Module 1

INTRODUCTION

- ❖ **Software Crisis**
- ❖ **Need for Software Engineering**
- ❖ **Professional Software Development**
- ❖ **Software Engineering Ethics**
- ❖ **Case Studies**

****

1. *INTRODUCTION*

- "We can't run the modern world without software"
- Industry, financial system, entertainment and so on….
- There are many different types of software systems, from simple embedded systems to complex, worldwide information systems.
- It is pointless to look for universal notations, methods, or techniques for software engineering because different types of software require different approaches
- Developing an organizational information system is completely different from developing a controller for a scientific instrument.

Software Failures

- **Increasing Demands**
    - Demands keep changing.
    - Systems have to be built and delivered more quickly.
    - Systems have to have new capabilities that were previously thought to be impossible.
    - Existing software engineering methods cannot cope and new software engineering techniques have to be developed to meet new these new demands.
- **Low Expectations**
    - It is relatively easy to write computer programs without using software engineering methods and techniques
    - Consequently, their software is often more expensive and less reliable than it should be.

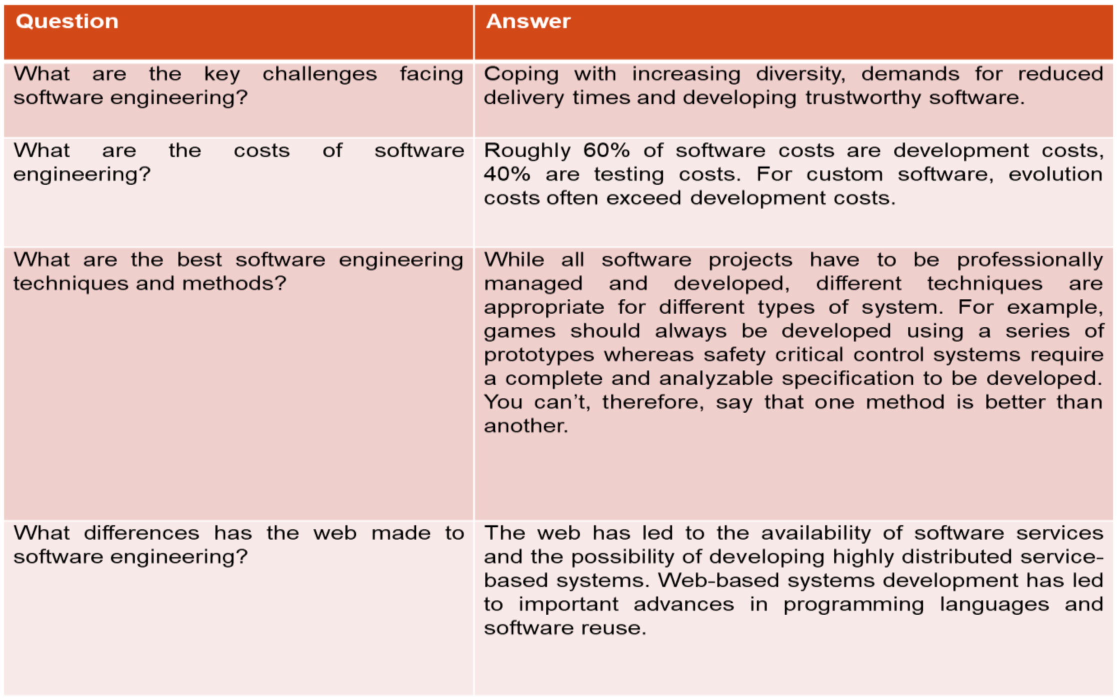
2. PROFESSIONAL SOFTWARE DEVELOPMENT

- Lots of people write programs for their own purpose
- However, the vast majority of software development is a professional activity where software is developed for specific business purposes, for inclusion in other devices, or as software products such as information systems, CAD systems, etc.
- Professional software, intended for use by someone apart from its developer, is usually developed by teams rather than individuals.
- It is maintained and changed throughout its life
- Software engineering is intended to support professional software development, rather than individual programming.
- It includes techniques that support program specification, design, and evolution.

Frequently Asked Questions about Software Engineering

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.

Frequently Asked Questions about Software Engineering cont….



Question	Answer
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.

- 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.
- Software engineers are concerned with developing software products (i.e., software which can be sold to a customer). **There are two kinds of software products:**

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.

2. Customized (or bespoke) products

- These are systems that are commissioned by a particular customer. A software contractor develops the software especially for that customer.
- An important difference between these types of software is that, in generic products, the organization that develops the software controls the software specification.
- For custom products, the specification is usually developed and controlled by the organization that is buying the software.
- The software developers must work to that specification.

Essential attributes of good software

Product characteristic	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 utilisation, 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.

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

1. **Engineering Discipline**
    - Engineers make things work. They apply theories, methods, and tools where these are appropriate
    - Try to discover solutions to problems even when there are no applicable theories and methods
2. **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 Process

- The systematic approach that is used in software engineering is sometimes called a software process.
- A software process is a sequence of activities that leads to the production of a software product.

1. **Software specification:** customers and engineers define the software that is to be produced and the constraints on its operation
2. **Software development:** the software is designed and programmed.

3. **Software validation:** the software is checked to ensure that it is what the customer requires
4. **Software evolution:** the software is modified to reflect changing customer and market requirements.

Challenges

There are three general issues that affect many different types of software:

1. **Heterogeneity:** Systems are required to operate as distributed systems across networks that include different types of computer and mobile devices
2. **Business and social change:** Business and society are changing incredibly quickly as emerging economies develop and new technologies become available
3. **Security and trust:** 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

- **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.
- **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
- **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.
- **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.
- **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.
- **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.
- **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.

- **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.

3. SOFTWARE ENGINEERING ETHICS

- Software engineering is carried out within a social and legal framework that limits the freedom of people working in that area.
- As a software engineer, you must accept that your job involves wider responsibilities than simply the application of technical skills
- You must also behave in an ethical and morally responsible way if you are to be respected as a professional engineer.
- You should uphold normal standards of honesty and integrity
- You should not use your skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession.

1. **Confidentiality:** Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed.
2. **Competence:** Engineers should not misrepresent their level of competence. They should not knowingly accept work which is out with their competence.
3. **Intellectual property rights:** Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected.
4. **Computer misuse:** Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

4. *CASE STUDIES*

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.

- Example: 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
- Example: a medical records system

3. A sensor-based data collection system

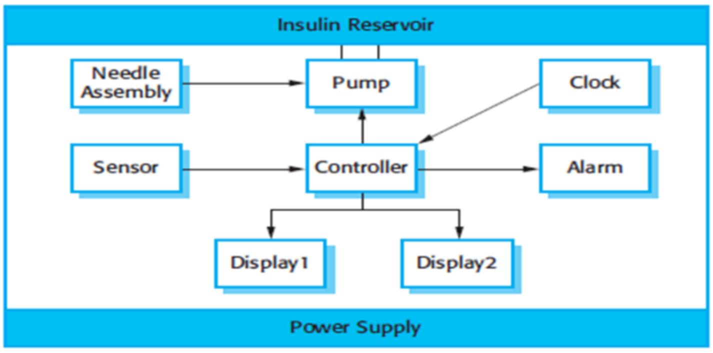
- This is a system whose primary purpose is to collect data from a set of sensors and process that data in some way.
- The key requirements of such systems are reliability, even in hostile environmental conditions, and maintainability.
- Example: a wilderness weather station.

1. An insulin pump control system

- An insulin pump is a medical system that simulates the operation of the pancreas (an internal organ).
- The software controlling this system is an embedded system, which collects information from a sensor and controls a pump that delivers a controlled dose of insulin to a user
- The conventional treatment of diabetes involves regular injections of genetically engineered insulin.
- Diabetics measure their blood sugar levels using an external meter and then calculate the dose of insulin that they should inject.
- The problem with this treatment is that the level of insulin required does not just depend on the blood glucose level but also on the time of the last insulin injection
- Current advances in developing miniaturized sensors have meant that it is now possible to develop automated insulin delivery systems
- These systems monitor blood sugar levels and deliver an appropriate dose of insulin when required
- A software-controlled insulin delivery system might work by using a microsensor embedded in the patient 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 sends signals to a miniaturized pump to deliver the insulin via a permanently attached needle.

Insulin Pump Hardware



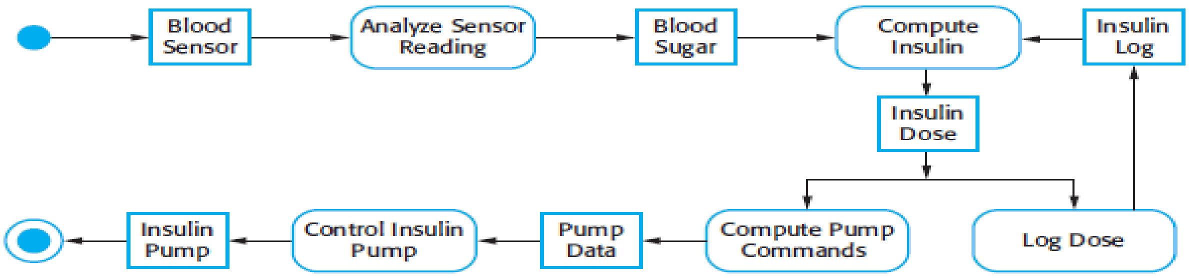


Fig: Activity Model

- Clearly, this is a safety-critical system.
- If the pump fails to operate or does not operate correctly, then the user's health may be damaged or they may fall into a coma because their blood sugar levels are too high or too low
- Two essential high-level requirements that this system must meet:
- The system shall be available to deliver insulin when required.
- The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.

2. 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

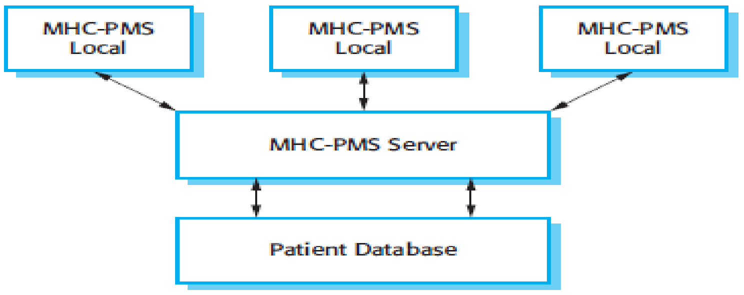


FIG: The Organization of the MHC-PMS

- Users of the system include clinical staff such as doctors, nurses, and health visitors (nurses who visit people at home to check on their treatment).
- Nonmedical users include receptionists who make appointments, medical records staff who maintain the records system, and administrative staff who generate reports
- 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:

- **Individual Care Management**
  - Create records for patients, edit the information in the system, view patient history, etc.
  - The system supports data summaries so that doctors who have not previously met a patient can quickly learn about the key problems and treatments that have been prescribed
- **Patient Monitoring**
  - The system regularly monitors the records of patients that are involved in treatment and issues warnings if possible, problems are detected.
  - If a patient has not seen a doctor for some time, a warning may be issued

- **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.
  - The overall design of the system has to take into account privacy and safety requirements

3. A Wilderness Weather Station

- To help monitor climate change and to improve the accuracy of weather forecasts in remote areas, the government of a country with large areas of wilderness decides to deploy several hundred weather stations in remote areas.
- These weather stations collect data from a set of instruments that measure temperature and pressure, sunshine, rainfall, wind speed, and wind direction
- Wilderness weather stations are part of a larger system, which is a weather information system that collects data from weather stations and makes it available to other systems for processing

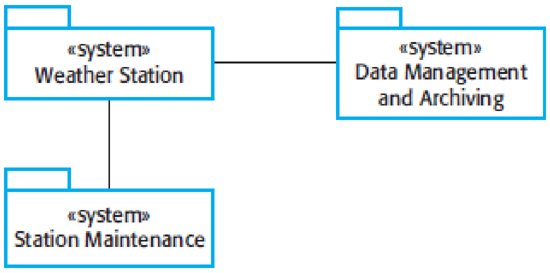

«system» Weather Station
«system» Data Management and Archiving
«system» Station Maintenance

Fig: The weather station's environment

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

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

****

SOFTWARE PROCESS

- **Waterfall Model**
- **Incremental Model**
- **Spiral Model**
- **Process Activity**

****

- 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
- New business software is now often developed by extending and modifying existing systems or by configuring and integrating off-the-shelf software or system components.

Four activities that is fundamental to software engineering:

- **Software specification**
  The functionality of the software and constraints on its operation must be defined.
- **Software design and implementation**
  The software to meet the specification must be produced.
- **Software validation**
  The software must be validated to ensure that it does what the customer wants.
- **Software evolution**
  The software must evolve to meet changing customer needs.

1. The Waterfall Model

- Plan-driven model
- This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing, and so on.

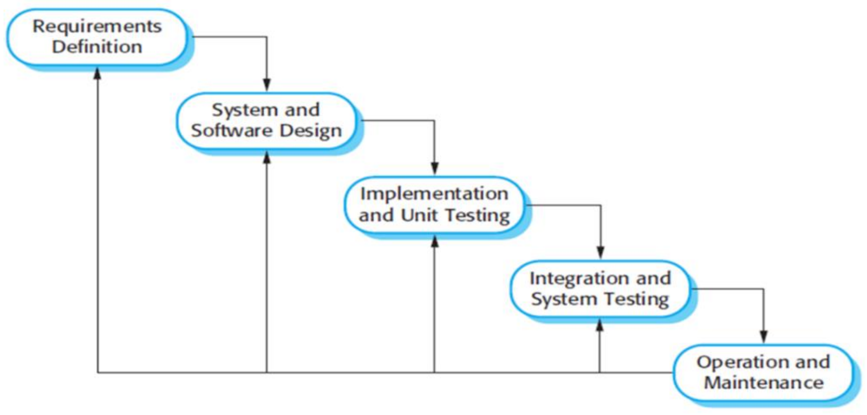

Requirements Definition
System and Software Design
Implementation and Unit Testing
Integration and System Testing
Operation and Maintenance

Fig: 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—in principle, you must plan and schedule all of the process activities before starting work on them.

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

- Normally (although not necessarily), 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 and enhancing the system's services as new requirements are discovered.

- The result of each phase is one or more documents that are approved ('signed off').
- The following phase should not start until the previous phase has finished.
- In practice, these stages overlap and feed information to each other.
- During design, problems with requirements are identified.
- During coding, design problems are found and so on.
- The software process is not a simple linear model but involves feedback from one phase to another.
- Documents produced in each phase may then have to be modified to reflect the changes made.

DRAWBACKS

- The main drawback of the waterfall model is the difficulty of accommodating change after the process is underway
- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
    - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.

2. Incremental Development

- This approach interleaves the activities of specification, development, and validation.
- The system is developed as a series of versions (increments), with each version adding functionality to the previous version.

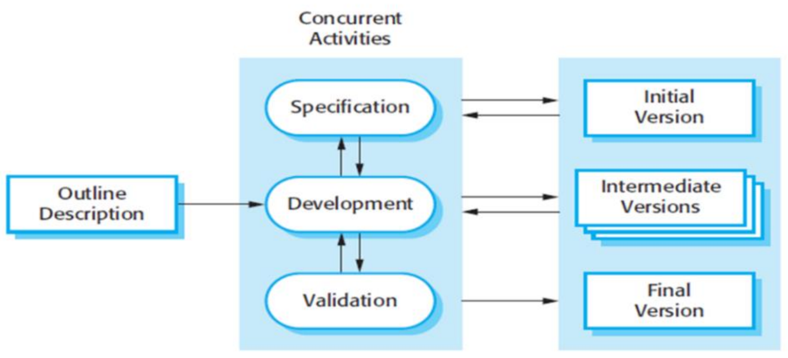


Fig: Incremental development

- 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

- Specification, development, and validation activities are interleaved rather than separate, with rapid feedback across activities
- By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed.
- Each increment or version of the system incorporates some of the functionality that is needed by the customer.
- Generally, the early increments of the system include the most important or most urgently required functionality.
- This means that the customer can evaluate the system at a relatively early stage in the development to see if it delivers what is required.
- If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments

BENEFITS

- 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.
  - Customers can comment on demonstrations of the software and see how much has been implemented.
- More rapid delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included
  - Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

PROBLMS

- 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.

3. Boehm's Spiral Model

- The software process is represented as a spiral, rather than a sequence of activities with some backtracking from one activity to another.

- Each loop in the spiral represents a phase of the software process. the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on.
- The spiral model combines change avoidance with change tolerance. It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks.

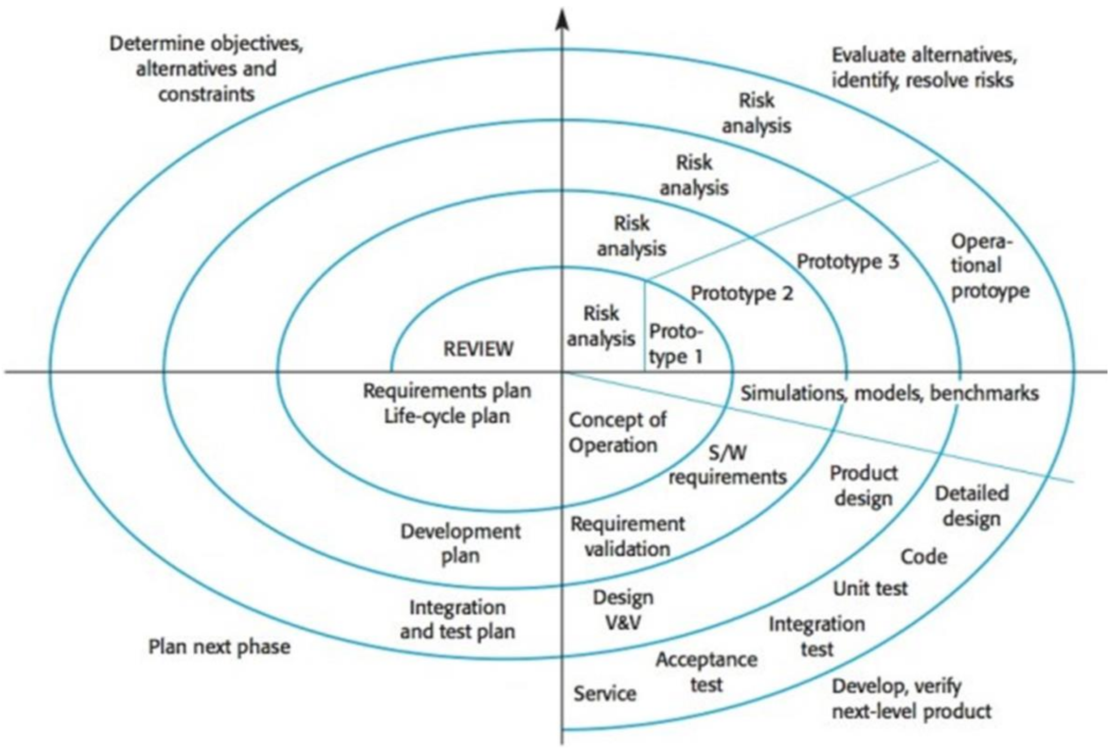


Fig: Boehm's spiral model of the software process

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. Alternative strategies, depending on these risks, may be planned.
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. If safety risks are the main consideration, development based on formal transformations may be the most appropriate process, and so on. If the main identified risk is sub-

system integration, the waterfall model may be the best development model to use.

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.

4. Process Activities

- Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system
- The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes.
- In the waterfall model, they are organized in sequence, whereas in incremental development they are interleaved.
- How these activities are carried out depends on the type of software, people, and organizational structures involved.

Software specification

- Software specification or requirements engineering 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
- Most critical stage as errors at this stage lead to problems at the later stages
- The requirements engineering process aims 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 developers need a more detailed system specification.

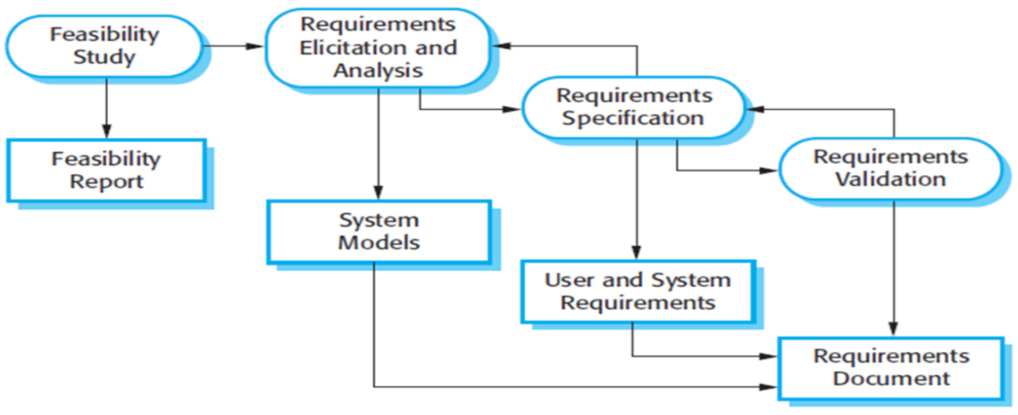


Fig: The requirements engineering process

Feasibility study

- An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies
- Proposed system will be cost effective?
- Can it be developed within budgetary constraints?
- Study should be cheap and quick
- The result should inform the decision of whether or not to go ahead with a more detailed analysis.

Requirements elicitation and analysis

- This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on.
- This may involve the development of one or more system models and prototypes.
- These help you understand the system to be specified.

Requirements specification

- Requirements specification 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 be provided.

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.

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.
- an abstract model of this process showing the inputs to the design process, process activities, and the documents produced as outputs from this process.

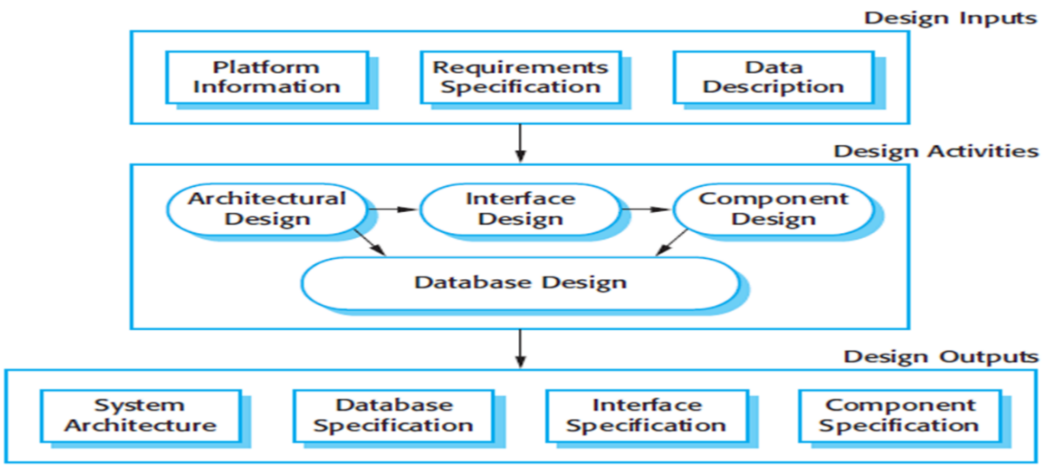


Fig: A general model of the design process

- Most software interfaces with other software systems. These include the operating system, database, middleware, and other application systems. These make up the 'software platform', the environment in which the software will execute
- The requirements specification is a description of the functionality the software must provide and its performance and dependability requirements
- If the system is to process existing data, then the description of that data may be included in the platform specification; otherwise, the data description must be an input to the design process so that the system data organization to be defined
- **Architectural Design**
    - Identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed
- **Interface Design**
    - 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

- **Component Design**
  - 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
  - Alternatively, it may be a list of changes to be made to a reusable component
  -
- **Database Design**
  - 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.

Software Validation

- Software validation or, more generally, verification and validation (V&V) is intended to show that a system 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 the software process from user requirements definition to program development.
- Except for small programs, systems should not be tested as a single, monolithic unit

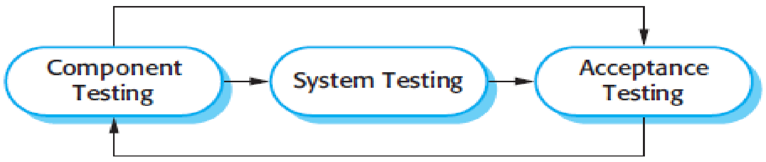


Fig: Stages of Testing

- Ideally, component defects are discovered early in the process, and interface problems are found when the system is integrated.

- As defects are discovered, the program must be debugged and this may require other stages in the testing process to be repeated.

  The stages in the testing process are:

- **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
- **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-functional requirements, and testing the emergent system properties
- **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.
  - Acceptance testing may also reveal requirements problems where the system's facilities do not really meet the user's needs or the system performance is unacceptable
- Normally, component development and testing processes are interleaved.
- Programmers make up their own test data and incrementally test the code as it is developed.
- This is an economically sensible approach, as the programmer knows the component and is therefore the best person to generate test cases
- 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' is often used.

- Beta testing involves delivering a system to a number of potential customers who agree to use that system.
- They report problems to the system developers.
- This exposes the product to real use and detects errors that may not have been anticipated by the system builders.
- After this feedback, the system is modified and released either for further beta testing or for general sale.
- If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment.
- In extreme programming, tests are developed along with the requirements before development starts
- When a plan-driven software process is used (e.g., for critical systems development), testing is driven by a set of test plans.
- An independent team of testers works from these pre-formulated test plans, which have been developed from the system specification and design.

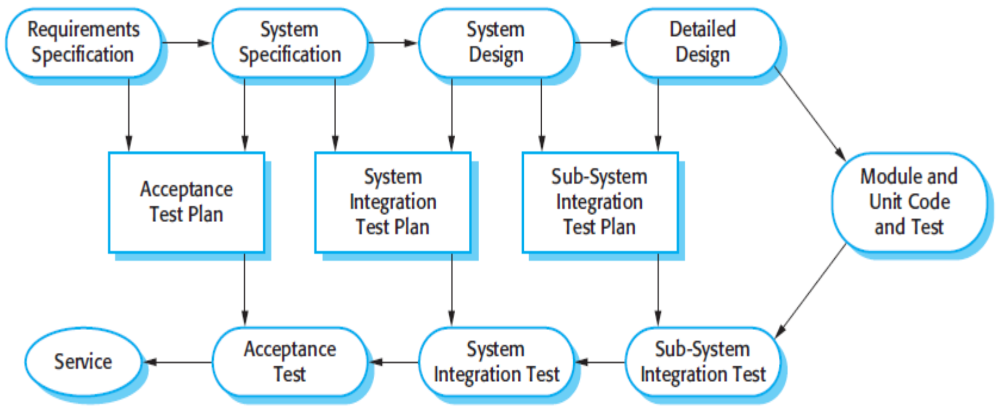

Requirements Specification
System Specification
System Design
Detailed Design
Acceptance Test Plan
System Integration Test Plan
Sub-System Integration Test Plan
Module and Unit Code and Test
Service
Acceptance Test
System Integration Test
Sub-System Integration Test

Fig: 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 is being incorporated in large, complex systems
- Once a decision has been made to manufacture hardware, it is very expensive to make changes to the hardware design.
- However, changes can be made to software at any time during or after the system development.
- People think of software development as a creative activity in which a software system is developed from an initial concept through to a working system.
- However, they sometimes think of software maintenance as dull and uninteresting.

- Although the costs of maintenance are often several times the initial development costs, maintenance processes are sometimes considered to be less challenging than original software development
- This distinction between development and maintenance is increasingly irrelevant.
- Hardly any software systems are completely new systems and it makes much more sense to see development and maintenance as a continuum.
- Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process where software is continually changed over its lifetime in response to changing requirements and customer needs

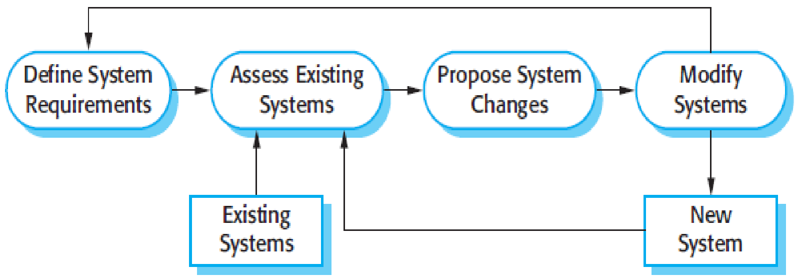


Fig: System Evolution

****

REQUIREMENTS ENGINEERING

- **Requirement Engineering Process**
- **Requirement Elicitation and Analysis**
- **Functional and Non-Functional Requirements**
- **Software Requirement Document**
- **Requirement Specification**
- **Requirement Validation**
- **Requirement Management**

****

Requirement Engineering Process

- 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).

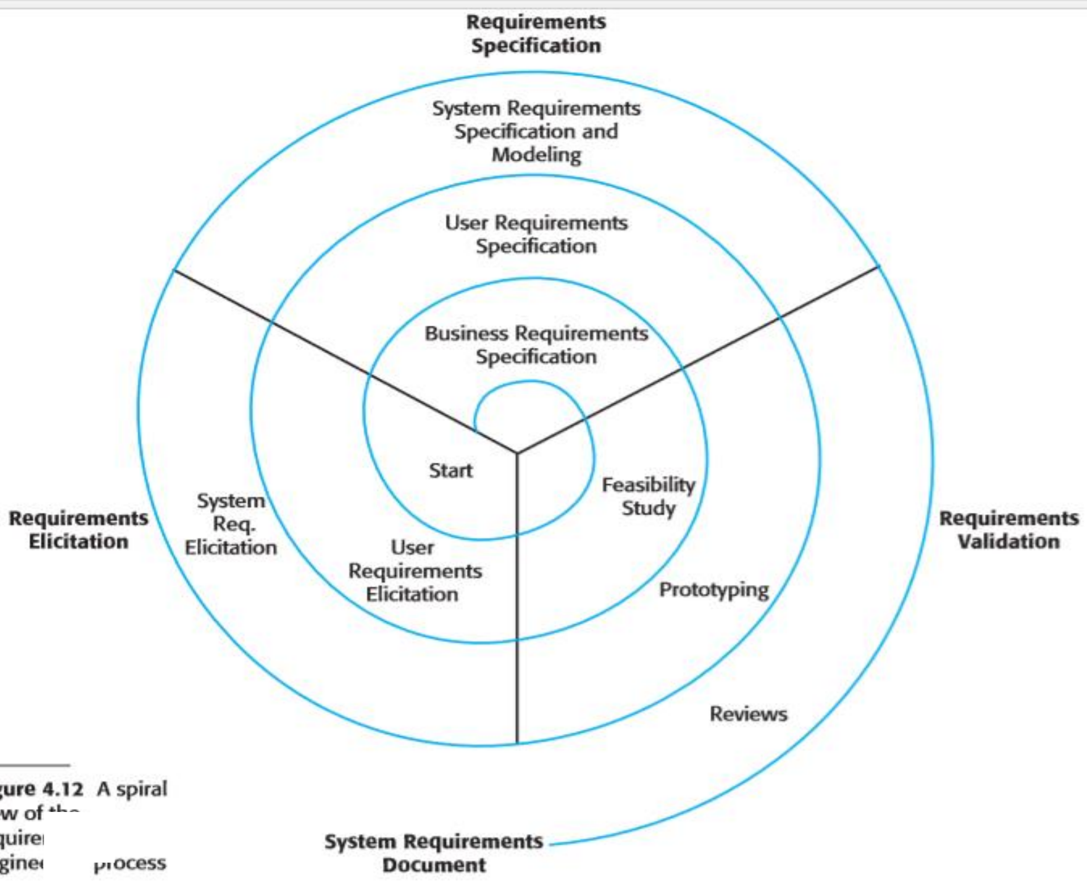

Requirements Specification
System Requirements Specification and Modeling
User Requirements Specification
Business Requirements Specification
Start
Feasibility Study
Prototyping
Reviews
Requirements Validation
Requirements Elicitation
System Req. Elicitation
User Requirements Elicitation
System Requirements Document

Figure 4.12 A spiral view of the require engine process

- Figure shows this interleaving. The activities are organized as an iterative process around a spiral, with the output being a system requirements document. The amount of time and effort devoted to each activity in each iteration depends on the stage of the overall process and the type of system being developed.
- Early in the process, most effort will be spent on understanding high-level business and non-functional requirements, and the user requirements for the system.
- Later in the process, in the outer rings of the spiral, more effort will be devoted to eliciting and understanding the detailed system requirements.
- 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 all of the user requirements have been elicited.
- Requirements elicitation, in particular, is a human-centered activity and people dislike the constraints imposed on it by rigid system models. In virtually all systems, requirements change.
- The people involved develop a better understanding of what they want the software to do; the organization buying the system changes; modifications are made to the system’s hardware, software, and organizational environment. The process of managing these changing requirements is called requirements management.

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.
- Requirements elicitation and analysis may involve a variety of different kinds of people in an organization

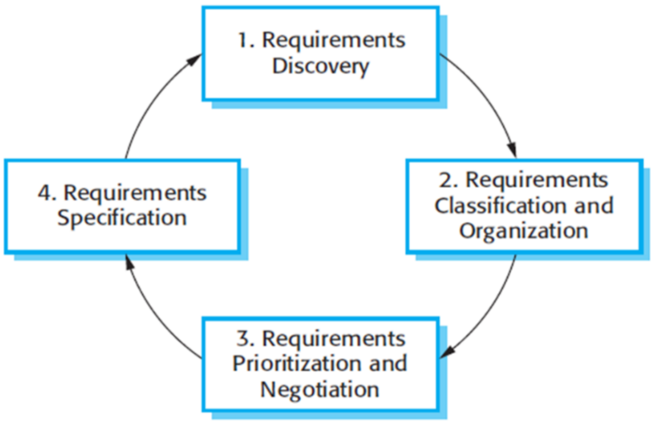

1. Requirements Discovery
2. Requirements Classification and Organization
3. Requirements Prioritization and Negotiation
4. Requirements Specification

Fig: The requirements elicitation and analysis process

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

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.

Requirements prioritization and negotiation

- Inevitably, when multiple stakeholders are involved, requirements will conflict.
- 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

Requirements specification

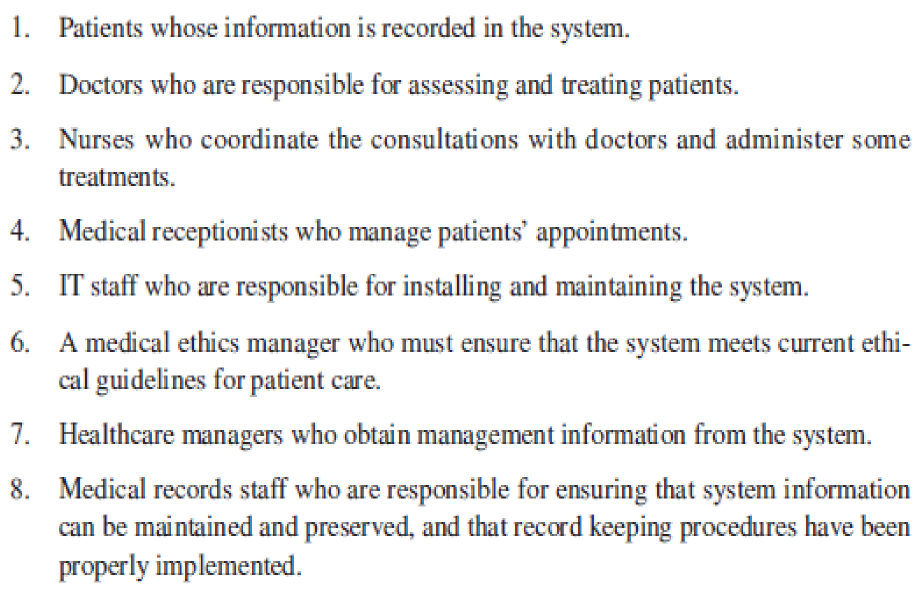
- The requirements are documented and input into the next round of the spiral
- Requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities.
- The process cycle starts with requirements discovery and ends with the requirements documentation.
- The analyst's understanding of the requirements improves with each round of the cycle.
- The cycle ends when the requirements document is complete

- 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 in the 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.

Requirement 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
- Sources of information during the requirements discovery phase include documentation, system stakeholders, and specifications of similar systems.
- You interact with stakeholders through interviews and observation and you may use scenarios and prototypes to help stakeholders understand what the system will be like.

- Stakeholders range from end-users of a system through managers to external stakeholders such as regulators, who certify the acceptability of the system. For example, system stakeholders for the mental healthcare patient information system include:



1. Patients whose information is recorded in the system.
2. Doctors who are responsible for assessing and treating patients.
3. Nurses who coordinate the consultations with doctors and administer some treatments.
4. Medical receptionists who manage patients' appointments.
5. IT staff who are responsible for installing and maintaining the system.
6. A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
7. Healthcare managers who obtain management information from the system.
8. Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

Interviewing:

- Formal or informal interviews with system stakeholders are part of most requirements engineering processes.
- In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed.
- Requirements are derived from the answers to these questions.

Interviews may be of two types:

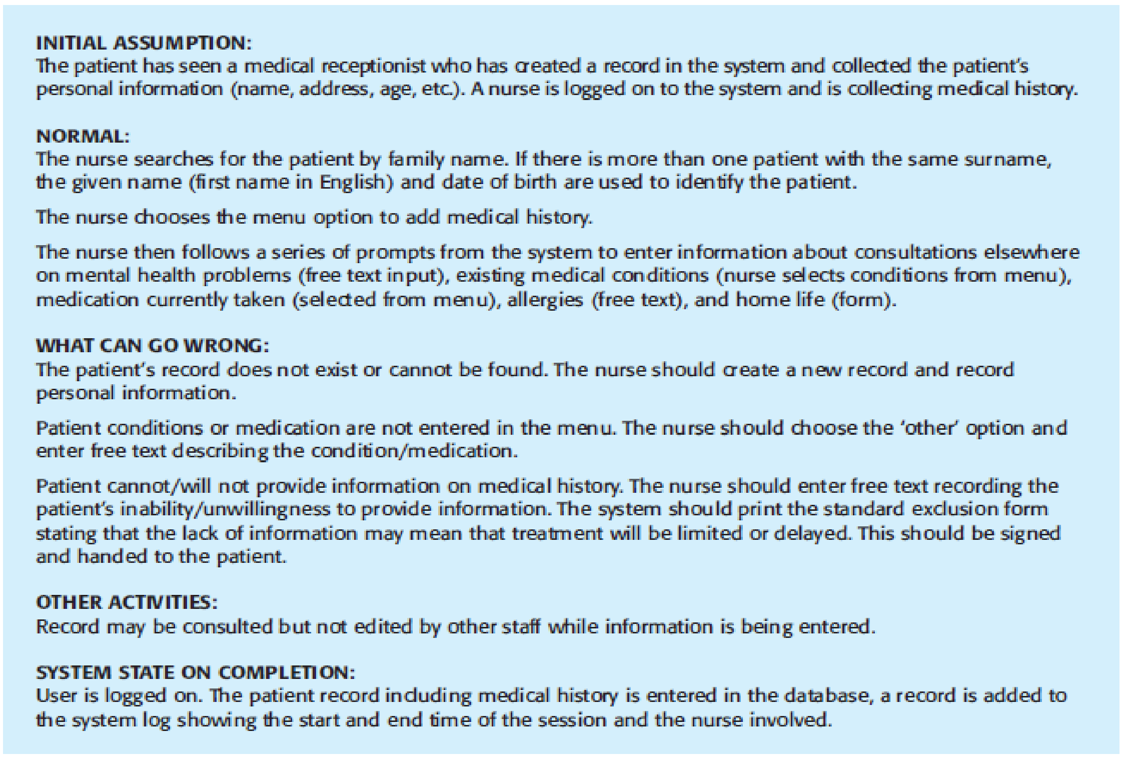
- Closed interviews, where the stakeholder answers a pre-defined set of questions
- Open interviews, in which there is no pre-defined agenda.
- The requirements engineering team explores a range of issues with system stakeholders and hence develop a better understanding of their needs.
- In practice, interviews with stakeholders are normally a mixture of both of these
- You may have to obtain the answer to certain questions but these usually lead on to other issues that are discussed in a less structured way.
- Completely open-ended discussions rarely work well
- Interviews are good for getting an overall understanding of what stakeholders do, how they might interact with the new system, and the difficulties that they face with current systems
- However, interviews are not so helpful in understanding the requirements from the application domain

- It can be difficult to elicit domain knowledge through interviews for two reasons:
- 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. They normally use terminology in a precise and subtle way that is easy for requirements engineers to misunderstand.
- 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.
- Interviews are also not an effective technique for eliciting knowledge about organizational requirements and constraints because there are subtle power relationships between the different people in the organization
- Effective interviewers have two characteristics:
- 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
- They prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system. Saying to people 'tell me what you want' is unlikely to result in useful information.

SCENARIOS:

- People usually find it easier to relate to real-life examples rather than abstract descriptions.
- They can understand and criticize a scenario of how they might interact with a software system.
- Requirements engineers can use the information gained from this discussion to formulate the actual system requirements.
- 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 at different 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



INITIAL ASSUMPTION:
The patient has seen a medical receptionist who has created a record in the system and collected the patient's personal information (name, address, age, etc.). A nurse is logged on to the system and is collecting medical history.

NORMAL:
The nurse searches for the patient by family name. If there is more than one patient with the same surname, the given name (first name in English) and date of birth are used to identify the patient.

The nurse chooses the menu option to add medical history.

The nurse then follows a series of prompts from the system to enter information about consultations elsewhere on mental health problems (free text input), existing medical conditions (nurse selects conditions from menu), medication currently taken (selected from menu), allergies (free text), and home life (form).

WHAT CAN GO WRONG:
The patient's record does not exist or cannot be found. The nurse should create a new record and record personal information.

Patient conditions or medication are not entered in the menu. The nurse should choose the 'other' option and enter free text describing the condition/medication.

Patient cannot/will not provide information on medical history. The nurse should enter free text recording the patient's inability/unwillingness to provide information. The system should print the standard exclusion form stating that the lack of information may mean that treatment will be limited or delayed. This should be signed and handed to the patient.

OTHER ACTIVITIES:
Record may be consulted but not edited by other staff while information is being entered.

SYSTEM STATE ON COMPLETION:
User is logged on. The patient record including medical history is entered in the database, a record is added to the system log showing the start and end time of the session and the nurse involved.

Fig: Scenario for collecting medical history in MHC-PMS

USECASE:

- Use cases are a requirements discovery technique that were first introduced in the Objectory method
- They have now become a fundamental feature of the unified modeling language
- In their simplest form, 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.
- Optionally, arrowheads may be added to lines to show how the interaction is initiated

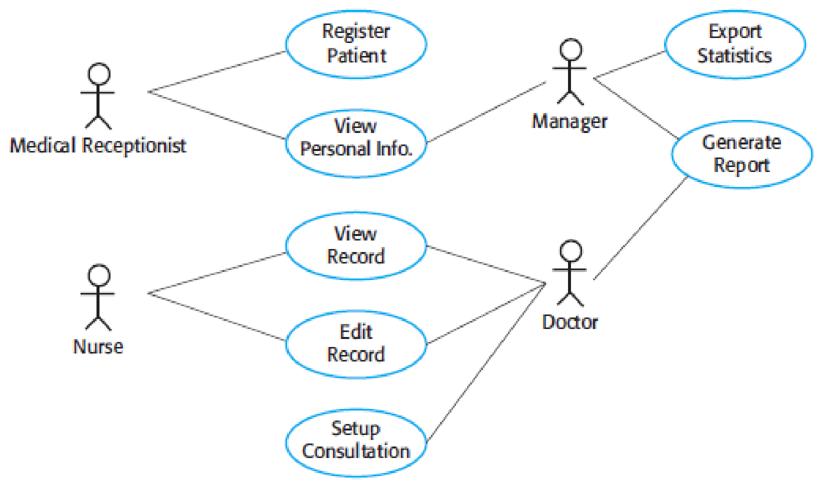


Fig: Use cases for the MHC-PMS

Ethnography:

- Software systems do not exist in isolation.
- They are used in a social and organizational context and software system requirements may be derived or constrained by that context.
- Satisfying these social and organizational requirements is often critical for the success of the system
- Ethnography is an observational technique that can be used to understand operational processes and help derive support requirements for these processes.
- An analyst immerses himself or herself in the working environment where the system will be used.
- The day-to-day work is observed and notes made of the actual tasks in which participants are involved.
- The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization
- Ethnography is particularly effective for discovering two types of requirements:
- Requirements that are derived from the way in which people actually work, rather than the way in which process definitions say they ought to work
- Requirements that are derived from cooperation and awareness of other people's activities
- Because of its focus on the end-user, this approach is not always appropriate for discovering organizational or domain requirements.
- They cannot always identify new features that should be added to a system.
- Ethnography is not, therefore, a complete approach to elicitation on its own.

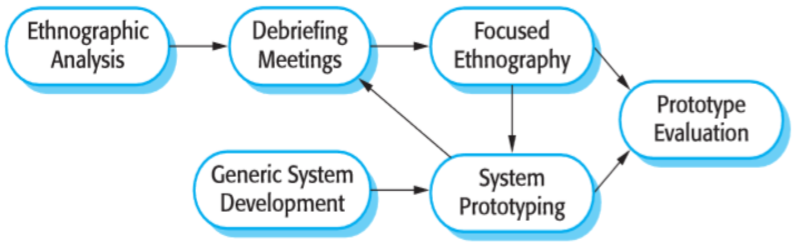


Fig: Ethnography and prototyping for requirements analysis

Functional and Non-Functional Requirements

- Software system requirements are often classified as functional requirements or non-functional requirements
- **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.
    - In some cases, the functional requirements may also explicitly state what the system should not do
- **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.
    - Non-functional requirements often apply to the system as a whole, rather than individual system features or services
- The distinction between different types of requirement is not as clear-cut
- A user requirement concerned with security, such as a statement limiting access to authorized users, may appear to be a non-functional requirement.
- However, when developed in more detail, this requirement may generate other requirements that are clearly functional, such as the need to include user authentication facilities in the system.

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
- More specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail
- Example: MHC-PMS

1. A user shall be able to search the appointments lists for all clinics.

2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.

3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

- It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation
- In the first requirement, the medical staff member specifying this may expect 'search' to mean that, given a patient name, the system looks for that name in all appointments at all clinics
- However, this is not explicit in the requirement. System developers may interpret the requirement in a different way and may implement a search so that the user has to choose a clinic then carry out the search. This obviously will involve more user input and so take longer.
- In principle, 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:

- Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users.
- They may relate to emergent system properties such as reliability, response time, and store occupancy.
- They may define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems
- 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
- System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable
- For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly
- Non-functional requirements arise through user needs, because of budget constraints, organizational policies, the need for interoperability with other software or hardware systems, or external factors such as safety regulations or privacy legislation

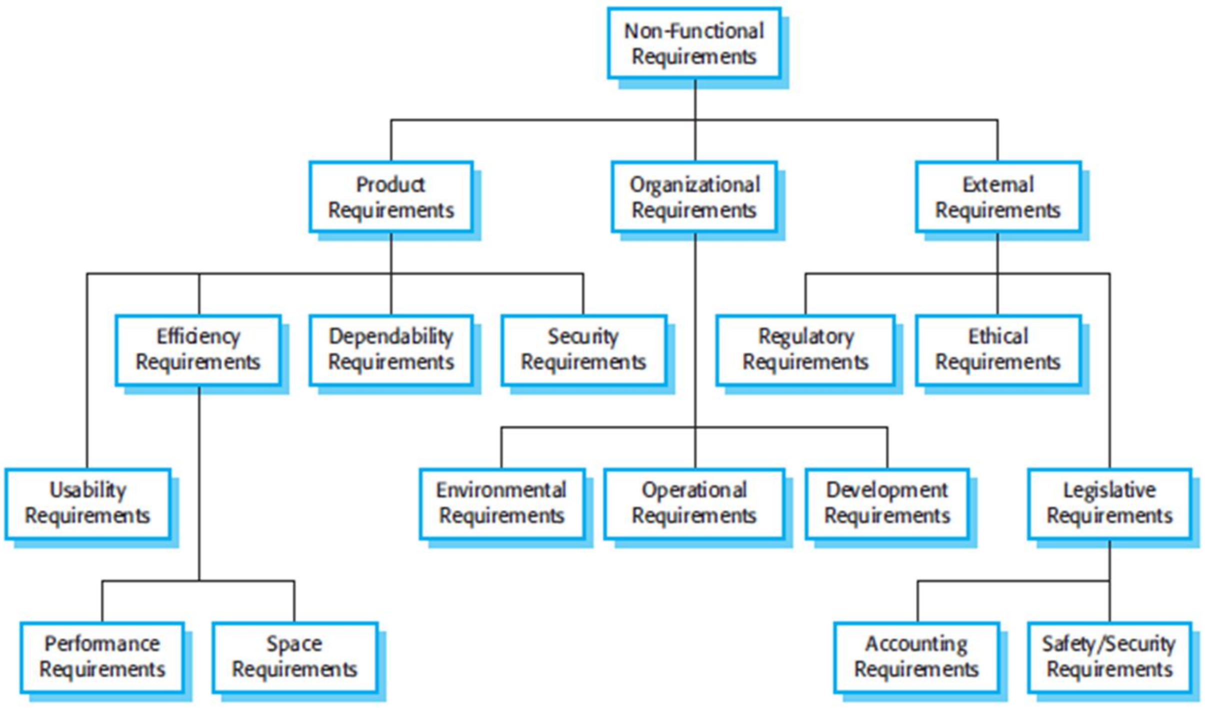
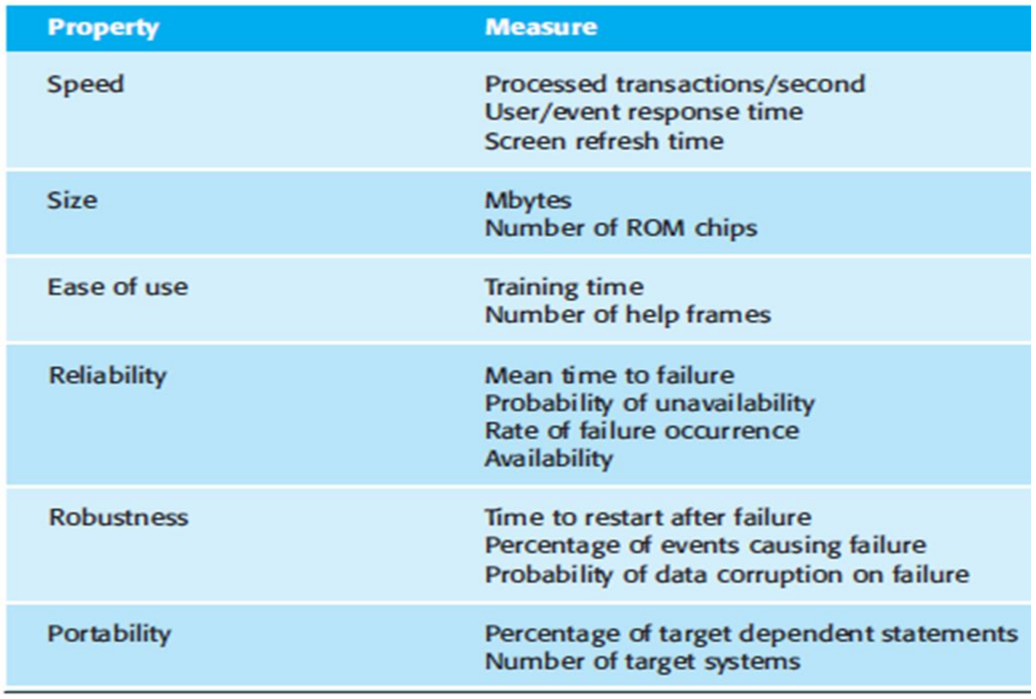


Fig: Types of non-functional requirement

- Whenever possible, you should write non-functional requirements quantitatively so that they can be objectively tested



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: Metrics for specifying non-functional requirements

- The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized
- Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.

The Software Requirements Document

- The software requirements document (sometimes called the software requirements specification or SRS) is an official statement of what the system developers should implement
- It should include both the user requirements for a system and a detailed specification of the system requirements
- Sometimes, the user and system requirements are integrated into a single description
- 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

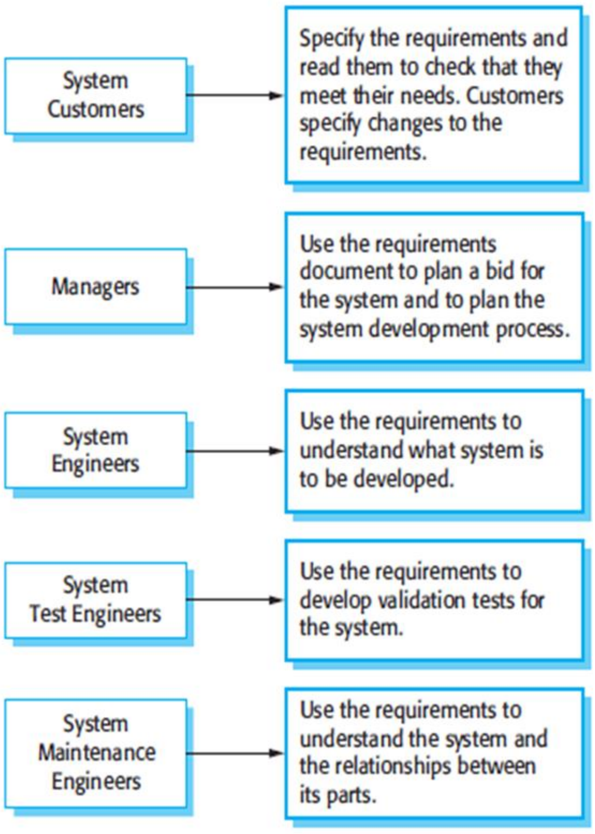
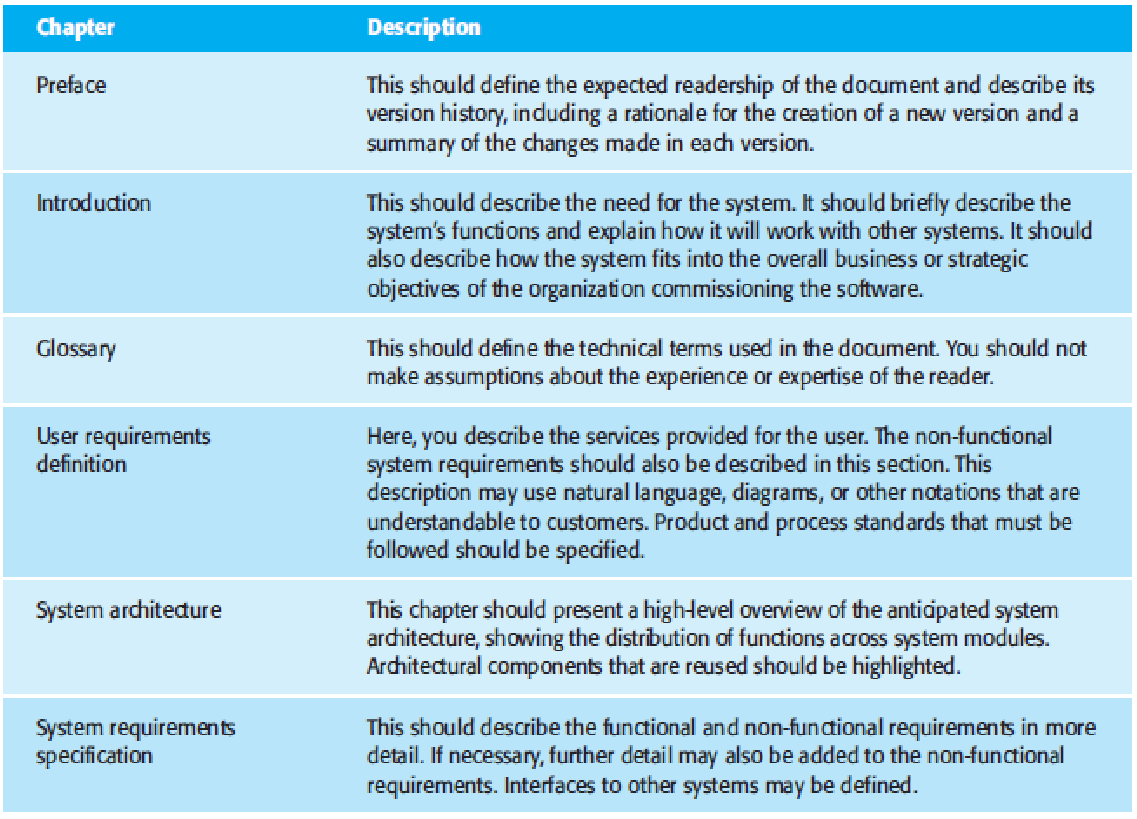
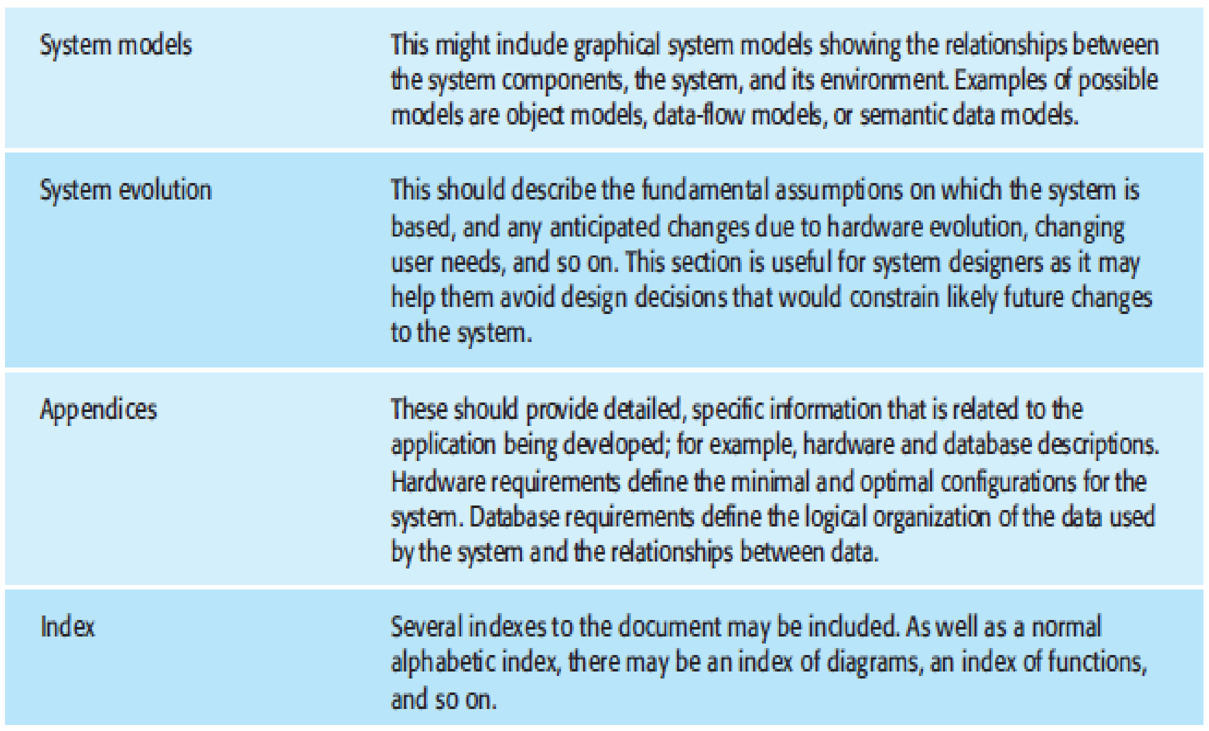

System Customers
Specify the requirements and read them to check that they meet their needs. Customers specify changes to the requirements.
Managers
Use the requirements document to plan a bid for the system and to plan the system development process.
System Engineers
Use the requirements to understand what system is to be developed.
System Test Engineers
Use the requirements to develop validation tests for the system.
System Maintenance Engineers
Use the requirements to understand the system and the relationships between its parts.

Fig: Users of a requirements document

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.



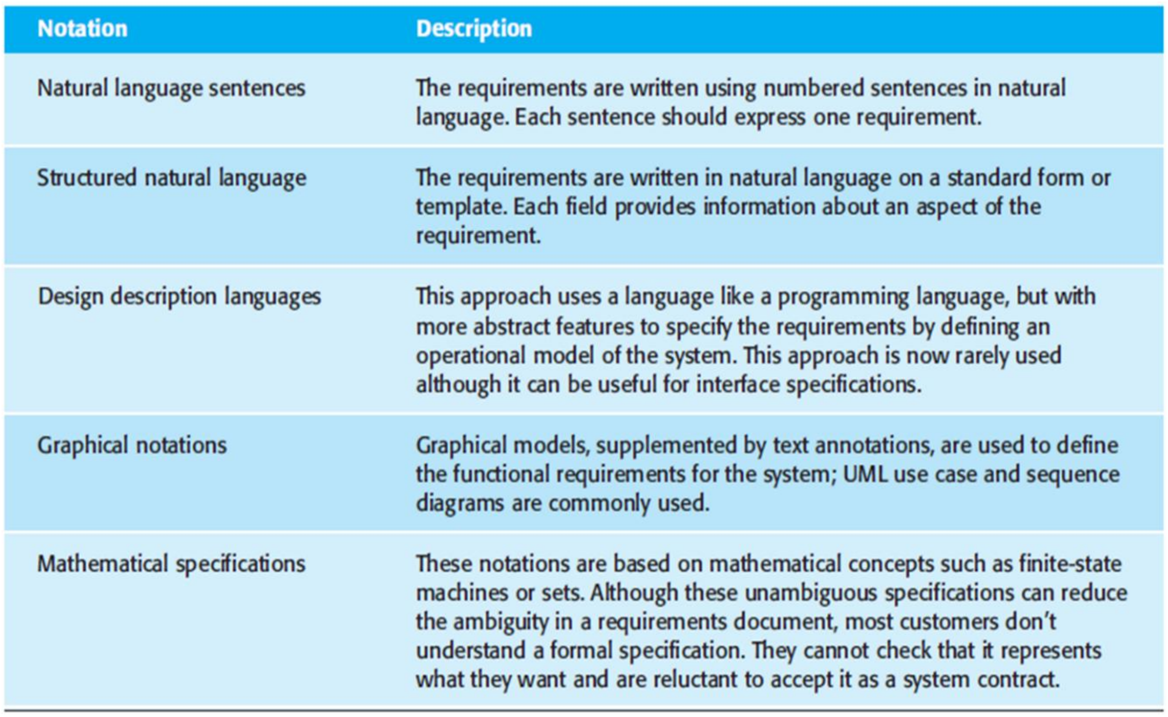
System models	This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions 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: The structure of a requirements document

Requirements Specification

- Requirements specification is the process of writing down the user and system requirements in a requirements document
- Ideally, the user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent
- The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge
- The requirements document should not include details of the system architecture or design
- If you are writing user requirements, you should not use software jargon, structured notations, or formal notations.
- You should write user requirements in natural language, with simple tables, forms, and intuitive diagrams
- 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
- Ideally, the system requirements should simply describe the external behavior of the system and its operational constraints.

- They should not be concerned with how the system should be designed or implemented.
- However, at the level of detail required to completely specify a complex software 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 up the system
- In most cases, systems must interoperate with existing systems, which constrain the design and impose requirements on the new system.
- The use of a specific architecture to satisfy non-functional requirements (such as N-version programming to achieve reliability)



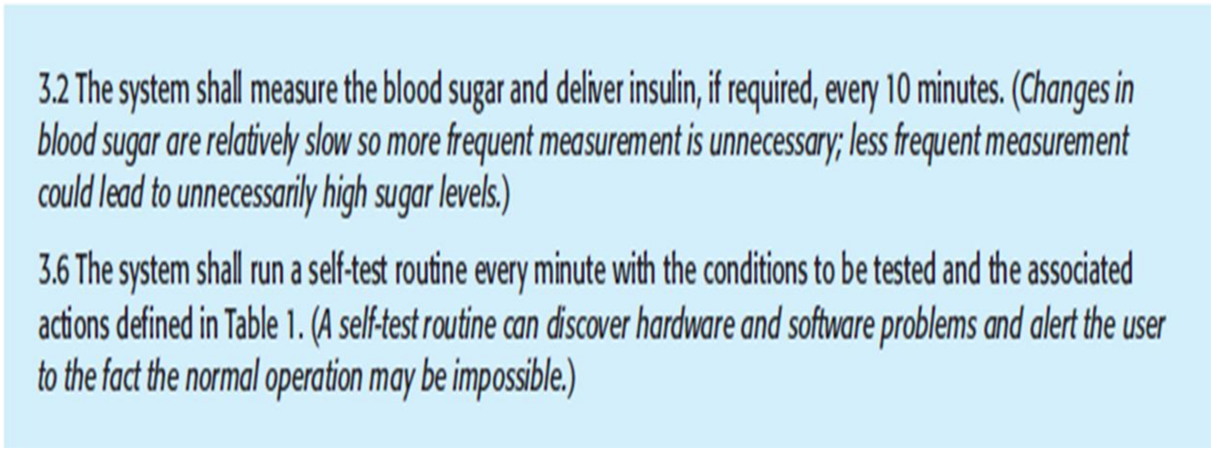
Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information 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: Ways of writing system requirements specification

Natural Language Specification:

- Natural language has been used to write requirements for software since the beginning of software engineering.
- It is expressive, intuitive, and universal.
- It is also potentially vague, ambiguous, and its meaning depends on the background of the reader
- To minimize misunderstandings when writing natural language requirements, follow some simple guidelines:

- Invent a standard format and ensure that all requirement definitions adhere to that format.
- Use language consistently to distinguish between mandatory and desirable requirements. Mandatory requirements are requirements that the system must support and are usually written using 'shall'. Desirable requirements are not essential and are written using 'should'
- Use text highlighting (bold, italic, or color) to pick out key parts of the requirement
- Do not assume that readers understand technical software engineering language
- Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included


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.*)

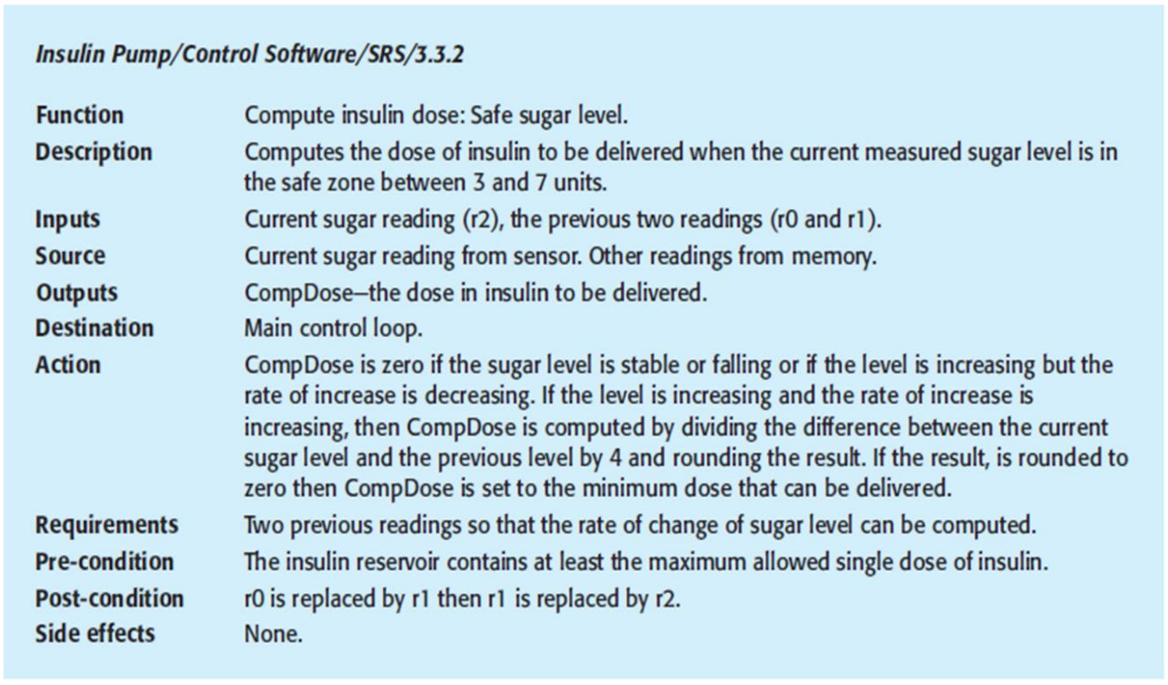
3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

Fig: Example requirements for the insulin pump software system

Structured Specification:

- Structured natural language is a way of writing system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way
- This approach maintains most of the expressiveness and understandability of natural language but ensures that some uniformity is imposed on the specification.
- Structured language notations use templates to specify system requirements.
- The specification may use programming language constructs to show alternatives and iteration, and may highlight key elements using shading or different fonts.
- When a standard form is used for specifying functional requirements, the following information 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 entities in the system that are used (the 'requires' part).

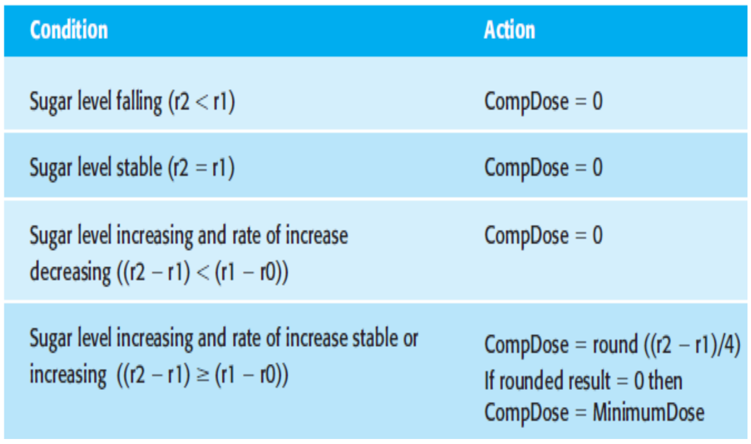
- A description of the action to be taken.
- If a functional approach is used, a pre-condition setting out what must be true before the function is called, and a post-condition specifying what is true after the function is called.
- A description of the side effects (if any) of the operation.
- Using structured specifications removes some of the problems of natural language specification.
- Variability in the specification is reduced and requirements are organized more effectively.
- However, it is still sometimes difficult to write requirements in a clear and unambiguous way, particularly when complex computations (e.g., how to calculate the insulin dose) are to be specified
- To address this problem, you can add extra information to natural language requirements, for example, by using tables or graphical models of the system.
- These can show how computations proceed, how the system state changes, how users interact with the system, and how sequences of actions are performed.



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: A structured specification of a requirement for an insulin pump



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

Fig: Tabular specification of computation for an insulin pump

Requirement Validation

- Requirements validation is the process of checking that requirements actually define the system that the customer really wants.
- Requirements validation is important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.
- During the requirements validation process, different types of checks should be carried out 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. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with different needs and any set of requirements is inevitably a compromise across the stakeholder community.

2**. Consistency checks** Requirements in the document should not conflict. That is, there should not be 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 checked to ensure that they can actually be implemented. These checks should also take account of the budget and schedule for the system development.

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 be able 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 used individually 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 meets their 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. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

Requirement Management

- The requirements for large software systems are always changing
- One reason for this is that these systems are usually developed to address 'wicked' problems—problems that cannot be completely defined
- Because the problem cannot be fully defined, the software requirements are bound to be incomplete
- During the software process, the stakeholders' understanding of the problem is constantly changing
- The system requirements must then also evolve to reflect this changed problem view.
- Once end-users have experience of a system, they will discover new needs and priorities.
- 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
    - Business priorities may change
    - New legislation and regulations may be introduced
- The people who pay for a system and the users of that system are rarely the same people.
    - System 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 user support if the system is to meet its goals
- Large systems usually have a diverse user community,

- Different users have different requirements and priorities that may be conflicting or contradictory.
- The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

- Requirements management is the process of understanding and controlling changes to system requirements.
- You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes.

Requirement Management Planning

- Planning is an essential first stage in the requirements management process
- The planning stage establishes the level of requirements management detail that is required
- During the requirements management stage, you have to decide on:

1.Requirements Identification:

Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and 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 requirements and the system design that should be recorded

4.Tool Support:

Requirements management involves the processing of large amounts of information about the requirements

Example: spreadsheets and simple database

- Requirements management needs automated support and the software tools for this should be chosen during the planning phase. You need tool support for:
  1. **Requirements Storage:** The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process
  2. **Change Management:** The process of change management is simplified if active tool support is available
  3. **Traceability Management:** Tool support for traceability allows related requirements to be discovered.

Requirement Change Management

- Requirements change management should be applied to all proposed changes to a system's requirements after the requirements document has been approved
- Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation.

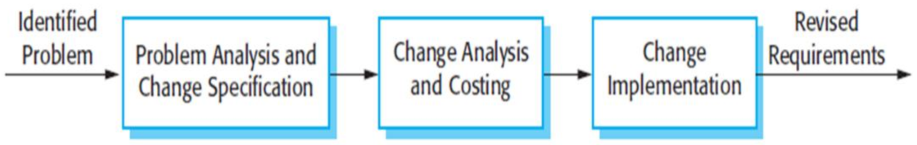

Identified Problem
Problem Analysis and Change Specification
Change Analysis and Costing
Change Implementation
Revised Requirements

Fig: Requirements change management

- There are three principal stages to a change management process:
- **Problem Analysis and Change Specification**
    - The process starts with an identified requirements problem or, sometimes, with a specific change 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.
- **Change Analysis and Costing**
    - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements.
    - The cost of making the change is estimated both in terms of modifications to the requirements document and, if appropriate, to the system design and implementation.
    - Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
- **Change Implementation**
    - The requirements document and, where necessary, the system design and implementation, are modified.
    - You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization.

Module 2

- **What Is Object Orientation?**
- **What Is OO Development?**
- **OO Themes**
- **Evidence for Usefulness of OO Development**
- **OO Modelling History**
- **Modelling**
- **Abstraction**
- **The Three Models**
- **Objects and Class Concepts**
- **Link and Association Concepts**
- **Generalization and Inheritance**
- **A Sample Class Model**
- **Navigation of Class Models**

****

What Is Object Oriented?

Software is organized as a collection of discrete objects that incorporate both State and behaviour.

Four aspects (characteristics) required by an OO approach are:

- **Identity**
- **Classification**
- **Polymorphism**
- **Inheritance**

Identity means that data is organized into discrete, distinguishable entities called objects.

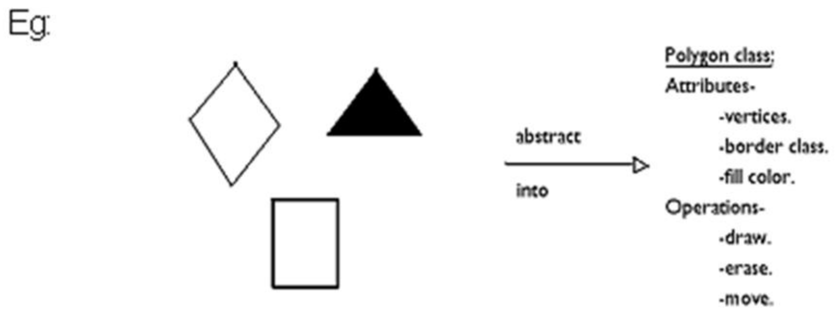
E.g. for objects: personal computer, bicycle

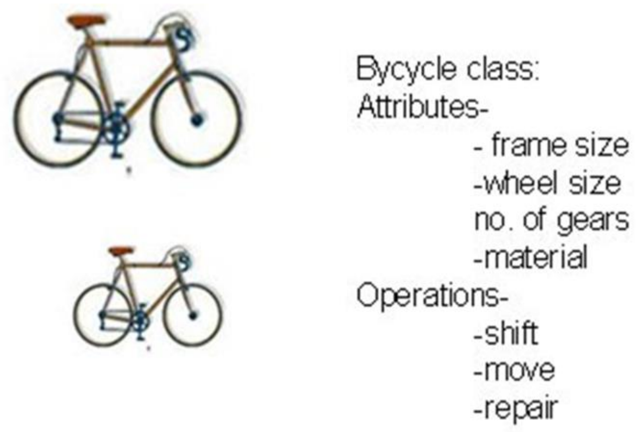
- **Objects** can be concrete (such as a file in a file system) or conceptual (such as scheduling policy in a multiprocessing OS).
- Each object has its own inherent identity. (i.e. two objects are distinct even if all their attribute values are identical).

Classification: It means that objects with same data structure (attribute) and behaviour (operations) are grouped into a class.

E.g. paragraph, monitor, chess piece.

- Each object is said to be an instance of its class.
- Fig below shows objects and classes: Each class describes a possibly infinite set of individual objects





Inheritance: It is the sharing of attributes and operations (features) among classes based on a hierarchical relationship. A super class has general information that sub classes refine and elaborate.
E.g. Scrolling window and fixed window are sub classes of window.

Polymorphism means that the same operation may behave differently for different classes.
For E.g. move operation behaves differently for a pawn than for the queen in a chess game.

What Is Object Oriented Development?

- Development refers to Software Life Cycle.
- OOD approach encourages software developers to work and think in terms of the application domain through most of the software engineering life cycle.

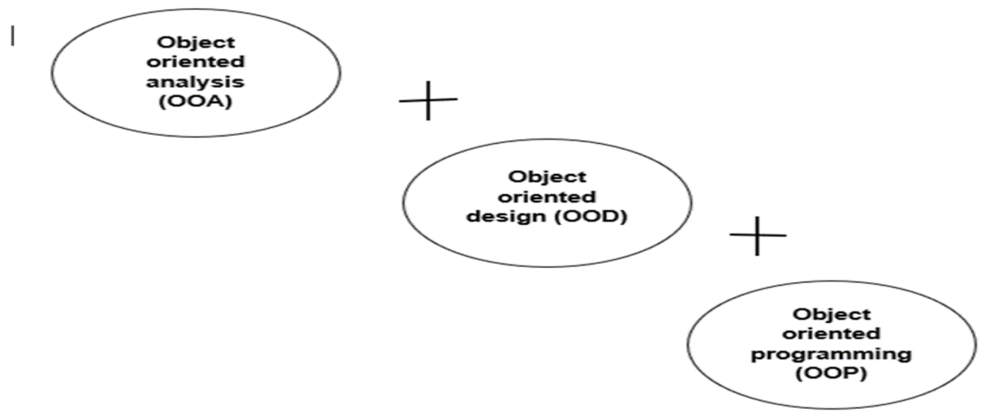

Object oriented analysis (OOA)
Object oriented design (OOD)
Object oriented programming (OOP)

Fig: Object Oriented Development

- **Development** refers to the software life cycle: Analysis, Design and Implementation.
- The essence of OO Development is the *identification* and *organization* of application concepts, rather than their final representation in a programming language. It's a conceptual process independent of programming languages.
- OO development is fundamentally a way of thinking and not a programming technique.

OO Methodology

The process for OO development and graphical notation for representing OO concepts consists of building a model of an application and then adding details to it during design. The methodology has the following stages:

- **System conception:** Software development begins with business analysis or users conceiving an application and formulating tentative requirements.

- **Analysis:** The analyst must work with the requestor to understand the problem, because problem statements are rarely complete or correct. The analysis model is a precise abstraction of what the desired system must do, not how it will be done. It should not contain implementation decisions.
  The analysis model has 2 parts:

  - ❖ **Domain model** - a description of the real-world objects reflected within the system E.g.: Domain objects for a stock broker
  - ❖ **Application model** - a description of the parts of the application system itself that are visible to the user.
    e.g.: - Application might include stock, bond, trade and commission.
- **System design**: The development teams devise a high – level strategy – the system architecture for solving the application problem. They also establish policies that will serve as a default for the subsequent, more detailed portions of design. The

system designer must decide what performance characteristics to optimize, choose a strategy of attacking the problem and make tentative resource allocations.

- **Class design**: The class designer adds details to the analysis model in accordance with the system design strategy. The focus of class design is the data structures and algorithms needed to implement each class.

- **Implementation**: Implementers translate the classes and relationships developed during class design into particular programming language, database or hardware.

OO Themes

. **Abstraction**:

- Abstraction lets you focus on essential aspects of an application while ignoring details i.e. focusing on what an object is and does, before deciding how to implement it.
- It's the most important skill required for OO development.

Encapsulation (information hiding):

- It separates the external aspects of an object (that are accessible to other objects) from the internal implementation details (that are hidden from other objects).
- Encapsulation prevents portions of a program from becoming so interdependent that a small change has massive ripple effects.

Combining data and behavior:

- Caller of an operation need not consider how many implementations exist.
- In OO system the data structure hierarchy matches the operation inheritance.
- hierarchy (fig).

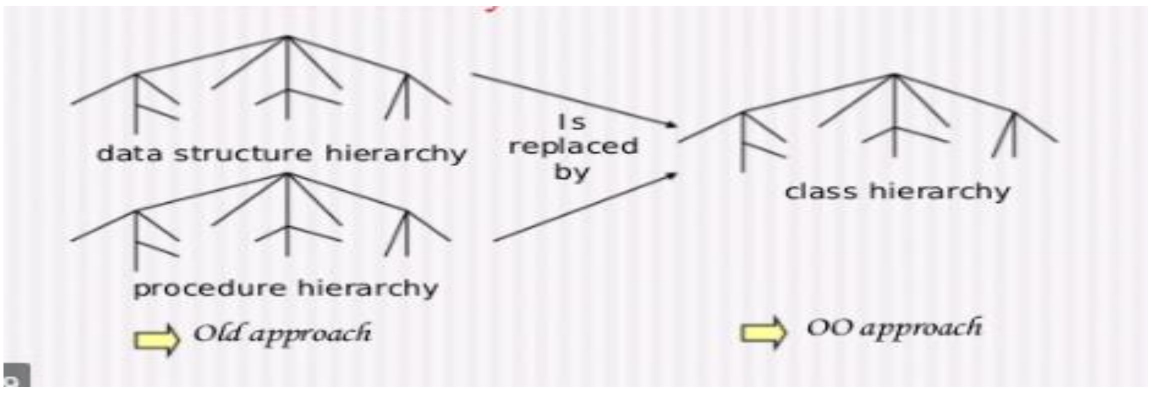


Fig: OO vs. Prior Approach

Sharing:

- OO techniques provide sharing at different levels.
- Inheritance of both data structure and behavior lets sub classes share common code.
- OO development not only lets you share information within an application, but also offers the prospect of reusing designs and code on future projects.

Emphasis on the essence of an object:

- OO development places a greater emphasis on data structure and a lesser emphasis on procedure structure than functional-decomposition methodologies.

Synergy:

- Identity, classification, polymorphism and inheritance characterize OO languages.

Three models

We use three kinds of models to describe a system from different viewpoints.

1. **Class Model—**for the objects in the system & their relationships.

- It describes the static structure of the objects in the system and their relationships.
- Class model contains class diagrams- a graph whose nodes are classes and arcs are relationships among the classes.

2. **State model—**for the life history of objects.
    - It describes the aspects of an object that change over time. It specifies and implements control with state diagrams-a graph whose nodes are states and whose arcs are transition between states caused by events.
3. **Interaction Model**—for the interaction among objects.
    - It describes how the objects in the system co-operate to achieve broader results. This model starts with use cases that are then elaborated with sequence and activity diagrams.
    - **Use case** – focuses on functionality of a system – i.e. what a system does for users.
    - **Sequence diagrams** – shows the object that interact and the time sequence of their interactions.
    - **Activity diagrams** – elaborates important processing steps.

Evidence for usefulness of OO development

- Applications at General Electric Research and Development Center.(1990)
- OO techniques for developing compilers, graphics, user interfaces, databases ,an OO language, etc.
- OO technology is a part of the computer science and software engineering mainstream.

- Important forums: (OOPSLA,ECOOP) Object Oriented Programming systems, Languages and applications. European Conference on OOP.

OO Modelling history

- Work at GE R&D led to OMT(Object-Modeling Technique) in 1991.
- Rumbaugh, Grady Booch on unifying the OMT and Booch Notaions in 1994.
- In 1996 the OMG(Object Management Group) issued a request for the proposals for a standard OO modeling notation.
- 1997 UML was accepted by OMG as a standard.
- In 2001 OMG released UML
- Added features and released UML in 2004.

****

MODELLING AS A DESIGN TECHNIQUE

MODELLING

Designers build many kinds of models for various purposes before constructing things.

Models serve several purposes –

- **Testing a physical entity before building it:** Medieval built scale models of Gothic Cathedrals to test the forces on the structures. Engineers test scale models of airplanes, cars and boats to improve their dynamics.
- **Communication with customers:** Architects and product designers build models to show their customers (note: mock-ups are demonstration products that imitate some of the external behavior of a system).
- **Visualization:** Storyboards of movies, TV shows and advertisements let writers see how their ideas flow.
- **Reduction of complexity:** Models reduce complexity to understand directly by separating out a small number of important things to do with at a time.

Abstraction:

- Is the selective examination of certain aspects of a problem.
- The goal of abstraction is to isolate those aspects that are important for some
- Purpose and suppress those aspects that are unimportant.

THE THREE MODELS

1. **Class Model**: represents the static, structural, “data” aspects of a system.

    - It describes the structure of objects in a system- their identity, their relationships to other objects, their attributes, and their operations.

- Goal in constructing class model is to capture those concepts from the real world that are important to an application.
- Class diagrams express the class model.

2. **State Model**: represents the temporal, behavioural, "control" aspects of a system.

- State model describes those aspects of objects concerned with time and the sequencing of operations – events that mark changes, states that define the context for events, and the organization of events and states.
- State diagram express the state model.
- Each state diagram shows the state and event sequences permitted in a system for one class of objects.
- State diagram refer to the other models.

- Actions and events in a state diagram become operations on objects in the class model. References between state diagrams become interactions in the interaction model.

3. **Interaction model** – represents the collaboration of individual objects, the "interaction" aspects of a system.

- Interaction model describes interactions between objects – how individual objects collaborate to achieve the behaviour of the system as a whole.
- The state and interaction models describe different aspects of behaviour, and you need both to describe behaviour fully.
- Use cases, sequence diagrams and activity diagrams document the interaction model.

****

CLASS MODELLING

OBJECT AND CLASS CONCEPT

• An object has three characteristics: state, behavior and a unique identification. Or

• An object is a concept, abstraction or thing with identity that has meaning for an application.

Objects

- Purpose of class modelling is to describe objects.
- An object is a concept, abstraction or thing with identity that has meaning for an application.
- Ex: Joe Smith, Infosys Company, process number 7648 and top window are objects.

Classes

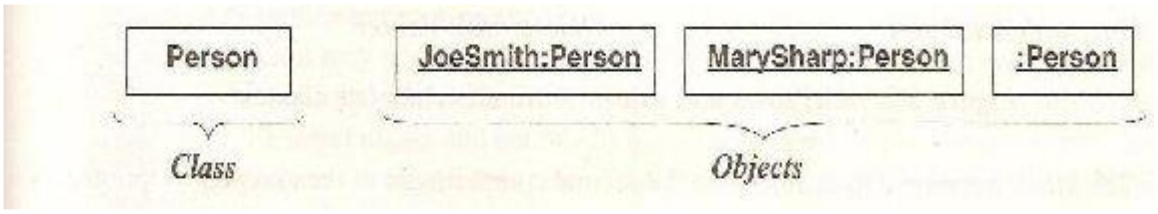
- An object is an instance or occurrence of a class.
- A class describes a group of objects with the same properties (attributes), behavior

(operations), kinds of relationships and semantics.

- Ex: Person, company, process and window are classes.

Class diagrams

- Class diagrams provide a graphic notation for modelling classes and their relationships, thereby describing possible objects.
- Note: An object diagram shows individual objects and their relationships. Useful for documenting test cases and discussing examples.
- Class diagrams are useful both for abstract modelling and for designing actual programs.



Conventions used (UML):

• UML symbol for both classes and objects is box.

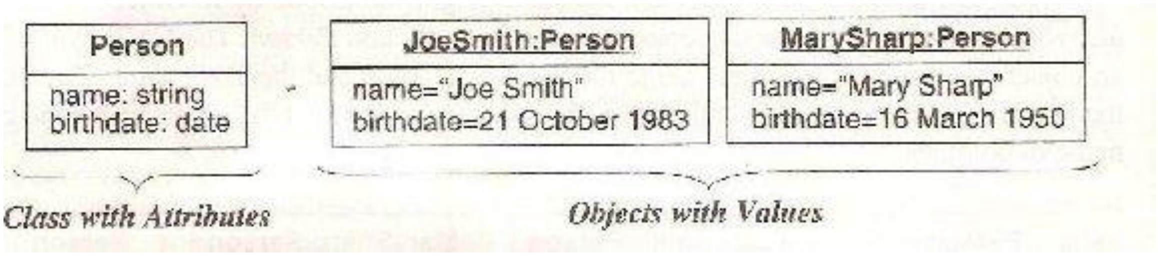
• Objects are modelled using box with object name followed by colon followed by class name, both the names are underlined.

• Use boldface to list class name, center the name in the box and capitalize the first letter. Use singular nouns for names of classes.

• To run together multiword names (such as JoeSmith), separate the words with intervening capital letter.

Values and Attributes:

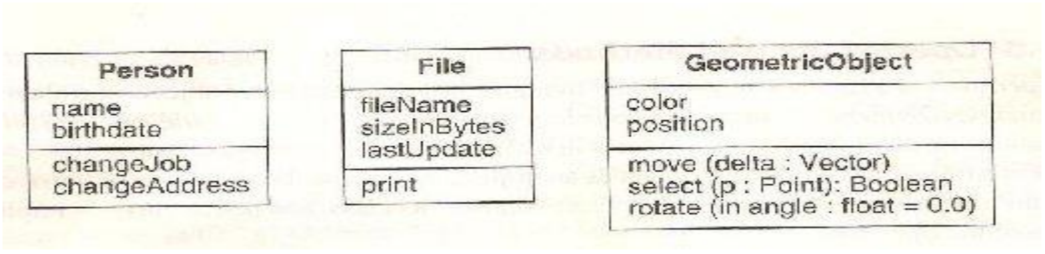
- **Value** is a piece of data.
- **Attribute** is a named property of a class that describes a value held by each object of the class.
- E.g. Attributes: Name, bdate, weight.
- Values: JoeSmith, 21 October 1983, 64.



- Conventions used (UML):
- List attributes in the 2nd compartment of the class box.
- A colon precedes the type; an equal sign precedes default value.
- Show attribute name in regular face, left align the name in the box and use small case for the first letter.

Operations and Methods:

- An **operation** is a function or procedure that maybe applied to or by objects in a class.
- E.g. Hire, fire and pay dividend are operations on Class Company. Open, close, hide and redisplay are operations on class window.
- A **method** is the implementation of an operation for a class.
- E.g. In class file, print is an operation you could implement different methods to print files.



- UML conventions used
- List operations in 3rd compartment of class box.
- List operation name in regular face, left align and use lower case for first letter.
- Optional details like argument list and return type may follow each operation name.
- Parenthesis enclose an argument list; commas separate the arguments. A colon precedes the result type.
- Note: We do not list operations for objects, because they do not vary among objects of same class.

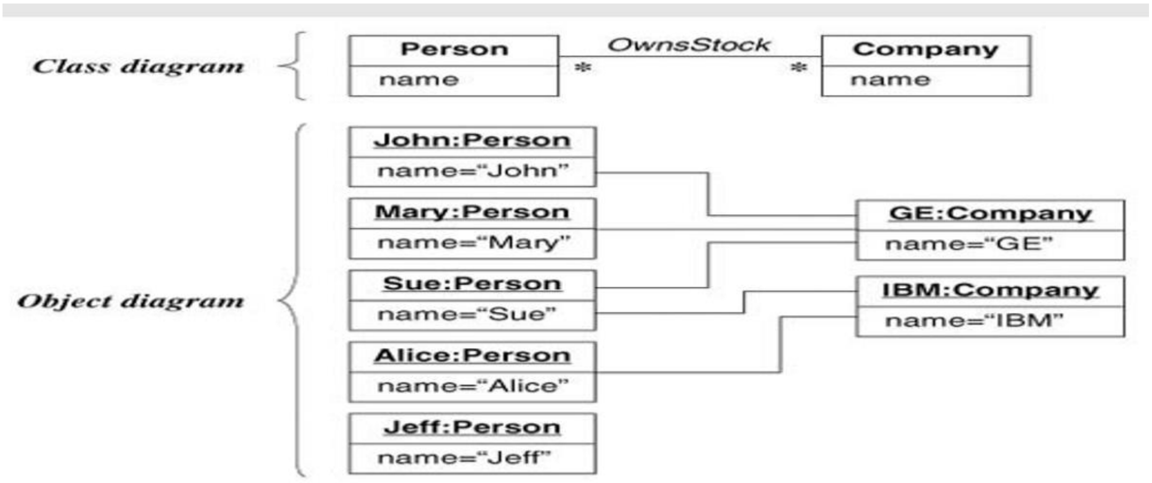
Links and Association

- A **link** is a physical or conceptual connection among objects.
    - E.g. JoeSmith *WorksFor* Simplex Company.
- Mathematically, we define a link as a tuple – that is, a list of objects.
- A link is an instance of an **association**.
- An **association** is a description of a group of links with common structure and common semantics.

  E.g. a person *WorksFor* a company.

  - An association describes a set of potential links in the same way that a class
  - Describes a set of potential objects.

- Fig shows many-to-many association (model for a financial application).



- Conventions used (UML):
- Link is a line between objects
- Association connects related classes and is also denoted by a line.
- Show link and association names in italics.

Multiplicity

Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class. Multiplicity constrains the number of related objects.

UML conventions:

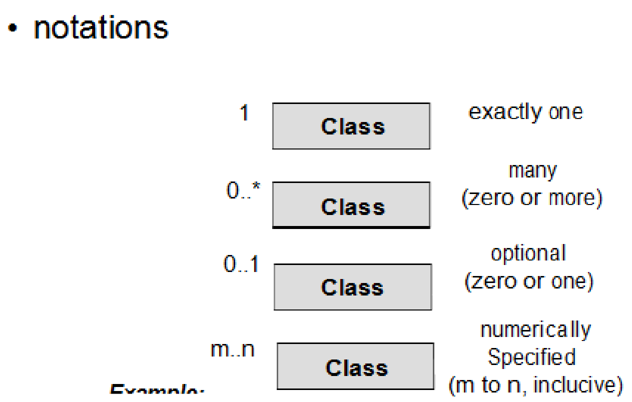
- UML diagrams explicitly lists multiplicity at the ends of association lines.
- UML specifies multiplicity with an interval, such as

“1” (exactly one)

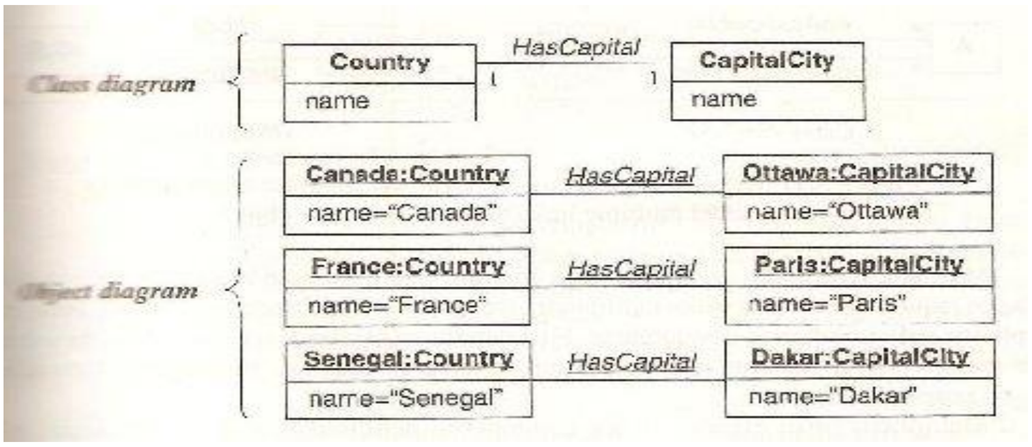
“1…” (one or more)

“3..5” (three to five, inclusive)

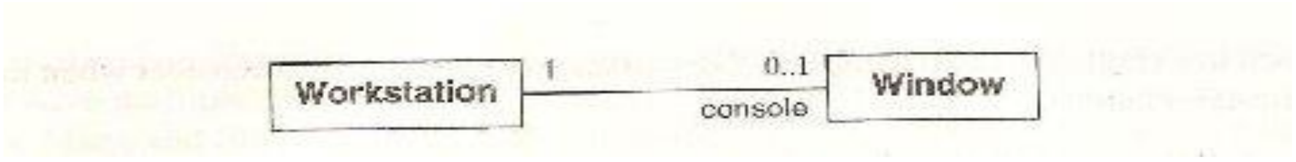
“*” (many, ie zero or more)



Below figure illustrates **one to one multiplicity**



Below figure illustrates **zero-or-one multiplicity.** A workstation may have one of its windows designated as the console to receive general error messages.



The following figure illustrates **Association vs. link**. If you want two links between the same objects, you must have two associations.

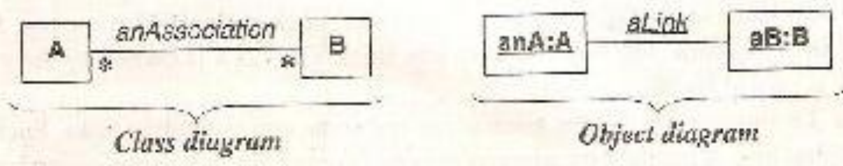


Figure 3.10 **Association vs. link.** A pair of objects can be instantiated at most once per association (except for bags and sequences).

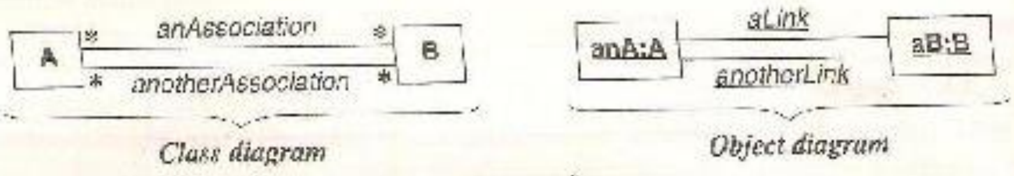
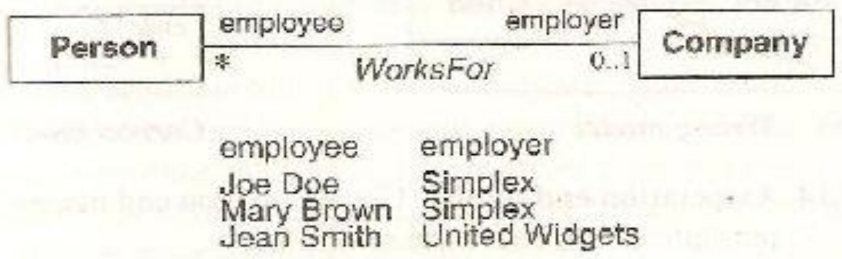


Figure 3.11 **Association vs. link.** You can use multiple associations to model multiple links between the same objects.

Association End Names

- Multiplicity implicitly refers to the ends of associations. For E.g. A one-to- many association has two ends –
- an end with a multiplicity of "one"
- an end with a multiplicity of "many"

You can not only assign a multiplicity to an association end, but you can give it a name as well.



employee	employer
Joe Doe	Simplex
Mary Brown	Simplex
Jean Smith	United Widgets

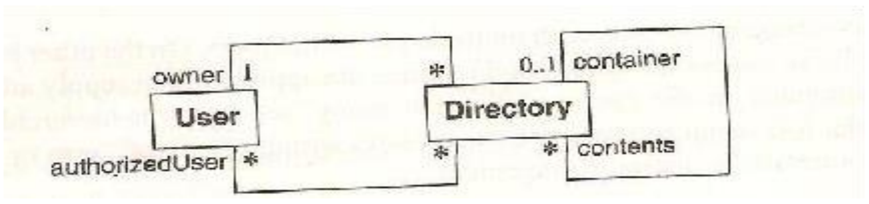
Association end names. Each end of an association can have a name.

A person is an employee with respect to company.

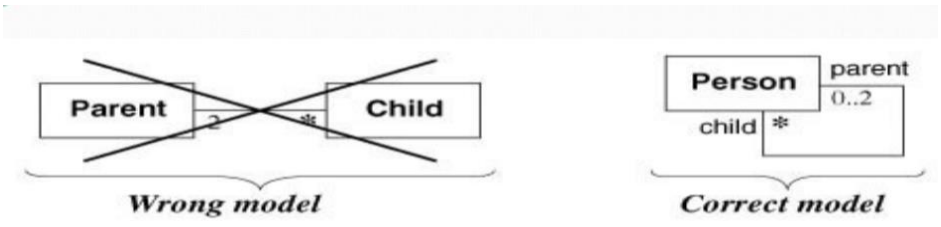
A company is an employer with respect to a person.

Note 1: Association end names are optional.

Note 2: Association end names are necessary for associations between two objects of the same class. They can also distinguish multiple associations between a pair of classes.
E.g. each directory has exactly one user who is an owner and many users who are authorized to use the directory. When there is only a single association between a pair of distinct classes, the names of the classes often suffice, and you may omit association end names.



Note 3: Association end names let you unify multiple references to the same class. When constructing class diagrams you should properly use association end names and not introduce a separate class for each reference as below fig shows.



Figure, in the wrong model, two instances represent a person with a child, one for the child and one for the parent. In the correct model, one person instance participants in two or more links, twice as a parent and zero or more times as a child.

Ordering

Ordering is an inherent part of association. You can indicate an ordered set of objects by writing "{ordered}" next to the appropriate association end.

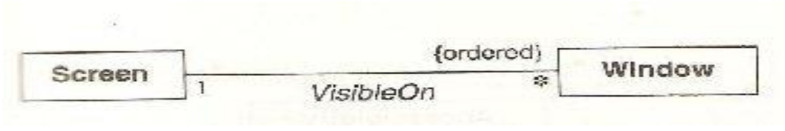


Fig: ordering sometimes occurs for "many" multiplicity

Bags and Sequence

- Normally, a binary association has **at most one link** for a pair of objects.
- However, you can permit **multiple links** for a pair of objects by annotating an association end with {bag} or {sequence}.
- A **bag** is a collection of elements with duplicates allowed.
- A **sequence** is an ordered collection of elements with duplicates allowed. Example:

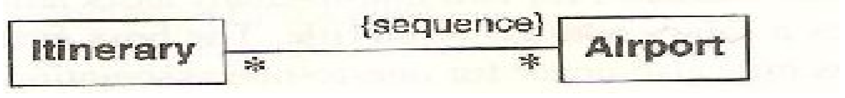


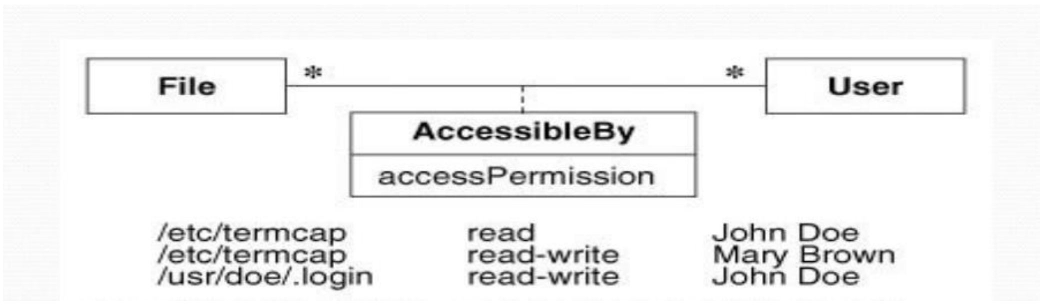
fig: an itinerary may visit multiple airports, so you should use {sequence} and not {ordered}

Note: {ordered} and {sequence} annotations are same, except that the first disallows duplicates and the other allows them.

Association Classes

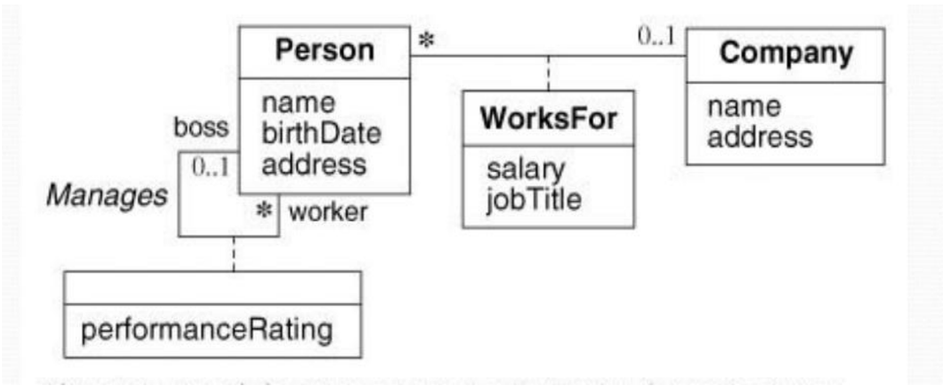
- An association class is an association that is also a class.
- Like the links of an association, the instances of an association class derive identity from instances of the constituent classes.
- Like a class, an association class can have attributes and operations and participate in associations.

Ex:

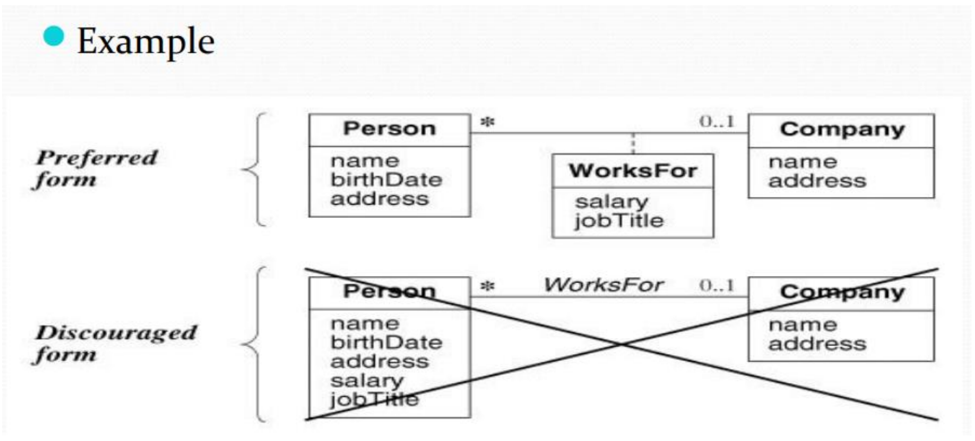


- **UML notation** for association class is a box attached to the association by a dashed line.
- **Note:** Attributes for association class unmistakably belong to the link and cannot be ascribed to either object. In the above figure, accessPermission is a joint property of File and user cannot be attached to either file or user alone without losing information.

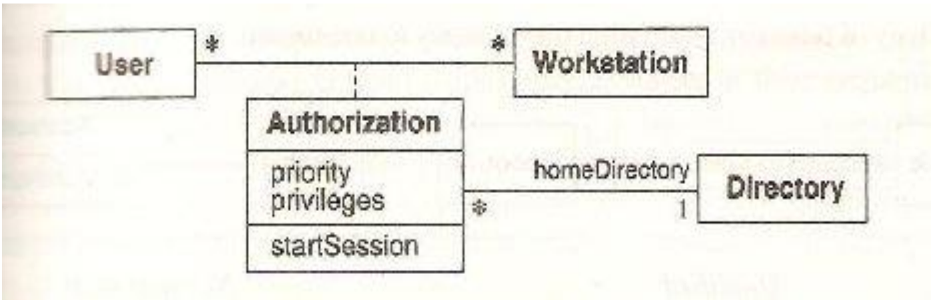
- Below figure presents attributes for two one-to-many relationships. Each person working for a company receives a salary and has job title. The boss evaluates the performance of each worker. Attributes may also occur for one-to-one associations.



Note 1: Figure shows how it's possible to fold attributes for one-to-one and one- to-many associations into the class opposite a "one" end. This is not possible for many-to-many associations.
As a rule, you should not fold such attributes into a class because the multiplicity of the association may change.



Note 2: An association class participating in an association.



Note 3: Association class vs. ordinary class

Qualified Associations

- A Qualified Association is an association in which an attribute called the qualifier disambiguates the objects for a "many" association ends. It is possible to define qualifiers for one-to-many and many-to-many associations.
- A qualifier selects among the target objects, reducing the effective multiplicity from "many" to "one".
- Ex 1: qualifier for associations with one to much multiplicity. A bank services multiple accounts. An account belongs to single bank. Within the context of a bank, the Account Number specifies a unique account. Bank and account are classes, and Account Number is a qualifier. Qualification reduces effective multiplicity of this association from one-to-many to one-to-one.

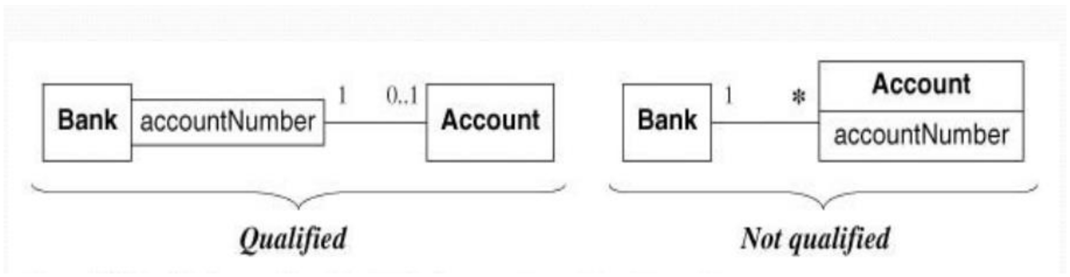
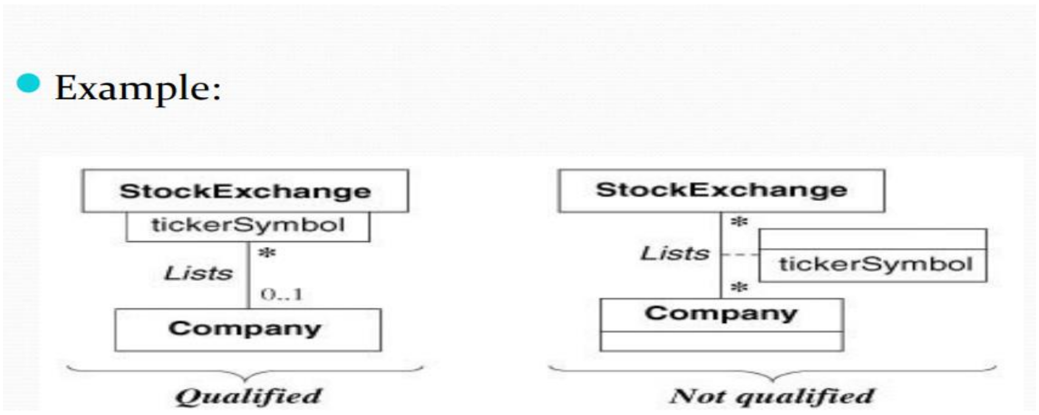


Fig: qualification increases the precision of a model. (note: however, both are acceptable)

Ex 2: a stock exchange lists many companies. However, it lists only one company with a given ticker symbol. A company may be listed on many stock exchanges, possibly under different symbols.

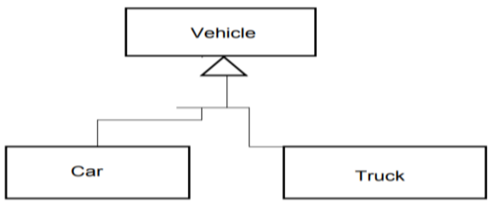


GENERALIZATION AND INHERITANCE

- **Generalization** is the relationship between a class (the superclass) and one or more variations of the class (the subclasses). Generalization organizes classes by their similarities and differences, structuring the description of objects.
- The superclass holds common attributes, operations and associations; the subclasses add specific attributes, operations and associations. Each subclass is said to **inherit** the features of its superclass.
- Generalization is called the "is-a" relationship.

  It is represented by a solid line with a large arrow head pointing towards the parent class.

  • Example:



- Fig (a) and Fig(b) (given in the following page) shows examples of generalization.

- **Fig (a) – Example of generalization for equipment.**
- Each object inherits features from one class at each level of generalization.
- **UML convention used:**
- Use large hollow arrowhead to denote generalization. The arrowhead points to superclass.
- **Fig (b) – inheritance for graphic figures.**
- The word written next to the generalization line in the diagram (i.e dimensionality) is a generalization set name. A generalization set name is an enumerated attribute that indicates which aspect of an object is being abstracted by a particular generalization. It is optional.

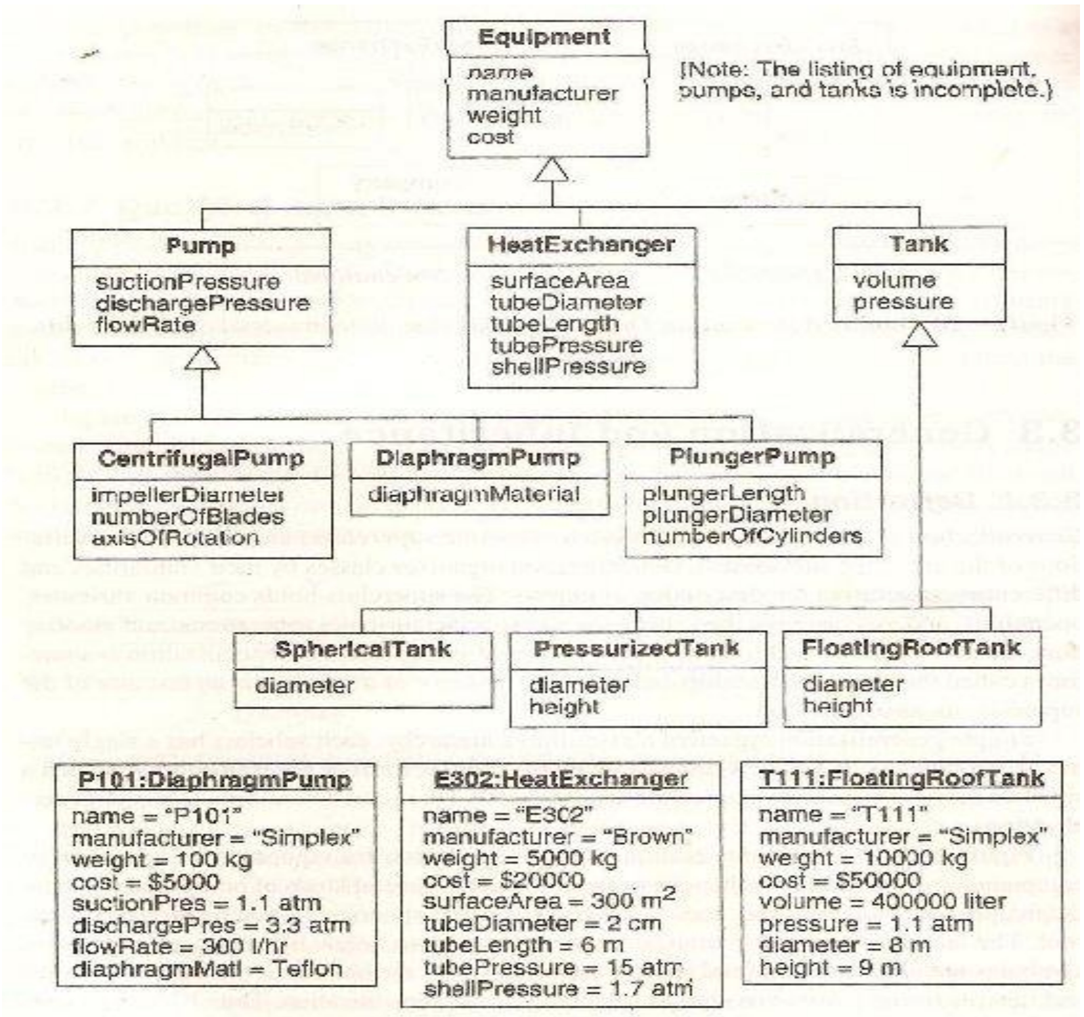
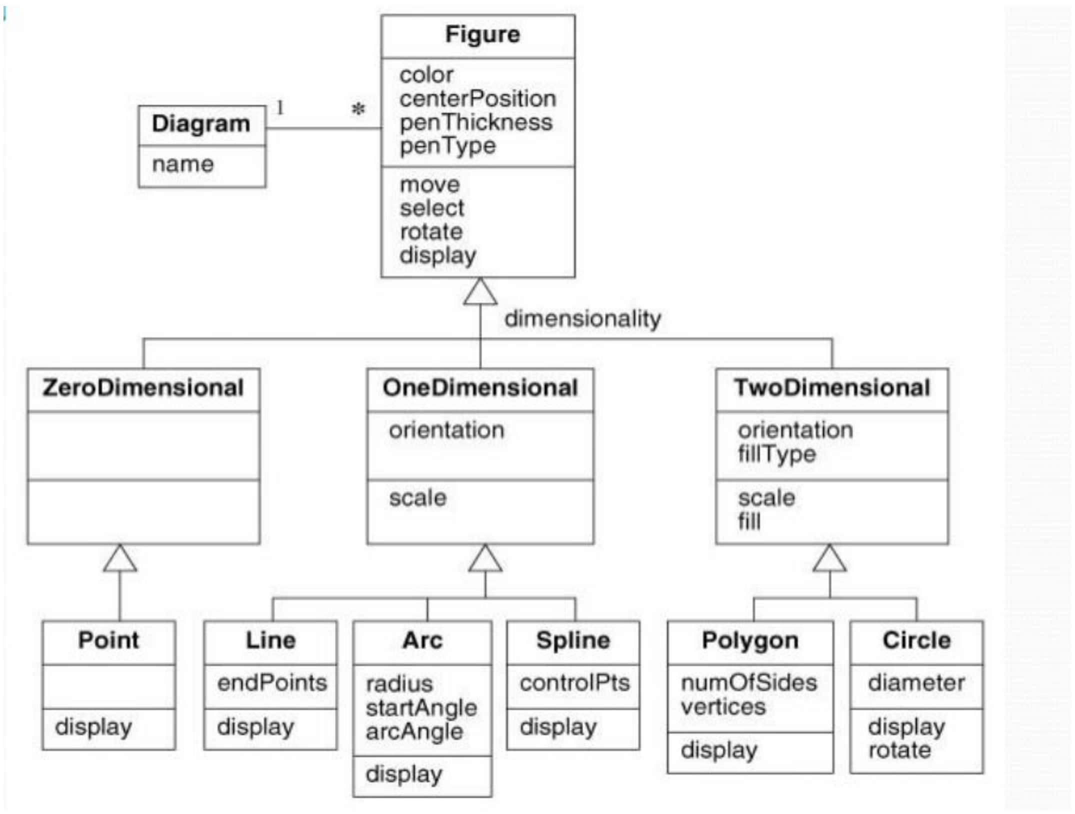


Fig (a)



Fig(b)

- **'**move**'**, **'**select**'**, 'rotate', and 'display' are operations that all subclasses inherit.
- **'**scale**'** applies to one-dimensional and two-dimensional figures.
- 'fill' applies only to two-dimensional figures.

Used for three purposes:

- **Support of polymorphism:** polymorphism increases the flexibility of software. Adding a new subclass and automatically inheriting superclass behavior.
- **Structuring the description of objects**: Forming a classification, organizing objects according to their similarities. It is much more profound than modelling each class individually and in isolation of other similar classes.
- **Enabling code reuse**: Reuse is more productive than repeatedly writing code from scratch.

Overriding features

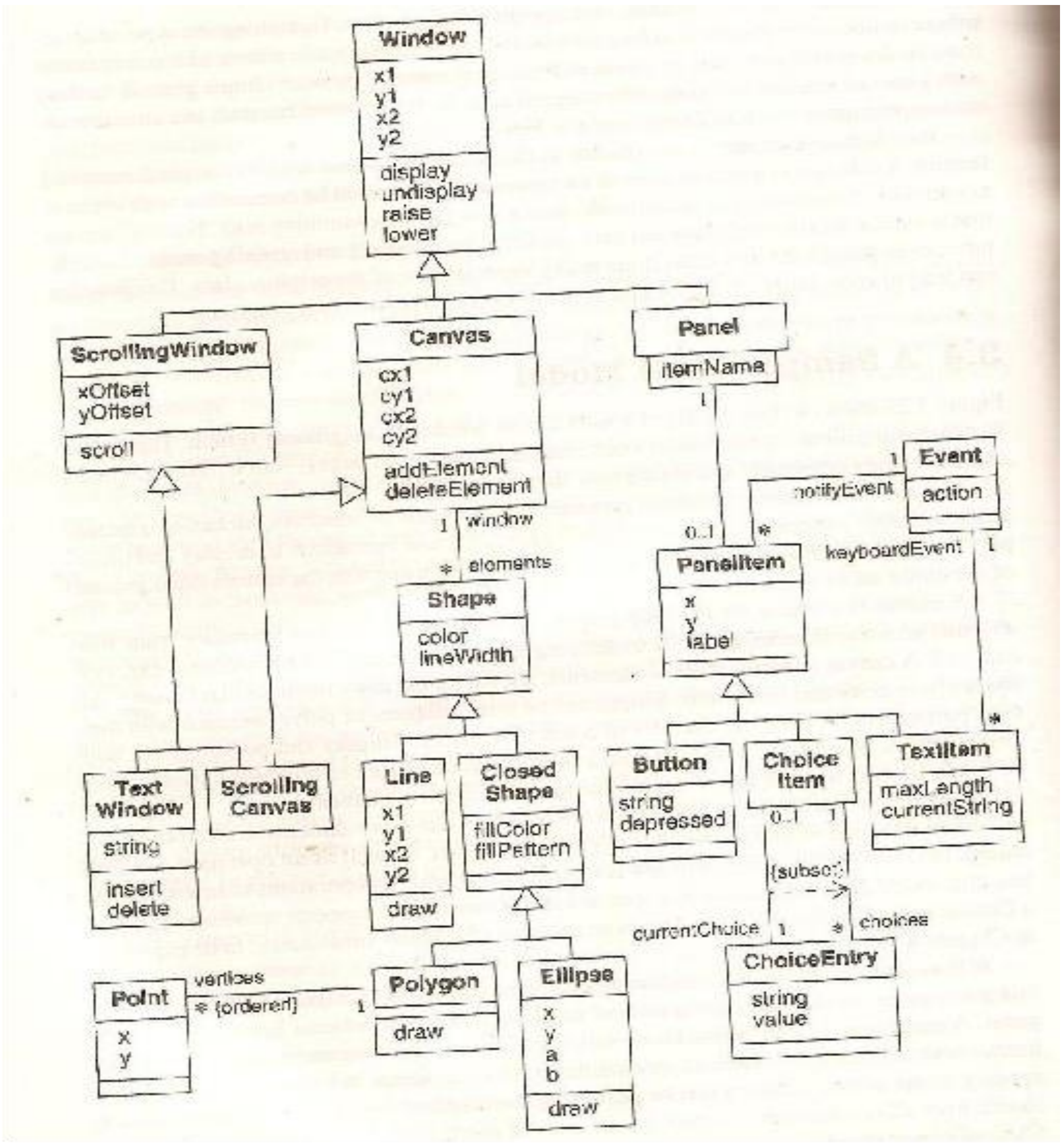
- A subclass may override a superclass feature by defining a feature with the same name. The overriding feature (subclass feature) refines and replaces the overridden feature (superclass feature).

  Why override feature?
- To specify behavior that depends on subclass.

- To tighten the specification of a feature.
- To improve performance.

- In fig (b) (previous page) each leaf subclasses had overridden 'display' feature.
- **Note:** You may override methods and default values of attributes. You should never override the signature, or form of a feature.

A Sample Class Model



NAVIGATION OF CLASS MODELS

- Class models are useful for more than just data structure. In particular, navigation of class model lets you express certain behavior. Furthermore, navigation exercises a class model and uncovers hidden flaws and omission, which you can then repair.
- UML incorporates a language that can be used for navigation, the object constraint language(OCL).

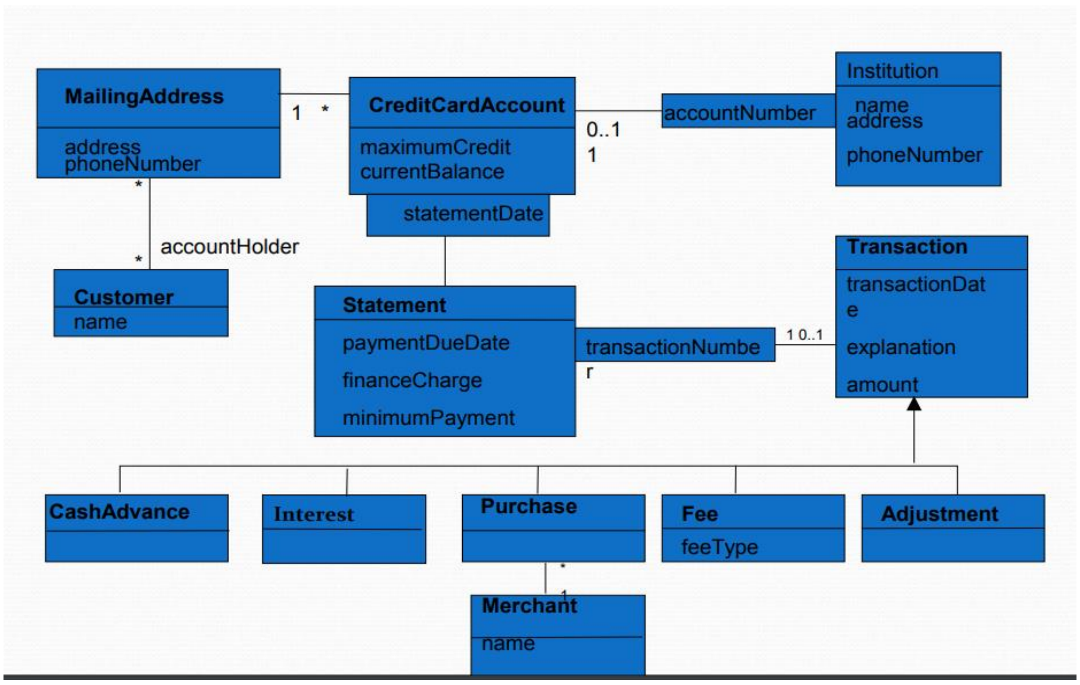


Fig: class model for managing credit card account

OCL can traverse the constructs in class models.

1. **Attributes:** You can traverse from an object to an attribute value. Syntax: source object followed by dot and then attribute name.
Ex: aCreditCardAccount.maximumcredit

2. **Operations: Y**ou can also invoke an operation for an object or collection of objects. Syntax: source object or object collection, followed by dot and then the operation followed by parenthesis even if it has no arguments. OCL has special operations that operate on entire collections (as opposed to operating on each object in a collection). Syntax for collection operation is: source object collection followed by "->", followed by the operation.

3. **Simple associations:** Dot notation is also used to traverse an association to a target end. Target end maybe indicated by an association end name, or class name (if there is no ambiguity).
Ex: refer fig in next page.
   - ➢ aCustomer.MailingAddress yields a set of addresses for a customer (the target end has "many" multiplicity).

- aCreditCardAccount.MailingAddress yields a single address(the target end has multiplicity of "one").

4. **Qualified associations:** The expression aCreditCardAccount.Statement [30 November 1999] finds the statement for a credit card account with the statement date of November 1999. The syntax is to enclose the qualifier value in brackets.

5. **Associations classes:** Given a link of an association class, you can find the constituent objects and vice versa.

6. **Generalization:** Traversal of a generalization hierarchy is implicit for the OCL notation.

7. **Filters:** Most common filter is 'select' operation. Ex:aStatement.Transaction->select(amount>$100).

Write an OCL expression for –

1. What transactions occurred for a credit card account within a time interval?

Soln: aCreditCardAccount.Statement.Transaction ->

select(aStartDate<=TransactionDate and

TransactionDate<=anEndDate)

2. What volumes of transactions were handled by an institution in the last year?

Soln: anInstitution.CreditCardAccount.Statement.Transaction ->

select(aStartDate<=TransactionDate and TransactionDate<=anEndDate).amount->sum( )

3. What customers patronized a merchant in the last year by any kind of credit card?

Soln: aMerchant.Purchase ->

select(aStartDate<=TransactionDate and transactionDate<=anEndDate).Statement.

CreditCardAccount.MailingAddress.Cu stomer ->asset( )

4. How many credit card accounts does a customer currently have?

Soln: aCustomer.MailingAddress.CreditCardAccount -> size( )

5. What is the total maximum credit for a customer for all accounts?

Soln: acustomer.MailingAddress.CreditCardAccount.

Maximumcredit->sum()