Atria Institute of Technology

Department of Information Science and Engineering

Bengaluru-560024

ACADEMIC YEAR: 2021-2022

ODD SEMESTER NOTES

Semester : 3rd Semester

Subject Name : Software Engineering

Subject Code : 18CS35

Faculty Name : Mrs. Asma Begum S

SOFTWARE ENGINEERING

1. INTRODUCTION, SOFTWARE PROCESSES, REQUIREMENTS ENGINEERING

INTRODUCTION

Software Crisis

- Software crisis is a term used in the early days of computing science for the difficulty of writinguseful and efficient computer programs in the required time.
- The software crisis was due to the rapid increases in computer power and the complexity ofthe problems that could be tackled.
- With the increase in the complexity of the software, many software problems arose because existingmethods were neither sufficient nor up to the mark.
- The causes of the software crisis were linked to the overall complexity of hardware and the software development process.
- The crisis manifested itself in several ways:
  - ✓ Projects running over-budget
  - ✓ Projects running over-time
  - ✓ Software was very inefficient
  - ✓ Software was of low quality
  - ✓ Software often did not meet requirements
  - ✓ Projects were unmanageable and code difficult to maintain
  - ✓ Software was never delivered

Need for Software Engineering

- The need of software engineering arises because of higher rate of change in user requirement and environment on which the software is working.
- The following factors contribute to the need of software engineering.
  - **Large Software:** As the size of software becomes large, engineering has to step to give it a scientific process.
  - **Scalability:** If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.
  - **Cost:** As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.
  - **Dynamic Nature:** The always growing and adapting nature of software hugely depends upon the environment in which user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.
  - **Quality Management:** Better process of software development provides better and quality software product.

Professional Software Development

- Software engineering is intended to support professional software development, rather thanindividual programming.
- It includes techniques that support program specification, design, and evolution, none of whichare normally relevant for personal software development.
- Software is not just the programs themselves but also all associated documentation and configuration data that is required to make these programs operate correctly.
- A professionally developed software system is often more than a single program.
- The system usually consists of a number of separate programs and configuration files that are used to set up these programs.
- It may include system documentation, which describes the structure of the system; user documentation, which explains how to use the system, and websites for users to download recent product information

→ Fig 1.1 gives frequently asked questions about software.

Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable, and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation, and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
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.
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times, and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs; 40% are testing costs. For custom software, evolution costs often exceed development costs.
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.
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.

Fig 1.1: Frequently asked questions about software

- There are two fundamental types of software product:

1. **Generic Products:** These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them. Examples of this type of product include software for PCs such as databases, word processors, drawing packages and project management tools.

- **Customized (or bespoke) Products:** These are systems which are commissioned by a particular customer. A software contractor develops the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process and air traffic control systems.Fig 1.2 gives the essential characteristics of a professional software system.

Product characteristics	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security, and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable, and compatible with other systems that they use.

Fig 1.2: Essential attributes of good software

Software Engineering

- 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.
- In this definition, there are two key phrases:
  - **Engineering discipline:** Engineers make things work. They apply theories, methods, and tools where these are appropriate. However, they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods
  - **All aspects of software production:** Software engineering is not just concerned with the technical processes of software development. It also includes activities such as software project management and the development of tools, methods, and theories to support software production.
- Software engineering is important for two reasons:
  1. More and more, individuals and society rely on advanced software systems. It helps users to produce reliable and trustworthy systems economically and quickly.
  2. It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems rather than just write the programs as if it was a personal programming project.
- There are four fundamental activities that are common to all software processes.These activities are:
  1. **Software Specification**, where customers and engineers define the software that is to be produced and the constraints on its operation.
  2. **Software Development**, where the software is designed and programmed.
  3. **Software Validation**, where the software is checked to ensure that it is what the customer requires.
  4. **Software Evolution**, where the software is modified to reflect changing customer and market requirements.
- Software engineering is related to both computer science and systems engineering:
  1. **Computer Science** is concerned with the theories and methods that underlie computers and software systems, whereas software engineering is concerned with the practical problems of producing software.
  2. **System Engineering** is concerned with all aspects of the development and evolution of complex systems where software plays a major role. System engineering is therefore concerned with hardware development, policy and process design and system deployment, as well as software engineering. System engineers are involved in specifying the system, defining its overall architecture, and then integrating the different parts to create the finished system. They are less concerned with the engineering of the system components (hardware, software, etc.)
- **There are three general issues that affect many different types of software**:
  1. **Heterogeneity:** Here it becomes necessary to integrate new software with older legacy systems written

in different programming languages. The challenge here is to develop techniques for building dependable software that is flexible enough to cope with this heterogeneity.

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.

3. **Security and Trust:** As software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote software systems accessed through a web page or web service interface.

- **Software Engineering Diversity**

There are many different types of application including:

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. Examples of such applications are office applications on a PC, CAD programs, photo manipulation software, etc.
2. **Interactive transaction-based applications:** These are applications that execute on a remote computer and that are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications where users can interact with a remote system to buy goods and services.
3. **Embedded control systems:** These are software control systems that control and manage hardware devices. Examples of embedded systems include the software in a mobile (cell) phone, software that controls anti-lock braking in a car, and software in a microwave oven to control the cooking process.
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. Examples of batch systems include periodic billing systems, such as phone billing systems, and salary payment systems.
5. **Entertainment systems:** These are systems that are primarily for personal use and which are intended to entertain the user. Most of these systems are games of one kind or another. The quality of the user interaction offered is the most important distinguishing characteristic of entertainment systems.
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. These are often computationally intensive and require high-performance parallel systems for execution.
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. The software has to interact with sensors and often is installed in a hostile environment such as inside an engine or in a remote location.
8. **Systems of systems:** These are systems that are composed of a number of other software systems. Some of these may be generic software products, such as a spreadsheet program. Other systems in the assembly may be specially written for that environment.

- **There are software engineering fundamentals that apply to all types of software system:**

- They should be developed using a managed and understood development process. The organization developing the software should plan the development process and have clear ideas of what will be produced and when it will be completed.
- Dependability and performance are important for all types of systems. Software should behave as expected, without failures and should be available for use when it is required. It should be safe in its operation and, as far as possible, should be secure against external attack. The system should perform efficiently and should not waste resources.
- Understanding and managing the software specification and requirements are important. It is important to understand what different customers and users of the system expect from it and then it is needed to manage their expectations so that a useful system can be delivered within budget and toschedule.
- Existing resources must be used efficiently. This means that, where appropriate, you should reuse software that has already been developed rather than write new software.

Software engineering and the Web

- The next stage in the development of web-based systems was the notion of web services.
- Web services are software components that deliver specific, useful functionality and which are accessed over the Web. - Applications are constructed by integrating these web services, whichmay be provided by different companies.
- In principle, this linking can be dynamic so that an application may use different web services each time that it is executed.
- Changes in the software organization, led to changes in the ways that web-based systems are engineered. For example:
  - Software reuse has become 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.
  - It is now generally recognized that it is impractical to specify all the requirements for such systems in advance. Web-based systems should be developed and delivered incrementally.
  - User interfaces are constrained by the capabilities of web browsers. Web forms with local scripting are more commonly used. Application interfaces on web-based systems are often poorer than the specially designed user interfaces on PC system products.

Software Engineering Ethics

- Some professional responsibilities includes:

- **Confidentiality:** You should normally respect the confidentiality of your employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
- **Competence:** You should not misrepresent your level of competence. You should not knowingly accept

work that is outside your competence.

- **Intellectual Property Rights:** You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.
- **Computer Misuse:** You should not use your technical skills to misuse other people's computers.

Software Engineering Code of Ethics and Professional Practice

ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. PUBLIC – Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER – Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT – Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT – Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT – Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION – Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES – Software engineers shall be fair to and supportive of their colleagues.
8. SELF – Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Fig 1.3: ACM/IEEE Code of Ethics

Case Studies

- 3 types of systems used as case studies are:

1. An Embedded System:

* This is a system where the software controls a hardware device and is embedded in that device.
* Issues in embedded systems typically include physical size, responsiveness, power management, etc.
* The example of an embedded system used here is a software system to control a medical device.

2. An Information System:

* This is a system whose primary purpose is to manage and provide access to a database of information.
* Issues in information systems include security, usability, privacy, and maintaining data integrity.
* The example of an information system t used here is a medical records system.

3. A Sensor-based Data Collection System:

* This is a system whose primary purpose is to collect data from a set of sensors and process thatdata in some way.
* The key requirements of such systems are reliability, even in hostile environmental conditions,and maintainability.

The example of a data collection system used here is a wilderness weather station.

An insulin pump control system

- An insulin pump is a medical system that simulates the operation of the pancreas.
- The software controlling this system is an embedded system, which collects information from asensor and controls a pump that delivers a controlled dose of insulin to a user.
- Diabetes is a relatively common condition where the human pancreas is unable to producesufficient quantities of a hormone called insulin.
- Insulin metabolizes glucose (sugar) in the blood.
- A software-controlled insulin delivery system might work by using a micro sensor embedded in thepatient to measure some blood parameter that is proportional to the sugar level.
- This is then sent to the pump controller.
- This controller computes the sugar level and the amount of insulin that is needed. It then sendssignals to a miniaturized pump to deliver the insulin via a permanently attached needle.
- Fig 1.4 shows the insulin pump hardware.
- Fig 1.5 is a UML activity model that illustrates how the software transforms an input blood sugarlevel to a sequence of commands that drive the insulin pump.
- Two essential high-level requirements that this system must meet includes:
    * The system shall be available to deliver insulin when required.

* The system shall perform reliably and deliver the correct amount of insulin to counteractthe
*
*
* current level of blood sugar.

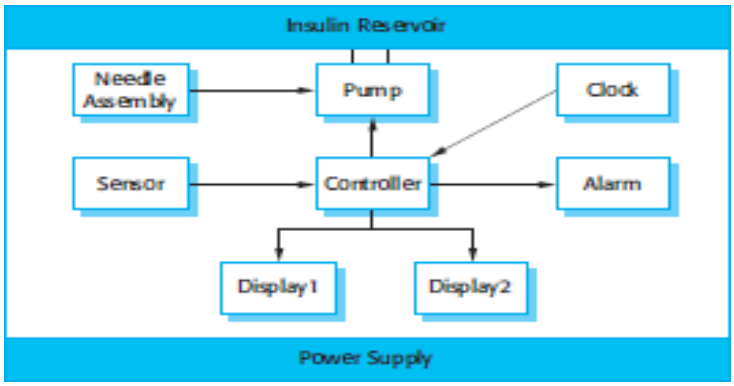


Fig 1.4: Insulin pump hardware

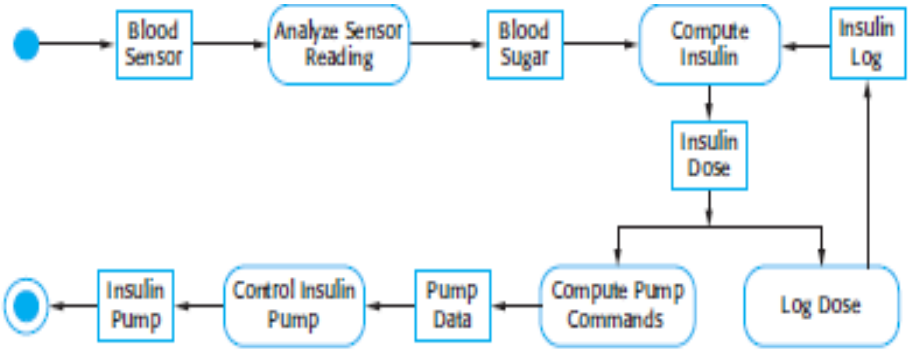


Fig 1.5: Activity model of the insulin pump

A patient information system for mental health care

- A patient information system to support mental health care is a medical information system that maintains information about patients suffering from mental health problems and the treatments that they have received.
- Most mental health patients do not require dedicated hospital treatment but need to attend specialist clinics regularly where they can meet a doctor who has detailed knowledge of their problems.

- To make it easier for patients to attend, these clinics are not just run in hospitals.
- They may also be held in local medical practices or community centers.

- The MHC-PMS (Mental Health Care-Patient Management System) is an information system that is intended for use in clinics. It makes use of a centralized database of patient information but has also been designed to run on a PC, so that it may be accessed and used from sites that do not have secure network connectivity.
- When the local systems have secure network access, they use patient information in the database but they can download and use local copies of patient records when they are disconnected.
- The system is not a complete medical records system so does not maintain information about other medical conditions.

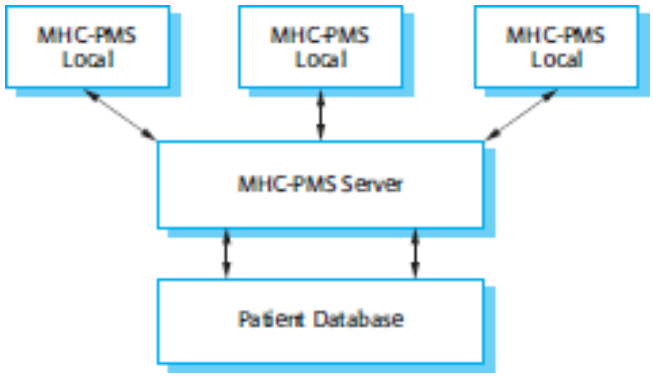


Fig 1.6: The organization of the MHC-PMS

- The MHC-PMS has two overall goals:
  - To generate management information that allows health service managers to assessperformance against local and government targets.
  - To provide medical staff with timely information to support the treatment of patients.
- The system is used to record information about patients (name, address, age, next of kin, etc.), consultations (date, doctor seen, subjective impressions of the patient, etc.), conditions, and treatments.
- Reports are generated at regular intervals for medical staff and health authority managers

- The key features of the system are:

1. **Individual care Management:** Clinicians can create records for patients, edit the information in the system, view patient history, etc.
2. **Patient Monitoring:** The system regularly monitors the records of patients that are involved in treatment and issues warnings if possible problems are detected.
3. **Administrative Reporting:** The system generates monthly management reports showing the number of patients treated at each clinic, the number of patients who have entered and left the care system, number of patients sectioned, the drugs prescribed and their costs, etc.

A wilderness weather station

- Wilderness weather stations are part of a larger system (Figure 1.5.4), which is a weather information system that collects data from weather stations and makes it available to other systems for processing.
  - The systems in Fig 1.7 are:
    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.
    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 in a form that can be retrieved by other systems, such as weather forecasting systems.
    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. It can update the embedded software in these systems. In the event of system problems, this system can also be used to remotely control a wilderness weather system
- Each weather station is battery-powered and must be entirely self-contained—there are no external power or network cables available.
- All communications are through a relatively slow-speed satellite link and the weather station must include some mechanism (solar or wind power) to charge its batteries.
- As they are deployed in wilderness areas, they are exposed to severe environmental conditions and may be damaged by animals.
- The station software is therefore not just concerned with data collection.
- It must also:
  - Monitor the instruments, power, and communication hardware and report faults to the management system.
  - 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.

* Allow for dynamic reconfiguration where parts of the software are replaced with new versionsand where backup instruments are switched into the system in the event of system failure.

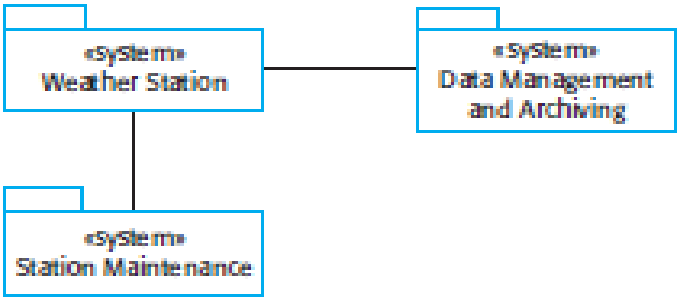


Fig 1.7: The weather station's environment

SOFTWARE PROCESSES

- A software process is a set of related activities that leads to the production of a software product.
- These activities may involve the development of software from scratch in a standard programming language like Java or C.
- 4 activities that are fundamental to software engineering:

1. **Software Specification:** The functionality of the software and constraints on its operation must bedefined.
2. **Software Design and Implementation:** The software to meet the specification must be produced
3. **Software Validation:** The software must be validated to ensure that it does what the customerwants.
4. **Software Evolution:** The software must evolve to meet changing customer needs.

- **Process descriptions may also include:**
  1. **Products**, which are the outcomes of a process activity. For example, the outcome of theactivity of architectural design may be a model of the software architecture.
  2. **Roles**, which reflect the responsibilities of the people involved in the process. Examples of rolesare project manager, configuration manager, programmer, etc.
  3. **Pre and post-conditions**, which are statements that are true before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a pre-condition may be that all requirements have been approved by the customer; after this activity is finished, a post-condition might be that the UML models describing the architecture have been reviewed

Models

- A software process model is a simplified representation of a software process.
- Each process model represents a process from a particular perspective, and thus provides onlypartial information about that process.

The Waterfall Model

- Because of the cascade from one phase to another, this model is known as the 'waterfall model'or software life cycle.
- The waterfall model is an example of a plan- driven process.
- The principal stages of the waterfall model [Fig 1.8] directly reflect the fundamental development activities:
  1. **Requirements analysis and definition:** The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
  2. **System and software design:** The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.
  3. **Implementation and unit testing:** During this stage, the software design is realized as a set of

programs or program units. Unit testing involves verifying that each unit meets its specification.

4. **Integration and system testing:** The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

5. **Operation and maintenance:** This is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units andenhancing the system's services as new requirements are discovered.

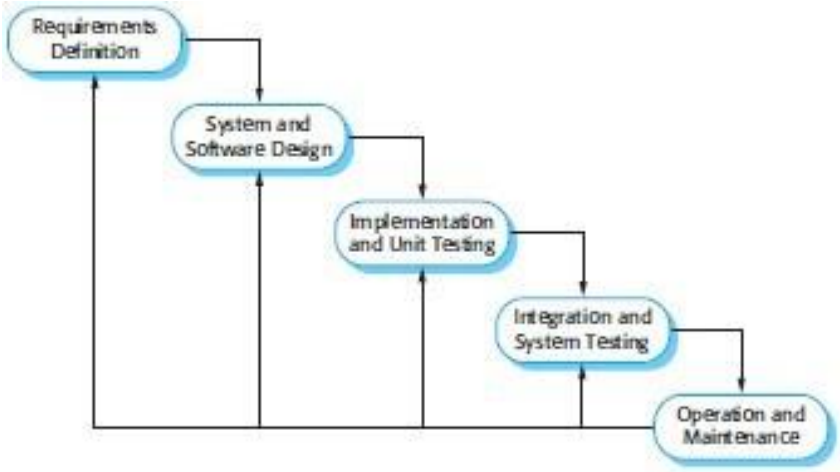


Fig 1.8: The Waterfall Model

- The waterfall model is consistent with other engineering process models and documentation is produced at each phase.
- This makes the process visible so managers can monitor progress against the development plan.
- Its major problem is the inflexible partitioning of the project into distinct stages.
- Commitments must be made at an early stage in the process, which makes it difficult to respond to changing customer requirements

Incremental Development

- Incremental development is based on the idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed [Fig 1.9].
- Incremental development has three important benefits, compared to the waterfall model:
  - 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.
  - It is easier to get customer feedback on the development work that has been done.
  - More rapid delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included.
- From a management perspective, the incremental approach has two problems:

- 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.
- 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.

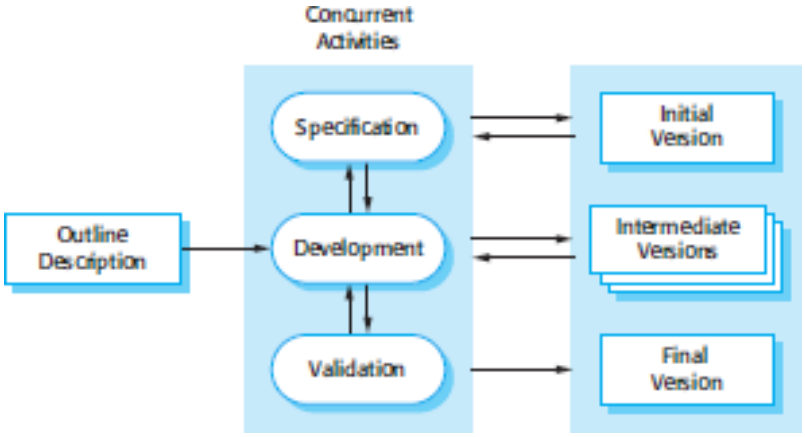


Fig 1.9: Incremental Development

Boehm's Spiral Model

→ Here, the software process is represented as a spiral, rather than a sequence of activities with some backtracking from one activity to another [Fig 1.10].

→ Each loop in the spiral represents a phase of the software process.

→ Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on.

→ Each loop in the spiral is split into four sectors:

1. **Objective Setting:** Specific objectives for that phase of the project are defined.Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified.
2. **Risk Assessment and Reduction:** For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

3. **Development and Validation:** After risk evaluation, a development model for the system is chosen. For example, throwaway prototyping may be the best development approach if user interface risks are dominant.
4. **Planning:** The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

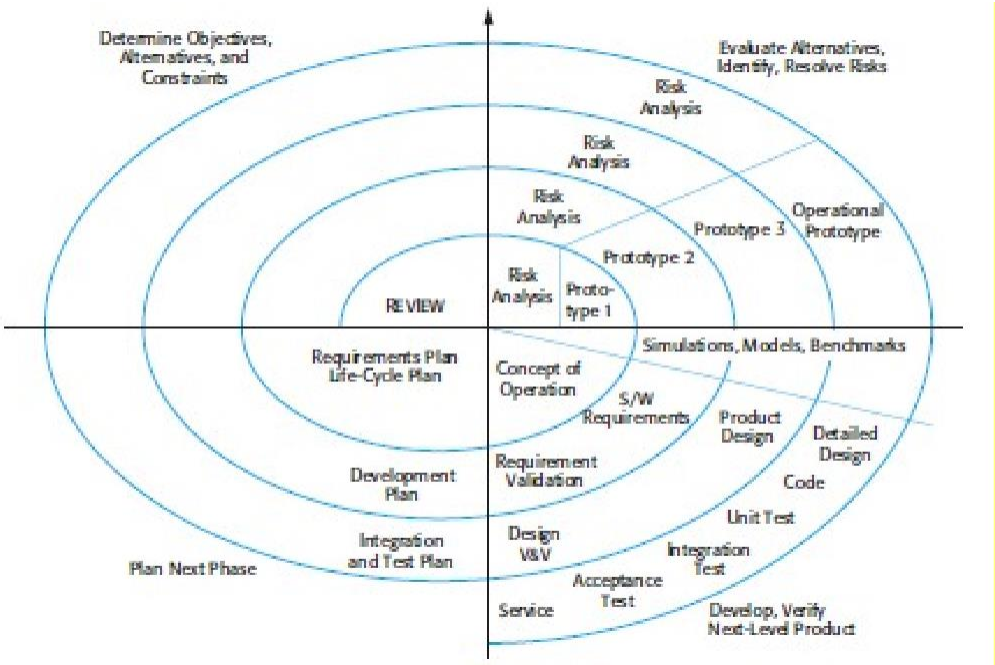


Fig 1.10: Boehm's spiral model of the software process

Process Activities

- The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes

Software Specification

- It is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development.
- The requirements engineering process aims (Fig 1.11) to produce an agreed requirements document that specifies a system satisfying stakeholder requirements.
- Requirements are usually presented at two levels of detail.
- End-users and customers need a high-level statement of the requirements; system developersneed a more detailed system specification.

- There are four main activities in the requirements engineering process:

1. Feasibility Study:

* An estimate is made of whether the identified user needs may be satisfied using currentsoftware and hardware technologies.

* The study considers whether the proposed system will be cost-effective from a business point of view and if it can be developed within existing budgetary constraints.

2. Requirements Elicitation and Analysis*:*

* Process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on.
* May involve the development of one or more system models and prototypes.

3. Requirements Specification:

* It is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements.
* Two types of requirements may be included in this document.
* User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to beprovided.

4. Requirements Validation:

* This activity checks the requirements for realism, consistency, and completeness.
* During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

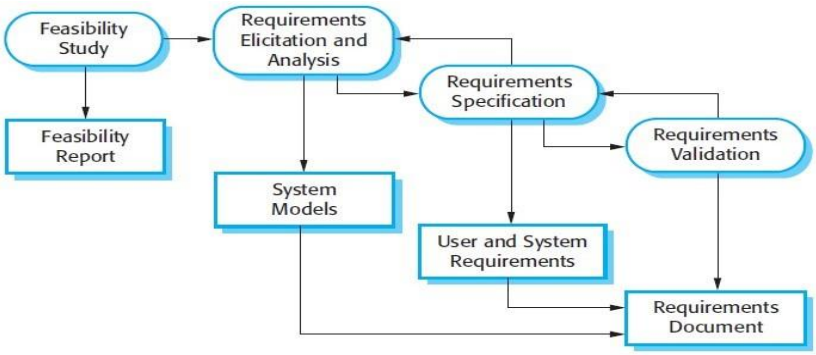


Fig 1.11: The Requirements Engineering Process

Software Design and Implementation

- The implementation stage of software development is the process of converting a system specification into an executable system.
- A software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used .
- Fig 1.12 shows an abstract model of this process showing the inputs to the design process, process activities, and the documents produced as outputs from this process.
- It shows four activities that may be part of the design process for information systems:

5. 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.

6. **Interface design**, where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.
7. **Component design**, where you take each system component and design how it will operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. The design model may be used to automatically generate an implementation.
8. **Database design**, where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.

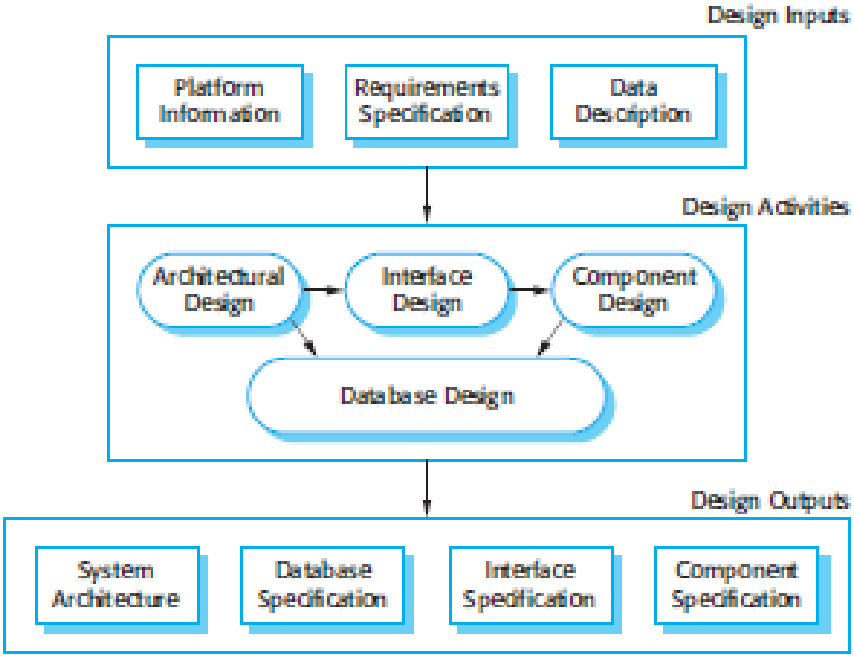


Fig 1.12: A general model of the design process

Software Validation

- Software validation or, more generally, verification and validation (V&V) is intended to show that asystem both conforms to its specification and that it meets the expectations of the system customer.
- Program testing, where the system is executed using simulated test data, is the principal validation technique.
- Validation may also involve checking processes, such as inspections and reviews, at each stage of thesoftware process from user requirements definition to program development.
- Fig 1.13 shows a three-stage testing process in which system components are tested then theintegrated system is tested and, finally, the system is tested with the customer's data.

- The stages in the testing process are:

1. Development Testing:

* The components making up the system are tested by the people developing the system.
* Each component is tested independently, without other system components.
* Components may be simple entities such as functions or object classes, or may be coherent groupings of these entities.

2. System Testing:

* System components are integrated to create a complete system.
* This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems.
* It is also concerned with showing that the system meets its functional and non-functionalrequirements, and testing the emergent system properties.

3. Acceptance Testing:

* This is the final stage in the testing process before the system is accepted for operational use.
* The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data.

- Fig 1.14 illustrates how test plans are the link between testing and development activities.
- This is sometimes called the V-model of development.
- Acceptance testing is sometimes called **'alpha testing'.**
- Custom systems are developed for a single client.
- The alpha testing process continues until the system developer and the client agree that the delivered system is an acceptable implementation of the requirements.
- When a system is to be marketed as a software product, a testing process called **'beta testing'** isoften used.
- Beta testing involves delivering a system to a number of potential customers who agree to usethat system. They report problems to the system developers.

Fig 1.13: Stages of testing

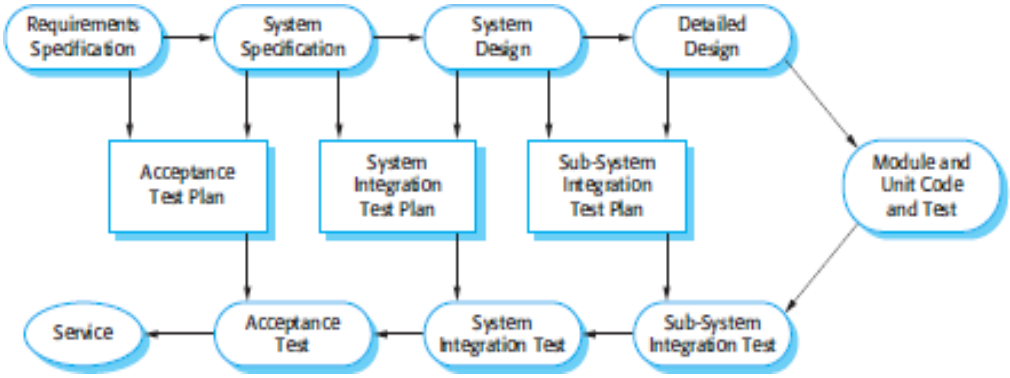


Fig 1.14: Testing phases in a plan-driven software process

Software Evolution

→ The flexibility of software systems is one of the main reasons why more and more software isbeing incorporated in large, complex systems.

→ Once a decision has been made to manufacture hardware, it is very expensive to make changes to thehardware design.

→ However, changes can be made to software at any time during or after the system development.

→ Even extensive changes are still much cheaper than corresponding changes to system hardware.

→ The fig 1.15 shows that software is continually changed over its lifetime in response to changingrequirements and customer needs

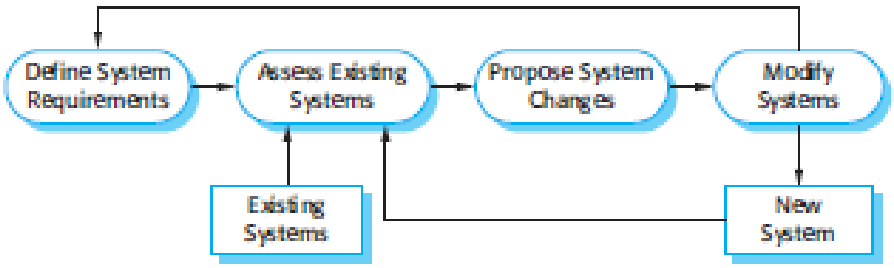


Fig 1.15: System Evolution

REQUIREMENTS ENGINEERING

Requirements Engineering Processes

→ Requirements engineering processes may include four high-level activities.

→ These focus on assessing if the system is useful to the business (feasibility study), discovering requirements (elicitation and analysis), converting these requirements into some standard form (specification), and checking that the requirements actually define the system that the customer wants (validation).

→ Fig 1.16 below shows this interleaving. The activities are organized as an iterative process around aspiral, with the output being a system requirements document.

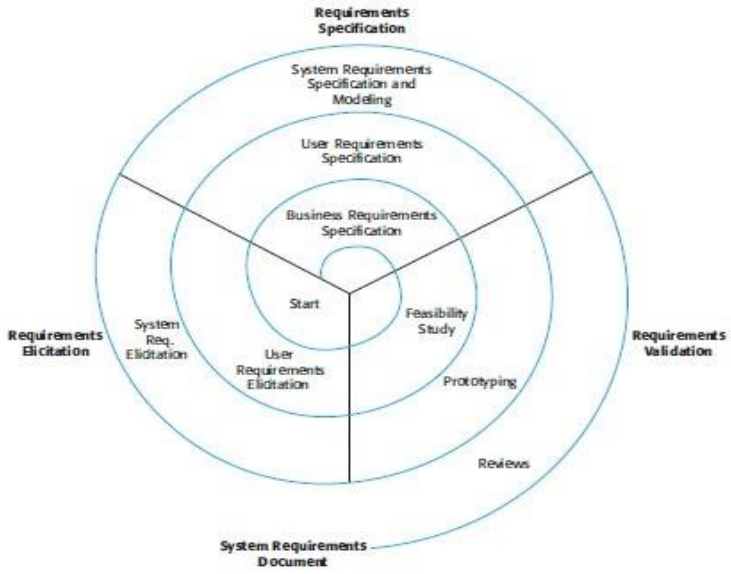


Fig 1.16: A spiral view of the requirements engineering process.

→ The amount of time and effort devoted to each activity in each iteration depends on the stage ofthe overall process and the type of system being developed.

→ This spiral model accommodates approaches to development where the requirements are developed to different levels of detail.

→ The number of iterations around the spiral can vary so the spiral can be exited after some or allof the user requirements have been elicited

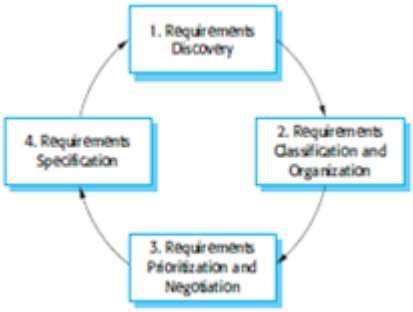
Requirements Elicitation and Analysis

→ After an initial feasibility study, the next stage of the requirements engineering process is requirements elicitation and analysis.

→ In this activity, software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.

→ A process model of the elicitation and analysis process is shown in fig 1.17.

Fig 1.17: The requirements elicitation and analysis process



→ The process activities are:

1. **Requirements discovery**: This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity.
2. **Requirements classification and organization:** This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub- systems and to associate requirements with each sub-system.
3. **Requirements prioritization and negotiation**: This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.
4. **Requirements specification**: The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced.

→ Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

* Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.
* Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers,

  without experience in the customer's domain, may not understand these requirements.
* Different stakeholders have different requirements and they may express these in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
* Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence inthe organization.
* The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

Requirements Discovery

→ Requirements discovery (sometime called requirements elicitation) is the process of gathering information about the required system and existing systems, and distilling the user and system requirements from this information.

→ For example, system stakeholders for the mental healthcare patient information system include:

* Patients whose information is recorded in the system.
* Doctors who are responsible for assessing and treating patients.
* Nurses who coordinate the consultations with doctors and administer some treatments.
* Medical receptionists who manage patients' appointments.
* IT staff who are responsible for installing and maintaining the system. These different requirements sources (stakeholders, domain, systems) can all be represented as system viewpoints with each viewpoint showing a subset of the requirements for the system.

Interviewing

→ In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed.

→ Interviews may be of two types:

1. Closed interviews, where the stakeholder answers a pre-defined set of questions.
2. Open interviews, in which there is no pre-defined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develops a better understanding of their needs.

→ It can be difficult to elicit domain knowledge through interviews for two reasons:

1. All application specialists use terminology and jargon that are specific to a domain. It is impossible for them to discuss domain requirements without using this terminology.
2. Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning. For example, for a librarian, it goes without saying that all acquisitions are catalogued before they are added to the library. However, this may not be obvious to the interviewer, and so it isn't taken into account in the requirements.

→ Effective interviewers have two characteristics:

1. They are open-minded, avoid pre-conceived ideas about the requirements, and are willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then they are willing to change their mind about the system.
2. They prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system. They find it much

easier to talk in a defined context rather than in general terms

Scenarios

→ They are descriptions of example interaction sessions.

→ Each scenario usually covers one or a small number of possible interactions.

→ Different forms of scenarios are developed and they provide different types of information atdifferent levels of detail about the system.

→ At its most general, a scenario may include:

* A description of what the system and users expects when the scenario starts.
* A description of the normal flow of events in the scenario.
* A description of what can go wrong and how this is handled.
* Information about other activities that might be going on at the same time.
* A description of the system state when the scenario finishes.

Use Cases

- A use case identifies the actors involved in an interaction and names the type of interaction.
- Use cases are documented using a high-level use case diagram.
- The set of use cases represents all of the possible interactions that will be described in the system requirements.
- Actors in the process, who may be human or other systems, are represented as stick figures.
- Each class of interaction is represented as a named ellipse.
- Lines link the actors with the interaction.
- Arrowheads may be added to lines to show how the interaction is initiated.
- Each use case should be documented with a textual description. These can then be linked to other models in the UML that will develop the scenario in more detail.
- For example, a brief description of the Setup Consultation use case from fig 1.18 below might be: ***Setup consultation allows two or more doctors, working in different offices, to view the same record at the same time. One doctor initiates the consultation by choosing the people involved from a drop-down menu of doctors who are online.***

The patient record is then displayed on their screens but only the initiating doctor can edit the record. In addition, a text chat window is created to help coordinate actions. It is assumed that a phone conference for voice communication will be separately set up.

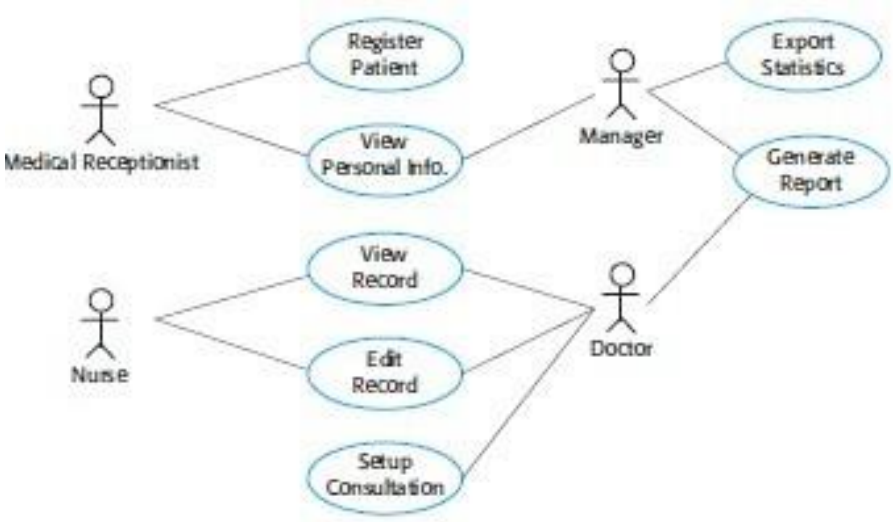


Fig 1.18 Use cases for the MHC-PMS

Ethnography

- Ethnography is an observational technique that can be used to understand operational processesand help derive support requirements for these processes.
- The value of ethnography is that it helps discover implicit system requirements that reflect theactual ways that people work, rather than the formal processes defined by the organization.

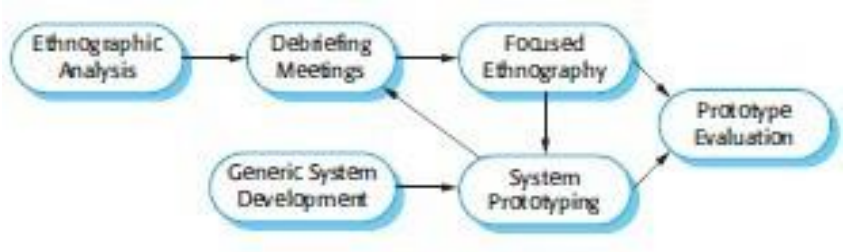


Fig 1.19: Ethnography and Prototyping for Requirements Analysis

→ Ethnography is particularly effective for discovering two types of requirements:

1. Requirements that are derived from the way in which people actually work, ratherthan the way in which process definitions say they ought to work
2. Requirements that are derived from cooperation and awareness of other people's activities.

→ Ethnography can be combined with prototyping as shown in fig 1.19.

Functional and Non functional Requirements

→ Software system requirements are often classified as functional requirements or non-functional requirements:

1. **Functional requirements:** These are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations.
2. **Non-functional requirements***:* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards.

Functional Requirements

→ The functional requirements for a system describe what the system should do.

→ These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements.

→ When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users.

→ Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems.

→ Examples for functional requirements for MHC-PMS system includes:

* A user shall be able to search the appointments lists for all clinics.
* The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
* Each staff member using the system shall be uniquely identified by his or her eight- digit employee number.

→ The functional requirements specification of a system should be both complete and consistent.

→ Completeness means that all services required by the user should be defined.

→ Consistency means that requirements should not have contradictory definitions.

Non-Functional Requirements

→ They are requirements that are not directly concerned with the specific services delivered by thesystem to its users.

→ They may relate to emergent system properties such as reliability, response time, and store occupancy.

→ Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole.

→ Non-functional requirements are often more critical than individual functional requirements

→ The implementation of these requirements may be diffused throughout the system.

There are two reasons for this:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components.
2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required. The figure below shows the classification of non- functional requirements

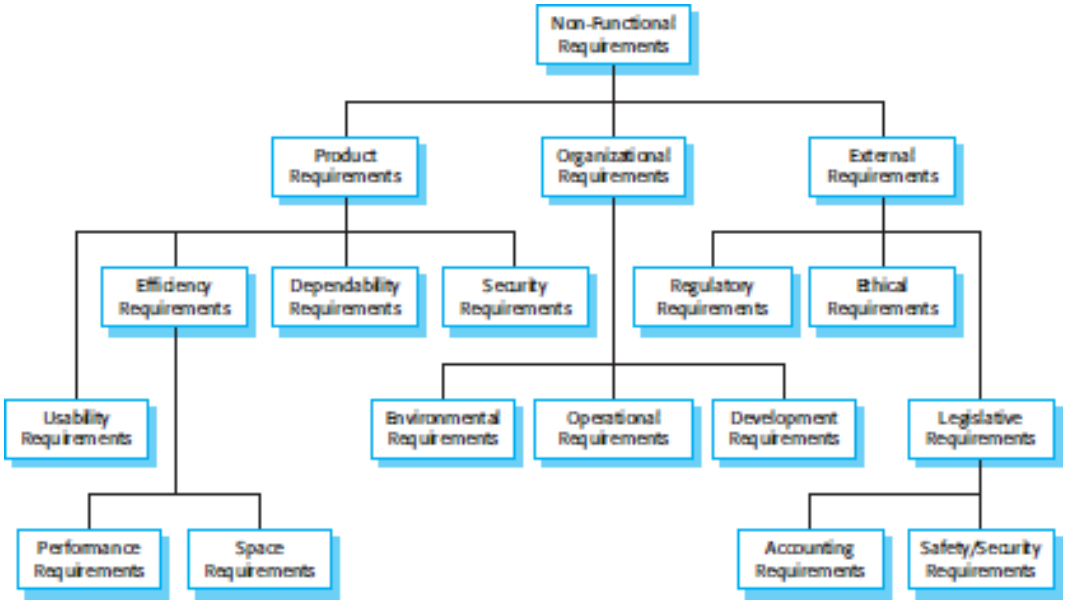


Fig 1.20: Types of Non-Functional Requirement

→ Fig 1.20 is a classification of non-functional requirements.

→ The various types include:

1. **Product Requirements*:***

* These requirements specify or constrain the behavior of the software.

* Examples include performance requirements on how fast the system must execute and how much memory it requires, reliability requirements that set out the acceptable failure rate, security requirements, and usability requirements.

2. Organizational Requirements:

* These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organization.

* Examples include operational process requirements that define how the system will be used, development process requirements that specify the programming language, the development environment or process standards to be used, and environmental requirements that specify the operating environment of the system.

3. External requirements:

* This broad heading covers all requirements that are derived from factors external to the

system and its development process.

* These may include regulatory requirements that set out what must be done for the system tobe approved for use by a regulator, such as a central bank.

→ The fig 1.21 below shows the metric used for specifying non-functional requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes 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

Fig 1.21: Metrics for specifying non functional requirements

The Software Requirements Document

→ The software requirements document (sometimes called the software requirements specificationor SRS) is an official statement of what the system developers should

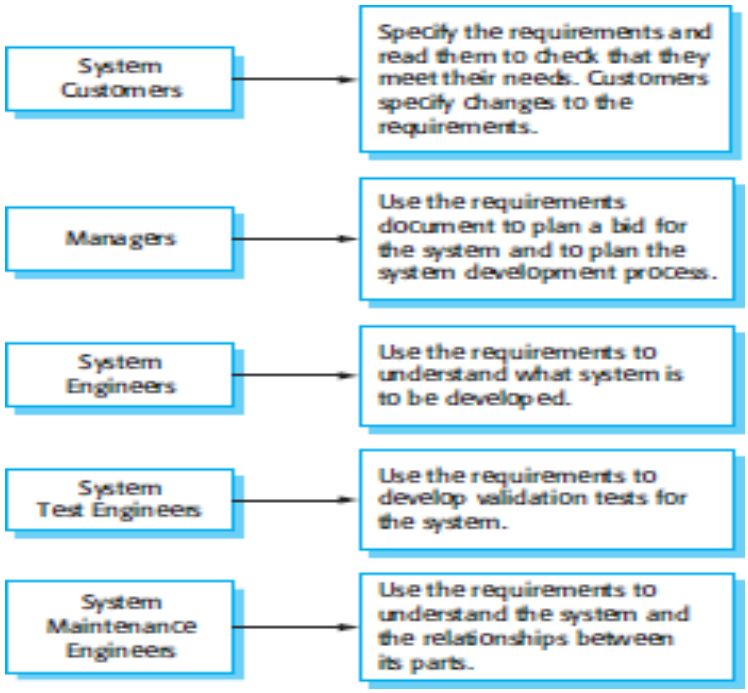
→ It should include both the user requirements for a system and a detailed specification of thesystem requirements.

→ The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software.

→ The users of requirements document is as shown below in fig 1.22.

Fig 1.22: Users of a Requirement Document

→ Figure 1.23 shows one possible organization for a requirements document that is based on an



IEEE standard for requirements documents.

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The non-functional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Fig 1.23: The Structure of a Requirements Document

Requirements Specification

- Requirements specification is the process of writing down the user and system requirements ina requirements document.
- System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design.
- They add detail and explain how the user requirements should be provided by the system.

- It is practically impossible to exclude all design information. There are several reasons for this:
- You may have to design an initial architecture of the system to help structure the requirements specification.
- The system requirements are organized according to the different sub-systems that make upthe system
- In most cases, systems must interoperate with existing systems, which constrain the designand impose requirements on the new system.
- The use of a specific architecture to satisfy non-functional requirements may be necessary.

The fig 1.24 below shows the different ways of writing system requirement specification.

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

Fig 1.24 Ways of writing a system Requirements specification

Natural Language Specification

- To minimize misunderstandings when writing natural language requirements, there are somesimple guidelines to be followed:

1. Invent a standard format and ensure that all requirement definitions adhere to that format.
2. Use language consistently to distinguish between mandatory and desirable requirements.
3. Use text highlighting (bold, italic, or color) to pick out key parts of the requirement.
4. Do not assume that readers understand technical software engineering language. It is easy forwords like ‘architecture’ and ‘module’ to be misunderstood. You should, therefore, avoid the use of abbreviations, and acronyms.
5. Whenever possible, you should try to associate a rationale with each user requirement.

→ Fig 1.25 illustrates how these guidelines may be used. It includes two requirements for theembedded software for the automated insulin pump

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

Fig 1.25: Example requirements for the insulin pump software system

Structured Specifications

→ Structured natural language is a way of writing system requirements where the freedom of therequirements writer is limited and all requirements are written in a standard way.

→ Structured language notations use templates to specify system requirements.

→ An example of a form-based specification, that defines how to calculate the dose of insulin tobe delivered when the blood sugar is within a safe band, as shown in fig 1.26.

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.

Fig 1.26: A structured specification of a requirement for an insulin pump

→ When a standard form is used for specifying functional requirements, the followinginformation should be included:

* A description of the function or entity being specified.
* A description of its inputs and where these come from.
* A description of its outputs and where these go to.
* Information about the information that is needed for the computation or other entitiesin the system that are used (the 'requires' part).
* A description of the action to be taken.
* If a functional approach is used, a pre-condition setting out what must be true beforethe function is called, and a post-condition specifying what is true after the function is

called.

* A description of the side effects (if any) of the operation.

Requirements Validation

→ Requirements validation is the process of checking that requirements actually define thesystem that the customer really wants.

→ It overlaps with analysis as it is concerned with finding problems with the requirements.

→ During the requirements validation process, different types of checks should be carriedout on the requirements in the requirements document.

⇒These checks include:

1. **Validity Checks*:*** A user may think that a system is needed to perform certain functions.
2. **Consistency Checks:** Requirements in the document should not conflict. That is, there should notbe contradictory constraints or different descriptions of the same system function.
3. **Completeness Checks:** The requirements document should include requirements that define all functions and the constraints intended by the system user.
4. **Realism Checks:** Using knowledge of existing technology, the requirements should be checkedto ensure that they can actually be implemented.
5. **Verifiability:** To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should beable to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

⇒There are a number of requirements validation techniques that can be usedindividually or in conjunction with one another:

1. **Requirements Reviews**: The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
2. **Prototyping**: In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meetstheir real needs.
3. **Test-Case Generation**: Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems.

Requirements Management

→ The requirements for large software systems are always changing.

→ Once a system has been installed and is regularly used, new requirements inevitably emerge.

→ There are several reasons why change is inevitable:

* The business and technical environment of the system always changes after installation. New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change
* The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for usersupport if the system is to meet its goals.
* Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.

Requirements Management Planning

- Planning is an essential first stage in the requirements management process. The planningstage establishes the level of requirements management detail that is required.
- During the requirements management stage, a decision is to be taken on:

1. Requirements Identification:

* Each requirement must be uniquely identified so that it can be cross- referenced with other requirements & used in traceability assessments.

2. A Change Management Process:

* This is the set of activities that assess the impact and cost of changes.

3. Traceability Policies:

* These policies define the relationships between each requirement and between the requirementsand the system design that should be recorded.
* The traceability policy should also define how these records should be maintained.

4. Tool Support:

* Requirements management involves the processing of large amounts of information about the requirements.
* Tools that may be usedrange from specialist requirements management systems to spreadsheets and simple database systems.
* Tool supports might be needed for:
  - ***a.*** ***Requirements Storage:*** The requirements should be maintained in a secure, managed datastore that is accessible to everyone involved in the requirements engineering process.
  - ***b.*** ***Change Management:*** The process of change management is simplified if active tool

support is available as shown in fig 1.27.

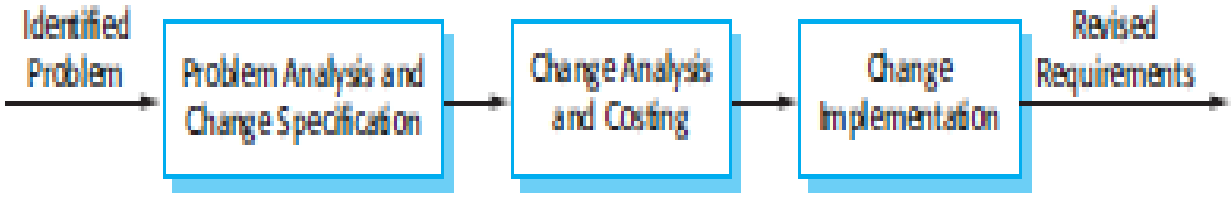


Fig 1.27: Requirements Change Management

c. ***Traceability Management:*** Tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

→ There are three principal stages to a change management process:

1. **Problem Analysis and Change Specification**:
   - The process starts with an identified requirements problem or, sometimes, with a specificchange proposal.
   - During this stage, the problem or the change proposal is analyzed to check that it is valid.
   - This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
2. **Change Analysis and Costing:**
   - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements.
3. **Change Implementation:**
   - The requirements document and, where necessary, the system design and implementation,are modified.
   - Requirements document will have to be organized so that changes can be made to it without extensive rewriting or reorganization.

SOFTWARE ENGINEERING [18CS35]

QUESTION BANK ON MODULE-I

1. What are the fundamental activities of a software Engineering?
2. With Neat diagram, explain the waterfall model of software development process.
3. With a diagram, explain the rational unified process.
4. What is requirement specification? Explain various ways of writing system requirements.
5. Why the understanding of requirements from stake holder is difficult task? Explain.
6. Explain the different Checks to be carried out during requirement validation process.
7. What is software? List the fundamental software engineering activities. Mention and explain the key challenges or the general issues facing software engineering.
8. List and explain any five software engineering code of ethics.
9. Write block diagram for illustrating incremental development model. State at least two benefits andthe problem in incremental development.
10. Explain functional, Non functional and domain requirements with at least one example for each.
11. Write the structure of requirement documents as suggested by IEEE standards.
12. List out all the stake holders' mental health cone patient management system (MHC-PMS). Write anote on interviewing stake holders for requirement discovery.
13. Explain briefly software engineering Ethics.
14. With neat block diagram explain the waterfall model.
15. Explain requirements engineering processes with suitable diagram.
16. With a neat diagram, explain insulin pump control system.
17. With neat diagram explain Boehm's spiral model.
18. Explain Ethnography in detail.

SOFTWARE ENGINEERING [18CS35]

ASSIGNMENT QUESTIONS ON MODULE-I

1. What are the fundamental activities of a software Engineering?
2. With Neat diagram, explain the waterfall model of software development process.
3. What is requirement specification? Explain various ways of writing system requirements.
4. What is software? List the fundamental software engineering activities. Mention and explain thekey challenges or the general issues facing software engineering.
5. List and explain any five software engineering code of ethics.
6. Write block diagram for illustrating incremental development model. State at least two benefits andthe problem in incremental development.
7. Explain functional, Non functional and domain requirements with at least one example for each
8. With neat block diagram explain the waterfall model.
9. Explain requirements engineering processes with suitable diagram.
10. With neat diagram explain Boehm's spiral model.

Module-3

System modeling

Objectives

The aim of this chapter is to introduce some types of system model that may be developed as part of the requirements engineering and system design processes. When you have read the chapter, you will: understand how graphical models can be used to represent software systems;_ understand why different types of model are required and the fundamental system modeling perspectives of context, interaction, structure, and behavior;
_ have been introduced to some of the diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling;_ be aware of the ideas underlying model-driven engineering, where a system is automatically generated from structural and behavioral models.

Contents

- Context models
- Interaction models
- Structural models
- Behavioral models
- Model-driven engineering

Introduction:

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.System modeling has generally come to mean representing the system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).
Models are used

- during the requirements engineering process to help derive the requirements for a system
- during the design process to describe the system to engineers implementing the system
- after implementation to document the system's structure and operation.

You may develop models of both the existing system and the system to be developed:
Models of the existing system are used during requirements engineering.

- They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system

Models of the new system are used during requirements engineering

- To help explain the proposed requirements to other system stakeholders.

- Engineers use these models to discuss design proposals and to document the system for implementation.

A model is an abstraction of a system being studied not an alternate representation.

Ideally, a representation of a system should maintain all the information about the entity being represented.

An abstraction deliberately simplifies and picks out the most salient characteristics.

You may develop different models to represent the system from different perspectives. For example:

- An external perspective: model the context or environment of the system.
- An interaction perspective:model the interactions between a system and its environment or between the components of a system.
- A structural perspective:model the organization of a system or the structure of the data that is processed by the system.
- A behavioural perspective:model the dynamic behaviour of the system and how it responds to events.

UML:

- Activity diagrams:shows the activities involved in a process or in data processing.
- Use case diagrams:shows the interactions between a system and its environment.
- Sequence diagrams:shows interactions between actors and the system and between system components.
- Class diagrams:shows the object classes in the system and the associations between these classes.
- State diagrams:shows how the system reacts to internal and external events.

Context models

At an early stage in the specification of a system, you should decide on the system boundaries. This involves working with system stakeholders to decide what functionality should be included in the system and what is provided by the system's environment. You may decide that automated support for some business processes should be implemented but others should be manual processes or supported by different systems. You should look at possible overlaps in functionality with existing systems and decide where new functionality should be implemented. These decisions should be made early in the process to limit the system costs and the time needed for understanding the system requirements and design.

In some cases, the boundary between a system and its environment is relatively clear. For example, where an automated system is replacing an existing manual or computerized system, the environment of the new system is usually the same as the existing system's environment. In other cases, there is more flexibility, and you decide what constitutes the boundary between the system and its environment during the requirements engineering process.

For example, say you are developing the specification for the patient information system for mental healthcare. This system is intended to manage information about patients attending mental health clinics and the treatments that have been prescribed. In developing the specification for this system, you have to decide whether the system should focus exclusively on collecting information about consultations (using other systems to collect personal information about patients) or whether it should also collect personal patient information. The advantage of relying on other systems for patient information is that you avoid duplicating data. The major disadvantage, however, is that using other systems may make it slower to access information. If these systems are unavailable, then the MHC-PMS cannot be used.

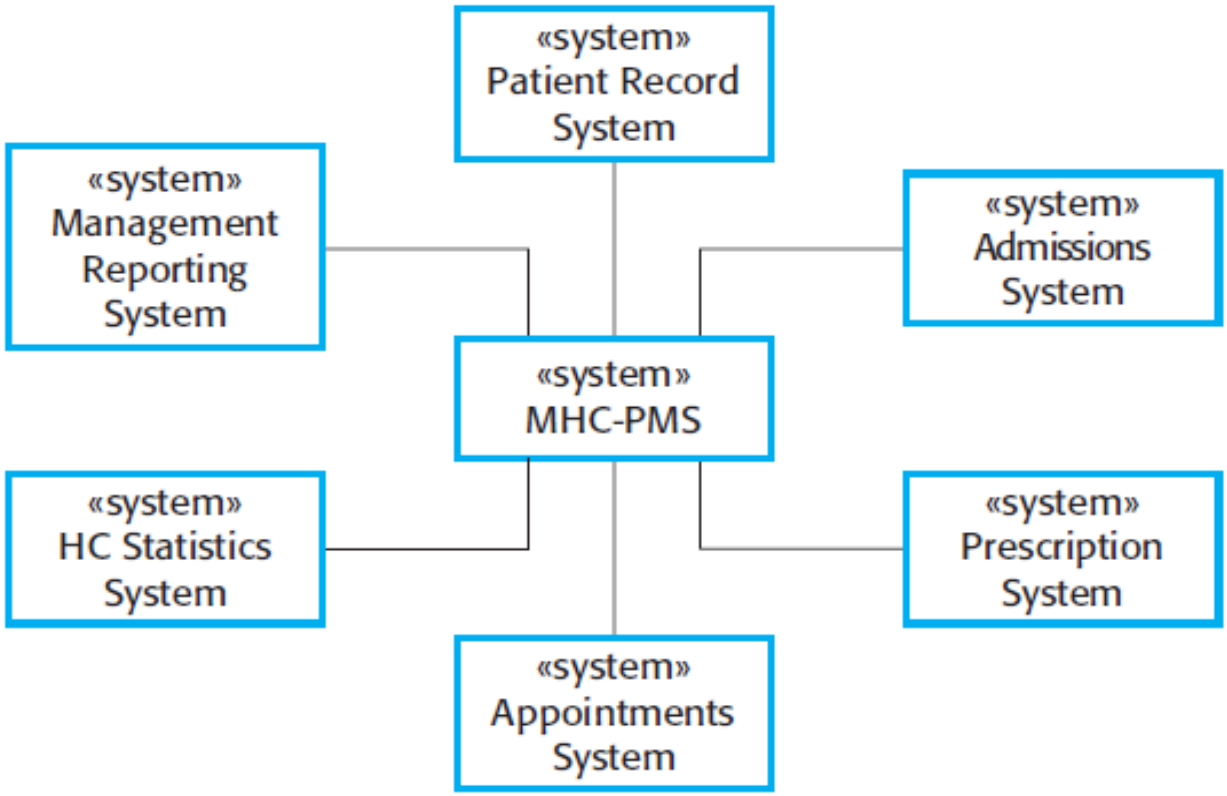


Figure 5.1 The context of the MHC-PMS

The definition of a system boundary is not a value-free judgment. Social and organizational concerns may mean that the position of a system boundary may be determined by non-technical factors. For example, a system boundary may be deliberately positioned so that the analysis process can all be carried out on one site; it may be chosen so that a particularly difficult manager need not be consulted; it may be positioned so that the system cost is increased and the system development division must therefore expand to design and implement the system. Once some decisions on the boundaries of the system have been made, part of the analysis activity is

the definition of that context and the dependencies that a system has on its environment. Normally, producing a simple architectural model is the first step in this activity. Figure 5.1 is a simple context model that shows the patient information system and the other systems in its environment. From Figure 5.1, you can see that the MHC-PMS is connected to an appointments system and a more general patient record system with which it shares data. The system is also connected to systems for management reporting and hospital bed allocation and a statistics system that collects information for research. Finally, it makes use of a prescription system to generate prescriptions for patients' medication. Context models normally show that the environment includes several other automated systems. However, they do not show the types of relationships between the systems in the environment and the system that is being specified. External systems might produce data for or consume data from the system. They might share data with the system, or they might be connected directly, through a network or not connected at all. They might be physically co-located or located in separate buildings. All of these relations may affect the requirements and design of the system being defined and must be taken into account. Therefore, simple context models are used along with other models, such as business process models. These describe human and automated processes in which particular software systems are used.

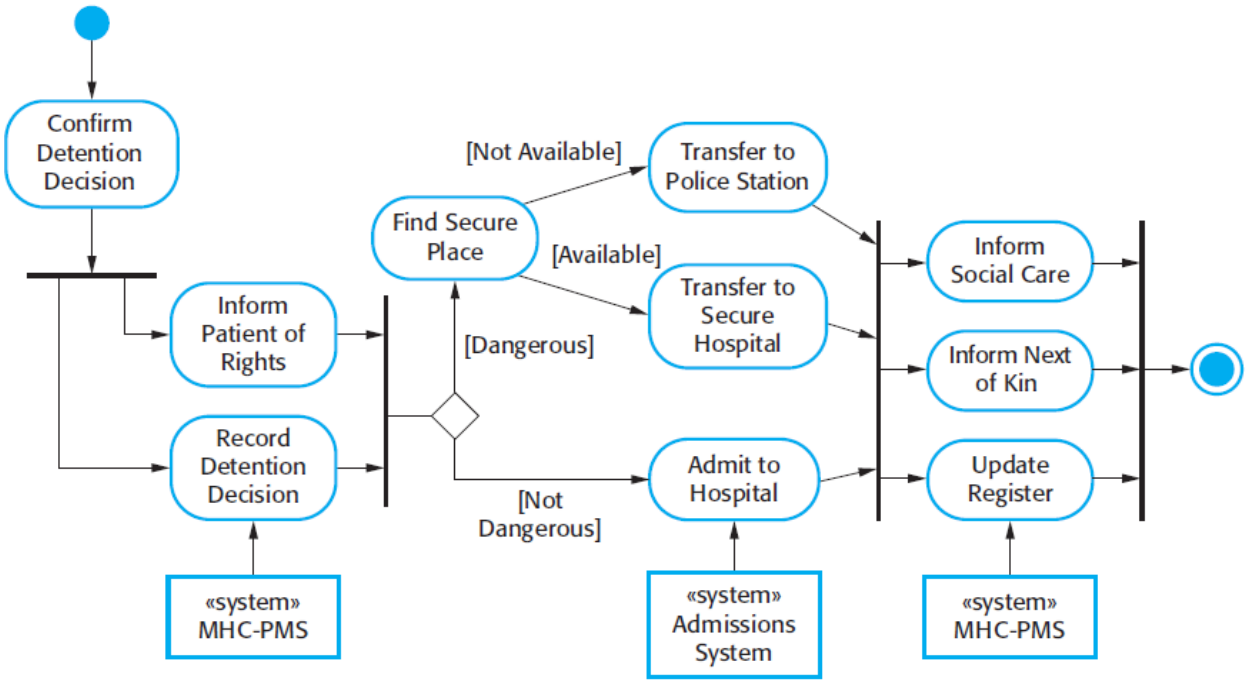


Figure 5.2 Process model of involuntary detention

Figure 5.2 is a model of an important system process that shows the processes in which the MHC-PMS is used. Sometimes, patients who are suffering from mental health problems may be a danger to others or to themselves. They may therefore have to be detained against their will in a hospital so that treatment can be administered. Such detention is subject to strict legal safeguards—for example, the decision to detain a patient must be regularly reviewed so that people are not held indefinitely without good reason. One of the functions of the MHC-PMS is to ensure that such safeguards are implemented.

Figure 5.2 is a UML activity diagram. Activity diagrams are intended to show the activities that make up a system process and the flow of control from one activity to another. The start of a process is indicated by a filled circle; the end by a filled circle inside another circle. Rectangles with round corners represent activities, that is, the specific sub-processes that must be carried out. You may include objects in activity charts. In Figure 5.2, I have shown the systems that are used to support different processes. I have indicated that these are separate systems using the UML stereotype feature. In a UML activity diagram, arrows represent the flow of work from one activity to another. A solid bar is used to indicate activity coordination. When the flow from more than one activity leads to a solid bar then all of these activities must be complete before progress is possible. When the flow from a solid bar leads to a number of activities, these may be executed in parallel. Therefore, in Figure 5.2, the activities to inform social care and the patient's next of kin, and to update the detention register may be concurrent.

Interaction models

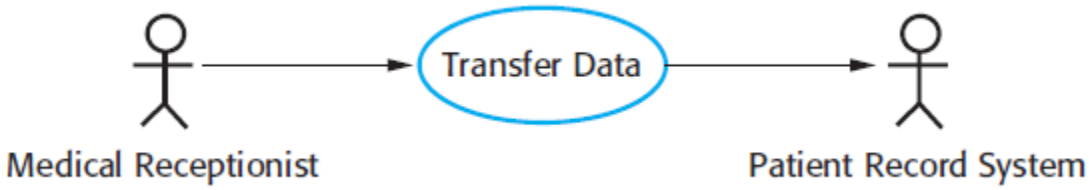
All systems involve interaction of some kind. This can be user interaction, which involves user inputs and outputs, interaction between the system being developed and other systems or interaction between the components of the system. Modeling user interaction is important as it helps to identify user requirements. Modeling system to system interaction highlights the communication problems that may arise. Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.

In this section, I cover two related approaches to interaction modeling:

1. Use case modeling, which is mostly used to model interactions between a system and external actors (users or other systems).
2. Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.

Use case modeling

Use case modeling was originally developed by Jacobson et al. (1993) in the 1990s and was incorporated into the first release of the UML (Rumbaugh et al., 1999). As I have discussed in Chapter 4, use case modeling is widely used to support requirements elicitation. A use case can be taken as a simple scenario that describes what a user expects from a system.



Notice that there are two actors in this use case: the operator who is transferring the data and the patient record system. The stick figure notation was originally developed to cover human interaction but it is also now used to represent other external systems and hardware. Formally, use case diagrams should use lines without arrows as arrows in the UML indicate the direction of

flow of messages. Obviously,in a use case messages pass in both directions. However, the arrows in Figure 5.3 are used informally to indicate that the medical receptionist initiates the transaction and data is transferred to the patient record system.

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.

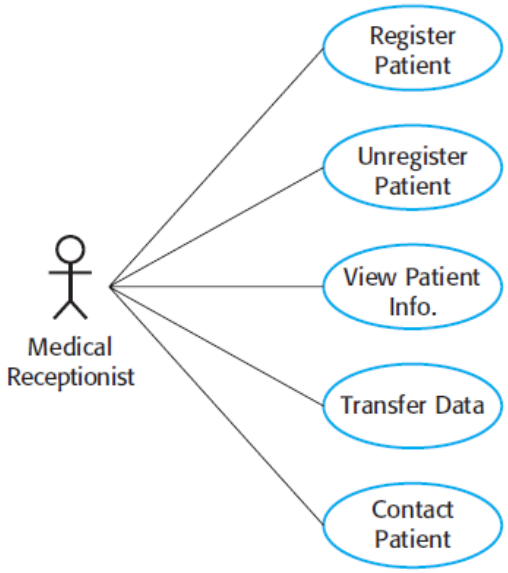


Figure 5.5 Use cases involving the role 'medical receptionist'

Sequence diagrams

Sequence diagrams in the UML are primarily used to model the interactions between the actors and the objects in a system and the interactions between the objects themselves. The UML has a rich syntax for sequence diagrams, which allows many different kinds of interaction to be

modeled. I don't have space to cover all possibilities here so I focus on the basics of this diagram type. As the name implies, a sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance. Figure 5.6 is an example of a sequence diagram that illustrates the basics of the notation. This diagram models the interactions involved in the View patient information use case, where a medical receptionist can see some patient information.

You can read Figure 5.6 as follows:
1. The medical receptionist triggers the ViewInfo method in an instance P of the PatientInfo object class, supplying the patient's identifier, PID. P is a user interface object, which isplayed as a form showing patient information.
2. The instance P calls the database to return the information required, supplying the receptionist's identifier to allow security checking (at this stage, we do not care where this UID comes from).
3. The database checks with an authorization system that the user is authorized for this action.
4. If authorized, the patient information is returned and a form on the user's screen is filled in. If authorization fails, then an error message is returned.

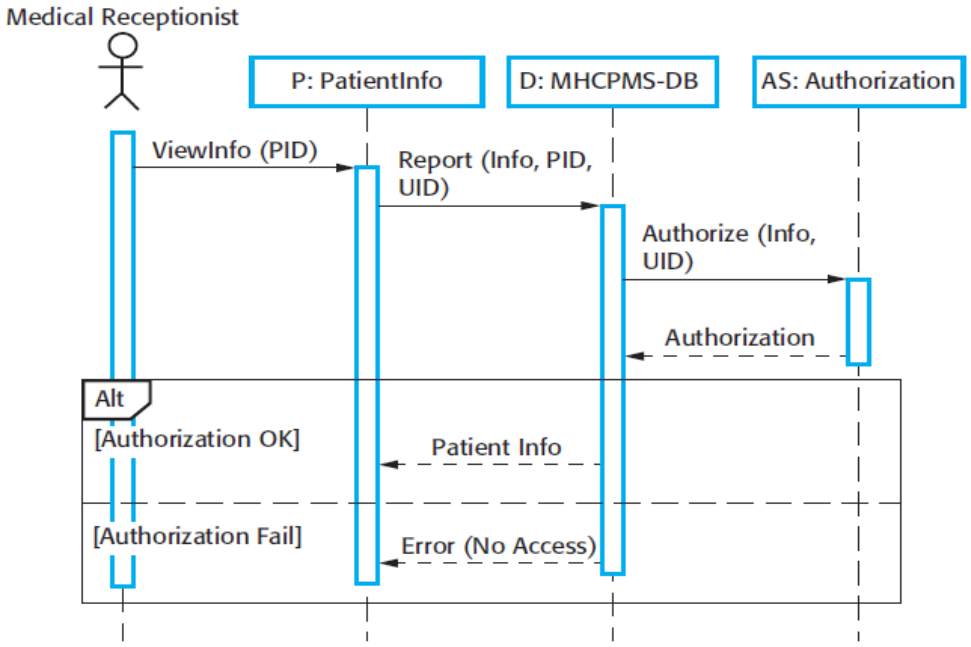


Figure 5.6 Sequence diagram for View patient information

Figure 5.7 is a second example of a sequence diagram from the same system that illustrates two additional features. These are the direct communication between the actors in the system and the creation of objects as part of a sequence of operations. In this example, an object of type Summary is created to hold the summary data that is to be uploaded to the PRS (patient record system).
You can read this diagram as follows:
1. The receptionist logs on to the PRS.
2. There are two options available. These allow the direct transfer of updated patient information to the PRS and the transfer of summary health data from the MHC-PMS to the PRS.

3. In each case, the receptionist's permissions are checked using the authorization system.
4. Personal information may be transferred directly from the user interface object to the PRS. Alternatively, a summary record may be created from the database and that record is then transferred.
5. On completion of the transfer, the PRS issues a status message and the user logs off.

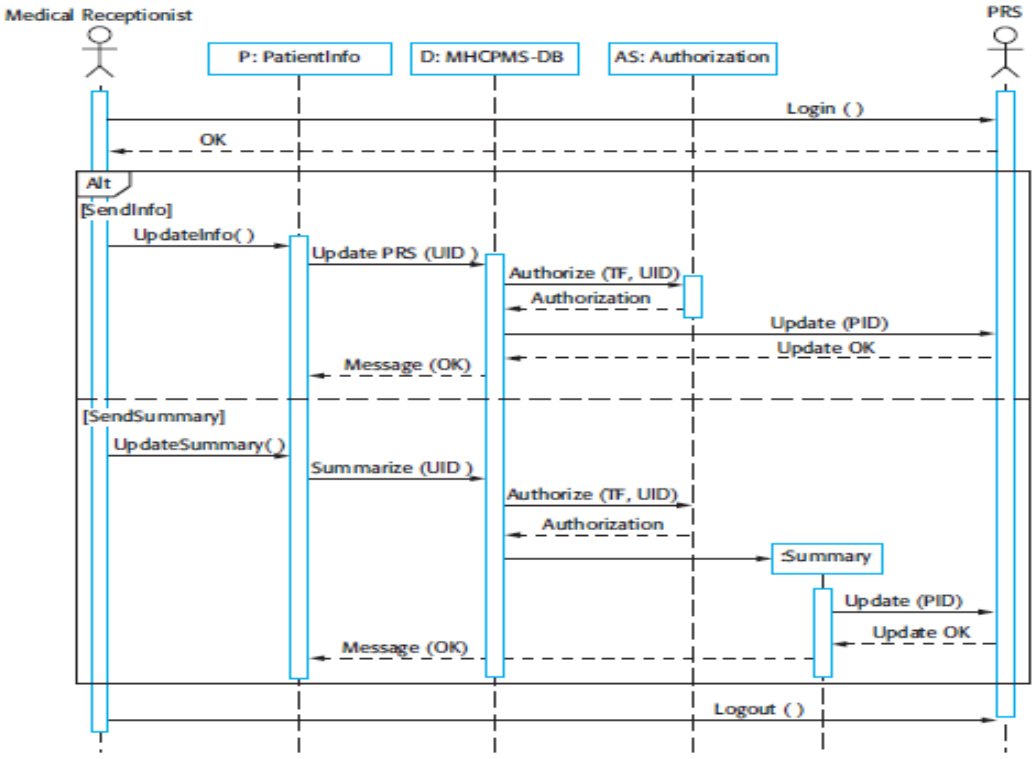


Figure 5.7 Sequence diagram for transfer data

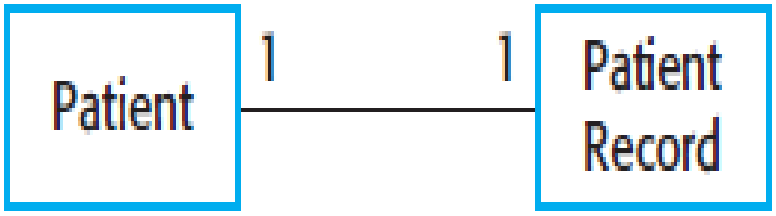
Structural models

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. These are not the same things—the dynamic organization of a system as a set of interacting threads may be very different from a static model of the system components.

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. Loosely, 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 a relationship between these classes. Consequently, each class may have to have some knowledge of its associated class. Class diagrams in the UML can be expressed at different levels of detail. When you are developing a model, the first stage is usually to look at

the world, identify the essential objects, and represent these as classes. The simplest way of writing these is to write the class name in a box. You can also simply note the existence of an association



In Figure 5.8, I illustrate a further feature of class diagrams—the ability to show how many objects are involved in the association. In this example, each end of the association is annotated with a 1, meaning that there is a 1:1 relationship between objects of these classes. That is, each patient has exactly one record and each record maintains information about exactly one patient. As you can see from later examples, other multiplicities are possible.

You can define that an exact number of objects are involved or, by using a *, as shown in Figure 5.9, that there are an indefinite number of objects involved in the association. Figure 5.9 develops this type of class diagram to show that objects of class Patient are also involved in relationships with a number of other classes. In this example, I show that you can name associations to give the reader an indication of the type of relationship that exists. The UML also allows the role of the objects participating in the association to be specified.

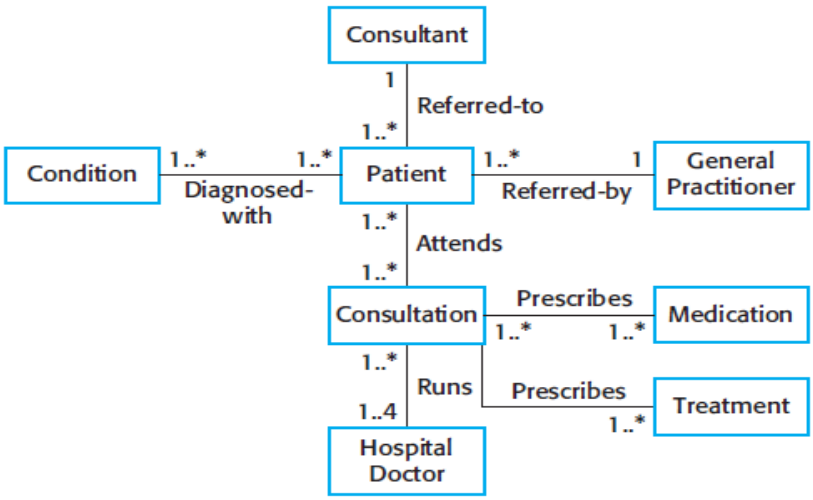


Figure 5.9 Classes and associations in the MHC-PMS

This is illustrated in Figure 5.10 where:
1. The name of the object class is in the top section.
2. The class attributes are in the middle section. This must include the attribute names and, optionally, their types.
3. The operations (called methods in Java and other OO programming languages) associated with the object class are in the lower section of the rectangle.

Figure 5.10 shows possible attributes and operations on the class Consultation. In this example, I assume that doctors record voice notes that are transcribed later to record details of the consultation. To prescribe medication, the doctor involved must use the Prescribe method to generate an electronic prescription.

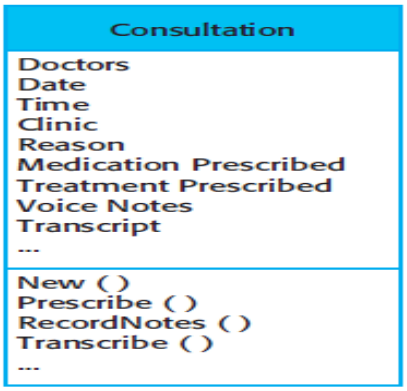


Figure 5.10 The consultation class

Generalization

Generalization is an everyday technique that we use to manage complexity. Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics ofthese classes. This allows us to infer that different members of these classes have some common characteristics (e.g., squirrels and rats are rodents). We can make general statements that apply to all class members (e.g., all rodents have teeth for gnawing). In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. This means that common information will be maintained in one place only. This is good design practice as it means that, if changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change. In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.

The UML has a specific type of association to denote generalization, as illustrated in Figure 5.11. The generalization is shown as an arrowhead pointing up to the more general class. This shows that general practitioners and hospital doctors can be generalized as doctors and that there are three types of Hospital Doctor—those that have just graduated from medical school and have to be supervised (Trainee Doctor); those that can work unsupervised as part of a consultant's team (Registered Doctor); and consultants, who are senior doctors with full decision-making responsibilities.

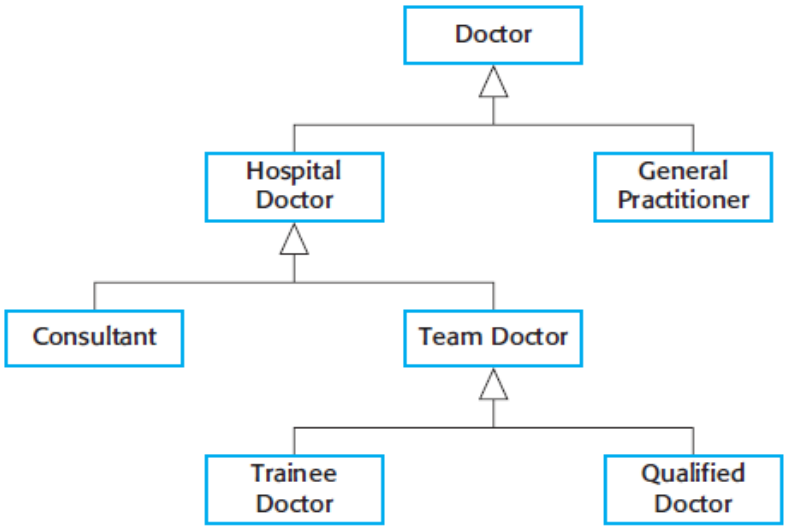


Figure 5.11 A generalization hierarchy

In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes. In essence, the lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations. For example, all doctors have a name and phone number; all hospital doctors have a staff number and a department but general practitioners don't have these attributes as they work independently. They do however, have a practice name and address. This is illustrated in Figure 5.12, which shows part of the generalization hierarchy that I have extended with class attributes. The operations associated with the class Doctor are intended to register and de-register that doctor with the MHC-PMS.

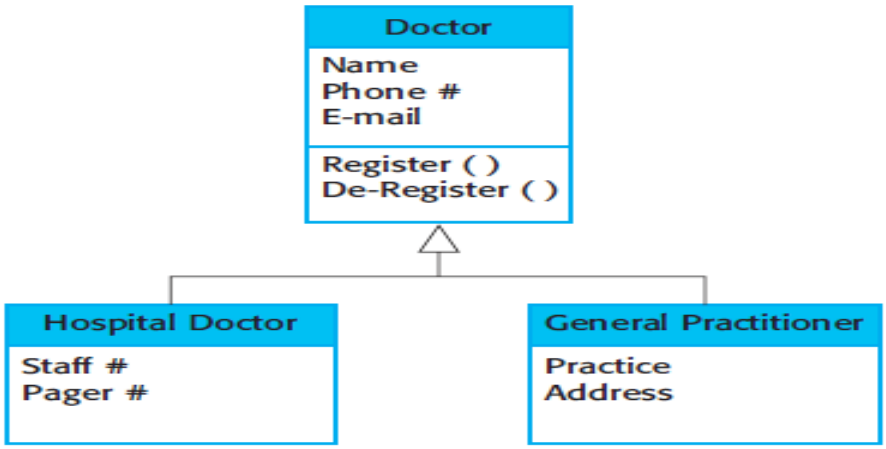


Figure 5.12 A generalization hierarchy with added detail

Aggregation

Objects in the real world are often composed of different parts. For example, a study pack for a course may be composed of a book, PowerPoint slides, quizzes, and recommendations for further reading. Sometimes in a system model, you need to illustrate this. The UML provides a special type of association between classes called aggregation that means that one object (the whole) is

composed of other objects (the parts). To show this, we use a diamond shape next to the class that represents the whole. This is shown in Figure 5.13, which shows that a patient record is a composition of Patient and an indefinite number of Consultations.

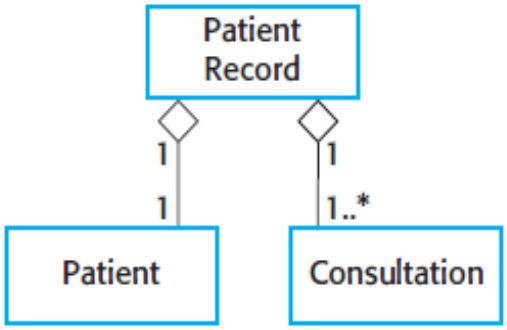


Figure 5.13 The aggregation association

Behavioral models

Behavioral models are models of the dynamic behavior of the 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.
You can think of these stimuli as being of two types:
1. *Data* Some data arrives that has to be processed by the system.
2. *Events* Some event happens that triggers system processing. Events may have associated data but this is not always the case.

Data-driven modeling

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. That is, they show the entire sequence of actions that take place from an input being processed to the corresponding output, which is the system's response. The UML does not support data-flow diagrams as they were originally proposed and used for modeling data processing. The reason for this is that DFDs focus on system functions and do not recognize system objects. However, because data-driven systems are so common in business, UML 2.0 introduced activity diagrams, which are similar to data-flow diagrams. For example, Figure 5.14 shows the chain of processing involved in the insulin pump software. In this diagram, you can see the processing steps (represented as activities) and the data flowing between these steps (represented as objects).

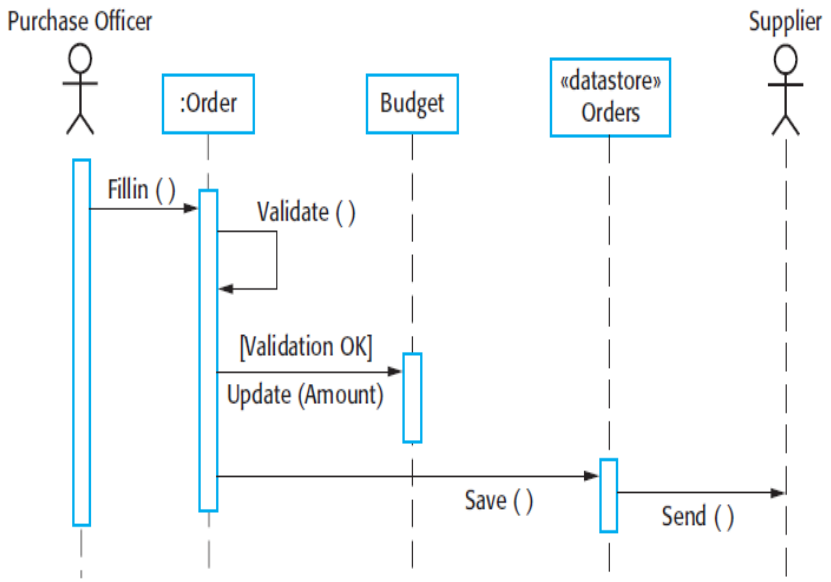


Figure 5.14 An activity Send ( ) model of the insulin pump's operation

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. For example, a system controlling a valve may move from a state 'Valve open' to a state 'Valve closed' when an operator command (the stimulus) is received.

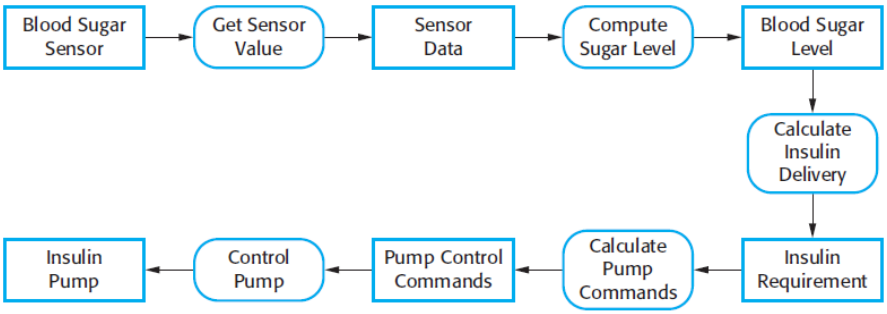


Figure 5.15 Order processing

The UML supports event-based modeling using state diagrams, which were based on Statecharts (Harel, 1987, 1988). State diagrams show system states and events that causetransitions from one state to another. They do not show the flow of data within the system but may include additional information on the computations carried out in each state. I use an example of control software for a very simple microwave oven to illustrate event-driven modeling. Real microwave ovens are actually much more complex than this system but the simplified system is easier to understand. This simple microwave has a switch to select full or half power, a numeric keypad to input the cooking time, a start/stop button, and an alphanumeric display.

I have assumed that the sequence of actions in using the microwave is:
1. Select the power level (either half power or full power).
2. Input the cooking time using a numeric keypad.
3. Press Start and the food is cooked for the given time.

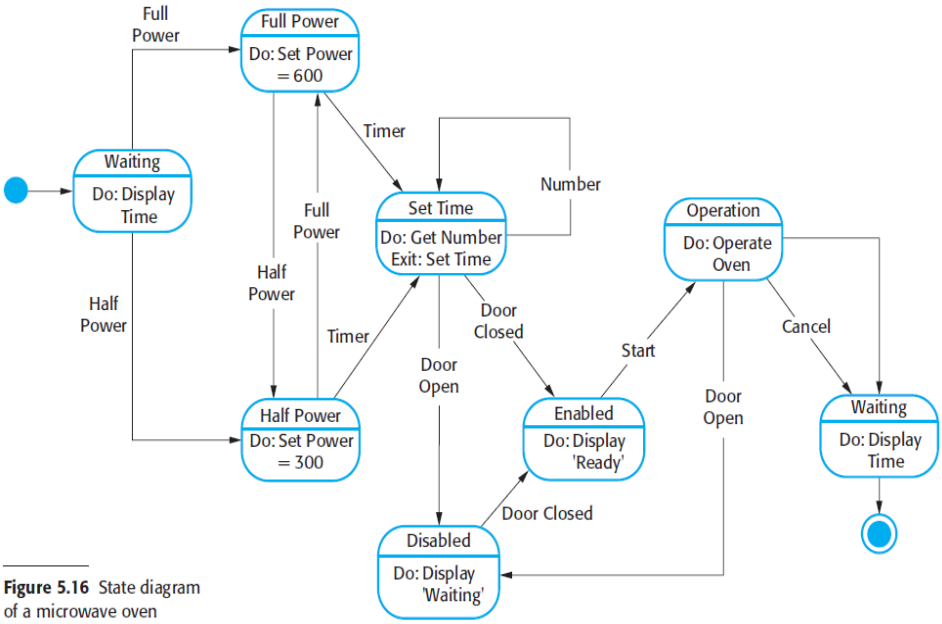


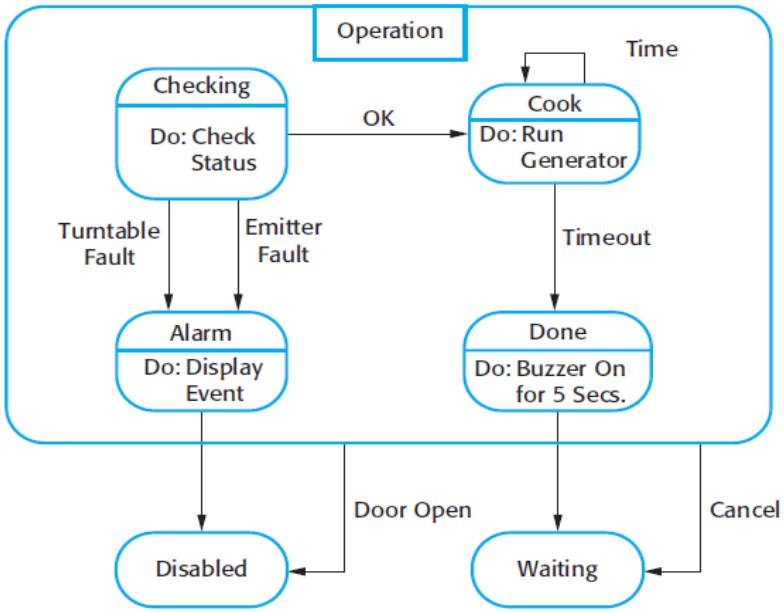
Figure 5.16 State diagram of a microwave oven

The UML notation lets you indicate the activity that takes place in a state. In a detailed system specification you have to provide more detail about both the stimuli and the system states. I illustrate this in Figure 5.17, which shows a tabular description of each state and how the stimuli that force state transitions are generated. The problem with state-based modeling is that the number of possible states increases rapidly.

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.

Figure 5.17 States and stimuli for the microwave oven

One way to do this is by using the notion of a superstate that encapsulates a number of separate states. This superstate looks like a single state on a high-level model but is then expanded to show more detail on a separate diagram. To illustrate this concept, consider the Operation state in Figure 5.15. This is a superstate that can be expanded, as illustrated in Figure 5.18. The Operation state includes a number of sub-states. It shows that operation starts with a status check and that if any problems are discovered an alarm is indicated and operation is disabled. Cooking involves running the microwave generator for the specified time; on completion, a buzzer is sounded. If the door is opened during operation, the system moves to the disabled state, as shown in Figure 5.15.



Model-driven engineering

Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process (Kent, 2002; Schmidt, 2006). The programs that execute on a hardware/software platform are then generated automatically from the models. 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.

The main arguments for and against MDE are:

1. *For MDE* Model-based engineering allows engineers to think about systems at a high level of abstraction, without concern for the details of their implementation. This reduces the likelihood of errors, speeds up the design and implementation process, and allows for the creation of reusable, platform-independent application models. By using powerful tools, system implementations can be generated for different platforms from the same model. Therefore, to

adapt the system to some new platform technology, it is only necessary to write a translator for that platform. When this is available, all platform-independent models can be rapidly rehosted on the new platform.

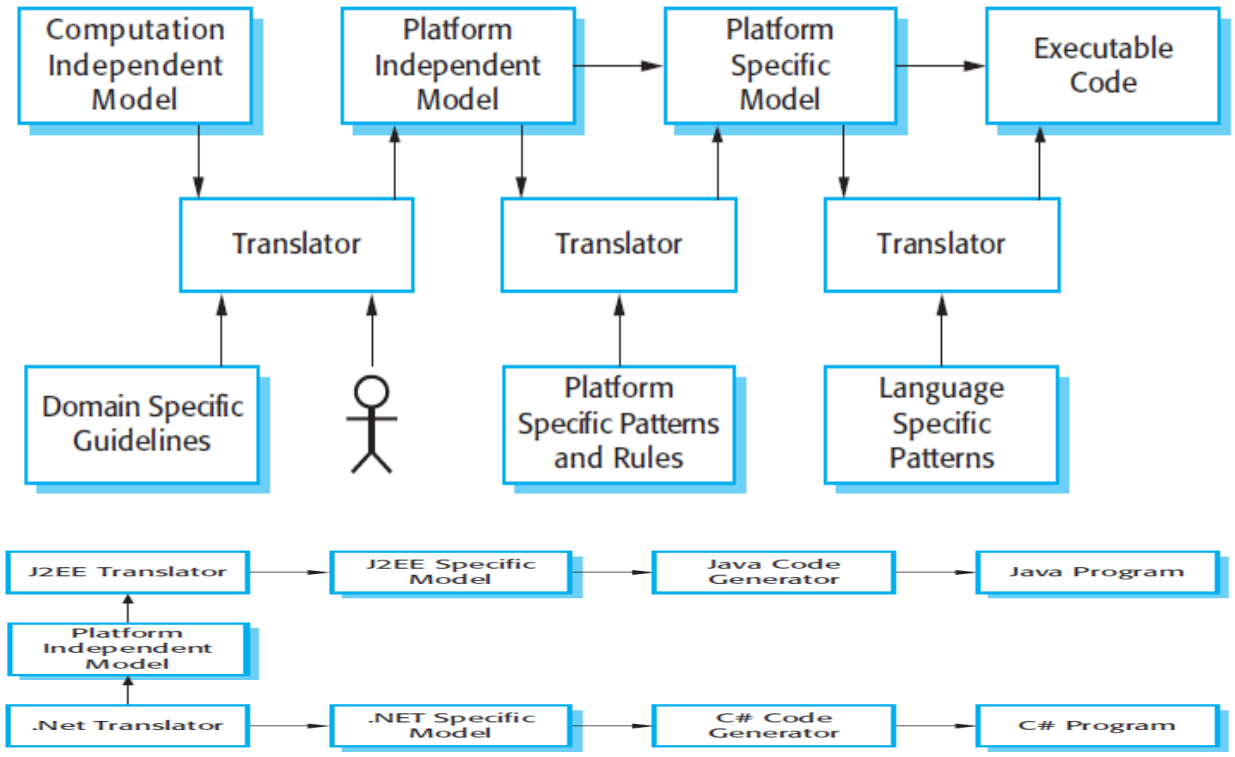
2. *Against MDE* As I discussed earlier in this chapter, models are a good way of facilitating discussions about a software design. However, it does not always follow that the abstractions that are supported by the model are the right abstractions for implementation. So, you may create informal design models but then go on to implement the system using an off-the-shelf, configurable package. Furthermore, the arguments for platform independence are only valid for large long-lifetime systems where the platforms become obsolete during a system's lifetime. However, for this class of systems, we know that implementation is not the major problem—requirements engineering, security and dependability, integration with legacy systems, and testing are more significant.

Model-driven architecture

Model-driven architecture is a model-focused approach to software design and implementation that uses a sub-set of UML models to describe a system. From a high-level platform independent model it is possible, in principle, to generate a working program without manual intervention.

The MDA method recommends that three types of abstract system model should be produced:

1. A computation independent model (CIM) that models the important domain abstractions used in the system. CIMs are sometimes called domain models. For example, there may be a security CIM in which you identify important security abstractions such as an asset and a role and a patient record CIM, in which you describe abstractions such as patients, consultations, etc.
2. A platform independent model (PIM) that models the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
3. Platform specific models (PSM) which are transformations of the platform independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform specific detail. So, the first-level PSM could be middleware-specific but database independent. When a specific database has been chosen, a database specific PSM can then be generated.



Executable UML

The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible. To achieve this, you have to be able to construct graphical models whose semantics are well defined. You also need a way of adding information to graphical models about the ways in which the operations defined in the model are implemented. This is possible using a subset of UML 2, called Executable UML or xUML. I don't have space here to describe the details of xUML, so I simply present a short overview of its main features. UML was designed as a language for supporting and documenting software design, not as a programming language. The designers of UML were not concerned with semantic details of the language but with its expressiveness. They introduced useful notions such as use case diagrams that help with the design but which are too informal to support execution.

To create an executable sub-set of UML, the number of model types has therefore been dramatically reduced to three key model types:
1. Domain models identify the principal concerns in the system. These are defined using UML class diagrams that include objects, attributes, and associations.
2. Class models, in which classes are defined, along with their attributes and operations.
3. State models, in which a state diagram is associated with each class and is used to describe the lifecycle of the class.

Design and implementation

Objectives

The objectives of this chapter are to introduce object-oriented software design using the UML and highlight important implementation concerns.
When you have read this chapter, you will:

_ understand the most important activities in a general, objectoriented design process;
_ understand some of the different models that may be used to document an object-oriented design;
_ know about the idea of design patterns and how these are a way of reusing design knowledge and experience;
_ have been introduced to key issues that have to be considered when implementing software, including software reuse and open-source development.

Contents

Object-oriented design using the UML

An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. The representation of the state is private and cannot be accessed directly from outside the object. Object-oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their interactions. When the design is realized as an executing program, the objects are created dynamically from these class definitions.

Object-oriented systems are easier to change than systems developed using functional approaches. Objects include both data and operations to manipulate that data. They may therefore be understood and modified as stand-alone entities. Changing the implementation of an object or adding services should not affect other system objects. Because objects are associated with things, there is often a clear mapping between real world entities (such as hardware components) and their controlling objects in the system.

This improves the understandability, and hence the maintainability, of the design. To develop a system design from concept to detailed, object-oriented design, there are several things that you need to do:

1. Understand and define the context and the external interactions with the system.
2. Design the system architecture.
3. Identify the principal objects in the system.
4. Develop design models.
5. Specify interfaces.

System context and interactions

The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment. This is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment. Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems. In this case, you need to decide how functionality is distributed between the control system for all of the weather stations, and the embedded software in the weather station itself.

System context models and interaction models present complementary views of the relationships between a system and its environment:

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.

The context model of a system may be represented using associations. Associations simply show that there are some relationships between the entities involved in the association. The nature of the relationships is now specified. You may therefore documentth e environment of the system using a simple block diagram, showing the entities in the system and their associations. This is illustrated in Figure 7.1, which shows that the systems in the environment of each weather station are a weather information system, an onboard satellite system, and a control system. The cardinality information on the link shows that there is one control system but several weather stations, one satellite, and one general weather information system.

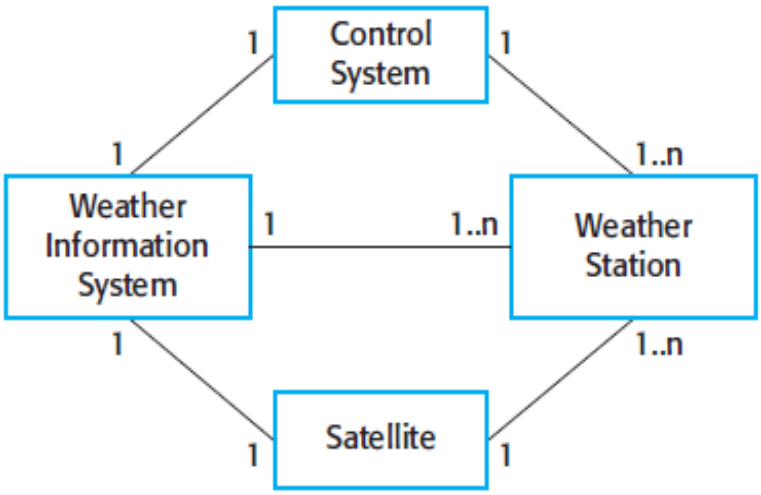


Figure 7.1 System context for the weather station

The use case model for the weather station is shown in Figure 7.2. This shows that the weather station interacts with the weather information system to report weather data and the status of the weather station hardware. Other interactions are with a control system that can issue specific weather station control commands

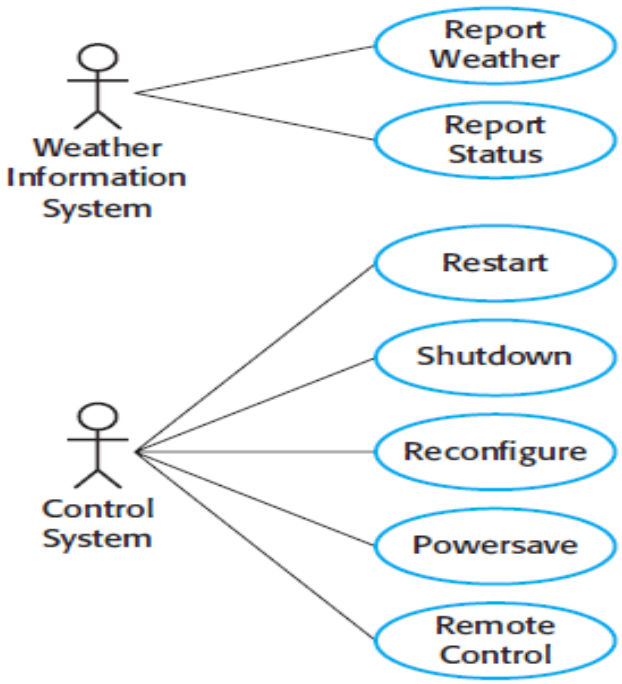


Figure 7.2 Weather Control station use cases

Architectural design

Once the interactions between the software system and the system's environment have been defined, you use this information as a basis for designing the system architecture. Of course, you need to combine this with your general knowledge of the principles of architectural design and with more detailed domain knowledge.

Figure 7.3 Use case description—Report weather

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Dat	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 are 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.

You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client–server model. However, this is not essential at this stage.

The high-level architectural design for the weather station software is shown in Figure 7.4. The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure, shown as the Communication link in Figure 7.4. Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them.

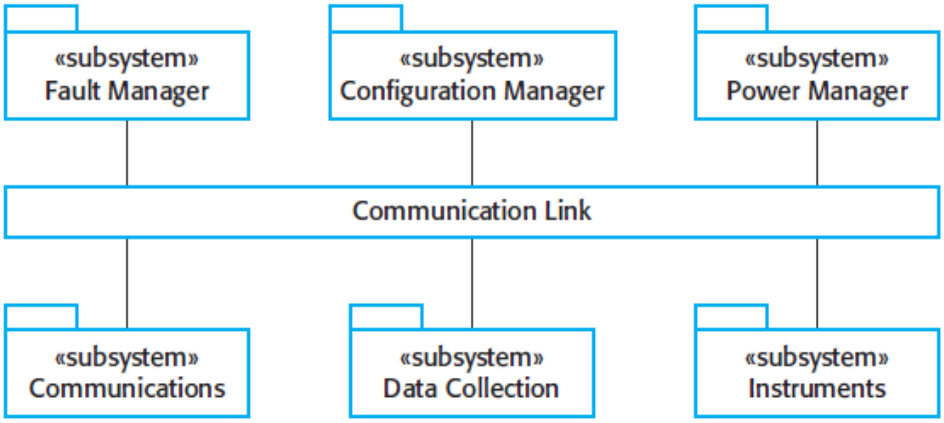


Figure 7.4 High-level architecture of the weather station

Object class identification

- By this stage in the design process, you should have some ideas about the essential objects in the system that you are designing.
- As your understanding of the design develops, you refine these ideas about the system objects.
- The use case description helps to identify objects and operations in the system

system object or objects that encapsulate the system interactions defined in the use cases. With these objects in mind, you can start to identify the object classes in the system.

There have been various proposals made about how to identify object classes in object-oriented systems:

1. Use a grammatical analysis of a natural language description of the system to be constructed. Objects and attributes are nouns; operations or services are verbs
2. Use tangible entities (things) in the application domain such as aircraft, roles such as manager or doctor, events such as requests, interactions such as meetings, locations such as offices, organizational units such as companies, and so on
3. Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes, and operations

In the wilderness weather station, object identification is based on the tangible hardware in the system. I don't have space to include all the system objects here, but I have shown five object classes in Figure 7.6. The Ground thermometer, Anemometer, and Barometer objects are application domain objects, and the WeatherStation and WeatherData objects have been identified from the system description and the scenario (use case) description:

1. The Weather Station object class provides the basic interface of the weather station with its environment. Its operations reflect the interactions shown in

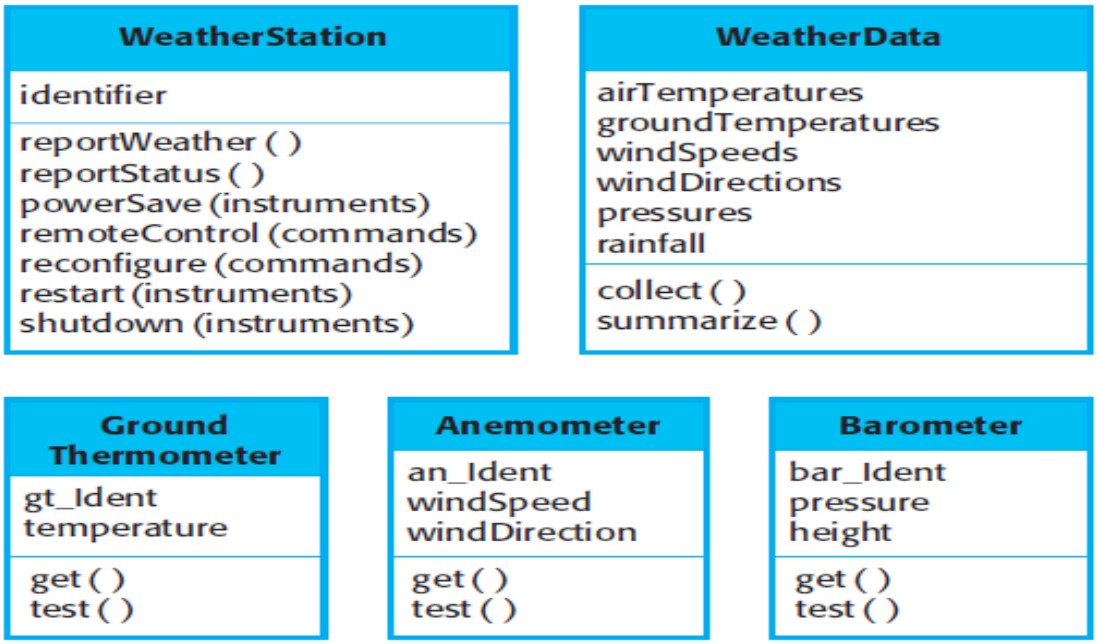


Figure 7.6 Weather station objects

Design models

Generally, you get around this type of conflict by developing models at different levels of detail. Where there are close links between requirements engineers, designers, and programmers, then abstract models may be all that are required. Specific design decisions may be made as the system is implemented, with problems resolved through informal discussions. When the links between system specifiers, designers, and programmers are indirect (e.g., where a system is being designed in one part of an organization but implemented elsewhere), then more detailed models are likely to be needed.

An important step in the design process, therefore, is to decide on the design models that you need and the level of detail required in these models. This depends on the type of system that is being developed. You design a sequential data-processing system in a different way from an embedded real-time system, so you will need different design models.

When you use the UML to develop a design, you will normally develop two kinds of design model:
1. Structural models, which describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalization (inheritance) relationships, uses/used-by relationships, and composition relationships.
2. Dynamic models, which describe the dynamic structure of the system and show the interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the state changes that are triggered by these object interactions.

In the early stages of the design process, I think there are three models that are particularly useful for adding detail to use case and architectural models:
1. Subsystem models, which that show logical groupings of objects into coherent subsystems. These are represented using a form of class diagram with each subsystem shown as a package with enclosed objects. Subsystem models are static (structural) models.
2. Sequence models, which show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic models.
3. State machine model, which show how individual objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic models.

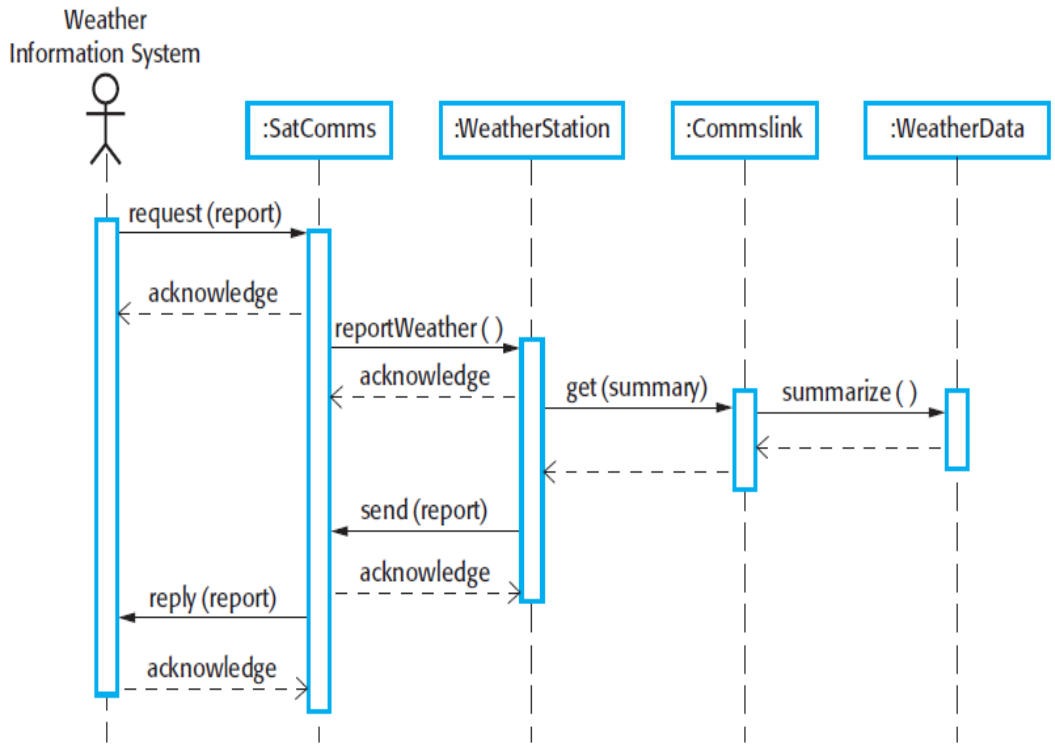


Figure 7.7 Sequence diagram describing data collection
Figure 7.7 is an example of a sequence model, shown as a UML sequence diagram.
This diagram shows the sequence of interactions that take place when an external system requests the summarized data from the weather station. You read sequence diagrams from top to bottom:

1. The SatComms object receives a request from the weather information system to collect a weather report from a weather station. It acknowledges receipt of this request. The stick arrowhead on the sent message indicates that the external system does not wait for a reply but can carry on with other processing.
2. SatComms sends a message to WeatherStation, via a satellite link, to create a summary of the collected weather data. Again, the stick arrowhead indicates that SatComms does not suspend itself waiting for a reply.
3. WeatherStation sends a message to a Commslink object to summarize the weather data. In this case, the squared-off style of arrowhead indicates that the instance of the WeatherStation object class waits for a reply.
4. Commslink calls the summarize method in the object Weather Data and waits for a reply.
5. The weather data summary is computed and returned to Weather Station via the Commslink object.
6. WeatherStation then calls the SatComms object to transmit the summarized data to the weather information system, through the satellite communications system.

Sequence diagrams are used to model the combined behavior of a group of objects but you may also want to summarize the behavior of an object or a subsystem in response to messages and events.
Figure 7.8 is a state diagram for the weather station system that shows how it responds to requests for various services.
You can read this diagram as follows:
1. If the system state is Shutdown then it can respond to a restart(), a reconfigure(), or a powerSave() message. The unlabeled arrow with the black blob indicates that the Shutdown state is the initial state. A restart() message causes a transition to normal operation. Both the powerSave() and reconfigure() messages cause a transition to a state in which the system reconfigures itself. The state diagram shows that reconfiguration is only allowed if the system has been shut down.
2. In the Running state, the system expects further messages. If a shutdown() message is received, the object returns to the shutdown state.
3. If a reportWeather() message is received, the system moves to the Summarizing state. When the summary is complete, the system moves to a Transmitting state where the information is transmitted to the remote system. It then returns to the Running state.
4. If a reportStatus() message is received, the system moves to the Testing state, then the Transmitting state, before returning to the Running state.
5. If a signal from the clock is received, the system moves to the Collecting state,where it collects data from the instruments. Each instrument is instructed in turn to collect its data from the associated sensors.
6. If a remoteControl() message is received, the system moves to a controlled state in which it responds to a different set of messages from the remote control room. These are not shown on this diagram.

State diagrams are useful high-level models of a system or an object's operation.You don't usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

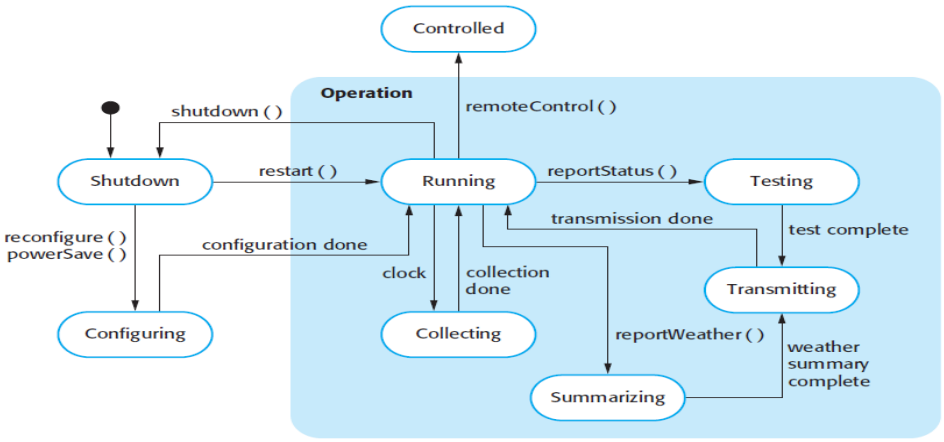
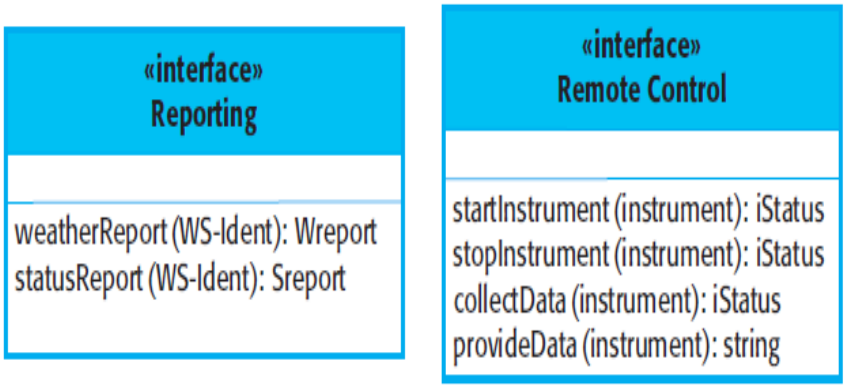


Figure 7.8 Weather station state diagram

Interface specification

An important part of any design process is the specification of the interfaces between the components in the design. You need to specify interfaces so that objects and subsystems can be designed in parallel. Once an interface has been specified, the developers of other objects may assume that interface will be implemented. Interface design is concerned with specifying the detail of the interface to an object or to a group of objects. This means defining the signatures and semantics of the services that are provided by the object or by a group of objects. Interfaces can be specified in the UML using the same notation as a class diagram.



Design patterns

Design patterns were derived from ideas put forward by Christopher Alexander (Alexander et al., 1977), who suggested that there were certain common patterns of building design that were inherently pleasing and effective. The pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings. The pattern is not a detailed specification. Rather, you can think of it as a description of accumulated wisdom and experience, a well-tried solution to a common problem.

A quote from the Hillside Group web site (http://hillside.net), which is dedicated to maintaining information about patterns, encapsulates their role in reuse:

Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.

Pattern name: Observer

Description: Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.

Problem description: In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.

This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

Solution description: This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.

The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
The UML model of the pattern is shown in Figure 7.12.

Consequences: The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Figure 7.10 The Observer pattern

Design patterns are usually associated with object-oriented design. Published patterns often rely on object characteristics such as inheritance and polymorphism to provide generality. However, the general principle of encapsulating experience in apattern is one that is equally applicable to any kind of software design. So, you could have configuration patterns for COTS systems. Patterns are a way of reusing the knowledge and experience of other designers.
The four essential elements of design patterns were defined by the 'Gang of Four' in their patterns book:
1. A name that is a meaningful reference to the pattern.
2. A description of the problem area that explains when the pattern may be applied.
3. A solution description of the parts of the design solution, their relationships, and their responsibilities. This is not a concrete design description. It is a templatefor a design solution that can be instantiated in different ways. This is often expressed graphically and shows theelationships between the objects and object classes in the solution.
4. A statement of the consequences—the results and trade-offs—of applying the pattern. This can help designers understand whether or not a pattern can be used in a particular situation.

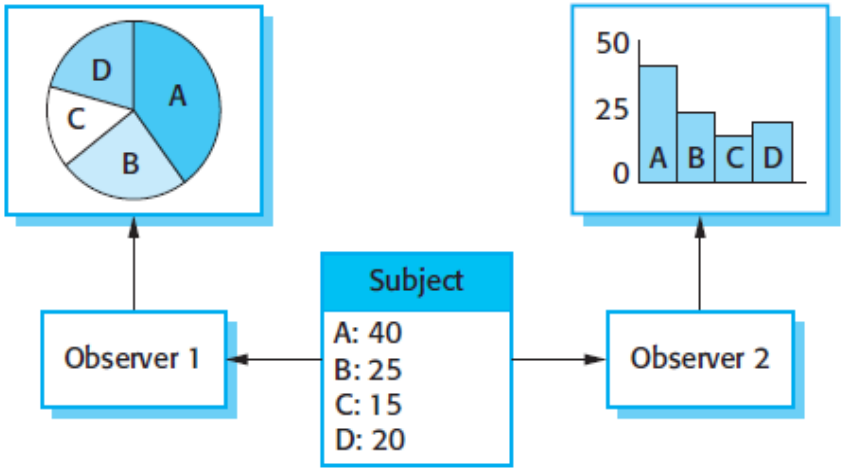


Figure 7.11 Multiple displays

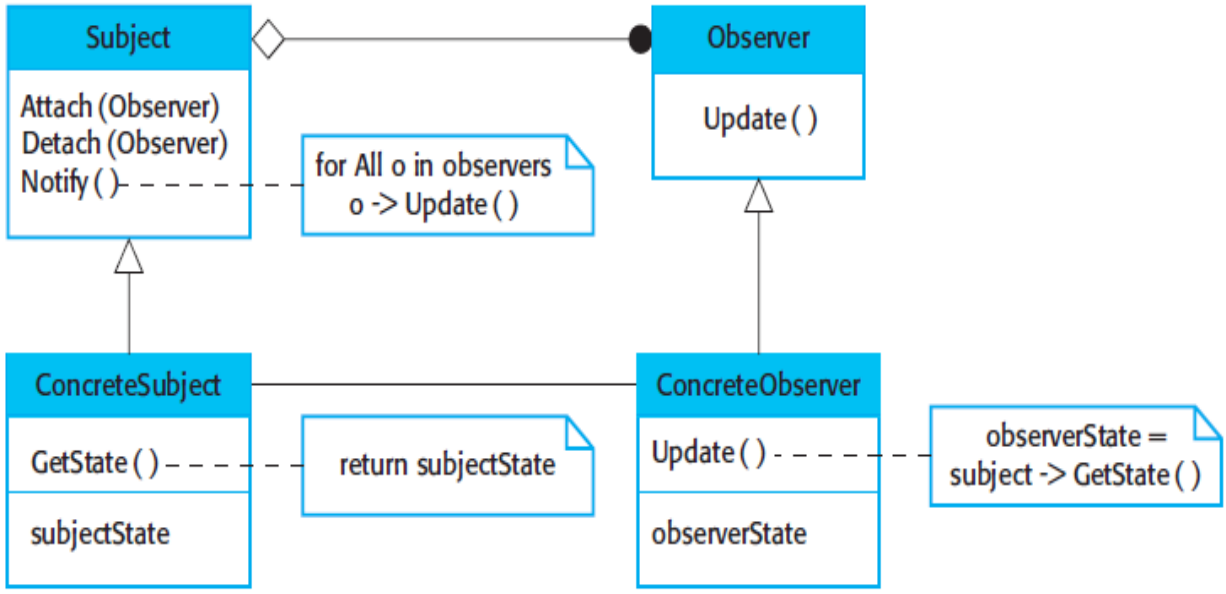


Figure 7.12 A UML model of the Observer pattern

Implementation issues

Software engineering includes all of the activities involved in software development from the initial requirements of the system through to maintenance and management of the deployed system. A critical stage of this process is, of course, system implementation, where you create an executable version of the software. Implementation may involve developing programs in high- or low-level programming languages or tailoring and adapting generic, off-the-shelf systems to meet the specific requirements of an organization. I introduce some aspects of implementation that are particularly important to software engineering that are often not covered in programming texts.

These are:
1. *Reuse* Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
2. *Configuration management* During the development process, many different versions of each software component are created. If you don't keep track of these versions in a configuration management system, you are liable to include the wrong versions of these components in your system.
3. *Host-target development* Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system). The host and target systems are sometimes of the same type but, often they are completely different.

Software reuse is possible at a number of different levels:
1. *The abstraction level* At this level, you don't reuse software directly but rather use knowledge of successful abstractions in the design of your software. Designpatterns and architectural patterns are ways of representing abstract knowledge for reuse.
2. *The object level* At this level, you directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need. For example, if you need to process mail messages in a Java program, you may use objects and methods from a JavaMail library.
3. *The component level* Components are collections of objects and object classes that operate together to provide related functions and services.
4. *The system level* At this level, you reuse entire application systems. This usually nvolves some kind of configuration of these systems.Sometimes this approach may involve reusing several different systems and integrating these to create a new system.

Configuration management

Configuration management is the name given to the general process of managing a changing software system. The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system. There are, therefore, three fundamental configuration management activities:

1. Version management, where support is provided to keep track of the diff erentversions of software components. Version management systems include facilities to coordinate development by several programmers.
2. System integration, where support is provided to help developers define whatversions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
3. Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed

Host-target development

Most software development is based on a host-target model. Software is developed on one computer (the host), but runs on a separate machine (the target). More generally, we can talk about a development platform and an execution platform. A platform is more than just hardware. It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
A software development platform should provide a range of tools to support software engineering processes.

These may include:
1. An integrated compiler and syntax-directed editing system that allows you to create, edit, and compile code.
2. A language debugging system.
3. Graphical editing tools, such as tools to edit UML models.
4. Testing tools, such as JUnit (Massol, 2003) that can automatically run a set of tests on a new version of a program.
5. Project support tools that help you organize the code for different development projects.

Issues that you have to consider in making this decision are:
1. *The hardware and software requirements of a component* If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
2. *The availability requirements of the system* High-availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
3. *Component communications* If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces communications latency, the delay between the time a message is sent by one component and received by another.

Open source development

Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code. In principle at least, any contributor to an open source project may report and fix bugs and propose new features and functionality. However, in practice, successful open source systems still rely on a core group of developers who control changes to the software.

For a company involved in software development, there are two open source issues that have to be considered:
1. Should the product that is being developed make use of open source components?
2. Should an open source approach be used for the software's development?
The answers to these questions depend on the type of software that is being developed and the background and experience of the development team. If you are developing a software product for

sale, then time to market and reduced costs are critical. If you are developing in a domain in which there are high-quality open source systems available, you can save time and money by using these systems. However, if you are developing software to a specific set of organizational requirements, then using open source components may not be an option. You may have to integrate your software with existing systems that are incompatible with available open source systems. Even then, however, it could be quicker and cheaper to modify the open source system rather than redevelop the functionality that you need.

Many companies believe that adopting an open source approach will reveal confidential business knowledge to their competitors and so are reluctant to adopt this development model. However, if you are working in a small company and you open source your software, this may reassure customers that they will be able to support the software if your company goes out of business.

Open source licensing

Although a fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code. Legally, the developer of the code (either a company or an individual) still owns the code.

Most open source licenses are derived from one of three general models:
1. The GNU General Public License (GPL). This is a so-called 'reciprocal' license that, simplistically, means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
2. The GNU Lesser General Public License (LGPL). This is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components. However, if you change the licensed component, then you must publish this as open source.
3. The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to republish any changes or modifications made to open source code.

You can include the code in proprietary systems that are sold. Bayersdorfer (2007) suggeststhat companies managing projects that use open source should:

1. Establish a system for maintaining information about open source components that are downloaded and used. You have to keep a copy of the license for each component that was valid at the time the component was used. Licenses may change so you need to know the conditions that you have agreed to.
2. Be aware of the different types of licenses and understand how a component is licensed before it is used. You may decide to use a component in one system but not in another because you plan to use these systems in different ways.
3. Be aware of evolution pathways for components. You need to know a bit about the open source project where components are developed to understand how they might change in future.
4. Educate people about open source. It's not enough to have procedures in place to ensure compliance with license conditions. You also need to educate developers about open source and open source licensing.

5. Have auditing systems in place. Developers, under tight deadlines, might be tempted to break the terms of a license. If possible, you should have software in place to detect and stop this.
6. Participate in the open source community. If you rely on open source products, you should participate in the community and help support their development.

Module-4 Software testing (page no 1-18)

8. Software Testing

(Page no 1-18)

8.1 Development testing	(Sec 8.1)
8.2 Test-driven development	(Sec 8.2)
8.3 Release testing	(Sec 8.3)
8.4 User testing	(Sec 8.4)
Automation	

9. Software Evolution

(Page no 19-last)

9.1 Evolution processes	(Sec 9.1).
9.2 Program evolution dynamics	(Sec 9.2).
9.3 Software maintenance	(Sec 9.3).
9.4 Legacy system management	(Sec 9.4).

Textbooks:

1. Ian Somerville: Software Engineering, 9th Edition, Pearson Education, 2012.
   (Listed topics only from Chapters 1,2,3,4, 5, 7, 8, 9, 23, and 24)
2. Michael Blaha, James Rumbaugh: Object Oriented Modeling and Design with UML, 2nd Edition, Pearson Education, 2005.

Reference Books:

1. Roger S. Pressman: Software Engineering-A Practitioners approach, 7th Edition, Tata McGraw Hill.
2. Pankaj Jalote: An Integrated Approach to Software Engineering, Wiley India

Question Paper Pattern:

- ✓ The question paper will have ten questions.
- ✓ Each full Question consisting of 20 marks
- ✓ There will be 2 full questions (with a maximum of four sub questions) from each module.
- ✓ Each full question will have sub questions covering all the topics under a module.
- ✓ The students will have to answer 5 full questions, selecting one full question from each module.

8.0 INTRODUCTION PROGRAM TESTING

- 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.
- When you test software, you execute a program using artificial data.
- You check the results of the test run for errors, anomalies or information about the program's non-functional attributes.
- Can reveal the presence of errors NOT their absence.
- Testing is part of a more general verification and validation process, which also includes static validation techniques.

PROGRAM TESTING GOALS:

The testing process has two distinct goals:

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. – **Leads to validation**
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. **-Leads to defect testing:**

VALIDATION AND DEFECT TESTING

Validation Testing: The first goal leads to **validation testing :** Where we expect the system to perform correctly using a given set of test cases that reflect the system's expected use. To demonstrate to the developer and the system customer that the software meets its requirements. A successful test shows that the system operates as intended.

Defect Testing: The second goal leads to **defect testing:** Where 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.

Of course, there is no definite boundary between these two approaches to testing. During validation testing, you will find defects in the system; during defect testing, some of the tests will show that the program meets its requirements.

The diagram shown in Figure 8.1 may help to explain the differences between validation testing and defect testing. Think of the system being tested as a black box. The system accepts inputs from some input set I and generates outputs in an output set O. Some of the outputs will be erroneous. These are the outputs in set Oe that are generated by the system in response to inputs in the set Ie. The priority in defect testing is to find those inputs in the set Ie because these reveal problems with the system. Validation testing involves testing with correct inputs that are outside Ie. These stimulate the system to generate the expected correct outputs.

Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance. It is always possible that a test that you have overlooked could discover further problems with the system.

Testing can only show the presence of errors, not their absence

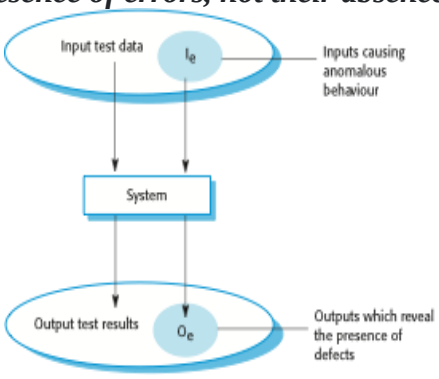


Fig 8.1 An input-output model of program testing

Testing is part of a broader process of software verification and validation (V & V). Verification and validation is not the same thing, although they are often confused. Verification and validation processes

are concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software. These checking processes start as soon as requirements become available and continue through all stages of the development process.

The aim of verification is to check that the software meets its stated functional and non-functional requirements.

Validation, however, is a more general process. The aim of validation is to ensure that the software meets the customer's expectations. It goes beyond simply checking conformance with the specification to demonstrating that the software does what the customer expects it to do. Validation is essential because requirements specifications do not always reflect the real wishes or needs of system customers and users.

Verification vs. validation

- **Verification:** "Are we building the product right".
  The software should conform to its specification.

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

V & V confidence:

- The aim of V & V is to establish confidence that the system is 'fit for purpose'. i.e means that the system must be good enough for its intended use
- The level of required confidence is depends on three

1. **System's purpose**: The level of confidence depends on how critical the software is to an organisation.
2. **User expectations of system user**: Users may have low expectations of certain kinds of software.
3. **Marketing environment for the system**: Getting a product to market early may be more important than finding defects in the program.

INSPECTIONS AND TESTING:

In addition to the software testing, the V &V process may involve software inspections and reviews.

Software inspections and reviews concerned with analysis of the static system representation, to discover problems *(*static V&V Technique) i.e. analyse and check the system requirements, design models, the program source code and even proposed test case -called static V & V tech.

May be supplement by tool-based document and code analysis. Discussed in Chapter 15.

Software inspection and testing concerned with exercising and observing product behaviour or different stages in the software process (dynamic V & V tech).

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

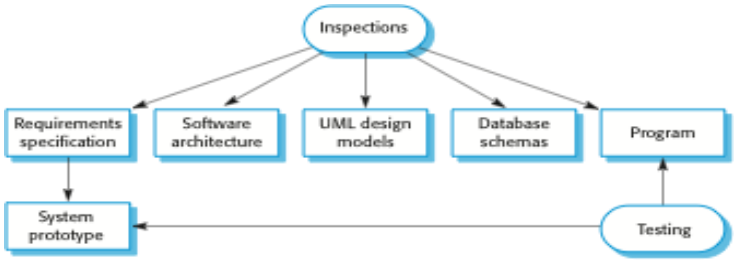


Fig 8.2 Inspections and testing

Figure 8.2 shows that software inspection and testing support V & V at different stages in the software process. The arrows indicate the stages in the process where the techniques may be used.

Software inspections mostly focus on the source code of a system but any readable representation of the software, such as its requirements or a design model, can be inspected. When you inspect a system, you *use knowledge of the system, its application domain*, and the *programming or modelling language to discover errors.*

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation.

- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

Advantages of software inspections over testing

1. 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.
2. 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.
3. 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.

Inspections and testing

However inspection cannot replace the testing:

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

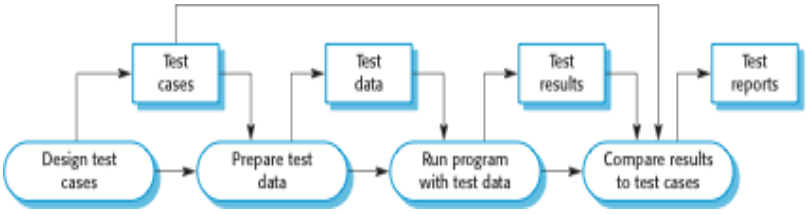


Figure 8.3 A model of the "traditional "software testing process

Figure 8.3 is an abstract model of the 'traditional' testing process, as used in plandriven development. Test cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested. Test data are the inputs that have been devised to test a system. Test data can sometimes be generated automatically, but automatic test case generation is impossible, as people who understand what the system is supposed to do must be involved to specify the expected test results. However, test execution can be automated. The expected results are automatically compared with the predicted results so there is no need for a person to look for errors and anomalies in the test run.

Stages of testing

- ✧ **Development testing**: where the system is tested during development to discover bugs and defects.
- ✧ **Release testing:** Where a separate testing team tests a complete version of the system before it is released to users.
- ✧ **User testing:** Where users or potential users of a system test the system in their own environment.

8.1 DEVELOPMENT TESTING

- ✧ Development testing includes three or all testing activities that are carried out by development team.
  - ***Unit testing***, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
  - ***Component testing***, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
  - ***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.

8.1.1 Unit testing

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

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

Object class testing: when we are testing object classes, we should design our test to provide all of the features of the object. This means complete object class test should coverage of a class involves

- Testing all operations associated with an object
- Set and check the value of all object attributes
- Put the object into all possible states. This means that we should simulate all events that cause a state change

Generalization or inheritance makes object class testing more complicated.

- We can't simply test an operation in the class where it is defined and assume that it will work as expected in the subclasses that inherit the operation.
- The operation that is inherited may make assumptions about other operations and attributes. These may not be valid in some subclasses that inherit the operation.
- We therefore have to test the inherited operation in all of the contexts where it is used.

To test the states of the weather station, we use a state model, using this model, we can identify sequences of state transitions that have to be tested and define event sequences to force these transitions. In principle, we should test every possible state transition sequence, although in practice this may be too expensive. Examples of state sequences that should be tested in the weather station include:

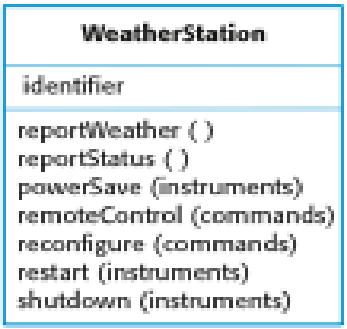


Figure 8.4 The weather station object interface

Weather station testing

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

Automated unit testing

- Whenever possible, we should automate unit testing, we make use of a test automation framework (such as J Unit) to write and run our program tests.
- Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success of otherwise of the tests.

Automated unit test has three parts

1. **A setup part**, where you initialize the system with the test case, namely the inputs and expected outputs.
2. **A call part**, where you call the object or method to be tested.
3. **An assertion part** where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.

8.1.2 Choosing unit test cases (Unit test effectiveness)

Testing is expensive and time consuming, so it is important that you choose effective unit test cases. Effectiveness, in this case, means two things:

1. The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
2. If there are defects in the component, these should be revealed by test cases.

This leads to two types of unit test case or test cases

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.

Two possible Testing strategies that can be effective in helping us to choose test cases

Two possible strategies here that can be effective in helping you choose test cases. These are:

1. **Partition testing**: Where you identify groups of inputs that have common characteristics and should be processed in the same way.
   - You should choose tests from within each of these groups.
2. **Guideline-based testing**: Where you use testing guidelines to choose test cases.
   - These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

Partition testing

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

Equivalence partitioning

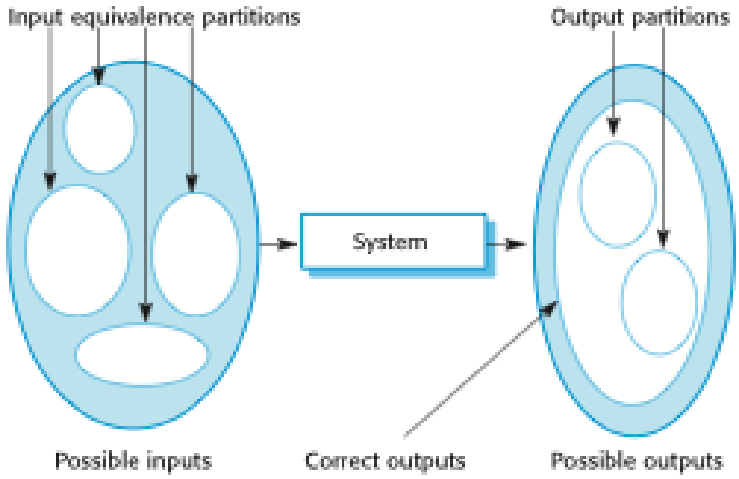
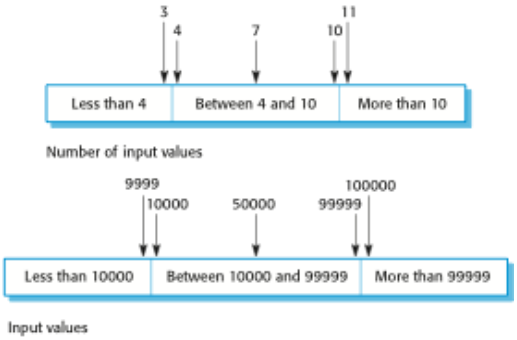


Figure 8.5 Equivalence partitioning

The input data and output results of a program often fall into a number of different classes with common characteristics. Examples of these classes are positive numbers, negative numbers, and menu selections. Programs normally behave in a comparable way for all members of a class. That is, if you test a program that does a computation and requires two positive numbers, then you would expect the program to behave in the same way for all positive numbers. Because of this equivalent behavior, these classes are sometimes called equivalence partitions or domains (Bezier, 1990). One systematic approach to test case design is based on identifying all input and output partitions for a system or component. Test cases are designed so that the inputs or outputs lie within these partitions. Partition testing can be used to design test cases for both systems and components.

In Figure 8.5, the large shaded ellipse on the left represents the set of all possible inputs to the program that is being tested. The smaller unshaded ellipses represent equivalence partitions. A program being tested should process all of the members of an input equivalence partitions in the same way. Output equivalence partitions are partitions within which all of the outputs have something in common. Sometimes there is a 1:1 mapping between input and output equivalence partitions. However, this is not always the case; you may need to define a separate input equivalence partition, where the only common characteristic of the inputs is that they generate outputs within the same output partition. The shaded area in the left ellipse represents inputs that are invalid. The shaded area in the right ellipse represents exceptions that may occur (i.e., responses to invalid inputs).



Testing guidelines (sequences)

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

General testing guidelines

1. Choose inputs that force the system to generate all error messages
2. Design inputs that cause input buffers to overflow
3. Repeat the same input or series of inputs numerous times
4. Force invalid outputs to be generated
5. Force computation results to be too large or too small.

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.

8.1.3 COMPONENT TESTING

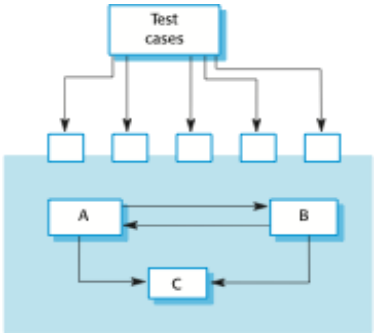
- Software components are often composite components that are made up of several interacting objects.
  - For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- We access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
  - We can assume that unit tests on the individual objects within the component have been completed.

Interface testing

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

There are different types of interface between program components and, consequently, different types of interface error that can occur:

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



Interface testing

Interface errors: Interface errors are one of the most common forms of error in complex systems (Lutz, 1993). These errors fall into three classes:

1. **Interface misuse**: A calling component calls another component and makes an error in its use of its interface e.g. error is common with parameter interfaces, where parameters be passed in the wrong type or in the wrong order or wrong number.
2. **Interface misunderstanding**: A calling component misunderstands the specification of the interface of the called component and makes assumptions about its behaviour.
   The called component does not behave as expected which then causes unexpected behaviour in the calling component.
   For example, a binary search method may be called with a parameter that is an unordered array. The search would then fail.
   i.e. A calling component embeds assumptions about the behaviour of the called component which are incorrect.
3. **Timing errors**: These occur in real-time systems that use a shared memory or a message-passing interface. The producer of data and the consumer of data may operate at different speeds. Unless particular care is taken in the interface design, the consumer can access out-of-date information because the producer of the information has not updated the shared interface information.
   i.e. The called and the calling component operate at different speeds and out-of-date information is accessed.

General guidelines for Interface testing:

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

8.1.4 SYSTEM TESTING

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

Difference between System and component testing

1. During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
2. Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
   - **In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.**

When we integrate components to create a system, we get emergent behaviour. This means that some elements of system functionality only become obvious when we put the components together. This may

be planned emergent behaviour, which has to be tested. Ex: is integration of authentication components with a component that that update information.

Therefore system testing should focus on

- Testing the interactions between the components and objects that make up a system.
- We may also test reusable components or systems to check that they work as expected when they are integrated with new components.
- This interaction testing should discover those component bugs that are only revealed when a component is used by other components in the system.
- Interaction testing also helps find misunderstandings, made by component developers, about other components in the system.

Use-case testing: Because of its focus on interactions, use case–based testing is an effective approach to system testing.

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

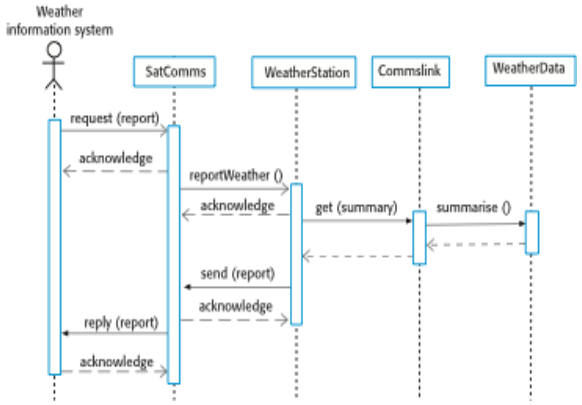


Figure 8.8 Collect weather data sequence chart

To illustrate this, used an example from the wilderness weather station system where the weather station is asked to report summarized weather data to a remote computer. The use case for this is described in Figure 7.3 (see previous chapter).

Figure 8.8 (which is a copy of Figure 7.7) shows the sequence of operations in the weather station when it responds to a request to collect data for the mapping system. We can use this diagram to identify operations that will be tested and to help design the test cases to execute the tests.

Therefore, issuing a request for a report will result in the execution of the following thread of methods:

SatComms: request → WeatherStation: reportWeather → Commslink: Get(summary) → WeatherData:summarize

The sequence diagram helps us to design the specific test cases that we need as it shows what inputs are required and what outputs are created:

1. *An input of a request for a report should have an associated acknowledgment. A report should ultimately be returned from the request.*
   During testing, we should create summarized data that can be used to check that the report is correctly organized.
2. *An input request for a report to WeatherStation results in a summarized report being generated.*
   We can test this in isolation by creating raw data corresponding to the summary that we have prepared for the test of SatComms and checking that the WeatherStation object correctly produces this summary.
   This raw data is also used to test the WeatherData object.

Testing policies:

For most systems, it is difficult to know how much system testing is essential and when you should to stop testing. Exhaustive testing, where every possible program execution sequence is tested, is impossible.

Testing, therefore, has to be based on a subset of possible test cases. Ideally, software companies should have policies for choosing this subset.

These policies might be based on general testing policies, such as a policy that all program statements should be executed at least once.

Alternatively, they may be based on experience of system usage and focus on testing the features of the operational system. For example:

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

8.2 TEST-DRIVEN DEVELOPMENT

- Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.
- Tests are written before code and 'passing' the tests is the critical driver of development.
- You develop code incrementally, along with a test for that increment. You don't move on to the next increment until the code that you have developed passes its test.
- TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

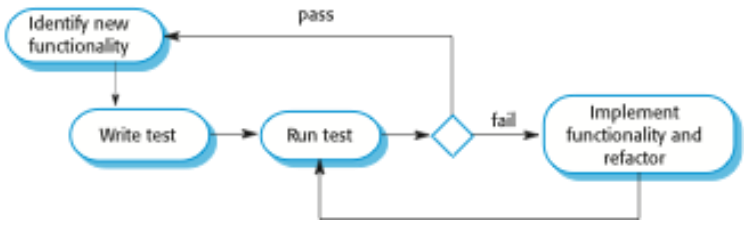


Figure 8.9 Test-driven developments

TDD process activities: The fundamental TDD process is shown in Figure 8.9. The steps in the process are as follows:

- Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement this as an automated test.
- Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- Implement the functionality and re-run the test.
- Once all tests run successfully, you move on to implementing the next chunk of functionality.

Benefits of test-driven development

1. **Code coverage**: Every code segment that you write has at least one associated test so all code written has at least one test.
2. **Regression testing**: A regression test suite is developed incrementally as a program is developed.

3. **Simplified debugging**: When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
4. **System documentation**: The tests themselves are a form of documentation that describe what the code should be doing.

Regression testing

- Regression testing is testing the system to check that changes have not 'broken' previously working code.
- In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are rerun every time a change is made to the program.
- Tests must run 'successfully' before the change is committed.

8.3 RELEASE TESTING

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

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

Release testing and system testing: There are two important distinctions between release testing and system testing during the development process:

- Release testing is a form of system testing.
- 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).

8.3.1. REQUIREMENTS BASED TESTING

- Requirements-based testing involves examining each requirement and developing a test or tests for it.
- For example, consider related requirements for the MHC-PMS (introduced in Chapter 1), which are concerned with checking for drug allergies:
  MHC-PMS requirements:
    - *If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.*
    - *If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.*

Requirements tests: To check if these requirements have been satisfied, you may need to develop several related tests:

1. Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
2. Set up a patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the warning is issued by the system.

3. Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
4. Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
5. Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.

8.3.2 SCENARIO TESTING

- Scenario testing is an approach to release testing where we devise typical scenarios of use and use these to develop test cases for the system.
- A scenario is a story that describes one way in which the system might be used. Scenarios should be realistic and real system users should be able to relate to them.
- If we have used scenarios as part of the requirements engineering process (described in Chapter 4), then you may be able to reuse these as testing scenarios.
    - A scenario test should be a narrative story that is credible and fairly complex. It should motivate stakeholders; that is, they should relate to the scenario and believe that it is important

It test the number of Features of MHC-PMC tested by scenario

1. Authentication by logging on to the system.
2. Downloading and uploading of specified patient records to a laptop.
3. Home visit scheduling.
4. Encryption and decryption of patient records on a mobile device.
5. Record retrieval and modification.
6. Links with the drugs database that maintains side-effect information.
7. The system for call prompting.

Kate is a nurse who specializes in mental health care. One of her responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, Kate logs into the MHC-PMS and uses it to print her schedule of home visits for that day, along with summary information about the patients to be visited. She requests that the records for these patients be downloaded to her laptop. She is prompted for her key phrase to encrypt the records on the laptop.

One of the patients that she visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. Kate looks up Jim's record and is prompted for her key phrase to decrypt the record. She checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so she notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. He agrees so Kate enters a prompt to call him when she gets back to the clinic to make an appointment with a physician. She ends the consultation and the system re-encrypts Jim's record. After, finishing her consultations, Kate returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for Kate of those patients who she has to contact for follow-up information and make clinic appointments.

Figure 8.10 A usage scenario for the MHC-PMS

8.3.3 PERFORMANCE TESTING

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Tests should reflect the profile of use of the system.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behavior.

8.4 USER TESTING

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

Three Types of user testing

1. **Alpha testing:** where Users of the software work with the development team to test the software at the developer's site.
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.
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.

Figure 8.11 The acceptance testing process

There are six Stages in the acceptance testing process

- **Define acceptance criteria:** *This stage should, ideally, take place early in the process before the contract for the system is signed. The acceptance criteria should be part of the system contract and be agreed between the customer and the developer. In practice, however, it can be difficult to define criteria so early in the process. Detailed requirements may not be available and there may be significant requirements change during the development process.*
- **Plan acceptance testing***: This involves deciding on the resources, time, and budget for acceptance testing and establishing a testing schedule. The acceptance test plan should also discuss the required coverage of the requirements and the order in which system features are tested. It should define risks to the testing process, such as system crashes and inadequate performance, and discuss how these risks can be mitigated.*
- **Derive acceptance tests:** *Once acceptance criteria have been established, tests have to be designed to check whether or not a system is acceptable. Acceptance tests should aim to test both the functional and non-functional characteristics (e.g., performance) of the system. They should, ideally, provide complete coverage of the system requirements. In practice, it is difficult to establish completely objective acceptance criteria. There is often scope for argument about whether or not a test shows that a criterion has definitely been met.*
- **Run acceptance tests**: *The agreed acceptance tests are executed on the system. Ideally, this should take place in the actual environment where the system will be used, but this may be disruptive and impractical. Therefore, a user testing environment may have to be set up to run these tests. It is difficult to automate this process as part of the acceptance tests may involve testing the interactions between end-users and the system. Some training of end-users may be required.*
- **Negotiate test results:** *It is very unlikely that all of the defined acceptance tests will pass and that there will be no problems with the system. If this is the case, then acceptance testing is complete and the system can be handed over. More commonly, some problems will be discovered. In such cases, the developer and*

the customer have to negotiate to decide if the system is good enough to be put into use. They must also agree on the developer's response to identified problems.

- **Reject/accept system**: *This stage involves a meeting between the developers and the customer to decide on whether or not the system should be accepted. If the system is not good enough for use, then further development is required to fix the identified problems. Once complete, the acceptance testing phase is repeated.*

✧

Agile methods and acceptance testing

- In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- There is no separate acceptance testing process.
- Main problem here is whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders.

Key points

- 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.
- 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.
- Test-first development is an approach to development where tests are written before the code to be tested.
- Scenario testing involves inventing a typical usage scenario and using this to derive test cases.
- 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.

Topics covered

- ✓ Evolution processes - Change processes for software systems
- ✓ Program evolution dynamics -Understanding software evolution
- ✓ Software maintenance -Making changes to operational software systems
- ✓ Legacy system management -Making decisions about software change

Software change

- ➢ Software change is inevitable
  - ✓ New requirements emerge when the software is used;
  - ✓ The business environment changes;
  - ✓ Errors must be repaired;
  - ✓ New computers and equipment is added to the system;
  - ✓ The performance or reliability of the system may have to be improved.
- ➢ A key problem for all organizations is implementing and managing change to their existing software systems.

Importance of evolution

- ➢ Organizations have huge investments in their software systems - they are critical business assets.
- ➢ To maintain the value of these assets to the business, they must be changed and updated.
- ➢ The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software.

A spiral model of development and evolution

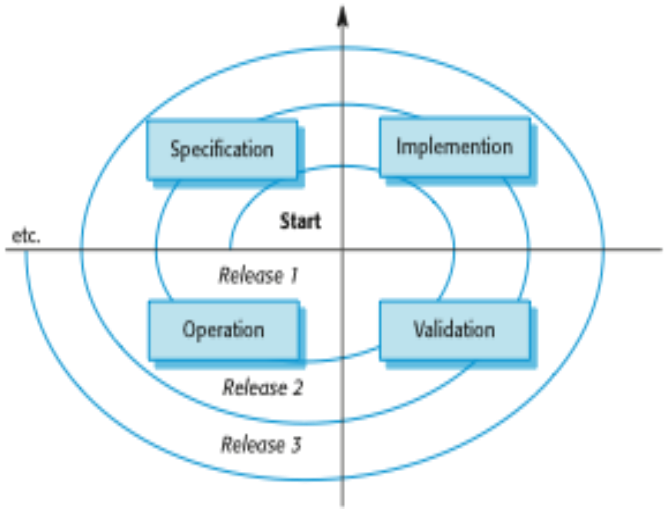


Figure 9.1 A spiral model of development and evolution

Evolution and servicing

Figure 9.2 Evolution and servicing

Evolution and servicing

- Evolution
  - ✓ The stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.
- Servicing
  - ✓ At this stage, the software remains useful but the only changes made are those required to keep it operational i.e. bug fixes and changes to reflect changes in the software's environment. No new functionality is added.
- Phase-out
  - ✓ The software may still be used but no further changes are made to it.

9.1 EVOLUTION PROCESSES

- Software evolution processes depend on
  - ✓ The type of software being maintained;
  - ✓ The development processes used;
  - ✓ The skills and experience of the people involved.
- Proposals for change are the driver for system evolution.
  - ✓ Should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated.
- Change identification and evolution continues throughout the system lifetime.

Change identification and evolution processes

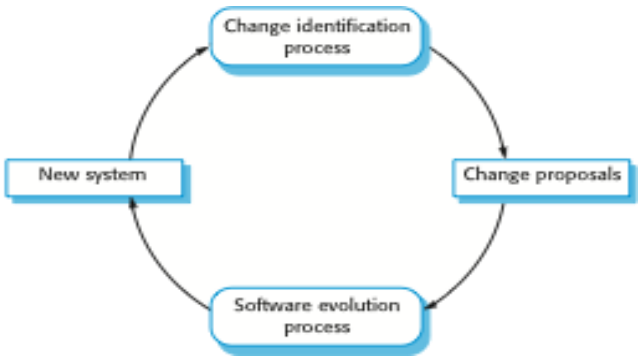


Figure 9.3 Change identification and evolution processes

The software evolution process

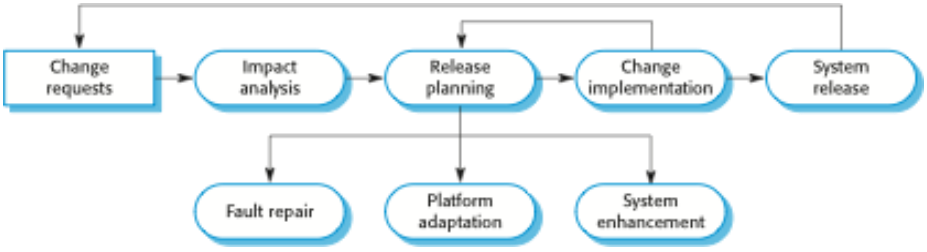


Figure 9.4 The software evolution process

Change implementation

Figure 9.5 Change implementation

Change implementation

- Iteration of the development process where the revisions to the system are designed, implemented and tested.
- A critical difference is that the first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for the change implementation.
- During the program understanding phase, you have to understand how the program is structured, how it delivers functionality and how the proposed change might affect the program.

Urgent change requests

- 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 to allow normal operation to continue;
  - ✓ 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

Figure 9.6 The emergency repair process

Agile methods and evolution

- Agile methods are based on incremental development so the transition from development to evolution is a seamless one.
  - ✓ Evolution is simply a continuation of the development process based on frequent system releases.
- Automated regression testing is particularly valuable when changes are made to a system.
- Changes may be expressed as additional user stories.

Handover problems

- Where the development team have used an agile approach but the evolution team is unfamiliar with agile methods and prefer a plan-based approach.

- ✓ The evolution team may expect detailed documentation to support evolution and this is not produced in agile processes.

- Where a plan-based approach has been used for development but the evolution team prefer to use agile methods.

  - ✓ The evolution team may have to start from scratch developing automated tests and the code in the system may not have been refactored and simplified as is expected in agile development.

9.2 PROGRAM EVOLUTION DYNAMICS

- *Program evolution dynamics* is the study of the processes of system change.
- After several major empirical studies, Lehman and Belady proposed that there were a number of 'laws' which applied to all systems as they evolved.
- There are sensible observations rather than laws. They are applicable to large systems developed by large organisations.
  - ✓ It is not clear if these are applicable to other types of software system.

Change is inevitable

- The system requirements are likely to change while the system is being developed because the environment is changing. Therefore a delivered system won't meet its requirements!
- Systems are tightly coupled with their environment. When a system is installed in an environment it changes that environment and therefore changes the system requirements.
- Systems MUST be changed if they are to remain useful in an environment.

Lehman's laws

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.
Law	Description
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.

Figure 9.7 Lehman's laws

Applicability of Lehman's laws

- Lehman's laws seem to be generally applicable to large, tailored systems developed by large organisations.
  - Confirmed in early 2000's by work by Lehman on the FEAST project.
- It is not clear how they should be modified for
  - ✓ Shrink-wrapped software products;
  - ✓ Systems that incorporate a significant number of COTS components;
  - ✓ Small organisations;
  - ✓ Medium sized systems.

Key points

- Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
- For custom systems, the costs of software maintenance usually exceed the software development costs.
- The process of software evolution is driven by requests for changes and includes change impact analysis, release planning and change implementation.
- Lehman's laws, such as the notion that change is continuous, describe a number of insights derived from long-term studies of system evolution.

9.3 SOFTWARE MAINTENANCE

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

Types of maintenance

- Maintenance to repair software faults
  - ✓ Changing a system to correct deficiencies in the way meets its requirements.
- Maintenance to adapt software to a different operating environment
  - ✓ Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- Maintenance to add to or modify the system's functionality
  - ✓ Modifying the system to satisfy new requirements.

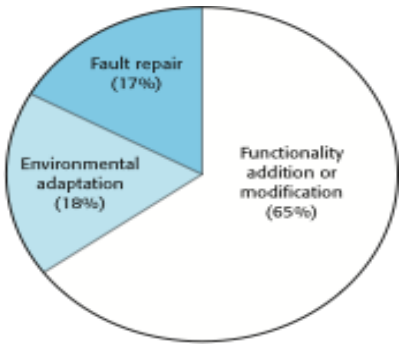


Figure 9.8 Maintenance effort distributions

Maintenance costs

- Usually greater than development costs (2* to 100* depending on the application).
- Affected by both technical and non-technical factors.
- Increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult.
- Ageing software can have high support costs (e.g. old languages, compilers etc.).

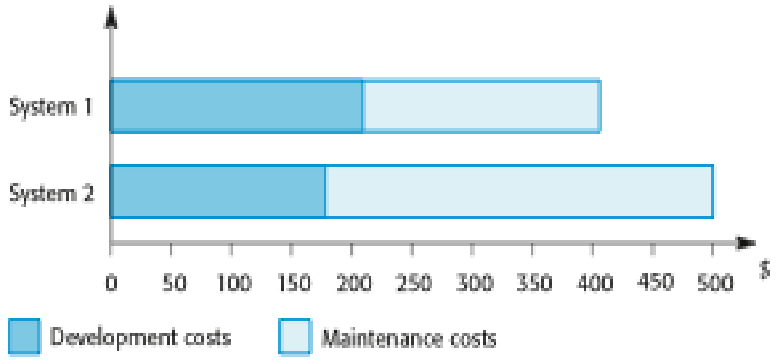


Figure 9.9 Development and maintenance costs

Maintenance cost factors

- Team stability
  - ✓ Maintenance costs are reduced if the same staff are involved with them for some time.
- Contractual responsibility
  - ✓ The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.
- Staff skills
  - ✓ Maintenance staff is often inexperienced and have limited domain knowledge.
- Program age and structure
  - ✓ As programs age, their structure is degraded and they become harder to understand and change.

9.3.1 MAINTENANCE PREDICTION

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

- Change acceptance depends on the maintainability of the components affected by the change;
- Implementing changes degrades the system and reduces its maintainability;
- Maintenance costs depend on the number of changes and costs of change depend on maintainability.

Maintenance prediction

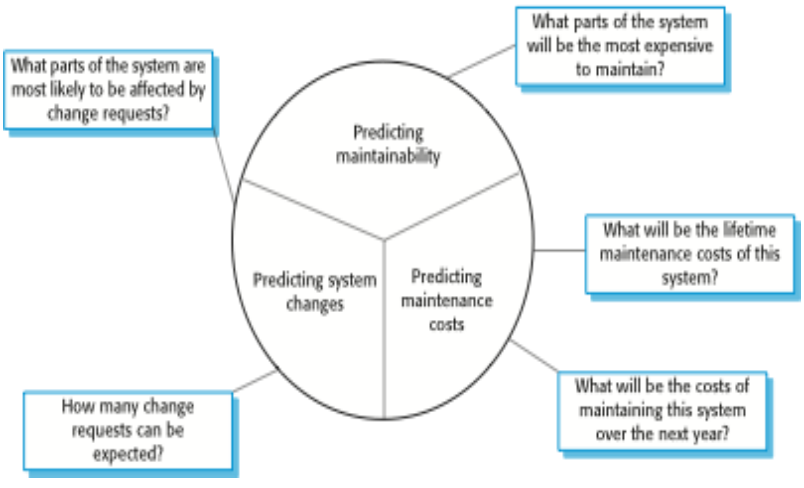


Figure 9.10 Maintenance prediction

Change prediction

- Predicting the number of changes requires and understanding of the relationships between a system and its environment.
- Tightly coupled systems require changes whenever the environment is changed.
- Factors influencing this relationship are
  - ✓ Number and complexity of system interfaces;
  - ✓ Number of inherently volatile system requirements;
  - ✓ The business processes where the system is used.

Complexity metrics

- Predictions of maintainability can be made by assessing the complexity of system components.
- Studies have shown that most maintenance effort is spent on a relatively small number of system components.
- Complexity depends on
  - ✓ Complexity of control structures;
  - ✓ Complexity of data structures;
  - ✓ Object, method (procedure) and module size.

Process metrics

- Process metrics may be used to assess maintainability
  - ✓ Number of requests for corrective maintenance;
  - ✓ Average time required for impact analysis;
  - ✓ Average time taken to implement a change request;
  - ✓ Number of outstanding change requests.
- If any or all of these is increasing, this may indicate a decline in maintainability.

9.3.2 SOFTWARE RE-ENGINEERING

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

Advantages of reengineering

- Reduced risk
  - ✓ There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- Reduced cost
  - ✓ The cost of re-engineering is often significantly less than the costs of developing new software.

The reengineering process

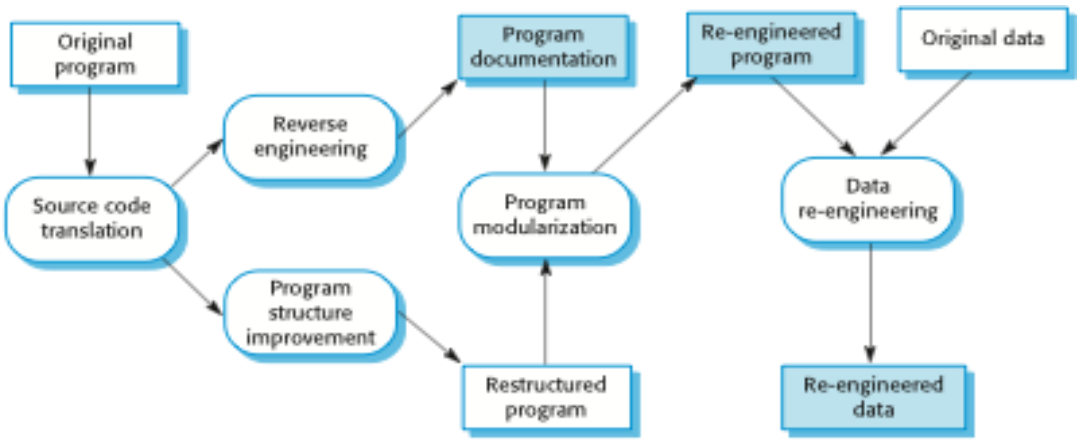


Figure 9.11 The reengineering process

Reengineering process activities

- Source code translation
  - ✓ Convert code to a new language.
- Reverse engineering
  - ✓ Analyze the program to understand it;
- Program structure improvement
  - ✓ Restructure automatically for understandability;
- Program modularization
  - ✓ Reorganize the program structure;
- Data reengineering
  - ✓ Clean-up and restructure system data.

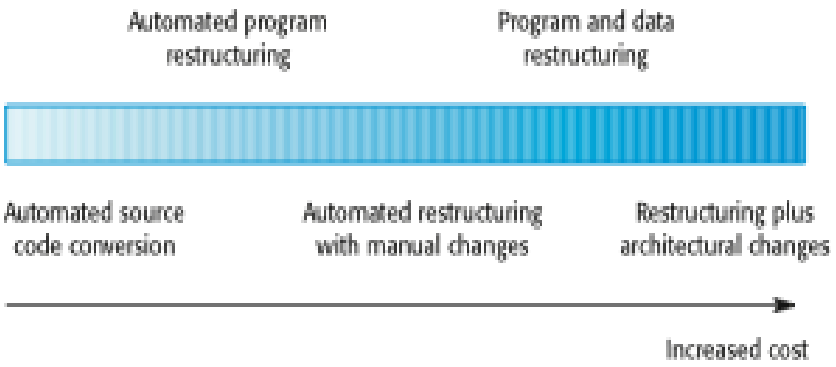


Figure 9.12 Reengineering approaches

Reengineering cost factors

- The quality of the software to be reengineered.
- The tool support available for reengineering.
- The extent of the data conversion which is required.
- The availability of expert staff for reengineering.
  - ✓ This can be a problem with old systems based on technology that is no longer widely used**.**

Preventative maintenance by refactoring

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

Refactoring and reengineering

- 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.

- 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.

'Bad smells' in program code

- **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.
- **Long methods**
  - ✓ If a method is too long, it should be redesigned as a number of shorter methods.
- **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.
- **Data clumping**
  - ✓ 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.
- **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.

9.4 LEGACY SYSTEM MANAGEMENT

- **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.

- **The strategy chosen should depend on the system quality and its business value.**

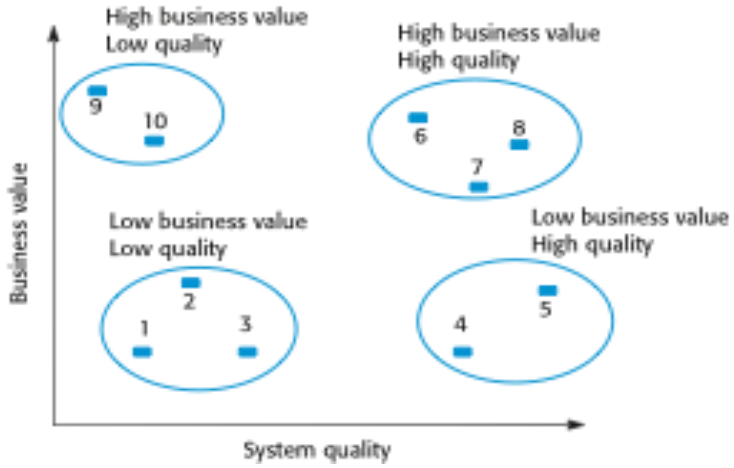


Figure 9.13 An example of a legacy system assessment

Legacy system categories/clusters

1. Low quality, low business value
   - ✓ These systems should be scrapped.
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.
3. High-quality, low-business value
   - ✓ Replace with COTS, scrap completely or maintain.
4. High-quality, high business value
   - ✓ Continue in operation using normal system maintenance.

Business value assessment

- **Assessment should take different viewpoints into account**
  - ✓ System end-users;
  - ✓ Business customers;
  - ✓ Line managers;
  - ✓ IT managers;
  - ✓ Senior managers.
- **Interview different stakeholders and collate results.**

Issues in business value assessment

1. The use of the system
   - ✓ If systems are only used occasionally or by a small number of people, they may have a low business value.
2. The business processes that are supported
   - ✓ A system may have a low business value if it forces the use of inefficient business processes.
3. System dependability
   - ✓ If a system is not dependable and the problems directly affect business customers, the system has a low business value.
4. The system outputs
   - ✓ If the business depends on system outputs, then the system has a high business value.

System quality assessment

- Business process assessment
  - ✓ How well does the business process support the current goals of the business?
- Environment assessment
  - ✓ How effective is the system's environment and how expensive is it to maintain?
- Application assessment
  - ✓ What is the quality of the application software system?

Business process assessment

- Use a viewpoint-oriented approach and seek answers from system stakeholders
  - ✓ Is there a defined process model and is it followed?
  - ✓ Do different parts of the organisation use different processes for the same function?
  - ✓ How has the process been adapted?
  - ✓ What are the relationships with other business processes and are these necessary?
  - ✓ Is the process effectively supported by the legacy application software?
- Example - a travel ordering system may have a low business value because of the widespread use of web-based ordering.

Factors used in environment assessment

FACTOR	QUESTIONS
Supplier stability	Is the supplier still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly but there could be significant economic and business benefits to moving to a more modern system.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?

Support requirements	What local support is required by the hardware and software? If there are high costs associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licences? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system? Is hardware emulation required?

Figure 9.14 Factors used in environment assessment

Factors used in application assessment

FACTOR	QUESTIONS
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent, and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up to date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there people available who have experience with the system?

Figure 9.15 Factors used in application assessment

System measurement

- You may collect quantitative data to make an assessment of the quality of the application system
    - ✓ The number of system change requests;
    - ✓ The number of different user interfaces used by the system;
    - ✓ The volume of data used by the system.

Key points

- There are 3 types of software maintenance, namely bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.
- Software re-engineering is concerned with re-structuring and re-documenting software to make it easier to understand and change.
- Refactoring, making program changes that preserve functionality, is a form of preventative maintenance.
- 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.

MODULE 5 PROJECT PLANNING AND QUALITY MANAGEMENT

23 PROJECT PLANNING

- Project planning is one of the most important jobs of a software project manager.
- The project plan, which is created at the start of a project, is used to communicate how the work will be done to the project team and customers, and to help assess progress on the project.
- Project planning takes place at three stages in a project life cycle:
  1. At the proposal stage, when there is bidding for a contract to develop or provide a software system, a plan may be required at this stage to help contractors decide if they have the resources to complete the work and to work out the price that they should quote to a customer.
  2. During the project startup phase, there is a need to plan who will work on the project, how the project will be broken down into increments, how resources will be allocated across your company, etc.
  3. Periodically throughout the project, when the plan is modified, in light of experience gained and information from monitoring the progress of the work, more information about the system being implemented and capabilities of development team are learnt.
- This information allows to make more accurate estimates of how long the work will take.

23.1 Software Pricing

- In principle, the price of a software product to a customer is simply the cost of development plus profit for the developer. Fig 4.1 shows the factors affecting software pricing
- It is essential to think about organizational concerns, the risks associated with the project, and the type of contract that will be used.
- These may cause the price to be adjusted upwards or downwards.
- Because of the organizational considerations involved, deciding on a project price should be a group activity involving marketing and sales staff, senior management,and project managers

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

Fig 4.1: Factors affecting software pricing

23.2 Plan Driven Development

- Plan-driven or plan-based development is an approach to software engineering where the development process is planned in detail.
- A project plan is created that records the work to be done, who will do it, the development schedule, and the work products.
- Managers use the plan to support project decision making and as a way of measuring progress.
- Plan-driven development is based on engineering project management techniques and can be thought of as the ‘traditional’ way of managing large software development projects.

- The principal argument against plan-driven development is that many early decisions have to be revised because of changes to the environment in which the software is to be developed and used

23.2.1 Project Plans

- In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown, and a schedule for carrying out the work.
- The plan should identify risks to the project and the software under development, and the approach that is taken to risk management.
- Plans normally include the following sections:

23.2.1.1 **Introduction:** Describes the objectives of the project and sets out the constraints (e.g., budget, time, etc.) that affect the management of the project.

23.2.1.2 **Project organization:** This describes the way in which the development team is organized, the people involved, and their roles in the team.

23.2.1.3 **Risk analysis:** This describes possible project risks, the likelihood of these risks arising, and the risk reduction strategies that are proposed.

23.2.1.4 **Hardware and software resource requirements:** This specifies the hardware and support software required to carry out the development. If hardware has to be bought, estimates of the prices and the delivery schedule may be included.

23.2.1.5 **Work breakdown:** This sets out the breakdown of the project into activities and identifies the milestones and deliverables associated with each activity.

23.2.1.6 **Project schedule:** Shows dependencies between activities, the estimated time required to reach each milestone, and the allocation of people to activities.

23.2.1.7 **Monitoring and reporting mechanisms:** This defines the management reports that should be produced, when these should be produced, and the project monitoring mechanisms to be used.

- The project plan supplements are as shown in fig 4.2.

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

Fig 4.2: Project plan supplements

23.2.2 The Planning Process

- Project planning is an iterative process that starts when an initial project plan is created during the project startup phase.
- Fig 4.3 is a UML activity diagram that shows a typical workflow for a project planning process.
- At the beginning of a planning process, it is necessary to assess the constraints affecting the project.
- These constraints are the required delivery date, staff available, overall budget, available tools, and so on.
- It is also necessary to identify the project milestones and deliverables.
- Milestones are points in the schedule against which progress can be accessed, for example, the handover of the system for testing.
- Deliverables are work products that are delivered to the customer.
- The process then enters a loop.

- You draw up an estimated schedule for the project and the activities defined in the schedule are initiated or given permission to continue.
- After some time (usually about two to three weeks), the progress must be reviewed and discrepancies must be noted from the planned schedule.
- Because initial estimates of project parameters are inevitably approximate, minor slippages are normal and thereby modifications will have to be made to the original plan.

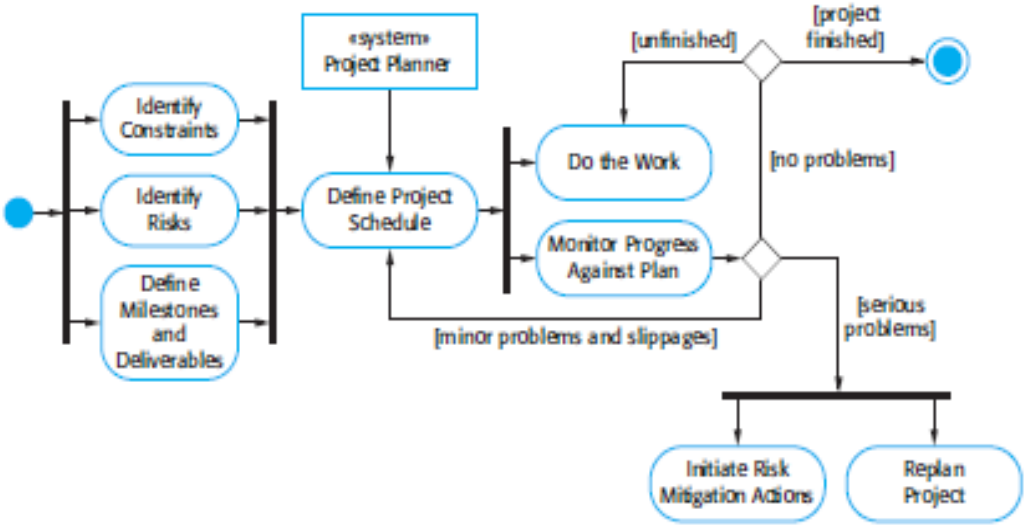


Fig 4.3: The project planning process

23.3 Project Scheduling

- Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed.
- Here there is an estimation of the calendar time needed to complete each task, the effort required, and who will work on the tasks that have been identified.
- It is essential to estimate the resources needed to complete each task, such as the disk space required on a server, the time required on specialized hardware, such as a simulator, and what the travel budget will be.
- Scheduling in plan-driven projects (Fig 4.4) involves breaking down the total work involved in a project into separate tasks and estimating the time required to complete each task.
- Tasks should normally last at least a week, and no longer than 2 months.
- Finer subdivision means that a disproportionate amount of time must be spent on re- planning and updating the project plan.
- The maximum amount of time for any task should be around 8 to 10 weeks.
- If it takes longer than this, the task should be subdivided for project planning and scheduling.

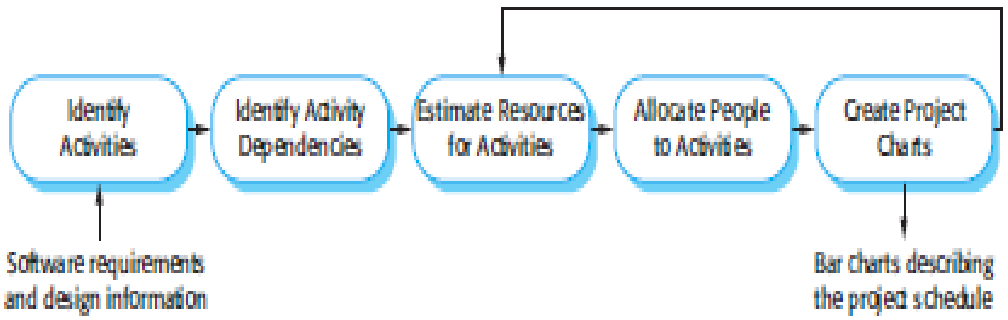


Fig 4.4: The project scheduling process

23.3.1 Schedule Representation

- Project schedules may simply be represented in a table or spreadsheet showing the tasks, effort, expected duration, and task dependencies.
- There are two types of representation that are commonly used:
  - 23.3.1.1 Bar charts, which are calendar-based, show who is responsible for each activity, the expected elapsed time, and when the activity is scheduled to begin and end. Bar charts are sometimes called 'Gantt charts'.
  - 23.3.1.2 Activity networks, which are network diagrams, show the dependencies between the different activities making up a project.
- Project activities are the basic planning element. Each activity has:

1. A duration in calendar days or months.
2. An effort estimate, which reflects the number of person-days or person- months to complete the work.
3. A deadline by which the activity should be completed.
4. A defined endpoint. This represents the tangible result of completing the activity. This could be a document, the holding of a review meeting, the successful execution of all tests, etc.

- When planning a project, milestones must also be defined; that is, each stage in the project where a progress assessment can be made.
- Each milestone should be documented by a short report that summarizes the progress made and the work done.
- Milestones may be associated with a single task or with groups of related activities.
- For example, in fig 4.5, milestone M1 is associated with task T1 and milestone M3 is associated with a pair of tasks, T2 and T4.
- A special kind of milestone is the production of a project deliverable.
- A deliverable is a work product that is delivered to the customer. It is the outcome of a significant project phase such as specification or design.
- Usually, the deliverables that are required are specified in the project contract and the customer's view of the project's progress depends on these deliverables.

Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

Fig 4.5: Tasks, durations and dependencies

- Fig 4.6 takes the information in Figure 4.5 and presents the project schedule in a graphical format.

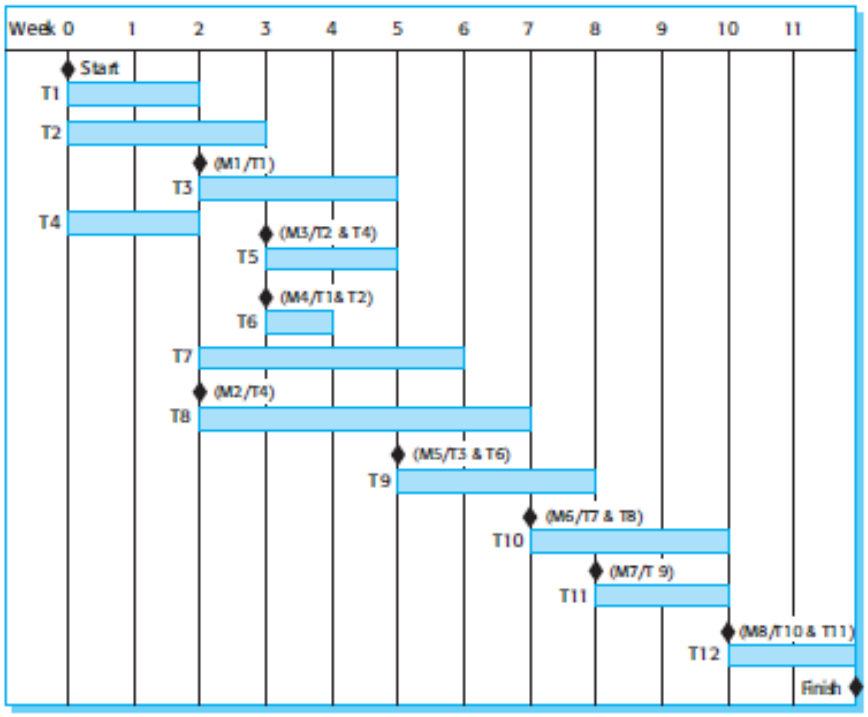


Fig 4.6: Activity bar chart

- It is a bar chart showing a project calendar and the start and finish dates of tasks.
- Reading from left to right, the bar chart clearly shows when tasks start and end.
- The milestones (M1, M2, etc.) are also shown on the bar chart
- In Fig 4.7, it is observed that Mary is a specialist, who works on only a single task in the project.

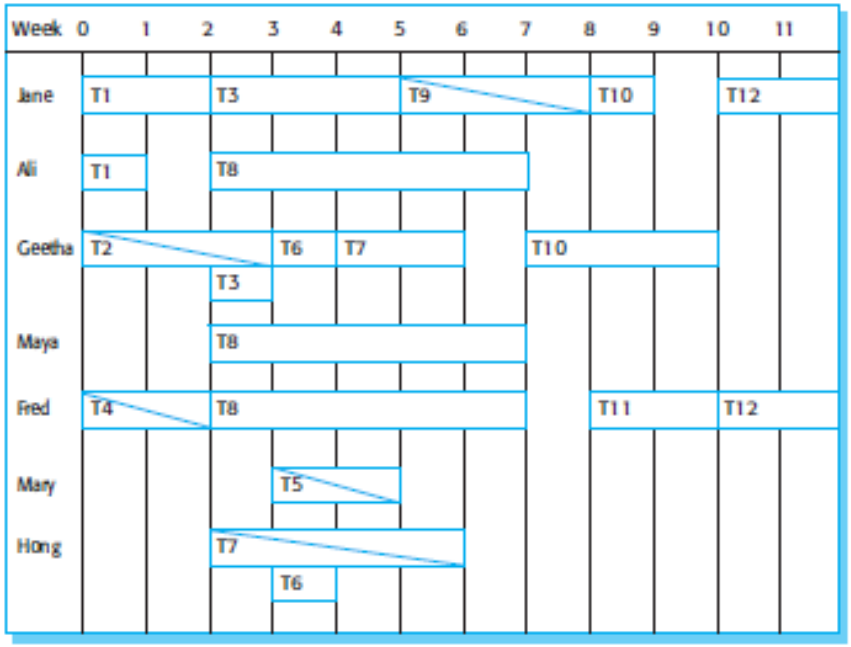


Fig 4.7: Staff allocation chart

- This can cause scheduling problems. If one project is delayed while a specialist is working on it, this may have a knock-on effect on other projects where the specialist is also required.
- These may then be delayed because the specialist is not available.
- Delays can cause serious problems with staff allocation, especially when people are working on several projects at the same time.
- If a task (T) is delayed, the people allocated may be assigned to other work (W).
- To complete this may take longer than the delay but, once assigned, they cannot simply be reassigned back to the original task, T.
- This may then lead to further delays in T as they complete W.

23.5 Estimation Techniques

- Project schedule estimation is difficult.
- There might be a need to make initial estimates on the basis of a high-level user requirements definition.
- Organizations need to make software effort and cost estimates.
- There are two types of technique that can be used to do this:
  1. **Experience-based Techniques**: The estimate of future effort requirements is based on the manager's experience of past projects and the application domain.
  2. **Algorithmic cost Modeling**: In this approach, a formulaic approach is used to compute the project effort based on estimates of product attributes, such as size, and process characteristics, such as experience of staff involved.
- During development planning, estimates become more and more accurate as the project progresses (Fig 4.8).
- Experience-based techniques rely on the manager's experience of past projects and the actual effort expended in these projects on activities that are related to software development.
- The difficulty with experience-based techniques is that a new software project may not have much in common with previous projects.

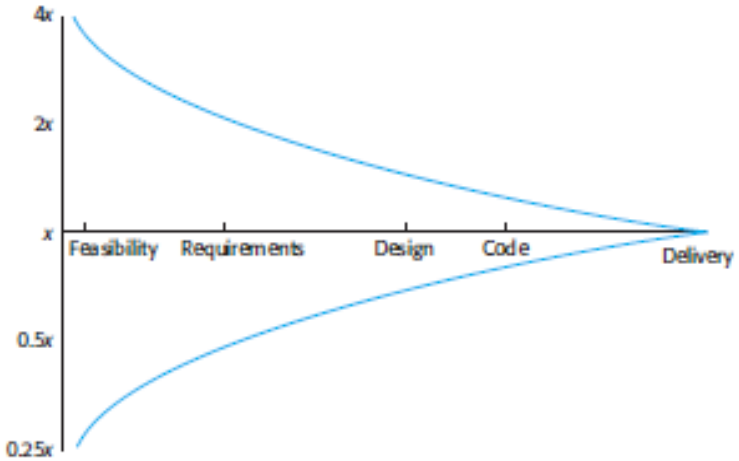


Fig 4.8: Estimate Uncertainty

23.5.1 Algorithmic Cost Modeling

- Algorithmic cost modeling uses a mathematical formula to predict project costs based on estimates of the project size; the type of software being developed; and other team, process, and product factors.
- An algorithmic cost model can be built by analyzing the costs and attributes of completed projects, and finding the closest-fit formula to actual experience.
- Algorithmic models for estimating effort in a software project are mostly based on a simple formula:

$$\textbf{Effort} = A * \text{Size}^{B} * M$$

- A is a constant factor which depends on local organizational practices and the type of software that is developed.
- Size may be either an assessment of the code size of the software or a functionality estimate expressed in function or application points.
- The value of exponent B usually lies between 1 and 1.5. M is a multiplier made by combining process, product, and development attributes, such as the dependability requirements for the software and the experience of the development team.
- All algorithmic models have similar problems:
  1. It is often difficult to estimate Size at an early stage in a project, when only the specification is available. Function-point and application-point estimates are easier to produce than estimates of code size but are still often inaccurate.
  2. The estimates of the factors contributing to B and M are subjective. Estimates vary from one person to another, depending on their background and experience of the type of system that is being developed

23.5.2 The COCOMO II Model

- This is an empirical model that was derived by collecting data from a large number of software projects.
- These data were analyzed to discover the formulae that were the best fit to the observations.
- The COCOMO II model takes into account more modern approaches to software development, such as rapid development using dynamic languages, development by component composition, and use of database programming.
- COCOMO II supports the spiral model of development.
- The sub-models (Fig 4.9) that are part of the COCOMO II model are:

1. **An application-composition model**: Models the effort required to develop systems that are created from reusable components, scripting, or database programming. Software size estimates are based on application points, and a simple size/productivity formula is used to estimate the effort required.
2. **An early design model**: This model is used during early stages of the system design after the requirements have been established.
3. **A reuse model**: This model is used to compute the effort required to integrate reusable components and/or automatically generated program code. It is normally used in conjunction with the post-architecture model.
4. **A post-architecture model:** Once the system architecture has been designed, a more accurate estimate of the software size can be made. Again, this model uses the standard formula for cost estimation discussed above.

- **The Application-Composition Model**
  - ✓ The application-composition model was introduced into COCOMO II to support the estimation of effort required for prototyping projects and for projects where the software is developed by composing existing components.
  - ✓ It is based on an estimate of weighted application points (sometimes called object points), divided by a standard estimate of application point productivity.
  - ✓ The estimate is then adjusted according to the difficulty of developing each application point.
  - ✓ Application composition usually involves significant software reuse.
  - It is almost certain that some of the application points in the system will be implemented using reusable components the final formula for effort computation for system prototypes is:

$$PM = (NAP * (1 - \%reuse / 100)) / PROD$$

Where,

PM → the effort estimate in person-months.
NAP → the total number of application points in the delivered system.
%reuse→ an estimate of the amount of reused code in the development.

- **The Early Design Model**
  - This model may be used during the early stages of a project, before a detailed architectural design for the system is available.
  - Early design estimates are most useful for option exploration where you need to compare different ways of implementing the user requirements.
  - The estimates produced at this stage are based on the standard formula for algorithmic models, namely:

$$\textbf{Effort} = A * \text{Size}^{B} * M$$

  - Boehm proposed that the coefficient A should be 2.94.
  - The exponent B reflects the increased effort required as the size of the project increases.
  - This can vary from 1.1 to 1.24 depending on the novelty of the project, the development flexibility, the risk resolution processes used, the cohesion of the development team, and the process maturity level of the organization.
  - This results in an effort computation as follows:

$$\textbf{PM} = 2.94 * \text{Size}^{(1.1 - 1.24)} * M$$

Where

$$M = PERS * RCPX * RUSE * PDIF * PREX * FCIL * SCED$$

  - The multiplier M is based on seven project and process attributes that increase or decrease the estimate.
  - The attributes used in the early design model are product reliability and complexity (RCPX), reuse required (RUSE), platform difficulty (PDIF), personnel capability (PERS), personnel experience (PREX), schedule (SCED), and support facilities (FCIL).
- **The Reuse Model**
  - COCOMO II considers two types of reused code.
  - 'Black-box' code is code that can be reused without understanding the code or making changes to it.
  - The development effort for black-box code is taken to be zero.
  - 'White box' code has to be adapted to integrate it with new code or other reused components.
  - A model (often in UML) is analyzed and code is generated to implement the objects specified in the model.
  - The COCOMO II reuse model includes a formula to estimate the effort required to integrate this generated code:

$\textbf{PM}_{\text{Auto}} = \textbf{(ASLOC * AT / 100) / ATPROD}$ // Estimate for generated code Where,

ASLOC → total number of lines of reused code, including code that is automatically generated.

AT → percentage of reused code that is automatically generated. ATPROD → productivity of engineers in integrating such code.

  - If there are a total of 20,000 lines of reused source code in a system and 30% of this is automatically generated, then the effort required to integrate the generated code is:

(20.000 * 30/100) / 2400 = 2.5 person months //Generated Code

  - The following formula is used to calculate the number of equivalent lines of source code:

$$\textbf{ESLOC} = \textbf{ASLOC} * \textbf{AAM}$$

Where,

ESLOC → the equivalent number of lines of new source code.

ASLOC → the number of lines of code in the components that have to be changed.

AAM → an Adaptation Adjustment Multiplier (AAM) which adjusts the estimate to reflect the additional effort required to reuse code.

- AAM is the sum of three components:

23.5.2.4.1 An adaptation component (referred to as AAF) that represents the costs of making changes to the reused code. The adaptation component includes subcomponents that take into account design, code, and integration changes.

23.5.2.4.2 An understanding component (referred to as SU) that represents the costs of understanding the code to be reused and the familiarity of the engineer with the code. SU ranges from 50 for complex unstructured code to 10 for well-written, object-oriented code.

23.5.2.4.3 An assessment factor (referred to as AA) that represents the costs of reuse decision making. That is, some analysis is always required to decide whether or not code can be reused, and this is included in the cost as AA. AA varies from 0 to 8 depending on the amount of analysis effort required.

- **The Post-Architecture Level**
  - The post-architecture model is the most detailed of the COCOMO II models.
  - It is used once an initial architectural design for the system is available so the subsystem structure is known.
  - Estimation for each part of the system can then be made.
  - The starting point for estimates produced at the post-architecture level is the same basic formula used in the early design estimates:
  - Estimate of the code size can be done using three parameters:
    i. An estimate of the total number of lines of new code to be developed (SLOC).
    ii. An estimate of the reuse costs based on an equivalent number of source lines of code (ESLOC), calculated using the reuse model.
    iii. An estimate of the number of lines of code that are likely to be modified because of changes to the system requirements. The value of the exponent B is based on five factors, as shown in Fig 4.9 These factors are rated on a six-point scale from 0 to 5, where 0 means 'extra high' and 5 means 'very low'.

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

Fig 4.9: Scale factors used in the exponent computation in the post-architecture model

- Possible values for the ratings used in exponent calculation are therefore:
  i. **Precedentedness**, rated low (4). This is a new project for the organization.
  ii. **Development flexibility**, rated very high (1). No client involvement in the development process so there are few externally imposed changes
  iii. **Architecture/risk resolution**, rated very low (5). There has been no risk analysis carried out.
  iv. **Team cohesion**, rated nominal (3). This is a new team so there is no information available on cohesion.
  v. **Process maturity**, rated nominal (3). Some process control is in place.

- Fig 4.10 shows how the cost driver attributes can influence effort estimates.

Exponent value	1.17
System size (including factors for reuse and requirements volatility)	128,000 DSI
Initial COCOMO estimate without cost drivers	**730 person-months**
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
Adjusted COCOMO estimate	**2,306 person-months**
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	**295 person-months**

Fig 4.10: The effect of cost drivers on effort estimates

23.5.3 Project Duration and Staffing

- The COCOMO model includes a formula to estimate the calendar time required to complete a project:
- TDEV is the nominal schedule for the project, in calendar months, ignoring any multiplier that is related to the project schedule.
- PM is the effort computed by the COCOMO model. B is the complexity-related Exponent
- There is a complex relationship between the number of people working on a project, the effort that will be devoted to the project, and the project delivery schedule.
- If four people can complete a project in 13 months (i.e., 52 person-months of effort), then you might think that by adding one more person, you can complete the work in 11 months (55 person-months of effort).
- The COCOMO model suggests that you will, in fact, need six people to finish the work in 11 months (66 person-months of effort).
- The reason for this is that adding people actually reduces the productivity of existing team members and so the actual increment of effort added is less than one person.
- As the project team increases in size, team members spend more time communicating and defining interfaces between the parts of the system developed by other people.
- Doubling the number of staff (for example) therefore does not mean that the duration of the project will be halved

24. QUALITY MANAGEMENT

- Software quality management for software systems has three principal concerns:
  1. At the organizational level, quality management is concerned with establishing a framework of organizational processes and standards that will lead to high quality software. This means that the quality management team should take responsibility for defining the software development processes to be used and standards that should apply to the software and related documentation, including the system requirements, design, and code.
  2. At the project level, quality management involves the application of specific quality processes, checking that these planned processes have been followed,and ensuring that the project outputs are conformant with the standards that are applicable to that project.
  3. Quality management at the project level is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.
- Quality assurance (QA) is the definition of processes and standards that should lead to high-quality products and the introduction of quality processes into the manufacturing process.
- Quality control is the application of these quality processes to weed out products that are not of the required level of quality.
- Quality management provides an independent check on the software development process.
- The quality management process checks the project deliverables to ensure that they are consistent with organizational standards and goals

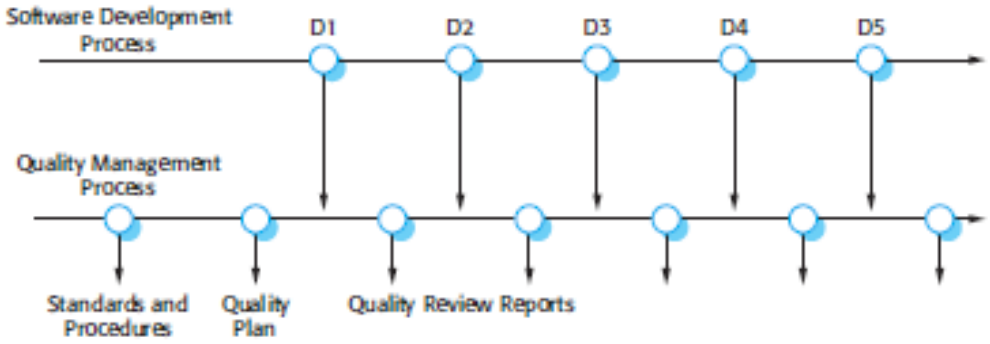


Fig 4.11: Quality management and software development

- An outline structure for quality plan includes:

1. **Product introduction:** A description of the product, its intended market, and the quality expectations for the product.
2. **Product plans:** The critical release dates and responsibilities for the product, along with plans for distribution and product servicing
3. **Process descriptions:** The development and service processes and standards that should be used for product development and management.
4. **Quality goals:** The quality goals and plans for the product, including an identification and justification of critical product quality attributes.
5. **Risks and risk management:** The key risks that might affect product quality and the actions to be taken to address these risks.

24.1 Software Quality

→ Different software quality attributes are as shown below

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

Fig 4.12: Software quality attributes

- A manufacturing process involves configuring, setting up, and operating the machines involved in the process.
- Once the machines are operating correctly, product quality naturally follows
- The quality of the product is measured and the process is changed until the quality level needed is achieved.
- Fig 4.13 illustrates this process-based approach to achieving product quality.

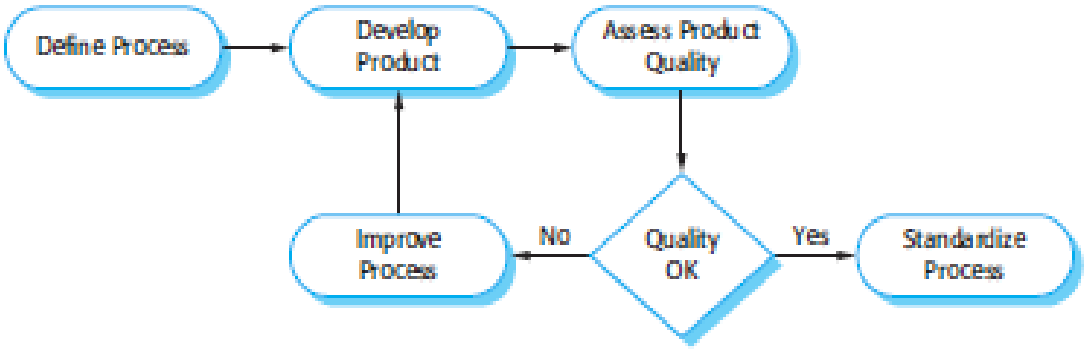


Fig 4.13: Process based quality

24.2 Software Standards

- Software standards are important for three reasons:
  1. Standards capture wisdom that is of value to the organization. They are based on knowledge about the best or most appropriate practice for the company. This knowledge is often only acquired after a great deal of trial and error.
  2. Standards provide a framework for defining what 'quality' means in a particular setting. This depends on setting standards that reflect user expectations for software dependability, usability, and performance.
  3. Standards assist continuity when work carried out by one person is taken up and continued by another. Standards ensure that all engineers within an organization adopt the same practices.
- There are two related types of software engineering standard that may be defined and used in software quality management:

1. **Product Standards:**
   - ✓ These apply to the software product being developed.
   - ✓ They include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, and coding standards, which define how a programming language should be used.
2. **Process Standards:**
   - ✓ These define the processes that should be followed during software development.
   - ✓ They should encapsulate good development practice.
   - ✓ Process standards may include definitions of specification, design and validation processes, process support tools, and a description of the documents that should be written during these processes.

24.3 Reviews and Inspections

- Reviews and inspections are QA activities that check the quality of project deliverables.
- This involves examining the software, its documentation and records of the process to discover errors and omissions and to see if quality standards have been followed.
- During a review, a group of people examine the software and its associated documentation, looking for potential problems and non-conformance with standards.
- The review team makes informed judgments about the level of quality of a system or project deliverable.
- Project managers may then use these assessments to make planning decisions and allocate resources to the development process.
- Quality reviews are based on documents that have been produced during the software development process.

- The purpose of reviews and inspections is to improve software quality, not to assess the performance of people in the development team.
- Reviewing is a public process of error detection, compared with the more private component-testing process.

24.3.1 The Review Process

- Review process is structured into 3 phases:

1. **Pre-Review Activities:**
   - ✓ These are preparatory activities that are essential for the review to be effective.
   - ✓ Pre-review activities are concerned with review planning and review preparation.
   - ✓ Review planning involves setting up a review team, arranging a time and place for the review, and distributing the documents to be reviewed.
   - ✓ During review preparation, the team may meet to get an overview of the software to be reviewed
2. **The Review Meeting**:
   - ✓ During the review meeting, an author of the document or program being reviewed should 'walk through' the document with the review team.
   - ✓ The review itself should be relatively short—two hours at most. One team member should chair the review and another should formally record all review decisions and actions to be taken.
3. **Post-Review Activities**:
   - ✓ After a review meeting has finished, the issues and problems raised during the review must be addressed.
   - ✓ This may involve fixing software bugs, refactoring software so that it conforms to quality standards, or rewriting documents.

24.3.2 Program Inspections

- Program inspections are 'peer reviews' where team members collaborate to find bugs in the program that is being developed.
- Inspections may be part of the software verification and validation processes.→ they complement testing as they do not require the program to be executed.
- This means that incomplete versions of the system can be verified and that representations such as UML models can be checked.
- During an inspection, a checklist of common programming errors is often used to focus the search for bugs.
- This checklist may be based on examples from books or from knowledge of defects that are common in a particular application domain.
- Possible checks are as shown in fig 4.14.

Fault class	Inspection check
Data faults	• Are all program variables initialized before their values are used? • Have all constants been named? • Should the upper bound of arrays be equal to the size of the array or Size -1? • If character strings are used, is a delimiter explicitly assigned? • Is there any possibility of buffer overflow?
Control faults	• For each conditional statement, is the condition correct? • Is each loop certain to terminate? • Are compound statements correctly bracketed? • In case statements, are all possible cases accounted for? • If a break is required after each case in case statements, has it been included?
Input/output faults	• Are all input variables used? • Are all output variables assigned a value before they are output? • Can unexpected inputs cause corruption?
Interface faults	• Do all function and method calls have the correct number of parameters? • Do formal and actual parameter types match? • Are the parameters in the right order? • If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	• If a linked structure is modified, have all links been correctly reassigned? • If dynamic storage is used, has space been allocated correctly? • Is space explicitly deallocated after it is no longer required?
Exception management faults	• Have all possible error conditions been taken into account?

Fig 4.14: An inspection checklist

24.4 Software Measurement and Metrics

- Software measurement is concerned with deriving a numeric value or profile for an attribute of a software component, system, or process.
- The long-term goal of software measurement is to use measurement in place of reviews to make judgments about software quality.
- Using software measurement, a system could ideally be assessed using a range of metrics and, from these measurements, a value for the quality of the system could be inferred.
- Software metric is a characteristic of a software system, system documentation, or development process that can be objectively measured.
- Examples of metrics include the size of a product in lines of code.
- Software metrics may be either control metrics or predictor metrics.
- Control metrics support process management, and predictor metrics helps to predict characteristics of the software.
- Control metrics are usually associated with software processes.
- Examples of control or process metrics are the average effort and the time required to repair reported defects.
- Predictor metrics are associated with the software itself and are sometimes known as ‘product metrics.
- Both control and predictor metrics may influence management decision making, as shown in fig 4.15.

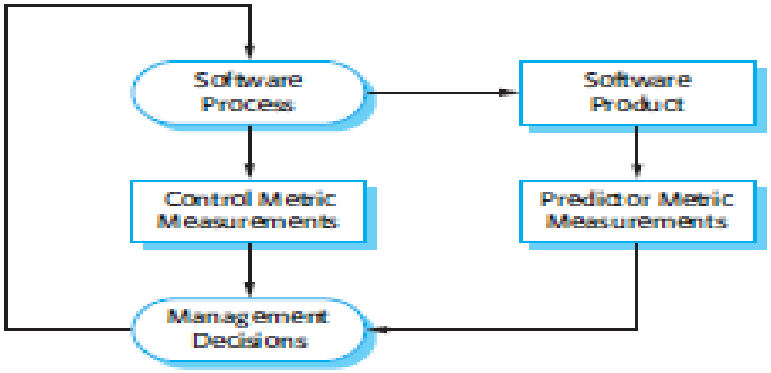


Fig 4.15: Predictor and control measurements

- There are two ways in which measurements of a software system may be used:
  1. **To assign a value to system quality attributes*:*** By measuring the characteristics of system components, such as their cyclomatic complexity, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.
  2. **To identify the system components whose quality is substandard:**

  Measurements can identify individual components with characteristics that deviate from the norm. For example, components can be measured to discover those with the highest complexity.
- Fig 4.16 shows some external software quality attributes and internal attributes that could, intuitively, be related to them.
- The diagram suggests that there may be relationships between external and internal attributes, but it does not say how these attributes are related.
- If the measure of the internal attribute is to be a useful predictor of the external software characteristic, three conditions must hold
  1. The internal attribute must be measured accurately. This is not always straightforward and it may require special-purpose tools to make the measurements.
  2. A relationship must exist between the attribute that can be measured and the external quality attribute that is of interest. That is, the value of the quality attribute must be related, in some way, to the value of the attribute than can be measured.
  3. This relationship between the internal and external attributes must be understood, validated, and expressed in terms of a formula or model. Model formulation involves identifying the functional form of the model (linear, exponential, etc.)by analysis of collected data, identifying the parameters that are to be included in the model, and calibrating these parameters using existing data.

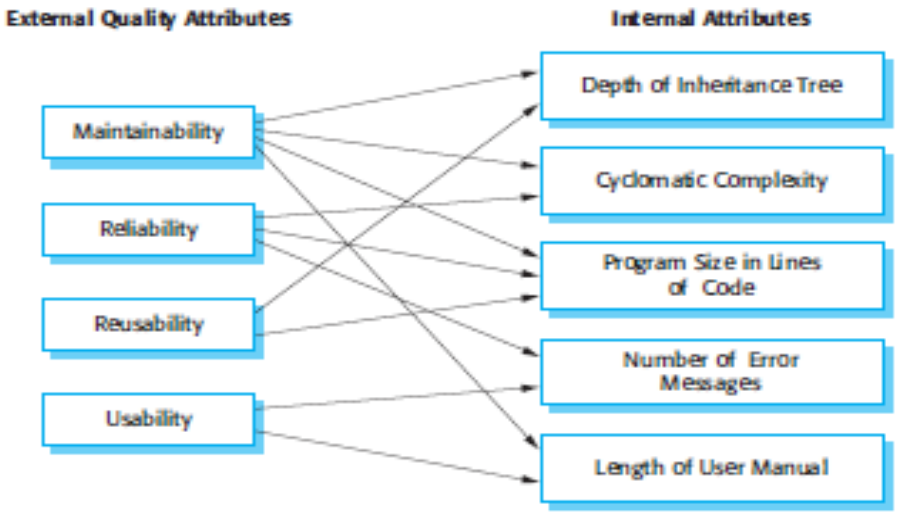


Fig 4.16: Relationships between internal and external software

24.4.1 Product Metrics

- Product metrics are predictor metrics that are used to measure internal attributes of a software system.
- Examples of product metrics include the system size, measured in lines of code, or the number of methods associated with each object class.
- Product metrics fall into two classes:

1. Dynamic metrics, which are collected by measurements made of a program in execution. These metrics can be collected during system testing or after the system has gone into use. An example might be the number of bug reports or the time taken to complete a computation.
2. Static metrics, which are collected by measurements made of representations of the system, such as the design, program, or documentation. Examples of static metrics are the code size and the average length of identifiers used.

- The metrics in fig 4.17 are applicable to any program but more specific object- oriented (OO) metrics have also been proposed

Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.
Length of identifiers	This is a measure of the average length of identifiers (names for variables, classes, methods, etc.) in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if-statements are hard to understand and potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value of a document's Fog index, the more difficult the document is to understand.

Fig 4.17: Static software product metrics

Fig 4.18 summarizes Chidamber and Kemerer's suite

Object-oriented metric	Description
Weighted methods per class (WMC)	This is the number of methods in each class, weighted by the complexity of each method. Therefore, a simple method may have a complexity of 1, and a large and complex method a much higher value. The larger the value for this metric, the more complex the object class. Complex objects are more likely to be difficult to understand. They may not be logically cohesive, so cannot be reused effectively as superclasses in an inheritance tree.
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.
Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent, and therefore it is more likely that changing one class will affect other classes in the program.
Response for a class (RFC)	RFC is a measure of the number of methods that could potentially be executed in response to a message received by an object of that class. Again, RFC is related to complexity. The higher the value for RFC, the more complex a class and hence the more likely it is that it will include errors.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

Fig 4.18: The CK object oriented metrics suite

24.4.2 Software Component Analysis

- A measurement process that may be part of a software quality assessment process is shown in fig 4.19.
- Each system component can be analyzed separately using a range of metrics.
- The values of these metrics may then be compared for different components and, perhaps, with historical measurement data collected on previous projects.
- The key stages in this component measurement process are:

1. **Choose measurements to be made**:
   - ✓ The questions that the measurement is intended to answer should be formulated and the measurements required to answer these questions defined. Measurements that are not directly relevant to these questions need not be collected.
2. ***Select components to be assessed**:*
   - ✓ You may not need to assess metric values for all of the components in a software system.
   - ✓ Sometimes, a representative selection of components can be selected for measurement, allowing to make an overall assessment of system quality.
3. **Measure component characteristics:**
   - ✓ The selected components are measured and the associated metric values computed.
   - ✓ This normally involves processing the component representation (design, code, etc.) using an automated data collection tool.
   - ✓ This tool may be specially written or may be a feature of design tools that are already in use.

4. **Identify anomalous measurements**:
   - ✓ After the component measurements have been made, it can be compared with each other and to previous measurements that have been recorded in a measurement database.
5. ***Analyze anomalous components****:*
   - ✓ When the components that have anomalous values for chosen metrics have been identified, it becomes necessary to examine them to decide whether or not these anomalous metric values mean that the quality of the component is compromised.
   - ✓ An anomalous metric value for complexity

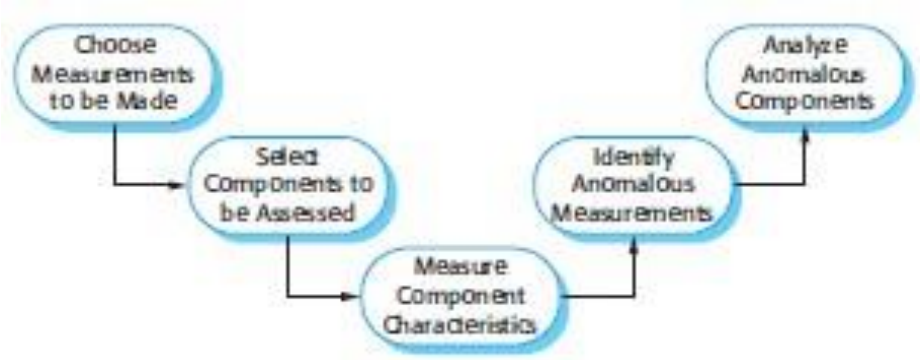


Fig 4.19: The process of product measurement

24.4.3 Measurement Ambiguity

- When you collect quantitative data about software and software processes, the data must be analyzed in order to be understood.
- It is easy to misinterpret data and to make inferences that are incorrect.
- There are several reasons for the users to make change requests:
  1. The software is not good enough and does not do what customers want it to do. They therefore request changes to deliver the functionality that they require.
  2. Alternatively, the software may be very good and so it is widely and heavily used. Change requests may be generated because there are many software users who creatively think of new things that could be done with the software.