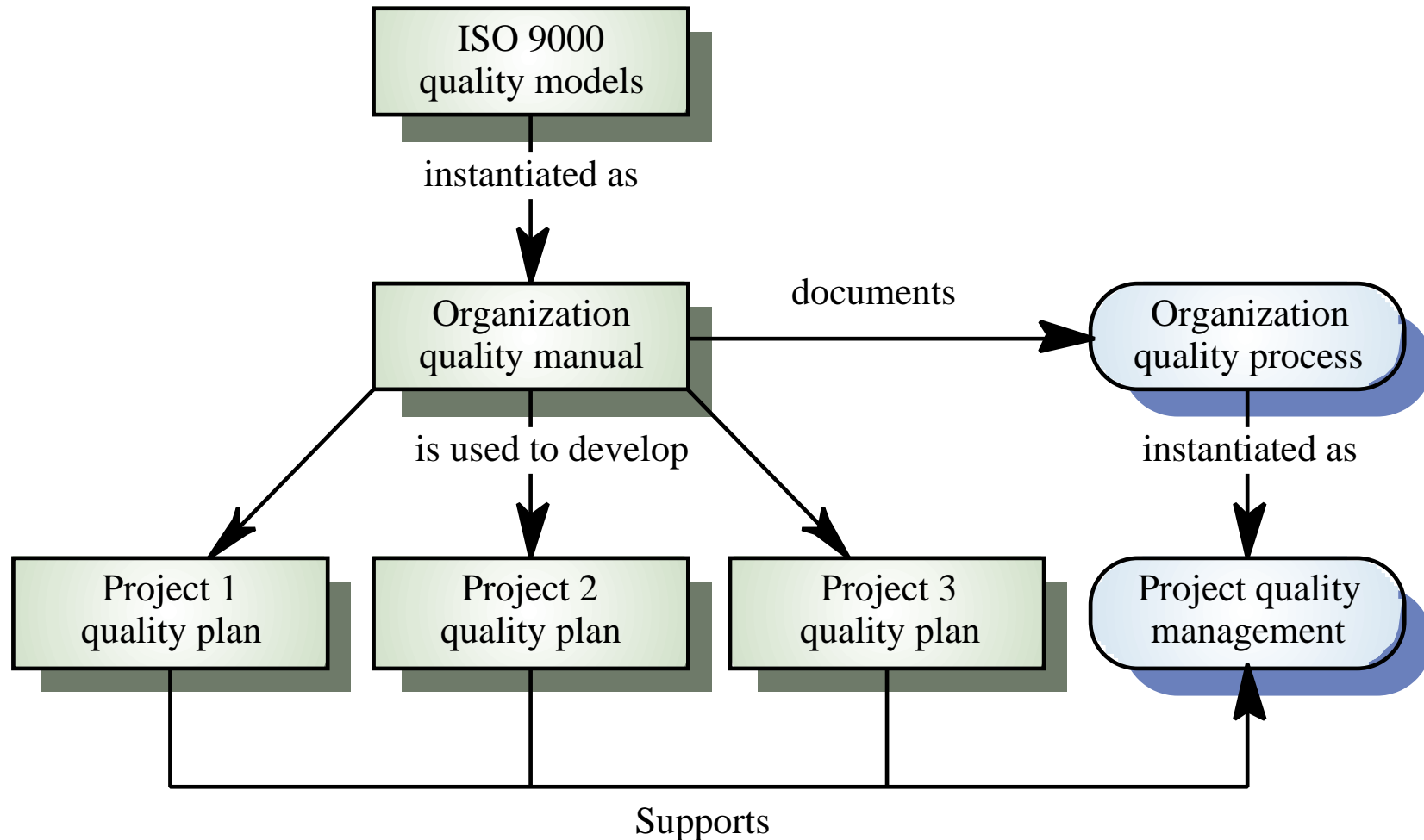
ISO 9001

Management responsibility	Quality system
Control of non-conforming products	Design control
Handling, storage, packaging and delivery	Purchasing
Purchaser-supplied products	Product identification and traceability
Process control	Inspection and testing
Inspection and test equipment	Inspection and test status
Contract review	Corrective action
Document control	Quality records
Internal quality audits	Training
Servicing	Statistical techniques

ISO 9000 certification

- Quality standards and procedures should be documented in an organisational quality manual
- External body may certify that an organisation's quality manual conforms to ISO 9000 standards
- Customers are, increasingly, demanding that suppliers are ISO 9000 certified

ISO 9000 and quality management



Quality assurance and standards

- Standards are the key to effective quality management
- They may be international, national, organizational or project standards
- Product standards define characteristics that all components should exhibit e.g. a common programming style
- Process standards define how the software process should be enacted

Importance of standards

- Encapsulation of best practice- avoids repetition of past mistakes
- Framework for quality assurance process - it involves checking standard compliance
- Provide continuity - new staff can understand the organisation by understand the standards applied

Product and process standards

Product standards	Process standards
Design review form	Design review conduct
Document naming standards	Submission of documents to CM
Procedure header format	Version release process
Ada programming style standard	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Problems with standards

- Not seen as relevant and up-to-date by software engineers
- Involve too much bureaucratic form filling
- Unsupported by software tools so tedious manual work is involved to maintain standards

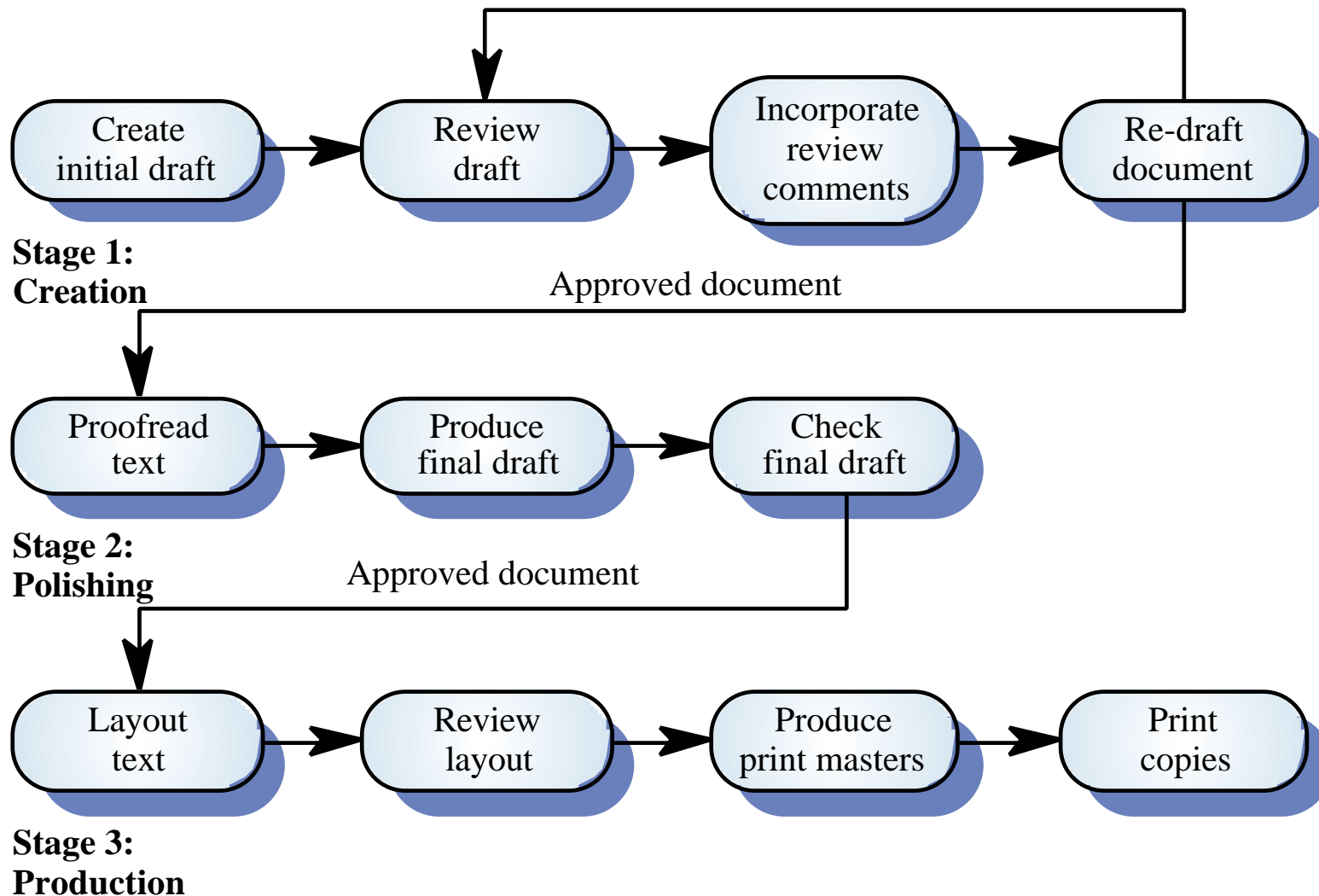
Standards development

- Involve practitioners in development. Engineers should understand the rationale underlying a standard
- Review standards and their usage regularly. Standards can quickly become outdated and this reduces their credibility amongst practitioners
- Detailed standards should have associated tool support. Excessive clerical work is the most significant complaint against standards

Documentation standards

- Particularly important - documents are the tangible manifestation of the software
- Documentation process standards
 - How documents should be developed, validated and maintained
- Document standards
 - Concerned with document contents, structure, and appearance
- Document interchange standards
 - How documents are stored and interchanged between different documentation systems

Documentation process



Document standards

- Document identification standards
 - How documents are uniquely identified
- Document structure standards
 - Standard structure for project documents
- Document presentation standards
 - Define fonts and styles, use of logos, etc.
- Document update standards
 - Define how changes from previous versions are reflected in a document

Document interchange standards

- Documents are produced using different systems and on different computers
- Interchange standards allow electronic documents to be exchanged, mailed, etc.
- Need for archiving. The lifetime of word processing systems may be much less than the lifetime of the software being documented
- XML is an emerging standard for document interchange which will be widely supported in future

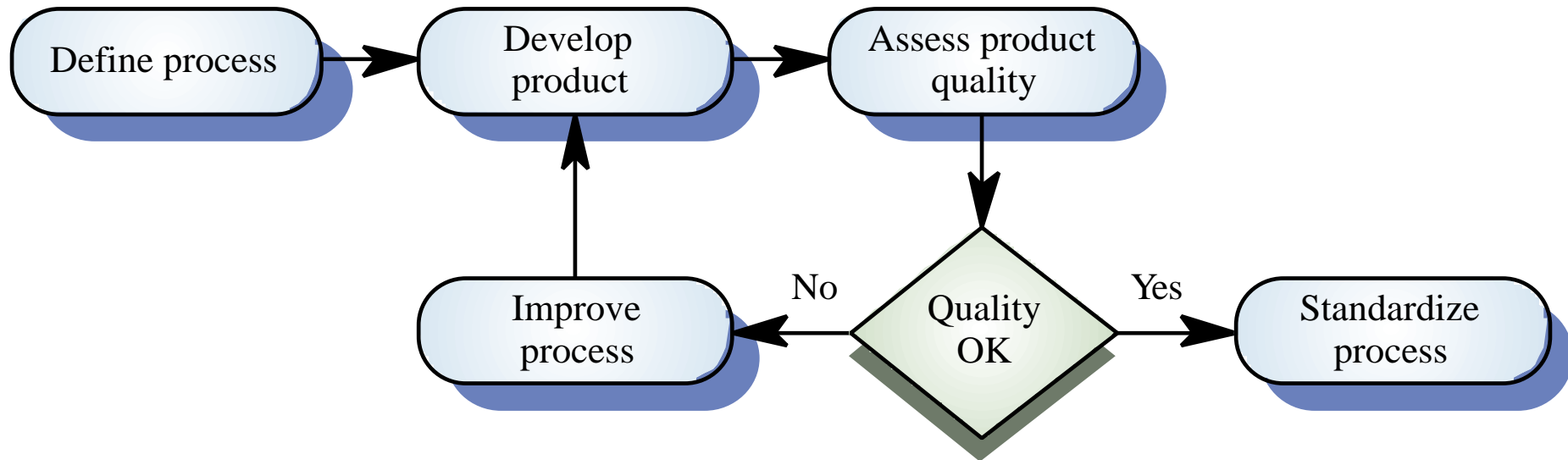
Process and product quality

- The quality of a developed product is influenced by the quality of the production process
- Particularly important in software development as some product quality attributes are hard to assess
- However, there is a very complex and poorly understood between software processes and product quality

Process-based quality

- Straightforward link between process and product in manufactured goods
- More complex for software because:
 - The application of individual skills and experience is particularly important in software development
 - External factors such as the novelty of an application or the need for an accelerated development schedule may impair product quality
- Care must be taken not to impose inappropriate process standards

Process-based quality



Practical process quality

- Define process standards such as how reviews should be conducted, configuration management, etc.
- Monitor the development process to ensure that standards are being followed
- Report on the process to project management and software procurer

Quality planning

- A quality plan sets out the desired product qualities and how these are assessed and define the most significant quality attributes
- It should define the quality assessment process
- It should set out which organisational standards should be applied and, if necessary, define new standards

Quality plan structure

- Product introduction
- Product plans
- Process descriptions
- Quality goals
- Risks and risk management
- Quality plans should be short, succinct documents
 - If they are too long, no-one will read them

Software quality attributes

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

Quality control

- Checking the software development process to ensure that procedures and standards are being followed
- Two approaches to quality control
 - Quality reviews
 - Automated software assessment and software measurement

Quality reviews

- The principal method of validating the quality of a process or of a product
- Group examined part or all of a process or system and its documentation to find potential problems
- There are different types of review with different objectives
 - Inspections for defect removal (product)
 - Reviews for progress assessment(product and process)
 - Quality reviews (product and standards)

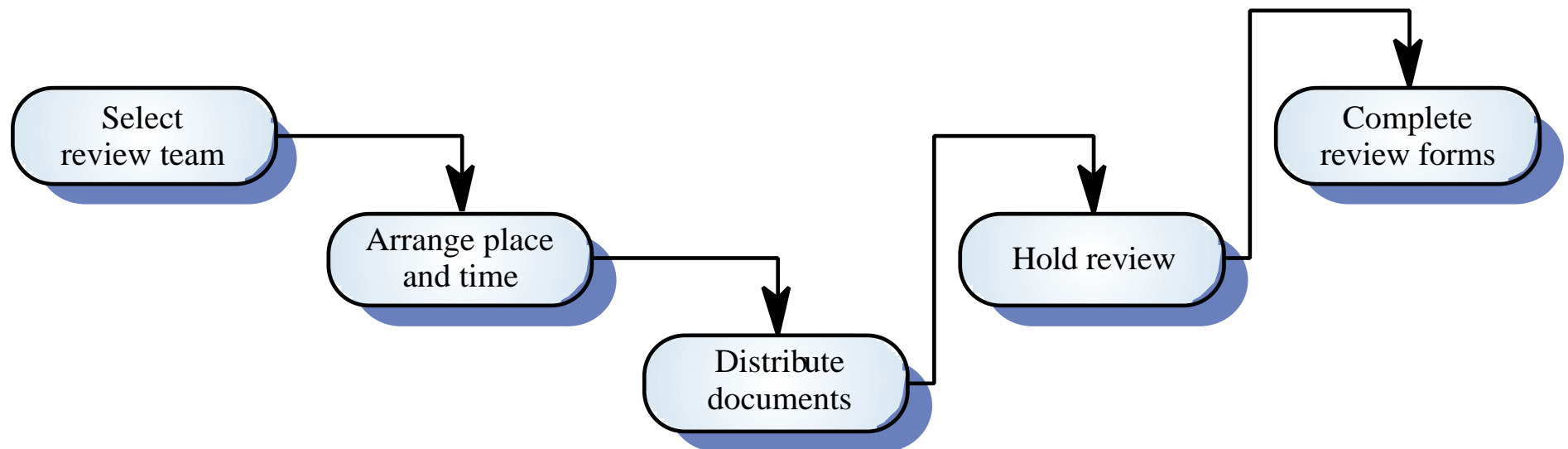
Types of review

Review type	Principal purpose
Design or program inspections	To detect detailed errors in the design or code and to check whether standards have been followed. The review should be driven by a checklist of possible errors.
Progress reviews	To provide information for management about the overall progress of the project. This is both a process and a product review and is concerned with costs, plans and schedules.
Quality reviews	To carry out a technical analysis of product components or documentation to find faults or mismatches between the specification and the design, code or documentation. It may also be concerned with broader quality issues such as adherence to standards and other quality attributes.

Quality reviews

- A group of people carefully examine part or all of a software system and its associated documentation.
- Code, designs, specifications, test plans, standards, etc. can all be reviewed.
- Software or documents may be 'signed off' at a review which signifies that progress to the next development stage has been approved by management.

The review process



Review functions

- Quality function - they are part of the general quality management process
- Project management function - they provide information for project managers
- Training and communication function - product knowledge is passed between development team members

Quality reviews

- Objective is the discovery of system defects and inconsistencies
- Any documents produced in the process may be reviewed
- Review teams should be relatively small and reviews should be fairly short
- Review should be recorded and records maintained

Review results

- Comments made during the review should be classified.
 - No action. No change to the software or documentation is required.
 - Refer for repair. Designer or programmer should correct an identified fault.
 - Reconsider overall design. The problem identified in the review impacts other parts of the design. Some overall judgement must be made about the most cost-effective way of solving the problem.
- Requirements and specification errors may have to be referred to the client.

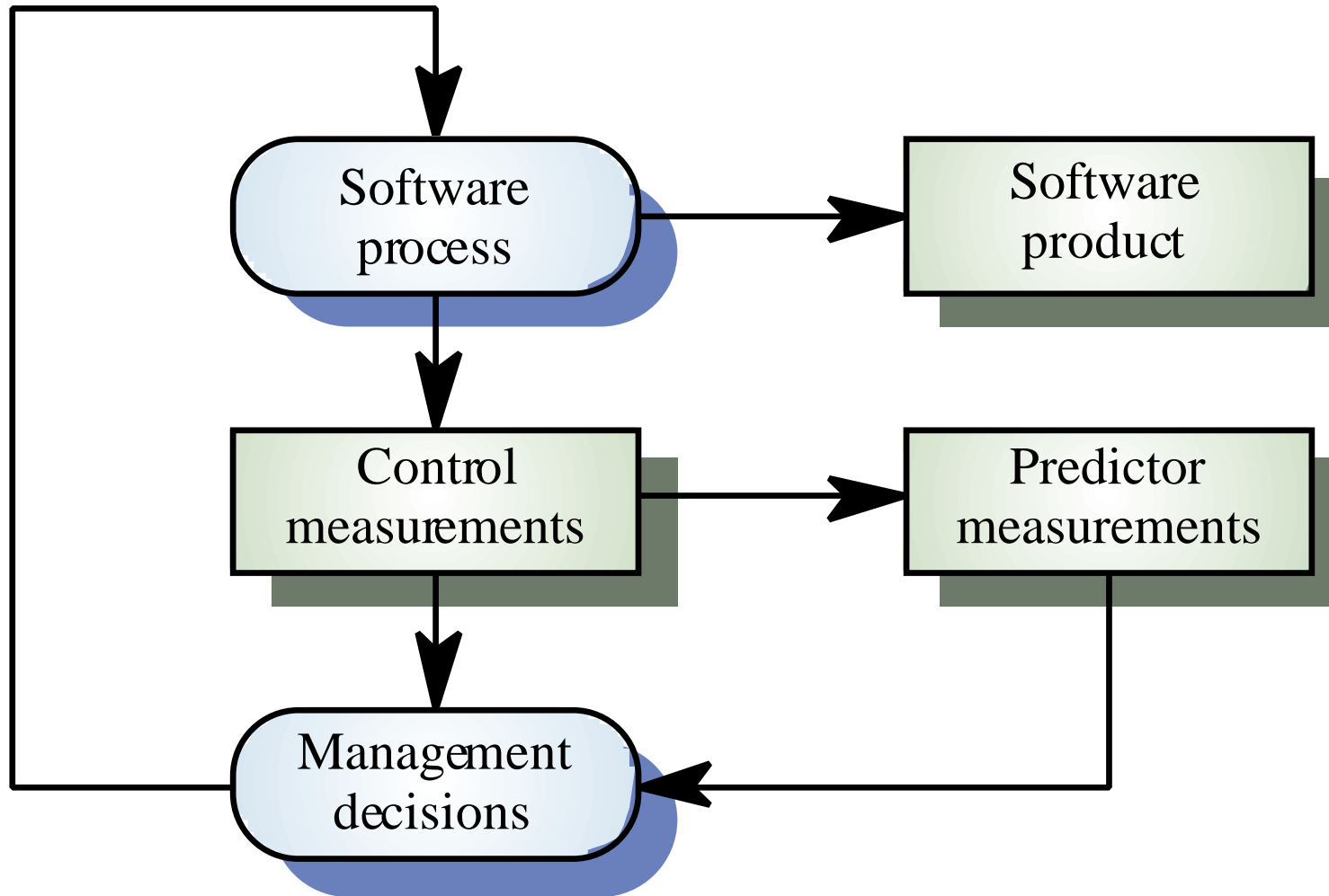
Software measurement and metrics

- Software measurement is concerned with deriving a numeric value for an attribute of a software product or process
- This allows for objective comparisons between techniques and processes
- Although some companies have introduced measurement programmes, the systematic use of measurement is still uncommon
- There are few standards in this area

Software metric

- Any type of measurement which relates to a software system, process or related documentation
 - Lines of code in a program, the Fog index, number of person-days required to develop a component
- Allow the software and the software process to be quantified
- Measures of the software process or product
- May be used to predict product attributes or to control the software process

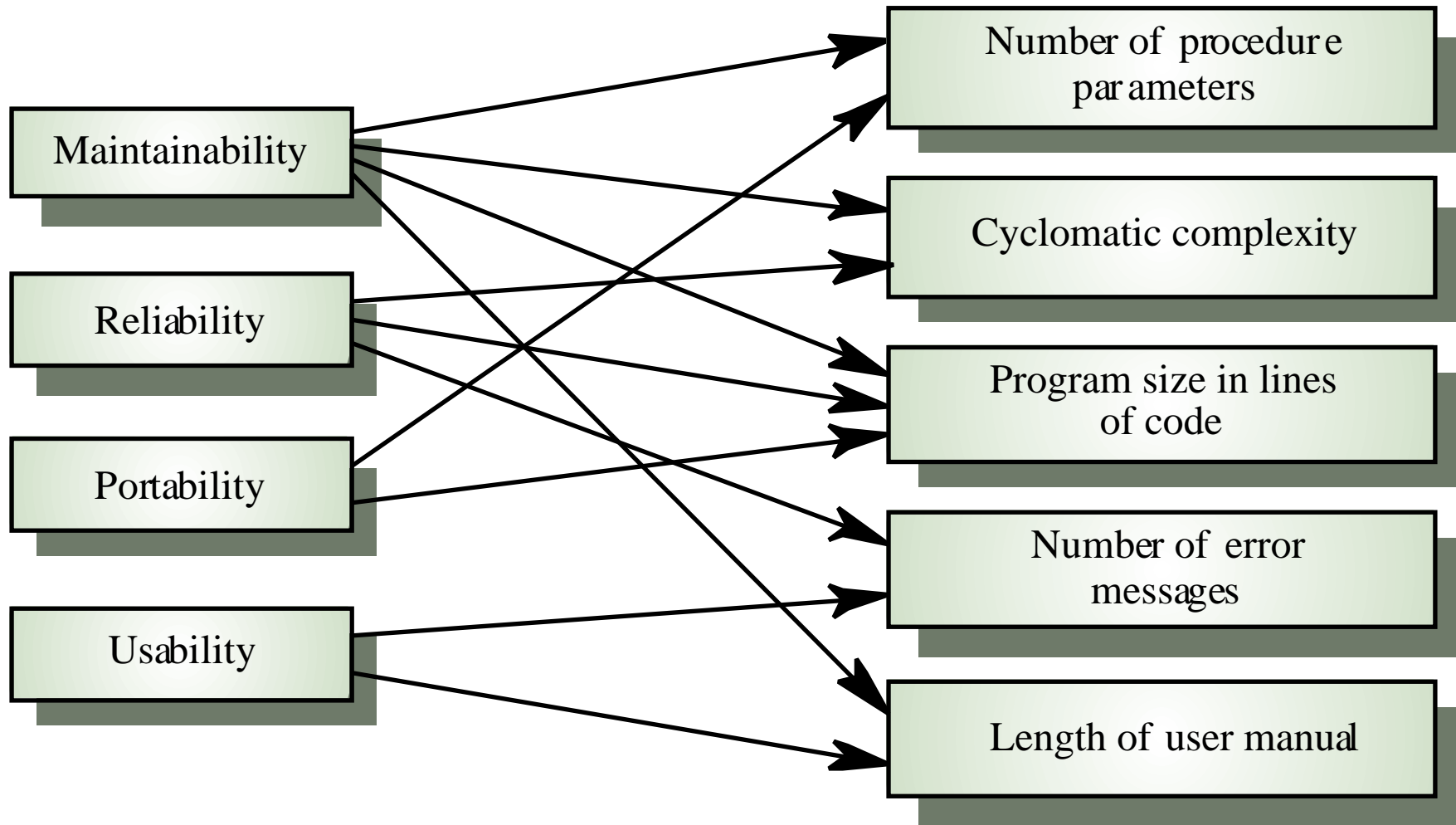
Predictor and control metrics



Metrics assumptions

- A software property can be measured
- The relationship exists between what we can measure and what we want to know
- This relationship has been formalized and validated
- It may be difficult to relate what can be measured to desirable quality attributes

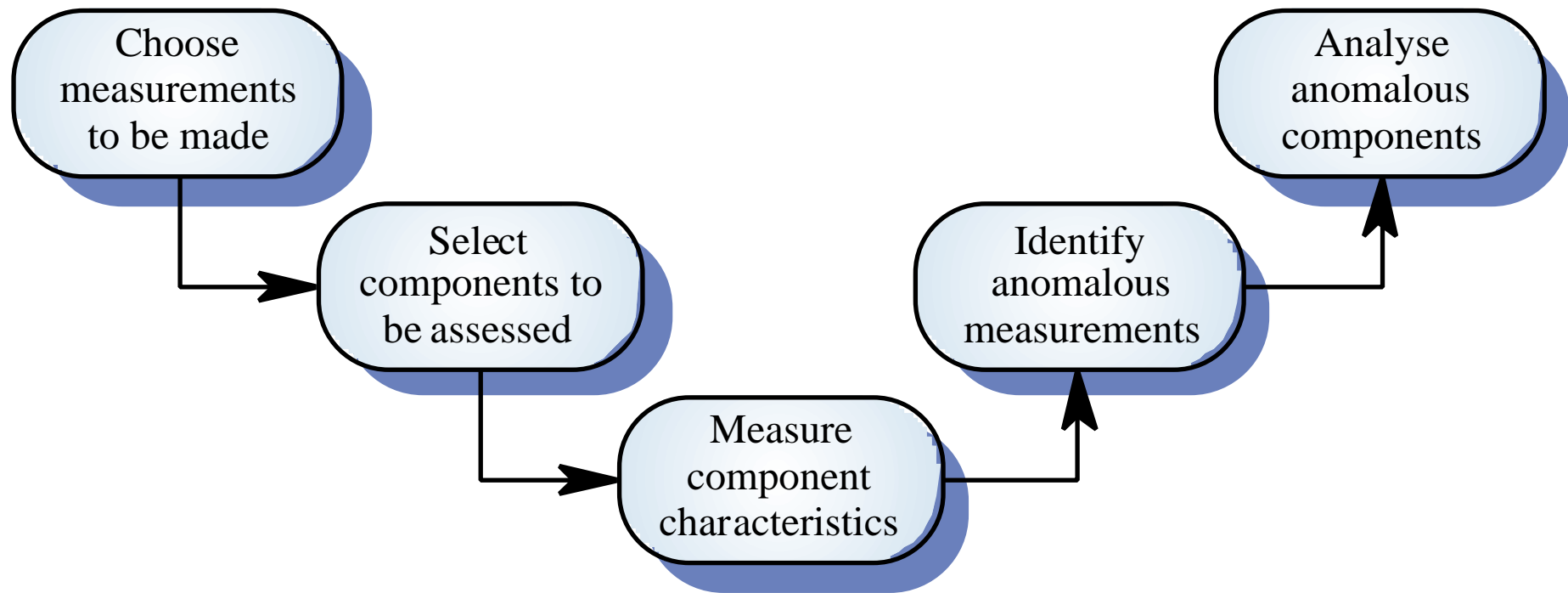
Internal and external attributes



The measurement process

- A software measurement process may be part of a quality control process
- Data collected during this process should be maintained as an organisational resource
- Once a measurement database has been established, comparisons across projects become possible

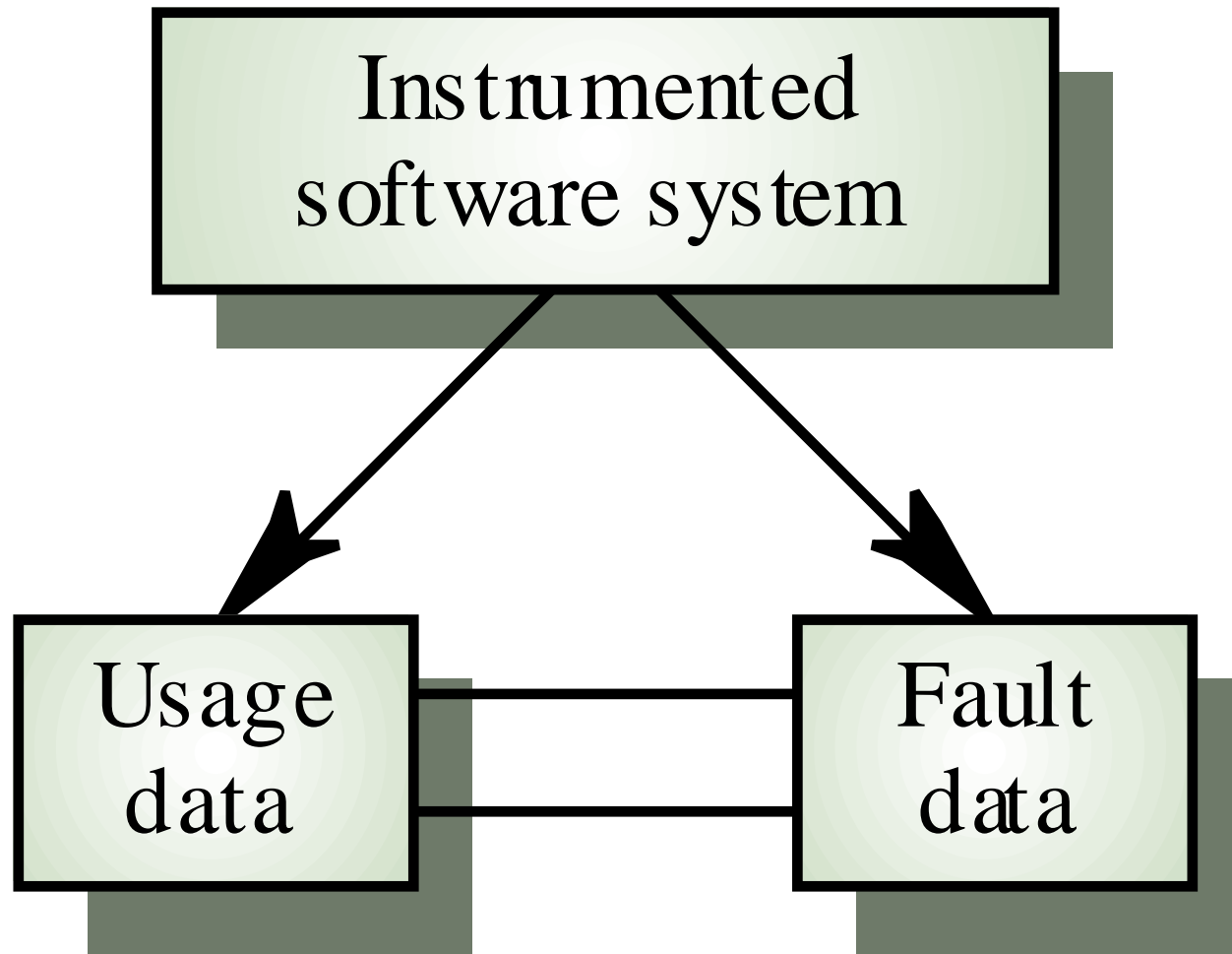
Product measurement process



Data collection

- A metrics programme should be based on a set of product and process data
- Data should be collected immediately (not in retrospect) and, if possible, automatically
- Three types of automatic data collection
 - Static product analysis
 - Dynamic product analysis
 - Process data collation

Automated data collection



Data accuracy

- Don't collect unnecessary data
 - The questions to be answered should be decided in advance and the required data identified
- Tell people why the data is being collected
 - It should not be part of personnel evaluation
- Don't rely on memory
 - Collect data when it is generated not after a project has finished

Product metrics

- A quality metric should be a predictor of product quality
- Classes of product metric
 - Dynamic metrics which are collected by measurements made of a program in execution
 - Static metrics which are collected by measurements made of the system representations
 - Dynamic metrics help assess efficiency and reliability; static metrics help assess complexity, understandability and maintainability

Dynamic and static metrics

- Dynamic metrics are closely related to software quality attributes
 - It is relatively easy to measure the response time of a system (performance attribute) or the number of failures (reliability attribute)
- Static metrics have an indirect relationship with quality attributes
 - You need to try and derive a relationship between these metrics and properties such as complexity, understandability and maintainability

Software product metrics

Software metric	Description
Fan in/Fan-out	Fan-in is a measure of the number of functions that call some other function (say X). Fan-out is the number of functions which are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a program's components, the more complex and error-prone that component is likely to be.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. The computation of cyclomatic complexity is covered in Chapter 20.
Length of identifiers	This is a measure of the average length of distinct identifiers in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if statements are hard to understand and are potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value for the Fog index, the more difficult the document may be to understand.

Object-oriented metrics

Object-oriented metric	Description
Depth of inheritance tree	This represents the number of discrete levels in the inheritance tree where sub-classes inherit attributes and operations (methods) from super-classes. The deeper the inheritance tree, the more complex the design as, potentially, many different object classes have to be understood to understand the object classes at the leaves of the tree.
Method fan-in/fan-out	This is directly related to fan-in and fan-out as described above and means essentially the same thing. However, it may be appropriate to make a distinction between calls from other methods within the object and calls from external methods.
Weighted methods per class	This is the number of methods included in a class weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1 and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be more difficult to understand. They may not be logically cohesive so cannot be reused effectively as super-classes in an inheritance tree.
Number of overriding operations	These are the number of operations in a super-class which are over-ridden in a sub-class. A high value for this metric indicates that the super-class used may not be an appropriate parent for the sub-class.

Measurement analysis

- It is not always obvious what data means
 - Analysing collected data is very difficult
- Professional statisticians should be consulted if available
- Data analysis must take local circumstances into account

Measurement surprises

- Reducing the number of faults in a program leads to an increased number of help desk calls
 - The program is now thought of as more reliable and so has a wider more diverse market. The percentage of users who call the help desk may have decreased but the total may increase
 - A more reliable system is used in a different way from a system where users work around the faults. This leads to more help desk calls

Key points

- Software quality management is concerned with ensuring that software meets its required standards
- Quality assurance procedures should be documented in an organisational quality manual
- Software standards are an encapsulation of best practice
- Reviews are the most widely used approach for assessing software quality

Key points

- Software measurement gathers information about both the software process and the software product
- Product quality metrics should be used to identify potentially problematical components
- There are no standardised and universally applicable software metrics

Process Improvement

- Understanding, Modelling and Improving the Software Process

Objectives

- To explain the principles of software process improvement
- To explain how software process factors influence software quality and productivity
- To introduce the SEI Capability Maturity Model and to explain why it is influential. To discuss the applicability of that model
- To explain why CMM-based improvement is not universally applicable

Topics covered

- Process and product quality
- Process analysis and modelling
- Process measurement
- The SEI process maturity model
- Process classification

Process improvement

- Understanding existing processes
- Introducing process changes to achieve organisational objectives which are usually focused on quality improvement, cost reduction and schedule acceleration
- Most process improvement work so far has focused on defect reduction. This reflects the increasing attention paid by industry to quality
- However, other process attributes can be the focus of improvement

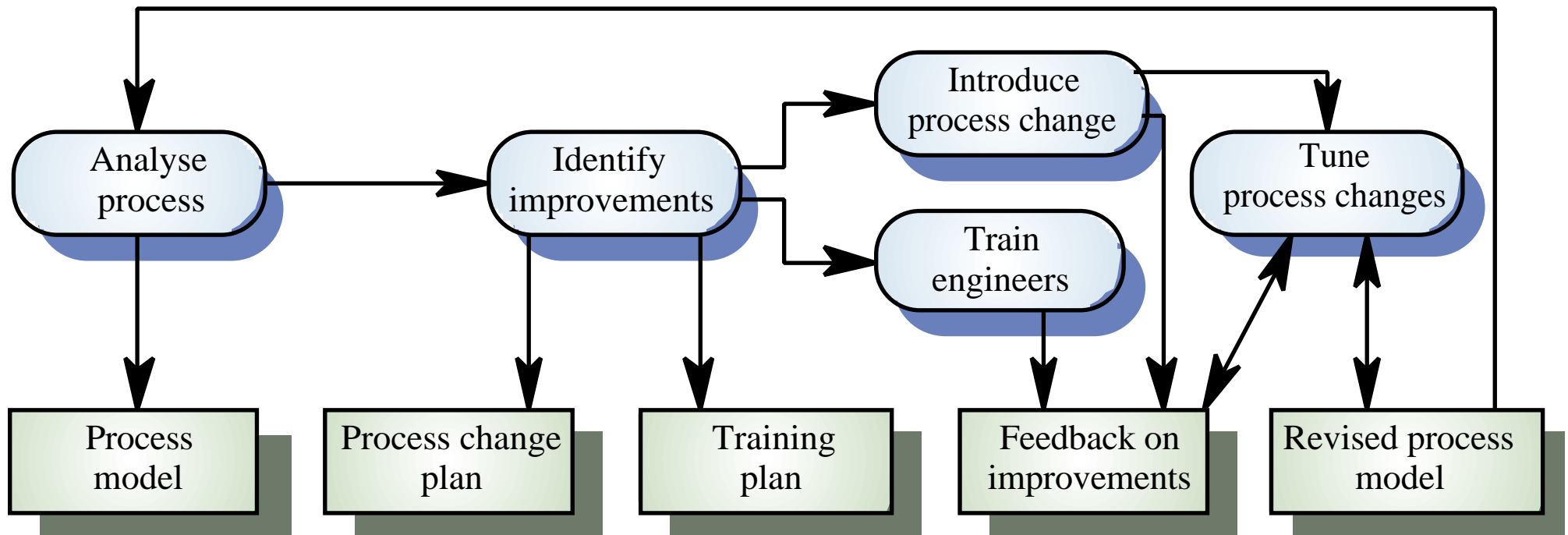
Process attributes

Process characteristic	Description
Understandability	To what extent is the process explicitly defined and how easy is it to understand the process definition?
Visibility	Do the process activities culminate in clear results so that the progress of the process is externally visible?
Supportability	To what extent can the process activities be supported by CASE tools?
Acceptability	Is the defined process acceptable to and usable by the engineers responsible for producing the software product?
Reliability	Is the process designed in such a way that process errors are avoided or trapped before they result in product errors?
Robustness	Can the process continue in spite of unexpected problems?
Maintainability	Can the process evolve to reflect changing organisational requirements or identified process improvements?
Rapidity	How fast can the process of delivering a system from a given specification be completed?

Process improvement stages

- Process analysis
 - Model and analyse (quantitatively if possible) existing processes
- Improvement identification
 - Identify quality, cost or schedule bottlenecks
- Process change introduction
 - Modify the process to remove identified bottlenecks
- Process change training
 - Train staff involved in new process proposals
- Change tuning
 - Evolve and improve process improvements

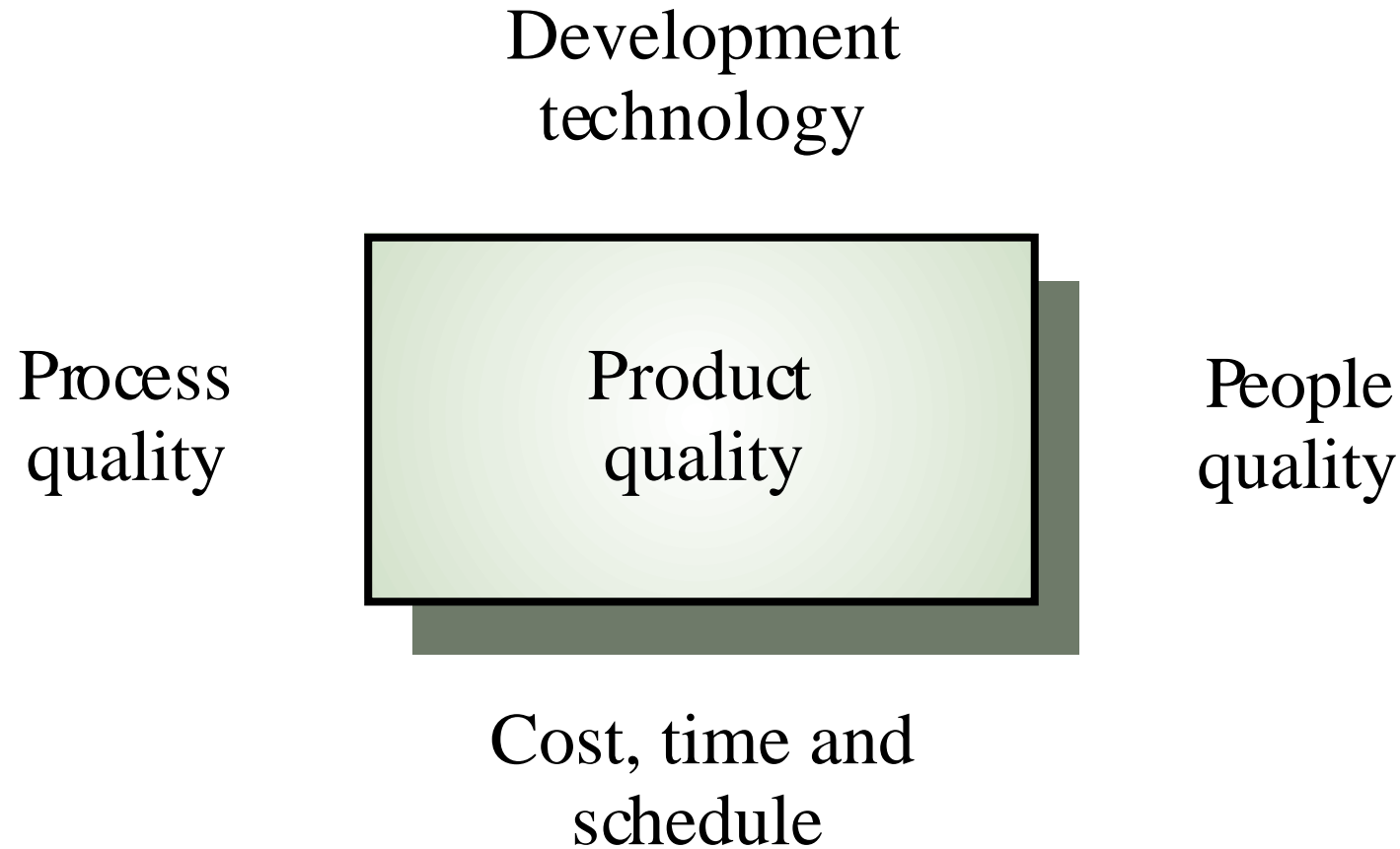
The process improvement process



Process and product quality

- Process quality and product quality are closely related
- A good process is usually required to produce a good product
- For manufactured goods, process is the principal quality determinant
- For design-based activity, other factors are also involved especially the capabilities of the designers

Principal product quality factors



Quality factors

- For large projects with ‘average’ capabilities, the development process determines product quality
- For small projects, the capabilities of the developers is the main determinant
- The development technology is particularly significant for small projects
- In all cases, if an unrealistic schedule is imposed then product quality will suffer

Process analysis and modelling

- Process analysis
 - The study of existing processes to understand the relationships between parts of the process and to compare them with other processes
- Process modelling
 - The documentation of a process which records the tasks, the roles and the entities used
 - Process models may be presented from different perspectives

Process analysis and modelling

- Study an existing process to understand its activities
- Produce an abstract model of the process. You should normally represent this graphically. Several different views (e.g. activities, deliverables, etc.) may be required
- Analyse the model to discover process problems. Involves discussing activities with stakeholders

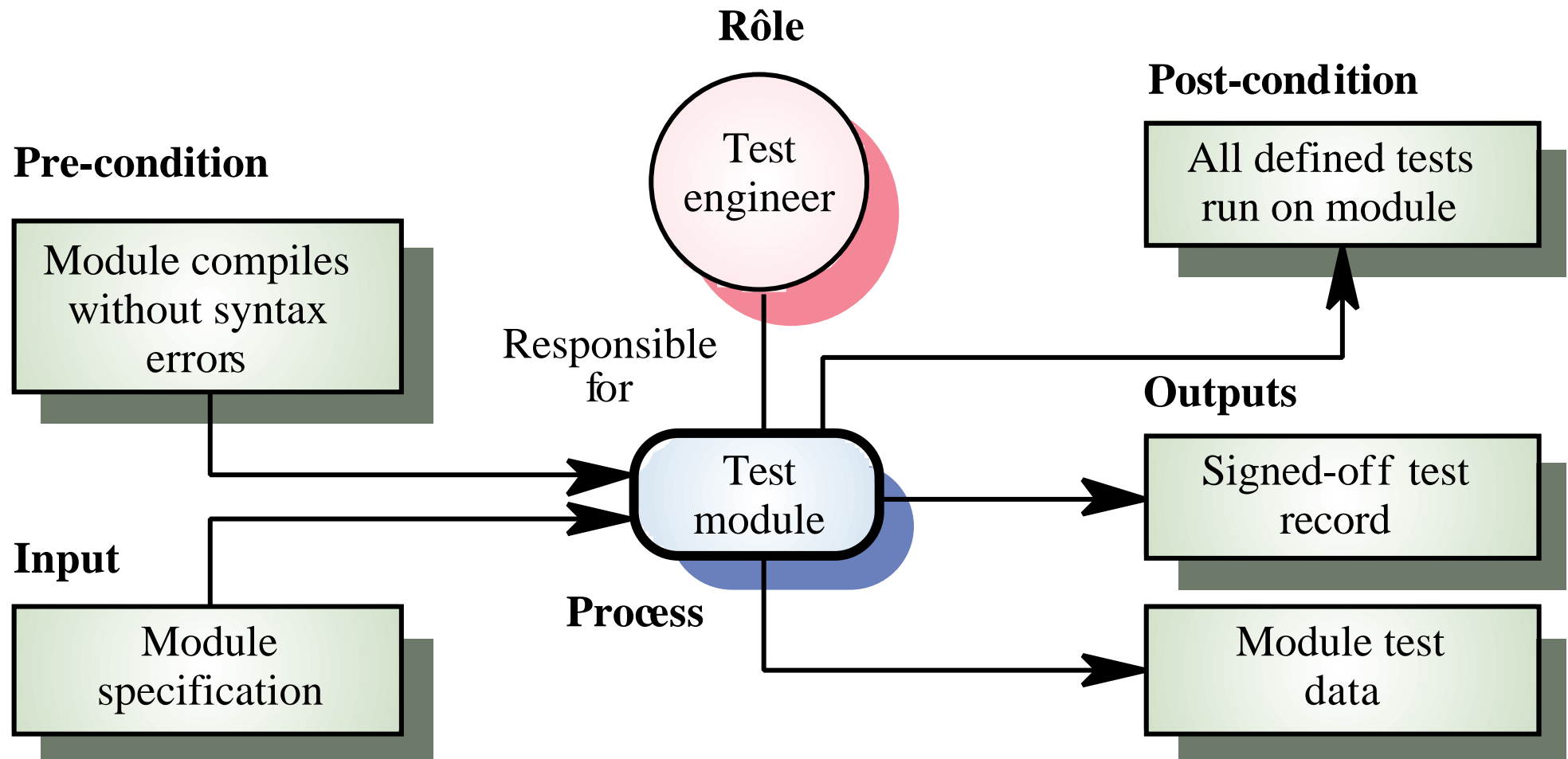
Process analysis techniques

- Published process models and process standards
 - It is always best to start process analysis with an existing model. People then may extend and change this.
- Questionnaires and interviews
 - Must be carefully designed. Participants may tell you what they think you want to hear
- Ethnographic analysis
 - Involves assimilating process knowledge by observation

Process model element	Description
Activity (represented by a round-edged rectangle with no drop shadow)	An activity has a clearly defined objective, entry and exit conditions. Examples of activities are preparing a set of test data to test a module, coding a function or a module, proof-reading a document, etc. Generally, an activity is atomic i.e. it is the responsibility of one person or group. It is not decomposed into sub-activities.
Process (represented by a round-edged rectangle with drop shadow)	A process is a set of activities which have some coherence and whose objective is generally agreed within an organisation. Examples of processes are requirements analysis, architectural design, test planning, etc.
Deliverable (represented by a rectangle with drop shadow)	A deliverable is a tangible output of an activity which is predicted in a project plan.
Condition (represented by a parallelogram)	A condition is either a pre-condition which must hold before a process or activity can start or a post-condition which holds after a process or activity has finished.
Role (represented by a circle with drop shadow)	A role is a bounded area of responsibility. Examples of roles might be configuration manager, test engineer, software designer, etc. One person may have several different roles and a single role may be associated with several different people.
Exception (not shown in examples here but may be represented as a double edged box)	An exception is a description of how to modify the process if some anticipated or unanticipated event occurs. Exceptions are often undefined and it is left to the ingenuity of the project managers and engineers to handle the exception.
Communication (represented by an arrow)	An interchange of information between people or between people and supporting computer systems. Communications may be informal or formal. Formal communications might be the approval of a deliverable by a project manager; informal communications might be the interchange of electronic mail to resolve ambiguities in a document.

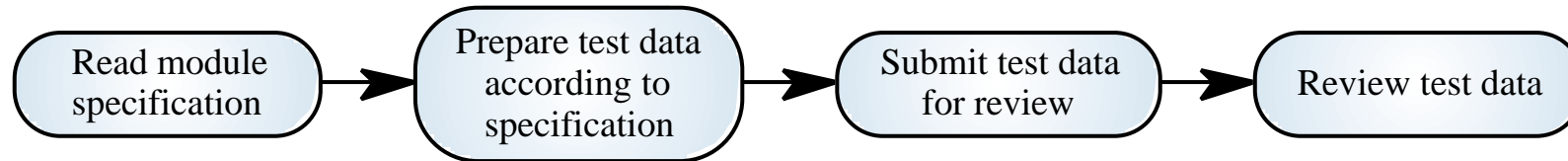
Elements of a process model

The module testing activity

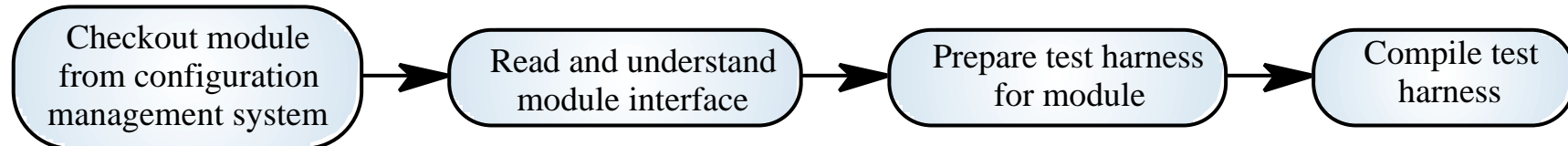


Activities in module testing

TEST DATA PREPARATION



MODULE TEST HARNESS PREPARATION



TEST EXECUTION



TEST REPORTING



Process exceptions

- Software processes are complex and process models cannot effectively represent how to handle exceptions
 - Several key people becoming ill just before a critical review
 - A complete failure of a communication processor so that no e-mail is available for several days
 - Organisational reorganisation
 - A need to respond to an unanticipated request for new proposals
- Under these circumstances, the model is suspended and managers use their initiative to deal with the exception

Process measurement

- Wherever possible, quantitative process data should be collected
 - However, where organisations do not have clearly defined process standards this is very difficult as you don't know what to measure. A process may have to be defined before any measurement is possible
- Process measurements should be used to assess process improvements
 - But this does not mean that measurements should drive the improvements. The improvement driver should be the organizational objectives

Classes of process measurement

- Time taken for process activities to be completed
 - E.g. Calendar time or effort to complete an activity or process
- Resources required for processes or activities
 - E.g. Total effort in person-days
- Number of occurrences of a particular event
 - E.g. Number of defects discovered

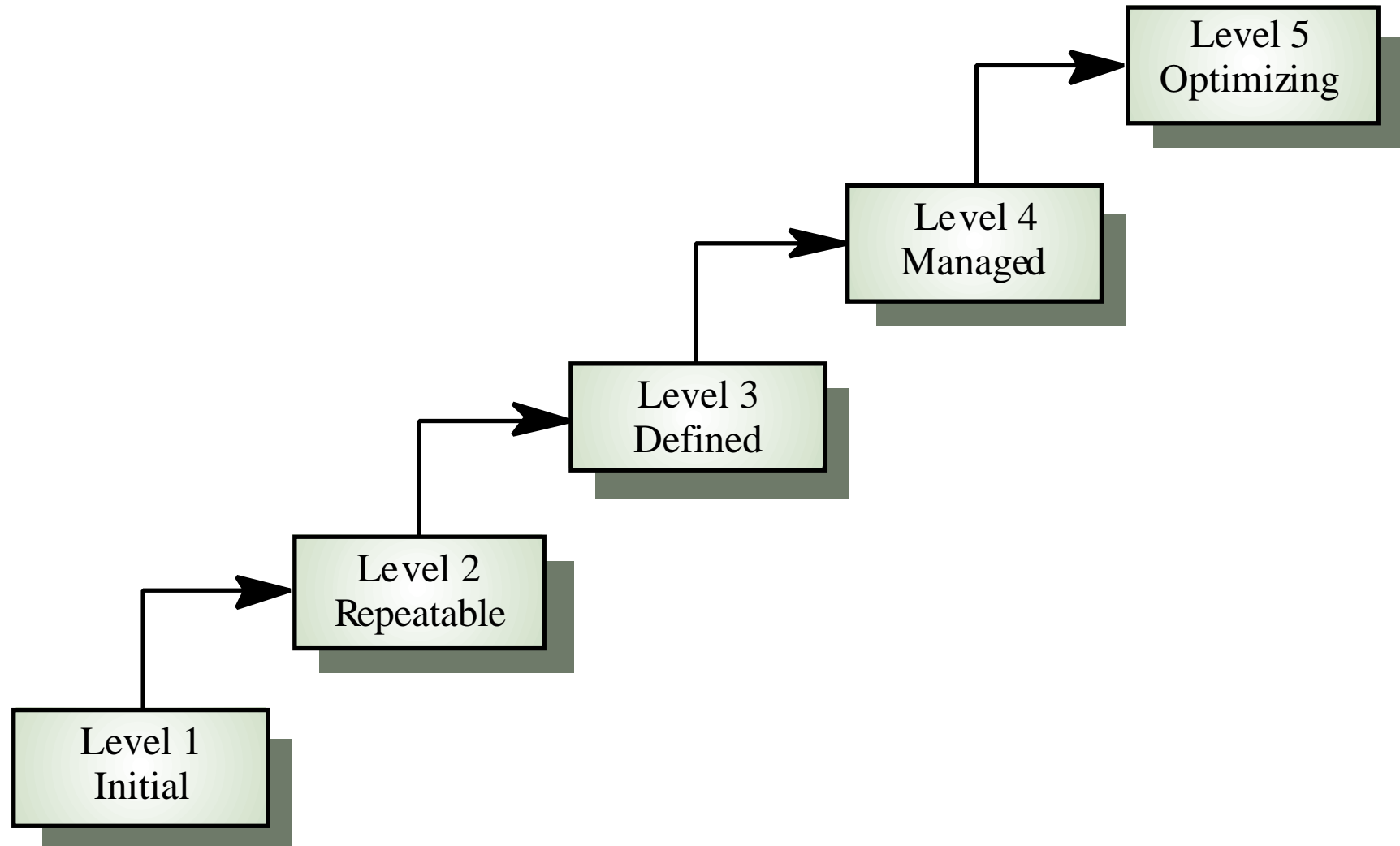
Goal-Question-Metric Paradigm

- Goals
 - What is the organisation trying to achieve? The objective of process improvement is to satisfy these goals
- Questions
 - Questions about areas of uncertainty related to the goals. You need process knowledge to derive these
- Metrics
 - Measurements to be collected to answer the questions

The Software Engineering Institute

- US Defense Dept. funded institute associated with Carnegie Mellon
- Mission is to promote software technology transfer particularly to defense contractors
- Maturity model proposed in mid-1980s, refined in early 1990s.
- Work has been very influential in process improvement

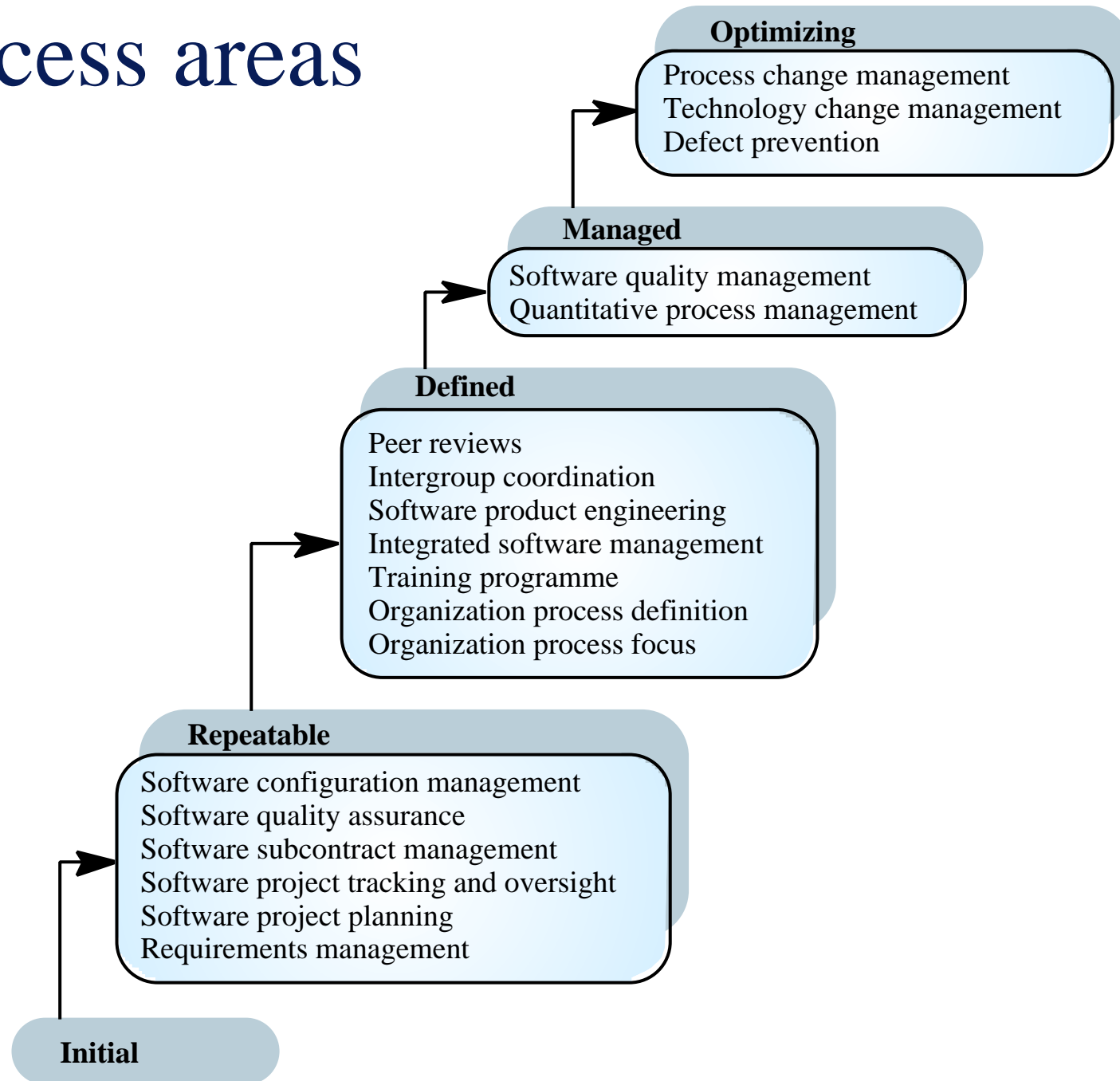
The SEI process maturity model



Maturity model levels

- Initial
 - Essentially uncontrolled
- Repeatable
 - Product management procedures defined and used
- Defined
 - Process management procedures and strategies defined and used
- Managed
 - Quality management strategies defined and used
- Optimising
 - Process improvement strategies defined and used

Key process areas



SEI model problems

- It focuses on project management rather than product development.
- It ignores the use of technologies such as rapid prototyping.
- It does not incorporate risk analysis as a key process area
- It does not define its domain of applicability

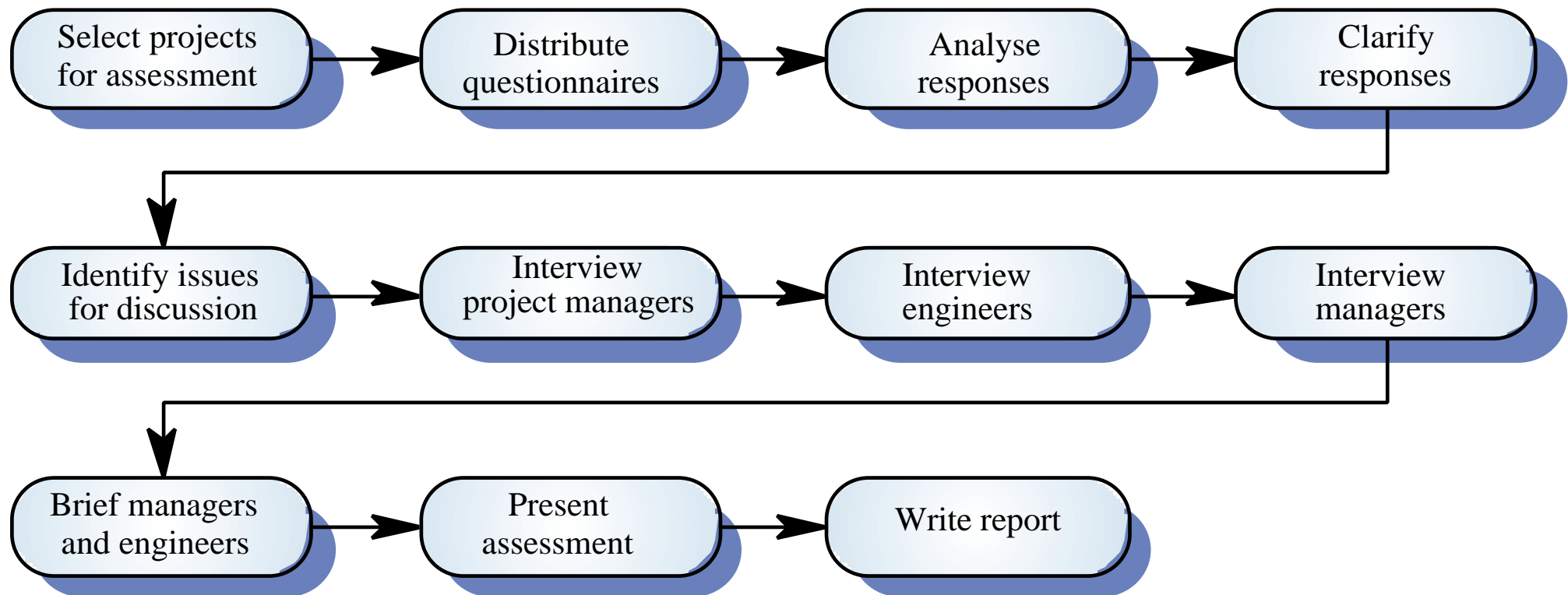
The CMM and ISO 9000

- There is a clear correlation between the key processes in the CMM and the quality management processes in ISO 9000
- The CMM is more detailed and prescriptive and includes a framework for improvement
- Organisations rated as level 2 in the CMM are likely to be ISO 9000 compliant

Capability assessment

- An important role of the SEI is to use the CMM to assess the capabilities of contractors bidding for US government defence contracts
- The model is intended to represent organisational capability not the practices used in particular projects
- Within the same organisation, there are often wide variations in processes used
- Capability assessment is questionnaire-based

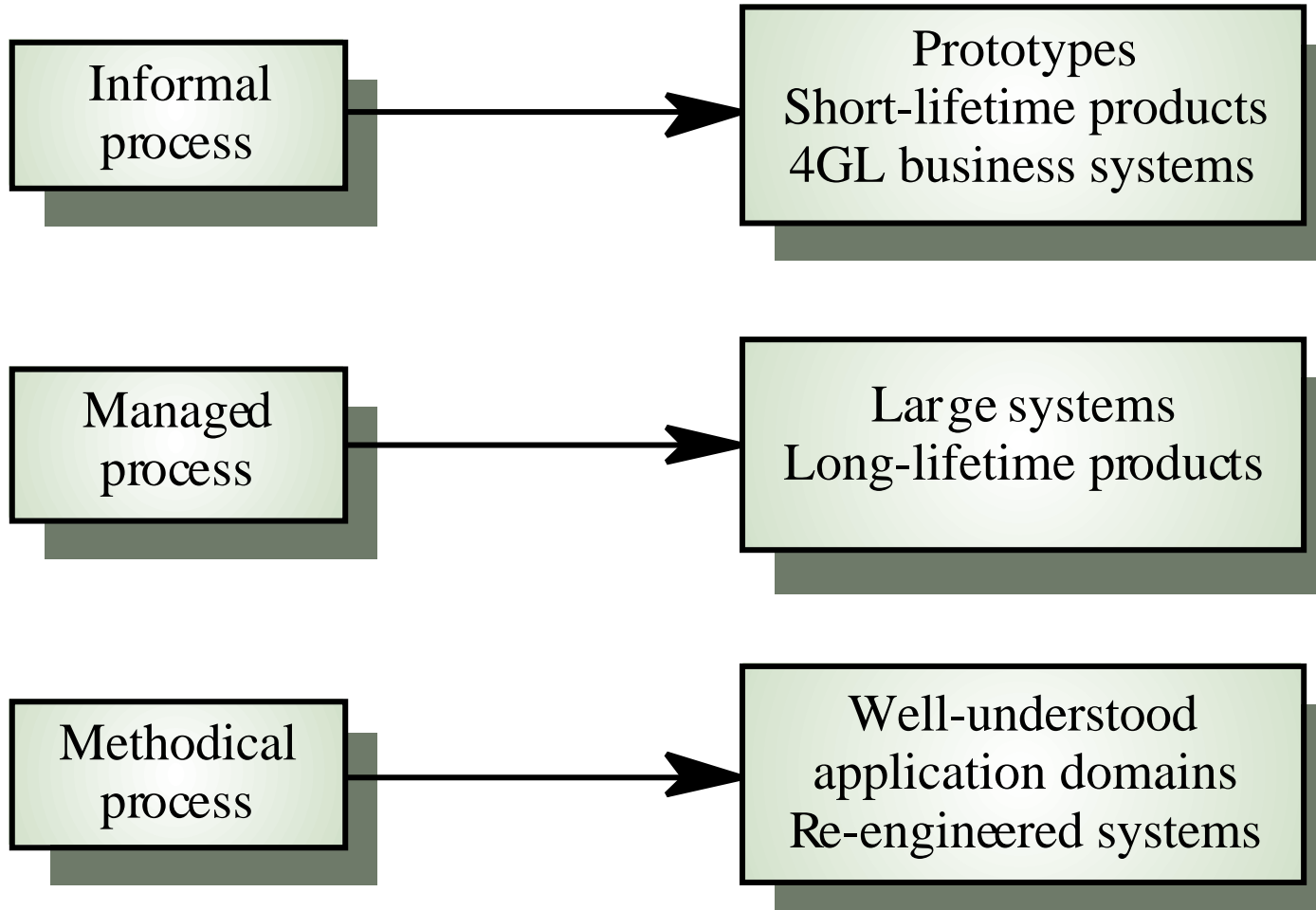
The capability assessment process



Process classification

- Informal
 - No detailed process model. Development team chose their own way of working
- Managed
 - Defined process model which drives the development process
- Methodical
 - Processes supported by some development method such as HOOD
- Supported
 - Processes supported by automated CASE tools

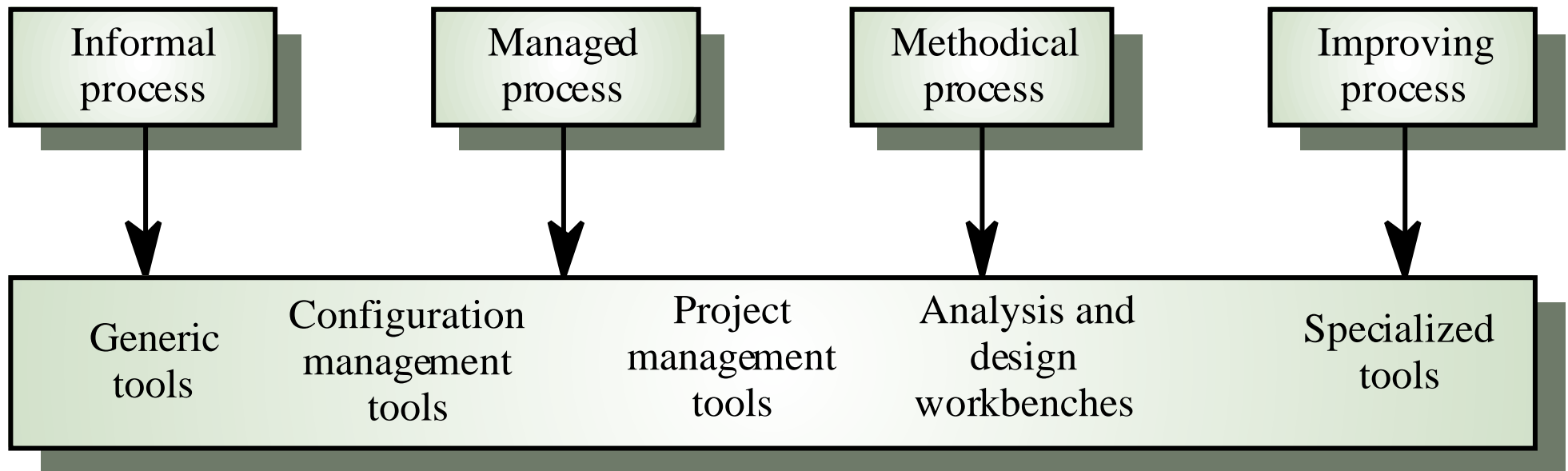
Process applicability



Process choice

- Process used should depend on type of product which is being developed
 - For large systems, management is usually the principal problem so you need a strictly managed process. For smaller systems, more informality is possible.
- There is no uniformly applicable process which should be standardised within an organisation
 - High costs may be incurred if you force an inappropriate process on a development team

Process tool support



Key points

- Process improvement involves process analysis, standardisation, measurement and change
- Process models include descriptions of tasks, activities, roles, exceptions, communications, deliverables and other processes
- Measurement should be used to answer specific questions about the software process used
- The three types of process metrics which can be collected are time metrics, resource utilisation metrics and event metrics

Key points

- The SEI model classifies software processes as initial, repeatable, defined, managed and optimising. It identifies key processes which should be used at each of these levels
- The SEI model is appropriate for large systems developed by large teams of engineers. It cannot be applied without modification in other situations
- Processes can be classified as informal, managed, methodical and improving. This classification can be used to identify process tool support

Legacy Systems

- Older software systems that remain vital to an organisation

Objectives

- To explain what is meant by a legacy system and why these systems are important
- To introduce common legacy system structures
- To briefly describe function-oriented design
- To explain how the value of legacy systems can be assessed

Topics covered

- Legacy system structures
- Legacy system design
- Legacy system assessment

Legacy systems

- Software systems that are developed specially for an organisation have a long lifetime
- Many software systems that are still in use were developed many years ago using technologies that are now obsolete
- These systems are still business critical that is, they are essential for the normal functioning of the business
- They have been given the name legacy systems

Legacy system replacement

- There is a significant business risk in simply scrapping a legacy system and replacing it with a system that has been developed using modern technology
 - Legacy systems rarely have a complete specification. During their lifetime they have undergone major changes which may not have been documented
 - Business processes are reliant on the legacy system
 - The system may embed business rules that are not formally documented elsewhere
 - New software development is risky and may not be successful

Legacy system change

- Systems must change in order to remain useful
- However, changing legacy systems is often expensive
 - Different parts implemented by different teams so no consistent programming style
 - The system may use an obsolete programming language
 - The system documentation is often out-of-date
 - The system structure may be corrupted by many years of maintenance
 - Techniques to save space or increase speed at the expense of understandability may have been used
 - File structures used may be incompatible

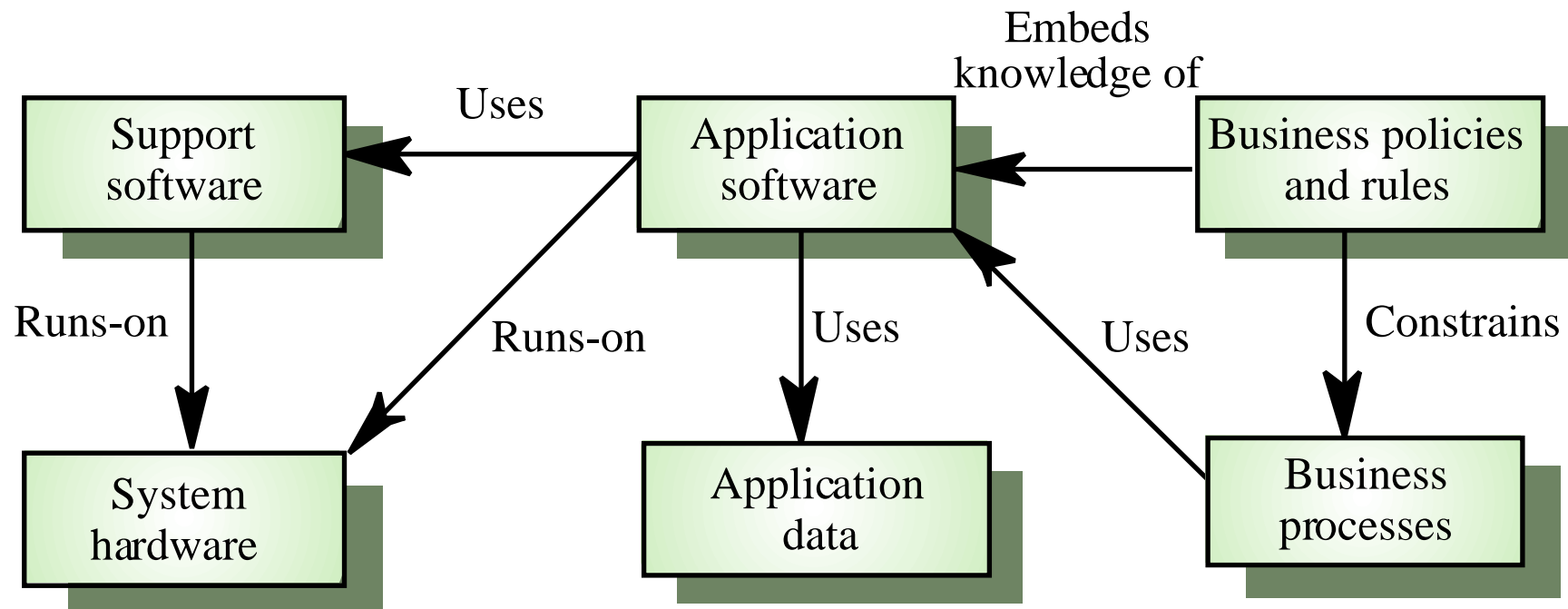
The legacy dilemma

- It is expensive and risky to replace the legacy system
- It is expensive to maintain the legacy system
- Businesses must weigh up the costs and risks and may choose to extend the system lifetime using techniques such as re-engineering.
- This is covered in Chapters 27 and 28

Legacy system structures

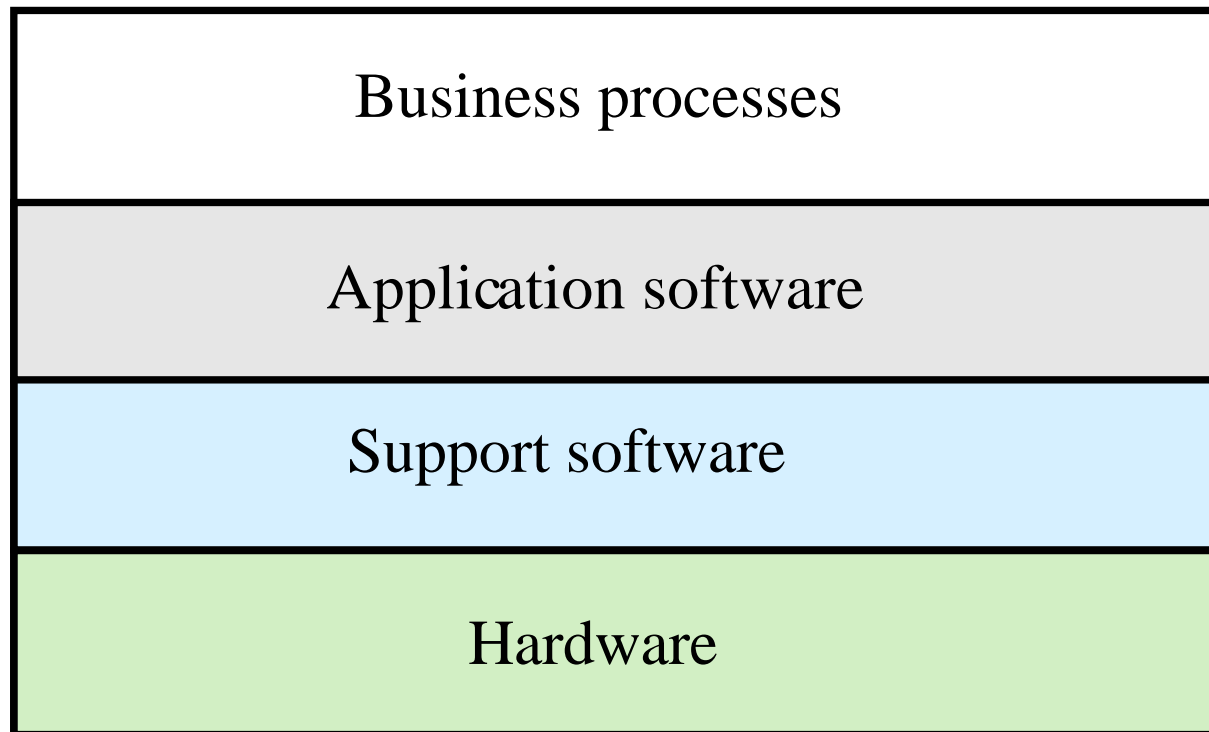
- Legacy systems can be considered to be socio-technical systems and not simply software systems
 - System hardware - may be mainframe hardware
 - Support software - operating systems and utilities
 - Application software - several different programs
 - Application data - data used by these programs that is often critical business information
 - Business processes - the processes that support a business objective and which rely on the legacy software and hardware
 - Business policies and rules - constraints on business operations

Legacy system components



Layered model

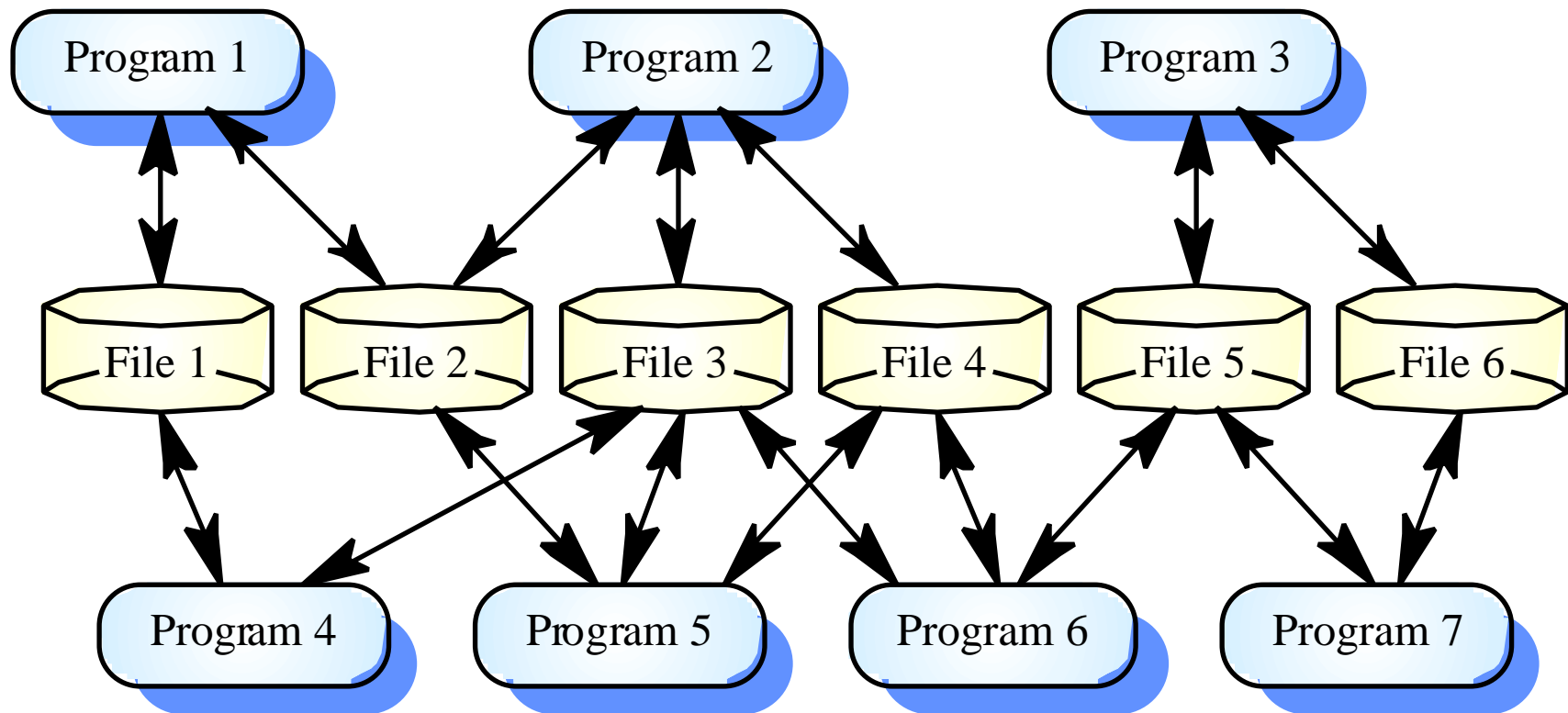
Socio-technical system



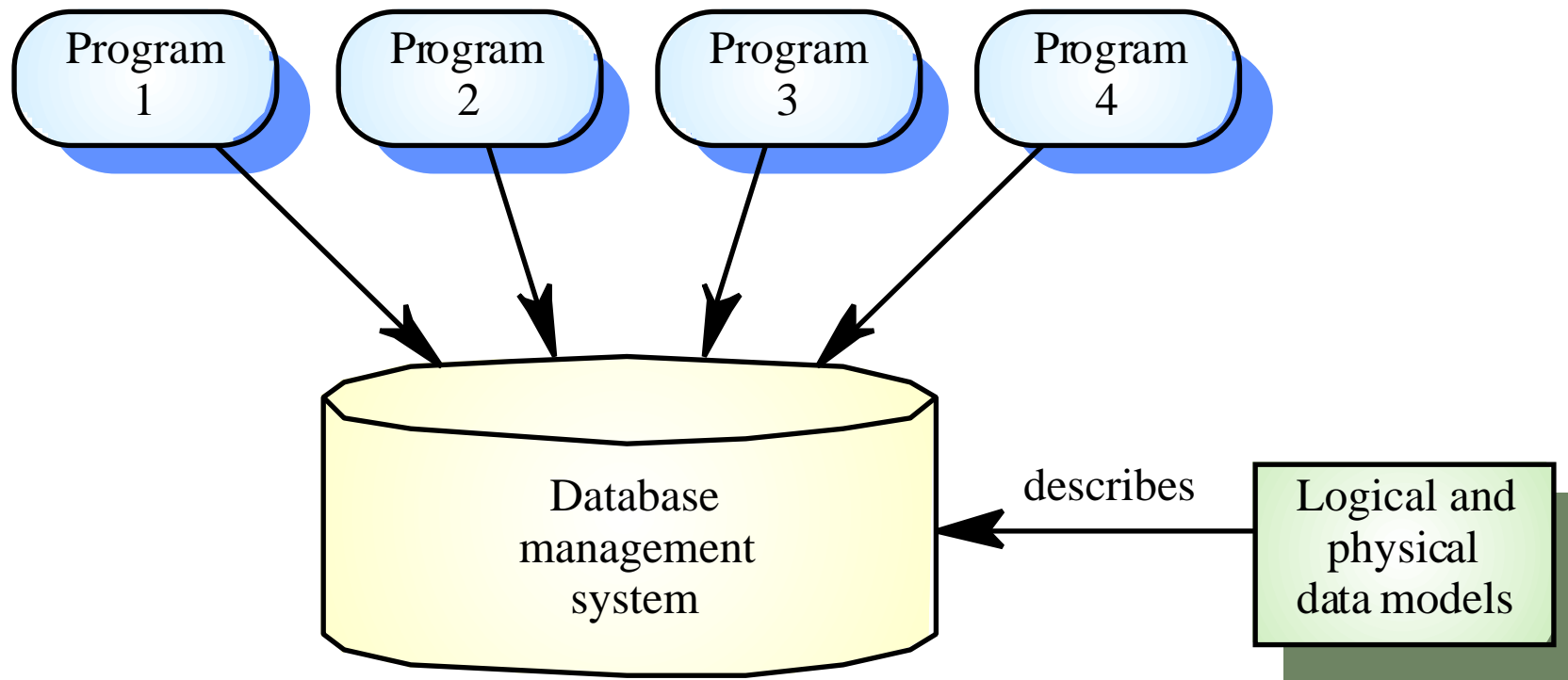
System change

- In principle, it should be possible to replace a layer in the system leaving the other layers unchanged
- In practice, this is usually impossible
 - Changing one layer introduces new facilities and higher level layers must then change to make use of these
 - Changing the software may slow it down so hardware changes are then required
 - It is often impossible to maintain hardware interfaces because of the wide gap between mainframes and client-server systems

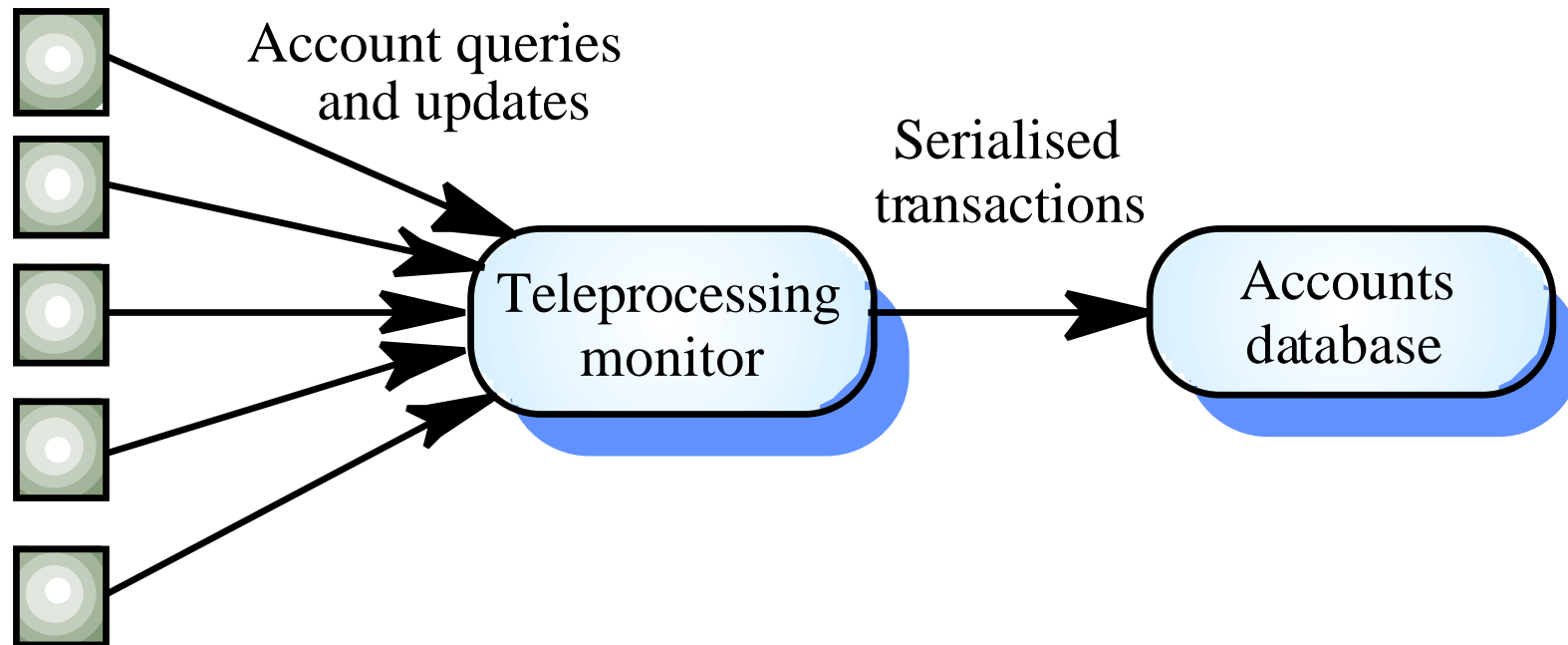
Legacy application system



Database-centred system



Transaction processing



ATMs and terminals

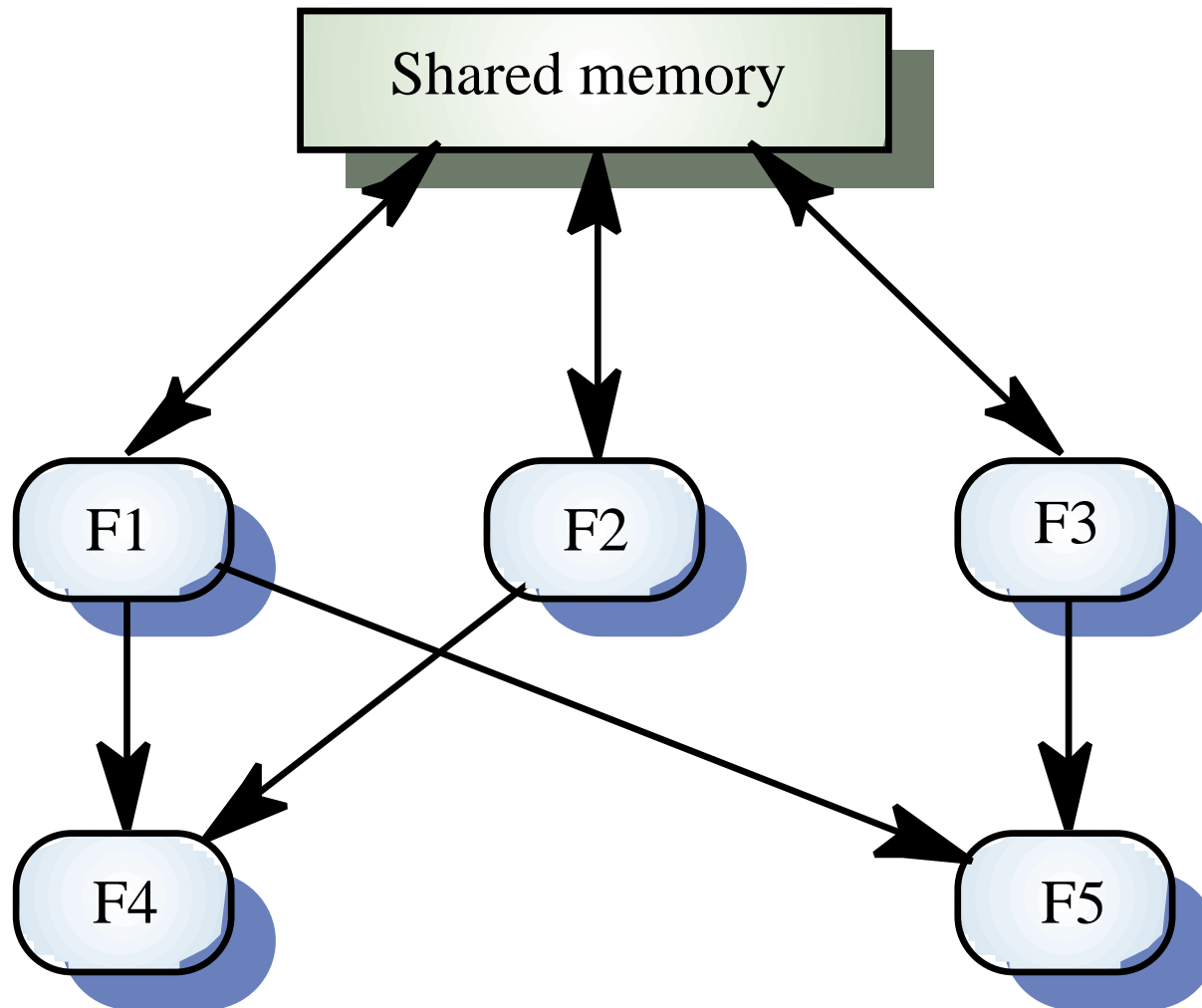
Legacy data

- The system may be file-based with incompatible files. The change required may be to move to a database-management system
- In legacy systems that use a DBMS the database management system may be obsolete and incompatible with other DBMSs used by the business
- The teleprocessing monitor may be designed for a particular DB and mainframe. Changing to a new DB may require a new TP monitor

Legacy system design

- Most legacy systems were designed before object-oriented development was used
- Rather than being organised as a set of interacting objects, these systems have been designed using a function-oriented design strategy
- Several methods and CASE tools are available to support function-oriented design and the approach is still used for many business applications

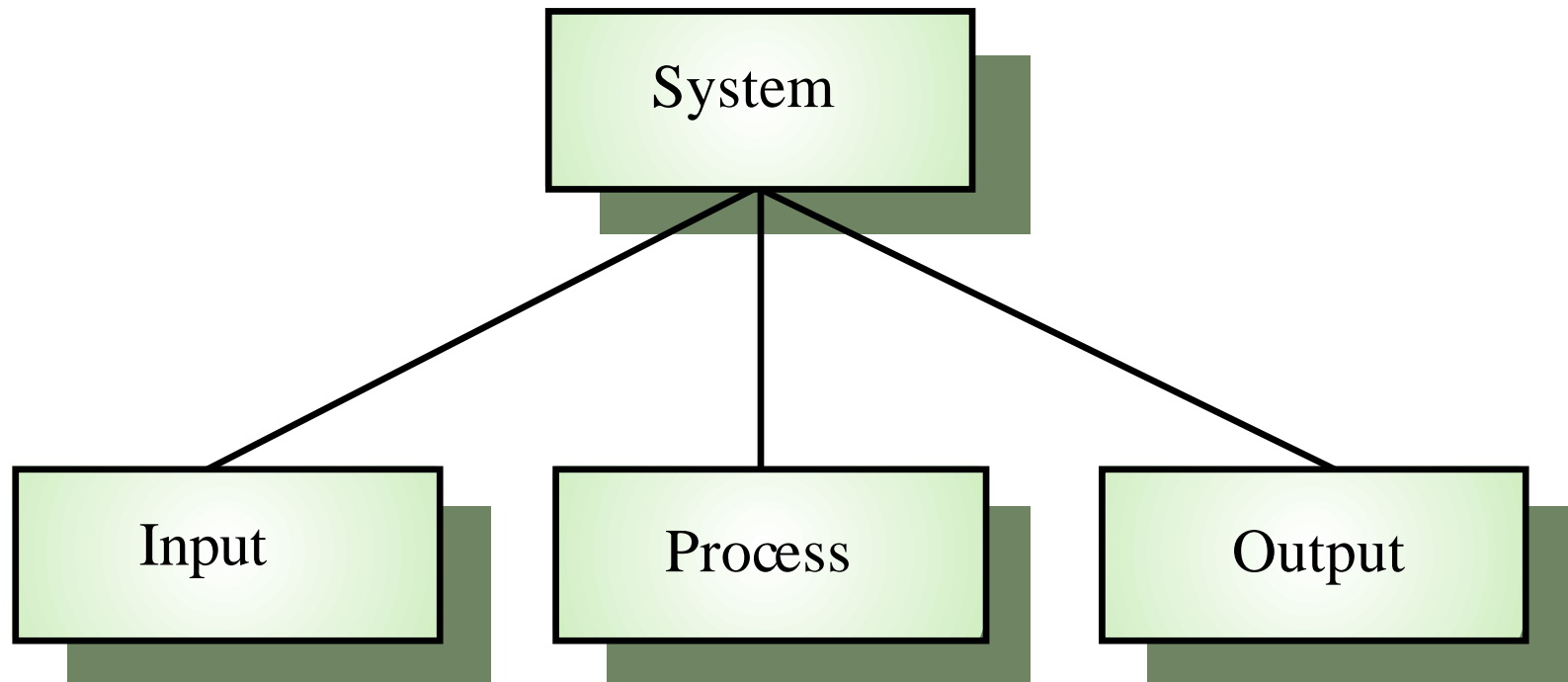
A function-oriented view of design



Functional design process

- Data-flow design
 - Model the data processing in the system using data-flow diagrams
- Structural decomposition
 - Model how functions are decomposed to sub-functions using graphical structure charts
- Detailed design
 - The entities in the design and their interfaces are described in detail. These may be recorded in a data dictionary and the design expressed using a PDL

Input-process-output model



Input-process-output

- Input components read and validate data from a terminal or file
- Processing components carry out some transformations on that data
- Output components format and print the results of the computation
- Input, process and output can all be represented as functions with data ‘flowing’ between them

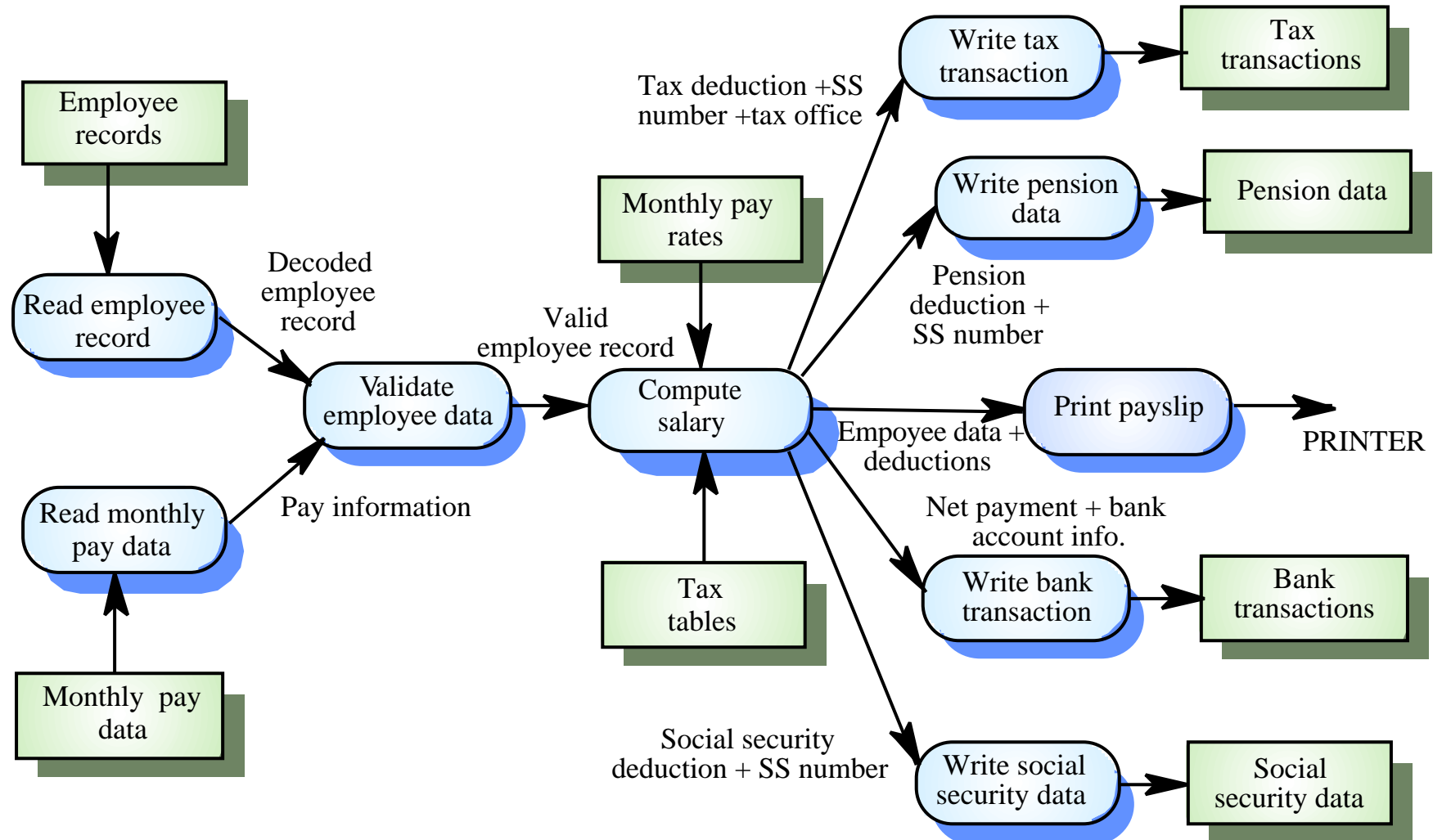
Functional design process

- Data-flow design
 - Model the data processing in the system using data-flow diagrams
- Structural decomposition
 - Model how functions are decomposed to sub-functions using graphical structure charts that reflect the input/process/output structure
- Detailed design
 - The functions in the design and their interfaces are described in detail.

Data flow diagrams

- Show how an input data item is functionally transformed by a system into an output data item
- Are an integral part of many design methods and are supported by many CASE systems
- May be translated into either a sequential or parallel design. In a sequential design, processing elements are functions or procedures; in a parallel design, processing elements are tasks or processes

Payroll system DFD



Payroll batch processing

- The functions on the left of the DFD are input functions
 - Read employee record, Read monthly pay data, Validate employee data
- The central function - Compute salary - carries out the processing
- The functions to the right are output functions
 - Write tax transaction, Write pension data, Print payslip, Write bank transaction, Write social security data

Transaction processing

- A ban ATM system is an example of a transaction processing system
- Transactions are stateless in that they do not rely on the result of previous transactions. Therefore, a functional approach is a natural way to implement transaction processing

INPUT

loop

repeat

```
Print_input_message (" Welcome - Please enter your card") ;  
until Card_input ;
```

```
Account_number := Read_card ;  
Get_account_details (PIN, Account_balance, Cash_available) ;
```

PROCESS

```
if Invalid_card (PIN) then
```

```
    Retain_card ;
```

```
    Print ("Card retained - please contact your bank") ;
```

```
else
```

repeat

```
    Print_operation_select_message ;
```

```
    Button := Get_button ;
```

```
    case Get_button is
```

```
        when Cash_only =>
```

```
            Dispense_cash (Cash_available, Amount_dispensed) ;
```

```
        when Print_balance =>
```

```
            Print_customer_balance (Account_balance) ;
```

```
        when Statement =>
```

```
            Order_statement (Account_number) ;
```

```
        when Check_book =>
```

```
            Order_checkbook (Account_number) ;
```

```
    end case ;
```

```
    Print ("Press CONTINUE for more services or STOP to finish");
```

```
    Button := Get_button ;
```

```
until Button = STOP ;
```

OUTPUT

```
Eject_card ;
```

```
Print ("Please take your card ) ;
```

```
Update_account_information (Account_number, Amount_dispensed) ;
```

```
end loop ;
```

Design
description of an
ATM

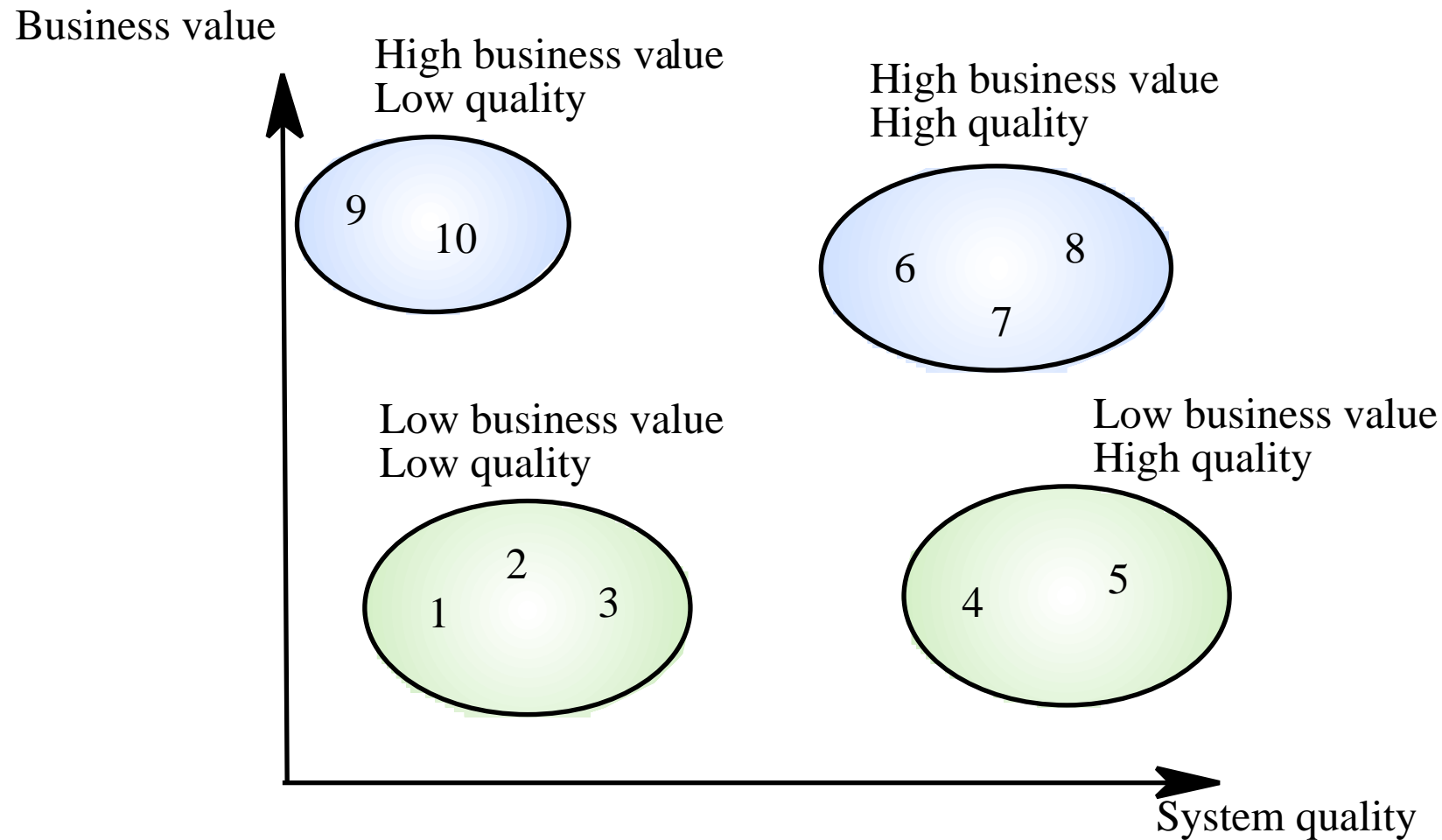
Using function-oriented design

- For some classes of system, such as some transaction processing systems, a function-oriented approach may be a better approach to design than an object-oriented approach
- Companies may have invested in CASE tools and methods for function-oriented design and may not wish to incur the costs and risks of moving to an object-oriented approach

Legacy system assessment

- Organisations that rely on legacy systems must choose a strategy for evolving these systems
 - Scrap the system completely and modify business processes so that it is no longer required
 - Continue maintaining the system
 - Transform the system by re-engineering to improve its maintainability
 - Replace the system with a new system
- The strategy chosen should depend on the system quality and its business value

System quality and business value



Legacy system categories

- Low quality, low business value
 - These systems should be scrapped
- Low-quality, high-business value
 - These make an important business contribution but are expensive to maintain. Should be re-engineered or replaced if a suitable system is available
- High-quality, low-business value
 - Replace with COTS, scrap completely or maintain
- High-quality, high business value
 - Continue in operation using normal system maintenance

Business value assessment

- Assessment should take different viewpoints into account
 - System end-users
 - Business customers
 - Line managers
 - IT managers
 - Senior managers
- Interview different stakeholders and collate results

System quality assessment

- Business process assessment
 - How well does the business process support the current goals of the business?
- Environment assessment
 - How effective is the system's environment and how expensive is it to maintain
- Application assessment
 - What is the quality of the application software system

Business process assessment

- Use a viewpoint-oriented approach and seek answers from system stakeholders
 - Is there a defined process model and is it followed?
 - Do different parts of the organisation use different processes for the same function?
 - How has the process been adapted?
 - What are the relationships with other business processes and are these necessary?
 - Is the process effectively supported by the legacy application software?

Environment assessment

Factor	Questions
Supplier stability	Is the supplier is still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, are the systems maintained by someone else?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly but there could be significant economic and business benefits to moving to more modern systems.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?
Support requirements	What local support is required by the hardware and software? If there are high costs associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licences? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers etc. be used with current versions of the operating system? Is hardware emulation required?

Application assessment

Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures which are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent and up-to-date?
Data	Is there an explicit data model for the system? To what extent is data duplicated in different files? Is the data used by the system up-to-date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there only a limited number of people who understand the system?

System measurement

- You may collect quantitative data to make an assessment of the quality of the application system
 - The number of system change requests
 - The number of different user interfaces used by the system
 - The volume of data used by the system

Key points

- A legacy system is an old system that still provides essential business services
- Legacy systems are not just application software but also include business processes, support software and hardware
- Most legacy systems are made up of several different programs and shared data
- A function-oriented approach has been used in the design of most legacy systems

Key points

- The structure of legacy business systems normally follows an input-process-output model
- The business value of a system and its quality should be used to choose an evolution strategy
- The business value reflects the system's effectiveness in supporting business goals
- System quality depends on business processes, the system's environment and the application software

Software change

- Managing the processes of software system change

Objectives

- To explain different strategies for changing software systems
 - Software maintenance
 - Architectural evolution
 - Software re-engineering
- To explain the principles of software maintenance
- To describe the transformation of legacy systems from centralised to distributed architectures

Topics covered

- Program evolution dynamics
- Software maintenance
- Architectural evolution

Software change

- Software change is inevitable
 - New requirements emerge when the software is used
 - The business environment changes
 - Errors must be repaired
 - New equipment must be accommodated
 - The performance or reliability may have to be improved
- A key problem for organisations is implementing and managing change to their legacy systems

Software change strategies

- Software maintenance
 - Changes are made in response to changed requirements but the fundamental software structure is stable
- Architectural transformation
 - The architecture of the system is modified generally from a centralised architecture to a distributed architecture
- Software re-engineering
 - No new functionality is added to the system but it is restructured and reorganised to facilitate future changes
- These strategies may be applied separately or together

Program evolution dynamics

- Program evolution dynamics is the study of the processes of system change
- After major empirical study, Lehman and Belady proposed that there were a number of ‘laws’ which applied to all systems as they evolved
- There are sensible observations rather than laws. They are applicable to large systems developed by large organisations. Perhaps less applicable in other cases

Lehman's laws

Law	Description
Continuing change	A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors are approximately invariant for each system release.
Organisational stability	Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.

Applicability of Lehman's laws

- This has not yet been established
- They are generally applicable to large, tailored systems developed by large organisations
- It is not clear how they should be modified for
 - Shrink-wrapped software products
 - Systems that incorporate a significant number of COTS components
 - Small organisations
 - Medium sized systems

Software maintenance

- Modifying a program after it has been put into use
- Maintenance does not normally involve major changes to the system's architecture
- Changes are implemented by modifying existing components and adding new components to the system

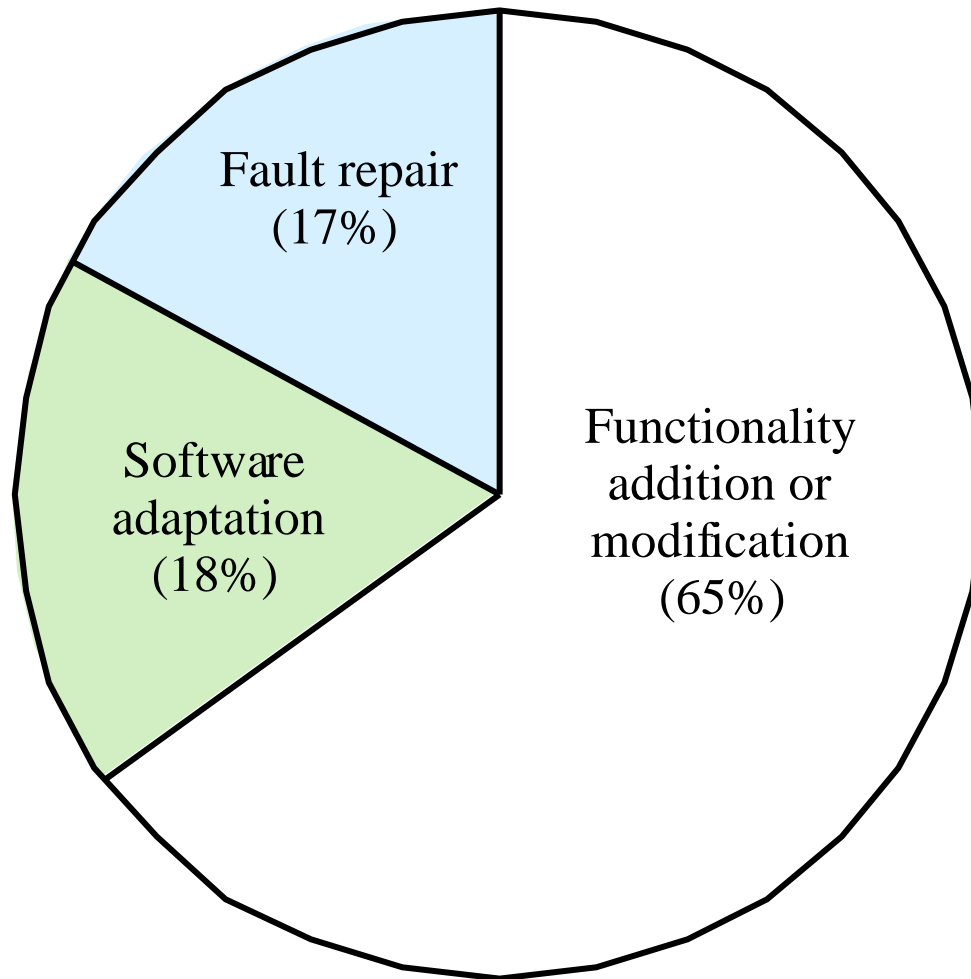
Maintenance is inevitable

- The system requirements are likely to change while the system is being developed because the environment is changing. Therefore a delivered system won't meet its requirements!
- Systems are tightly coupled with their environment. When a system is installed in an environment it changes that environment and therefore changes the system requirements.
- Systems **MUST** be maintained therefore if they are to remain useful in an environment

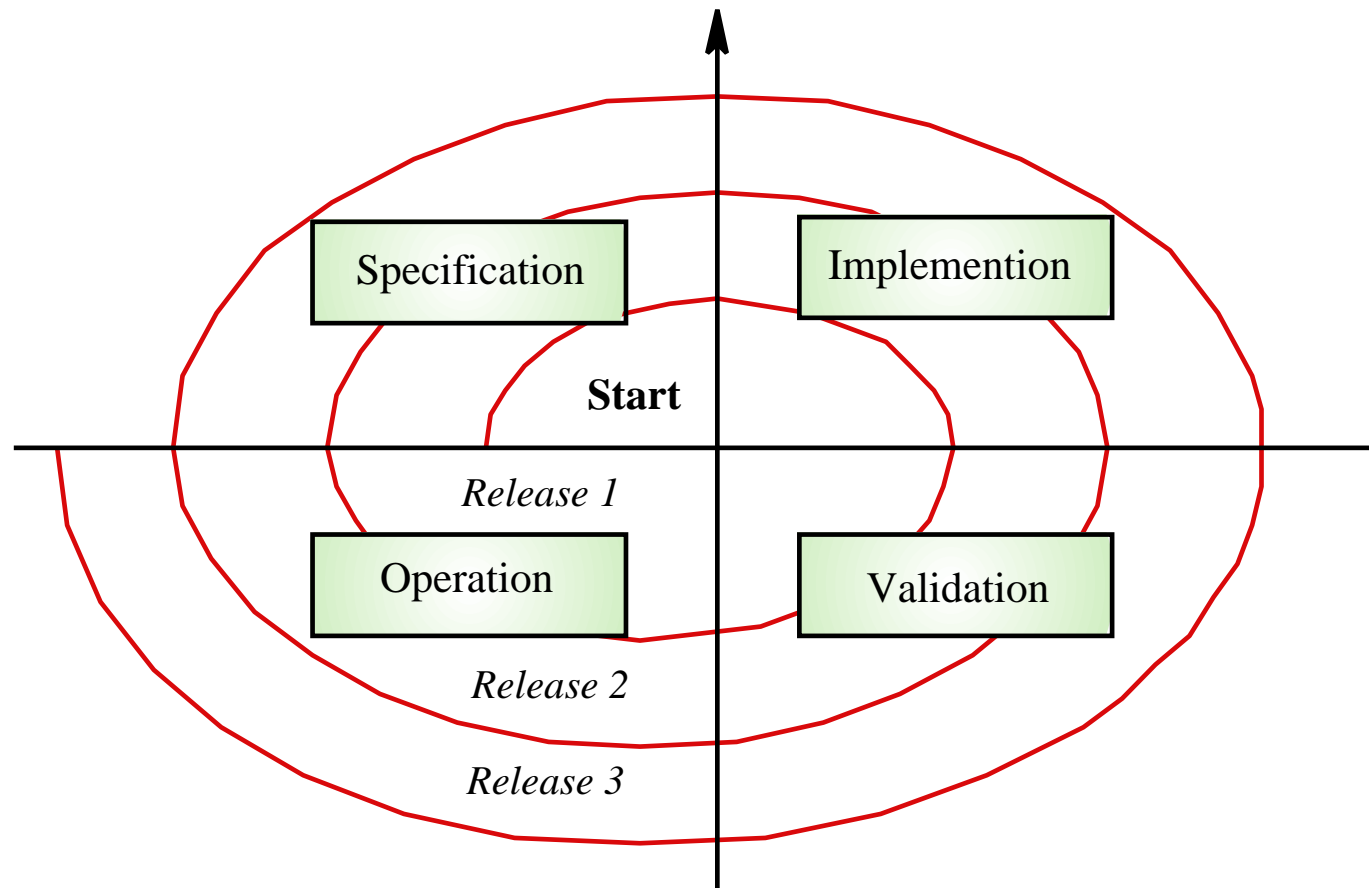
Types of maintenance

- Maintenance to repair software faults
 - Changing a system to correct deficiencies in the way meets its requirements
- Maintenance to adapt software to a different operating environment
 - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation
- Maintenance to add to or modify the system's functionality
 - Modifying the system to satisfy new requirements

Distribution of maintenance effort



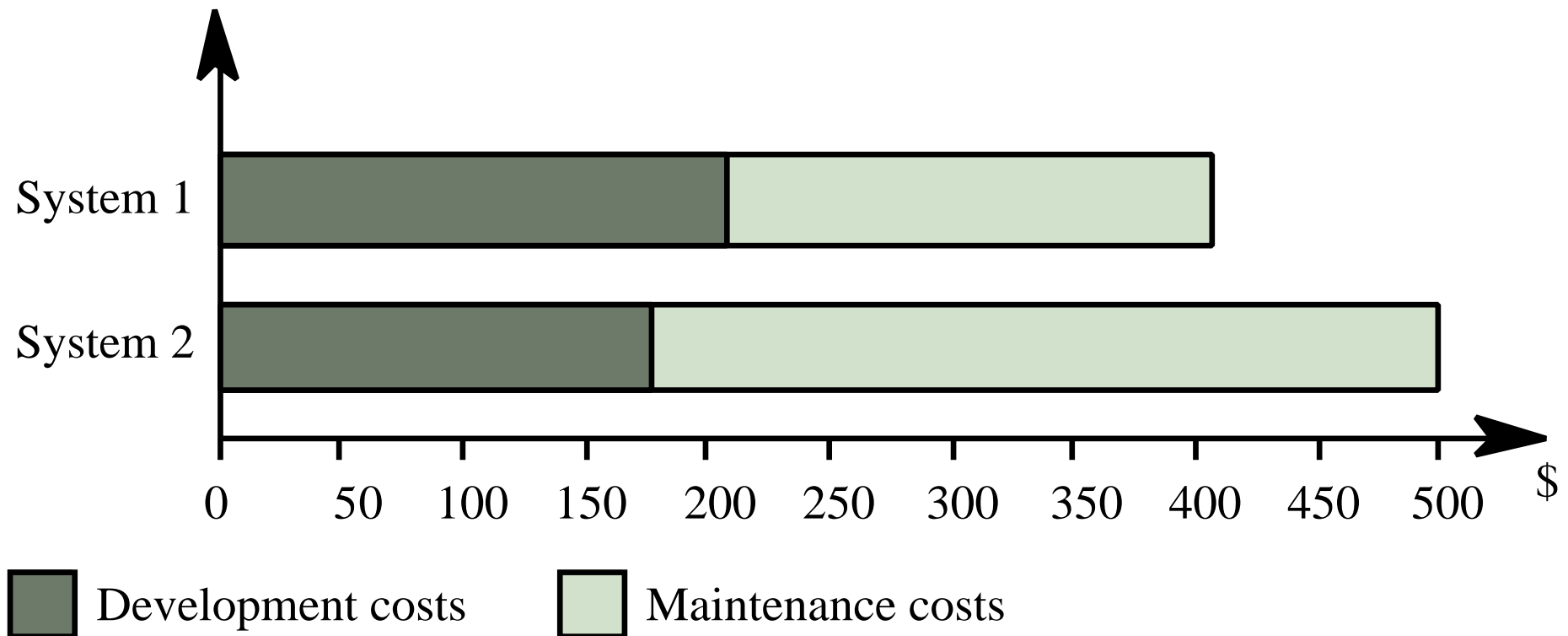
Spiral maintenance model



Maintenance costs

- Usually greater than development costs (2* to 100* depending on the application)
- Affected by both technical and non-technical factors
- Increases as software is maintained.
Maintenance corrupts the software structure so makes further maintenance more difficult.
- Ageing software can have high support costs (e.g. old languages, compilers etc.)

Development/maintenance costs



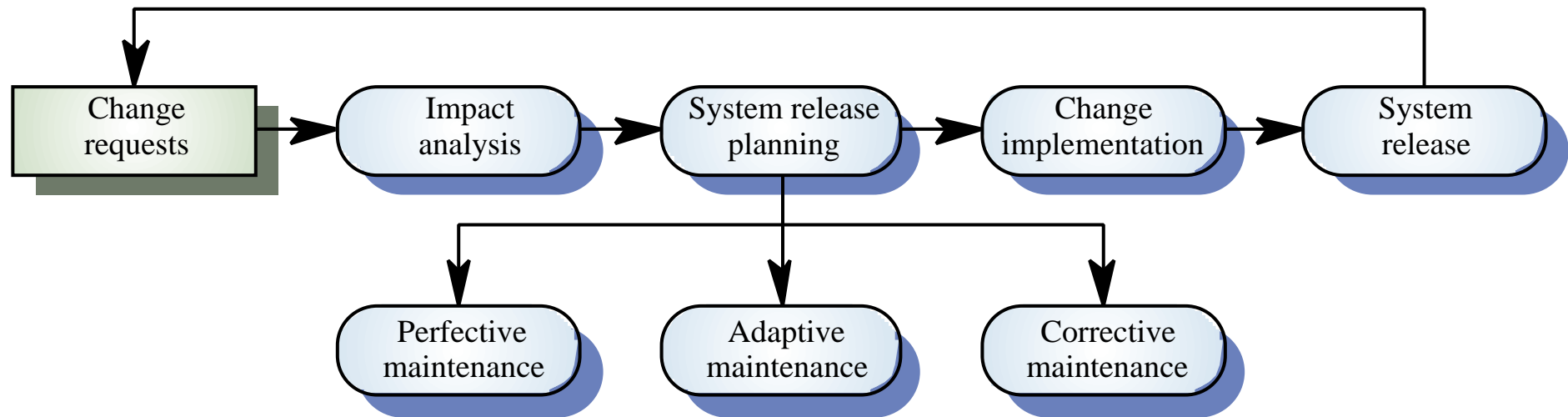
Maintenance cost factors

- Team stability
 - Maintenance costs are reduced if the same staff are involved with them for some time
- Contractual responsibility
 - The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change
- Staff skills
 - Maintenance staff are often inexperienced and have limited domain knowledge
- Program age and structure
 - As programs age, their structure is degraded and they become harder to understand and change

Evolutionary software

- Rather than think of separate development and maintenance phases, evolutionary software is software that is designed so that it can continuously evolve throughout its lifetime

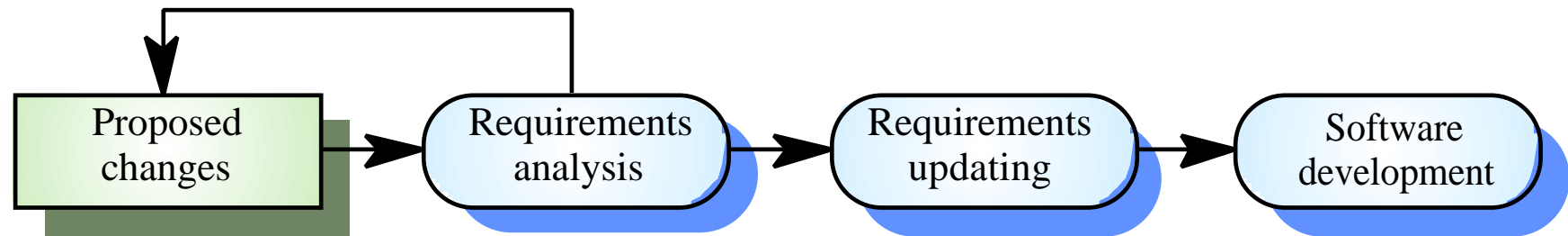
The maintenance process



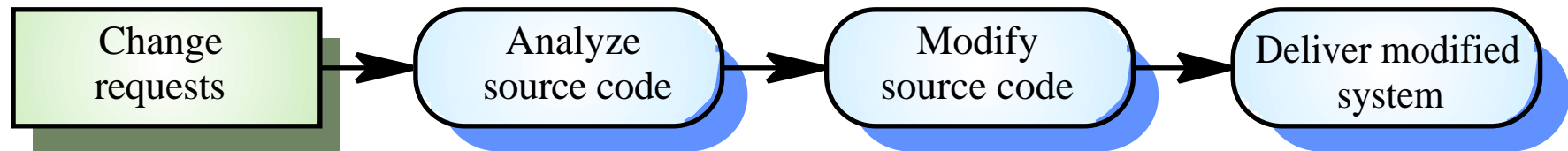
Change requests

- Change requests are requests for system changes from users, customers or management
- In principle, all change requests should be carefully analysed as part of the maintenance process and then implemented
- In practice, some change requests must be implemented urgently
 - Fault repair
 - Changes to the system's environment
 - Urgently required business changes

Change implementation



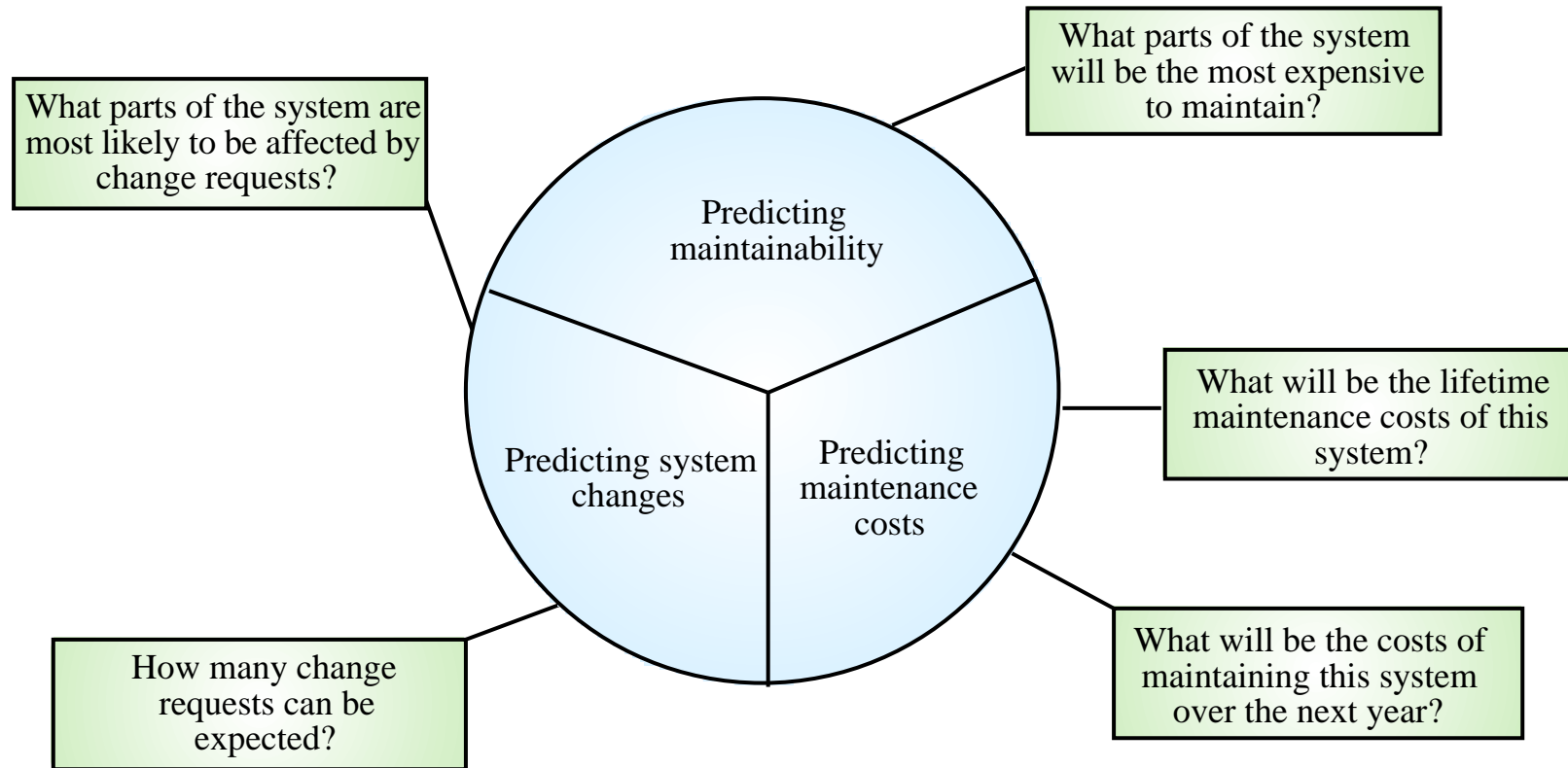
Emergency repair



Maintenance prediction

- Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs
 - Change acceptance depends on the maintainability of the components affected by the change
 - Implementing changes degrades the system and reduces its maintainability
 - Maintenance costs depend on the number of changes and costs of change depend on maintainability

Maintenance prediction



Change prediction

- Predicting the number of changes requires and understanding of the relationships between a system and its environment
- Tightly coupled systems require changes whenever the environment is changed
- Factors influencing this relationship are
 - Number and complexity of system interfaces
 - Number of inherently volatile system requirements
 - The business processes where the system is used

Complexity metrics

- Predictions of maintainability can be made by assessing the complexity of system components
- Studies have shown that most maintenance effort is spent on a relatively small number of system components
- Complexity depends on
 - Complexity of control structures
 - Complexity of data structures
 - Procedure and module size

Process metrics

- Process measurements may be used to assess maintainability
 - Number of requests for corrective maintenance
 - Average time required for impact analysis
 - Average time taken to implement a change request
 - Number of outstanding change requests
- If any or all of these is increasing, this may indicate a decline in maintainability

Architectural evolution

- There is a need to convert many legacy systems from a centralised architecture to a client-server architecture
- Change drivers
 - Hardware costs. Servers are cheaper than mainframes
 - User interface expectations. Users expect graphical user interfaces
 - Distributed access to systems. Users wish to access the system from different, geographically separated, computers

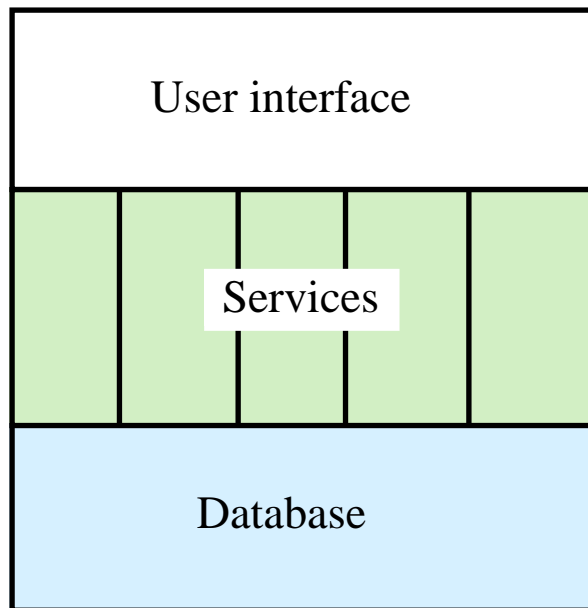
Distribution factors

Factor	Description
Business importance	Returns on the investment of distributing a legacy system depend on its importance to the business and how long it will remain important. If distribution provides more efficient support for stable business processes then it is more likely to be a cost-effective evolution strategy.
System age	The older the system the more difficult it will be to modify its architecture because previous changes will have degraded the structure of the system.
System structure	The more modular the system, the easier it will be to change the architecture. If the application logic, the data management and the user interface of the system are closely intertwined, it will be difficult to separate functions for migration.
Hardware procurement policies	Application distribution may be necessary if there is company policy to replace expensive mainframe computers with cheaper servers. .

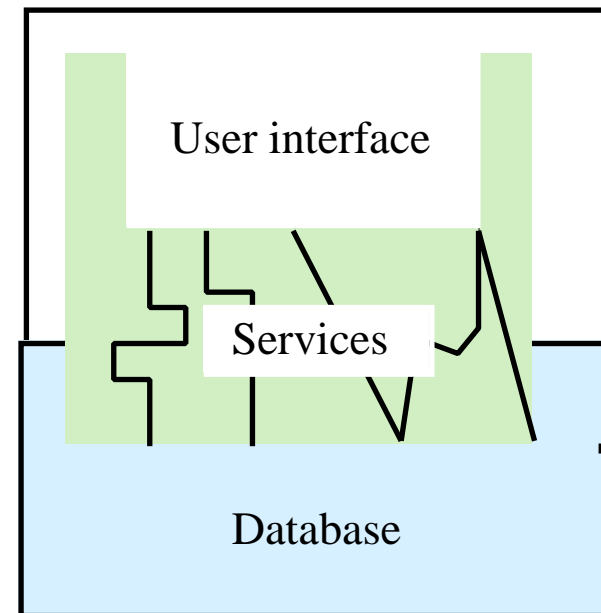
Legacy system structure

- Ideally, for distribution, there should be a clear separation between the user interface, the system services and the system data management
- In practice, these are usually intermingled in older legacy systems

Legacy system structures

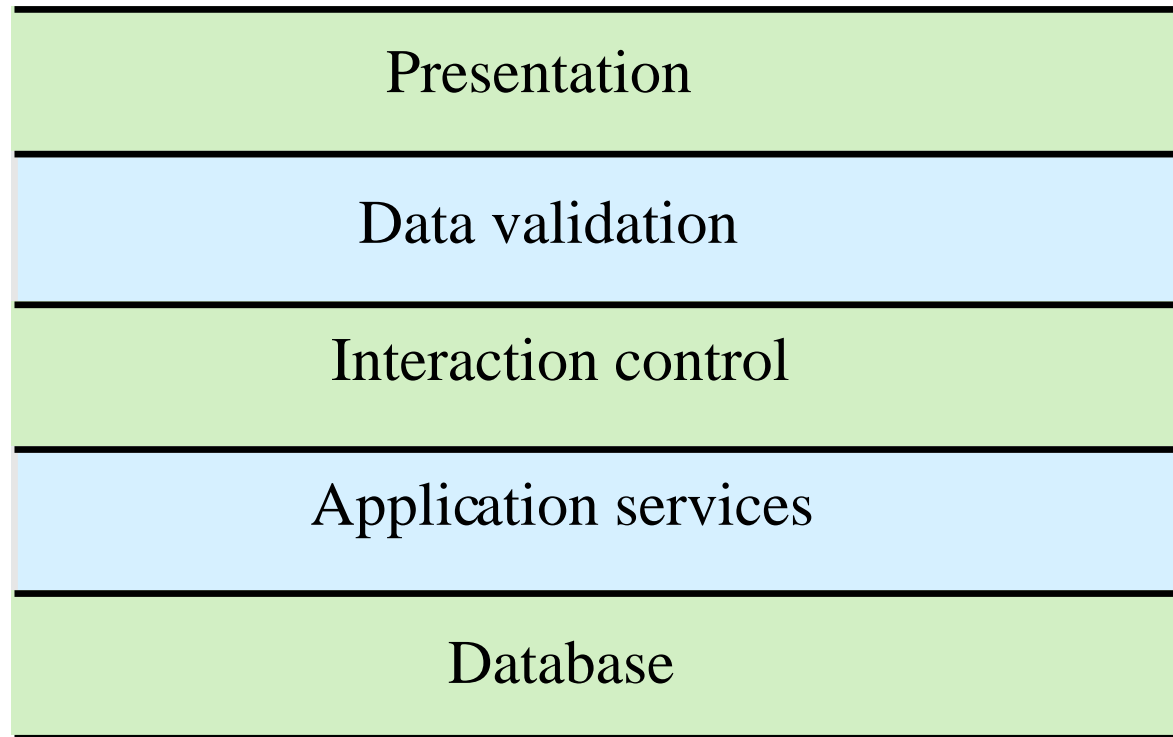


Ideal model for distribution

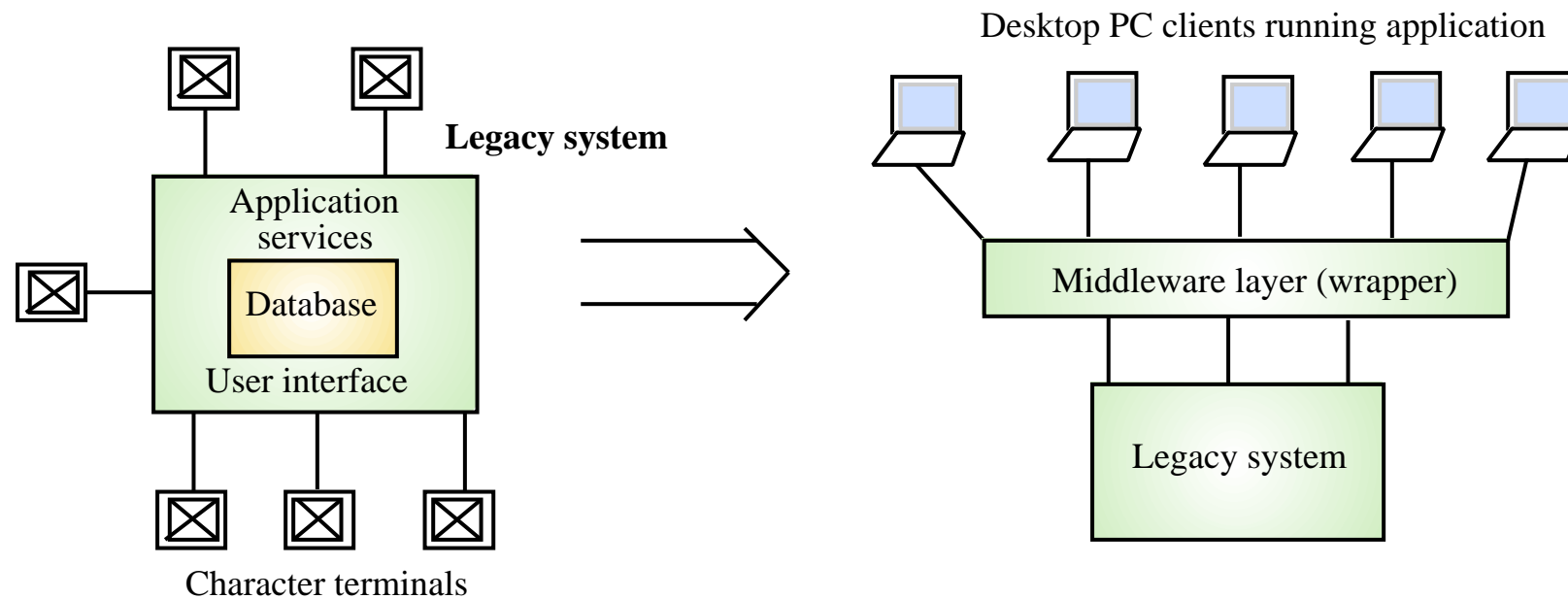


Real legacy systems

Layered distribution model



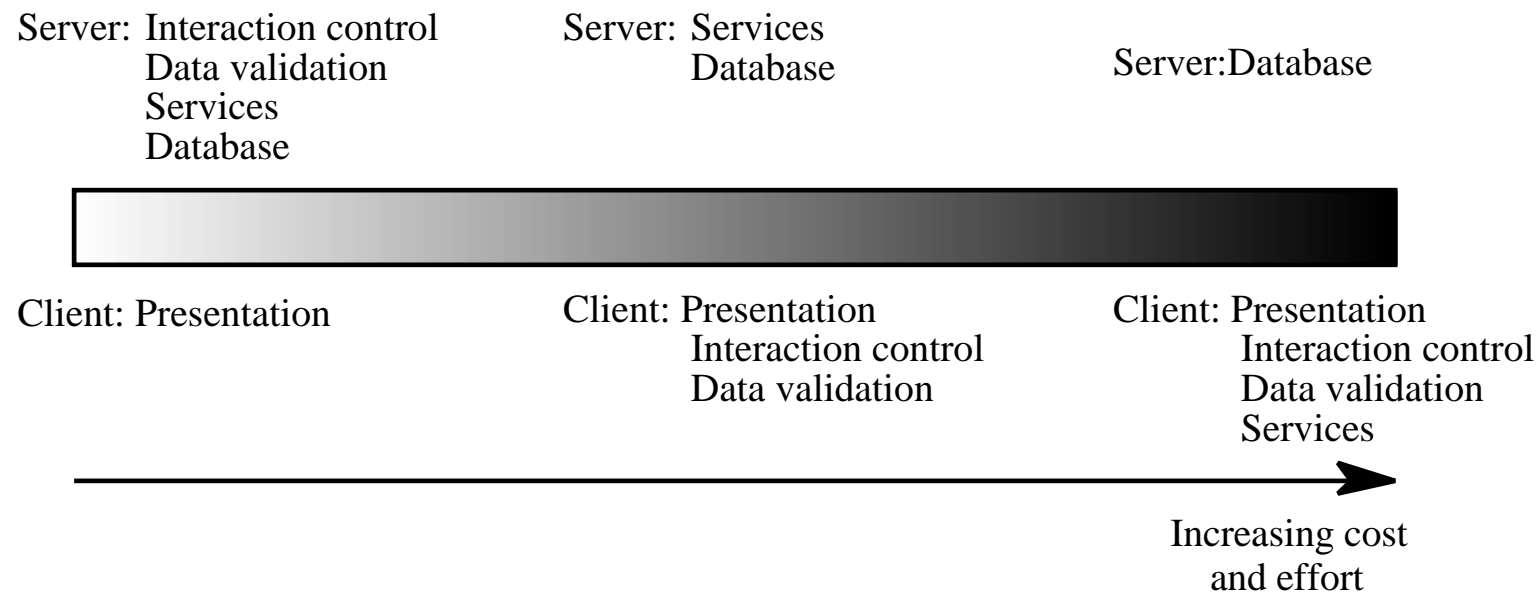
Legacy system distribution



Distribution options

- The more that is distributed from the server to the client, the higher the costs of architectural evolution
- The simplest distribution model is UI distribution where only the user interface is implemented on the server
- The most complex option is where the server simply provides data management and application services are implemented on the client

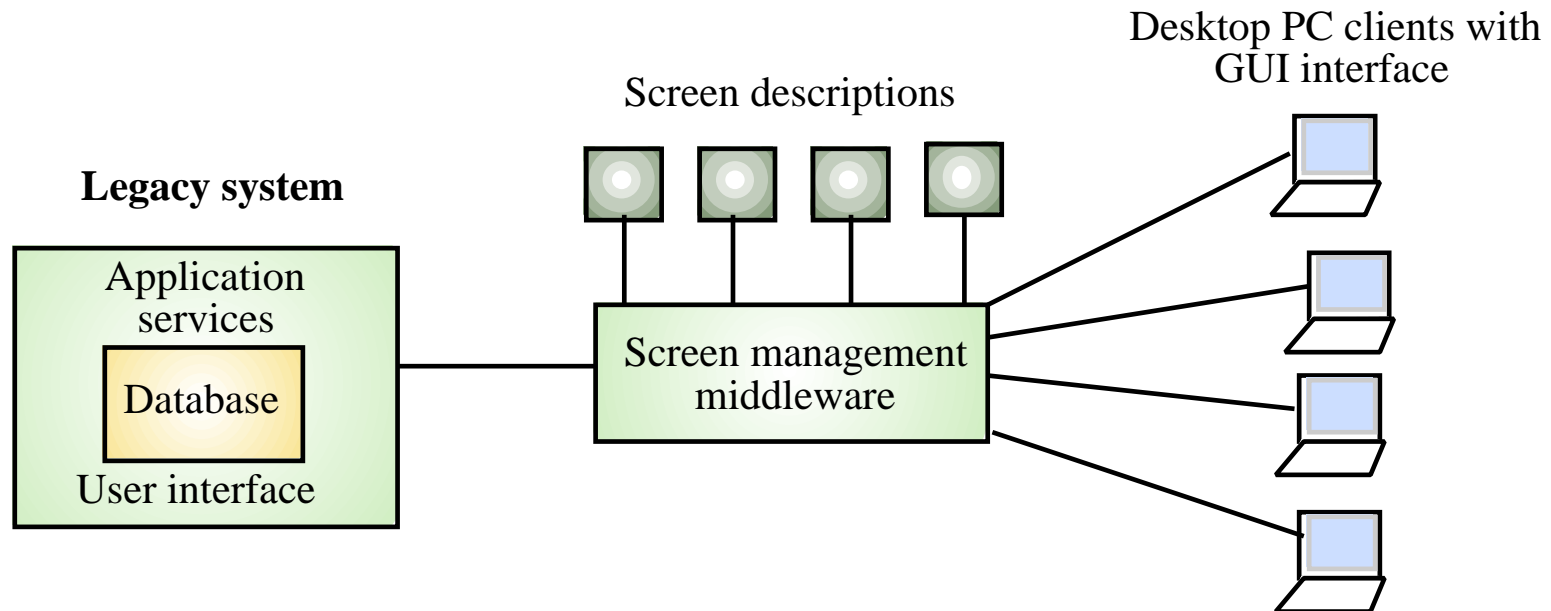
Distribution option spectrum



User interface distribution

- UI distribution takes advantage of the local processing power on PCs to implement a graphical user interface
- Where there is a clear separation between the UI and the application then the legacy system can be modified to distribute the UI
- Otherwise, screen management middleware can translate text interfaces to graphical interfaces

User interface distribution



UI migration strategies

Strategy	Advantages	Disadvantages
Implementation using the window management system	Access to all UI functions so no real restrictions on UI design Better UI performance	Platform dependent May be more difficult to achieve interface consistency
Implementation using a web browser	Platform independent Lower training costs due to user familiarity with the WWW Easier to achieve interface consistency	Potentially poorer UI performance Interface design is constrained by the facilities provided by web browsers

Key points

- Software change strategies include software maintenance, architectural evolution and software re-engineering
- Lehman's Laws are invariant relationships that affect the evolution of a software system
- Maintenance types are
 - Maintenance for repair
 - Maintenance for a new operating environment
 - Maintenance to implement new requirements

Key points

- The costs of software change usually exceed the costs of software development
- Factors influencing maintenance costs include staff stability, the nature of the development contract, skill shortages and degraded system structure
- Architectural evolution is concerned with evolving centralised to distributed architectures
- A distributed user interface can be supported using screen management middleware

Software re-engineering

- Reorganising and modifying existing software systems to make them more maintainable

Objectives

- To explain why software re-engineering is a cost-effective option for system evolution
- To describe the activities involved in the software re-engineering process
- To distinguish between software and data re-engineering and to explain the problems of data re-engineering

Topics covered

- Source code translation
- Reverse engineering
- Program structure improvement
- Program modularisation
- Data re-engineering

System re-engineering

- Re-structuring or re-writing part or all of a legacy system without changing its functionality
- Applicable where some but not all sub-systems of a larger system require frequent maintenance
- Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented

When to re-engineer

- When system changes are mostly confined to part of the system then re-engineer that part
- When hardware or software support becomes obsolete
- When tools to support re-structuring are available

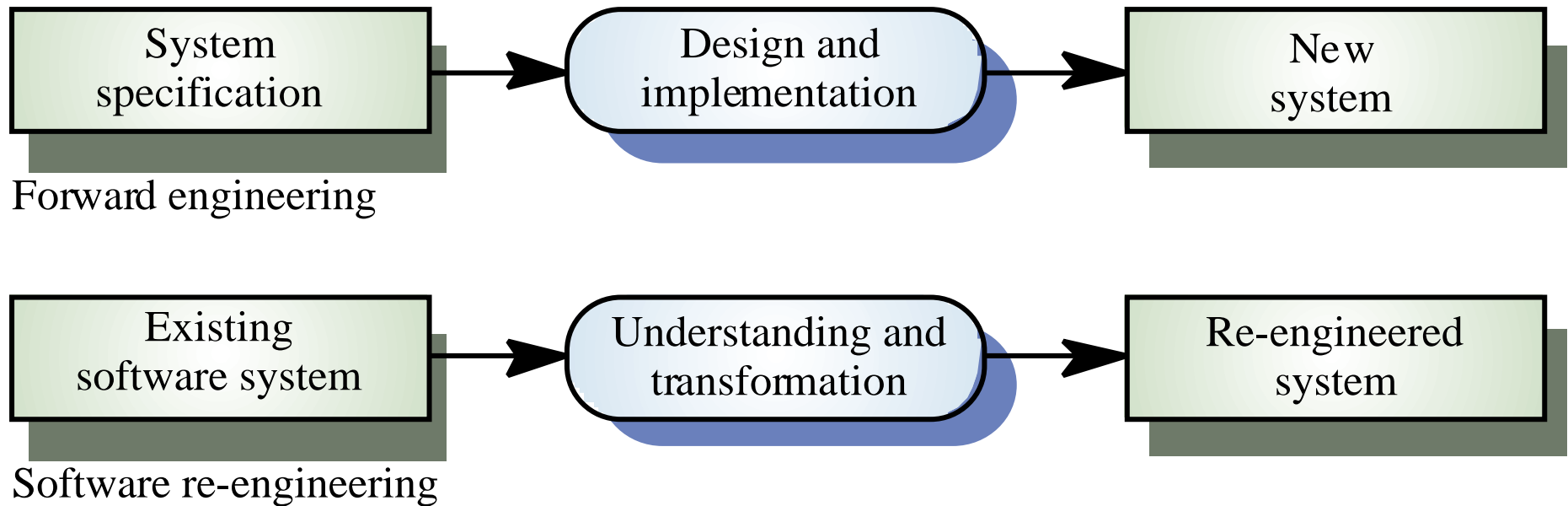
Re-engineering advantages

- Reduced risk
 - There is a high risk in new software development. There may be development problems, staffing problems and specification problems
- Reduced cost
 - The cost of re-engineering is often significantly less than the costs of developing new software

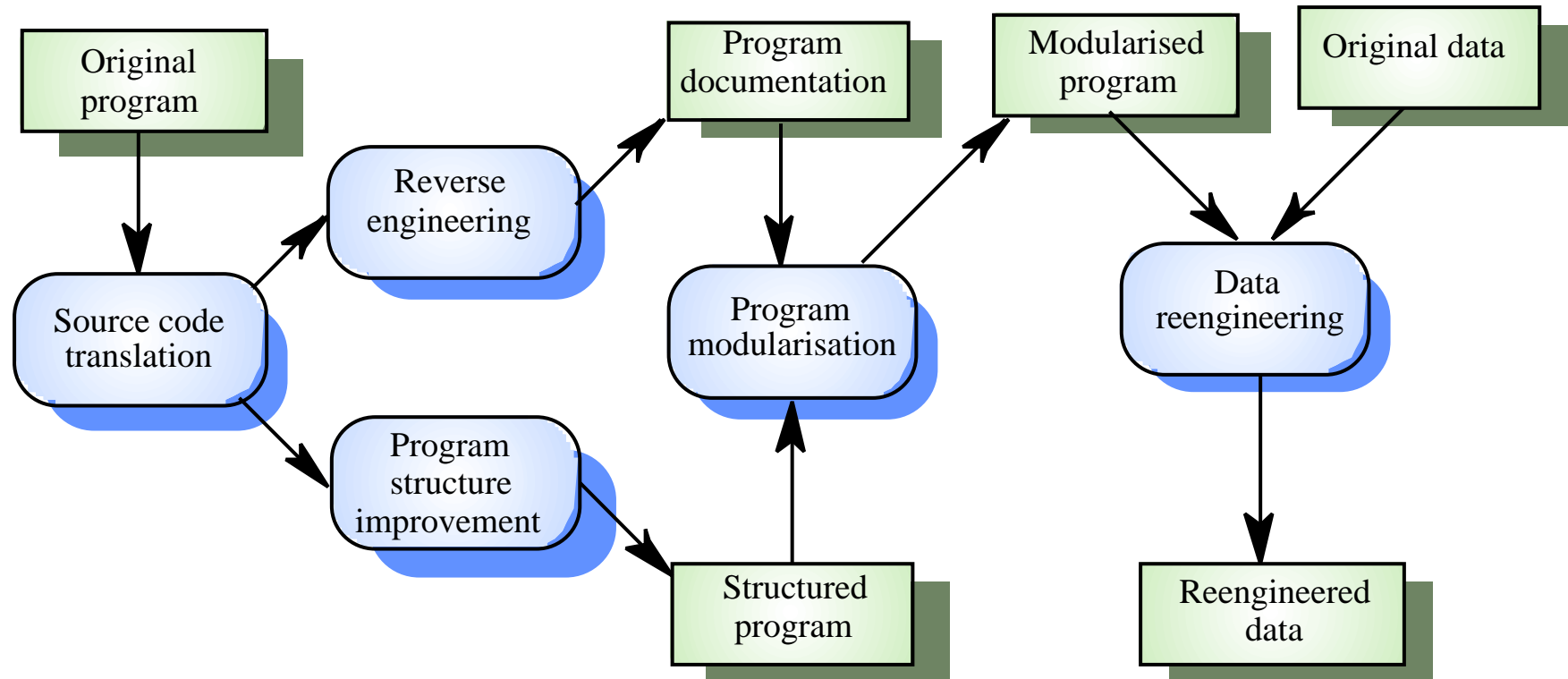
Business process re-engineering

- Concerned with re-designing business processes to make them more responsive and more efficient
- Often reliant on the introduction of new computer systems to support the revised processes
- May force software re-engineering as the legacy systems are designed to support existing processes

Forward engineering and re-engineering



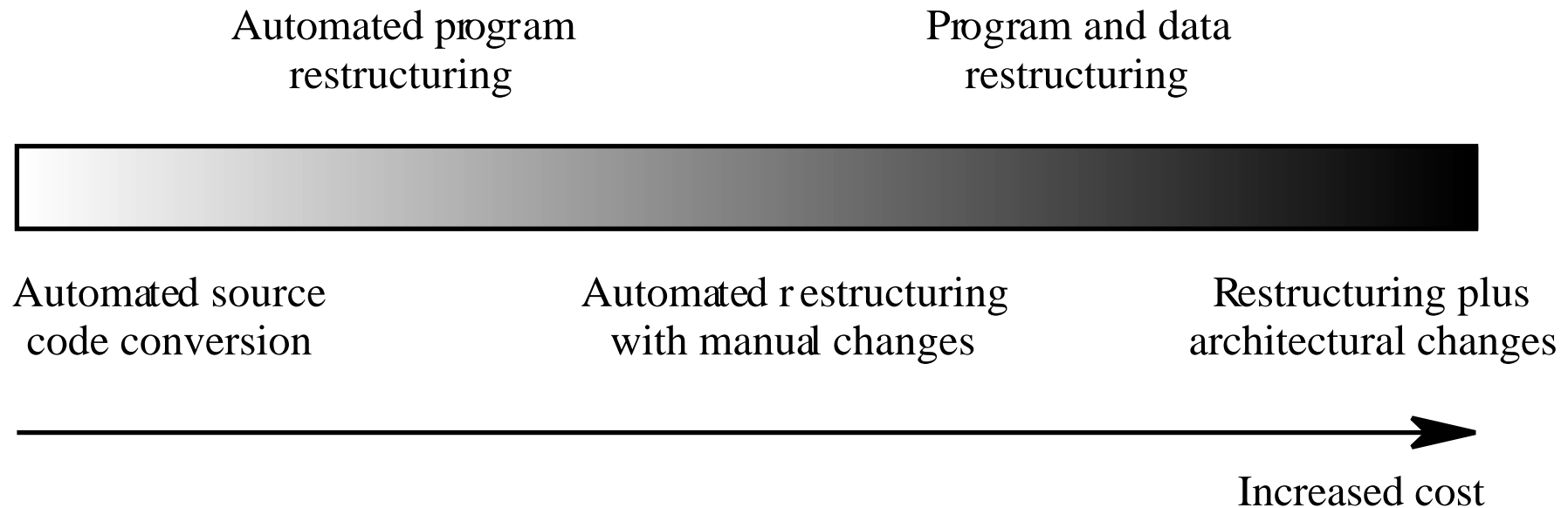
The re-engineering process



Re-engineering cost factors

- The quality of the software to be re-engineered
- The tool support available for re-engineering
- The extent of the data conversion which is required
- The availability of expert staff for re-engineering

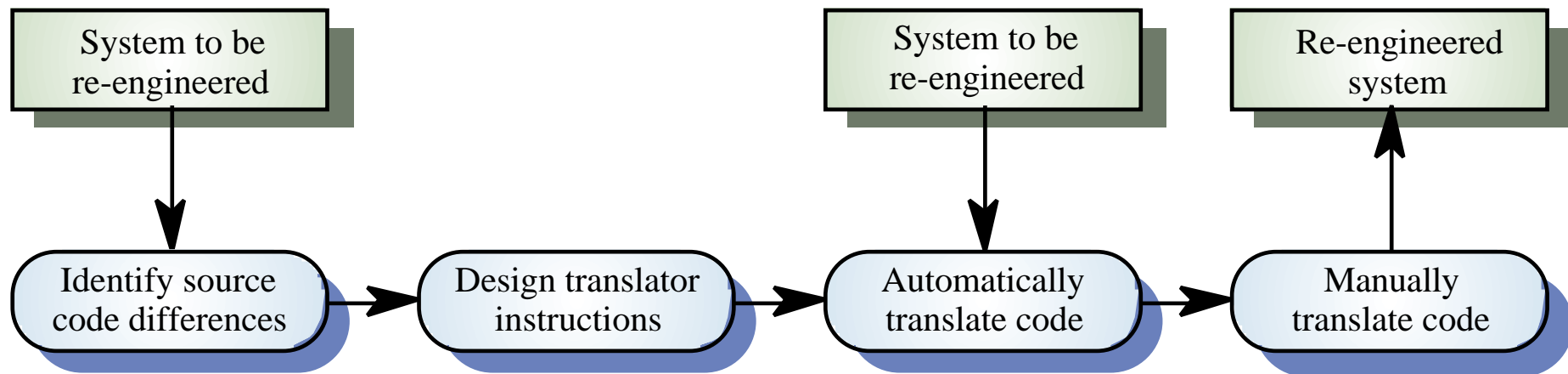
Re-engineering approaches



Source code translation

- Involves converting the code from one language (or language version) to another e.g. FORTRAN to C
- May be necessary because of:
 - Hardware platform update
 - Staff skill shortages
 - Organisational policy changes
- Only realistic if an automatic translator is available

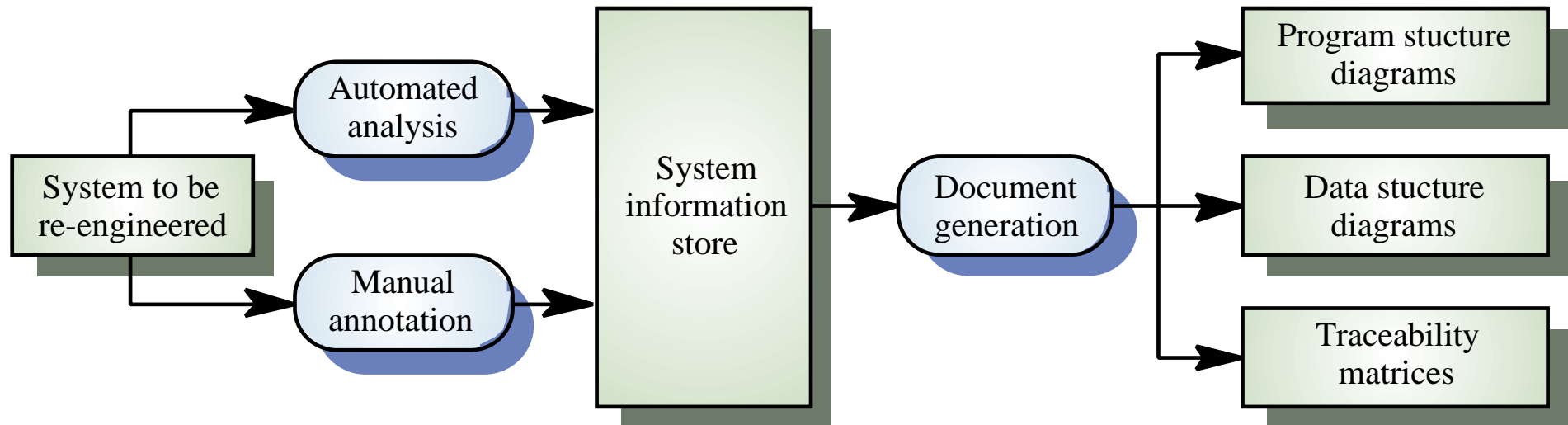
The program translation process



Reverse engineering

- Analysing software with a view to understanding its design and specification
- May be part of a re-engineering process but may also be used to re-specify a system for re-implementation
- Builds a program data base and generates information from this
- Program understanding tools (browsers, cross-reference generators, etc.) may be used in this process

The reverse engineering process



Reverse engineering

- Reverse engineering often precedes re-engineering but is sometimes worthwhile in its own right
 - The design and specification of a system may be reverse engineered so that they can be an input to the requirements specification process for the system's replacement
 - The design and specification may be reverse engineered to support program maintenance

Program structure improvement

- Maintenance tends to corrupt the structure of a program. It becomes harder and harder to understand
- The program may be automatically restructured to remove unconditional branches
- Conditions may be simplified to make them more readable

Spaghetti logic

```
Start:  Get (Time-on, Time-off, Time, Setting, Temp, Switch)
        if Switch = off goto off
        if Switch = on goto on
        goto Cntrld
off:    if Heating-status = on goto Sw-off
        goto loop
on:     if Heating-status = off goto Sw-on
        goto loop
Cntrld: if Time = Time-on goto on
        if Time = Time-off goto off
        if Time < Time-on goto Start
        if Time > Time-off goto Start
        if Temp > Setting then goto off
        if Temp < Setting then goto on
Sw-off: Heating-status := off
        goto Switch
Sw-on:  Heating-status := on
Switch: Switch-heating
loop:   goto Start
```

Structured control logic

```
loop
  -- The Get statement finds values for the given variables from the system's
  -- environment.
  Get (Time-on, Time-off, Time, Setting, Temp, Switch) ;
  case Switch of
    when On => if Heating-status = off then
      Switch-heating ; Heating-status := on ;
    end if ;
    when Off => if Heating-status = on then
      Switch-heating ; Heating-status := off ;
    end if ;
    when Controlled =>
      if Time >= Time-on and Time <= Time-off then
        if Temp > Setting and Heating-status = on then
          Switch-heating; Heating-status = off;
        elsif Temp < Setting and Heating-status = off then
          Switch-heating; Heating-status := on ;
        end if ;
      end if ;
    end case ;
  end loop ;
```


Condition simplification

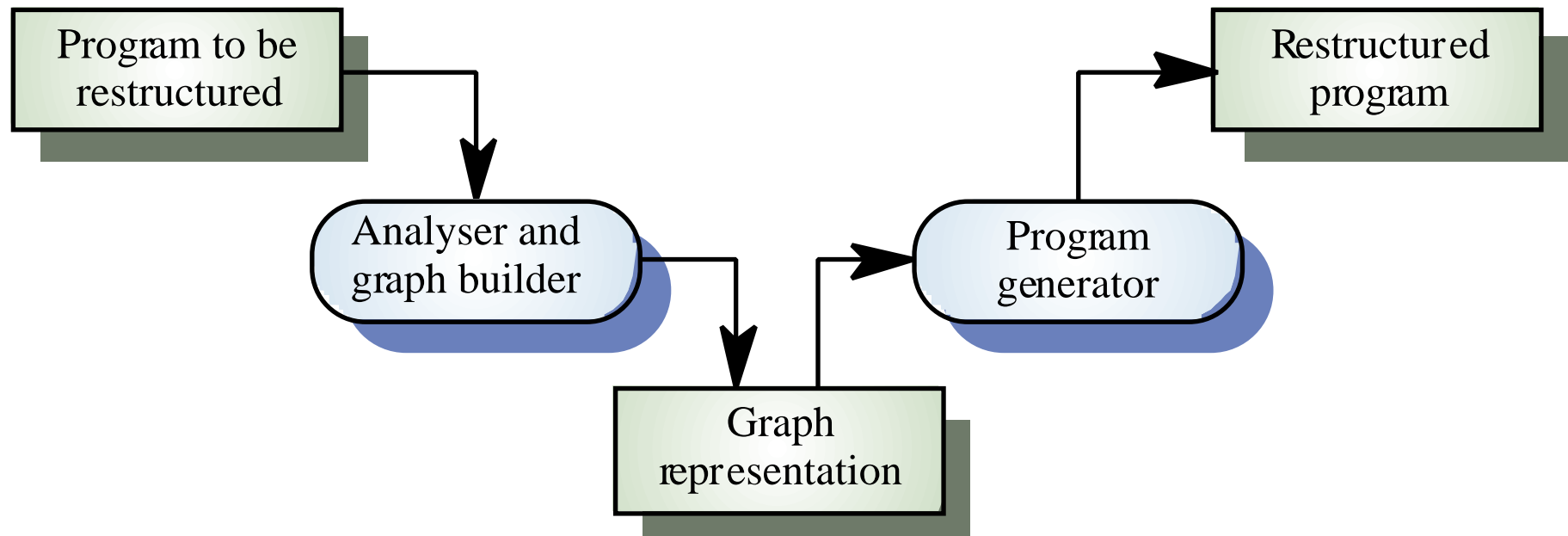
-- Complex condition

if not (A > B and (C < D or not (E > F)))...

-- Simplified condition

if (A <= B and (C>= D or E > F)...

Automatic program restructuring



Restructuring problems

- Problems with re-structuring are:
 - Loss of comments
 - Loss of documentation
 - Heavy computational demands
- Restructuring doesn't help with poor modularisation where related components are dispersed throughout the code
- The understandability of data-driven programs may not be improved by re-structuring

Program modularisation

- The process of re-organising a program so that related program parts are collected together in a single module
- Usually a manual process that is carried out by program inspection and re-organisation

Module types

- Data abstractions
 - Abstract data types where datastructures and associated operations are grouped
- Hardware modules
 - All functions required to interface with a hardware unit
- Functional modules
 - Modules containing functions that carry out closely related tasks
- Process support modules
 - Modules where the functions support a business process or process fragment

Recovering data abstractions

- Many legacy systems use shared tables and global data to save memory space
- Causes problems because changes have a wide impact in the system
- Shared global data may be converted to objects or ADTs
 - Analyse common data areas to identify logical abstractions
 - Create an ADT or object for these abstractions
 - Use a browser to find all data references and replace with reference to the data abstraction

Data abstraction recovery

- Analyse common data areas to identify logical abstractions
- Create an abstract data type or object class for each of these abstractions
- Provide functions to access and update each field of the data abstraction
- Use a program browser to find calls to these data abstractions and replace these with the new defined functions

Data re-engineering

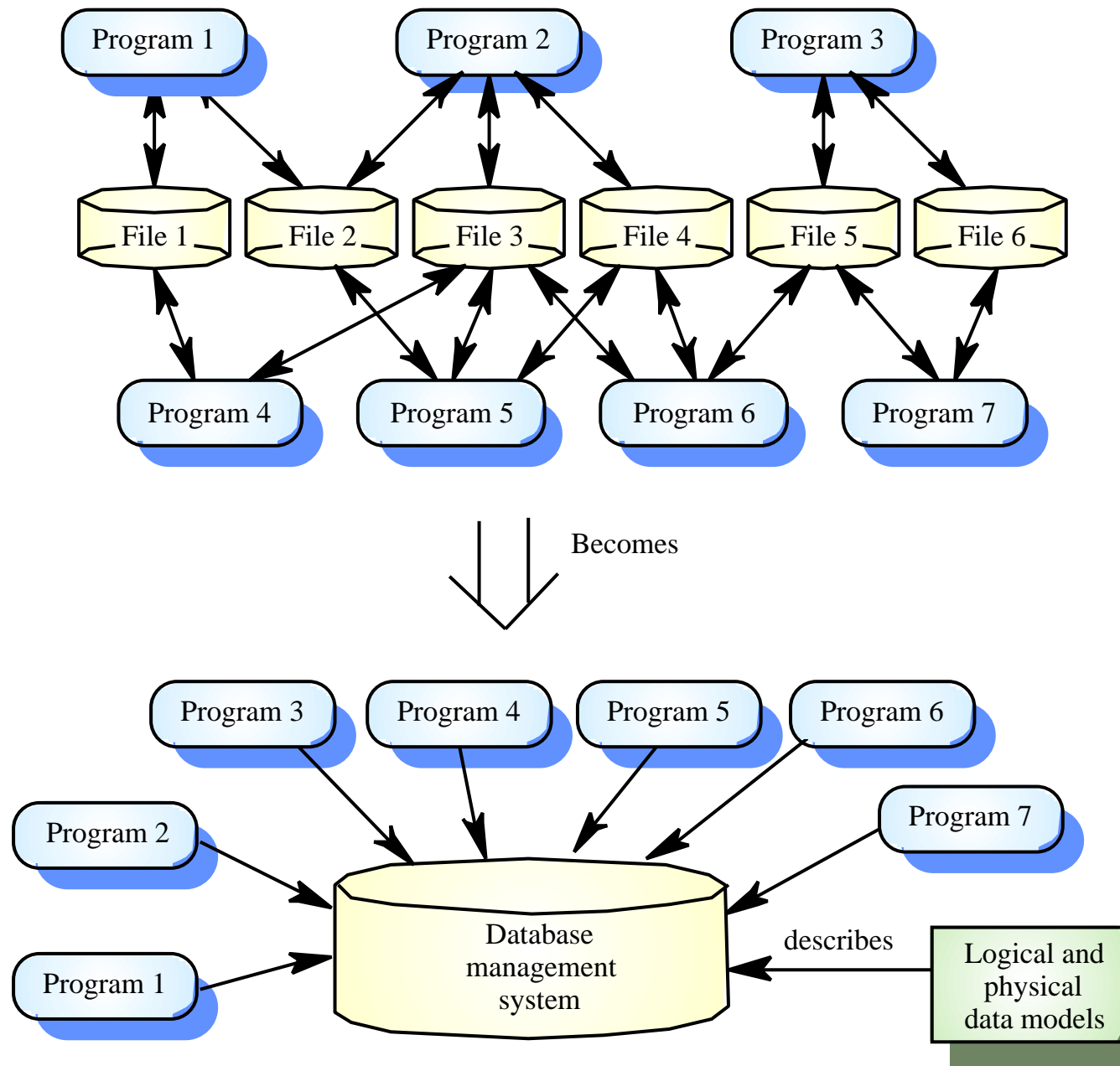
- Involves analysing and reorganising the data structures (and sometimes the data values) in a program
- May be part of the process of migrating from a file-based system to a DBMS-based system or changing from one DBMS to another
- Objective is to create a managed data environment

Approaches to data re-engineering

Approach	Description
Data cleanup	The data records and values are analysed to improve their quality. Duplicates are removed, redundant information is deleted and a consistent format applied to all records. This should not normally require any associated program changes.
Data extension	In this case, the data and associated programs are re-engineered to remove limits on the data processing. This may require changes to programs to increase field lengths, modify upper limits on the tables, etc. The data itself may then have to be rewritten and cleaned up to reflect the program changes.
Data migration	In this case, data is moved into the control of a modern database management system. The data may be stored in separate files or may be managed by an older type of DBMS.

Data problems

- End-users want data on their desktop machines rather than in a file system. They need to be able to download this data from a DBMS
- Systems may have to process much more data than was originally intended by their designers
- Redundant data may be stored in different formats in different places in the system



Data
migration

Data problems

- Data naming problems
 - Names may be hard to understand. The same data may have different names in different programs
- Field length problems
 - The same item may be assigned different lengths in different programs
- Record organisation problems
 - Records representing the same entity may be organised differently in different programs
- Hard-coded literals
- No data dictionary

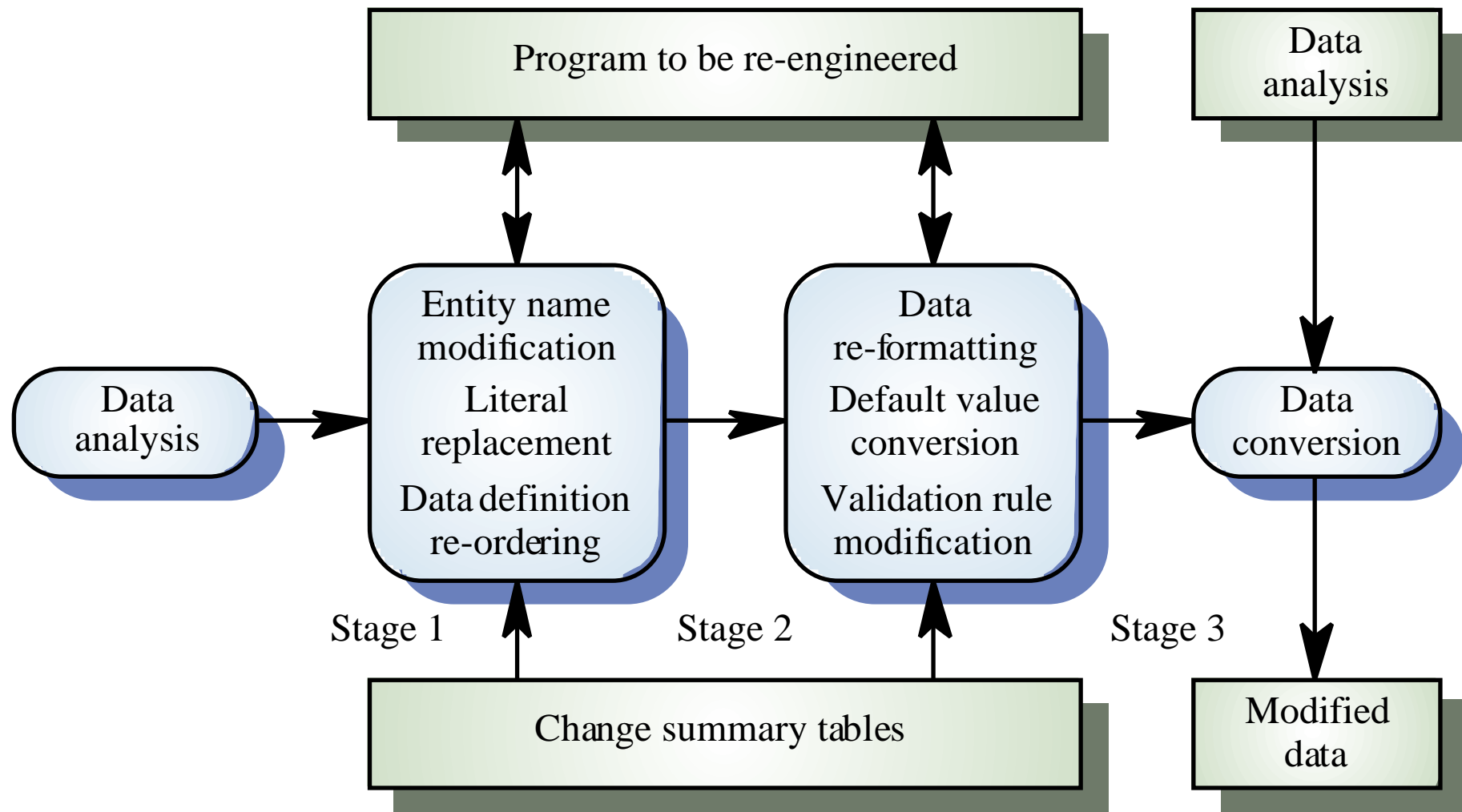
Data value inconsistencies

Data inconsistency	Description
Inconsistent default values	Different programs assign different default values to the same logical data items. This causes problems for programs other than those that created the data. The problem is compounded when missing values are assigned a default value that is valid. The missing data cannot then be discovered.
Inconsistent units	The same information is represented in different units in different programs. For example, in the US or the UK, weight data may be represented in pounds in older programs but in kilograms in more recent systems. A major problem of this type has arisen in Europe with the introduction of a single European currency. Legacy systems have been written to deal with national currency units and data has to be converted to euros.
Inconsistent validation rules	Different programs apply different data validation rules. Data written by one program may be rejected by another. This is a particular problem for archival data which may not have been updated in line with changes to data validation rules.
Inconsistent representation semantics	Programs assume some meaning in the way items are represented. For example, some programs may assume that upper-case text means an address. Programs may use different conventions and may therefore reject data which is semantically valid.
Inconsistent handling of negative values	Some programs reject negative values for entities which must always be positive. Others, however, may accept these as negative values or fail to recognise them as negative and convert them to a positive value.

Data conversion

- Data re-engineering may involve changing the data structure organisation without changing the data values
- Data value conversion is very expensive. Special-purpose programs have to be written to carry out the conversion

The data re-engineering process



Key points

- The objective of re-engineering is to improve the system structure to make it easier to understand and maintain
- The re-engineering process involves source code translation, reverse engineering, program structure improvement, program modularisation and data re-engineering
- Source code translation is the automatic conversion of of program in one language to another

Key points

- Reverse engineering is the process of deriving the system design and specification from its source code
- Program structure improvement replaces unstructured control constructs with while loops and simple conditionals
- Program modularisation involves reorganisation to group related items
- Data re-engineering may be necessary because of inconsistent data management

Configuration management

- Managing the products of system change

Objectives

- To explain the importance of software configuration management (CM)
- To describe key CM activities namely CM planning, change management, version management and system building
- To discuss the use of CASE tools to support configuration management processes

Topics covered

- Configuration management planning
- Change management
- Version and release management
- System building
- CASE tools for configuration management

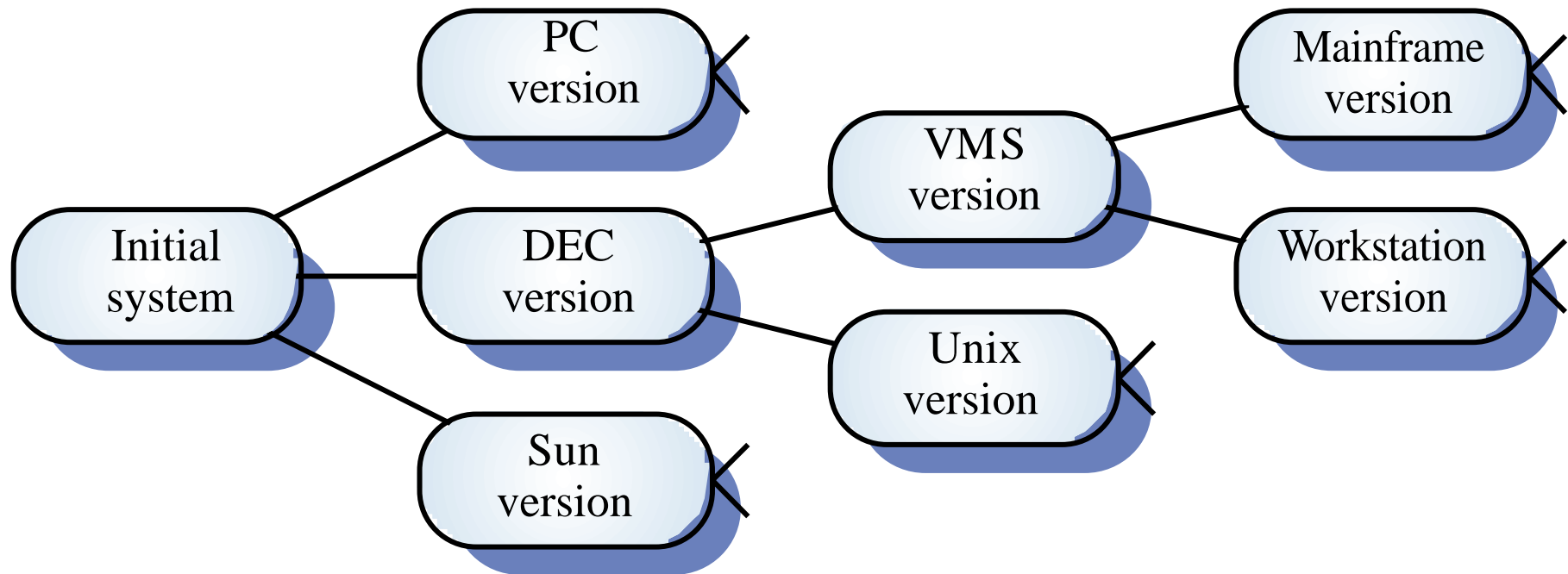
Configuration management

- New versions of software systems are created as they change
 - For different machines/OS
 - Offering different functionality
 - Tailored for particular user requirements
- Configuration management is concerned with managing evolving software systems
 - System change is a team activity
 - CM aims to control the costs and effort involved in making changes to a system

Configuration management

- Involves the development and application of procedures and standards to manage an evolving software product
- May be seen as part of a more general quality management process
- When released to CM, software systems are sometimes called *baselines* as they are a starting point for further development

System families



CM standards

- CM should always be based on a set of standards which are applied within an organisation
- Standards should define how items are identified, how changes are controlled and how new versions are managed
- Standards may be based on external CM standards (e.g. IEEE standard for CM)
- Existing standards are based on a waterfall process model - new standards are needed for evolutionary development

Concurrent development and testing

- A time for delivery of system components is agreed
- A new version of a system is built from these components by compiling and linking them
- This new version is delivered for testing using pre-defined tests
- Faults that are discovered during testing are documented and returned to the system developers

Daily system building

- It is easier to find problems that stem from component interactions early in the process
- This encourages thorough unit testing - developers are under pressure not to ‘break the build’
- A stringent change management process is required to keep track of problems that have been discovered and repaired

Configuration management planning

- All products of the software process may have to be managed
 - Specifications
 - Designs
 - Programs
 - Test data
 - User manuals
- Thousands of separate documents are generated for a large software system

CM planning

- Starts during the early phases of the project
- Must define the documents or document classes which are to be managed (Formal documents)
- Documents which might be required for future system maintenance should be identified and specified as managed documents

The CM plan

- Defines the types of documents to be managed and a document naming scheme
- Defines who takes responsibility for the CM procedures and creation of baselines
- Defines policies for change control and version management
- Defines the CM records which must be maintained

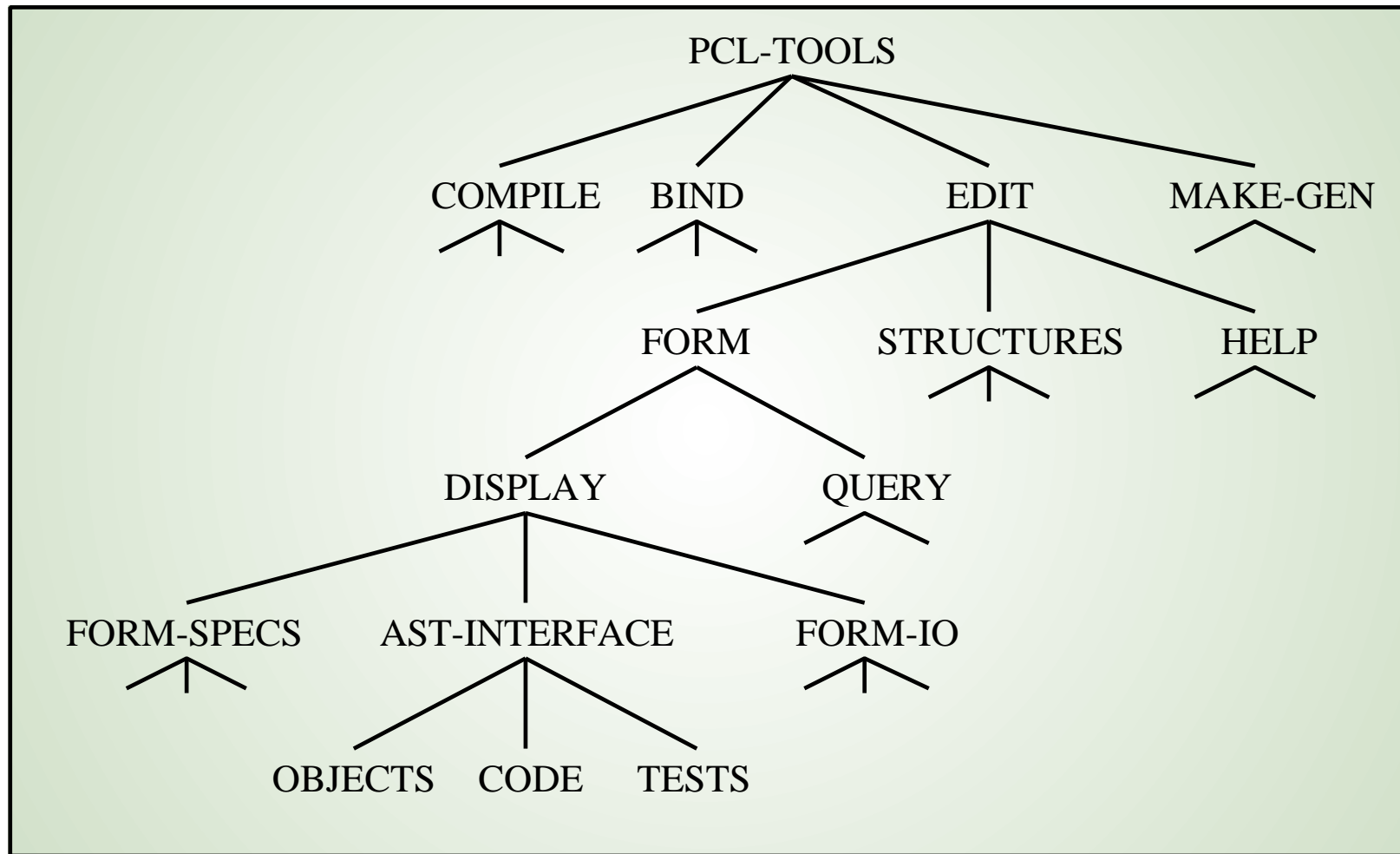
The CM plan

- Describes the tools which should be used to assist the CM process and any limitations on their use
- Defines the process of tool use
- Defines the CM database used to record configuration information
- May include information such as the CM of external software, process auditing, etc.

Configuration item identification

- Large projects typically produce thousands of documents which must be uniquely identified
- Some of these documents must be maintained for the lifetime of the software
- Document naming scheme should be defined so that related documents have related names.
- A hierarchical scheme with multi-level names is probably the most flexible approach

Configuration hierarchy



The configuration database

- All CM information should be maintained in a configuration database
- This should allow queries about configurations to be answered
 - Who has a particular system version?
 - What platform is required for a particular version?
 - What versions are affected by a change to component X?
 - How many reported faults in version T?
- The CM database should preferably be linked to the software being managed

CM database implementation

- May be part of an integrated environment to support software development. The CM database and the managed documents are all maintained on the same system
- CASE tools may be integrated with this so that there is a close relationship between the CASE tools and the CM tools
- More commonly, the CM database is maintained separately as this is cheaper and more flexible

Change management

- Software systems are subject to continual change requests
 - From users
 - From developers
 - From market forces
- Change management is concerned with keeping managing of these changes and ensuring that they are implemented in the most cost-effective way

The change management process

Request change by completing a change request form

Analyze change request

if change is valid **then**

 Assess how change might be implemented

 Assess change cost

 Submit request to change control board

if change is accepted **then**

repeat

 make changes to software

 submit changed software for quality approval

until software quality is adequate

 create new system version

else

 reject change request

else

 reject change request

Change request form

- Definition of change request form is part of the CM planning process
- Records change required, suggestor of change, reason why change was suggested and urgency of change(from requestor of the change)
- Records change evaluation, impact analysis, change cost and recommendations (System maintenance staff)

Change request form

Change Request Form	
Project: Proteus/PCL-Tools	Number: 23/94
Change requester: I. Sommerville	Date: 1/12/98
Requested change: When a component is selected from the structure, display the name of the file where it is stored.	
Change analyser: G. Dean	Analysis date: 10/12/98
Components affected: Display-Icon.Select, Display-Icon.Display	
Associated components: FileTable	
Change assessment: Relatively simple to implement as a file name table is available. Requires the design and implementation of a display field. No changes to associated components are required.	
Change priority: Low	
Change implementation:	
Estimated effort: 0.5 days	
Date to CCB: 15/12/98	CCB decision date: 1/2/99
CCB decision: Accept change. Change to be implemented in Release 2.1.	
Change implementor:	Date of change:
Date submitted to QA:	QA decision:
Date submitted to CM:	
Comments	

Change tracking tools

- A major problem in change management is tracking change status
- Change tracking tools keep track the status of each change request and automatically ensure that change requests are sent to the right people at the right time.
- Integrated with E-mail systems allowing electronic change request distribution

Change control board

- Changes should be reviewed by an external group who decide whether or not they are cost-effective from a strategic and organizational viewpoint rather than a technical viewpoint
- Should be independent of project responsible for system. The group is sometimes called a change control board
- May include representatives from client and contractor staff

Derivation history

- Record of changes applied to a document or code component
- Should record, in outline, the change made, the rationale for the change, who made the change and when it was implemented
- May be included as a comment in code. If a standard prologue style is used for the derivation history, tools can process this automatically

Component header information

```
// PROTEUS project (ESPRIT 6087)
//
// PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE
//
// Object: PCL-Tool-Desc
// Author: G. Dean
// Creation date: 10th November 1998
//
// © Lancaster University 1998
//
// Modification history
// Version      Modifier Date      Change      Reason
// 1.0          J. Jones  1/12/1998   Add header  Submitted to CM
// 1.1          G. Dean   9/4/1999   New field   Change req. R07/99
```

Version and release management

- Invent identification scheme for system versions
- Plan when new system version is to be produced
- Ensure that version management procedures and tools are properly applied
- Plan and distribute new system releases

Versions/variants/releases

- *Version* An instance of a system which is functionally distinct in some way from other system instances
- *Variant* An instance of a system which is functionally identical but non-functionally distinct from other instances of a system
- *Release* An instance of a system which is distributed to users outside of the development team

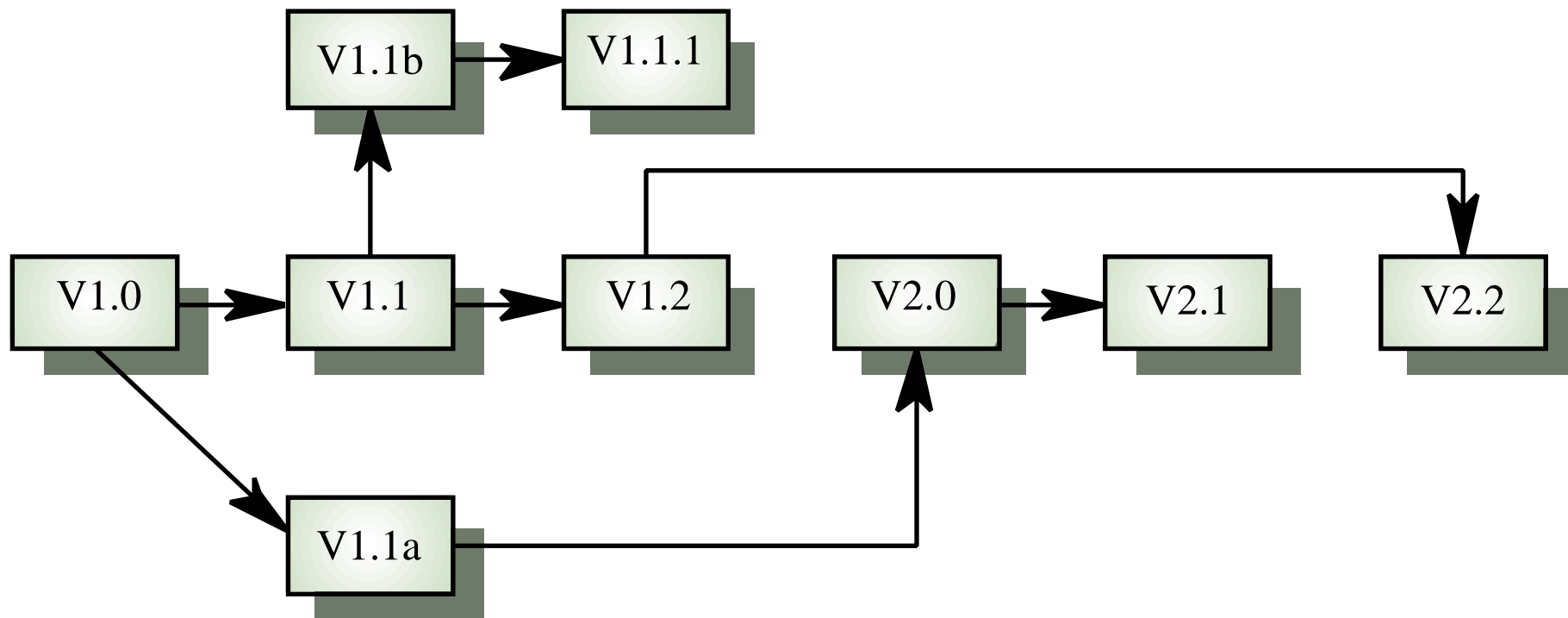
Version identification

- Procedures for version identification should define an unambiguous way of identifying component versions
- Three basic techniques for component identification
 - Version numbering
 - Attribute-based identification
 - Change-oriented identification

Version numbering

- Simple naming scheme uses a linear derivation e.g. V1, V1.1, V1.2, V2.1, V2.2 etc.
- Actual derivation structure is a tree or a network rather than a sequence
- Names are not meaningful.
- Hierarchical naming scheme may be better

Version derivation structure



Attribute-based identification

- Attributes can be associated with a version with the combination of attributes identifying that version
- Examples of attributes are Date, Creator, Programming Language, Customer, Status etc.
- More flexible than an explicit naming scheme for version retrieval; Can cause problems with uniqueness
- Needs an associated name for easy reference

Attribute-based queries

- An important advantage of attribute-based identification is that it can support queries so that you can find ‘the most recent version in Java’ etc.
- Example
 - AC3D (language =Java, platform = NT4, date = Jan 1999)

Change-oriented identification

- Integrates versions and the changes made to create these versions
- Used for systems rather than components
- Each proposed change has a change set that describes changes made to implement that change
- Change sets are applied in sequence so that, in principle, a version of the system that incorporates an arbitrary set of changes may be created

Release management

- Releases must incorporate changes forced on the system by errors discovered by users and by hardware changes
- They must also incorporate new system functionality
- Release planning is concerned with when to issue a system version as a release

System releases

- Not just a set of executable programs
- May also include
 - Configuration files defining how the release is configured for a particular installation
 - Data files needed for system operation
 - An installation program or shell script to install the system on target hardware
 - Electronic and paper documentation
 - Packaging and associated publicity
- Systems are now normally released on CD-ROM or as downloadable installation files from the web

Release problems

- Customer may not want a new release of the system
 - They may be happy with their current system as the new version may provide unwanted functionality
- Release management must not assume that all previous releases have been accepted. All files required for a release should be re-created when a new release is installed

Release decision making

- Preparing and distributing a system release is an expensive process
- Factors such as the technical quality of the system, competition, marketing requirements and customer change requests should all influence the decision of when to issue a new system release

System release strategy

Factor	Description
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. However, minor system faults may be repaired by issuing patches (often distributed over the Internet) that can be applied to the current release of the system.
Lehman's fifth law (see Chapter 27)	This suggests that the increment of functionality which is included in each release is approximately constant. Therefore, if there has been a system release with significant new functionality, then it may have to be followed by a repair release.
Competition	A new system release may be necessary because a competing product is available.
Marketing requirements	The marketing department of an organisation may have made a commitment for releases to be available at a particular date.
Customer change proposals	For customised systems, customers may have made and paid for a specific set of system change proposals and they expect a system release as soon as these have been implemented.

Release creation

- Release creation involves collecting all files and documentation required to create a system release
- Configuration descriptions have to be written for different hardware and installation scripts have to be written
- The specific release must be documented to record exactly what files were used to create it. This allows it to be re-created if necessary

System building

- The process of compiling and linking software components into an executable system
- Different systems are built from different combinations of components
- Invariably supported by automated tools that are driven by ‘build scripts’

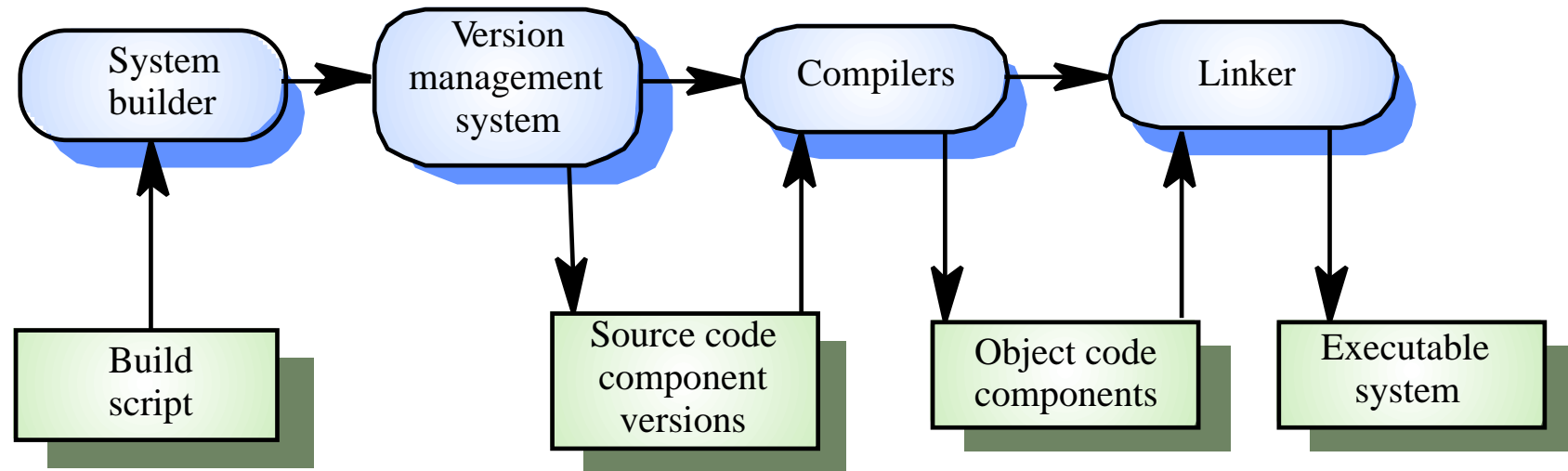
System building problems

- Do the build instructions include all required components?
 - When there are many hundreds of components making up a system, it is easy to miss one out. This should normally be detected by the linker
- Is the appropriate component version specified?
 - A more significant problem. A system built with the wrong version may work initially but fail after delivery
- Are all data files available?
 - The build should not rely on 'standard' data files. Standards vary from place to place

System building problems

- Are data file references within components correct?
 - Embedding absolute names in code almost always causes problems as naming conventions differ from place to place
- Is the system being built for the right platform
 - Sometimes must build for a specific OS version or hardware configuration
- Is the right version of the compiler and other software tools specified?
 - Different compiler versions may actually generate different code and the compiled component will exhibit different behaviour

System building



System representation

- Systems are normally represented for building by specifying the file name to be processed by building tools. Dependencies between these are described to the building tools
- Mistakes can be made as users lose track of which objects are stored in which files
- A system modelling language addresses this problem by using a logical rather than a physical system representation

CASE tools for configuration management

- CM processes are standardised and involve applying pre-defined procedures
- Large amounts of data must be managed
- CASE tool support for CM is therefore essential
- Mature CASE tools to support configuration management are available ranging from stand-alone tools to integrated CM workbenches

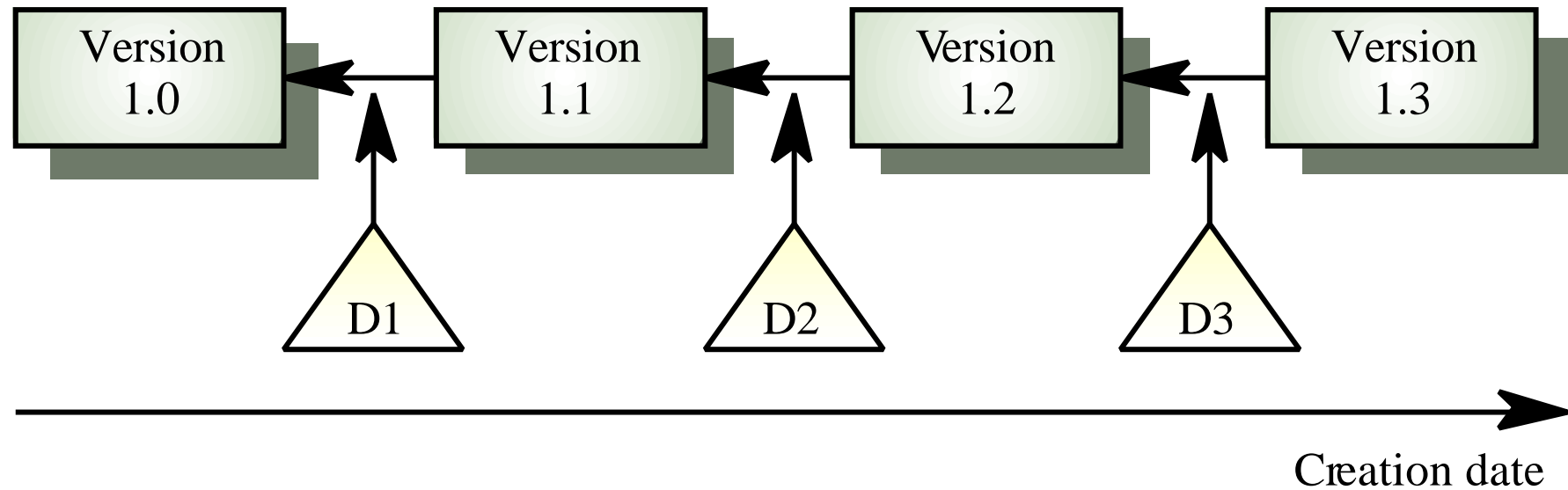
Change management tools

- Change management is a procedural process so it can be modelled and integrated with a version management system
- Change management tools
 - Form editor to support processing the change request forms
 - Workflow system to define who does what and to automate information transfer
 - Change database that manages change proposals and is linked to a VM system

Version management tools

- Version and release identification
 - Systems assign identifiers automatically when a new version is submitted to the system
- Storage management.
 - System stores the differences between versions rather than all the version code
- Change history recording
 - Record reasons for version creation
- Independent development
 - Only one version at a time may be checked out for change.
Parallel working on different versions

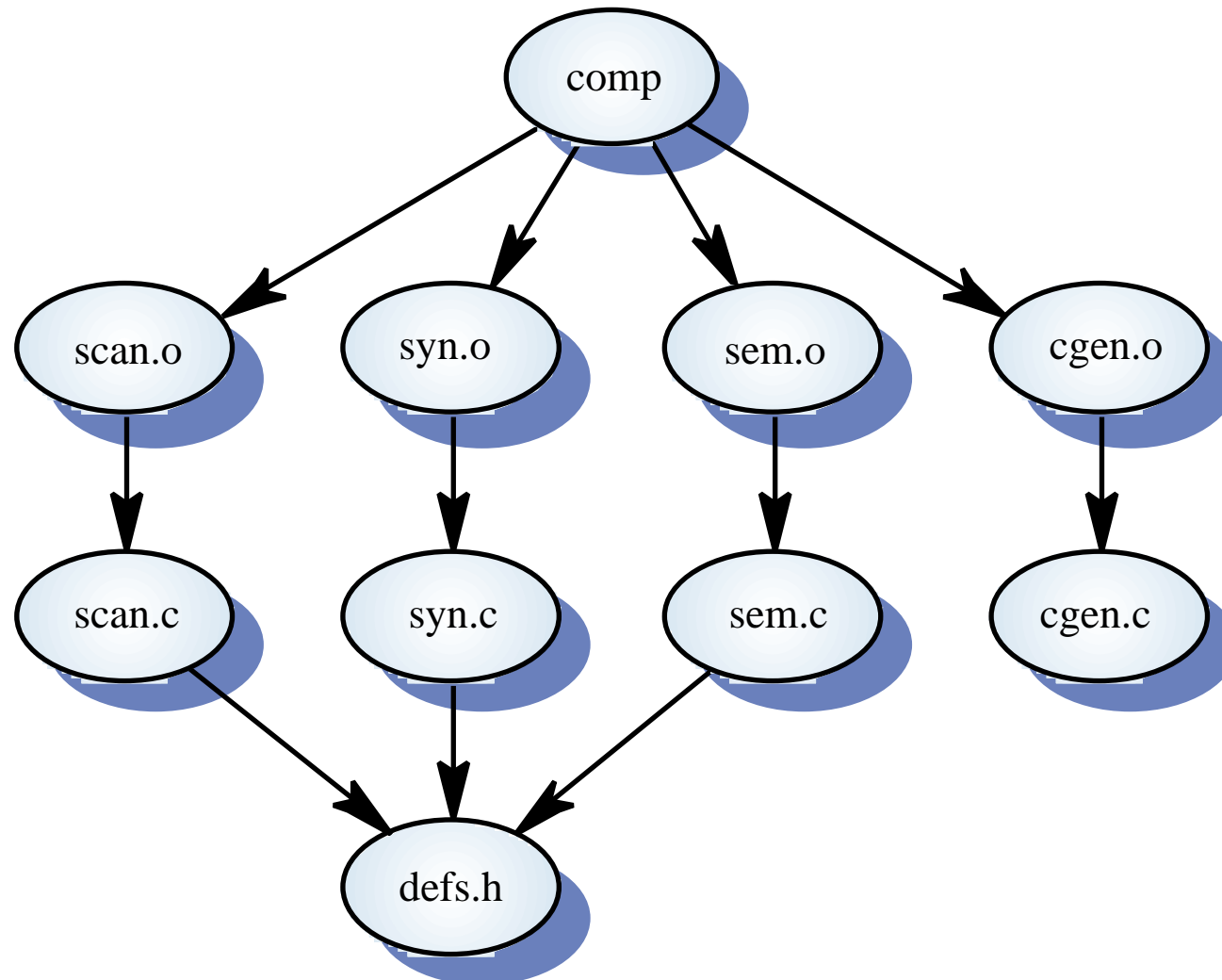
Delta-based versioning



System building

- Building a large system is computationally expensive and may take several hours
- Hundreds of files may be involved
- System building tools may provide
 - A dependency specification language and interpreter
 - Tool selection and instantiation support
 - Distributed compilation
 - Derived object management

Component dependencies



Key points

- Configuration management is the management of system change to software products
- A formal document naming scheme should be established and documents should be managed in a database
- The configuration data base should record information about changes and change requests
- A consistent scheme of version identification should be established using version numbers, attributes or change sets

Key points

- System releases include executable code, data, configuration files and documentation
- System building involves assembling components into a system
- CASE tools are available to support all CM activities
- CASE tools may be stand-alone tools or may be integrated systems which integrate support for version management, system building and change management