《软件工程（双语）》

参考教材：

《Software engineering》9th Edition Ian Sommervile， Pearson Education, 机械工业出版社，2011

参考书目：

1、Software Engineering Theory and Practice(Second Edition 影印版), Shari Lawrence Pfleeger，Pearson Education, 2001

2、《软件工程》第四版 张海藩 清华大学出版社，2007

3、软件工程， 王忠群主编 中国科学技术大学出版社 2009-11-1 4、Software engineering : a practitioner's approach / Roger S. Pressman. 6th ed. Pressman, Roger S. China Machine Press, 2008

说明：

斜体部分是可选讲授内容, 带星号的习题为可选。

Chapter 1 (1) Introduction

- Getting started with software engineering

1.1 Topics covered

A.1 Professional software development

What is meant by software engineering

A.2 Software engineering ethics

A brief introduction to ethical issues that affect software engineering.

1.2 Importance of Software engineering

- The economies of ALL developed nations are dependent on software.
- More and more systems are software controlled
- Expenditure on software represents a significant fraction of GNP (gross National product) in all developed countries.(GNP 与 GDP 的关系是：GNP 等于 GDP 加上本国投在国外的资本和劳务的收入再减去外国投在本国的资本和劳务的收入。)
- Software costs often dominate computer system costs. The costs of software on a PC are often greater than the hardware cost.
- Software engineering is concerned with cost-effective software development

1.3 FAQs about software engineering

A.1 What is software?

Computer programs and associated documentation

B.1 Software products may be developed for a particular customer or may be developed for a general market

B.2 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

A.2 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

B.1 Maintainability

Software must evolve to meet changing needs

B.2 Dependability

Software must be trustworthy

B.3 Efficiency

Software should not make wasteful use of system resources

B.4 Acceptability

Software must accepted by the users for which it was designed. This means it must be understandable, usable and compatible with other systems.

A.3 What is software engineering?

Software engineering is an engineering discipline that is concerned with all aspects of software production.

A.4 What are the fundamental software engineering activities?

Software specification, software development, software validation and software evolution

A.5 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

A.6 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 more general process.

A.7 What are the key challenges facing software engineering?

B.1 Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software

B.2 Heterogeneity

- Developing techniques for building software that can cope with heterogeneous platforms and execution environments;

B.3 Delivery

- Developing techniques that lead to faster delivery of software;

B.4 Trust

- Developing techniques that demonstrate that software can be trusted by its users

A.8 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

A.9 What are the best software engineering techniques and methods?

While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. You can't, therefore, say that one method is better than another.

A.10 What differences has the web made to software engineering?

The web has led to the availability of software services and the possibility of developing highly distributed service-based systems. Web-based systems development has led to important advances in programming languages and software reuse.

1.4 Software engineering

1.4.1 Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.

A.1 Engineering discipline

Using appropriate theories and methods to solve problems bearing in mind organizational and financial constraints.

A.2 All aspects of software production

Not just technical process of development. Also project management and the development of tools, methods etc. to support software production.

1.4.2 General issues that affect most software

A.1 Heterogeneity

Increasingly, systems are required to operate as distributed systems across networks that include different types of computer and mobile devices.

A.2 Business and social change

Business and society are changing incredibly quickly as emerging economies develop and new technologies become available. They need to be able to change their existing software and to

rapidly develop new software.

A.3 Security and trust

As software is intertwined with all aspects of our lives, it is essential that we can trust that software.

1.4.3 Software engineering diversity

B.1 here are many different types of software system and there is no universal set of software techniques that is applicable to all of these.

B.2 The software engineering methods and tools used depend on the type of application being developed, the requirements of the customer and the background of the development team.

A.2 Application types

B.1 Stand-alone applications

These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network.

B.2 Interactive transaction-based applications

Applications that execute on a remote computer and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications.

B.3 Embedded control systems

These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system.

B.4 Batch processing systems

These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs.

B.5 Entertainment systems

These are systems that are primarily for personal use and which are intended to entertain the user.

B.6 Systems for modeling and simulation

These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects.

B.7 Data collection systems

These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing.

B.8 Systems of systems

These are systems that are composed of a number of other software systems.

1.4.4 Software engineering fundamentals

A.1 Some fundamental principles apply to all types of software system, irrespective of the development techniques used:

B.1 Systems should be developed using a managed and understood development process. Of course, different processes are used for different types of software.

B.2 Dependability and performance are important for all types of system.

B.3 Understanding and managing the software specification and requirements (what the software should do) are important.

B.4 Where appropriate, you should reuse software that has already been developed rather than write new software.

1.4.5 Software engineering and the web

B.1 The Web is now a platform for running application and organizations are increasingly developing web-based systems rather than local systems.

B.2 Web services (discussed in Chapter 19) allow application functionality to be accessed over the web.

B.3 Cloud computing is an approach to the provision of computer services where applications run remotely on the 'cloud'.

C.1 Users do not buy software buy pay according to use.

B.4 Software reuse is the dominant approach for constructing web-based systems.

When building these systems, you think about how you can assemble them from pre-existing software components and systems.

B.5 Web-based systems should be developed and delivered incrementally.

It is now generally recognized that it is impractical to specify all the requirements for such systems in advance.

B.6 User interfaces are constrained by the capabilities of web browsers.

Technologies such as AJAX allow rich interfaces to be created within a web browser but are still difficult to use. Web forms with local scripting are more commonly used.

Ajax 包括：

- XHTML 和 CSS
- 使用文档对象模型(Document Object Model)作动态显示和交互
- 使用 XML 和 XSLT 做数据交互和操作
- 使用 XMLHttpRequest 进行异步数据接收
- 使用 JavaScript 将它们绑定在一起

A.2 Web-based software engineering

B.1 Web-based systems are complex distributed systems but the fundamental principles of software engineering discussed previously are as applicable to them as they are to any other types of system.

B.2 The fundamental ideas of software engineering, discussed in the previous section, apply to web-based software in the same way that they apply to other types of software system

1.5 Software engineering ethics

1.5.1 Concept

- 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 but involves following a set of principles that are morally correct.

1.5.2 Issues of professional responsibility

A.1 Confidentiality

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

A.2 Competence

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

A.3 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.

A.4 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).

1.5.3 ACM/IEEE Code of Ethics

(Association for Computing Machinery/Institute of Electrical and Electronics Engineers)

A.1 Concept

- 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.

A.2 Ethical principles

C.1 PUBLIC

Software engineers shall act consistently with the public interest.

C.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.

C.3 PRODUCT

Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

C.4 JUDGMENT

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

C.5 MANAGEMENT

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

C.6 PROFESSION

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

C.7 COLLEAGUES

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

C.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.

A.3 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

1.6 Key points

A.1 Software engineering is an engineering discipline that is concerned with all aspects of software production.

A.2 Essential software product attributes are maintainability, dependability and security, efficiency and acceptability.

A.3 The high-level activities of specification, development, validation and evolution are part of all software processes.

A.4 The fundamental notions of software engineering are universally applicable to all types of system development.

A.5 There are many different types of system and each requires appropriate software engineering tools and techniques for their development.

A.6 The fundamental ideas of software engineering are applicable to all types of software system.

1.7 Excercises(Homework): P25

1.3.

Chapter 2 (2) Software processes

2.1 Topics covered

1. Software process models
2. Process activities
3. Coping with change

2.2 The software process

2.2.1 Introduction

A.1 A structured set of activities required to develop a software system.

B.1 Many different software processes but all involve:

1. Specification – defining what the system should do;
2. Design and implementation – defining the organization of the system and implementing the system;
3. Validation – checking that it does what the customer wants;
4. Evolution – changing the system in response to changing customer needs.

A.2 A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.

2.2.2 Plan-driven and agile processes

1. Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
2. In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
3. In practice, most practical processes include elements of both plan-driven and agile approaches.
4. There are no right or wrong software processes.

2.3 Software process models

2.3.1 Generic software process models

A.1 The waterfall model

Plan-driven model. Separate and distinct phases of specification and development.

A.2 Incremental development

Specification, development and validation are interleaved. May be plan-driven or agile.

A.3 Reuse-oriented software engineering

The system is assembled from existing components. May be plan-driven or agile.

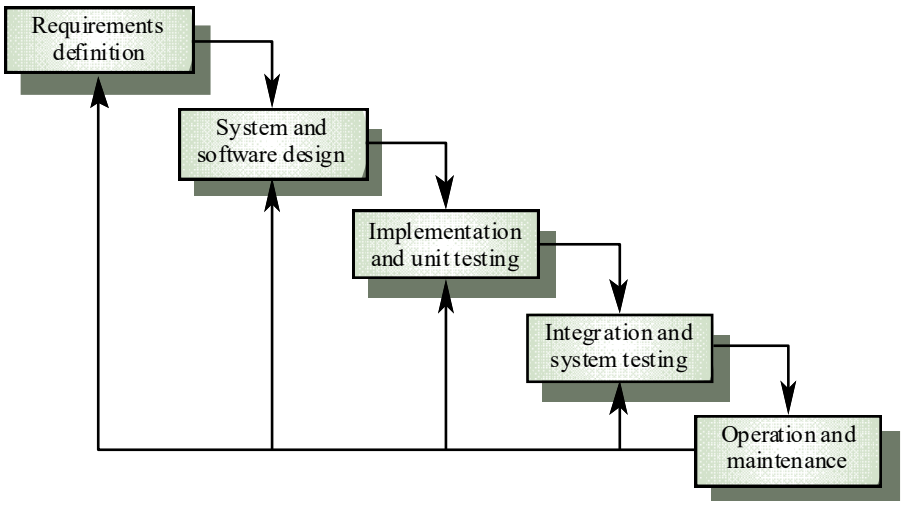
A.4 In practice, most large systems are developed using a process that incorporates elements from all of these models.

2.3.2 Waterfall model

B.1 Waterfall model phases

1. Requirements analysis and definition
2. System and software design
3. Implementation and unit testing
4. Integration and system testing
5. Operation and maintenance

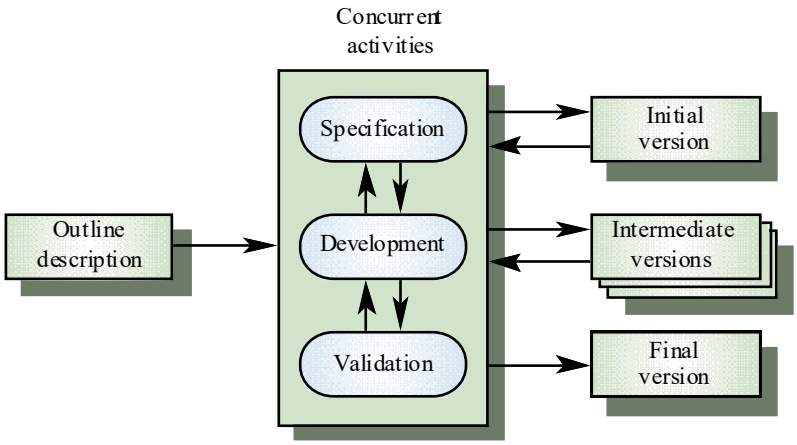
Waterfall model



B.2 Waterfall model problems

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

2.3.3 Incremental development



A.1 Incremental development benefits

B.1 The cost of accommodating changing customer requirements is reduced.

The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.

B.2 It is easier to get customer feedback on the development work that has been done.

Customers can comment on demonstrations of the software and see how much has been implemented.

B.3 More rapid delivery and deployment of useful software to the customer is possible.

Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

A.2 Incremental development problems

B.1 The process is not visible.

Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.

B.2 System structure tends to degrade as new increments are added.

Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

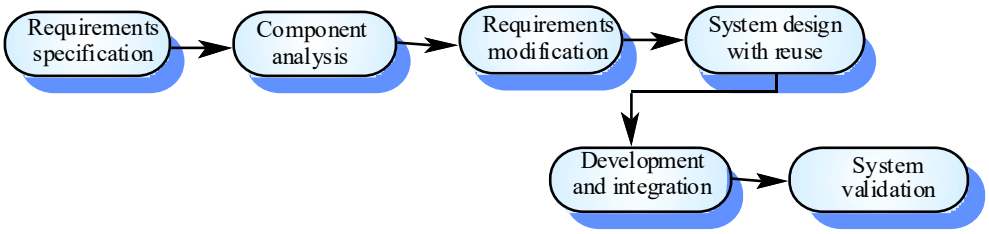
2.3.4 Reuse-oriented software engineering

A.1 Based on systematic reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems.

A.2 Process stages

1. Component analysis;
2. Requirements modification;
3. System design with reuse;
4. Development and integration.

B.1 Reuse is now the standard approach for building many types of business system



2.3.5 Types of software component

1. Web services that are developed according to service standards and which are available for remote invocation.
2. Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
3. Stand-alone software systems (COTS) that are configured for use in a particular environment.

2.4 Process activities

2.4.1 Software specification

The process of establishing what services are required and the constraints on the system's operation and development.

A.1 Requirements engineering process

B.1 Feasibility study

Is it technically and financially feasible to build the system?

B.2 Requirements elicitation and analysis

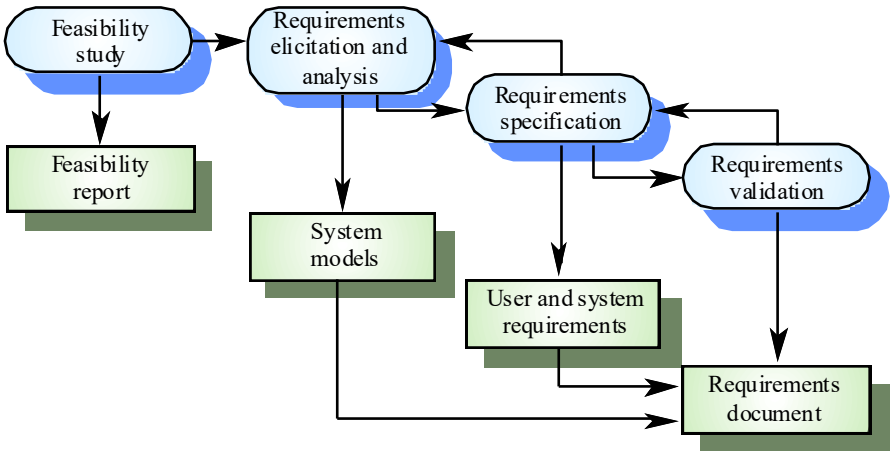
What do the system stakeholders require or expect from the system?

B.3 Requirements specification

Defining the requirements in detail

B.4 Requirements validation

Checking the validity of the requirements



2.4.2 Software design and implementation

The process of converting the system specification into an executable system.

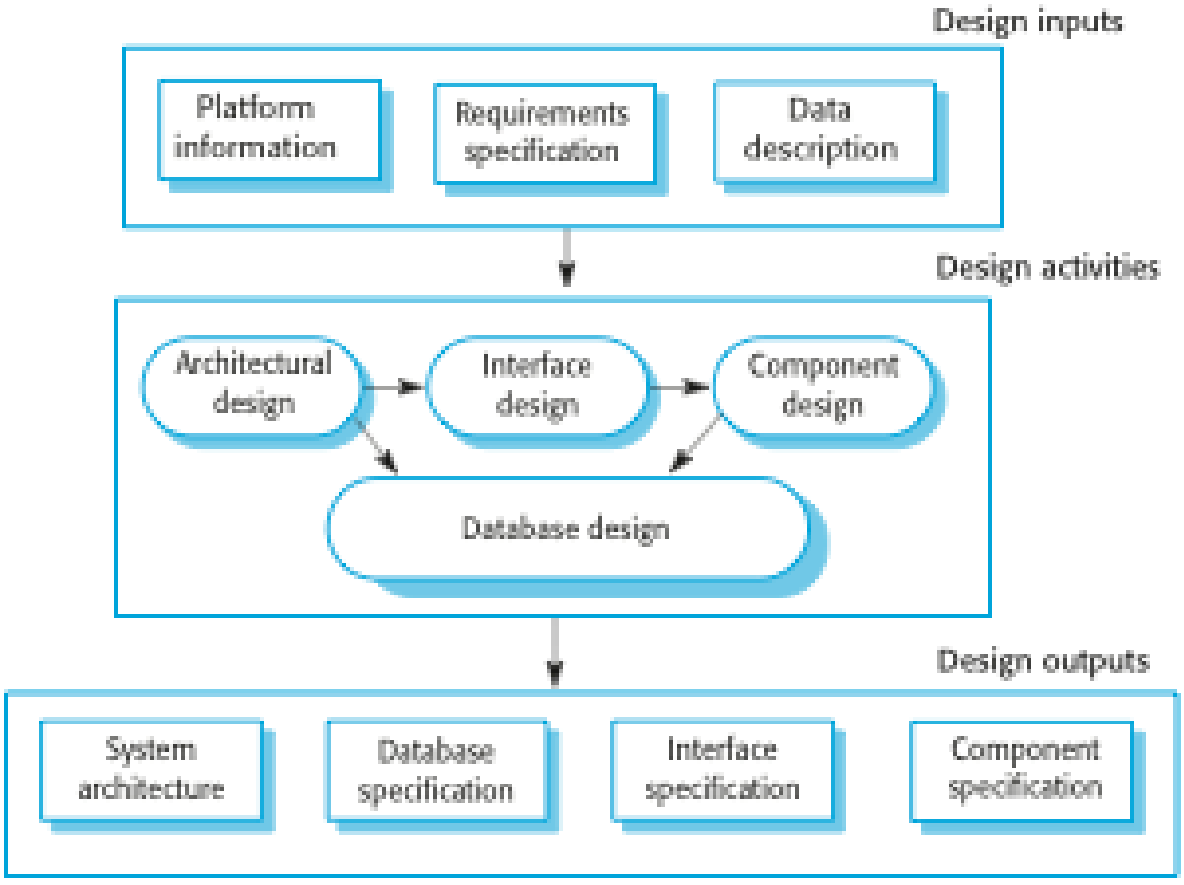
A.1 Software design

Design a software structure that realises the specification;

A.2 Implementation

Translate this structure into an executable program;
The activities of design and implementation are closely related and may be inter-leaved.

A.3 A general model of the design process



A.4 Design activities

B.1 Architectural design,
where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships and how they are distributed.

B.2 Interface design,
where you define the interfaces between system components.

B.3 Component design,
where you take each system component and design how it will operate.

B.4 Database design,

where you design the system data structures and how these are to be represented in a database.

2.4.3 Software validation

1. Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
2. Involves checking and review processes and system testing.
3. 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.
4. Testing is the most commonly used V & V activity.

A.1 Testing stages

B.1 Development or component testing

Individual components are tested independently;

Components may be functions or objects or coherent groupings of these entities.

B.2 System testing

Testing of the system as a whole. Testing of emergent properties is particularly important.

B.3 Acceptance testing

Testing with customer data to check that the system meets the customer's needs.

2.4.4 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.

2.5 Coping with change

2.5.1 Introduction

A.1 Change is inevitable in all large software projects.

1. Business changes lead to new and changed system requirements
2. New technologies open up new possibilities for improving implementations
3. Changing platforms require application changes

A.2 Change leads to rework so the costs of change include both rework (e.g. re-analysing requirements) as well as the costs of implementing new functionality

A.3 Reducing the costs of rework

B.1 Change avoidance

where the software process includes activities that can anticipate possible changes before

significant rework is required.

For example, a prototype system may be developed to show some key features of the system to customers.

B.2 Change tolerance

where the process is designed so that changes can be accommodated at relatively low cost.

This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have be altered to incorporate the change.

2.5.2 Software prototyping

A.1 Introduction

B.1 A prototype is an initial version of a system used to demonstrate concepts and try out design options.

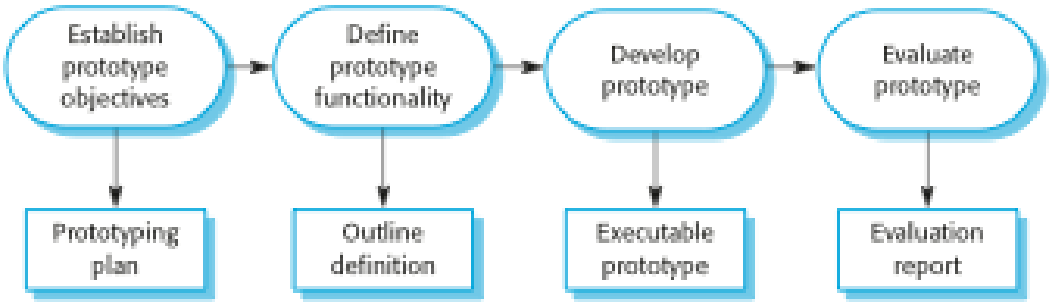
B.2 A prototype can be used in:

1. The requirements engineering process to help with requirements elicitation and validation;
2. In design processes to explore options and develop a UI design;
3. In the testing process to run back-to-back tests.

A.2 Benefits of prototyping

1. Improved system usability.
2. A closer match to users' real needs.
3. Improved design quality.
4. Improved maintainability.
5. Reduced development effort.

A.3 The process of prototype development



A.4 Prototype development

B.1 Introduction

C.1 May be based on rapid prototyping languages or tools

C.2 May involve leaving out functionality

1. Prototype should focus on areas of the product that are not well-understood;
2. Error checking and recovery may not be included in the prototype;

3. Focus on functional rather than non-functional requirements such as reliability and security

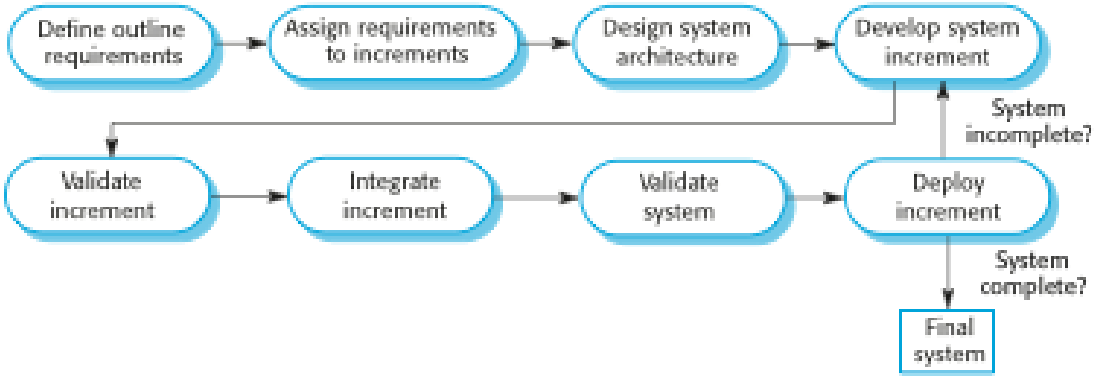
B.2 Throw-away prototypes

C.1 Prototypes should be discarded after development as they are not a good basis for a production system:

1. It may be impossible to tune the system to meet non-functional requirements;
2. Prototypes are normally undocumented;
3. The prototype structure is usually degraded through rapid change;
4. The prototype probably will not meet normal organisational quality standards.

2.5.3 Incremental delivery

1. 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.
2. User requirements are prioritised and the highest priority requirements are included in early increments.
3. Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.



B.1 Incremental delivery advantages

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

B.2 Incremental delivery problems

C.1 Most systems require a set of basic facilities that are used by different parts of the system.

As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.

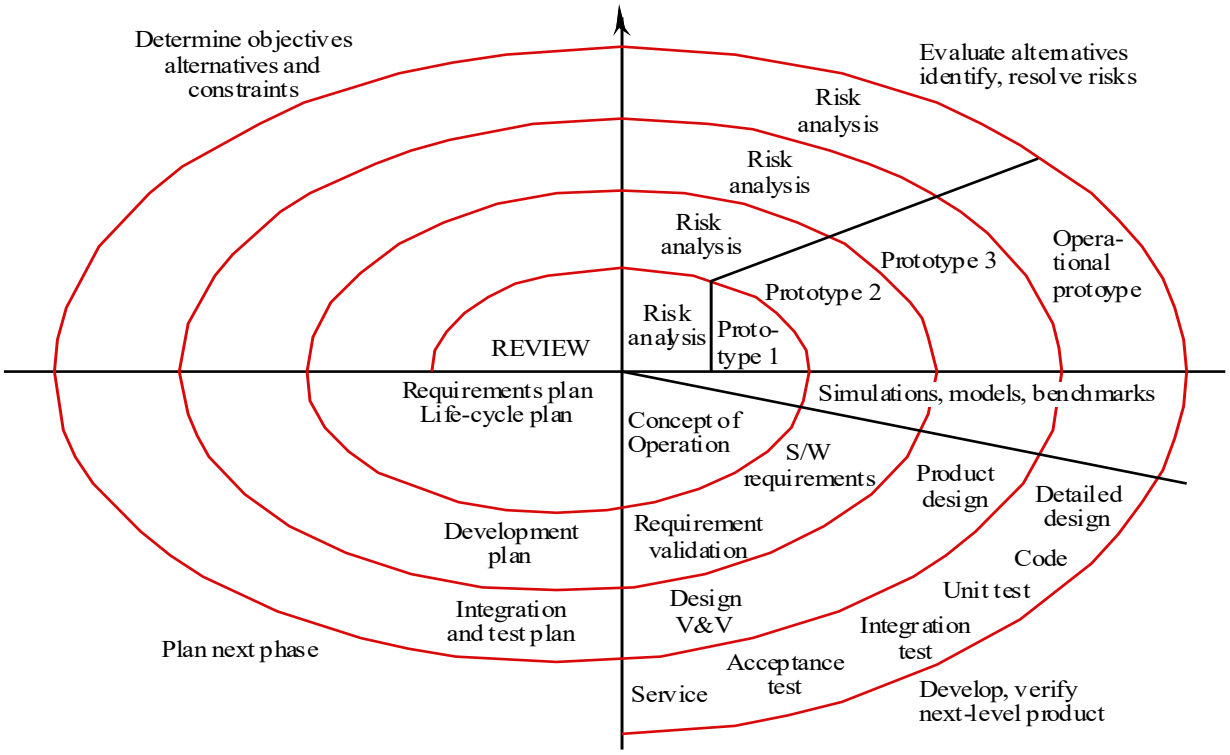
C.2 The essence of iterative processes is that the specification is developed in conjunction with the software.

However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract.

2.5.4 Boehm’s spiral model

A.1 Concept

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



A.2 Spiral model sectors

B.1 Objective setting

Specific objectives for the phase are identified.

B.2 Risk assessment and reduction

Risks are assessed and activities put in place to reduce the key risks.

B.3 Development and validation

A development model for the system is chosen which can be any of the generic models.

B.4 Planning

The project is reviewed and the next phase of the spiral is planned.

A.3 Spiral model usage

1. Spiral model has been very influential in helping people think about iteration in software processes and introducing the risk-driven approach to development.
2. In practice, however, the model is rarely used as published for practical software development

2.6 Key points

1. Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
2. General process models describe the organization of software processes. Examples of these general models include the ‘waterfall’ model， incremental development, and reuse-oriented development.
3. Requirements engineering is the process of developing a software specification.
4. Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
5. Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
6. Software evolution takes place when you change existing software systems to meet new requirements. The software must evolve to remain useful.
7. Processes should include activities to cope with change. This may involve a prototyping phase that helps avoid poor decisions on requirements and design.
8. Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.

2.7 Excercises(Homework): P54

2.1, *2.4

Chapter 3 (4) Requirements Engineering

3.1 Topics covered

1. Functional and non-functional requirements
2. The software requirements document
3. Requirements specification
4. Requirements engineering processes
5. Requirements elicitation and analysis
6. Requirements validation
7. Requirements management

3.2 Requirements engineering

3.2.1 What is a requirement?

B.1 It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification

B.2 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

3.2.2 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

3.2.3 Types of requirement

B.1 User requirements

Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers

B.2 System requirements

A structured document setting out detailed descriptions of the system services. Written as a contract between client and contractor

User requirement definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g. 10mg, 20 mg, etc.) separate reports shall be created for each dose unit.
1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

3.2.4 Functional and non-functional requirements

A.1 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.

B.1 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.

A.2 Non-functional requirements

constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.

B.1 Non-functional classifications

C.1 Product requirements

Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

C.2 Organisational requirements

Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.

C.3 External requirements

Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

B.2 Non-functional requirements examples

C.1 Product requirement

4.C.8 It shall be possible for all necessary communication between the APSE(Ada Programming Support Environment) and the user to be expressed in the standard Ada character set

C.2 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

C.3 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

B.3 Requirements measures

Example: 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.

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

A.3 Domain requirements

Requirements that come from the application domain of the system and that reflect characteristics of that domain

B.1 examples

C.1

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.

C.2

The deceleration of the train shall be computed as:

Dtrain = Dcontrol + Dgradient

where Dgradient is 9.81ms2 * compensated gradient/alpha and where the values of 9.81ms2 /alpha are known for different types of train.

3.3 The requirements document

3.3.1 Concept

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

3.3.2 Requirements document structure

1. Introduction
2. Glossary
3. User requirements definition
4. System architecture
5. System requirements specification
6. System models
7. System evolution
8. Appendices
9. Index

3.3.3 Ways of writing a system requirements specification

A.1 Natural language

The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.

A.2 Structured natural language

The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.

B.1 Form-based specifications

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading (r2); the previous two readings (r0 and r1).

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

Action

CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

Requirements

Two previous readings so that the rate of change of sugar level can be computed.

Pre-condition

The insulin reservoir contains at least the maximum allowed single dose of insulin.

Post-condition r0 is replaced by r1 then r1 is replaced by r2.

Side effects None.

B.2 Tabular specification

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 – r1) < (r1 – r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing ((r2 – r1) ≥ (r1 – r0))	CompDose = round ((r2 – r1)/4) If rounded result = 0 then CompDose = MinimumDose

A.3 Design description languages

This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.

```
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
```

A.4 Graphical notations

Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.

A.5 Mathematical specifications

These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document,

most customers don’t understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

3.4 Requirements engineering processes

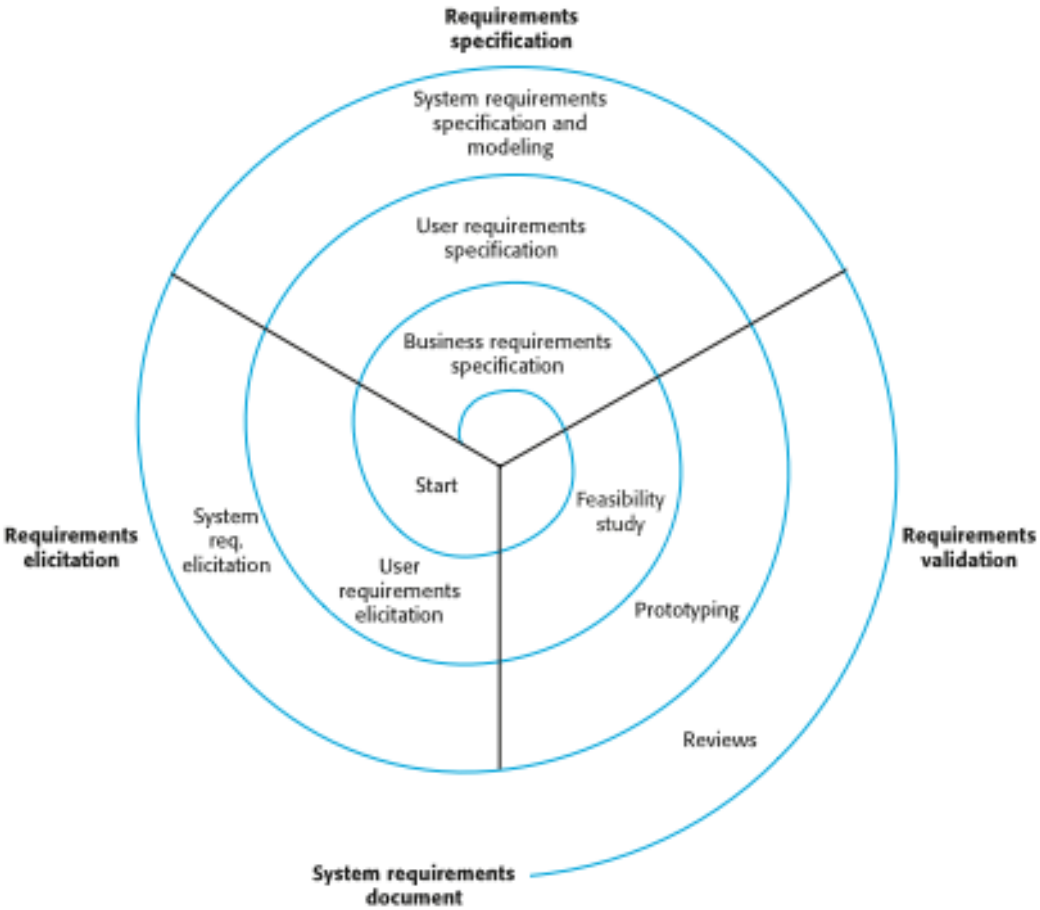
A.1 The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements

A.2 However, there are a number of generic activities common to all processes

1. Feasibility studies
2. Requirements elicitation and analysis
3. Requirements validation
4. Requirements management

A.3 In practice, RE is an iterative activity in which these processes are interleaved.

B.1 A spiral view of the requirements engineering process



3.4.2 Feasibility studies

B.1 A feasibility study decides whether or not the proposed system is worthwhile

B.2 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

3.4.3 Elicitation and analysis

A.1 Introduction

- 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*

A.2 Problems of requirements analysis

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

A.3 Process activities

B.1 Requirements discovery

Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.

B.2 Requirements classification and organisation

Groups related requirements and organises them into coherent clusters.

B.3 Prioritisation and negotiation

Prioritising requirements and resolving requirements conflicts.

B.4 Requirements documentation

Requirements are documented.

3.4.4 techniches of Requirements discovery

A.1 Interviewing

B.1 Formal or informal interviews with stakeholders are part of most RE processes.

B.2 Types of interview

1. Closed interviews based on pre-determined list of questions
2. Open interviews where various issues are explored with stakeholders.

B.3 Normally a mix of closed and open-ended interviewing.

B.4 Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.

A.2 Scenarios

B.1 Scenarios are real-life examples of how a system can be used.

B.2 They should include

- A description of the starting situation;
- A description of the normal flow of events;
- A description of what can go wrong;
- Information about other concurrent activities;
- A description of the state when the scenario finishes

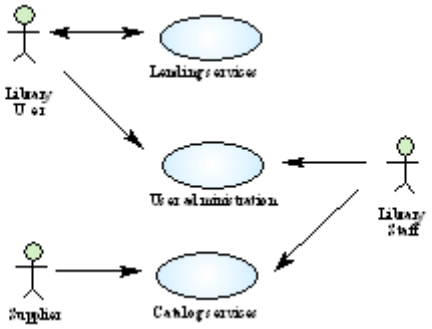
A.3 Use cases

B.1 Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself

B.2 A set of use cases should describe all possible interactions with the system

Lending use-case

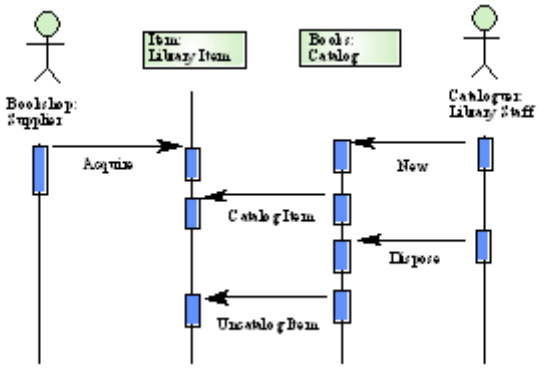
Library use-cases



B.3 High-level graphical model supplemented by more detailed tabular description (see Chapter 5).

B.4 Sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system

Catalogue management(Sequence diagram)



A.4 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

3.4.5 Requirements validation

A.1 Concerned with demonstrating that the requirements define the system that the customer really wants

A.2 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

3.4.6 Requirements change management

B.1 Requirements management is the process of managing changing requirements during the requirements engineering process and system development

B.2 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

3.5 Key points

1. Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
2. Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.

3. Non-functional requirements often constrain the system being developed and the development process being used.
4. They often relate to the emergent properties of the system and therefore apply to the system as a whole.
5. The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.
6. The requirements engineering process is an iterative process including requirements elicitation, specification and validation.
7. Requirements elicitation and analysis is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organization, requirements negotiation and requirements documentation.
8. You can use a range of techniques for requirements elicitation including interviews, scenarios, use-cases and ethnography.
9. Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism and verifiability.
10. Business, organizational and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.

3.6 Excercises(Homework): P116

4.2，*4.4

Chapter 4 (5) System Modeling

4.1 Topics covered

1. Context models
2. Interaction models
3. Structural models
4. Behavioral models

4.2 System modeling

4.2.1 Introduction

1. System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
2. System modeling has now come to mean representing a system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).
3. System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.

4.2.2 Use of graphical models

A.1 As a means of facilitating discussion about an existing or proposed system

Incomplete and incorrect models are OK as their role is to support discussion.

A.2 As a way of documenting an existing system

Models should be an accurate representation of the system but need not be complete.

A.3 As a detailed system description that can be used to generate a system implementation

Models have to be both correct and complete.

4.2.3 UML diagram types

A.1 Activity diagrams

which show the activities involved in a process or in data processing .

A.2 Use case diagrams

which show the interactions between a system and its environment.
Sequence diagrams

A.3 which show interactions between actors and the system and between system components.

A.4 Class diagrams

which show the object classes in the system and the associations between these classes.

A.5 State diagrams

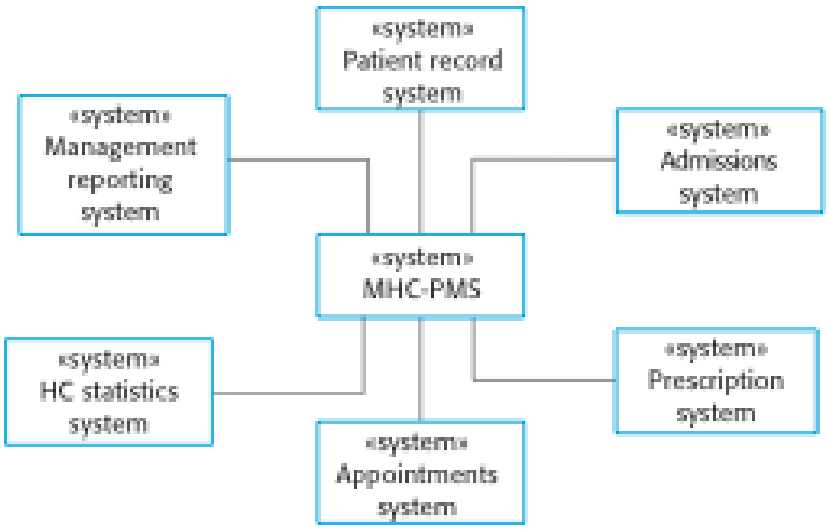
which show how the system reacts to internal and external events.

4.3 Context models

4.3.1 Introduction

- Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- Social and organisational concerns may affect the decision on where to position system boundaries.
- Architectural models show the system and its relationship with other systems.
    - The context of the MHC-PMS

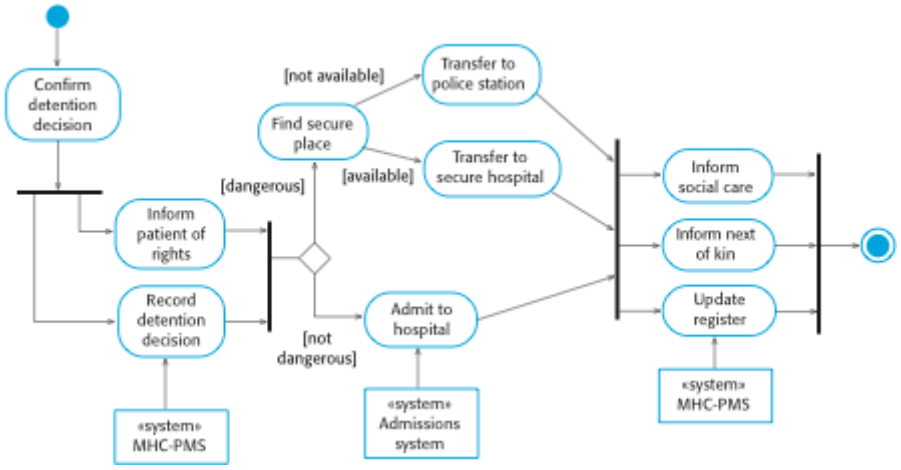
(Mental Health Care-Patient Management System)



4.3.2 Process perspective

- Process models reveal how the system being developed is used in broader business processes.
- UML activity diagrams may be used to define business process models.

Process model of involuntary detention



next of kin 最近的亲属

4.4 Interaction models

4.4.1 Concept

A.1 Modeling system-to-system interaction highlights the communication problems that may arise.

A.2 Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.

A.3 Use case diagrams and sequence diagrams may be used for interaction modeling Both of these models are required for a description of the system's behaviour

4.4.2 Use case modeling

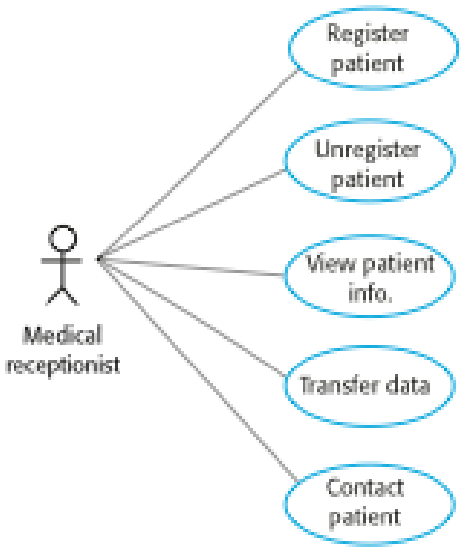
A.1 Transfer-data use case

A.2 Tabular description of the 'Transfer data' use-case

MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)

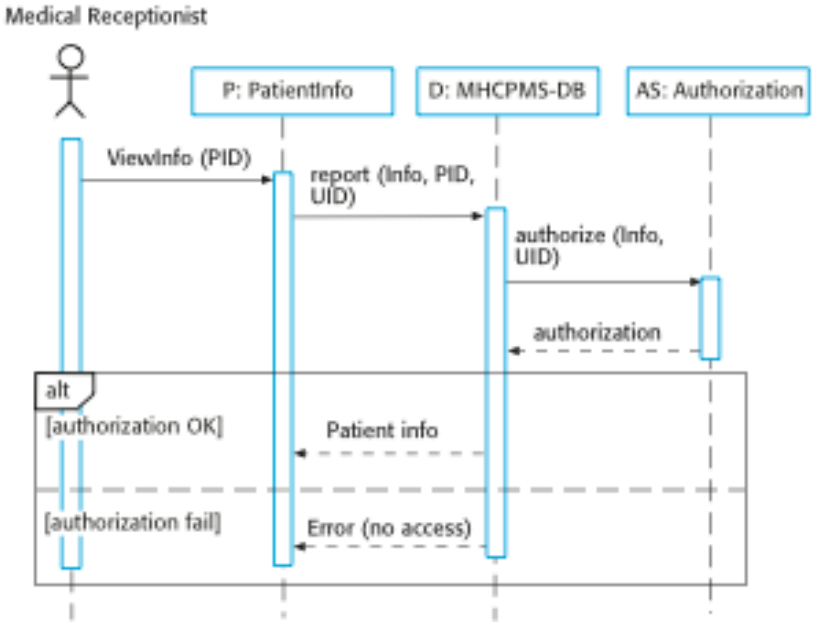
Description	A receptionist may transfer data from the MHC-PMS to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

A.3 Use cases in the MHC-PMS involving the role 'Medical Receptionist'



4.4.3 Sequence diagrams

Sequence diagram for View patient information



4.5 Structural models

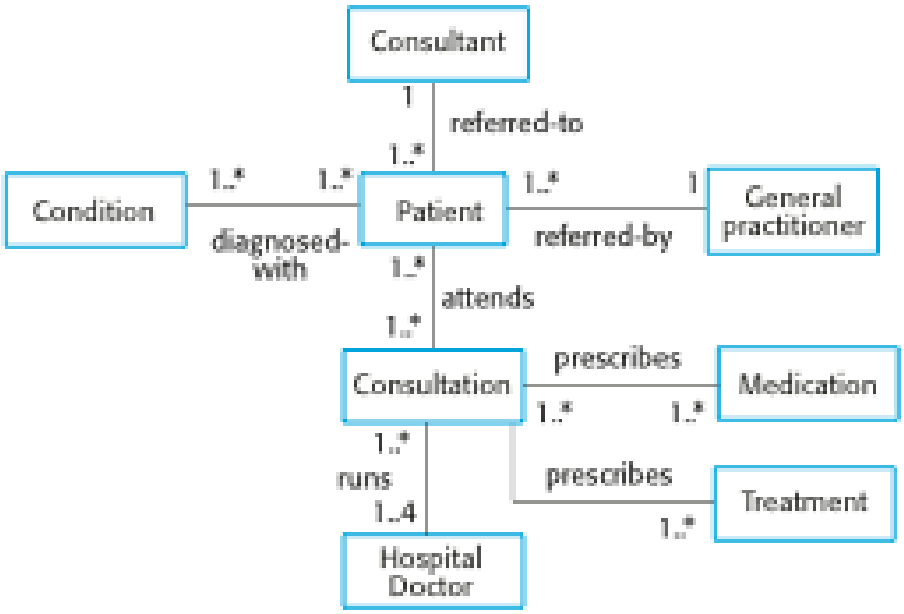
- Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.
- Structural models may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
- You create structural models of a system when you are discussing and designing the system architecture.

4.5.1 Class diagrams

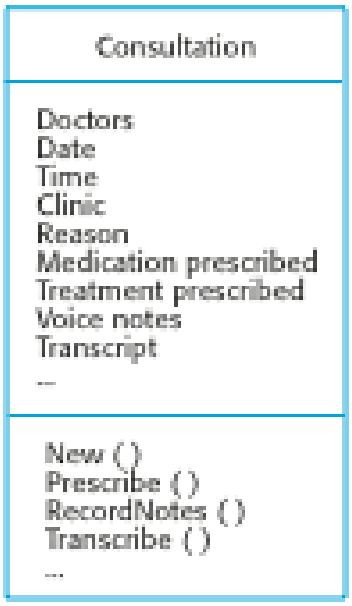
- Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.
- An object class can be thought of as a general definition of one kind of system object.
- An association is a link between classes that indicates that there is some relationship between these classes.

A.1 UML classes and association

B.1 Classes and associations in the MHC-PMS



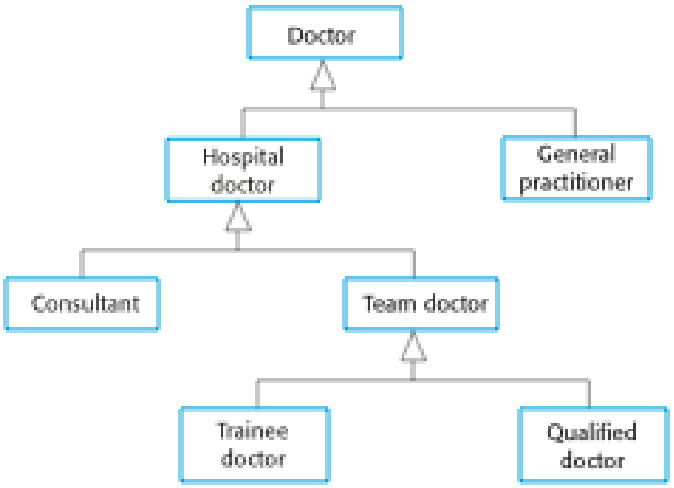
B.2 The Consultation class



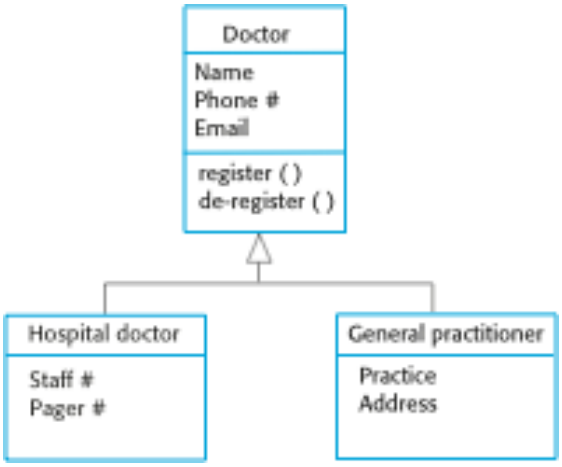
A.2 Generalization

In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.

B.1 A generalization hierarchy



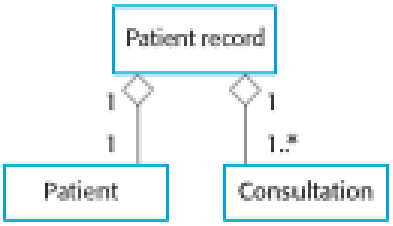
B.2 A generalization hierarchy with added detail



A.3 Object class aggregation models

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

B.1 The aggregation association



4.6 Behavioral models

4.6.1 Introduction

A.1 Behavioral models are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.

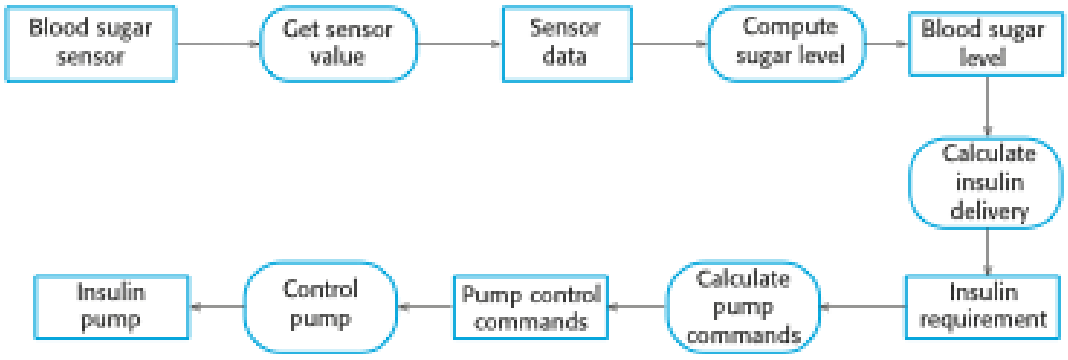
A.2 You can think of these stimuli as being of two types:

- Data Some data arrives that has to be processed by the system.
- Events Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

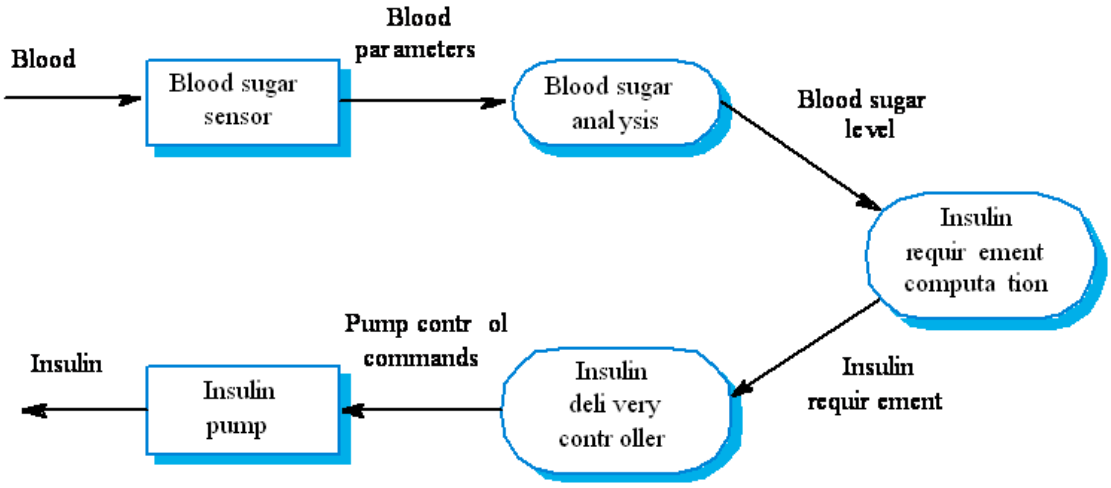
4.6.2 Data-driven modeling

- Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.
- Data-driven models show the sequence of actions involved in processing input data and generating an associated output.
- They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.

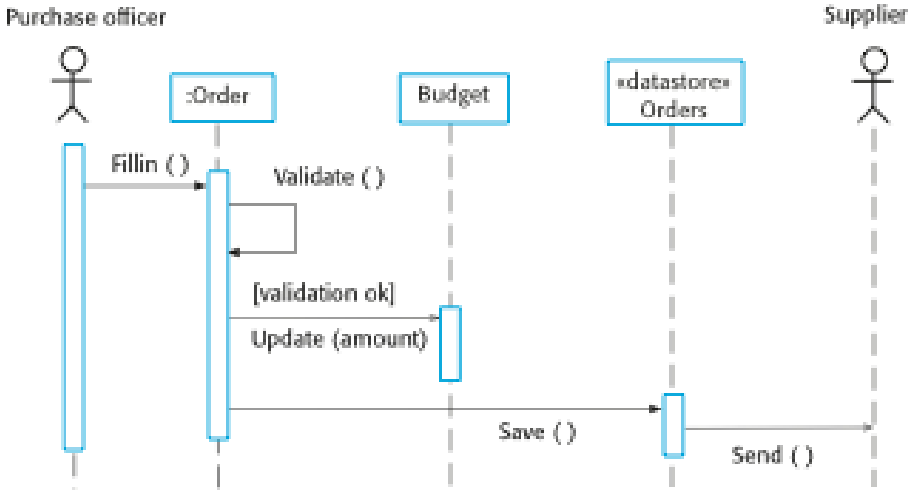
An activity model of the insulin pump's operation



Insulin pump DFD（Data flow diagrams）



Order processing



4.6.3 Event-driven modeling

- Event-driven modeling shows how a system responds to external and internal events.
- It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

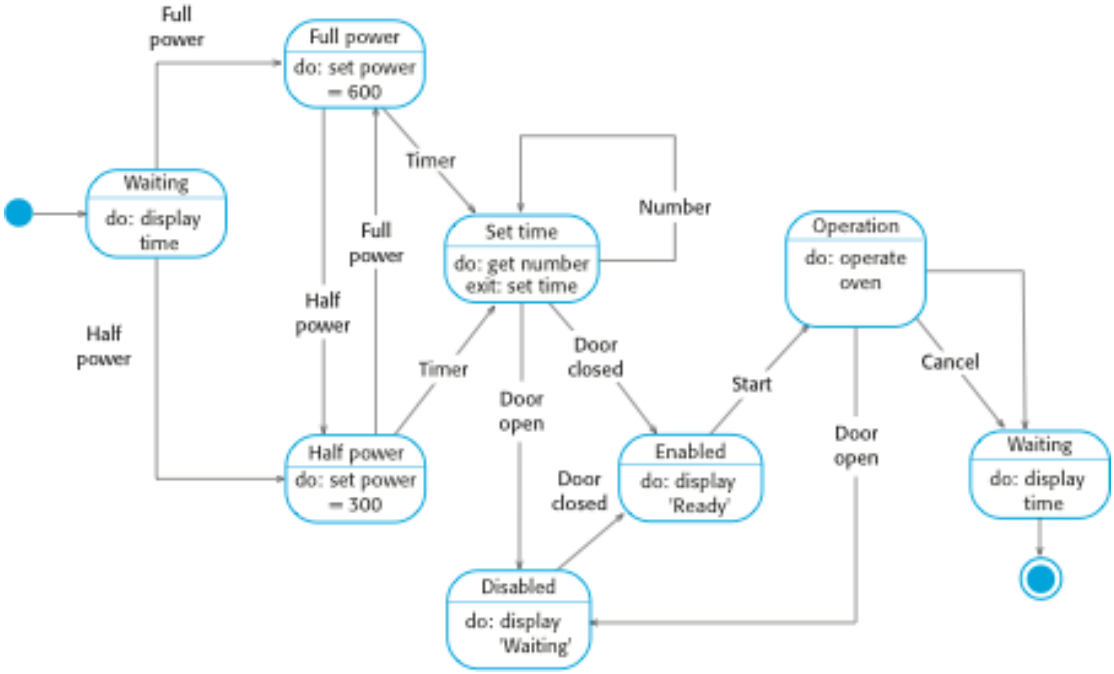
A.1 State machine models

1. These model the behaviour of the system in response to external and internal events.
2. They show the system's responses to stimuli so are often used for modelling real-time systems.
3. 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.

4. Statecharts are an integral part of the UML and are used to represent state machine models.

Microwave oven model



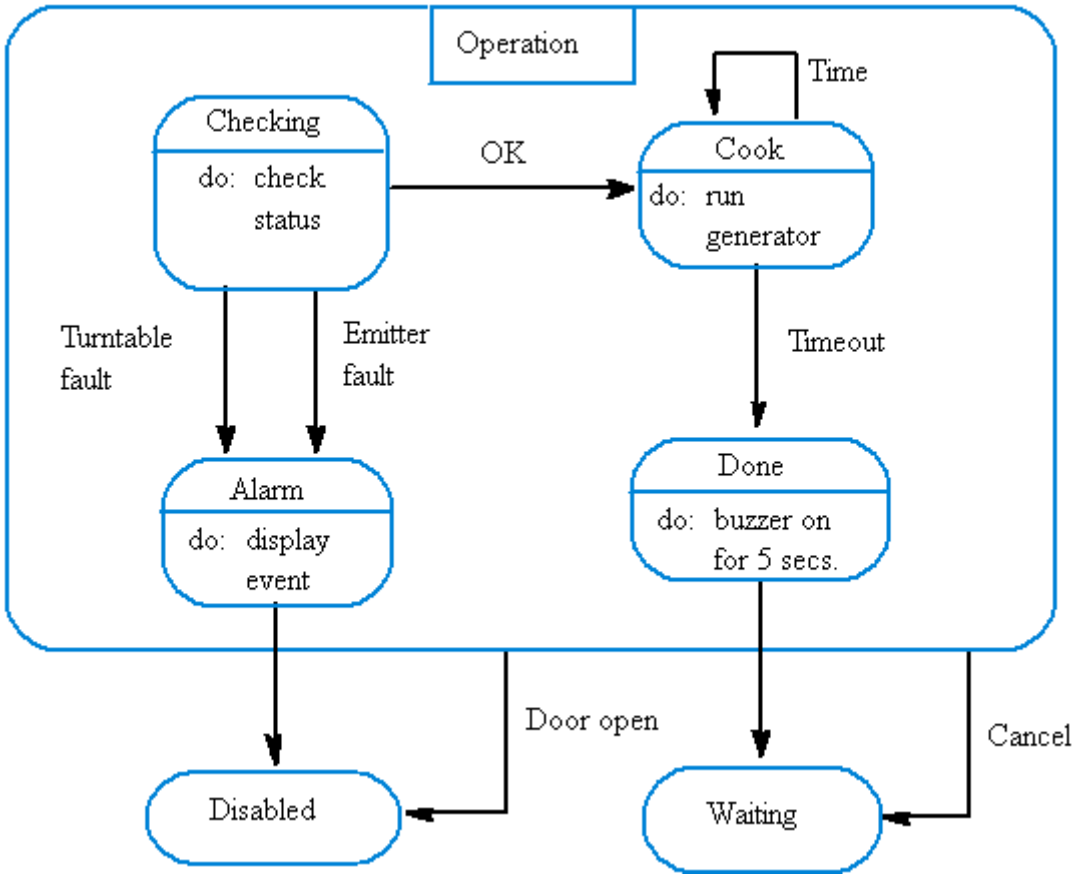
States and stimuli for the microwave oven

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 five seconds. Oven light is on. Display shows ‘Cooking complete’ while buzzer is sounding.

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

Microwave oven operation



4.7 Model-driven engineering

4.8 Introduction

1. Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process.
2. The programs that execute on a hardware/software platform are then generated automatically from the models.
3. Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

4.8.1 Usage of model-driven engineering

A.1 Model-driven engineering is still at an early stage of development, and it is unclear whether or not it will have a significant effect on software engineering practice.

B.1 Pros

Allows systems to be considered at higher levels of abstraction

Generating code automatically means that it is cheaper to adapt systems to new platforms.

B.2 Cons

Models for abstraction and not necessarily right for implementation.

Savings from generating code may be outweighed by the costs of developing translators for new platforms.

4.8.2 Executable UML

1. The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible.
2. This is possible using a subset of UML 2, called Executable UML or xUML.

4.9 Key points

1. A model is an abstract view of a system that ignores system details. Complementary system models can be developed to show the system's context, interactions, structure and behavior.
2. Context models show how a system that is being modeled is positioned in an environment with other systems and processes.
3. Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
4. Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.
5. Behavioral models are used to describe the dynamic behavior of an executing system. This behavior can be modeled from the perspective of the data processed by the system, or by the

events that stimulate responses from a system.

6. Activity diagrams may be used to model the processing of data, where each activity represents one process step.
7. State diagrams are used to model a system's behavior in response to internal or external events.
8. Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.

4.10 Excercises(Homework): P143-144

5.2，5.5，5.6，5.7

Chapter 5 (6) Architectural Design

5.1 Topics covered

1. Architectural design decisions
2. Architectural views
3. Architectural patterns
4. Application architectures

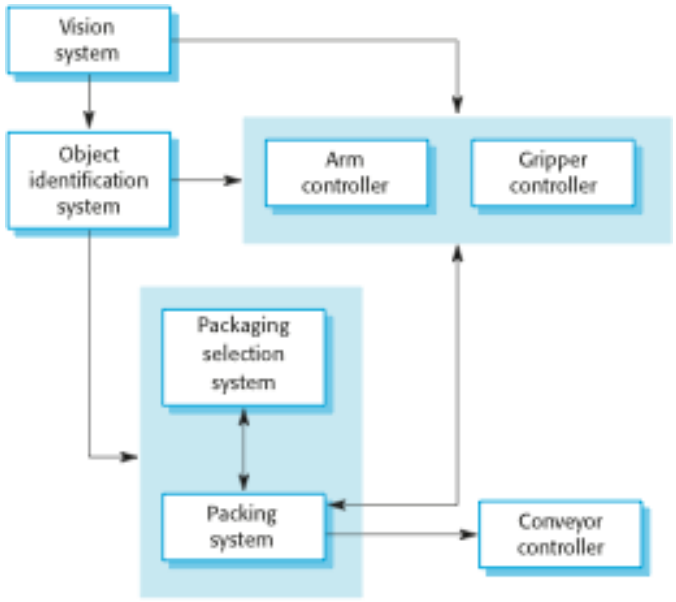
5.2 Architecture design

5.2.1 Software architecture design

A.1 Software architecture design

- 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

The architecture of a packing robot control system



A.2 Advantages of explicit architecture

B.1 Stakeholder communication

Architecture may be used as a focus of discussion by system stakeholders.

B.2 System analysis

Means that analysis of whether the system can meet its non-functional requirements is possible.

B.3 Large-scale reuse

1. The architecture may be reusable across a range of systems
2. Product-line architectures may be developed.

A.3 Architectural representations

1. Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
2. But these have been criticised because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
3. Depends on the use of architectural models.The requirements for model semantics depends on how the models are used.

5.2.2 Architectural design decisions

A.1 Architectural design decisions

1. Is there a generic application architecture that can be used?
2. How will the system be distributed?
3. What architectural styles are appropriate?
4. What approach will be used to structure the system?
5. How will the system be decomposed into modules?
6. What control strategy should be used?
7. How will the architectural design be evaluated?
8. How should the architecture be documented?

A.2 Architecture and system characteristics

B.1 Performance

Localise critical operations and minimise communications. Use large rather than fine-grain components.

B.2 Security

Use a layered architecture with critical assets in the inner layers.

B.3 Safety

Localise safety-critical features in a small number of sub-systems.

B.4 Availability

Include redundant components and mechanisms for fault tolerance.

B.5 Maintainability

Use fine-grain, replaceable components.

5.2.3 Architectural views

A.1 Architectural views

1. What views or perspectives are useful when designing and documenting a system’s architecture?
2. What notations should be used for describing architectural models?
3. Each architectural model only shows one view or perspective of the system.

A.2 4 + 1 view model of software architecture

1. A logical view, which shows the key abstractions in the system as objects or object classes.

2. A process view, which shows how, at run-time, the system is composed of interacting processes.
3. A development view, which shows how the software is decomposed for development.
4. A physical view, which shows the system hardware and how software components are distributed across the processors in the system.
5. Related using use cases or scenarios (+1)

5.3 Architectural patterns

5.3.1 Architectural patterns

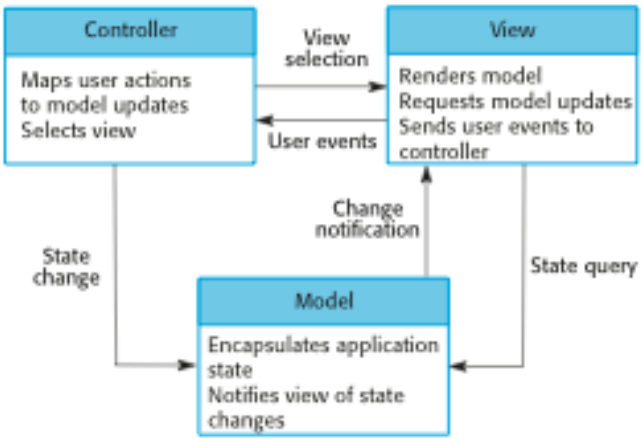
1. Patterns are a means of representing, sharing and reusing knowledge.
2. An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
3. Patterns should include information about when they are and when the are not useful.
4. Patterns may be represented using tabular and graphical descriptions.

5.3.2 The Model-View-Controller (MVC) pattern

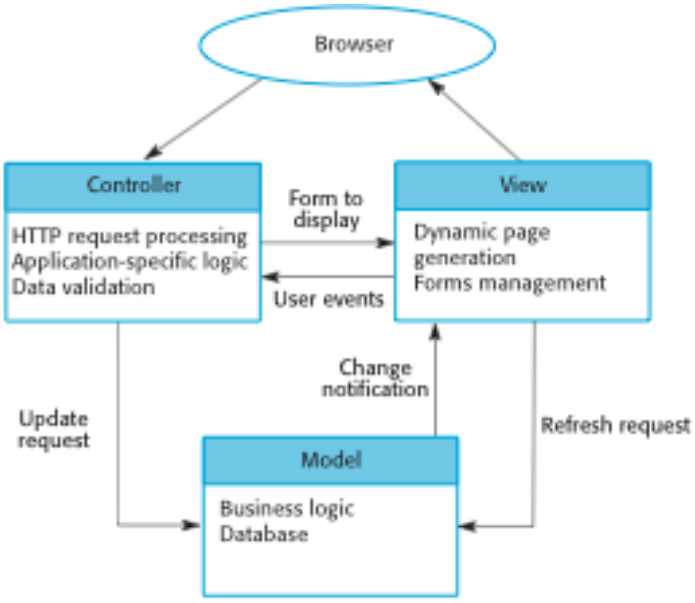
A.1 The Model-View-Controller (MVC) pattern

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

A.2 The organization of the Model-View-Controller



A.3 Web application architecture using the MVC pattern



5.3.3 Layered architecture

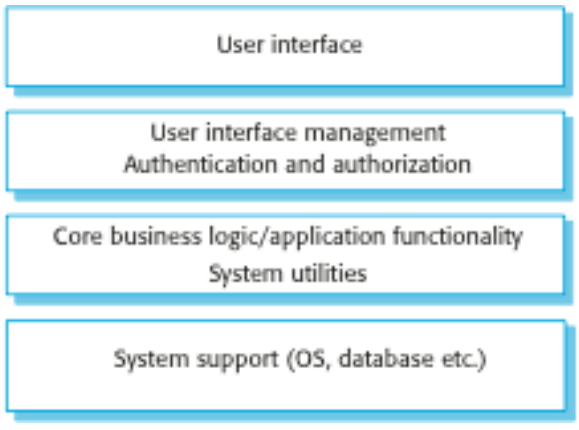
A.1 Layered architecture

1. Used to model the interfacing of sub-systems.
2. Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
3. Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.

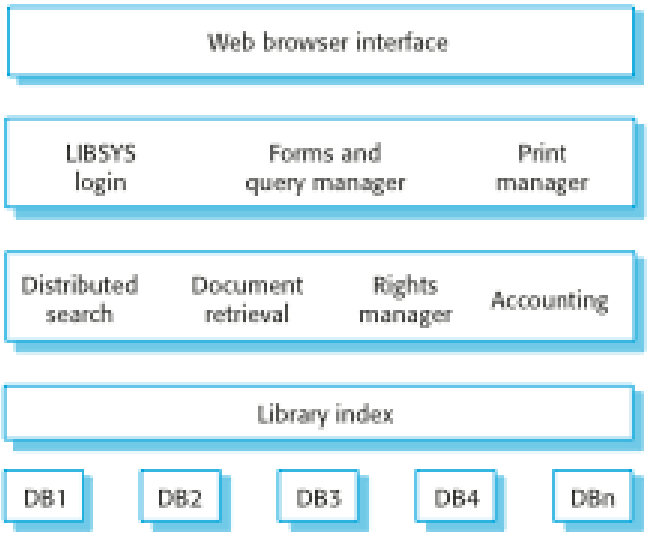
A.2 The Layered architecture pattern

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

A.3 A generic layered architecture



A.4 The architecture of the LIBSYS system



5.3.4 Repository architecture

A.1 Repository architecture

B.1 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.

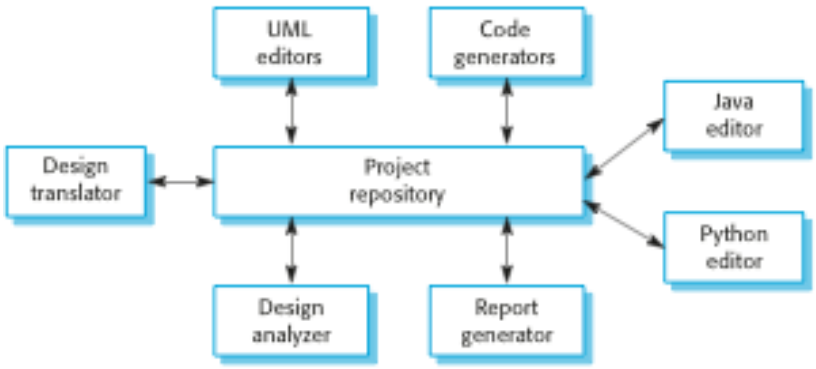
B.2 When large amounts of data are to be shared, the repository model of sharing is most commonly used a this is an efficient data sharing mechanism.

A.2 The Repository pattern

Name	**Repository**
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one

	place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

A.3 A repository architecture for an IDE



5.3.5 Client-server architecture

A.1 Client-server architecture

B.1 Distributed system model which shows how data and processing is distributed across a range of components.

Can be implemented on a single computer.

B.2 Set of stand-alone servers which provide specific services such as printing, data management, etc.

B.3 Set of clients which call on these services.

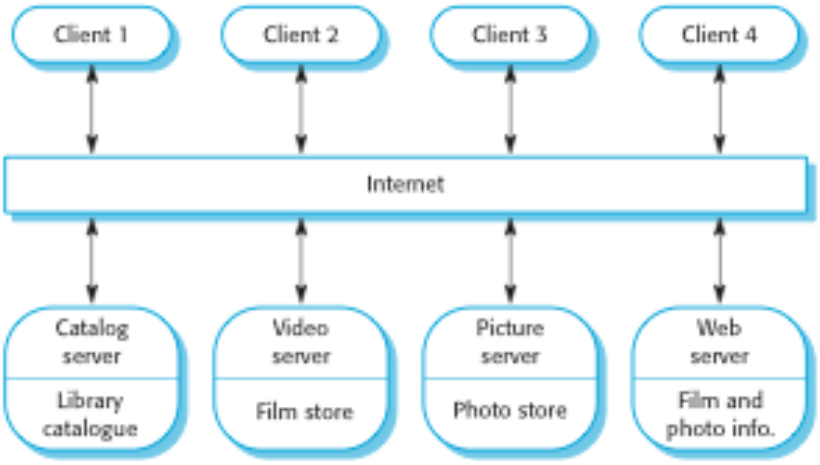
B.4 Network which allows clients to access servers.

A.2 The Client–server pattern

Name	**Client-server**
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be

	replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

A.3 A client–server architecture for a film library



5.3.6 Pipe and filter architecture

A.1 Pipe and filter architecture

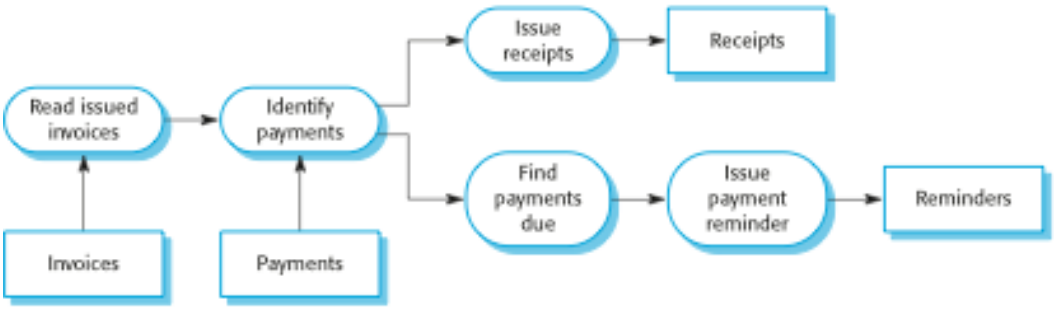
1. Functional transformations process their inputs to produce outputs.
2. May be referred to as a pipe and filter model (as in UNIX shell).
3. Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing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

A.2 The pipe and filter pattern

Name	**Pipe and filter**
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.

When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

A.3 An example of the pipe and filter architecture



5.4 Application architectures

5.4.1 Application architectures

1. As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
2. A generic application architecture is an architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements.

5.4.2 Examples of application types

A.1 Data processing applications

Data driven applications that process data in batches without explicit user intervention during the

processing.

A.2 Transaction processing applications

Data-centred applications that process user requests and update information in a system database.

A.3 Event processing systems

Applications where system actions depend on interpreting events from the system's environment.

A.4 Language processing systems

Applications where the users' intentions are specified in a formal language that is processed and interpreted by the system.

5.4.3 Transaction processing systems

A.1 Transaction processing systems

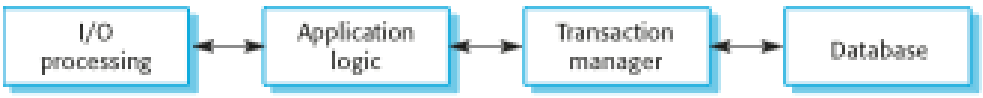
B.1 Process user requests for information from a database or requests to update the database.

B.2 From a user perspective a transaction is:

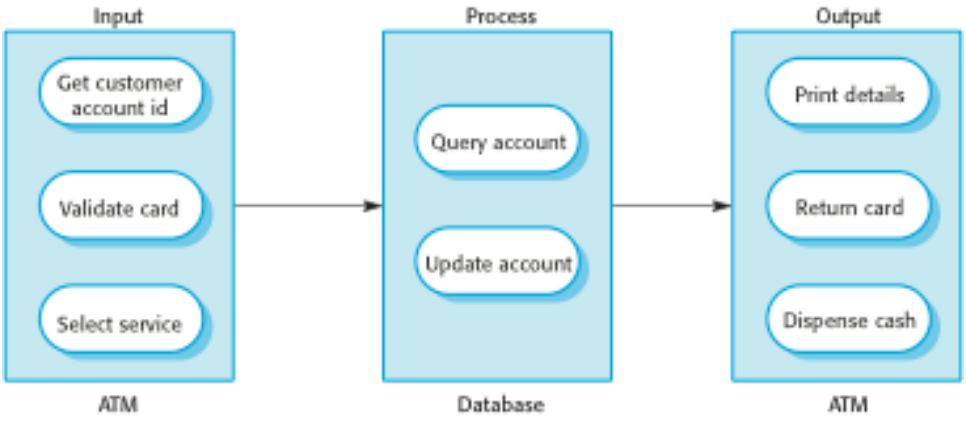
1. Any coherent sequence of operations that satisfies a goal;
2. For example - find the times of flights from London to Paris.

B.3 Users make asynchronous requests for service which are then processed by a transaction manager.

A.2 The structure of transaction processing applications



A.3 The software architecture of an ATM system



5.4.4 Information systems architecture

A.1 Information systems architecture

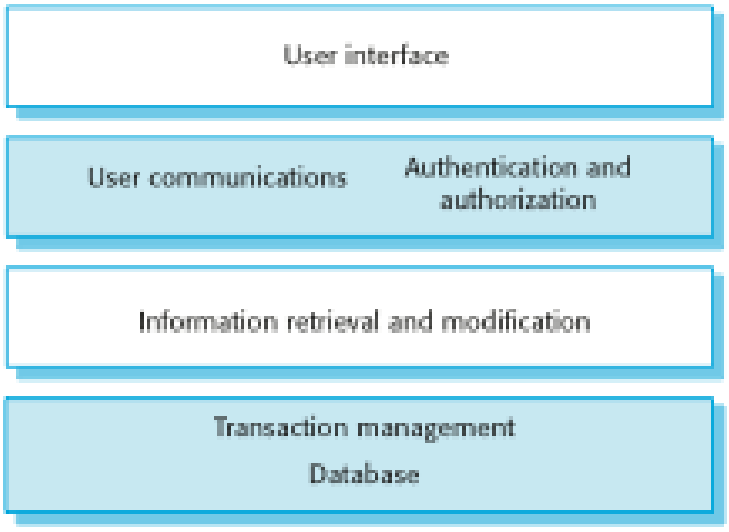
B.1 Information systems have a generic architecture that can be organised as a layered architecture.

B.2 These are transaction-based systems as interaction with these systems generally involves database transactions.

B.3 Layers include:

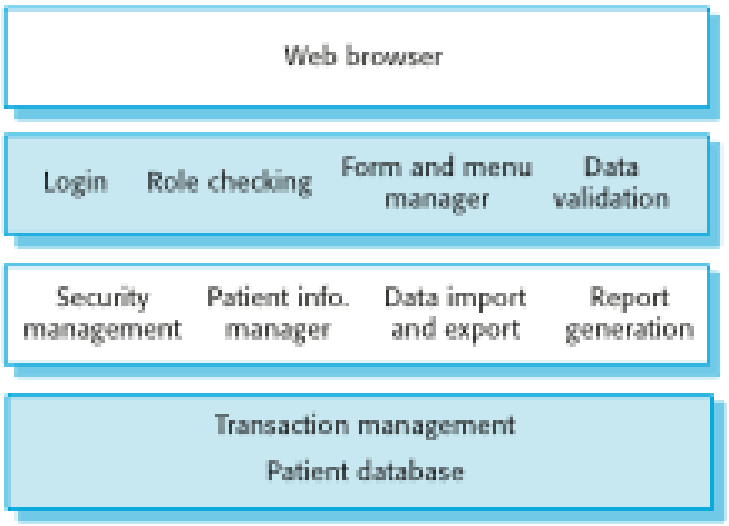
1. The user interface
2. User communications
3. Information retrieval
4. System database

A.2 Layered information system architecture

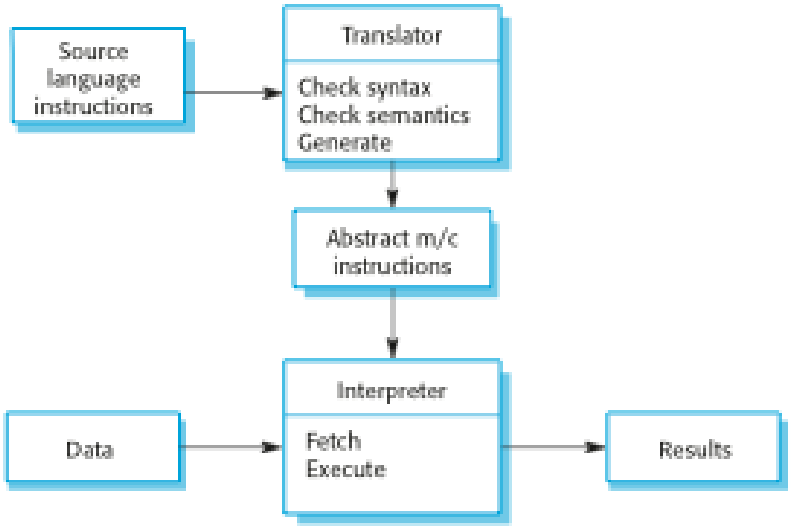


A.3 Web-based information systems

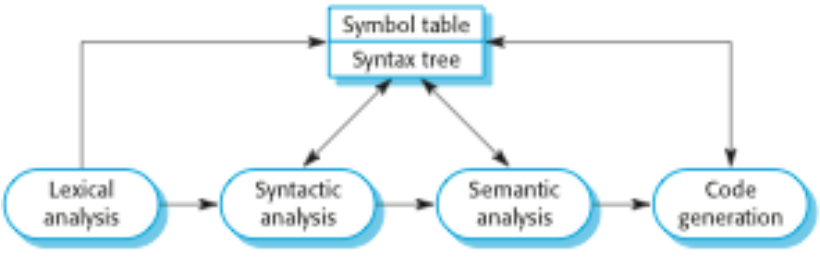
B.1 The architecture of the MHC-PMS



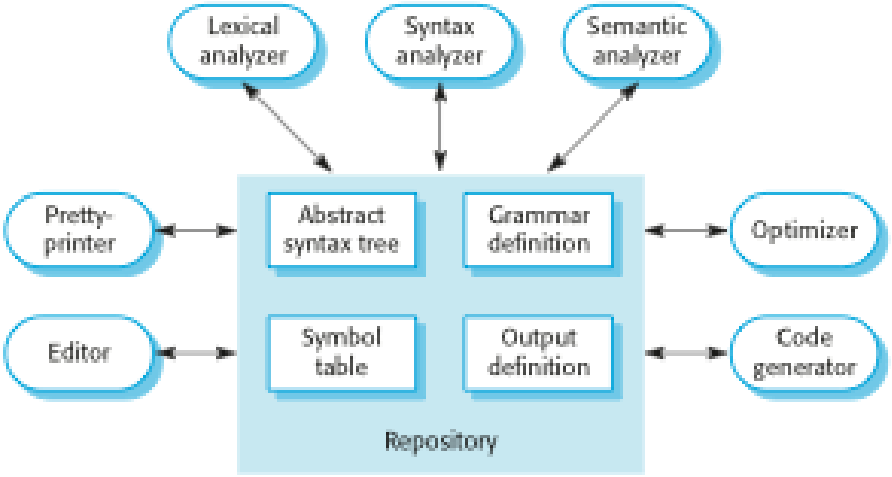
5.4.5 Language processing systems



A.1 A pipe and filter compiler architecture



A.2 A repository architecture for a language processing system



5.5 Key points

1. A software architecture is a description of how a software system is organized.
2. Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
3. Architectures may be documented from several different perspectives or viewssuch as a conceptual view, a logical view, a process view, and a development view.
4. Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.
5. Models of application systems architectures help us understand and compare applications, validate application system designs and assess large-scale components for reuse.
6. Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users.
7. Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.

5.6 Excercises(Homework): P173-174

6.1,6.3,6.9

Chapter 6 (7) Design and Implementation

6.1 Topics covered

1. Object-oriented design using the UML
2. Design patterns
3. Implementation issues
4. Open source development

6.2 Design and implementation

6.2.1 Design and implementation

Software design and implementation is the stage in the software engineering process at which an executable software system is developed.

A.1 Build or buy

B.1 In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.

For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.

B.2 When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

6.2.2 面向数据流的设计方法

变换分析是一系列设计步骤的总称，经过这些步骤把具有变换流特点的数据流图按预先确定的模式映射成软件结构。

D.1 1. 例子

假设的仪表板将完成下述功能：

(1) 通过模数转换实现传感器和微处理机接口；

(2) 在发光二极管面板上显示数据；

(3) 指示每小时英里数(mph)，行驶的里程，每加仑油行驶的英里数(mpg)等等；

(4) 指示加速或减速；

(5) 超速警告：如果车速超过 55 英里/小时，则发出超速警告铃声。

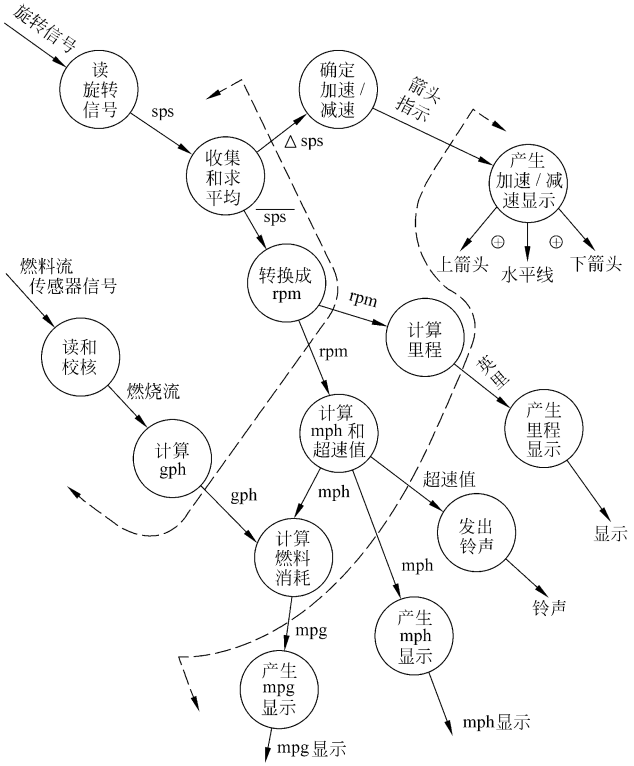


图 5.11 数字仪表板系统的数据流图

D.2　2. 设计步骤

- 第 1 步　复查基本系统模型。
- 第 2 步　复查并精化数据流图。

假设在需求分析阶段产生的数字仪表板系统的数据流图如图 5.11（见书 97 页）所示。

- 第 3 步　确定数据流图具有变换特性还是事务特性。
- 第 4 步　确定输入流和输出流的边界，从而孤立出变换中心。
- 第 5 步　完成“第一级分解”。

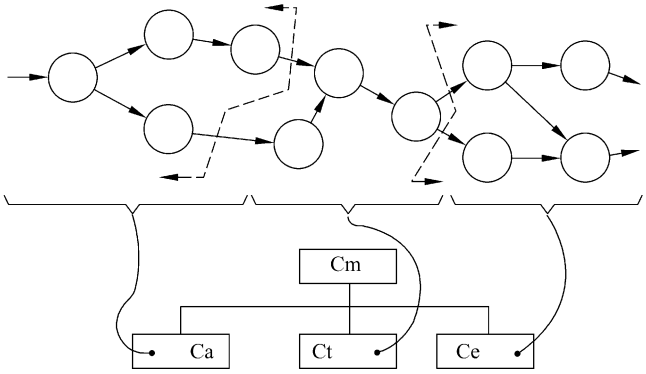


图 5.13 第一级分解的方法

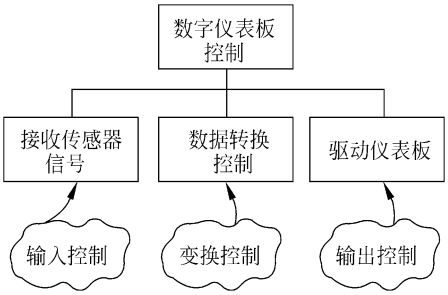


图 5.14 数字仪表板系统的第一级分解

- 第 6 步 完成“第二级分解”。

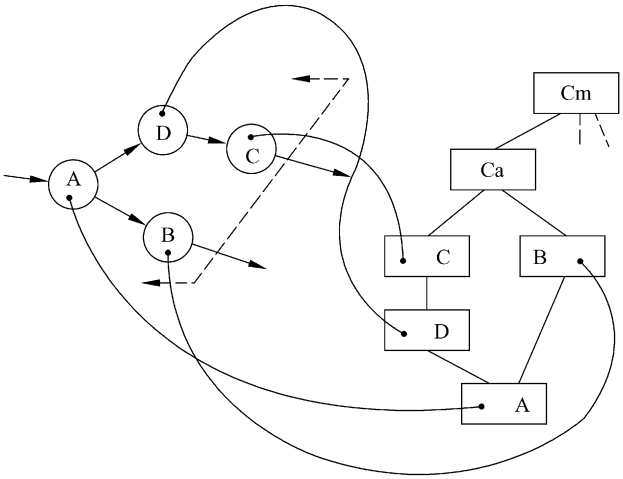


图 5.15 第二级分解的方法

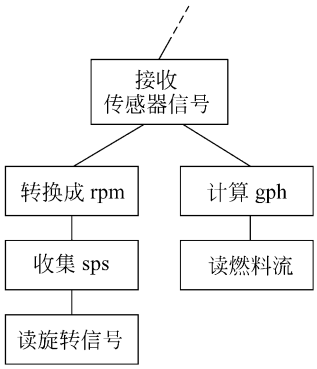


图 5.16 未经精化的输入结构

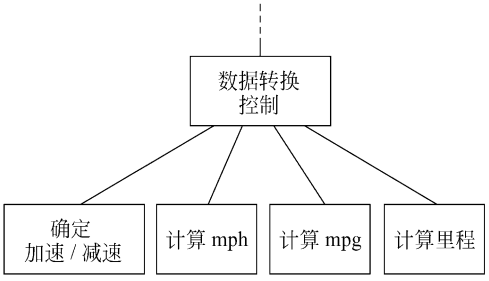


图 5.17 未经精化的变换结构

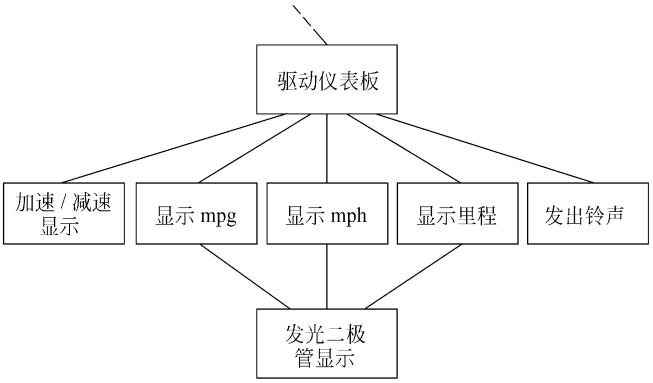


图 5.18 未经精化的输出结构

- 第 7 步 使用设计度量和启发式规则对第一次分割得到的软件结构进一步精化。

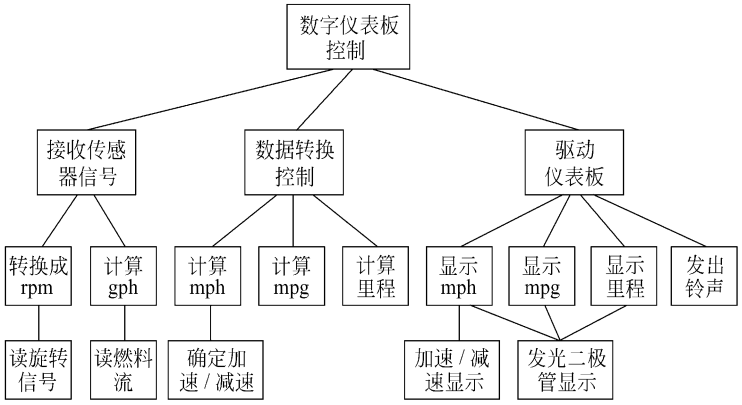


图 5.19 精化后的数字仪表板系统的软件结构

6.3 An object-oriented design process

6.3.1 An object-oriented design process

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

6.3.2 Process stages

A.1 There are a variety of different object-oriented design processes that depend on the organization using the process.

A.2 Common activities in these processes include:

1. Define the context and modes of use of the system;
2. Design the system architecture;
3. Identify the principal system objects;
4. Develop design models;
5. Specify object interfaces.

A.3 Process illustrated here using a design for a wilderness weather station.

6.3.3 a wilderness weather station

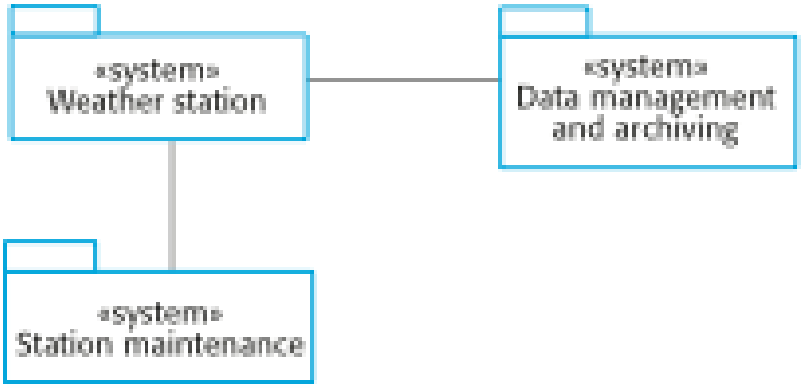
A.1 weather station

B.1 The government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas.

B.2 Weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed and wind direction.

The weather station includes a number of instruments that measure weather parameters such as the wind speed and direction, the ground and air temperatures, the barometric pressure and the rainfall over a 24-hour period. Each of these instruments is controlled by a software system that takes parameter readings periodically and manages the data collected from the instruments.

A.2 The weather station's environment



A.3 Weather information system

B.1 The weather station system

This is responsible for collecting weather data, carrying out some initial data processing and transmitting it to the data management system.

B.2 The data management and archiving system

This system collects the data from all of the wilderness weather stations, carries out data processing and analysis and archives the data.

B.3 The station maintenance system

This system can communicate by satellite with all wilderness weather stations to monitor the health of these systems and provide reports of problems.

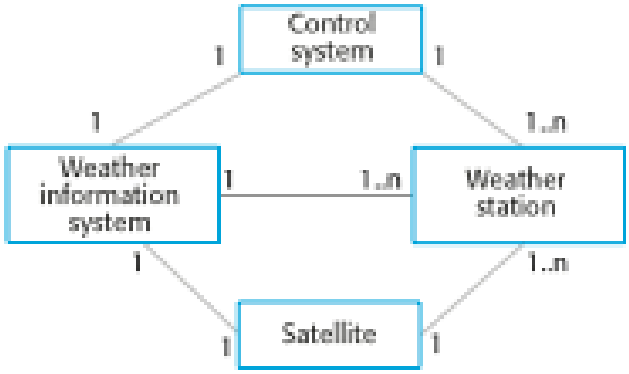
B.4 Additional software functionality

1. Monitor the instruments, power and communication hardware and report faults to the management system.
2. Manage the system power, ensuring that batteries are charged whenever the environmental conditions permit but also that generators are shut down in potentially damaging weather conditions, such as high wind.
3. Support dynamic reconfiguration where parts of the software are replaced with new versions and where backup instruments are switched into the system in the event of system failure.

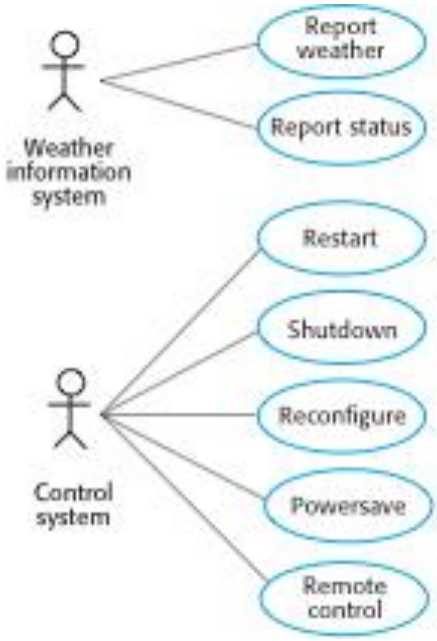
6.3.4 System context and interactions

1. A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
2. An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

A.1 System context for the weather station



A.2 Weather station use cases



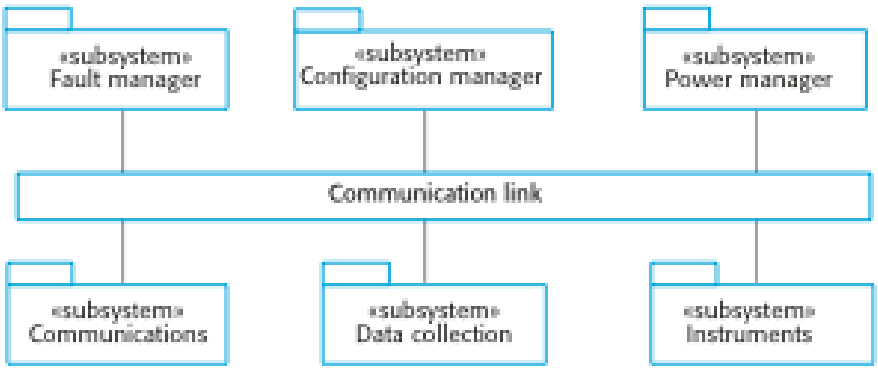
B.1 Use case description—Report weather

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information 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 five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

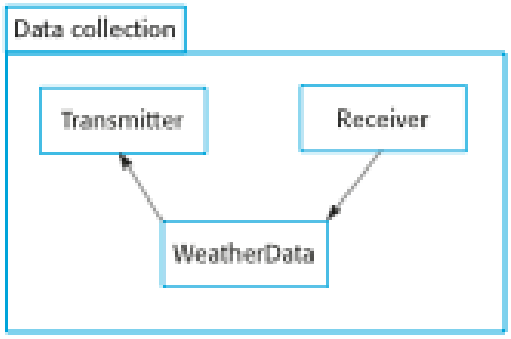
6.3.5 Architectural design

The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure

High-level architecture of the weather station



Architecture of data collection system



6.3.6 Object class identification

A.1 Object class identification

1. Identifying object classes is often a difficult part of object oriented design.
2. There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
3. Object identification is an iterative process. You are unlikely to get it right first time.

A.2 Approaches to identification

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

A.3 Weather station object classes

B.1 Weather station description

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

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

B.2 Object class identification in the weather station system may be based on the tangible hardware and data in the system:

C.1 Ground thermometer, Anemometer, Barometer

Application domain objects that are ‘hardware’ objects related to the instruments in the system.

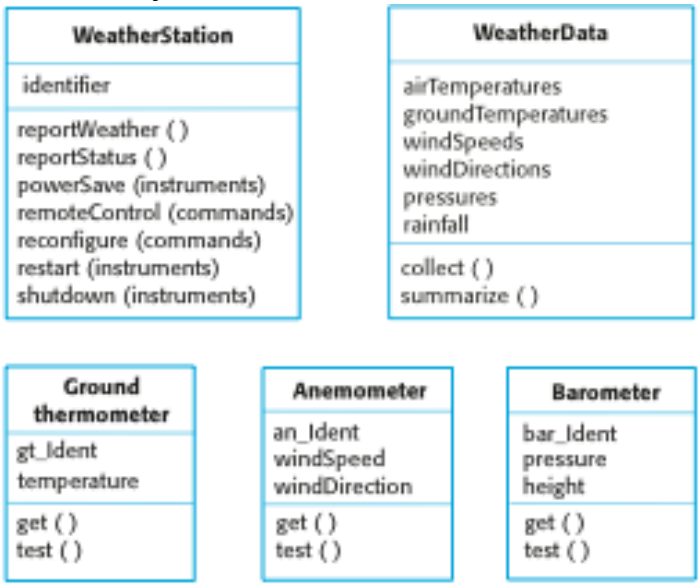
C.2 Weather station

The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.

C.3 Weather data

Encapsulates the summarized data from the instruments.

B.3 Weather station object classes



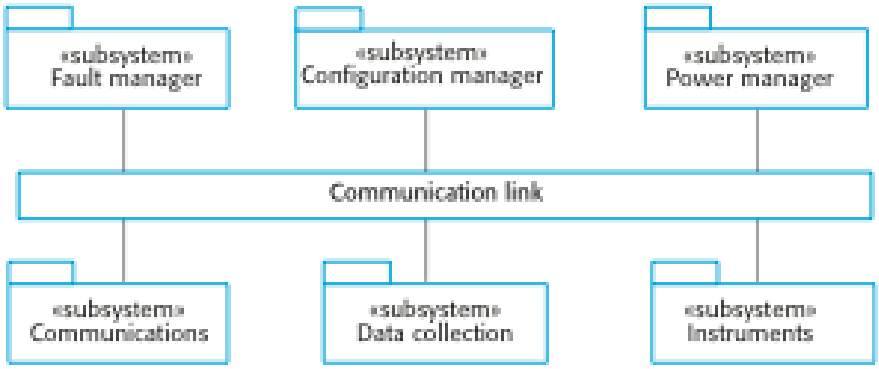
6.3.7 Design models

A.1 Design models

1. Static models describe the static structure of the system in terms of object classes and relationships.
2. Dynamic models describe the dynamic interactions between objects.

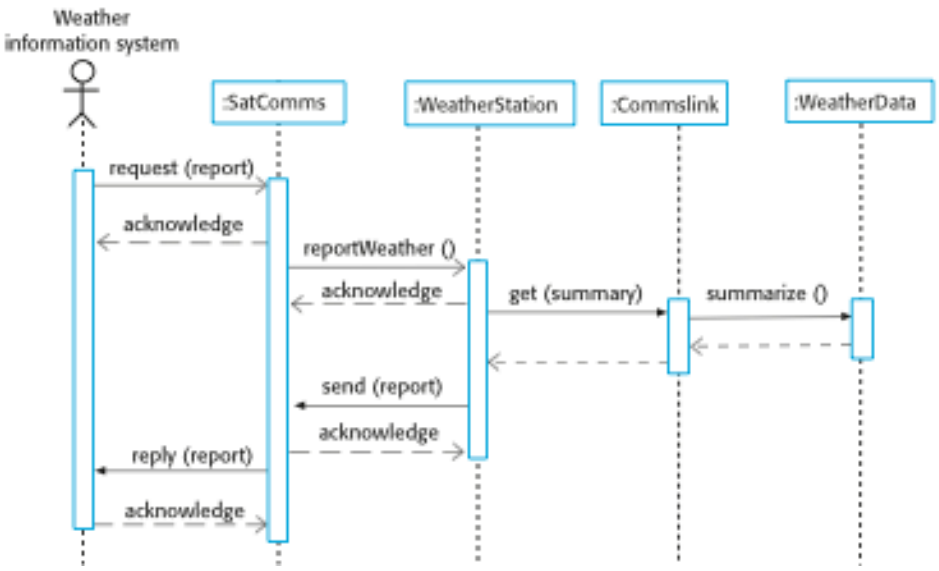
A.2 Examples of design models

B.1 Subsystem models that show logical groupings of objects into coherent subsystems.



B.2 Sequence models that show the sequence of object interactions.

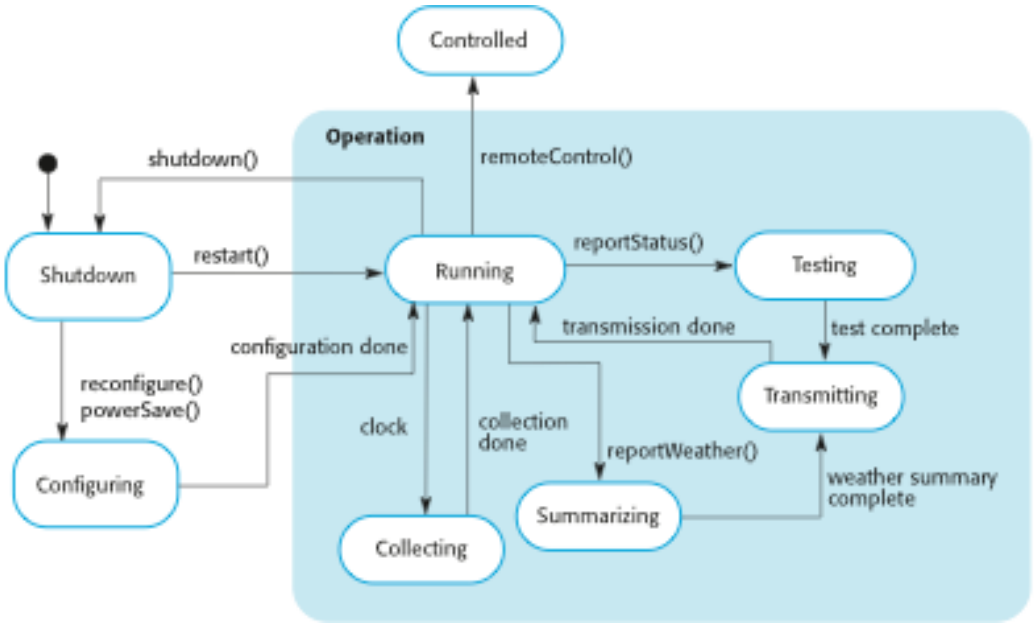
Sequence diagram describing data collection



B.3 State machine models that show how individual objects change their state in response to events.

- You don't usually need a state diagram for all of the objects in the system.

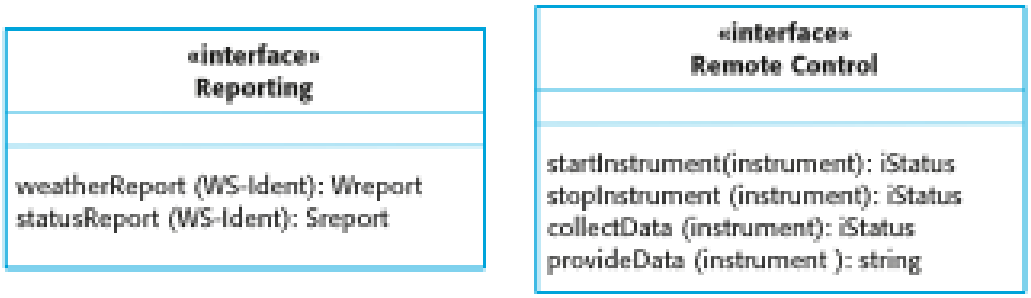
Weather station state diagram



B.4 Other models include use-case models, aggregation models, generalisation models, etc.

6.3.8 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.



6.4 Key points

1. Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
2. The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
3. A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
4. Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.

6.5 Excercises(Homework): P202-203

7.1, 7.3

Chapter 7 (8) Software Testing

7.1 Topics covered

1. Development testing
2. Test-driven development
3. Release testing

7.2 Program testing

7.2.1 Program testing

1. Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
2. When you test software, you execute a program using artificial data.
3. You check the results of the test run for errors, anomalies or information about the program's non-functional attributes.
4. Can reveal the presence of errors NOT their
   absence.
5. Testing is part of a more general verification and validation process, which also includes static validation techniques.

A.1 Program testing goals

B.1 To demonstrate to the developer and the customer that the software meets its requirements.

For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.

B.2 To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.

Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.

7.2.2 Validation and defect testing

A.1 Validation testing

1. You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
2. To demonstrate to the developer and the system customer that the software meets its requirements
3. A successful test shows that the system operates as intended.

A.2 Defect testing

1. The test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.
2. To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification
3. A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

A.3 Verification vs validation

B.1 Verification:
"Are we building the product right".

The software should conform to its specification.

B.2 Validation:

"Are we building the right product".

The software should do what the user really requires

A.4 V & V confidence

B.1 Aim of V & V is to establish confidence that the system is 'fit for purpose'.

B.2 Depends on system's purpose, user expectations and marketing environment

C.1 Software purpose

The level of confidence depends on how critical the software is to an organisation.

C.2 User expectations

Users may have low expectations of certain kinds of software.

C.3 Marketing environment

Getting a product to market early may be more important than finding defects in the program.

A.5 Inspections and testing

B.1 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.

Discussed in Chapter 15.

B.2 Software testing Concerned with exercising and observing product behaviour (dynamic verification)

The system is executed with test data and its operational behaviour is observed.

B.3 Software inspections

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

C.1 Advantages of inspections

During testing, errors can mask (hide) other errors. Because inspection is a static process, you don't have to be concerned with interactions between errors.

Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.

As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

B.4 Inspections and testing

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

A.6 Stages of testing

1. Development testing, where the system is tested during development to discover bugs and defects.
2. Release testing, where a separate testing team test a complete version of the system before it is released to users.
3. User testing, where users or potential users of a system test the system in their own environment.

7.3 Development testing

7.3.1 Development testing

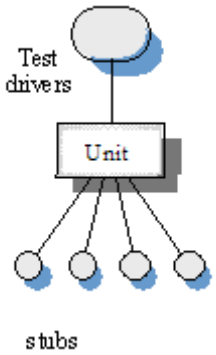
1. Development testing includes all testing activities that are carried out by the team developing the system.
2. Unit testing, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
3. Component testing, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
4. System testing, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

7.3.2 Unit testing

A.1 Unit testing

B.1 Unit testing is the process of testing individual components in isolation.
B.2 It is a defect testing process.
B.3 Units may be:

1. Individual functions or methods within an object
2. Object classes with several attributes and methods
3. Composite components with defined interfaces used to access their functionality.



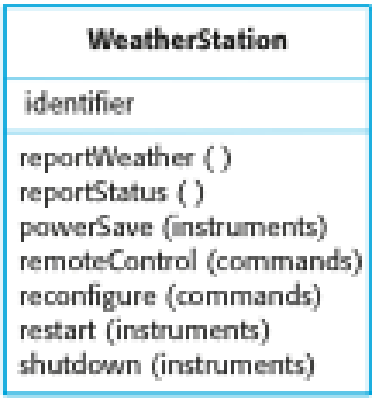
A.2 Object class testing

C.1 Complete test coverage of a class involves

1. Testing all operations associated with an object
2. Setting and interrogating all object attributes
3. Exercising the object in all possible states.

C.2 Inheritance makes it more difficult to design object class tests as the information to be tested is not localised.

B.2 The weather station object interface



C.1 Weather station testing

D.1 Need to define test cases for reportWeather, calibrate, test, startup and shutdown.

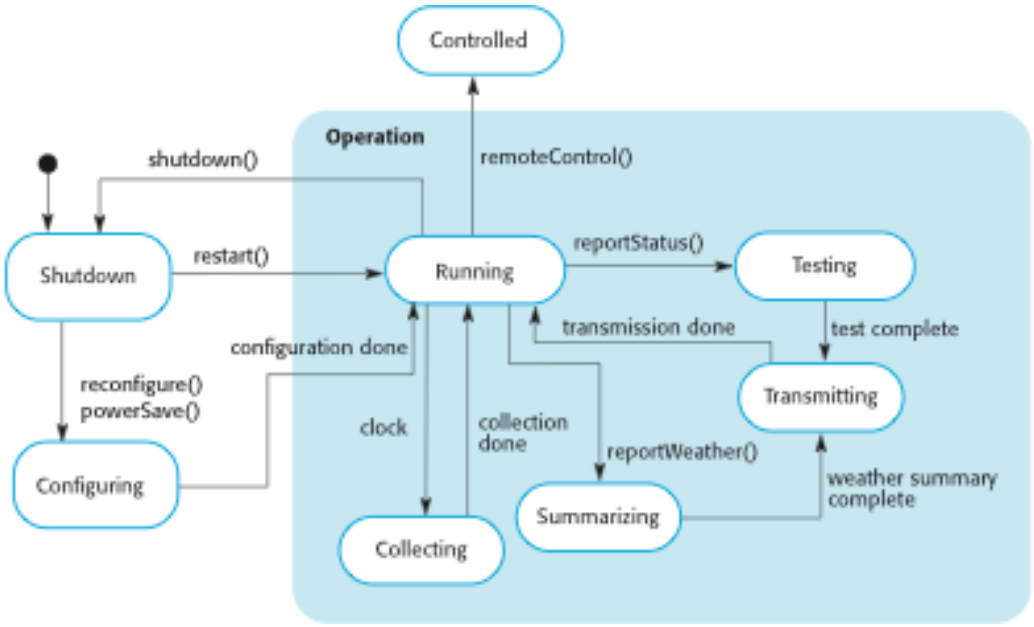
D.2 Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions

D.3 For example:

Shutdown -> Running-> Shutdown

Configuring-> Running-> Testing -> Transmitting -> Running

Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running



C.2 Automated testing

1. Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
2. In automated unit testing, you make use of a test automation framework (such as JUnit) to

write and run your program tests.

7.3.3 Testing strategies

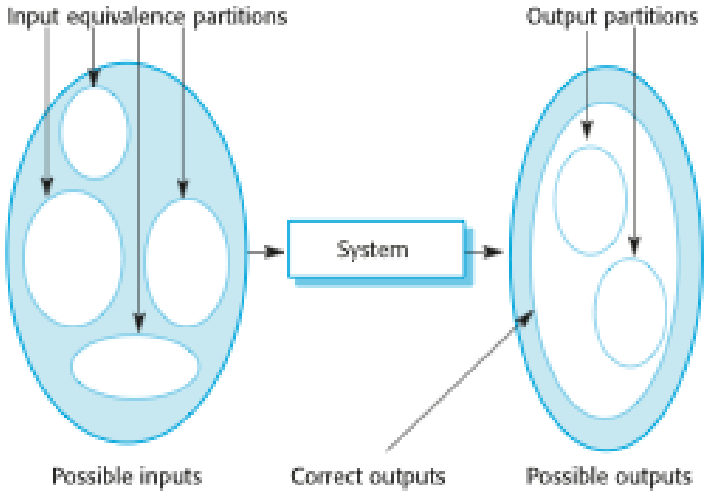
A.1 Testing strategies

1. The first of these should reflect normal operation of a program and should show that the component works as expected.
2. The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

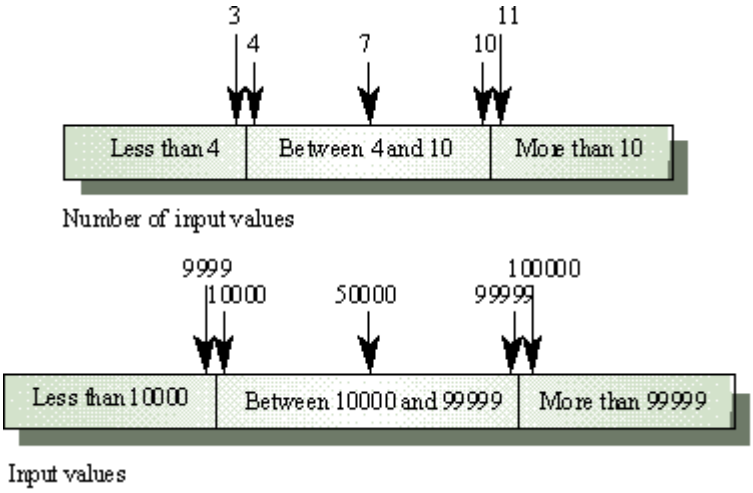
A.2 Partition testing

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

Equivalence partitioning



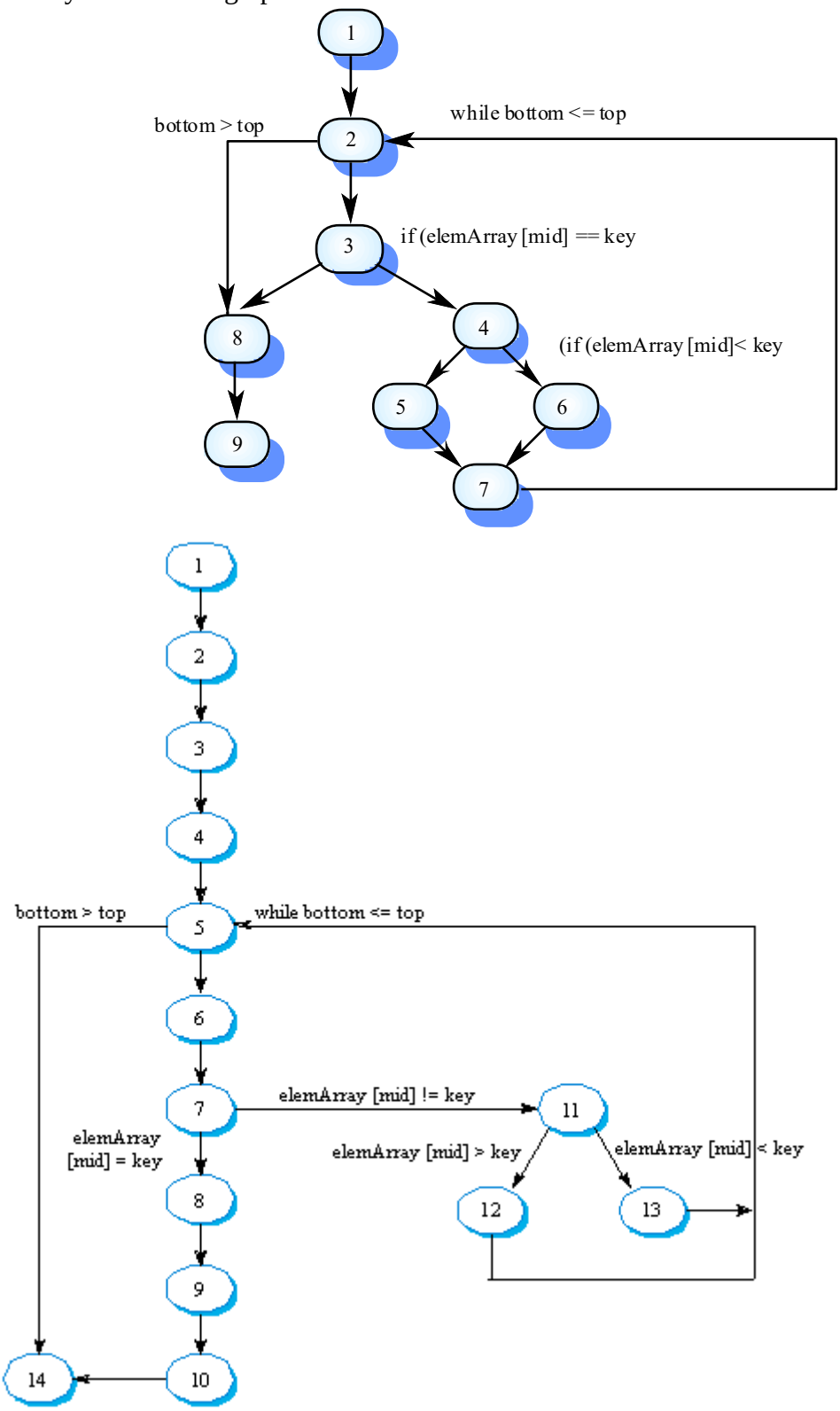
Equivalence partitions



A.3 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

Binary search flow graph



B.1 Independent paths

- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
- 1, 2, 3, 4, 5, 14

- 1, 2, 3, 4, 5, 6, 7, 11, 12, 5, …
- 1, 2, 3, 4, 6, 7, 2, 11, 13, 5, …
- Test cases should be derived so that all of these paths are executed

7.3.4 Component testing

A.1 Component testing

B.1 Software components are often composite components that are made up of several interacting objects.

B.2 You access the functionality of these objects through the defined component interface.

B.3 Testing composite components should therefore focus on showing that the component interface behaves according to its specification.

You can assume that unit tests on the individual objects within the component have been completed.

A.2 Interface testing

Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.

B.1 Interface types

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

B.2 Interface errors

C.1 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.

C.2 Interface misunderstanding

A calling component embeds assumptions about the behaviour of the called component which are incorrect.

C.3 Timing errors

The called and the calling component operate at different speeds and out-of-date information is accessed.

7.3.5 System testing

A.1 System testing

1. System testing during development involves integrating components to create a version of the system and then testing the integrated system.
2. The focus in system testing is testing the interactions between components.
3. System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
4. System testing tests the emergent behaviour of a system.

A.2 Integration testing

B.1 Introduction

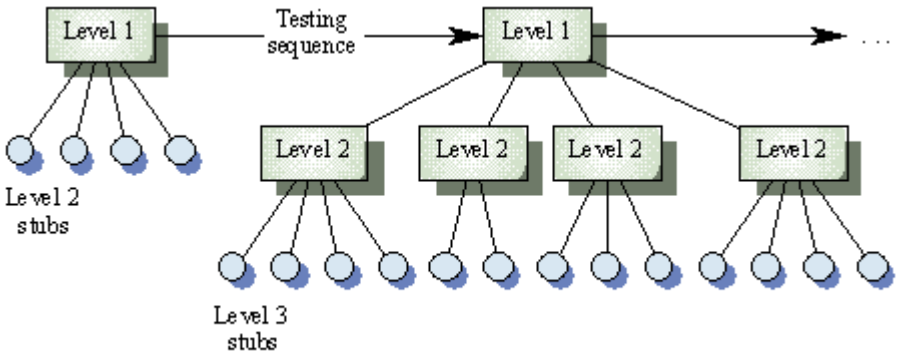
C.1 Involves building a system from its components and testing it for problems that arise from component interactions.

C.2 To simplify error localisation, systems should be incrementally integrated.

B.2 Approaches to integration testing

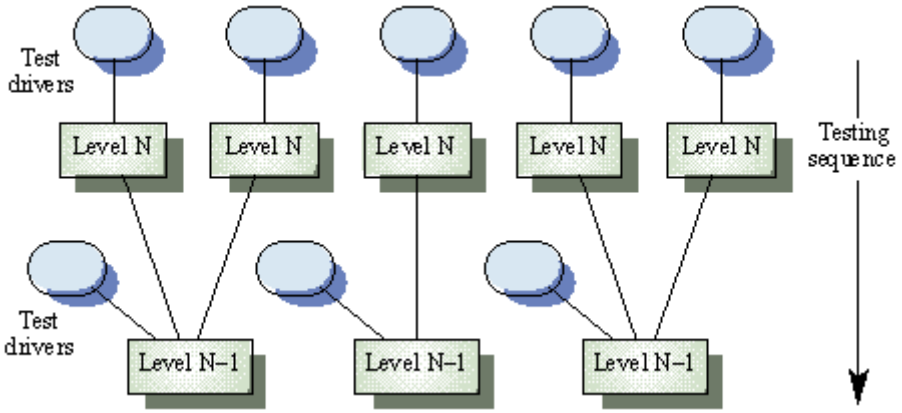
C.1 Top-down testing

Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate



C.2 Bottom-up testing

Integrate individual components in levels until the complete system is created



C.3 In practice, most integration involves a combination of these strategies

A.3 Use-case testing

1. The use-cases developed to identify system interactions can be used as a basis for system testing.
2. Each use case usually involves several system components so testing the use case forces these interactions to occur.
3. The sequence diagrams associated with the use case documents the components and interactions that are being tested.

A.4 Testing policies

B.1 Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.

B.2 Examples of testing policies:

1. All system functions that are accessed through menus should be tested.
2. Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
3. Where user input is provided, all functions must be tested with both correct and incorrect input.

7.4 Release testing

7.4.1 Release testing

A.1 Release testing

C.1 Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.

C.2 The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.

Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.

C.3 Release testing is usually a black-box testing process where tests are only derived from the system specification.

B.2 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

B.3 Requirements based testing

Requirements-based testing involves examining each requirement and developing a test or tests for it.

A.2 Release testing and system testing

B.1 Release testing is a form of system testing.

B.2 Important differences:

1. A separate team that has not been involved in the system development, should be responsible for release testing.
2. System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

7.4.2 Performance testing

A.1 Performance testing

1. Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
2. Tests should reflect the profile of use of the system.

3. Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
4. Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behaviour.

A.2 Stress testing

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

7.5 User testing

7.5.1 User testing

A.1 User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.

A.2 User testing is essential, even when comprehensive system and release testing have been carried out.

The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

7.5.2 Types of user testing

A.1 Alpha testing

Users of the software work with the development team to test the software at the developer's site.

A.2 Beta testing

A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.

A.3 Acceptance testing

Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

7.5.3 The acceptance testing process

A.1 Stages in the acceptance testing process

1. Define acceptance criteria
2. Plan acceptance testing
3. Derive acceptance tests
4. Run acceptance tests
5. Negotiate test results
6. Reject/accept system

7.6 Key points

1. Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.
2. Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers.
3. Development testing includes unit testing, in which you test individual objects and methods component testing in which you test related groups of objects and system testing, in which you test partial or complete systems.
4. When testing software, you should try to ‘break’ the software by using experience and guidelines to choose types of test case that have been effective in discovering defects in other systems.
5. Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.
6. Test-first development is an approach to development where tests are written before the code to be tested.
7. Scenario testing involves inventing a typical usage scenario and using this to derive test cases.
8. Acceptance testing is a user testing process where the aim is to decide if the software is good enough to be deployed and used in its operational environment.

7.7 Excercises(Homework): P232

8.2，8.4

Chapter 8 (9) Software Evolution

8.1 Topics covered

1. Evolution processes
2. Program evolution dynamics
3. Software maintenance
4. Legacy system management

8.2 Software Evolution

A.1 Software change

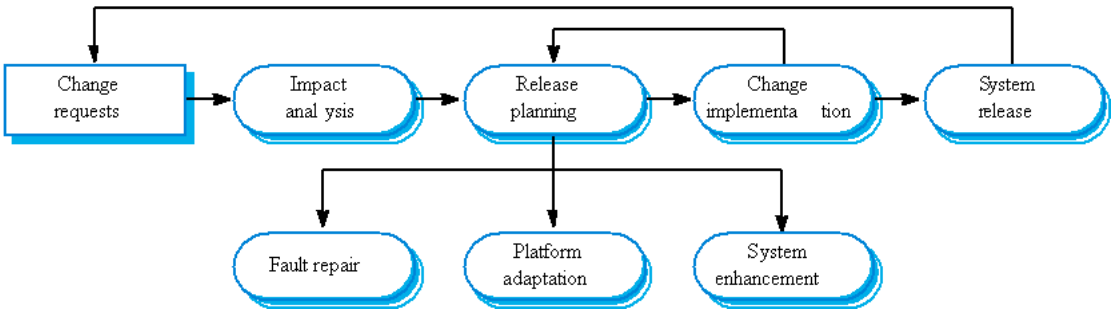
B.1 Software change is inevitable

1. New requirements emerge when the software is used;
2. The business environment changes;
3. Errors must be repaired;
4. New computers and equipment is added to the system;
5. The performance or reliability of the system may have to be improved.

B.2 A key problem for all organizations is implementing and managing change to their existing software systems.

8.3 Evolution processes

A.1 Evolution processes



A.2 Urgent change requests

B.1 Urgent changes may have to be implemented without going through all stages of the software engineering process

- If a serious system fault has to be repaired;
- If changes to the system's environment (e.g. an OS upgrade) have unexpected effects;
- If there are business changes that require a very rapid response (e.g. the release of a competing product).

The emergency repair process

8.4 Program evolution dynamics

A.1 Program evolution dynamics is the study of the processes of system change.

A.2 After several major empirical studies, Lehman and Belady proposed that there were a number of ‘laws’ which applied to all systems as they evolved.

A.3 There are sensible observations rather than laws. They are applicable to large systems developed by large organisations.

It is not clear if these are applicable to other types of software system.

Law	Description
Continuing change	A program that is used in a real-world environment must necessarily change, or else 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 is approximately invariant for each system release.
Organizational 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.
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will decline unless they are modified to reflect changes in their operational environment.
Feedback system	Evolution processes incorporate multiagent, multiloop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

8.5 Software maintenance

8.5.1 Software maintenance

1. Modifying a program after it has been put into use.
2. The term is mostly used for changing custom software. Generic software products are said to evolve to create new versions.
3. Maintenance does not normally involve major changes to the system's architecture.
4. Changes are implemented by modifying existing components and adding new components to the system.

A.1 Types of maintenance

B.1 Maintenance to repair software faults

Changing a system to correct deficiencies in the way meets its requirements.

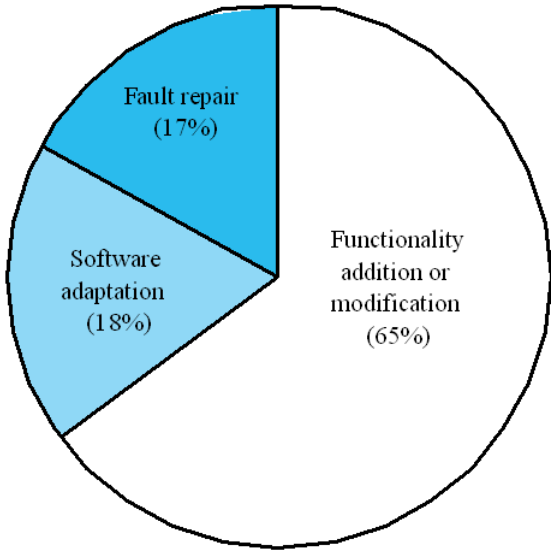
B.2 Maintenance to adapt software to a different operating environment

Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.

B.3 Maintenance to add to or modify the system's functionality

Modifying the system to satisfy new requirements.

A.2 Distribution of maintenance effort



A.3 Maintenance costs

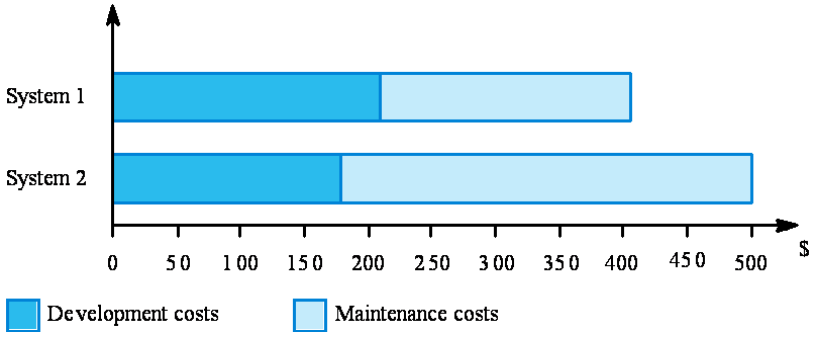
B.1 Usually greater than development costs (depending on the application).

B.2 Affected by both technical and non-technical factors.

B.3 Increases as software is maintained.

Maintenance corrupts the software structure so makes further maintenance more difficult.

B.4 Ageing software can have high support costs (e.g. old languages, compilers etc.).



A.4 Maintenance cost factors

B.1 Team stability

Maintenance costs are reduced if the same staff are involved with them for some time.

B.2 Contractual responsibility

The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.

B.3 Staff skills

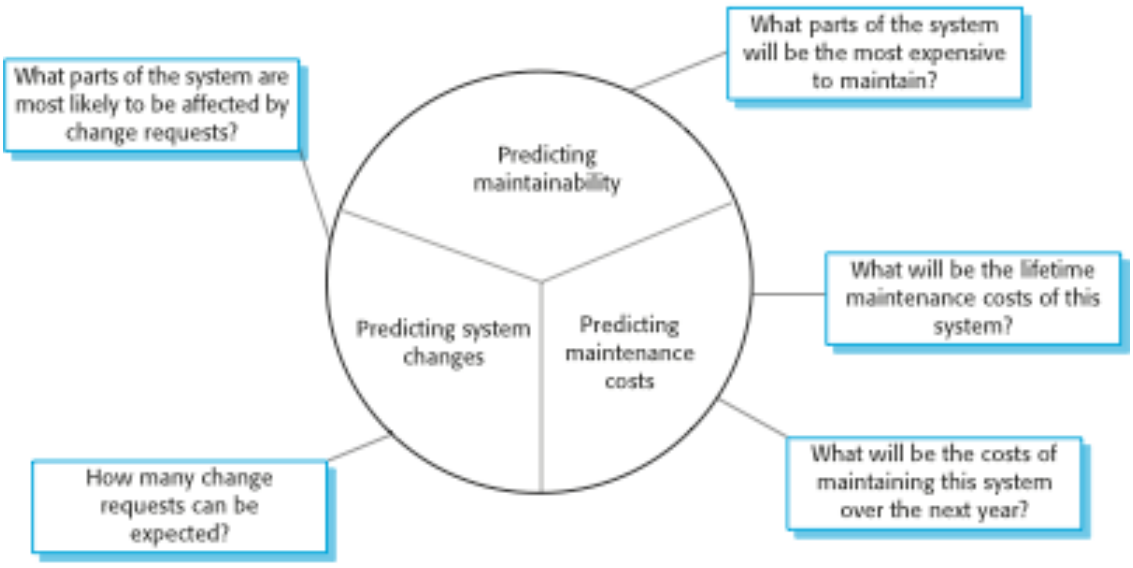
Maintenance staff are often inexperienced and have limited domain knowledge.

B.4 Program age and structure

As programs age, their structure is degraded and they become harder to understand and change.

8.5.2 Maintenance prediction

Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs



A.1 Complexity metrics

B.1 Predictions of maintainability can be made by assessing the complexity of system components.

B.2 Studies have shown that most maintenance effort is spent on a relatively small number of system components.

B.3 Complexity depends on

1. Complexity of control structures;
2. Complexity of data structures;
3. Object, method (procedure) and module size.

A.2 Process metrics

B.1 Process metrics may be used to assess maintainability

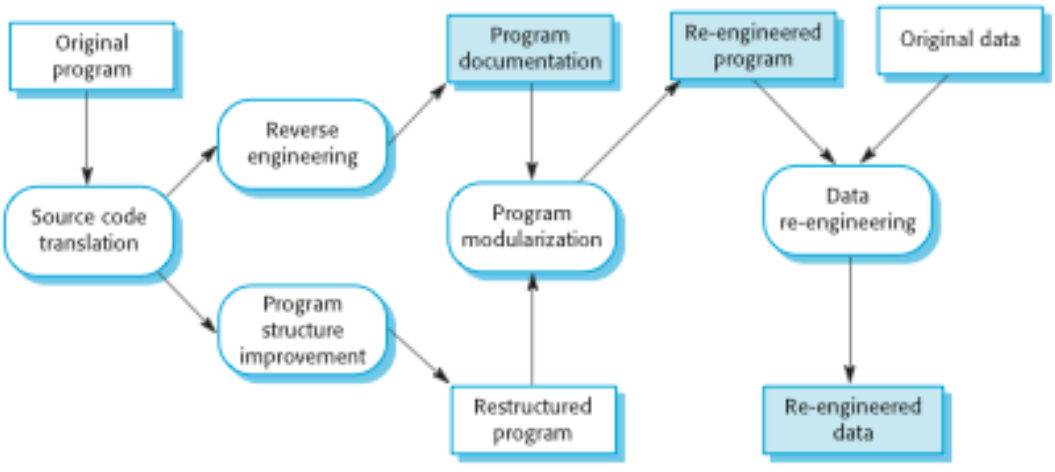
1. Number of requests for corrective maintenance;
2. Average time required for impact analysis;
3. Average time taken to implement a change request;
4. Number of outstanding change requests.

B.2 If any or all of these is increasing, this may indicate a decline in maintainability.

8.5.3 System re-engineering

1. Re-structuring or re-writing part or all of a legacy system without changing its functionality.
2. Applicable where some but not all sub-systems of a larger system require frequent maintenance.
3. Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented.

A.1 The reengineering process



B.1 Source code translation

Convert code to a new language.

B.2 Reverse engineering

Analyse the program to understand it;

B.3 Program structure improvement

Restructure automatically for understandability;

B.4 Program modularisation

Reorganise the program structure;

B.5 Data reengineering

Clean-up and restructure system data.

8.5.4 Preventative maintenance by refactoring

A.1 Preventative maintenance by refactoring

1. Refactoring is the process of making improvements to a program to slow down degradation through change.
2. You can think of refactoring as ‘preventative maintenance’ that reduces the problems of future change.
3. Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.
4. When you refactor a program, you should not add functionality but rather concentrate on program improvement.

A.2 Refactoring and reengineering

1. Re-engineering takes place after a system has been maintained for some time and maintenance costs are increasing. You use automated tools to process and re-engineer a legacy system to create a new system that is more maintainable.
2. Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

A.3 ‘Bad smells’ in program code

B.1 Duplicate code

The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.

B.2 Long methods

If a method is too long, it should be redesigned as a number of shorter methods.

B.3 Switch (case) statements

These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.

B.4 Data clumping v. 笨重地走

Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program. These can often be replaced with an object that encapsulates all of the data.

B.5 Speculative generality

This occurs when developers include generality in a program in case it is required in the future.

This can often simply be removed.

8.6 Legacy system management

8.6.1 Legacy system management

A.1 Organisations that rely on legacy systems must choose a strategy for evolving these systems

1. Scrap the system completely and modify business processes so that it is no longer required;
2. Continue maintaining the system;
3. Transform the system by re-engineering to improve its maintainability;
4. Replace the system with a new system.

A.2 The strategy chosen should depend on the system quality and its business value.

8.6.2 Legacy system categories

A.1 Low quality, low business value

These systems should be scrapped.

A.2 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.

A.3 High-quality, low-business value

Replace with COTS, scrap completely or maintain.

A.4 High-quality, high business value

Continue in operation using normal system maintenance.

8.7 Key points

1. Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
2. For custom systems, the costs of software maintenance usually exceed the software development costs.
3. The process of software evolution is driven by requests for changes and includes change impact analysis, release planning and change implementation.
4. Lehman’s laws, such as the notion that change is continuous, describe a number of insights derived from long-term studies of system evolution.
5. There are 3 types of software maintenance, namely bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.
6. Software re-engineering is concerned with re-structuring and re-documenting software to

make it easier to understand and change.

7. Refactoring, making program changes that preserve functionality, is a form of preventative maintenance.
8. The business value of a legacy system and the quality of the application should be assessed to help decide if a system should be replaced, transformed or maintained.

8.8 Excercises(Homework): P232

8.2，8.4

Chapter 9 (3) Agile Software Development

9.1 Topics covered

1. Agile methods
2. Plan-driven and agile development
3. Extreme programming
4. Agile project management
5. Scaling agile methods

9.2 Agile methods

9.2.1 aim of agile methods

A.1 Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:

1. Focus on the code rather than the design
2. Are based on an iterative approach to software development
3. Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.

A.2 The aim of agile methods is to reduce overheads in the software process (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.

9.2.2 The principles of agile methods

A.1 Customer involvement

Customers should be closely involved throughout the development process. Their role is provide

and prioritize new system requirements and to evaluate the iterations of the system.

A.2 Incremental delivery

The software is developed in increments with the customer specifying the requirements to be included in each increment.

A.3 People not process

The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive 规定的 processes.

A.4 Embrace change

Expect the system requirements to change and so design the system to accommodate these changes.

A.5 Maintain simplicity

Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

9.2.3 Agile method applicability

1. Product development where a software company is developing a small or medium-sized product for sale.
2. Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.
3. Because of their focus on small, tightly-integrated teams, there are problems in scaling agile methods to large systems

9.2.4 Problems with agile methods

1. It can be difficult to keep the interest of customers who are involved in the process.
2. Team members may be unsuited to the intense involvement that characterises agile methods.
3. Prioritising changes can be difficult where there are multiple stakeholders.
4. Maintaining simplicity requires extra work.
5. Contracts may be a problem as with other approaches to iterative development.

9.3 Plan-driven and agile development

9.3.1 Plan-driven development

1. A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
2. Not necessarily waterfall model – plan-driven, incremental development is possible
3. Iteration occurs within activities.

9.3.2 Agile development

Specification, design, implementation and testing are inter-leaved and the outputs from the

development process are decided through a process of negotiation during the software development process.

9.3.3 Technical, human, organizational issues

Most projects include elements of plan-driven and agile processes. Deciding on the balance depends on:

A.1 Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.

A.2 Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.

A.3 How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.

A.4 What is the expected system lifetime?

Long-lifetime systems may require more design documentation to communicate the original intentions of the system developers to the support team.

A.5 What technologies are available to support system development?

Agile methods rely on good tools to keep track of an evolving design

A.6 How is the development team organized?

If the development team is distributed or if part of the development is being outsourced, then you may need to develop design documents to communicate across the development teams.

A.7 Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.

A.8 How good are the designers and programmers in the development team?

It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code

A.9 Is the system subject to external regulation?

If a system has to be approved by an external regulator (e.g. the FAA approve software that is critical to the operation of an aircraft) then you will probably be required to produce detailed documentation as part of the system safety case.

9.4 Extreme programming

9.4.1 Introduction

Perhaps the best-known and most widely used agile method.

A.1 Extreme Programming (XP) takes an ‘extreme’ approach to iterative development.

1. New versions may be built several times per day;
2. Increments are delivered to customers every 2 weeks;
3. All tests must be run for every build and the build is only accepted if tests run successfully.

9.4.2 Extreme programming practices

A.1 Incremental planning

Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development ‘Tasks’.

A.2 Small releases

The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.

A.3 Simple design

Enough design is carried out to meet the current requirements and no more.

A.4 Test-first development

An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.

A.5 Refactoring

All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable

A.6 Pair programming

Developers work in pairs, checking each other’s work and providing the support to always do a good job.

A.7 Collective ownership

The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.

A.8 Continuous integration

As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.

A.9 Sustainable pace

Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity

A.10 On-site customer

A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

9.4.3 Pair programming

1. In XP, programmers work in pairs, sitting together to develop code.
2. This helps develop common ownership of code and spreads knowledge across the team.
3. It serves as an informal review process as each line of code is looked at by more than 1 person.
4. It encourages refactoring as the whole team can benefit from this.
5. Measurements suggest that development productivity with pair programming is similar to that of two people working independently.
6. In pair programming, programmers sit together at the same workstation to develop the software.
7. Pairs are created dynamically so that all team members work with each other during the development process.
8. The sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave.
9. Pair programming is not necessarily inefficient and there is evidence that a pair working together is more efficient than 2 programmers working separately.

9.5 Key points

1. Agile methods are incremental development methods that focus on rapid development, frequent releases of the software, reducing process overheads and producing high-quality code. They involve the customer directly in the development process.
2. The decision on whether to use an agile or a plan-driven approach to development should depend on the type of software being developed, the capabilities of the development team and the culture of the company developing the system.
3. Extreme programming is a well-known agile method that integrates a range of good programming practices such as frequent releases of the software, continuous software improvement and customer participation in the development team.

9.6 Excercises(Homework):