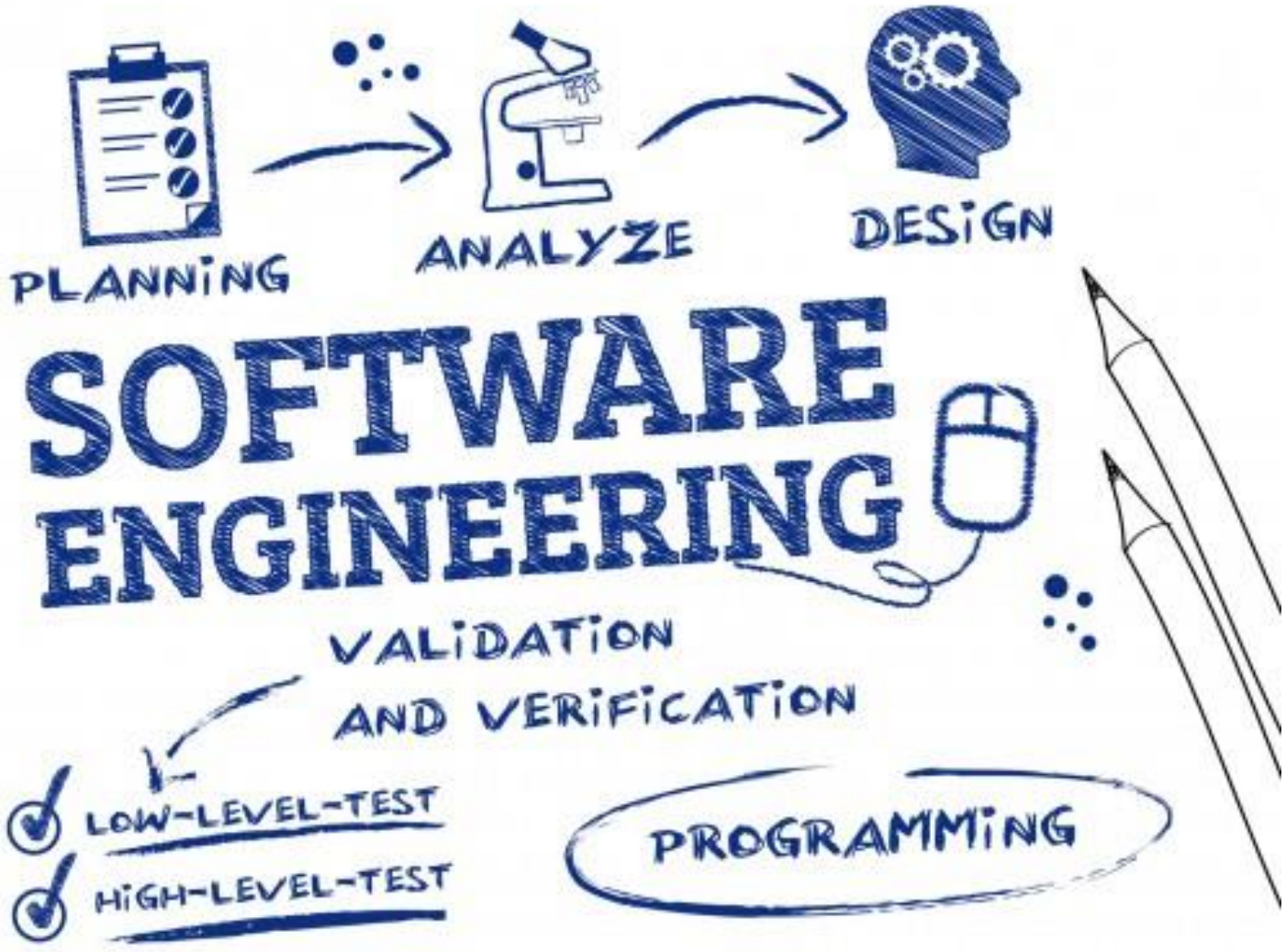


Prof. Tirup Parmar

S.Y. B.Sc(IT) – Sem4

Software Engineering Notes - 2017

Mumbai University

Course Code: USIT404

Syllabus

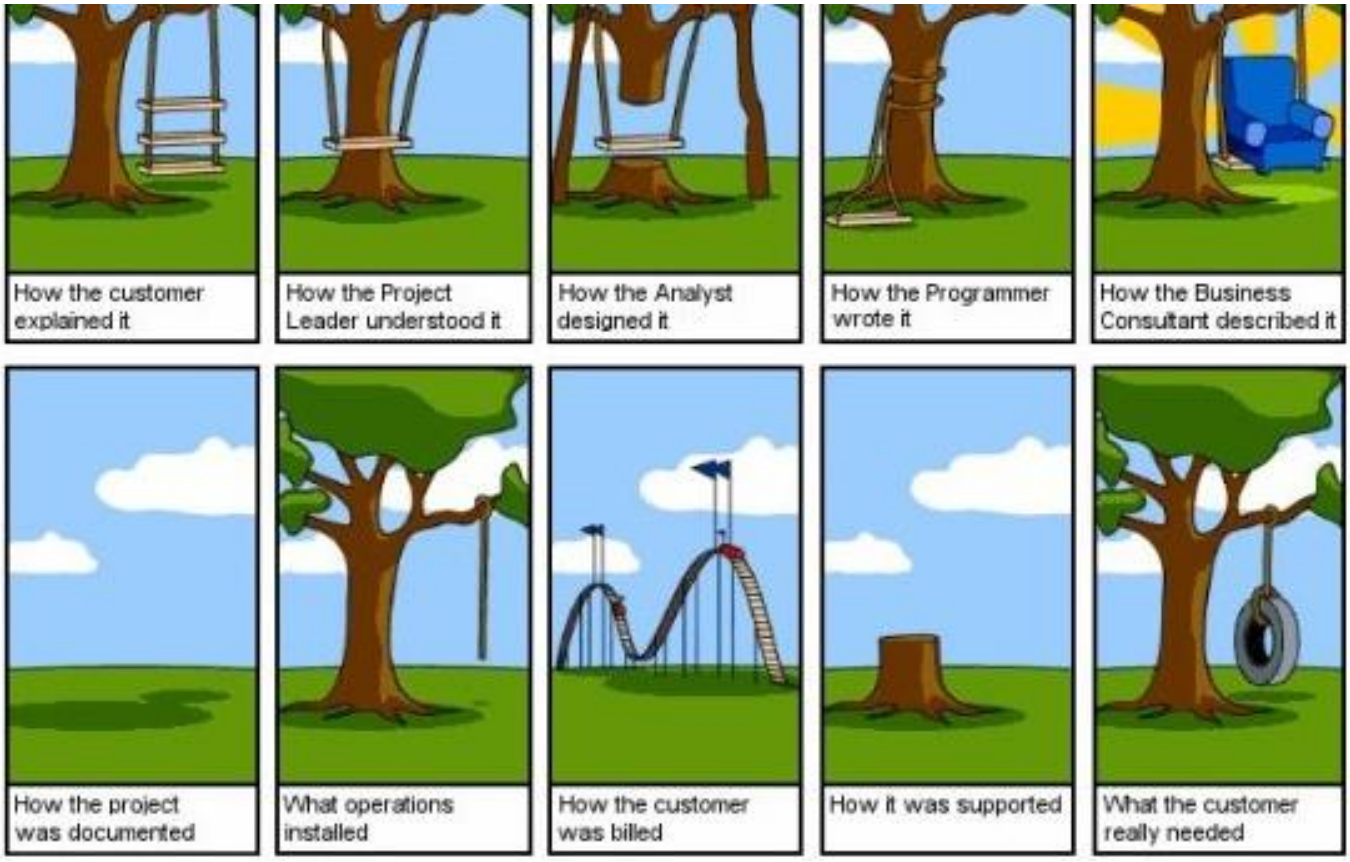
Unit	Details
1	**Introduction:** What is software engineering? Software Development Life Cycle, Requirements Analysis, Software Design, Coding, Testing, Maintenance etc. **Software Requirements:** Functional and Non-functional requirements, User Requirements, System Requirements, Interface Specification, Documentation of the software requirements. **Software Processes:** Process and Project, Component Software Processes. **Software Development Process Models.** • Waterfall Model. • Prototyping. • Iterative Development. • Rational Unified Process. • The RAD Model • Time boxing Model. **Agile software development**: Agile methods, Plan-driven and agile development, Extreme programming, Agile project management, Scaling agile methods.
2	**Socio-technical system:** Essential characteristics of socio technical systems, Emergent System Properties, Systems Engineering, Components of system such as organization, people and computers, Dealing Legacy Systems. **Critical system:** Types of critical system, A simple safety critical system, Dependability of a system, Availability and Reliability, Safety and Security of Software systems. **Requirements Engineering Processes:** Feasibility study, Requirements elicitation and analysis, Requirements Validations, Requirements Management. **System Models:** Models and its types, Context Models, Behavioural Models, Data Models, Object Models, Structured Methods.
3	**Architectural Design:** Architectural Design Decisions, System Organisation, Modular Decomposition Styles, Control Styles, Reference Architectures. **User Interface Design:** Need of UI design, Design issues, The UI design Process, User analysis, User Interface Prototyping, Interface Evaluation. **Project Management** Software Project Management, Management activities, Project Planning, Project Scheduling, Risk Management. **Quality Management:** Process and Product Quality, Quality assurance and Standards, Quality Planning, Quality Control, Software Measurement and Metrics.

UNIT I

Problems in software development

Common Issues:-

- The final Software doesn´t fulfill the needs of the customer.
- Hard to extend and improve: if you want to add a functionality later is mission impossible.
- Bad documentation.
- Bad quality: frequent errors, hard to use, ...
- More time and costs than expected



Conclusion

Programming is NOT enough!

It is not enough to do your best: you must Know what to do, and THEN do your best.
-- *W. Edwards Deming*

And Since…

A clever person solves a problem.
A wise person avoids it.
- Albert Einstein

Solution

Software Engineering

What is it?

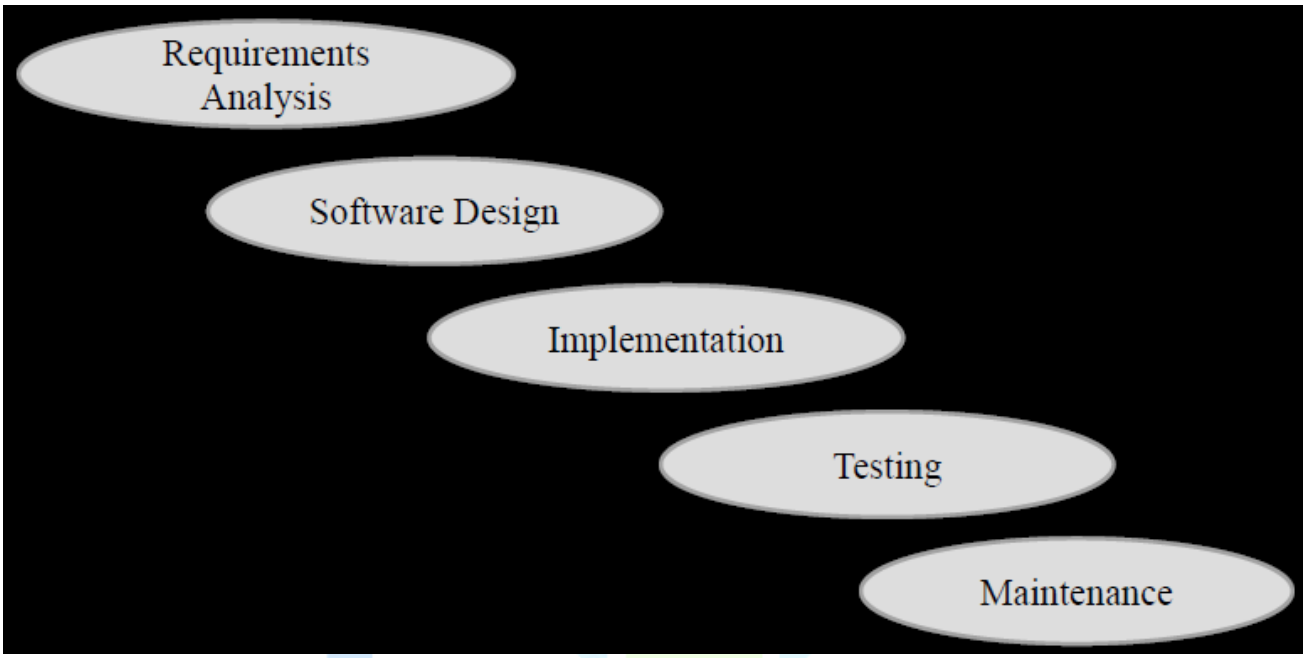
The study and application of methodologies to develop quality software that fulfill customer needs.

Objetive

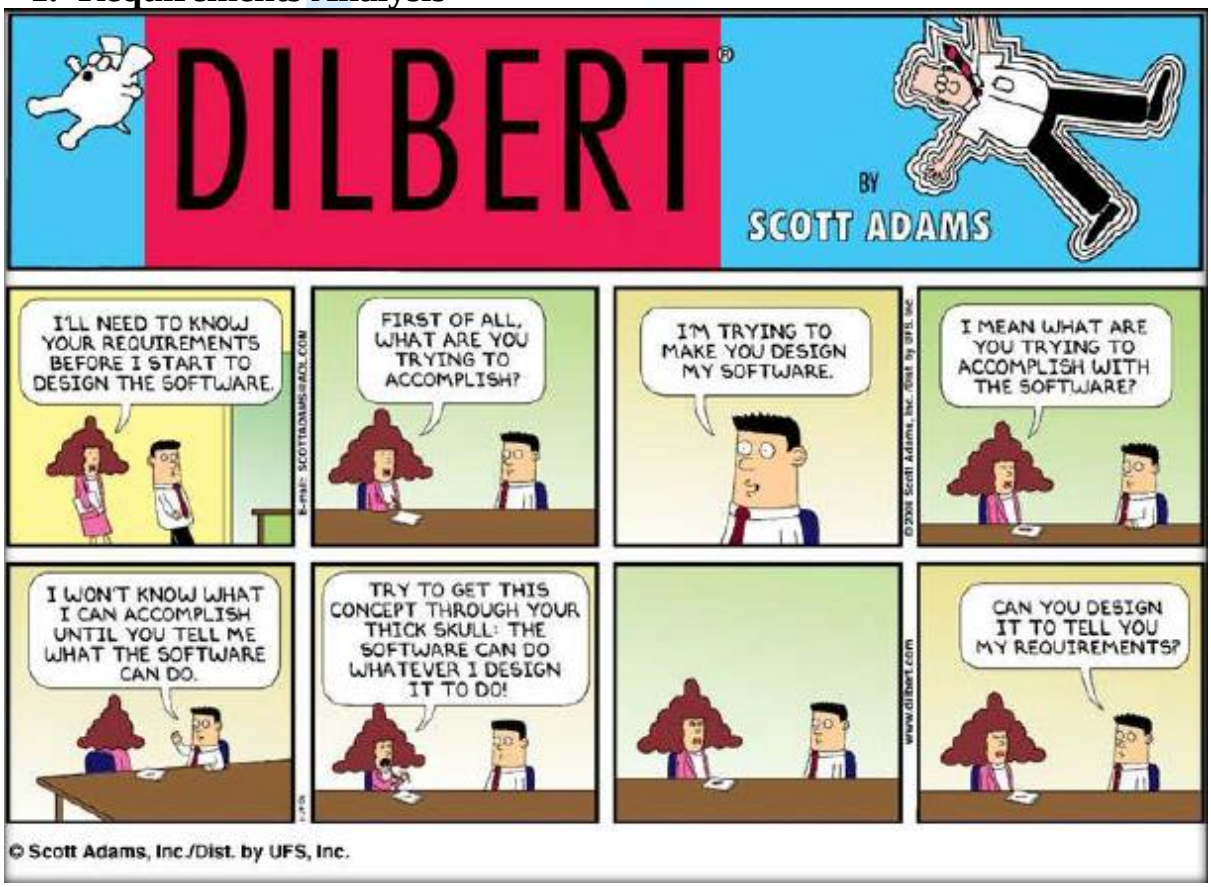
To produce software that is:
• On time: is deliver at the established date.
• Reliable: doesn´t crash.
• Complete: good documentation, fulfill customer needs.

The team

Stages for software development

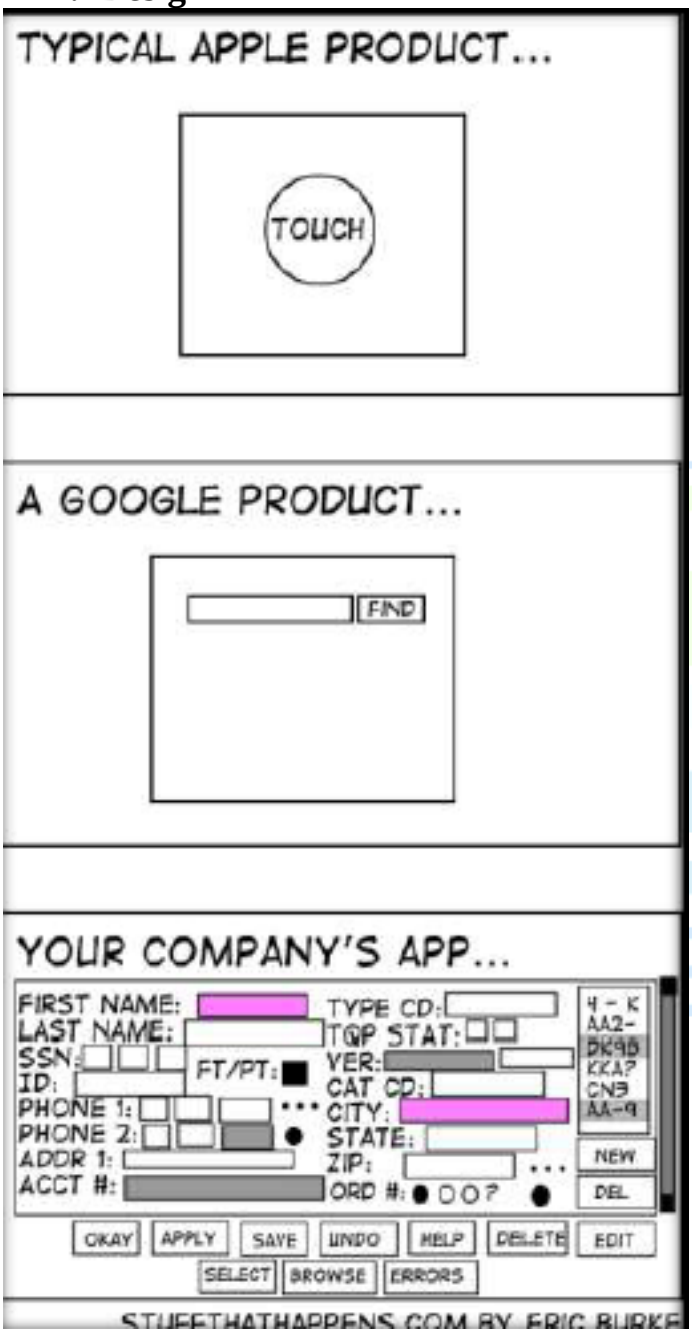


1. Requirements Analysis



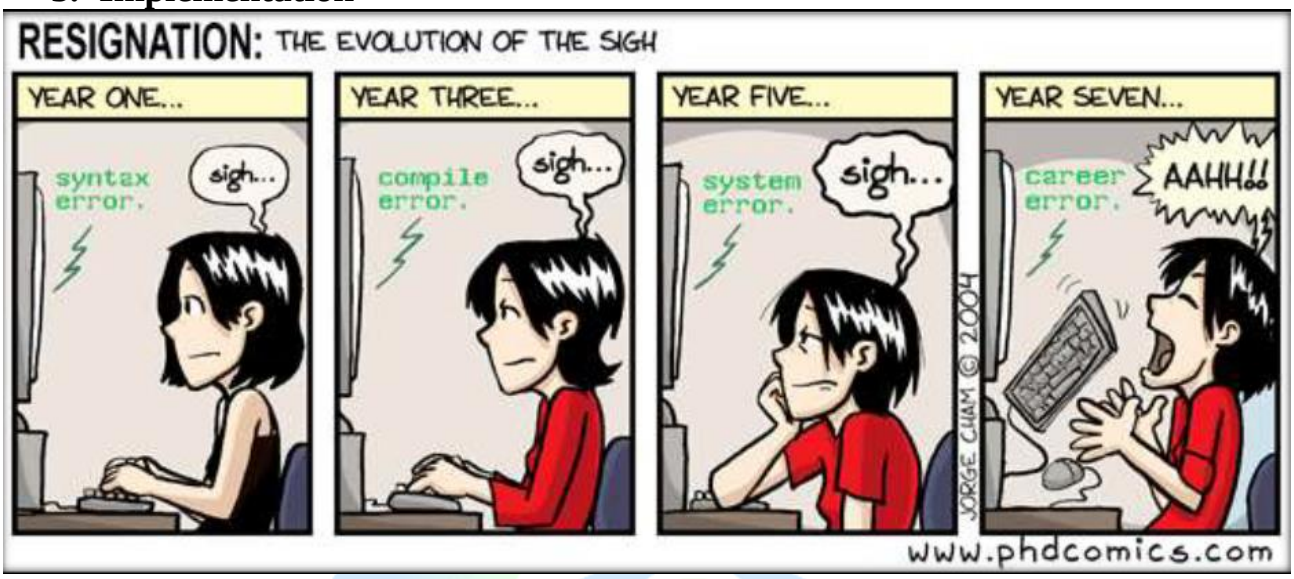
Find out what the client want the software to do

2. **Design**



Planning the software solution

3. Implementation



Code!!!

4. Testing

Executing the application trying to find software bugs

5. Maintenance

Any activity oriented to change an existing software product.

SO lets start in real now with the actual book definitions :P

Important Key terms

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.

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.

INTRODUCTION TO SOFTWARE ENGINEERING

The term ***software engineering*** is composed of two words, software and engineering.

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

So, we can define ***software engineering*** as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

IEEE defines software engineering as:
The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are absolutely necessary to achieve a good quality software cost effectively.

Without using software engineering principles it would be difficult to develop large programs. In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes. Software engineering helps to reduce this programming complexity. Software engineering principles use two important techniques to reduce problem complexity: ***abstraction*** and ***decomposition*****.** The principle of abstraction implies that a problem can be simplified by omitting irrelevant details. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose. Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on. Abstraction is a powerful way of reducing the complexity of the problem. The other approach to tackle problem complexity is decomposition. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one. However, in this technique any random decomposition of a problem into smaller parts will not help. The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different components can be combined to get the full solution. A good decomposition of a problem should minimize interactions among various components. If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

Q-1) Explain the need of Software Engineering.

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

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

Q-2) What are the 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.

Q-3) Explain the CHARACTERESTICS OF GOOD SOFTWARE.

A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

- Operational
- Transitional
- Maintenance

Well-engineered and crafted software is expected to have the following characteristics:

1. **Operational**

This tells us how well software works in operations. It can be measured on:

- Budget
- Usability
- Efficiency
- Correctness
- Functionality
- Dependability
- Security
- Safety

2. **Transitional**

This aspect is important when the software is moved from one platform to another:

- Portability
- Interoperability
- Reusability
- Adaptability

3. **Maintenance**

This aspect briefs about how well a software has the capabilities to maintain itself in the ever-changing environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products

Q-4) Explain the different Application types.

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

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

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

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

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

6. **Systems for modeling and simulation**
   - These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects.

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

8. **Systems of systems**
   - These are systems that are composed of a number of other software systems.

SOFTWARE DEVELOPMENT LIFE CYCLE

Q-5) What is software life cycle model? & What is the NEED FOR A SOFTWARE LIFE CYCLE MODEL

LIFE CYCLE MODEL

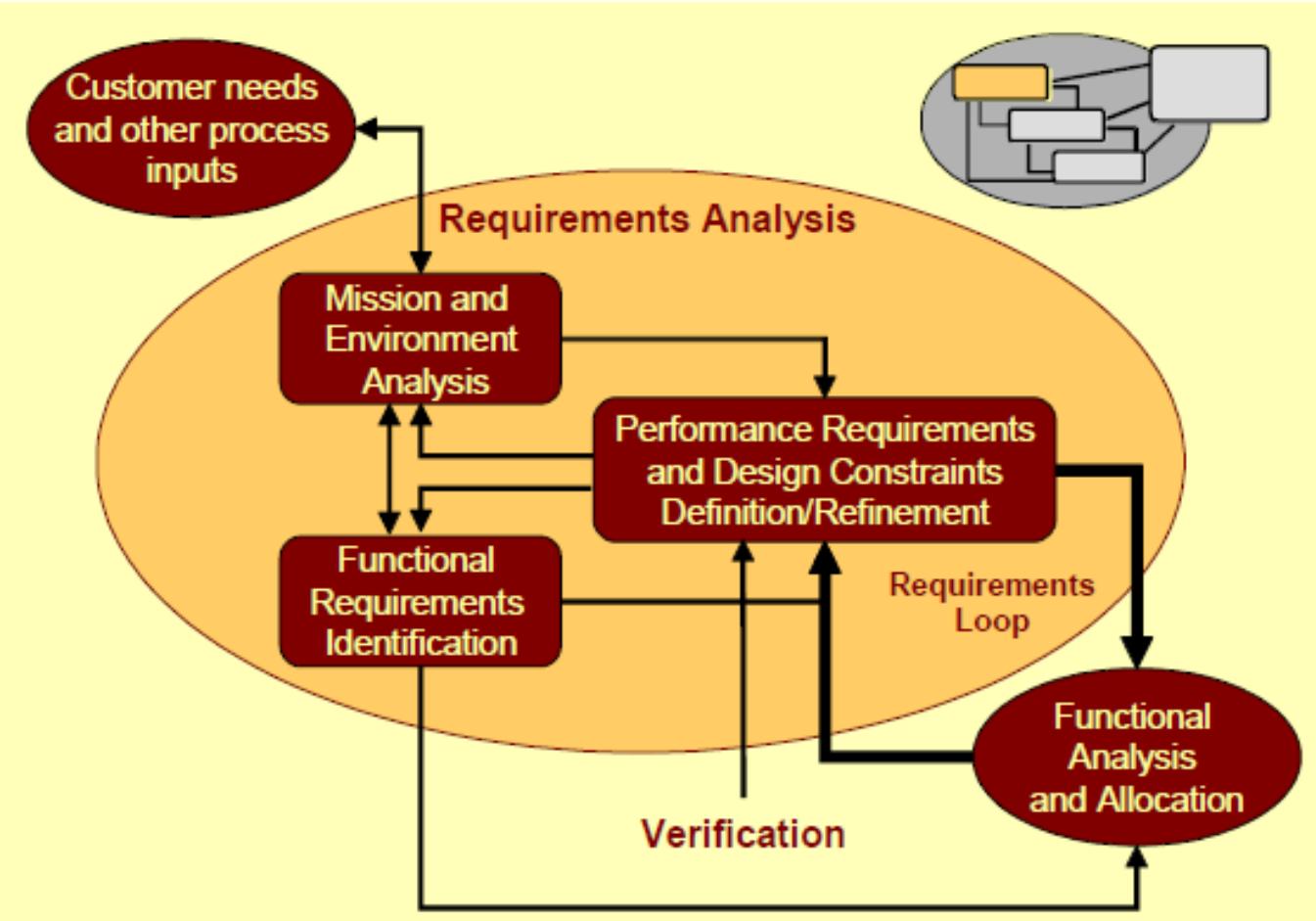
- A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle.

- A life cycle model represents all the activities required to make a software product transit through its life cycle phases.
- It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to retirement.
- Different life cycle models may map the basic development activities to phases in different ways.
- Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out.

THE NEED FOR A SOFTWARE LIFE CYCLE MODEL

- The development team must identify a suitable life cycle model for the particular project and then adhere to it.
- Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner.
- When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure.
- This problem can be illustrated by using an example. Suppose a software development problem is divided into several parts and the parts are assigned to the team members.
- From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like.
- It is possible that one member might start writing the code for his part, another might decide to prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him.
- This would be one of the perfect recipes for project failure. A software life cycle model defines entry and exit criteria for every phase.
- A phase can start only if its phase-entry criteria have been satisfied. So without software life cycle model the entry and exit criteria for a phase cannot be recognized.
- Without software life cycle models it becomes difficult for software project managers to monitor the progress of the project.

What are the requirements ?



Requirements Analysis (Step 1) is one of the first activities of the System Engineering Process and functions somewhat as an interface between the internal activities and the external sources providing inputs to the process. It examines, evaluates, and translates the external inputs into a set of functional and performance requirements that are the basis for the Functional Analysis and Allocation. It links with the Functional Analysis and Allocation to form the

requirements loop of the System Engineering Process. The goal of requirements analysis is to determine the needs that make up a system to satisfy an overall need.

Software Design

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

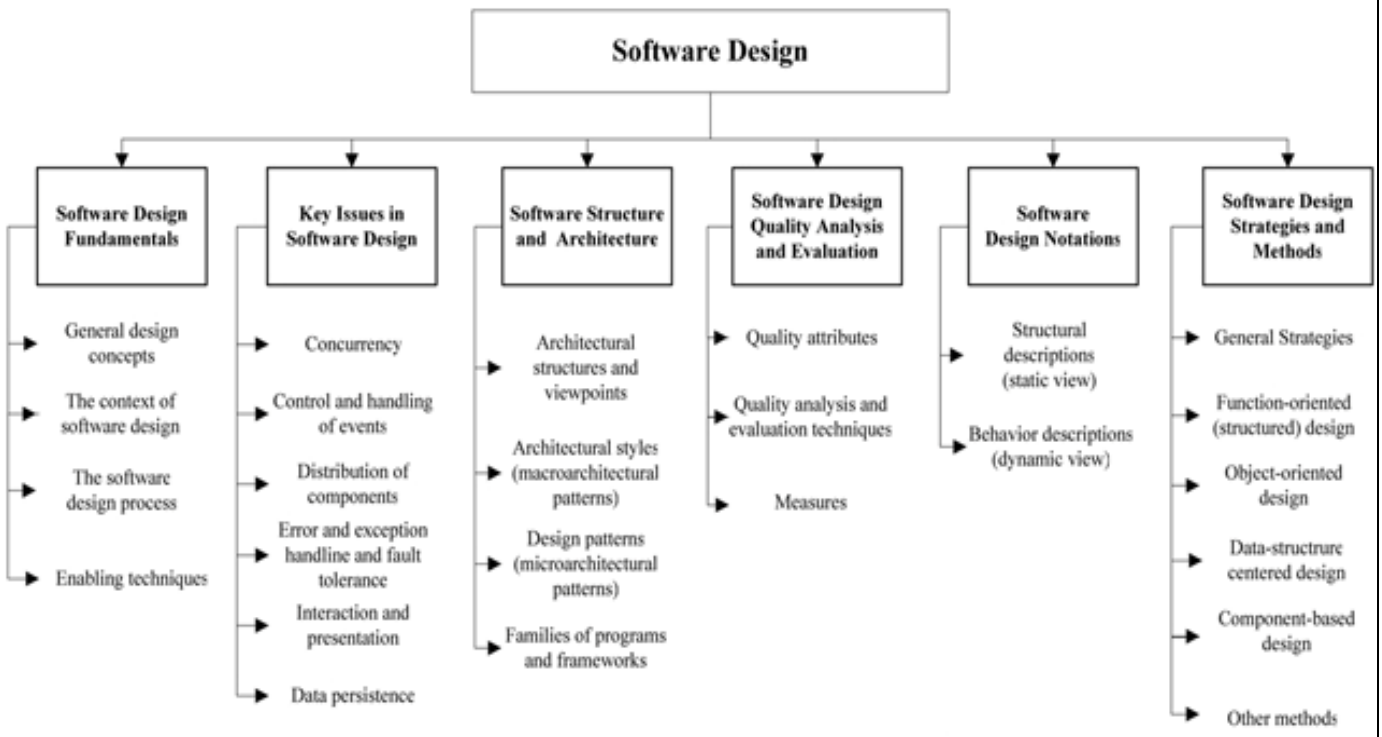
For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.

Software Design Levels

Software design yields three levels of results:

- **Architectural Design -** The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
- **High-level Design-** The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.
- **Detailed Design-** Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.



Coding

Writing an efficient software code requires a thorough knowledge of programming. This knowledge can be implemented by following a coding style which comprises several guidelines that help in writing the software code efficiently and with minimum errors. These guidelines, known as **coding guidelines,** are used to implement individual programming language constructs, comments, formatting, and so on. These guidelines, if followed, help in preventing errors, controlling the complexity of the program, and increasing the readability and understandability of the program.

Implementing Coding Guidelines

If coding guidelines are used in a proper manner, errors can be detected at the time of writing the software code. Such detection in early stages helps in increasing the performance of the software as well as reducing the additional and unplanned costs of correcting and removing errors. Moreover, if a well-defined coding guideline is applied, the program yields a software system that is easy to comprehend and maintain. Some of the coding guidelines that are followed in a programming language are listed below.

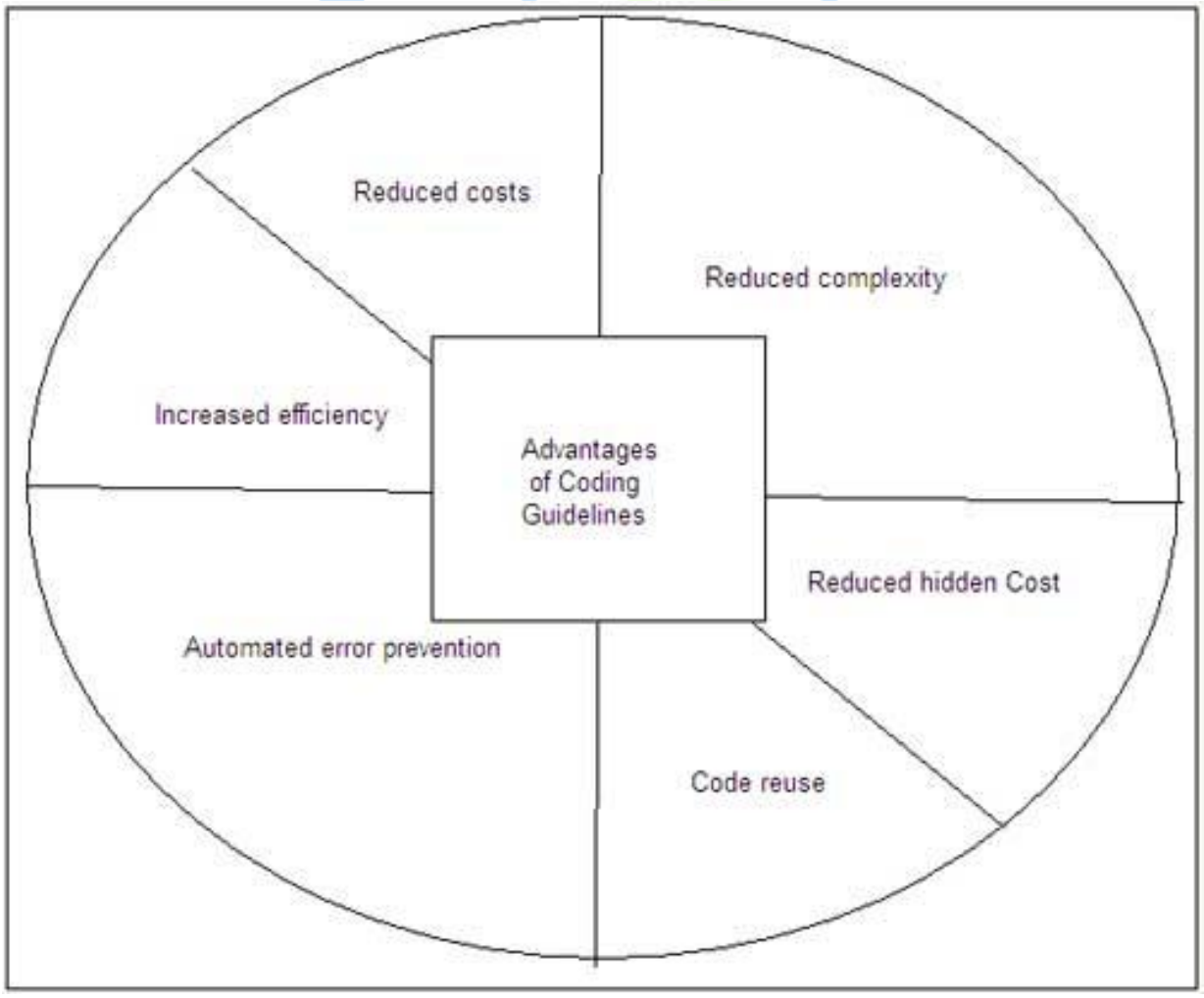
- All the codes should be properly commented before being submitted to the review team.
- All curly braces should start from a new line.
- All class names should start with the abbreviation of each group. For example, AA and CM can be used instead of academic administration and course management, respectively.

- Errors should be mentioned in the following format: [error code]: [explanation]. For example, 0102: null pointer exception, where 0102 indicates the error code and null pointer exception is the name of the error.
- Every 'if statement should be followed by a curly braces even if there exists only a single statement.
- Every file should contain information about the author of the file, modification date, and version information.

Similarly, some of the commonly used coding guidelines in a database (organized collection of information that is systematically organized for easy access and analysis) are listed below.

- Table names should start with TBL. For example, TBL_STUDENT.
- If table names contain one word, field names should start with the first three characters of the name of the table. For example, STU_FIRSTNAME.
- Every table should have a primary key.
- Long data type (or database equivalent) should be used for the primary key.

Advantages of Coding Guidelines



- **Increased efficiency:** Coding guidelines can be used effectively to save time spent on gathering unnecessary details. These guidelines increase the efficiency of the software team while the software development phase is carried out. An efficient software code is fast and economical. Software coding guidelines are used to increase efficiency by making the team productive, thus, ensuring that the software is delivered to the user on time.
- **Reduced costs:** Coding guidelines are beneficial in reducing the cost incurred on the software project. This is possible since coding guidelines help in detecting errors in the early stages of the software development. Note that if errors are discovered after the software is delivered to the user, the process of rectifying them becomes expensive as additional costs are incurred on late detection, rework, and retesting of the entire software code.
- **Reduced complexity:** The written software code can be either simple or complex. Generally, it is observed that a complex segment of software code is more susceptible to errors than a segment containing a simple software code. This is because a complex software code reduces readability as well as understandability. In addition, the complex software code may be inefficient in functioning and use of resources. However, if the code is written using the given coding guidelines, the problem of complexity can be significantly avoided as the probability of error occurrence reduces substantially.
- **Reduced hidden costs:** Coding guidelines, if adhered to in a proper manner, help to achieve a high-quality software code. The software quality determines the efficiency of the software. Software quality is the degree to which user requirements are accomplished in the software along with conformity to standards. Note that if quality is not considered while developing the software, the cost for activities such as fixing errors, redesigning the software, and providing technical support increases considerably.
- **Code reuse:** Using coding guidelines, software developers are able to write a code that is more robust and create individual modules of the software code. The reason for making separate code segment is to enable reusability of the modules used in the software. A reusable module can be used a number of times in different modules in one or more software.
- **Automated error prevention:** The coding guidelines enable Automated Error Prevention (AEP).This assures that each time error occurs in software, the software development activity is improved to prevent similar errors in future. AEP begins with detecting errors in the software, isolating its cause, and then searching the cause of error generation. Coding guidelines are useful in preventing errors as they allow implementation of requirements that prevent the most common and damaging errors in the software code.

Software Testing

Software Testing is evaluation of the software against requirements gathered from users and system specifications. Testing is conducted at the phase level in software development life cycle or at module level in program code. Software testing comprises of Validation and Verification.

Software Validation

Validation is process of examining whether or not the software satisfies the user requirements. It is carried out at the end of the SDLC. If the software matches requirements for which it was made, it is validated.

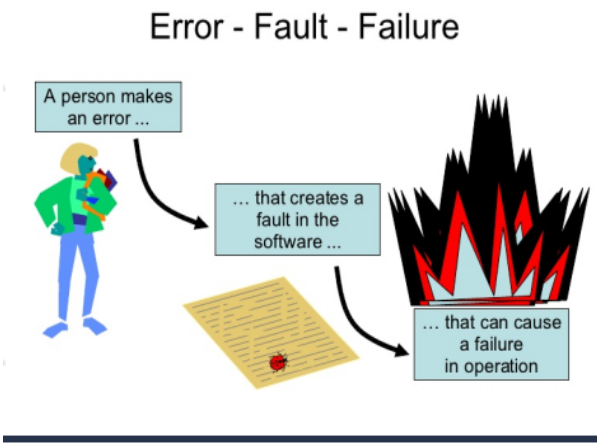
- Validation ensures the product under development is as per the user requirements.
- Validation answers the question – "Are we developing the product which attempts all that user needs from this software ?".
- Validation emphasizes on user requirements.

Software Verification

Verification is the process of confirming if the software is meeting the business requirements, and is developed adhering to the proper specifications and methodologies.

- Verification ensures the product being developed is according to design specifications.
- Verification answers the question– "Are we developing this product by firmly following all design specifications ?"
- Verifications concentrates on the design and system specifications.

Target of the test are -



1. A FAILURE IS NON-FULFILLMENT OF REQUIREMENTS, A DISCREPANCY BETWEEN ACTUAL RESULT OR BEHAVIOR (IDENTIFIED IN REQUIREMENT SPECIFICATION)
2. FAILURE IN A SOFTWARE MEANS THE SOFTWARE DOESN'T PERFORM FUNCTIONAL REQUIREMENTS, THE FAILURE OCCUR IN S/W BECAUSE OF FAULT. EVERY FAULT, OR DEFECT OR BUG IN S/W IS PRESENT SINCE IT WAS DEVELOPED OR CHANGED.
3. A FAILURE HAS ITS ROOTS IN A FAULT IN THE SOFTWARE. THESE FAULT IS ALSO CALLED A DEFECT OR INTERNAL ERROR PROGRAMMER GENERALLY USE A TERM BUG FOR FAULT E.G. OF FAULT E.G. OF FAULT CAN BE THE PROGRAMMER MIGHT BE WRONGLY PROGRAMMED OR FORGOTTEN CODE IN APPLICATION.
4. IT IS POSSIBLE THAT A FAULT IS HIDDEN BY ONE OR MORE OTHER FAULTS IN DIFFERENT PARTS OF APPLICATION. IN THAT CASE, A FAILURE ONLY OCCURS AFTER THE MASKING DEFECTS HAVE BEEN CORRECTED.
5. THE CAUSE OF FAULT OR DEFECT IS AN ERROR OR MISTAKES BY PERSON. E.G. WRONG PROGRAMMING BY DEVELOPER OR A MISUNDERSTANDING OF COMMANDS IN PROGRAMMING LANGUAGE.
6. HOWEVER FAULTS MAY BE EVEN CAUSE BY ENVIRONMENTAL CONDITIONS LIKE RADIATION, MAGNETISM ETC. THAT INTRODUCE HARDWARE PROBLEM.

Manual Vs Automated Testing

Testing can either be done manually or using an automated testing tool:

- **Manual** - This testing is performed without taking help of automated testing tools. The software tester prepares test cases for different sections and levels of the code, executes the tests and reports the result to the manager.

  Manual testing is time and resource consuming. The tester needs to confirm whether or not right test cases are used. Major portion of testing involves manual testing.

- **Automated** This testing is a testing procedure done with aid of automated testing tools. The limitations with manual testing can be overcome using automated test tools.

A test needs to check if a webpage can be opened in Internet Explorer. This can be easily done with manual testing. But to check if the web-server can take the load of 1 million users, it is quite impossible to test manually.

There are software and hardware tools which helps tester in conducting load testing, stress testing, regression testing.

Testing Approaches

Tests can be conducted based on two approaches –

- Functionality testing
- Implementation testing

When functionality is being tested without taking the actual implementation in concern it is known as black-box testing. The other side is known as white-box testing where not only functionality is tested but the way it is implemented is also analyzed.

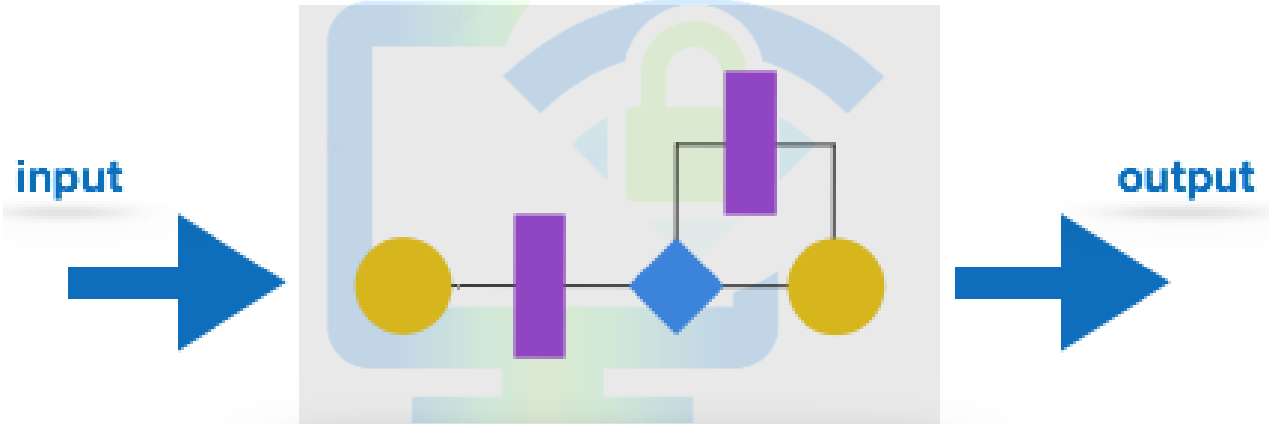
Exhaustive tests are the best-desired method for a perfect testing. Every single possible value in the range of the input and output values is tested. It is not possible to test each and every value in real world scenario if the range of values is large.

Black-box testing

It is carried out to test functionality of the program. It is also called 'Behavioral' testing. The tester in this case, has a set of input values and respective desired results. On providing input, if the output matches with the desired results, the program is tested 'ok', and problematic otherwise.

White-box testing

It is conducted to test program and its implementation, in order to improve code efficiency or structure. It is also known as 'Structural' testing.



In this testing method, the design and structure of the code are known to the tester. Programmers of the code conduct this test on the code.

Testing Levels

Testing itself may be defined at various levels of SDLC. The testing process runs parallel to software development. Before jumping on the next stage, a stage is tested, validated and verified.

Testing separately is done just to make sure that there are no hidden bugs or issues left in the software. Software is tested on various levels -

Unit Testing

While coding, the programmer performs some tests on that unit of program to know if it is error free. Testing is performed under white-box testing approach. Unit testing helps developers decide that individual units of the program are working as per requirement and are error free.

Integration Testing

Even if the units of software are working fine individually, there is a need to find out if the units if integrated together would also work without errors. For example, argument passing and data updation etc.

System Testing

The software is compiled as product and then it is tested as a whole. This can be accomplished using one or more of the following tests:

- **Functionality testing** - Tests all functionalities of the software against the requirement.
- **Performance testing** - This test proves how efficient the software is. It tests the effectiveness and average time taken by the software to do desired task. Performance testing is done by means of load testing and stress testing where the software is put under high user and data load under various environment conditions.
- **Security & Portability** - These tests are done when the software is meant to work on various platforms and accessed by number of persons.

Acceptance Testing

When the software is ready to hand over to the customer it has to go through last phase of testing where it is tested for user-interaction and response. This is important because even if the software matches all user requirements and if user does not like the way it appears or works, it may be rejected.

- **Alpha testing** - The team of developer themselves perform alpha testing by using the system as if it is being used in work environment. They try to find out how user would react to some action in software and how the system should respond to inputs.
- **Beta testing** - After the software is tested internally, it is handed over to the users to use it under their production environment only for testing purpose. This is not as yet the delivered product. Developers expect that users at this stage will bring minute problems, which were skipped to attend.

Software Maintenance

Software maintenance is widely accepted part of SDLC now a days. It stands for all the modifications and updations done after the delivery of software product. There are number of reasons, why modifications are required, some of them are briefly mentioned below:

- **Market Conditions** - Policies, which changes over the time, such as taxation and newly introduced constraints like, how to maintain bookkeeping, may trigger need for modification.
- **Client Requirements** - Over the time, customer may ask for new features or functions in the software.
- **Host Modifications** - If any of the hardware and/or platform (such as operating system) of the target host changes, software changes are needed to keep adaptability.
- **Organization Changes** - If there is any business level change at client end, such as reduction of organization strength, acquiring another company, organization venturing into new business, need to modify in the original software may arise.

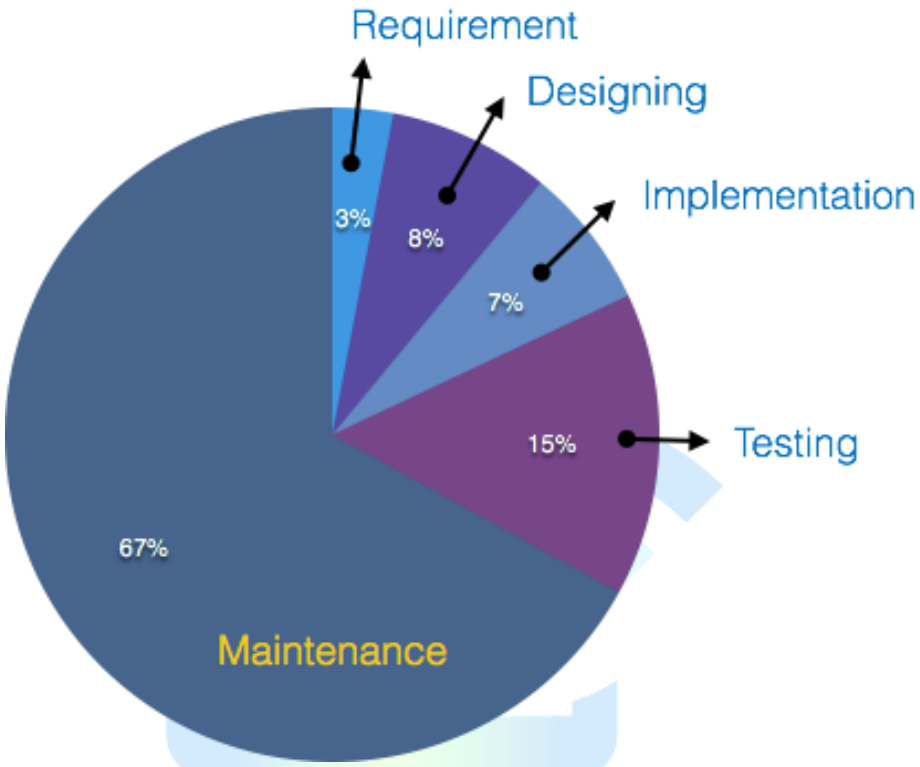
Types of maintenance

In a software lifetime, type of maintenance may vary based on its nature. It may be just a routine maintenance tasks as some bug discovered by some user or it may be a large event in itself based on maintenance size or nature. Following are some types of maintenance based on their characteristics:

- **Corrective Maintenance** - This includes modifications and updations done in order to correct or fix problems, which are either discovered by user or concluded by user error reports.
- **Adaptive Maintenance** - This includes modifications and updations applied to keep the software product up-to date and tuned to the ever changing world of technology and business environment.
- **Perfective Maintenance** - This includes modifications and updates done in order to keep the software usable over long period of time. It includes new features, new user requirements for refining the software and improve its reliability and performance.
- **Preventive Maintenance** - This includes modifications and updations to prevent future problems of the software. It aims to attend problems, which are not significant at this moment but may cause serious issues in future.

Cost of Maintenance

Reports suggest that the cost of maintenance is high. A study on estimating software maintenance found that the cost of maintenance is as high as 67% of the cost of entire software process cycle.



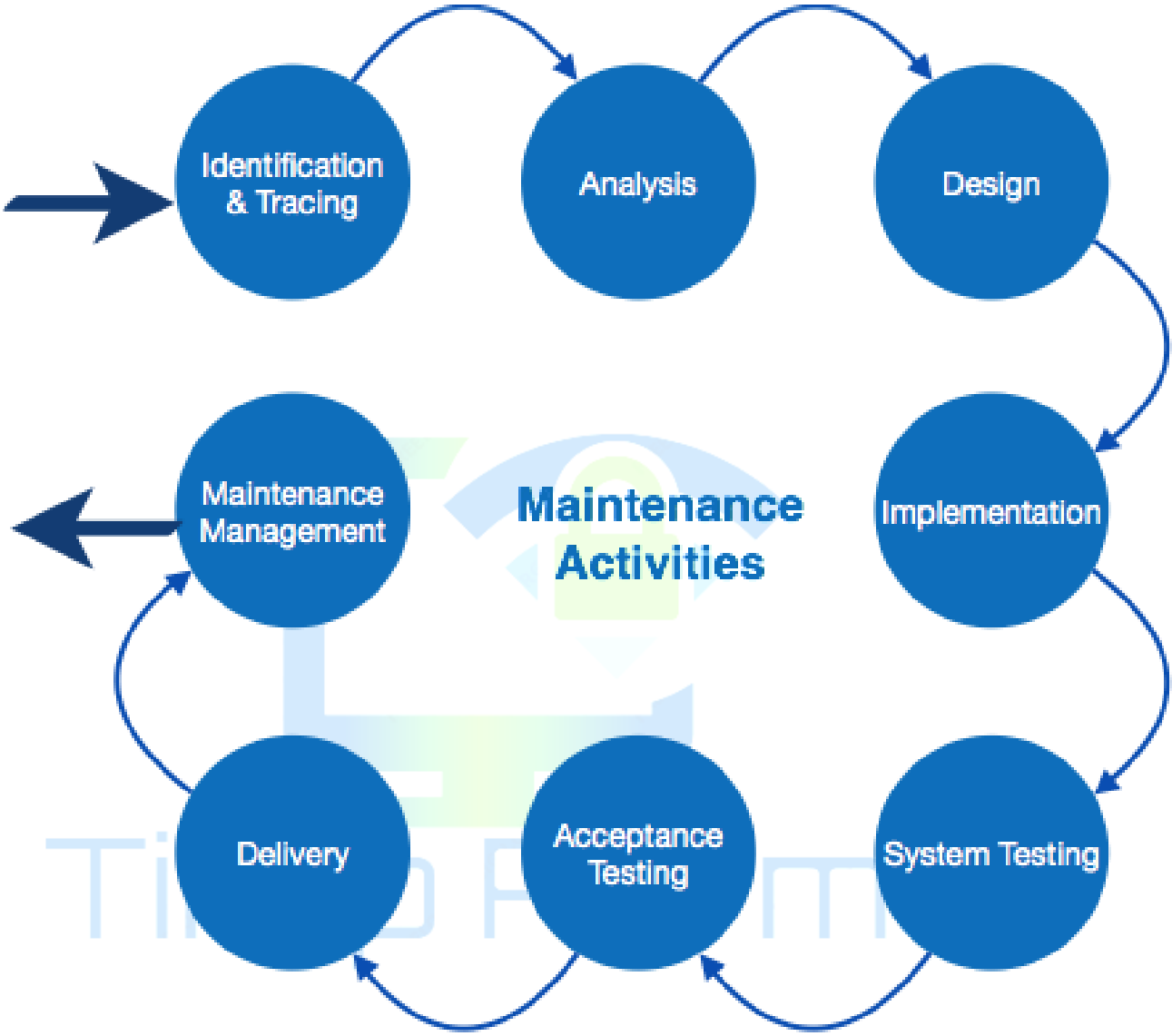
Maintenance Activities

IEEE provides a framework for sequential maintenance process activities. It can be used in iterative manner and can be extended so that customized items and processes can be included.

These activities go hand-in-hand with each of the following phase:

- **Identification & Tracing** - It involves activities pertaining to identification of requirement of modification or maintenance. It is generated by user or system may itself report via logs or error messages.Here, the maintenance type is classified also.
- **Analysis** - The modification is analyzed for its impact on the system including safety and security implications. If probable impact is severe, alternative solution is looked for. A set of required modifications is then materialized into requirement specifications. The cost of modification/maintenance is analyzed and estimation is concluded.
- **Design** - New modules, which need to be replaced or modified, are designed against requirement specifications set in the previous stage. Test cases are created for validation and verification.

- **Implementation** - The new modules are coded with the help of structured design created in the design step.Every programmer is expected to do unit testing in parallel.



- **System Testing** - Integration testing is done among newly created modules. Integration testing is also carried out between new modules and the system. Finally the system is tested as a whole, following regressive testing procedures.
- **Acceptance Testing** - After testing the system internally, it is tested for acceptance with the help of users. If at this state, user complaints some issues they are addressed or noted to address in next iteration.

- **Delivery** - After acceptance test, the system is deployed all over the organization either by small update package or fresh installation of the system. The final testing takes place at client end after the software is delivered.

  Training facility is provided if required, in addition to the hard copy of user manual.

- **Maintenance management** - Configuration management is an essential part of system maintenance. It is aided with version control tools to control versions, semi-version or patch management.

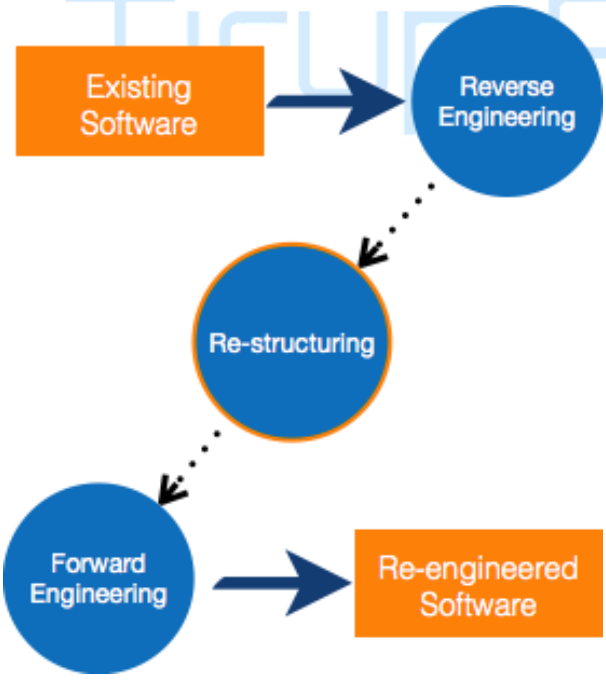
Software Re-engineering

When we need to update the software to keep it to the current market, without impacting its functionality, it is called software re-engineering. It is a thorough process where the design of software is changed and programs are re-written.

Legacy software cannot keep tuning with the latest technology available in the market. As the hardware become obsolete, updating of software becomes a headache. Even if software grows old with time, its functionality does not.

For example, initially Unix was developed in assembly language. When language C came into existence, Unix was re-engineered in C, because working in assembly language was difficult.

Other than this, sometimes programmers notice that few parts of software need more maintenance than others and they also need re-engineering.



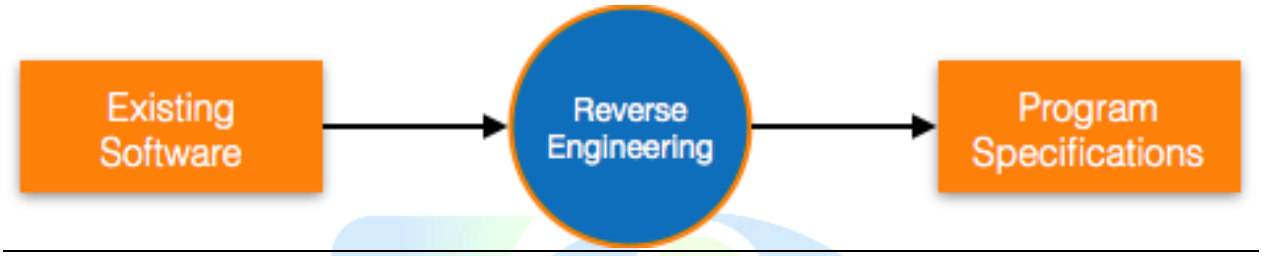
Re-Engineering Process

- **Decide** what to re-engineer. Is it whole software or a part of it?
- **Perform** Reverse Engineering, in order to obtain specifications of existing software.
- **Restructure Program** if required. For example, changing function-oriented programs into object-oriented programs.
- **Re-structure data** as required.
- **Apply Forward engineering** concepts in order to get re-engineered software.

Reverse Engineering

It is a process to achieve system specification by thoroughly analyzing, understanding the existing system. This process can be seen as reverse SDLC model, i.e. we try to get higher abstraction level by analyzing lower abstraction levels.

An existing system is previously implemented design, about which we know nothing. Designers then do reverse engineering by looking at the code and try to get the design. With design in hand, they try to conclude the specifications. Thus, going in reverse from code to system specification.



Software Requirements:

Objectives
• To introduce the concepts of user requirements and system requirements
• To describe functional and non-functional requirements
• To explain how software requirements may be organised in a requirements document

Requirements engineering

• The process of finding out, analysing, documenting, and checking the services that the customer requires from a system and its operational constraints is called RE.
• Requirement may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.

Types of requirement

• User requirements (high level abstract requirements)
– Statements in natural language plus diagrams of what services the system provides and its operational constraints.Written for customers.
• System requirements (description of what system should do)
– A structured document(also called functional specification) setting out detailed descriptions of the system's functions,services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

Definitions and specifications

User requirement Definition

1. LIBSYS shall keep track of all data required by copyright licensing agencies in INDIA and elsewhere

System Requirements Specification

1.1 On making a request for a document from LIBSYS, the requestor shall be presented with a form that records details of user and the request made

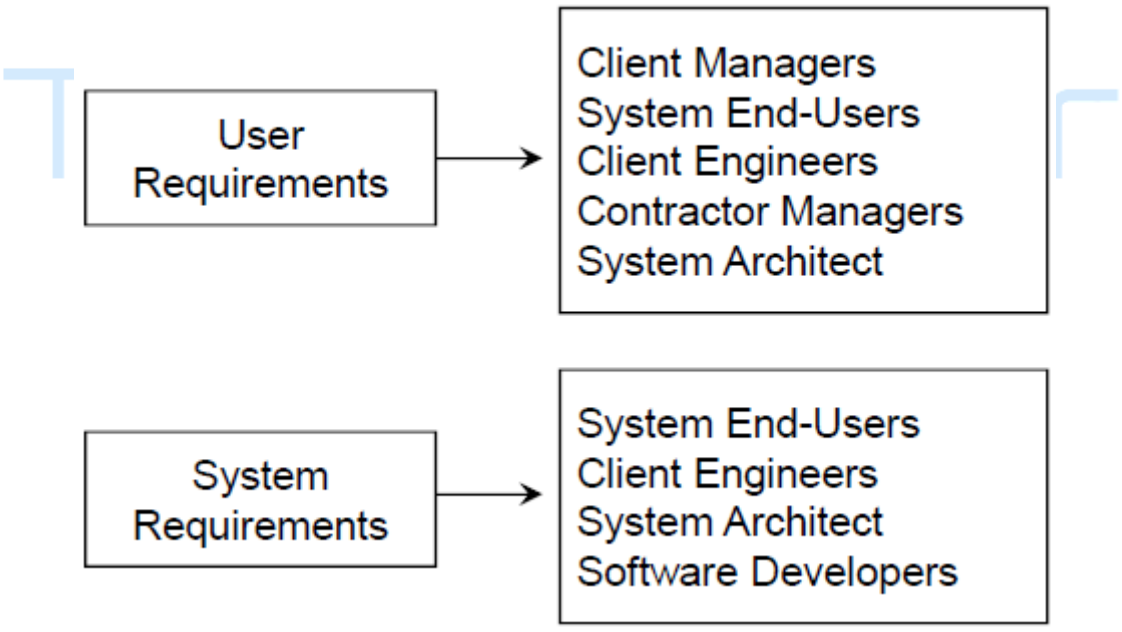
1.2 LIBSYS request forms shall be stored on the system for 5 years from the date of the request.

1.3 All LIBSYS request forms must be indexed by user, by the name of the material requested and by the supplier of the request

1.4 LIBSYS shall maintain a log of all requests that have been made to the system.

1.5 For material where authors lending rights apply, loan details shall be sent monthly to copyright licensing agencies that have registered with LIBSYS.

Requirements readers



Functional and non-functional requirements

• Software system requirements are classified as:

• Functional requirements

– Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations(and sometimes what it should NOT do).

• Non-functional requirements

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

• Domain requirements

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

Functional requirements

• Describe what the system should do.
• Depend on the type of software, expected users and the type of system where the software is used.
• Functional **user requirements** may be high-level statements of what the system should do but functional **system requirements** should describe the system services in detail, its i/o, exceptions and so on.

Example: The LIBSYS system

• A library system that provides a single interface to a number of databases of articles in different libraries.
• Users can search for, download and print these articles for personal study.

Examples of functional requirements of LIBSYS

• The user shall be able to search either all of the initial set of databases or select a subset from it.
• The system shall provide **appropriate viewers** for the user to read documents in the document store.
• Every order shall be allocated a unique identifier (ORDER_ID) which the user shall be able to copy to the account's permanent storage area.

Requirements imprecision

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

Requirements completeness and consistency

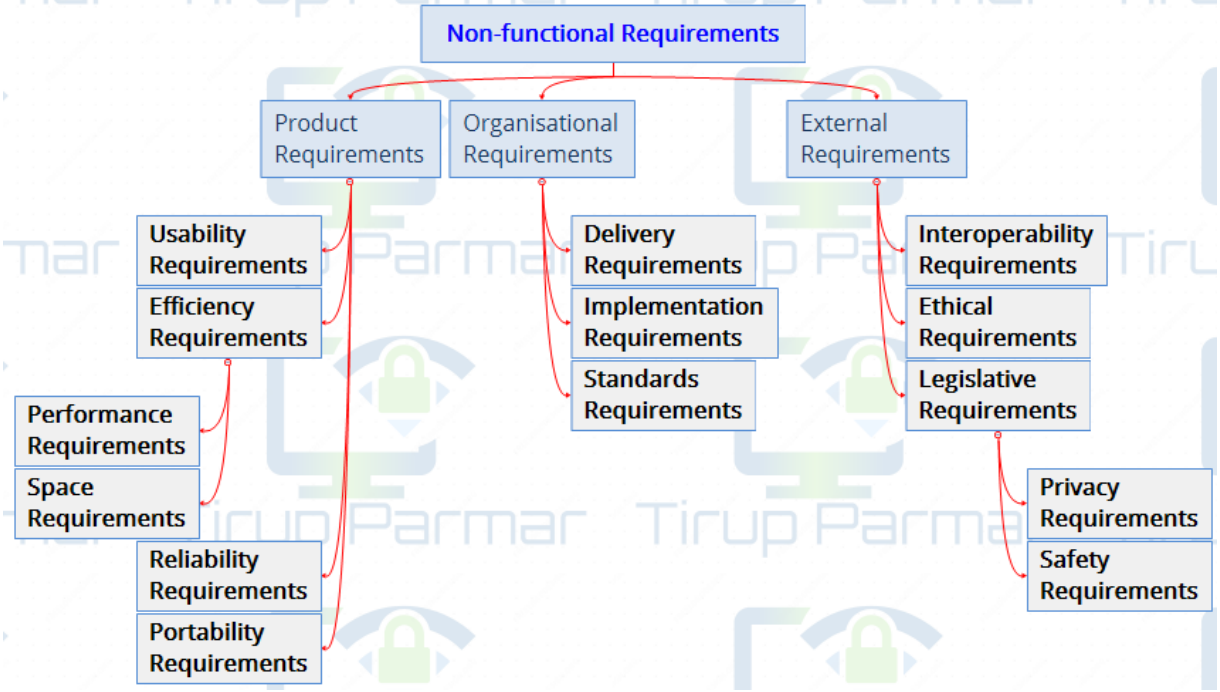
• In principle, requirements should be both complete and consistent.
• Completeness
  – All services required by the user should be defined.
• Consistent
  – Requirements should NOT have conflicts or contradictions in the descriptions.
• In practice, it is impossible to produce a complete and consistent requirements document.

Non-functional requirements

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

Non-functional requirement types

• **Product requirements**
  – Specify product behaviour. E.g. execution speed, memory required, reliability (failure rate), portability requirements, usability requirements, etc.
• **Organisational requirements**
  – Derived from customer and developer organisational policies and procedures. e.g. process standards used, implementation requirements, delivery requirements, etc.
• **External requirements**
  – Derived from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, ethical requirements etc.



LIBSYS Non-functional requirements

• Product requirement

8.1 The user interface for LIBSYS shall be implemented as simple HTML without frames or Java applets.

• Organisational requirement

9.3.2 The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95.

• External requirement

7.6.5 The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system.

Problems in non-functional requirements

• Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.

• Goal

– A general intention of the user such as ease of use, recovery from failure, etc.

• Verifiable non-functional requirement

– A statement using some measure that can be objectively tested.

• Goals are helpful to developers as they convey the intentions of the system users.

Examples

• A system goal

– The system should be easy to use by experienced controllers and should be organised in such a way that user errors are minimised.

• A verifiable non-functional requirement

– Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day.

Requirements Measures

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

Requirements interaction

• Conflicts between different non-functional requirements are common in complex systems.

• **Example :Spacecraft system**

– To minimise weight, the number of separate chips in the system should be minimised.

– To minimise power consumption, lower power chips should be used.

– However, using low power chips may mean that more chips have to be used. Which is the most critical requirement?

Domain requirements

• Derived from the application domain and describe system characteristics and features that reflect the domain.

• Domain requirements be new functional requirements, constraints on existing requirements or define specific computations.

• If domain requirements are not satisfied, the system may be unworkable.

Example of LIBSYS domain requirements

• There shall be a standard user interface to all databases which shall be based on the Z39.50 standard.

• Because of copyright restrictions, some documents must be deleted immediately on arrival. Depending on the user's requirements, these documents will either be printed locally on the system server for manually forwarding to the user or routed to a network printer.

Example : Train protection system

• Computation of deceleration to stop the train: The deceleration of the train shall be computed as:

$$D_{train} = D_{control} + D_{gradient}$$

where D is 9.81ms2 $D_{gradient}$ * compensated gradient/alpha and where the values of 9.81ms2 /alpha are known for different types of train.

• Problem: Since it is written in application domain language its difficult to understand by s/w engineers.

Domain requirements problems

• **Understandability**
  – Requirements are expressed in the language of the application domain;
  – This is often not understood by software engineers developing the system.

• **Implicitness**
  – Domain specialists understand the area so well that they do not think of making the domain requirements explicit.

User requirements

• Should describe functional and non-functional requirements in such a way that they are understandable by system users who don't have detailed technical knowledge.

• User requirements are defined using natural language, tables and diagrams as these can be understood by all users.

Problems with natural language(UR)

• **Lack of clarity**
  – Precision is difficult without making the document difficult to read.

• **Requirements confusion**
  – Functional and non-functional requirements tend to be mixed-up.

• **Requirements amalgamation**
  – Several different requirements may be expressed together.

A user requirement for accounting system in LIBSYS

4..5 LIBSYS shall provide a financial accounting system that maintains records of all payments made by users of the system. System managers may configure this system so that regular users may receive discounted rates.

A user requirement for Editor grid in LIBSYS

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

Requirement problems

• Database requirements includes both conceptual and detailed information
  – Describes the concept of a financial accounting system that is to be included in LIBSYS;
  – However, it also includes the detail that managers can configure this system - this is unnecessary at this level.
• Grid requirement mixes three different kinds of requirement
  – Conceptual functional requirement is the need for a grid. It presents rationale for this
  – Non-functional requirement of grid units (inches/cm)
  – Non-functional UI requirement grid switching(on/off)

Guidelines for writing User Requirements

• Invent a standard format and use it for all requirements.
• Use language in a consistent way. Use **shall** for mandatory requirements, **should** for desirable requirements.
• Use text highlighting(**Bold**, *Italic* or Colour) to identify key parts of the requirement.
• Avoid the use of computer jargon.

System requirements

• More detailed specifications of system functions, services and constraints than user requirements.
• They are intended to be a basis for designing the system.
• They may be incorporated into the system contract.
• Natural Language(NL) is often used to write system requirements specification as well as user requirements.

Inseparable : Requirements and design

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

Problems with NL specification(Sys Req.)

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

Notations for requirements specification

• These can be used as alternatives to NL specification

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

Structured Language Specifications

• The freedom of the requirements writer is limited by a predefined template for requirements.
• All requirements are written in a standard way.
• The terminology used in the description may be limited.
• The advantage is that the most of the expressiveness of natural language is maintained but a degree of uniformity is imposed on the specification.

Form-based Approach

• One or more standard forms/templates are designed to express the requirements.
It should include the following information.

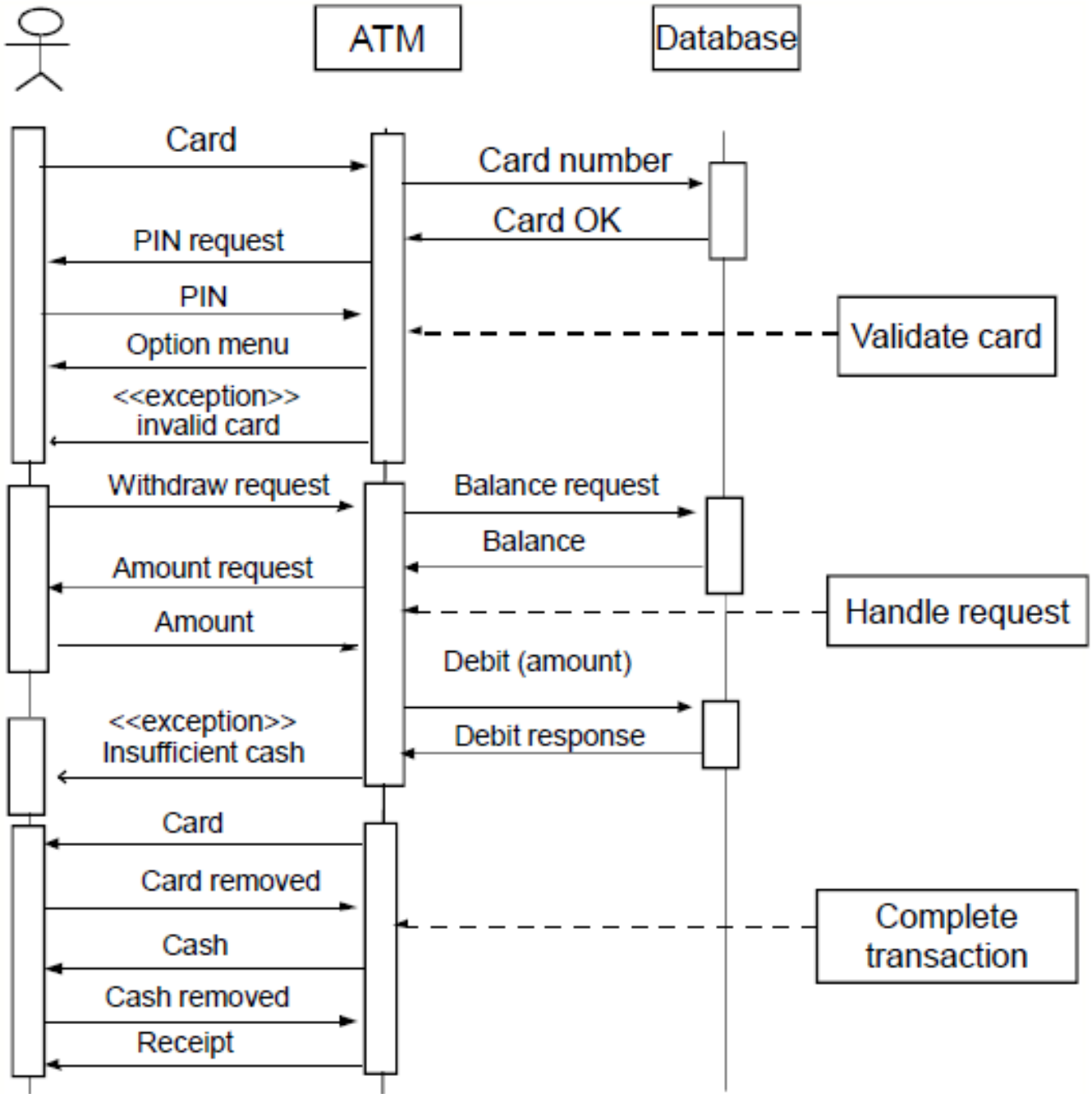
– Description of the function or entity.
– Description of inputs and where they come from.
– Description of outputs and where they go to.
– Indication of what other entities required.
– Description of action to be taken.
– Pre and post conditions what must be true before & after function call(if appropriate).
– Description the side effects (if any) of the function.

Graphical models

• Graphical models are most useful when you need to show how state changes or where you need to describe a sequence of actions.
• Example Sequence diagrams

• These show the sequence of events that take place during some user interaction with a system.
• Read them from top to bottom to see the order of the actions that take place.
• **Cash withdrawal from an ATM**
  – Validate card : By checking the card number and user's PIN
  – Handle request : user requests are handled. Query database for withdrawal
  – Complete transaction: return the card and deliver cash & receipt.

Sequence diagram of ATM withdrawal



Interface specification

• Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements.

• **Three types of interface may have to be defined**
  – Procedural interfaces : APIs;
  – Data structures : Passed from one subsystem to another
  – Data representations (ordering of bits) : for existing subsystem

• Formal notations(Program Design Language - PDL) are an effective technique for interface specification.

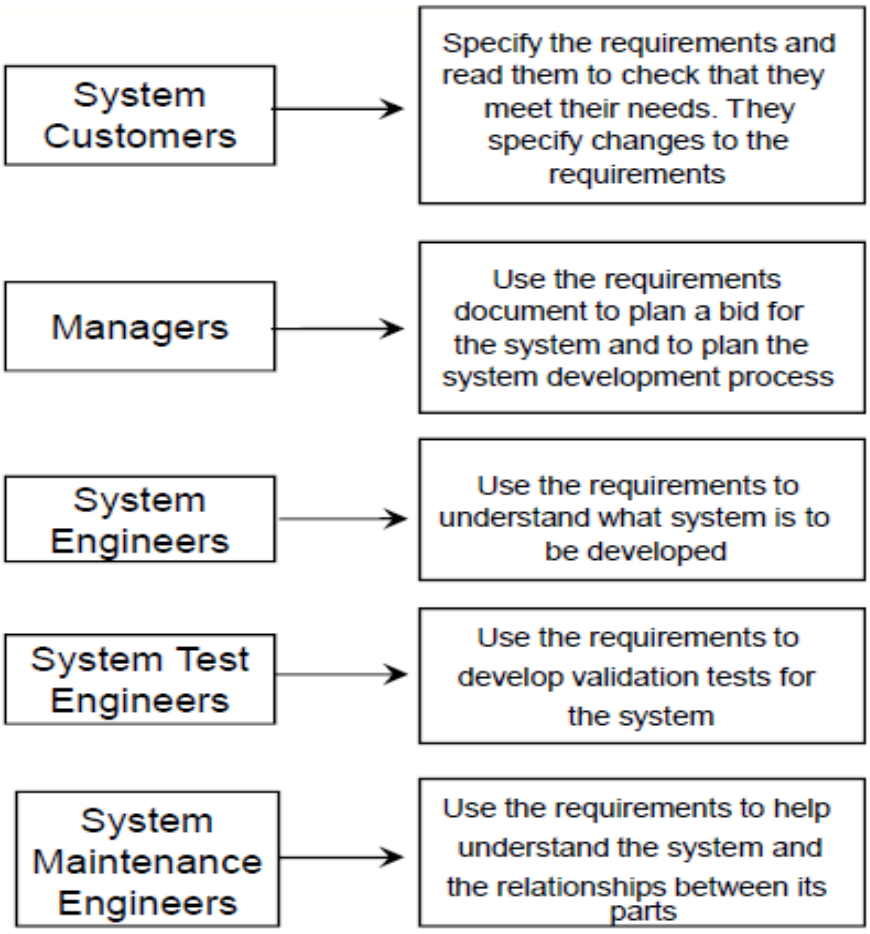
The Software Requirements Document

• The requirements document is the official statement of what is required of the system developers.

• Should include both a definition of user requirements and a specification of the system requirements.

• It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it

Users of a requirements document



Software Processes:

A software process (also knows as software methodology) is a set of related activities that leads to the production of the software. These activities may involve the development of the software from the scratch, or, modifying an existing system.

Any software process must include the following four activities:

1. **Software specification** (or requirements engineering): Define the main functionalities of the software and the constrains around them.
2. S**oftware design and implementation**: The software is to be designed and programmed.
3. **Software verification and validation**: The software must conforms to it's specification and meets the customer needs.
4. **Software evolution** (software maintenance): The software is being modified to meet customer and market requirements changes.

In practice, they include sub-activities such as requirements validation, architectural design, unit testing, …etc.

There are also **supporting activities** such as configuration and change management, quality assurance, project management, user experience.

Along with **other activities** aim to **improve** the above activities by introducing new techniques, tools, following the best practice, process standardization (so the diversity of software processes is reduced), etc.

When we talk about a process, we usually talk about the activities in it. However, a process also includes the process description, which includes:

1. **Products**: The outcomes of the an activity. For example, the outcome of architectural design maybe a model for the software architecture.
2. **Roles**: The responsibilities of the people involved in the process. For example, the project manager, programmer, etc.
3. **Pre and post conditions**: The conditions that must be true before and after an activity. For example, the pre condition of the architectural design is the requirements have been approved by the customer, while the post condition is the diagrams describing the architectural have been reviewed.

Software process is complex, it relies on making decisions. There's no ideal process and most organizations have developed their own software process.

For example, an organization works on critical systems has a very structured process, while with business systems, with rapidly changing requirements, a less formal, flexible process is likely to be more effective.

Plan-driven and agile processes

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

Process and Project

A process is a sequence of steps performed for a given purpose. As mentioned earlier, while developing (industrial strength) software, the purpose is to develop software to satisfy the needs of some users or clients, as shown in Figure 2.1. A software project is one instance of this problem, and the development process is what is used to achieve this purpose.

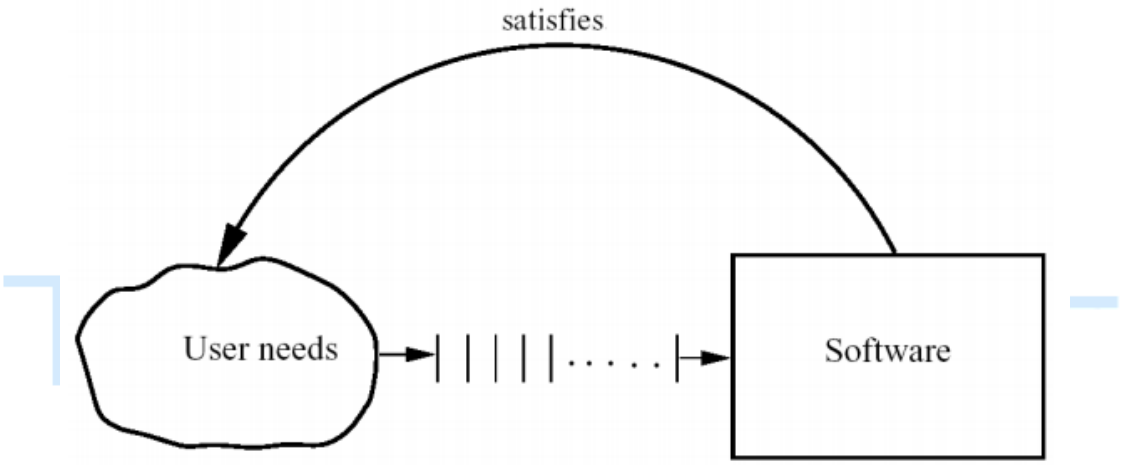


Figure 2.1: Basic problem

So, for a project its development process plays a key role—it is by following the process the desired end goal of delivering the software is achieved. However, as discussed earlier, it is not sufficient to just reach the final goal of having the desired software, but we want that the project be done at low cost and in low cycle time, and deliver high-quality software. The role of process increases due to these additional goals, and though many processes can achieve the basic goal of developing software in Figure 2.1, to achieve high Q&P we need some "optimum" process. It is this goal that makes designing a process a challenge.

We must distinguish process specification or description from the process itself. A process is a dynamic entity which captures the actions performed. Process specification, on the other hand, is a description of process which presumably can be followed in some project to achieve the goal for which the process is designed.

In a project, a process specification may be used as the process the project plans to follow. The actual process is what is actually done in the project. Note that the actual process can be different from the planned process, and ensuring that the specified process is being followed is a nontrivial problem. However, in this book, we will assume that the planned and actual processes are the same and will not distinguish between the two and will use the term process to refer to both.

A process model specifies a general process, which is “optimum” for a class of projects. That is, in the situations for which the model is applicable, using the process model as the project’s process will lead to the goal of developing software with high Q&P. A process model is essentially a compilation of best practices into a “recipe” for success in the project. In other words, a process is a means to reach the goals of high quality, low cost, and low cycle time, and a process model provides a process structure that is well suited for a class of projects.

A process is often specified at a high level as a sequence of stages. The sequence of steps for a stage is the process for that stage, and is often referred to as a subprocess of the process.

Component Software Processes

As defined above, a process is the sequence of steps executed to achieve a goal. Since many different goals may have to be satisfied while developing software, multiple processes are needed. Many of these do not concern software engineering, though they do impact software development. These could be considered nonsoftware process. Business processes, social processes, and training processes are all examples of processes that come under this. These processes also affect the software development activity but are beyond the purview of software engineering.

The processes that deal with the technical and management issues of software development are collectively called the software process. As a software project will have to engineer a solution and properly manage the project, there are clearly two major components in a software process—a development process and a project management process. The development process specifies all the engineering activities that need to be performed, whereas the management process specifies how to plan and control these activities so that cost, schedule, quality, and other objectives are met. Effective development and project management processes are the key to achieving the objectives of delivering the desired software satisfying the user needs, while ensuring high productivity and quality.

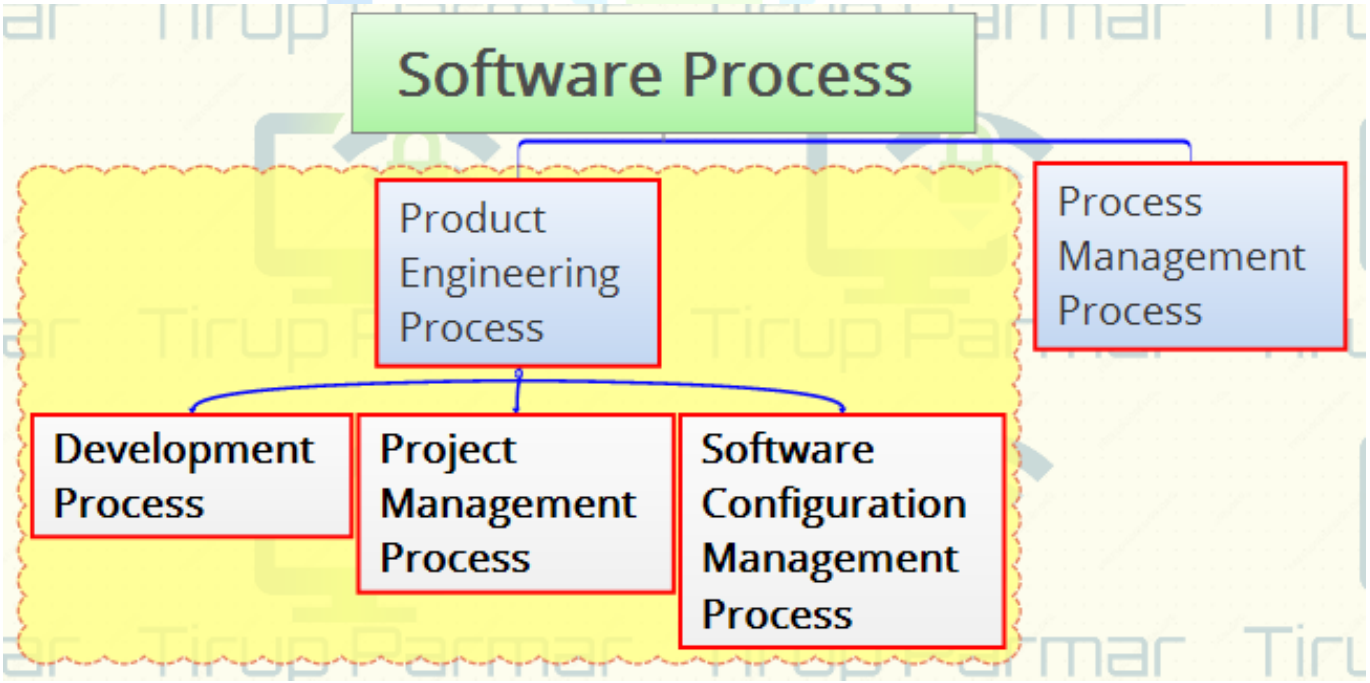
During the project many products are produced which are typically composed of many items (for example, the final source code may be composed of many source files). These items keep evolving as the project proceeds, creating many versions on the way. As development processes generally do not focus on evolution and changes, to handle them another process called software configuration control process is often used. The objective of this component process is to primarily deal with managing change, so that the integrity of the products is not violated despite changes.

These three constituent processes focus on the projects and the products and can be considered as comprising the product engineering processes, as their main objective is to produce the desired product. If the software process can be viewed as a static entity, then

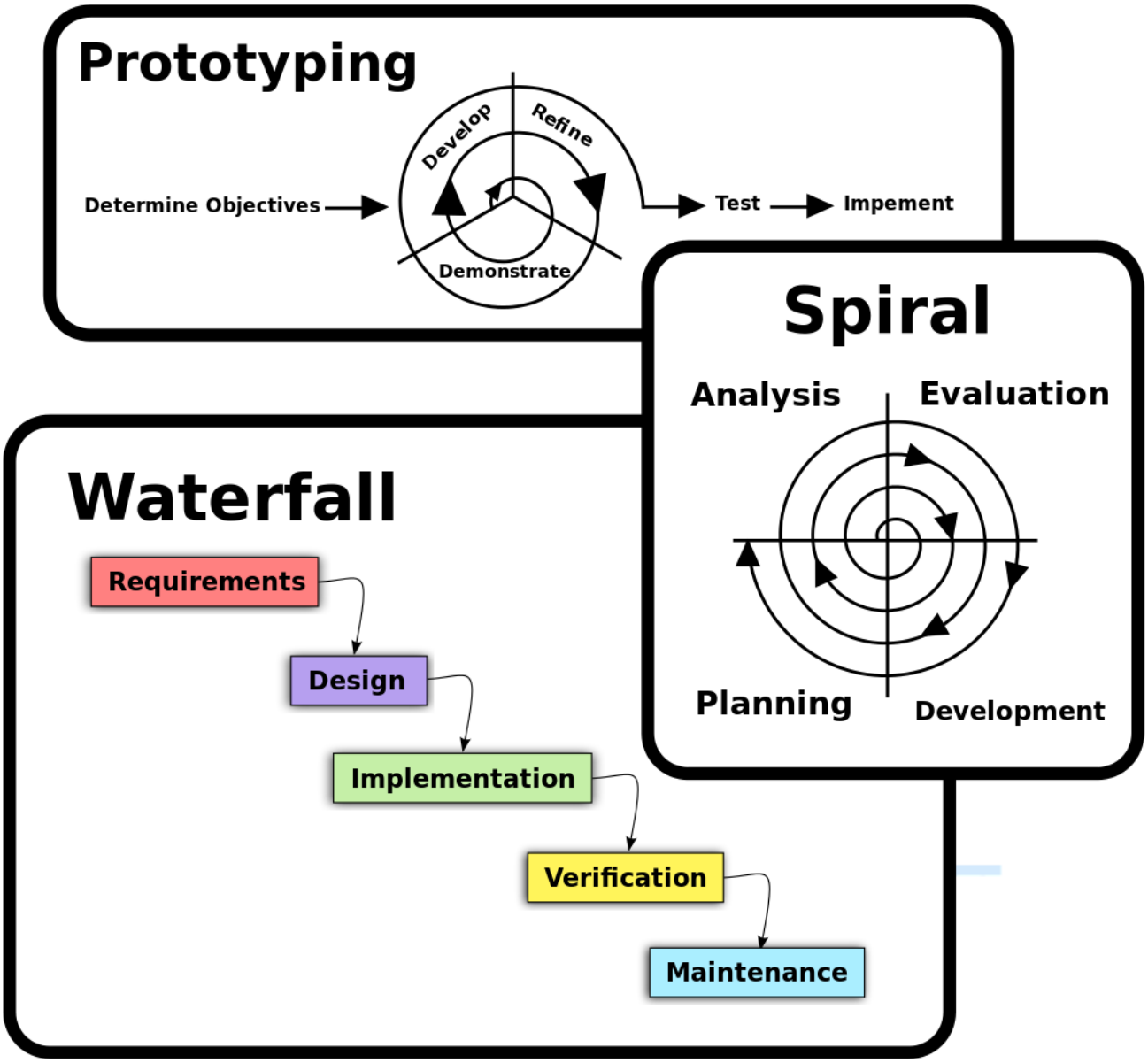
these three component processes will suffice. However, a software process itself is a dynamic entity, as it must change to adapt to our increased understanding about software development and availability of newer technologies and tools. Due to this, a process to manage the software process is needed.

The basic objective of the process management process is to improve the software process. By improvement, we mean that the capability of the process to produce quality goods at low cost is improved. For this, the current software process is studied, frequently by studying the projects that have been done using the process. The whole process of understanding the current process, analyzing its properties, determining how to improve, and then affecting the improvement is dealt with by the process management process.

The relationship between these major component processes is shown in Figure 2.2. These component processes are distinct not only in the type of activities performed in them, but typically also in the people who perform the activities specified by the process. In a typical project, development activities are performed by programmers, designers, testers, etc.; the project management process activities are performed by the project management; configuration control process activities are performed by a group generally called the configuration controller; and the process management process activities are performed by the software engineering process group (SEPG).



Software Development Process Models



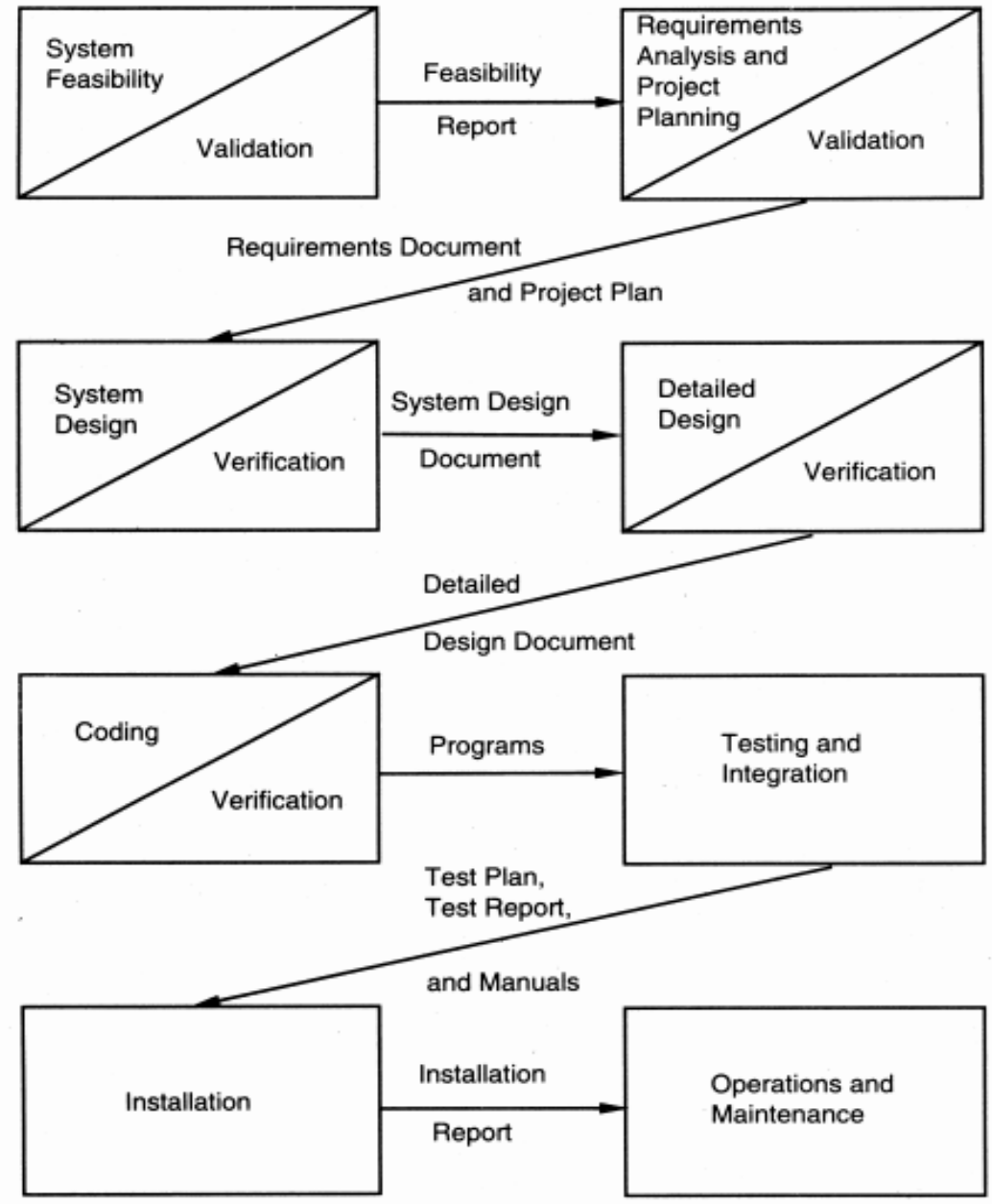
For the software development process, the goal is to produce a high-quality software product. It therefore focuses on activities directly related to production of the software, for example, design, coding, and testing. As the development process specifies the major development and quality control activities that need to be performed in the project, it forms the core of the software process. The management process is often decided based on the development process.

A project's development process defines the tasks the project should perform, and the order in which they should be done. A process limits the degrees of freedom for a project by specifying what types of activities must be undertaken and in what order, such that the "shortest" (or the most efficient) path is obtained from the user needs to the software satisfying these needs. The process drives a project and heavily influences the outcome.

As discussed earlier, a process model specifies a general process, usually as a set of stages in which a project should be divided, the order in which the stages should be executed, and any other constraints and conditions on the execution of stages. The basic premise behind a process model is that, in the situations for which the model is applicable, using the process model as the project's process will lead to low cost, high quality, reduced cycle time, or provide other benefits. In other words, the process model provides generic guidelines for developing a suitable process for a project.

Due to the importance of the development process, various models have been proposed. In this section we will discuss some of the major models.

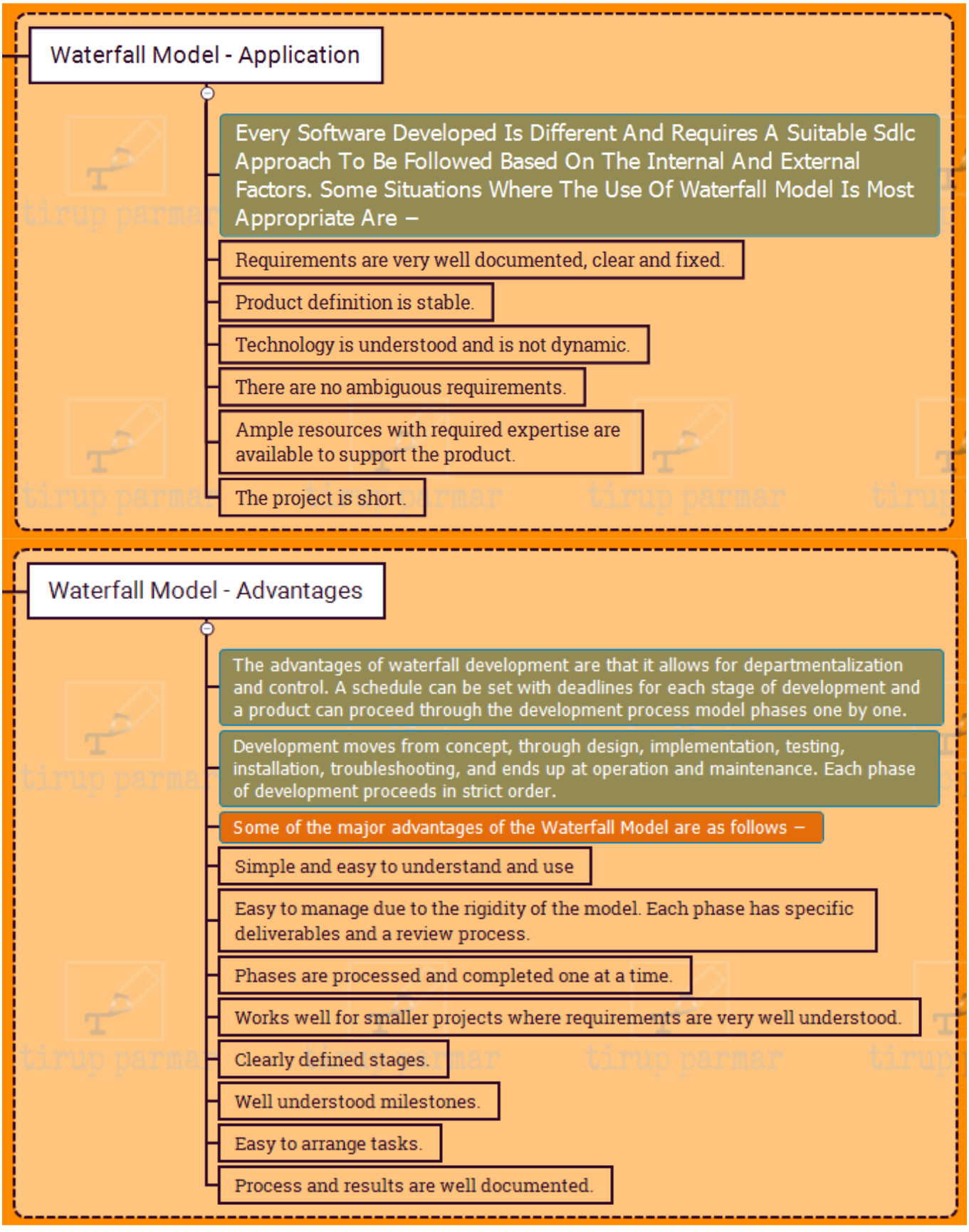
Q) Explain waterfall model with its advantages and disadvantages.

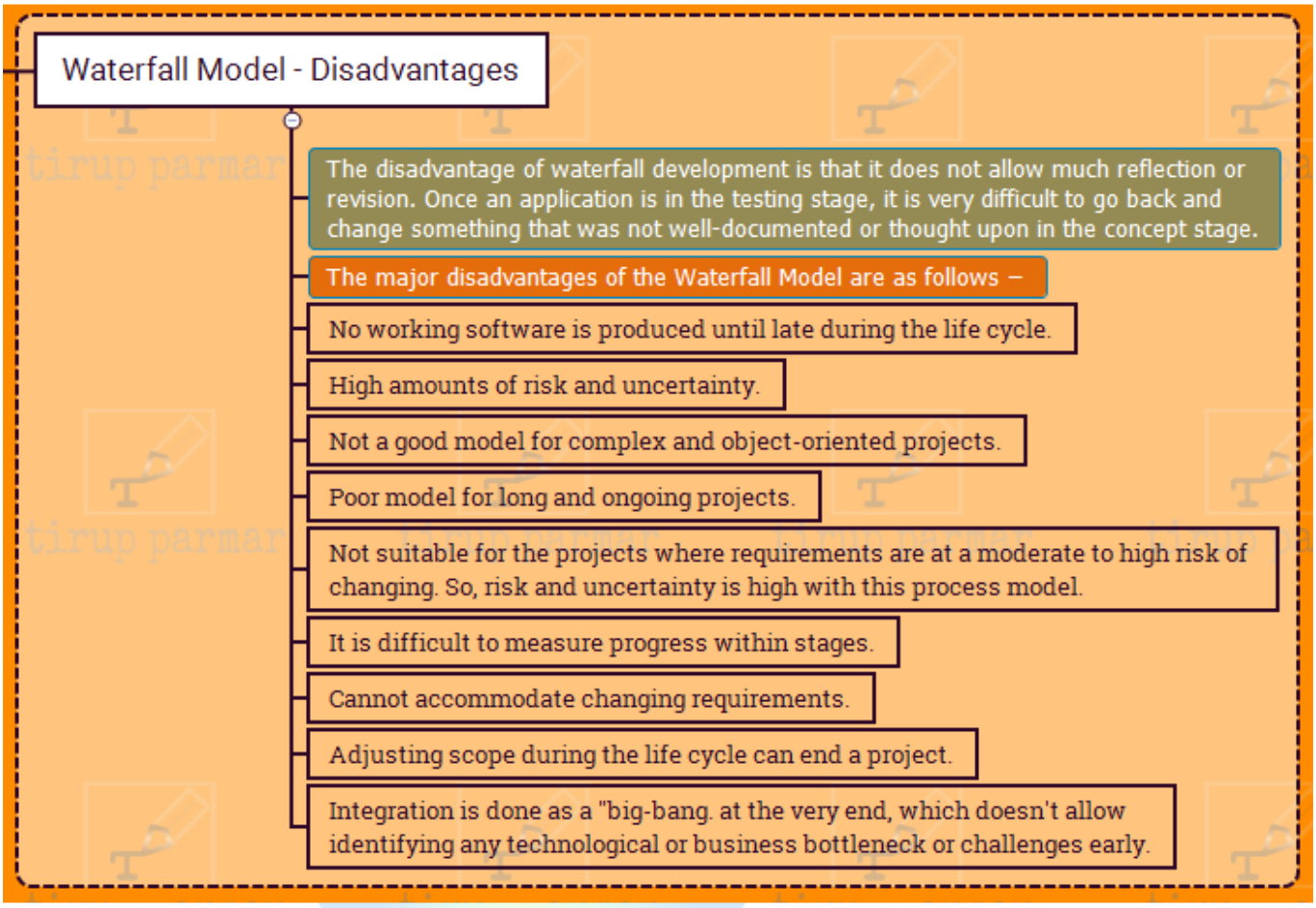


Waterfall approach was first SDLC Model to be used widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate phases. In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

The sequential phases in Waterfall model are –

- Requirement Gathering and analysis – All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
- System Design – The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
- Implementation – With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.
- Integration and Testing – All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
- Deployment of system – Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.
- Maintenance – There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.


Waterfall Model - Application
Every Software Developed Is Different And Requires A Suitable Sdlc Approach To Be Followed Based On The Internal And External Factors. Some Situations Where The Use Of Waterfall Model Is Most Appropriate Are –
Requirements are very well documented, clear and fixed.
Product definition is stable.
Technology is understood and is not dynamic.
There are no ambiguous requirements.
Ample resources with required expertise are available to support the product.
The project is short.
Waterfall Model - Advantages
The advantages of waterfall development are that it allows for departmentalization and control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process model phases one by one.
Development moves from concept, through design, implementation, testing, installation, troubleshooting, and ends up at operation and maintenance. Each phase of development proceeds in strict order.
Some of the major advantages of the Waterfall Model are as follows –
Simple and easy to understand and use
Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
Phases are processed and completed one at a time.
Works well for smaller projects where requirements are very well understood.
Clearly defined stages.
Well understood milestones.
Easy to arrange tasks.
Process and results are well documented.



PRTOTYPING MODEL

Prototype

A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions.

The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations.

A prototype usually turns out to be a very crude version of the actual system.

So, a prototype is useful when a customer or developer is not sure of the requirements, or of algorithms, efficiency, business rules, response time, etc.

In prototyping, the client is involved throughout the development process, which increases the likelihood of client acceptance of the final implementation.

While some prototypes are developed with the expectation that they will be discarded, it is possible in some cases to evolve from prototype to working system.

A software prototype can be used:

[1] In the **requirements engineering**, a prototype can help with the elicitation and validation of system requirements.
It allows the users to experiment with the system, and so, refine the requirements. They may get new ideas for requirements, and find areas of strength and weakness in the software.
Furthermore, as the prototype is developed, it may reveal errors and in the requirements. The specification maybe then modified to reflect the changes.
[2] In the **system design**, a prototype can help to carry out deign experiments to check the feasibility of a proposed design.
For example, a database design may be prototype-d and tested to check it supports efficient data access for the most common user queries.

Other Needs for a prototype in software development
There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs:

- how the screens might look like
- how the user interface would behave
- how the system would produce outputs

Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.
A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with the product development. Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.
A prototype of the actual product is preferred in situations such as:
• User requirements are not complete
• Technical issues are not clear

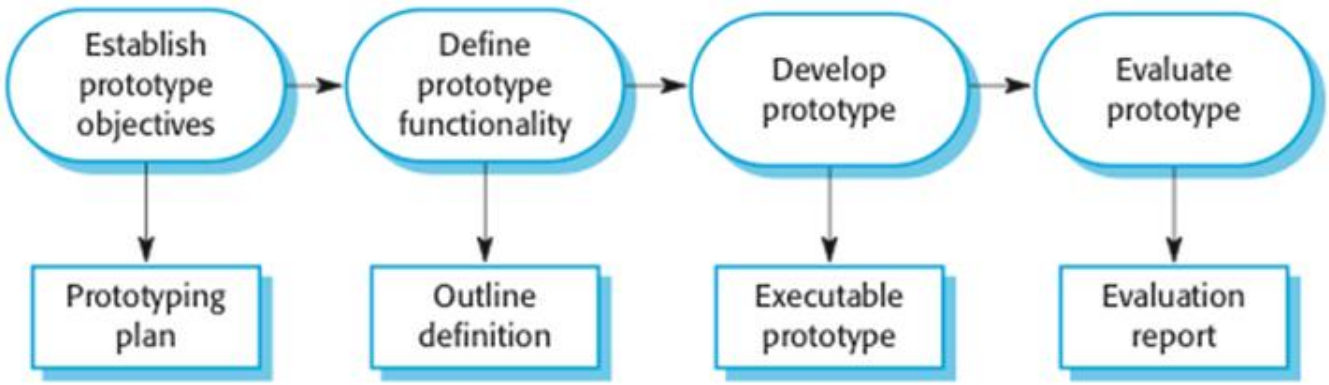


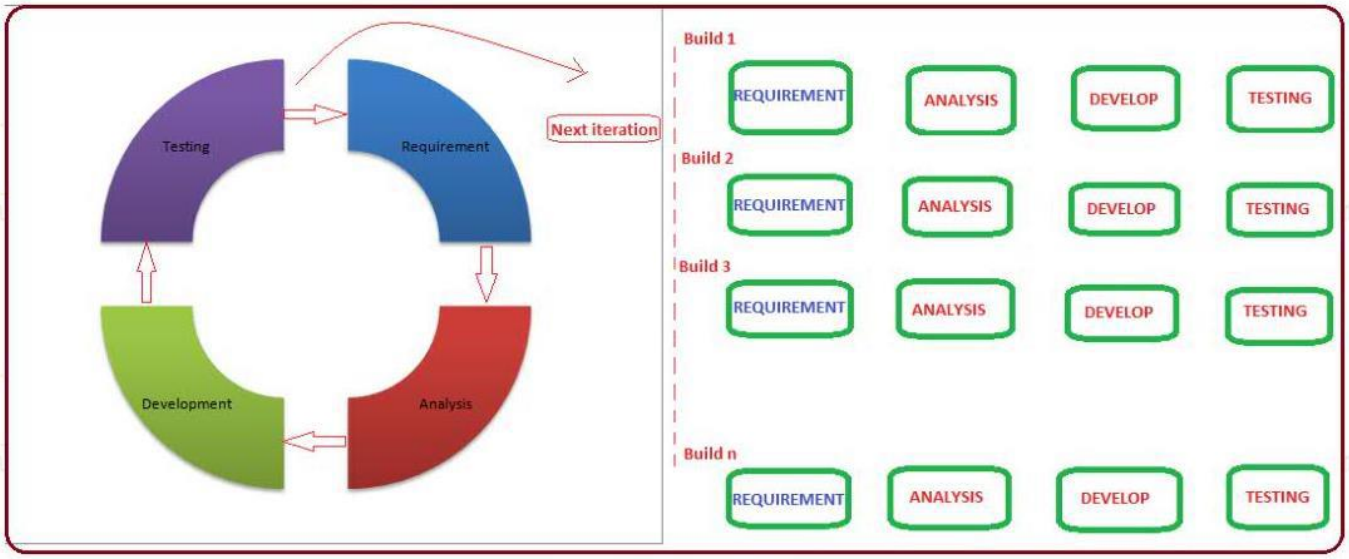
Fig. The process of prototype development

The phases of a prototype are:

1. **Establish objectives**: The objectives of the prototype should be made explicit from the start of the process. Is it to validate system requirements, or demonstrate feasibility, etc.
2. **Define prototype functionality**: Decide what are the inputs and the expected output from a prototype. To reduce the prototyping costs and accelerate the delivery schedule, you may ignore some functionality, such as response time and memory utilization unless they are relevant to the objective of the prototype.
3. **Develop the prototype**: The initial prototype is developed that includes only user interfaces.
4. **Evaluate the prototype**: Once the users are trained to use the prototype, they then discover requirements errors. Using the feedback both the specifications and the prototype can be improved. If changes are introduced, then a repeat of steps 3 and 4 may be needed.

Prototyping is not a standalone, complete development methodology, but rather an approach to be used in the context of a full methodology (such as incremental, spiral, etc).

Iterative Model



Iterative model

- Requirements may change at a previous stage of development and may be taken care of in the next stage
- Feedback loop at every stage in development
- Regression testing increasingly important on all iterations one after another
- Testing plan to allow more testing at each subsequent delivery phase
- More practical than the waterfall model
- Limitation:
  - o They are many cycles of waterfall model
  - o Fixed price projects have problem of estimation
  - o Product architecture and design becomes fragile due to many iteration

Common approach for iterative development is to do the requirements and the architecture design in a standard waterfall or prototyping approach, but deliver the software iteratively. That is, the building of the system, which is the most time and effort-consuming task, is done iteratively, though most of the requirements are specified upfront. We can view this approach as having one iteration delivering the requirements and the architecture plan,

and then further iterations delivering the software in increments. At the start of each delivery iteration, which requirements will be implemented in this release are decided, and then the design is enhanced and code developed to implement the requirements. The iteration ends with delivery of a working software system providing some value to the end user. Selecting of requirements for an iteration is done primarily based on the value the requirement provides to the end users and how critical they are for supporting other requirements. This approach is shown in Figure

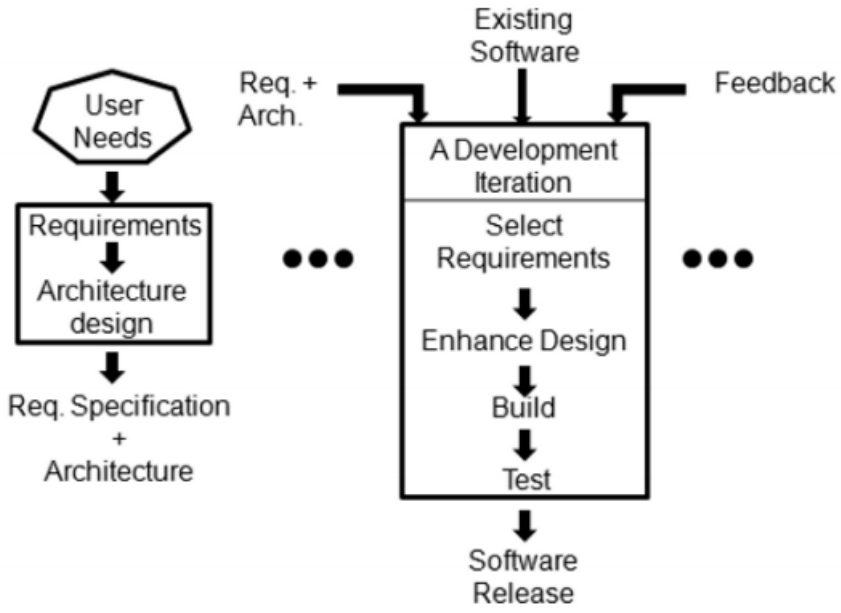


Figure 2.6: Iterative delivery approach

The advantage of this approach is that as the requirements are mostly known upfront, an overall view of the system is available and a proper architecture can be designed which can remain relatively stable. With this, hopefully rework in development iterations will diminish. At the same time, the value to the end customer is delivered iteratively so it does not have the all-or-nothing risk. Also, since the delivery is being done incrementally, and planning and execution of each iteration is done separately, feedback from an iteration can be incorporated in the next iteration. Even new requirements that may get uncovered can also be incorporated. Hence, this model of iterative development also provides some of the benefits of the model discussed above.

The iterative approach is becoming extremely popular, despite some diffi- culties in using it in this context. There are a few key reasons for its increasing popularity. First and foremost, in today's world clients do not want to invest too much without seeing returns. In the current business scenario, it is preferable to see returns continuously of the investment made. The iterative model permits this—after each iteration some working software is delivered, and the risk to the client is therefore limited. Second, as businesses are changing rapidly today, they never really know the "complete" requirements for the software, and there is a need to constantly add new capabilities to the software to adapt the business to changing situations. Iterative process allows this. Third, each iteration provides a working system for feedback, which helps in developing stable requirements for the next iteration. Below we will describe some other process models, all of them using some iterative approach.

The Rational Unified Process

The Rational Unified Process (RUP) methodology is an example of a modern software process model that has been derived from the UML and the associated Unified Software Development Process. The RUP recognises that conventional process models present a single view of the process. In contrast, the RUP is described from three perspectives

1. A dynamic perspective that shows the phases of the model over time.
2. A static perspective that shows the process activities.
3. A practice perspective that suggests good practices to be used during the process.

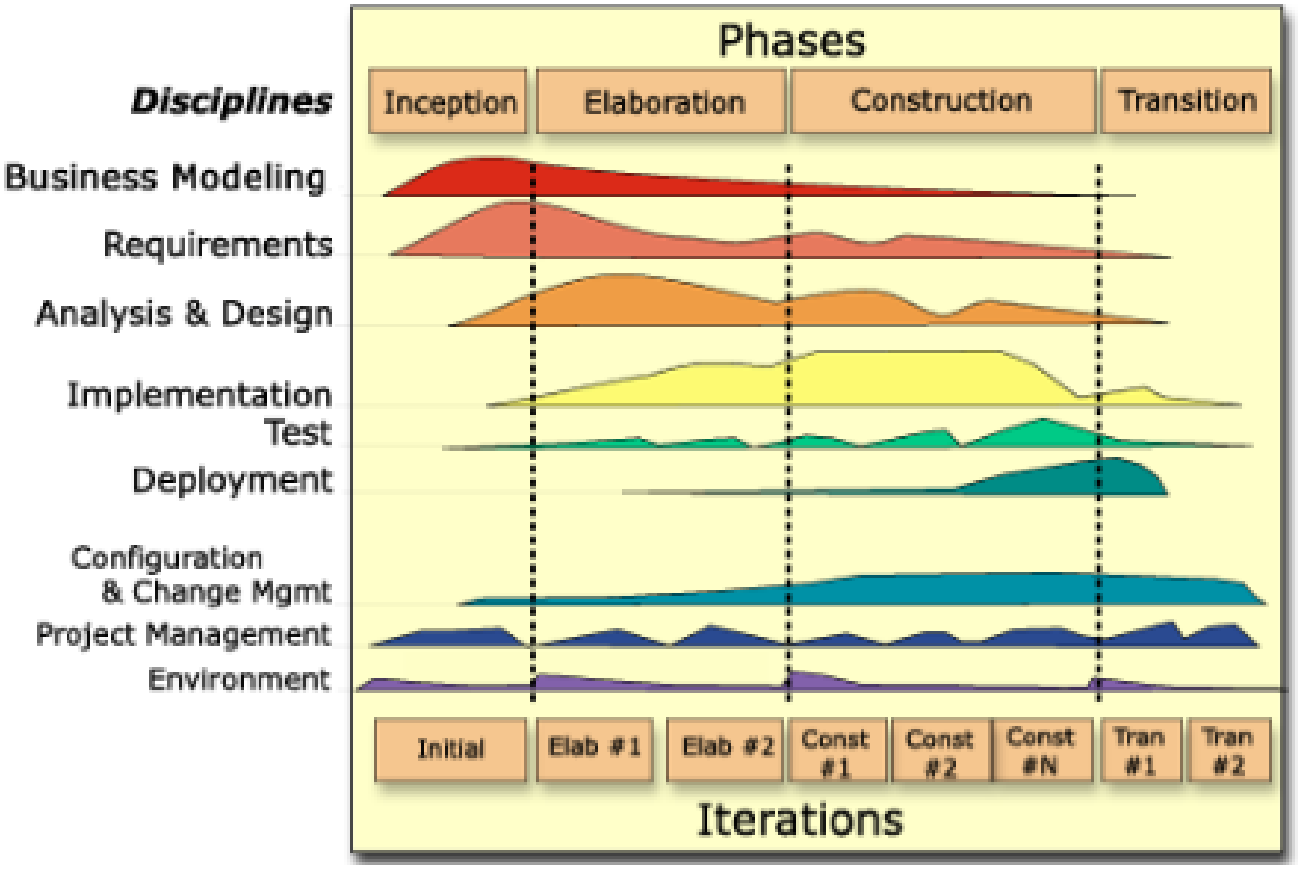


Figure 3.6. Phases of Rational Unified Process.

The RUP (Figure 3.6.) identifies four discrete development phases in the software process that are not equated with process activities. The phases in the RUP are more closely related to business rather than technical concerns. These phases in the RUP are:

1. Inception. The goal of the inception phase is to establish a business case for the system, identify all external entities, i.e. people and systems that will interact with the system

and define these interactions. Then this information is used to assess the contribution that the system makes to the business.
2. Elaboration. The goals of the elaboration phase are to develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan and identify key project risks.
3. Construction. The construction phase is essentially concerned with system design, programming and testing.
4. Transition. The final phase of the RUP is concerned with moving the system from the development community to the user community and making it work in a real environment.

Iteration within the RUP is supported in two ways. Each phase may be enacted in an iterative way with the results developed incrementally. In addition, the whole set of phases may also be enacted incrementally.

The static view of the RUP focuses on the activities that take place during the development process. These are called workflows in the RUP description. There are six core process workflows identified in the process and three core supporting workflows. The RUP has been designed in conjunction with the UML so the workflow description is oriented around associated UML models. The core engineering and support workflows are the followings:

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

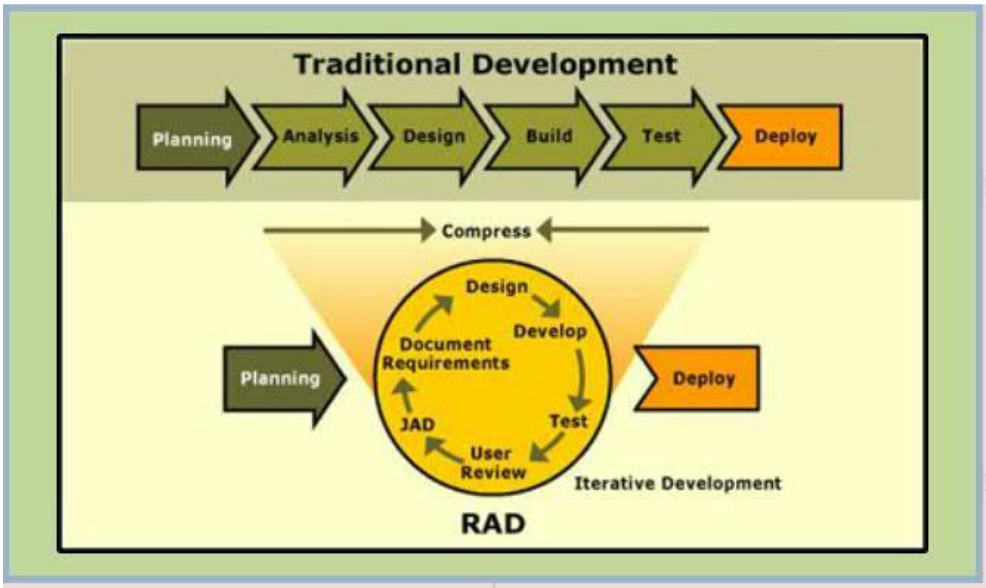
The advantage in presenting dynamic and static views is that phases of the development process are not associated with specific workflows. In principle at least, all of the RUP workflows may be active at all stages of the process. Of course, most effort will probably be spent on workflows such as business modelling and requirements at the early phases of the process and in testing and deployment in the later phases.

The practice perspective on the RUP describes good software engineering practices that are recommended for use in systems development. Six fundamental best practices are recommended:

1. Develop software iteratively. Plan increments of the system based on customer priorities and develop and deliver the highest priority system features early in the development process.
2. Manage requirements. Explicitly document the customer's requirements and keep track of changes to these requirements. Analyze the impact of changes on the system before accepting them.
3. Use component-based architectures. Structure the system architecture into components.
4. Visually model software. Use graphical UML models to present static and dynamic views of the software.
5. Verify software quality. Ensure that the software meets the organisational quality standards.
6. Control changes to software. Manage changes to the software using a change management system and configuration management procedures and tools.

The RUP is not a suitable process for all types of development but it does represent a new generation of generic processes. The most important innovations are the separation of phases and workflows, and the recognition that deploying software in a user's environment is part of the process. Phases are dynamic and have goals. Workflows are static and are technical activities that are not associated with a single phase but may be used throughout the development to achieve the goals of each phase.

The RAD(Rapid Application Development) Model



- ☐Usable software created at fast speed with the involvement of the user for every development stage (functionality)
- ☐Miniature form of spiral model where requirements are added in small chunks and refined with every iteration
- ☐Components or functions are time-boxed, delivered and then assembled into working prototype
- ☐Early validation of technical risks and rapid response to changing customer requirements
- ☐Limitation:
  - o Refactoring is the main constraint
  - o Involves huge cycles of retesting and regression testing
  - o Efforts of integration are huge
  - o Risk of never achieving closure
  - o Hard to use with legacy systems
  - o Requires a system that can be modularized

Timeboxing Model

To speed up development, parallelism between the different iterations can be employed. That is, a new iteration commences before the system produced by the current iteration is released, and hence development of a new release happens in parallel with the development of the current release. By starting an iteration before the previous iteration has completed, it is possible to reduce the average delivery time for iterations. However, to support parallel execution, each iteration has to be structured properly and teams have to be organized suitably. The timeboxing model proposes an approach for these

In the timeboxing model, the basic unit of development is a time box, which is of fixed duration. Since the duration is fixed, a key factor in selecting the requirements or features to be built in a time box is what can be fit into the time box. This is in contrast to regular iterative approaches where the functionality is selected and then the time to deliver is determined. Timeboxing changes the perspective of development and makes the schedule a nonnegotiable and a high-priority commitment.

Each time box is divided into a sequence of stages, like in the waterfall model. Each stage performs some clearly defined task for the iteration and produces a clearly defined output. The model also requires that the duration of each stage, that is, the time it takes to complete the task of that stage, is approximately the same. Furthermore, the model requires that there be a dedicated team for each stage. That is, the team for a stage performs only tasks of that stage—tasks for other stages are performed by their respective teams. This is quite different from other iterative models where the implicit assumption is that the same team performs all the different tasks of the project or the iteration.

Having time-boxed iterations with stages of equal duration and having dedicated teams renders itself to pipelining of different iterations. (Pipelining is a concept from hardware in which different instructions are executed in parallel, with the execution of a new instruction starting once the first stage of the previous instruction is finished.)

To illustrate the use of this model, consider a time box consisting of three stages: requirement specification, build, and deployment. The requirement stage is executed by its team of analysts and ends with a prioritized list of requirements to be built in this iteration along with a high-level design. The build team develops the code for implementing the requirements, and performs the testing. The tested code is then handed over to the deployment team, which performs predeployment tests, and then installs the system for production use. These three stages are such that they can be done in approximately equal time in an iteration.

With a time box of three stages, the project proceeds as follows. When the requirements team has finished requirements for timebox-1, the requirements are given to the build team for building the software. The requirements team then goes on and starts preparing the requirements for timebox-2. When the build for timebox-1 is completed, the code is handed over to the deployment team, and the build team moves on to build code for requirements for timebox- 2, and the requirements team moves on to doing requirements for timebox-3. This pipelined execution of the timeboxing process is shown in Figure

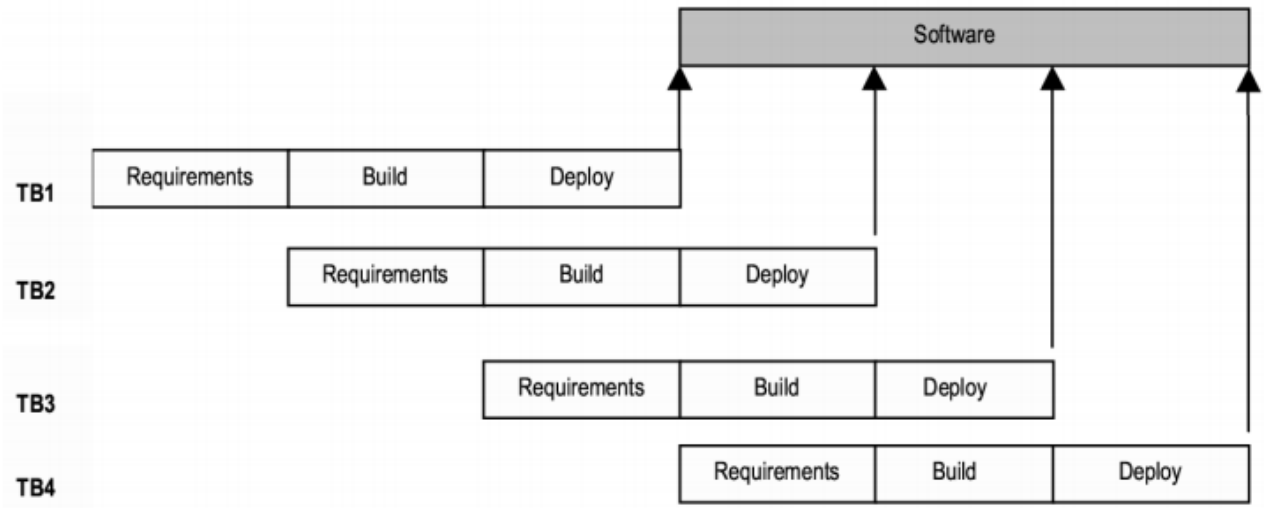


Figure : Executing the timeboxing process model.

With a three-stage time box, at most three iterations can be concurrently in progress. If the time box is of size T days, then the first software delivery will occur after T days. The subsequent deliveries, however, will take place after every T/3 days. For example, if the time box duration T is 9 weeks (and each stage duration is 3 weeks), the first delivery is made 9 weeks after the start of the project. The second delivery is made after 12 weeks, the third after 15 weeks, and so on. Contrast this with a linear execution of iterations, in which the first delivery will be made after 9 weeks, the second after 18 weeks, the third after 27 weeks, and so on.

There are three teams working on the project—the requirements team, the build team, and the deployment team. The teamwise activity for the 3-stage pipeline discussed above is shown in Figure

It should be clear that the duration of each iteration has not been reduced.

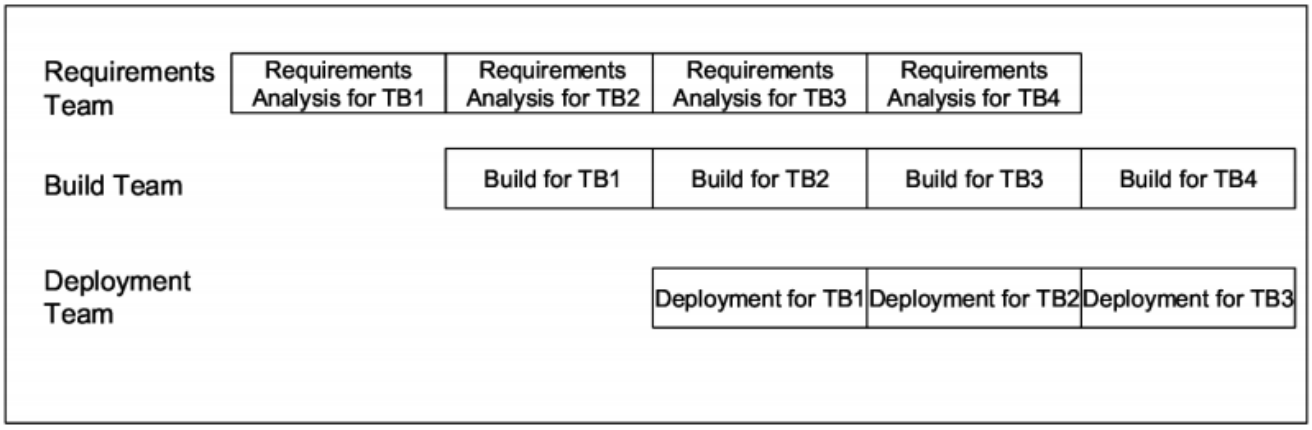


Figure : Tasks of different teams.

The total work done in a time box and the effort spent in it also remains the same—the same amount of software is delivered at the end of each iteration as the time box undergoes the same stages. If the effort and time spent in each iteration also remains the same, then what

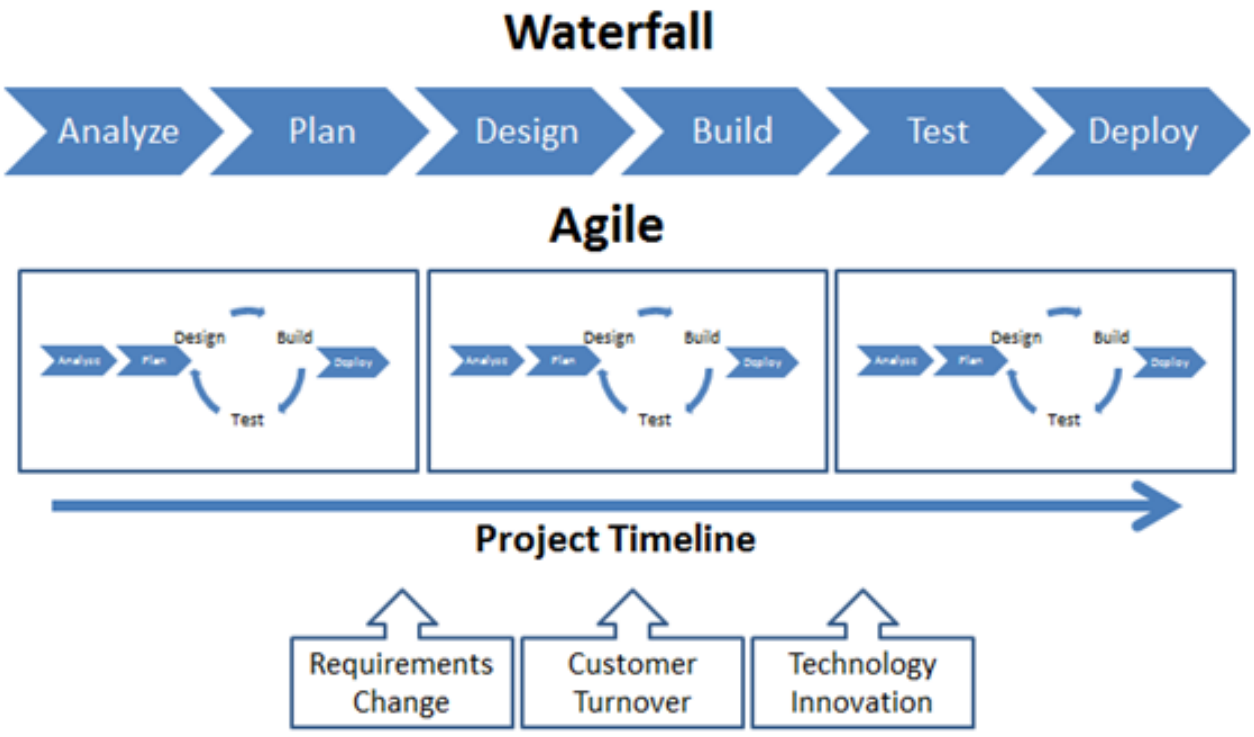
is the cost of reducing the delivery time? The real cost of this reduced time is in the resources used in this model. With timeboxing, there are dedicated teams for different stages and the total team size for the project is the sum of teams of different stages. This is the main difference from the situation where there is a single team which performs all the stages and the entire team works on the same iteration.

Hence, the timeboxing provides an approach for utilizing additional manpower to reduce the delivery time. It is well known that with standard methods of executing projects, we cannot compress the cycle time of a project substantially by adding more manpower. However, through the timeboxing model, we can use more manpower in a manner such that by parallel execution of different stages we are able to deliver software quicker. In other words, it provides a way of shortening delivery times through the use of additional manpower.

Timeboxing is well suited for projects that require a large number of features to be developed in a short time around a stable architecture using stable technologies. These features should be such that there is some flexibility in grouping them for building a meaningful system in an iteration that provides value to the users. The main cost of this model is the increased complexity of project management (and managing the products being developed) as multiple developments are concurrently active. Also, the impact of unusual situations in an iteration can be quite disruptive

Agile

Agility is flexibility, it is a state of dynamic, adapted to the specific circumstances.
The agile methods refers to a group of software development models based on the incremental and iterative approach, in which the increments are small and typically, new releases of the system are created and made available to customers every few weeks.



Rapid software development

✧ Rapid development and delivery is now often the most important requirement for software systems
- Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
- Software has to evolve quickly to reflect changing business needs.

✧ Rapid software development
- Specification, design and implementation are inter-leaved
- System is developed as a series of versions with stakeholders involved in version evaluation
- User interfaces are often developed using an IDE and graphical toolset.

Agile methods

✧ Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
- Focus on the code rather than the design
- Are based on an iterative approach to software development
- Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.

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

Agile manifesto

✧*We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
- *Individuals and interactions over processes and tools Working software over comprehensive documentation Customer collaboration over contract negotiation Responding to change over following a plan*

✧ *That is, whilethere is value in the items on the right, we value the items on the left more.*

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

The principles of agile methods

They involve customers in the development process to propose requirements changes. They minimize documentation by using informal communications rather than formal meetings with written documents.

They are best suited for application where the requirements change rapidly during the development process.

There are a number of different agile methods available such as: Scrum, Crystal, Agile Modeling (AM), Extreme Programming (XP), etc.

Agile method applicability

✧ Product development where a software company is developing a small or medium-sized product for sale.

✧ Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.

✧ Because of their focus on small, tightly-integrated teams, there are problems in scaling agile methods to large systems.

Problems with agile methods

✧ It can be difficult to keep the interest of customers who are involved in the process.

✧ Team members may be unsuited to the intense involvement that characterises agile methods.

✧ Prioritising changes can be difficult where there are multiple stakeholders.

- ✧ Maintaining simplicity requires extra work.
- ✧ Contracts may be a problem as with other approaches to iterative development.

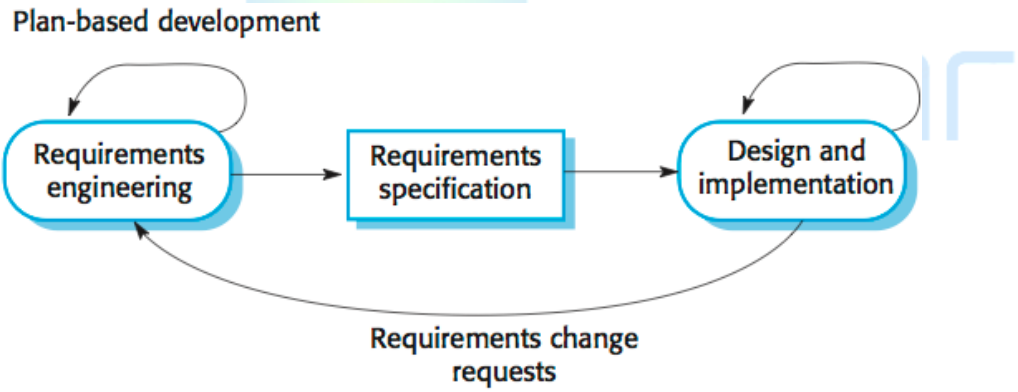
Agile methods and software maintenance

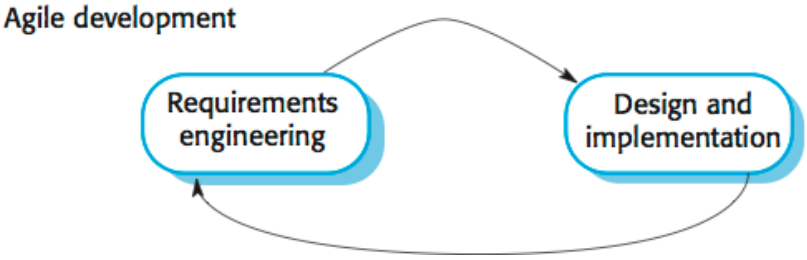
✧ Most organizations spend more on maintaining existing software than they do on new software development. So, if agile methods are to be successful, they have to support maintenance as well as original development.

- ✧ Two key issues:
  - Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
  - Can agile methods be used effectively for evolving a system in response to customer change requests?
- ✧ Problems may arise if original development team cannot be maintained.

Plan-driven and agile development

- ✧ Plan-driven development
  - A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
  - Not necessarily waterfall model – plan-driven, incremental development is possible
  - Iteration occurs within activities.
- ✧ Agile development
  - Specification, design, implementation and testing are interleaved and the outputs from the development process are decided through a process of negotiation during the software development process.





Technical, human, organizational issues

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

- Is it important to have a very detailed specification and design before moving to implementation? If so, you probably need to use a plan-driven approach.
- Is an incremental delivery strategy, where you deliver the software to customers and get rapid feedback from them, realistic? If so, consider using agile methods.
- How large is the system that is being developed? Agile methods are most effective when the system can be developed with a small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams so a plan-driven approach may have to be used.
- What type of system is being developed?
  - Plan-driven approaches may be required for systems that require a lot of analysis before implementation (e.g. real-time system with complex timing requirements).
- What is the expected system lifetime?
  - Long-lifetime systems may require more design documentation to communicate the original intentions of the system developers to the support team.
- What technologies are available to support system development?
  - Agile methods rely on good tools to keep track of an evolving design
- How is the development team organized?
  - If the development team is distributed or if part of the development is being outsourced, then you may need to develop design documents to communicate across the development teams.
- Are there cultural or organizational issues that may affect the system development?
  - Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering.
- How good are the designers and programmers in the development team?
  - It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code
- Is the system subject to external regulation?
  - If a system has to be approved by an external regulator (e.g. the FAA approve software that is critical to the operation of an aircraft) then you will probably be required to produce detailed documentation as part of the system safety case.

Extreme programming

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

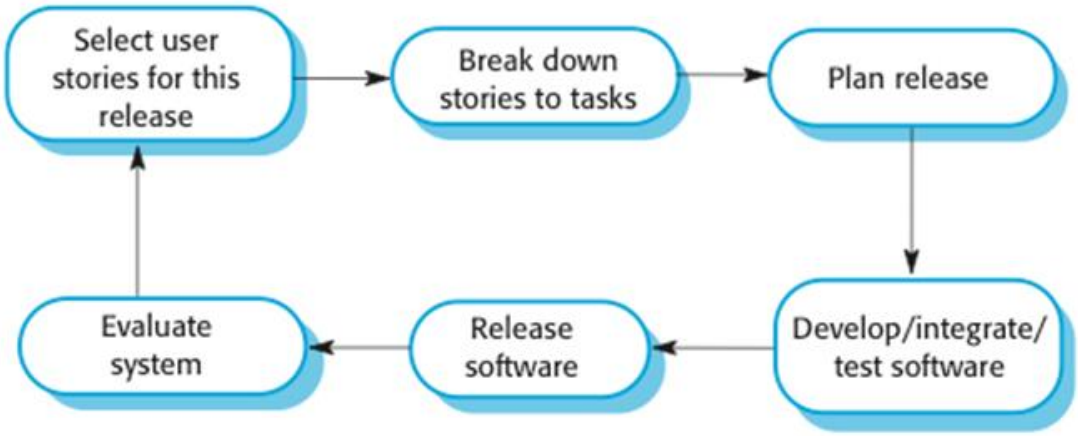
✧ Extreme Programming (XP) takes an ‘extreme’ approach to iterative development.

- New versions may be built several times per day;
- Increments are delivered to customers every 2 weeks;

- All tests must be run for every build and the build is only accepted if tests run successfully.

XP and agile principles

✧ Incremental development is supported through small, frequent system releases.
✧ Customer involvement means full-time customer engagement with the team.
✧ People not process through pair programming, collective ownership and a process that avoids long working hours.
✧ Change supported through regular system releases.
✧ Maintaining simplicity through constant refactoring of code.



Extreme programming practices (a)

Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

Extreme programming practices (b)

Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

Requirements scenarios

✧ In XP, a customer or user is part of the XP team and is responsible for making decisions on requirements.
✧ User requirements are expressed as scenarios or user stories.
✧ These are written on cards and the development team break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
✧ The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.

A 'prescribing medication' story

The record of the patient must be open for input. Click on the medication field and select either «current medication», «new medication» or «formulary».
If you select «current medication», you will be asked to check the dose: if you wish to change the dose, enter the new dose then confirm the prescription.
If you choose «new medication», the system assumes that you know which medication you wish to prescribe. Type the first few letters of the drug name. You will then see a list of possible drugs starting with these letters. Choose the required medication. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.
If you choose «formulary», you will be presented with a search box for the approved formulary. Search for the drug required then select it. You will then be asked to check that the medication you have selected is correct. Enter the dose then confirm the prescription.

Examples of task cards for prescribing medication

Task 1: Change dose of prescribed drug

Task 2: Formulary selection

Task 3: Dose checking

Dose checking is a safety precaution to check that the doctor has not prescribed a dangerously small or large dose.
Using the formulary id for the generic drug name, lookup the formulary and retrieve the recommended maximum and minimum dose.
Check the prescribed dose against the minimum and maximum. If outside the range, issue an error message saying that the dose is too high or too low.
If within the range, enable the 'Confirm' button.

XP and change

✧ Conventional wisdom in software engineering is to design for change. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.
✧ XP, however, maintains that this is not worthwhile as changes cannot be reliably anticipated.
✧ Rather, it proposes constant code improvement (refactoring) to make changes easier when they have to be implemented.

Refactoring

✧ Programming team look for possible software improvements and make these improvements even where there is no immediate need for them.
✧ This improves the understandability of the software and so reduces the need for documentation.
✧ Changes are easier to make because the code is well structured and clear.
✧ However, some changes requires architecture refactoring and this is much more expensive.

Examples of refactoring

✧ Re-organization of a class hierarchy to remove duplicate code.
✧ Tidying up and renaming attributes and methods to make them easier to understand.
✧ The replacement of inline code with calls to methods that have been included in a program library.

Testing in XP

✧ Testing is central to XP and XP has developed an approach where the program is tested after every change has been made.
✧ XP testing features:
- Test-first development.
- Incremental test development from scenarios.
- User involvement in test development and validation.
- Automated test harnesses are used to run all component tests each time that a new release is built.

Test-first development

✧Writing tests before code clarifies the requirements to be implemented.
✧ Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.
- Usually relies on a testing framework such as Junit.

✧ All previous and new tests are run automatically when new functionality is added, thus checking that the new functionality has not introduced errors.

Customer involvement

✧ The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system.
✧ The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs.
✧ However, people adopting the customer role have limited time available and so cannot work full-time with the development team. They may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process.

Test case description for dose checking

Test 4: Dose Checking

Input:
1: A number in mg representing a single dose of the drug.
2: A number representing the number of single doses per day.

Tests:
1: Test for inputs where the single dose is correct but the frequency is too high.
2: Test for inputs where the single dose is too high and too low.
3: Test for inputs where the single dose frequency is too high and too low
4: Test for inputs where the single dose frequency is in the permitted range.

Output:
Ok or error message indicating that the dose is outside safe range

Test automation

✧ Test automation means that tests are written as executable components before the task is implemented
  - These testing components should be stand-alone, should simulate the submission of input to be tested and should check that the result meets the output specification. An automated test framework (e.g. Junit) is a system that makes it easy to write executable tests and submit a set of tests for execution.

✧ As testing is automated, there is always a set of tests that can be quickly and easily executed
  - Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

XP testing difficulties

✧ Programmers prefer programming to testing and sometimes they take short cuts when writing tests.

For example, they may write incomplete tests that do not check for all possible exceptions that may occur.

✧ Some tests can be very difficult to write incrementally.

For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.

✧ It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage.

Pair programming

✧ In XP, programmers work in pairs, sitting together to develop code.

✧ This helps develop common ownership of code and spreads knowledge across the team.

✧ It serves as an informal review process as each line of code is looked at by more than 1 person.

✧ It encourages refactoring as the whole team can benefit from this.

✧ Measurements suggest that development productivity with pair programming is similar to that of two people working independently.

✧ In pair programming, programmers sit together at the same workstation to develop the software.

✧ Pairs are created dynamically so that all team members work with each other during the development process.

✧ The sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave.

✧ Pair programming is not necessarily inefficient and there is evidence that a pair working together is more efficient than 2 programmers working separately.

Advantages of pair programming

✧ It supports the idea of collective ownership and responsibility for the system.

- Individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.

✧ It acts as an informal review process because each line of code is looked at by at least two people.

✧ It helps support refactoring, which is a process of software improvement.

- Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

Agile project management

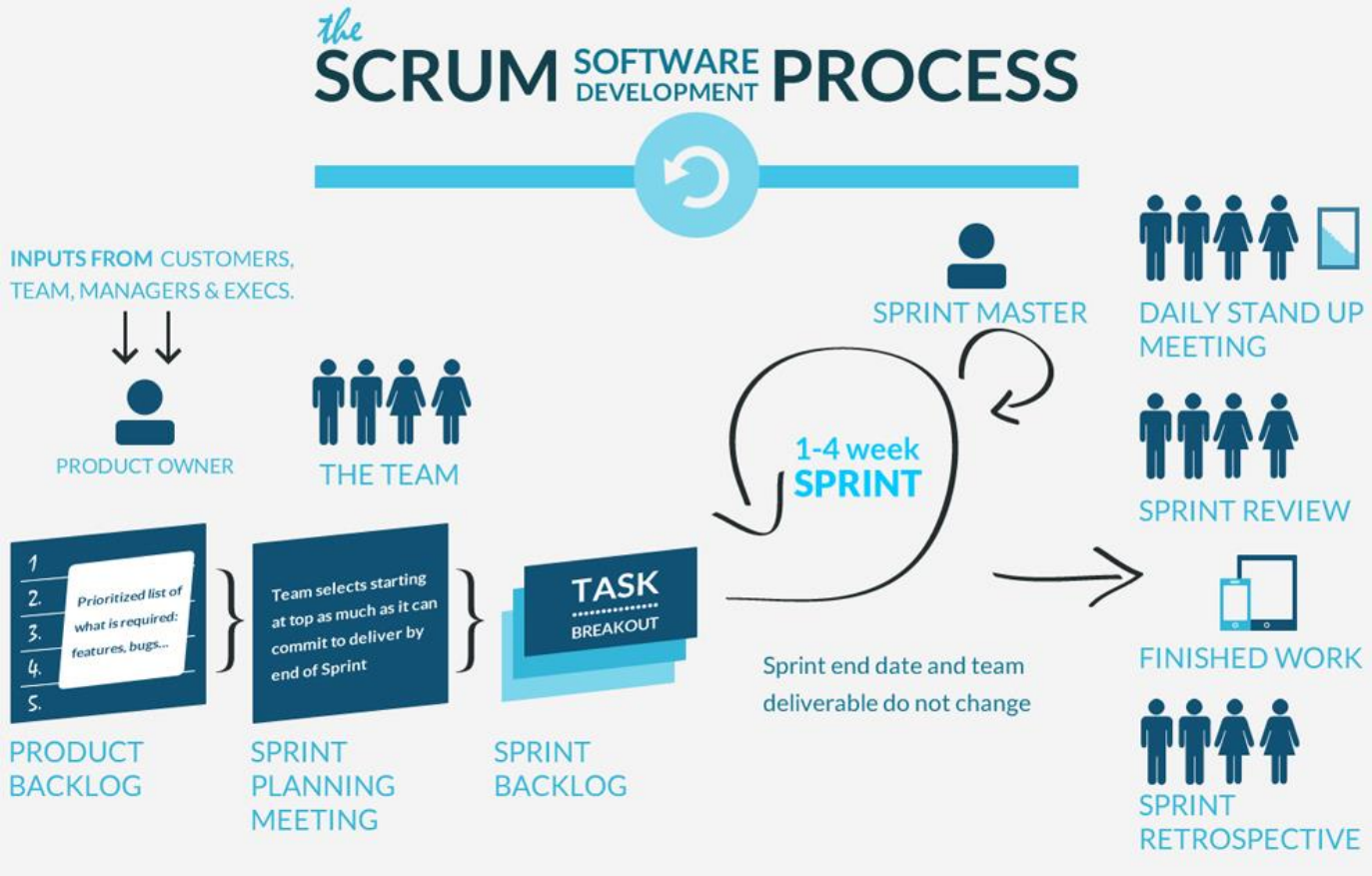
✧ The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project.

✧ The standard approach to project management is plandriven. Managers draw up a plan for the project showing what should be delivered, when it should be delivered and who will work on the development of the project deliverables.

✧ Agile project management requires a different approach, which is adapted to incremental development and the particular strengths of agile methods.

SCRUM Software Development Process

A scrum is an action associated with the game of rugby. It is enacted when players huddle together with the objective of moving the ball strategically towards the goal post. The SCRUM agile development methodology is derived from this action and draws from the rugby scrum some principles which are embedded throughout the life-cycle of a SCRUM enabled project.



The Product Backlog

As in any sport, teams are rarely successful without a "game plan" aimed at providing a transparent and unambiguous mechanism towards achieving a common goal. This mechanism, called the Product Backlog is achieved when a Product Owner creates a list of Stories based on initial client requirements. These Stories describe business functionality which will contribute towards a shippable software product. The nature of SCRUM and its Sprints allow for products to be delivered iteratively and by maintaining a Product Backlog; business functionality may increase or decrease in size based on decisions throughout the lifespan of a project.

Sprints

Like the game of rugby, SCRUM is enacted through the process of Stop, Start intervals (2 weeks to a month) known as Sprints. The objective of SCRUM is to foster transparency, accountability and enable agile processes. To achieve this, a number of Milestones are undertaken in a Sprint. Sprint Milestones carry open invitations. However, only Sprint Team members are allowed to participate. Team members are comprised of the Product Owner, Scrum Master and the Development team (including QA). Together they are equally accountable to the success of a project.

Sprint Stand-Ups

On a daily basis, Team members participate in a time-boxed (15 minutes) Sprint Stand-Up in which the developers specifically address Stories by explaining:

1. What I did yesterday
2. What I am doing today
3. My impediments

A Stand-Up provides insight into the direction in which the project is heading and allows for issues to be addressed sooner in the development cycle. The role of the Scrum Master is much like a referee in which it ensures that the team remains focused towards achieving agreed upon Sprint goals by removing impediments and, managing the relationship between the Product Owner and the Development team.

Sprint Planning

During a Sprint, a Planning session allows the Team to ruminate on and further "Groom" stories which are selected from the Product Backlog and prioritised by the Product Owner for the upcoming Sprint. Towards the end of a Sprint, the Team hosts an open Review showcasing the Sprint goals achieved in the current Sprint. Finally, the Sprint is completed with a Retrospective, providing the Team an opportunity to openly express their thoughts on what they perceived were "Good" or "Bad" events in a Sprint. The things learnt from the Retrospective provide for Team growth and increased optimisation as issues are addressed and solutions are put in place.

Why SCRUM is better

In essence SCRUM is a simple agile product development framework implemented to manage the complex task of creating software. It creates self-organising teams who learn to manage themselves efficiently, as team members soon realise accountability through the actions of selecting and reviewing their own work.

For clients, SCRUM offers complete transparency and an enhanced sense of ownership as they are actively involved in the collaboration of each Sprint Milestone and possess a shared responsibility for the maintenance of the Product Backlog.

On the whole, SCRUM promotes self-management, improved communication, improved quality and a controlled and iterative approach to product releases.

Scaling agile methods

✧ Agile methods have proved to be successful for small and medium sized projects that can be developed by a small co-located team.
✧ It is sometimes argued that the success of these methods comes because of improved communications which is possible when everyone is working together.
✧ Scaling up agile methods involves changing these to cope with larger, longer projects where there are multiple development teams, perhaps working in different locations.

Large systems development

✧ Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones.
✧ Large systems are 'brownfield systems', that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development.
✧ Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development.
✧ Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.
✧ Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
✧ Large systems usually have a diverse set of stakeholders.It is practically impossible to involve all of these different stakeholders in the development process.

Scaling out and scaling up

✧ 'Scaling up' is concerned with using agile methods for developing large software systems that cannot be developed by a small team.
✧ 'Scaling out' is concerned with how agile methods can be introduced across a large organization with many years of software development experience.
✧ When scaling agile methods it is essential to maintain agile fundamentals
  - Flexible planning, frequent system releases, continuous integration, test-driven development and good team communications.

Scaling up to large systems

✧ For large systems development, it is not possible to focus only on the code of the system. You need to do more up-front design and system documentation
✧ Cross-team communication mechanisms have to be designed and used. This should involve regular phone and video conferences between team members and frequent, short electronic meetings where teams update each other on progress.

✧ Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible. However, it is essential to maintain frequent system builds and regular releases of the system.

Scaling out to large companies

✧ Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach.

✧ Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods.

✧ Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities.

✧ There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

UNIT II

Socio-technical Systems

Objectives

• Know what a socio-technical system is and the distinction between a socio-technical system and a computer-based system
• Introduce Introduce the concept concept of emergent emergent system properties properties such as reliability, performance, safety and security
• Understand system engineering process activities
• Understand why the organisational context of a system affects its design and use
• Understand legacy systems and why these are critical to many businesses

What is a system?

• A system is a purposeful collection of inter-related components working together to achieve some common objective.
• A system may include software, mechanical, electrical and electronic hardware and be operated by people. electrical and electronic hardware and be operated by people.
• System components are dependent on other system components
• The properties and behaviour of system components are inextricably(can't escape) intermingle

System categories

• Technical computer-based systems
  – Systems that include hardware and software but where the operators and operational processes are not normally considered to be part of the system. The system is not self-aware. E.g.: TV, Mobile phone
• Socio-technical systems
  – Systems that include technical systems but also operational processes and people who use and interact with the technical system. Socio-technical systems are governed by organisational policies and rules. E.g.: Book publication

Socio-technical system characteristics

• Emergent properties
  – Properties of the system of a whole that depend on the system components and their relationships.
• Non-deterministic
  – They do not always produce the same output when presented with the same input because the system's behaviour is partially dependent on human operators.
• Complex relationships with organisational objectives
  – The extent to which the system supports organisational objectives does not just depend on the system itself.

Emergent properties

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

Examples of emergent properties

Property	Description
Volume	The volume of a system (the total space occupied) varies depending on how the component assemblies are arranged and connected.
Reliability	System reliability depends on component reliability but unexpected interactions can cause new types of failure and therefore affect the reliability of the system.
Security	The security of the system (its ability to resist attack) is a complex property that cannot be easily measured. Attacks may be devised that were not anticipated by the system designers and so may defeat built-in safeguards.
Repairability	This property reflects how easy it is to fix a problem with the system once it has been discovered. It depends on being able to diagnose the problem, access the components that are faulty and modify or replace these components.
Usability	This property reflects how easy it is to use the system. It depends on the technical system components, its operators and its operating environment.

Types of emergent property

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

E.g. System reliability

• To illustrate the complexity of emergent properties, consider the property of system reliability
• Because of component inter-dependencies, faults can be propagated propagated through through the system.
• System failures often occur because of unforeseen inter-relationships between components.
• It is probably impossible to anticipate all possible component relationships.

Influences on reliability

• Hardware reliability
  – What is the probability of a hardware component failing and how long does it take to repair that component?
• Software reliability
  – How likely is it that a software component will produce an incorrect output. Software failure is usually distinct from hardware failure in that software does not wear out.
• Operator reliability
  – How likely is it that the operator of a system will make an error?

Reliability relationships

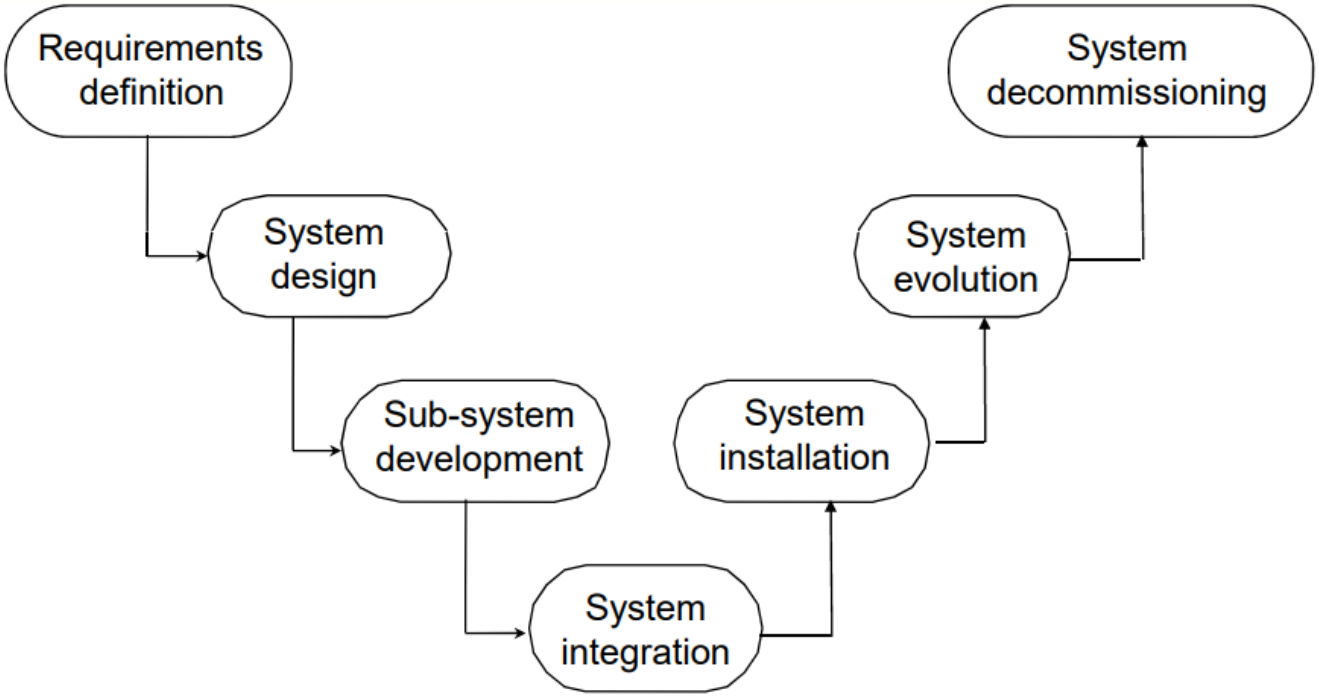
• Hardware failure can generate spurious signals that are outside the range of inputs expected by the software.
• Software errors can cause alarms to be activated which cause operator stress and lead to operator errors.
• The environment in which a system is installed can affect its reliability.

Systems engineering

• Is a activity of specifying, designing, implementing, validating, deploying and maintaining socio-technical systems.
• Concerned Concerned with the services services provided provided by the system, constraints on its construction and operation and the ways in which it is used.

The systems engineering process

• Usually follows a 'waterfall' model

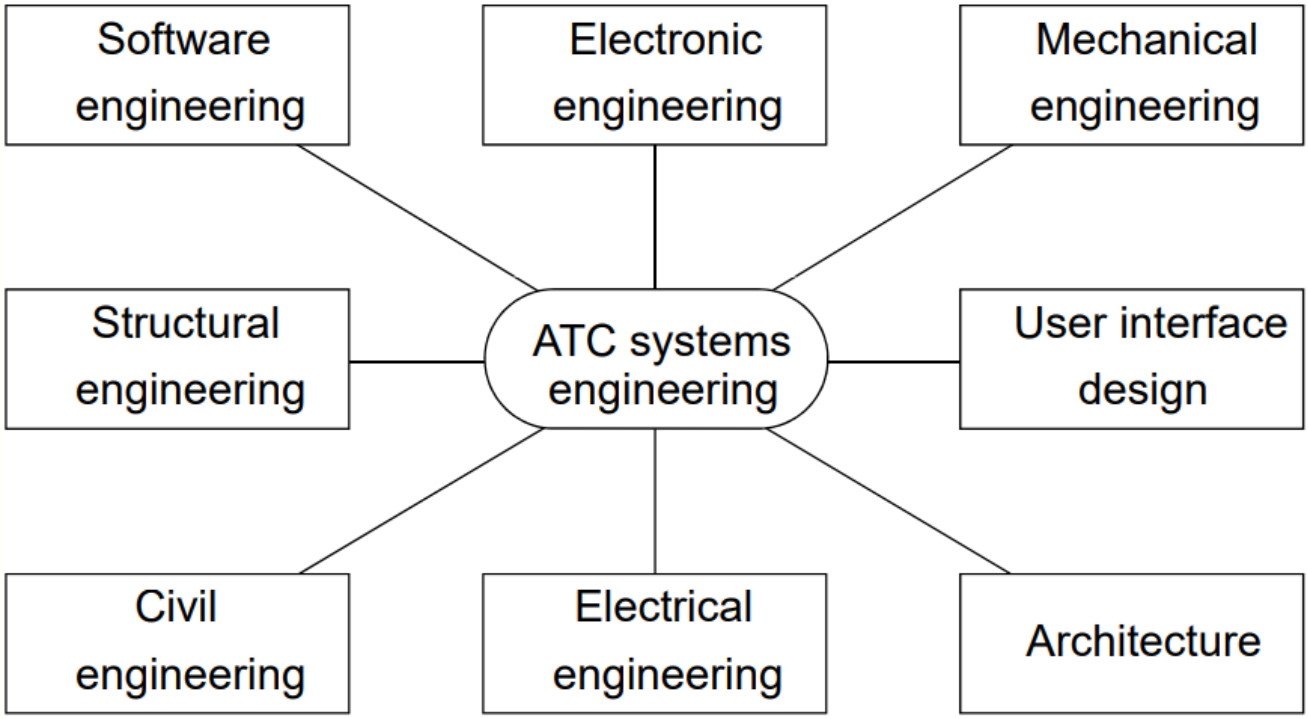


The system engineering process

• Distinction between system Engineering process and software development process:

1. Limited scope for rework during system development: Little scope for iteration between phases because hardware changes changes are very expensive expensive. Software Software may have to compensate compensate for hardware problems. E.g.: Mobile base station
2. Interdisciplinary involvement: Much scope for misunderstanding here. Different disciplines use a different vocabulary and much negotiation is required. Engineers may have personal agendas to fulfil.

Inter-disciplinary involvement



1. System requirements definition

• Specifies what the system should do(its functions) and its essential and desirable properties
• Three types of requirement defined at this stage

– Abstract functional requirements. The Basic functions that the system must provide provide are defined defined in an abstract abstract way;
– System properties. Non-functional requirements such as availability performance and safety for the system in general are defined;
– Characteristics that the system must NOT exhibit. Specify what system must NOT do.

System objectives

• Should also define overall organisational objectives for the system.
• Should define why a system is being procured for a particular environment.
• Functional objectives

– To provide a fire and intruder alarm system for the building which will provide internal and external warning of fire or unauthorized intrusion.

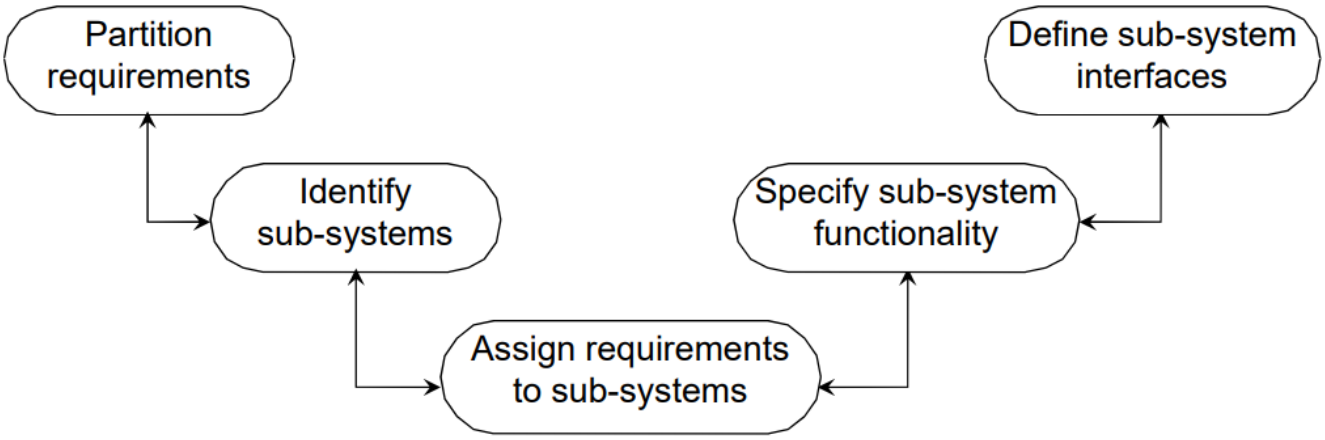
• Organisational objectives
  – To ensure that the normal functioning of work carried out in the building is not seriously disrupted by events such as fire and unauthorized intrusion.

System requirements problems

• Complex systems are usually developed to address wicked problems
  – Problems that are not fully understood;
  – Changing Changing as the system is being specified specified.
• Must anticipate hardware/communications developments over the lifetime of the system.
• Hard to define non-functional requirements (particularly) without knowing the component structure of the system.

2. The system design process

• Concerned with how the system functionality is to be provided by the components of the system.



The system design process Activities

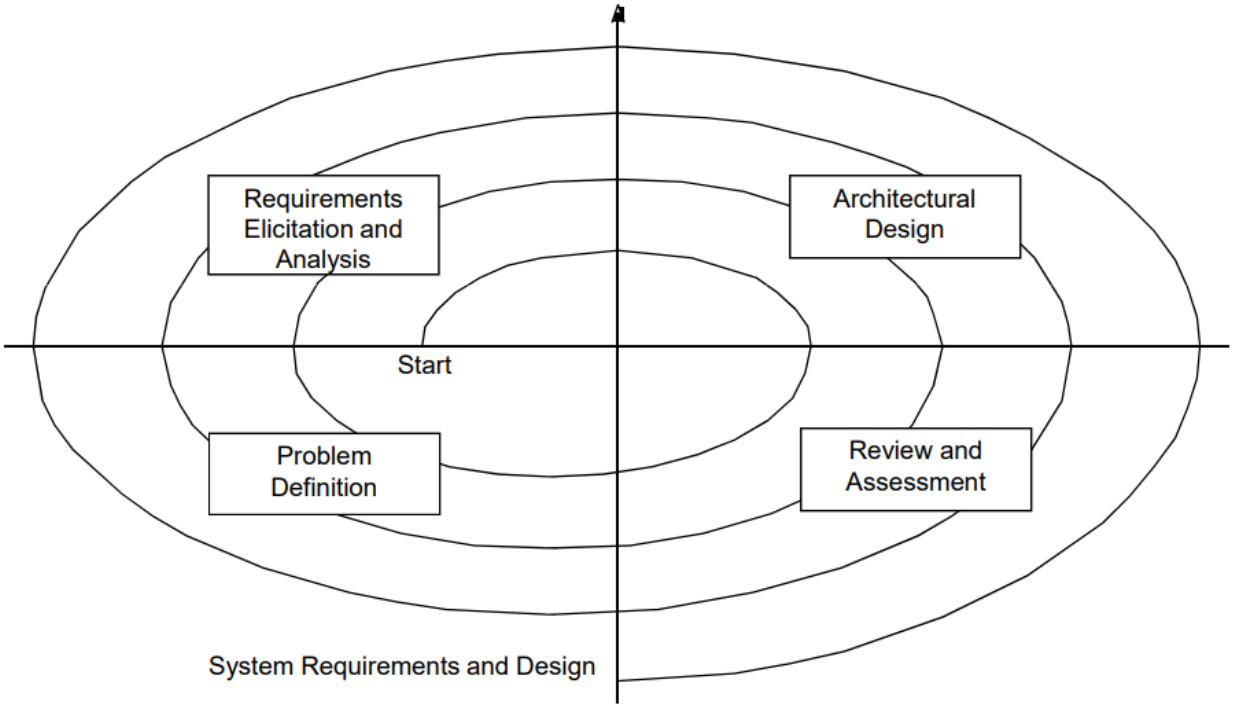
1. Partition requirements
   – Analyse and Organise requirements into related groups.
2. Identify sub-systems
   – Identify a set of sub-systems which collectively can meet the system requirements. Groups of requirements relates to subsystem.
3. Assign requirements requirements to sub-systems systems
   – Straight forward if the requirements portioning is used to drive subsystem identification. Causes particular problems when COTS(Commercial Off-TheShelf) are integrated.
4. Specify sub-system functionality.
   – Specify specific function provided by each subsystem and identify relation between each subsystem.
5. Define sub-system interfaces(Critical activity)
   – Interfaces that are provided and required by each subsystem for parallel sub-system development.

Requirements and design

• Requirements engineering and system design are inextricably linked.

• Constraints posed by existing systems limit design choices so the actual design to be used may be a requirement.
• Initial design may be necessary to structure the requirements.
• As you do design, you learn more about the requirements.
• These linked process may be thought as Spiral.

Spiral model of requirements/design



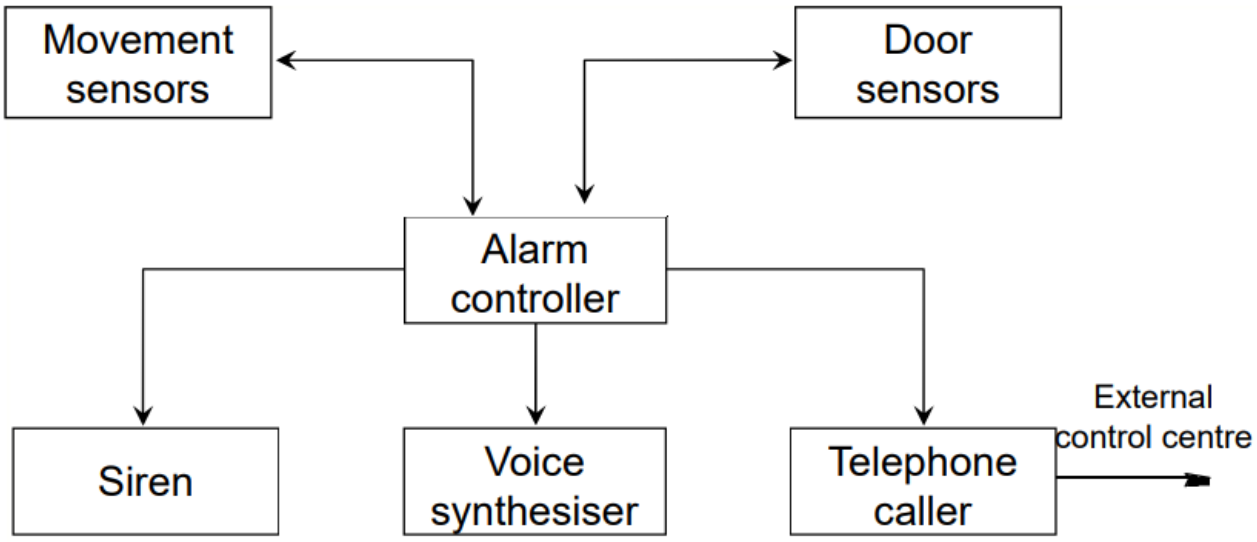
System design problems

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

3.System modelling

• An architectural model presents an abstract view of the sub-systems making up a system. Usually presented as a block diagram
• System is decomposed decomposed into set of interacting interacting subsystems.
• May identify different types of functional component in the model

Burglar alarm system



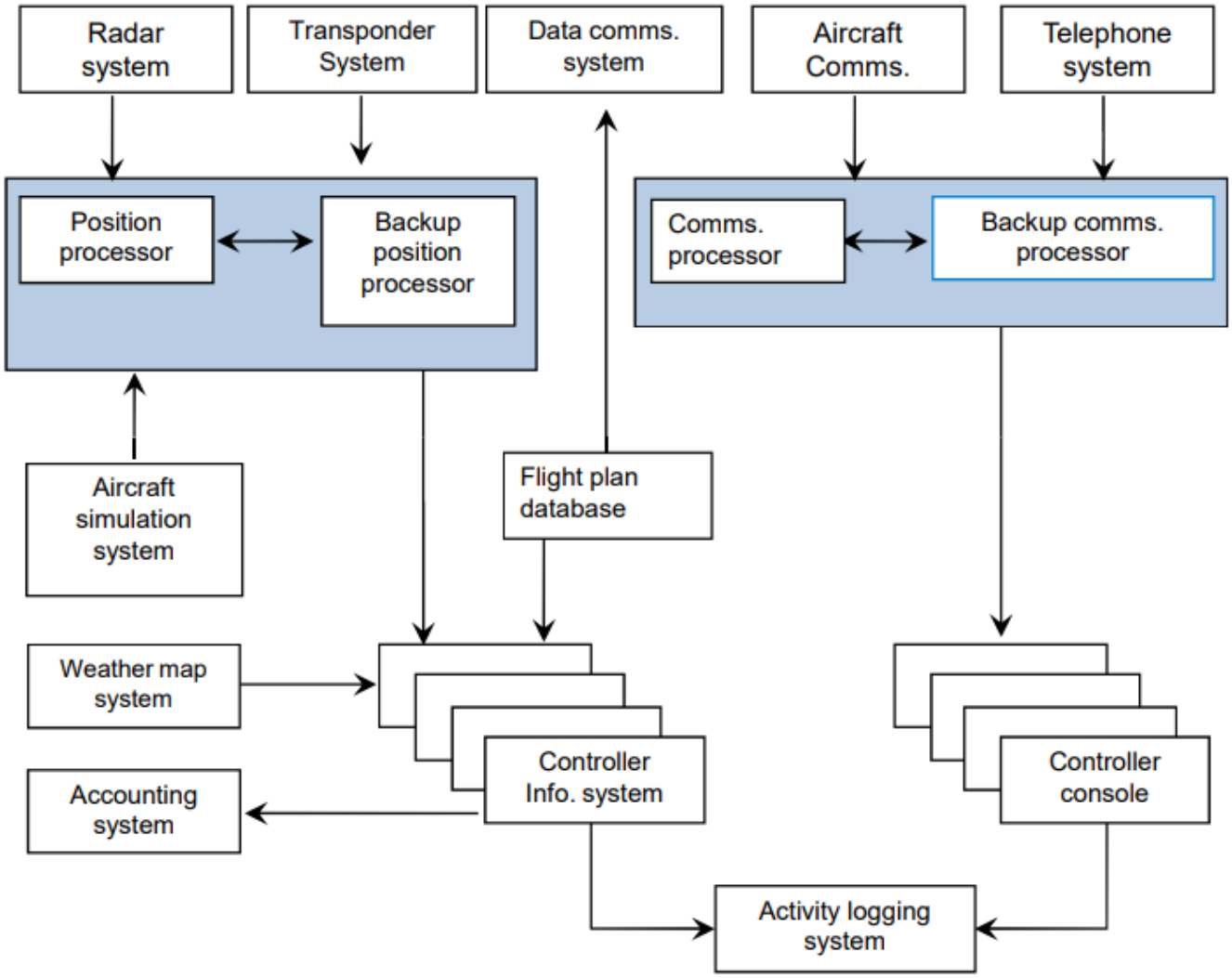
Sub-system description

Sub-system	Description
Movement sensors	Detects movement in the rooms monitored by the system
Door sensors	Detects door opening in the external doors of the building
Alarm controller	Controls the operation of the system
Siren	Emits an audible warning when an intruder is suspected
Voice synthesizer	Synthesizes a voice message giving the location of the suspected intruder
Telephone caller	Makes external calls to notify security, the police, etc.

System Architecture model

• The System architectural model is used to identify hardware and software components.
• Subsystems are appropriately classified to their functions before making decisions about software/hardware trade-offs.
• For all size of systems Block diagram can be used. For example ATC System.

ATC system architecture



4. Sub-system development

• Subsystems identified during system design are implemented.
• Subsystem are usually developed in parallel
• May involve involve some COTS (Commercial (Commercial Off-the-Shelf) systems procurement.
• Lack of communication across implementation teams.
• Bureaucratic and slow mechanism for proposing system changes means that the development schedule may be extended because of the need for rework.

5. System integration

• The process of putting hardware, software and people together to make a system.
• Integration can be done using 'big bang' approach(all at once).
• But, normally incremental integration is done so that sub-systems are integrated integrated one at a time, which reduces reduces the cost of error location and avoids all subsystem development to finish at the same time.
• Interface problems between sub-systems are usually found at this stage.
• After completion, the system has to be installed in the customer's environment

6. System evolution

• Large systems have a long lifetime. They must evolve to correct errors in the original requirements and meet the emerged requirements.
• Evolution is inherently costly
  – Changes must be analysed from a technical and business perspective;
  – Sub-systems are not independent, changes to one system may effect performance or behaviour of other subsystems. So, change is anticipated.
  – There is rarely a rationale for original design decisions;
  – System structure is corrupted as changes are made to it.

7. System decommissioning

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

Organisations / people /Computer systems

• Socio-technical systems are organisational systems intended to help deliver some organisational or business goal.
• If you do not understand the organisational environment where a system is used, the system is less likely to meet the real needs of the business and its users.

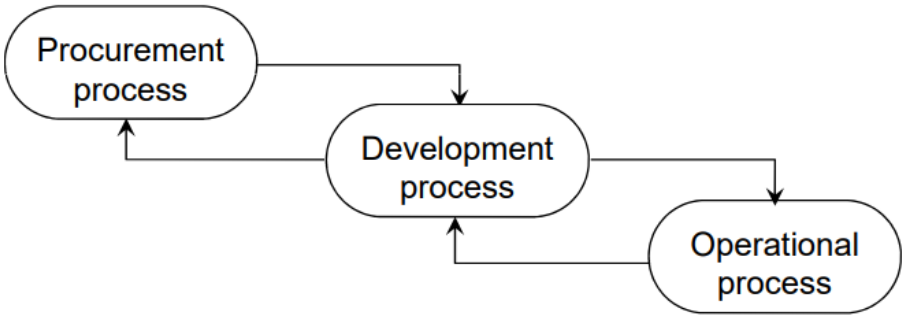
Human, Social and Organisational factors

• Human & organisational factors from systems environment that affect the system design includes:
• Process changes
  – Does the system require changes to the work processes in the environment? . If yes training required.
• Job changes
  – Does the system de-skill the users in an environment or cause them to change the way they work?
• Organisational changes
  – Does the system change the political power structure in an organisation?

Organisational processes

• The processes of systems engineering overlap and interact with organisational procurement processes.
• Operational processes are the processes involved in using the system for its intended purpose. For new systems, these have to be defined defined as part of the system design.
• Operational processes should be designed to be flexible and should not force operations to be done in a particular way. It is important that human operators can use their initiative if problems arise.

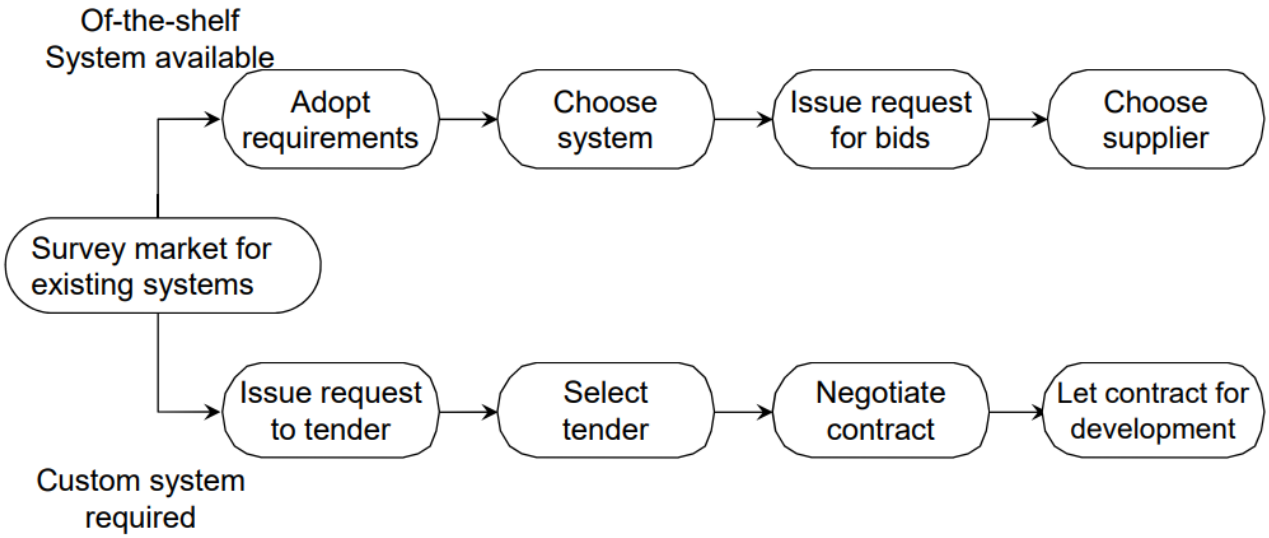
Procurement/development processes



System procurement

• Acquiring a system for an organization to meet some need.
• Some system specification and architectural design is usually necessary before procurement
  – You need a specification to let a contract for system development You need a specification to let a contract for system development
  – The specification may allow you to buy a commercial off-theshelf (COTS) system. Almost always cheaper than developing a system from scratch
• Large complex systems usually consist of a mix of off the shelf and specially designed components. The procurement processes for these different types of component are usually different.

The system procurement process



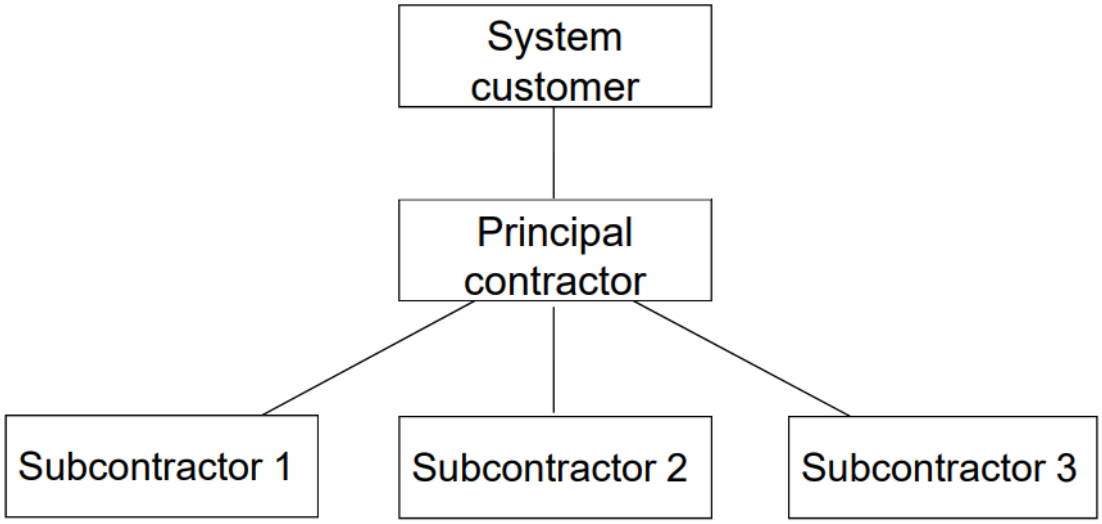
Procurement issues

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

Contractors and sub-contractors

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

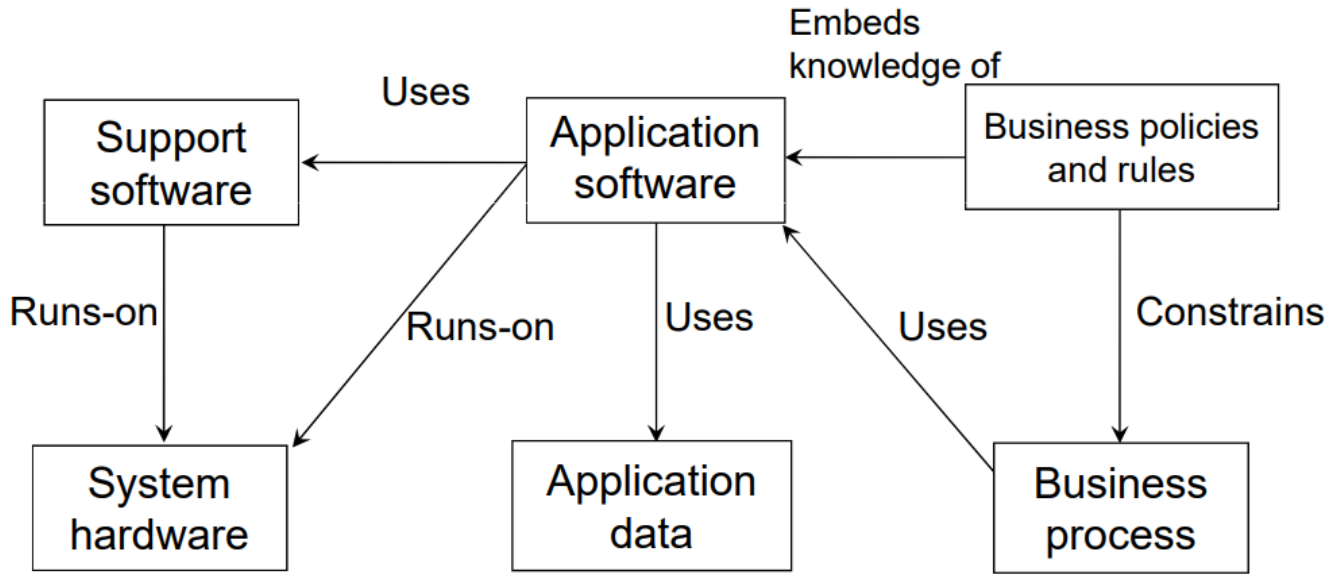
Contractor/Sub-contractor model



Legacy systems

• Socio-technical systems that have been developed in the past using old or obsolete technology.
• Crucial to the operation of a business and it is often too risky to discard these systems
  – Bank customer accounting system;
  – Aircraft maintenance system.
• Legacy systems constrain new business processes and consume a high proportion of company budgets.

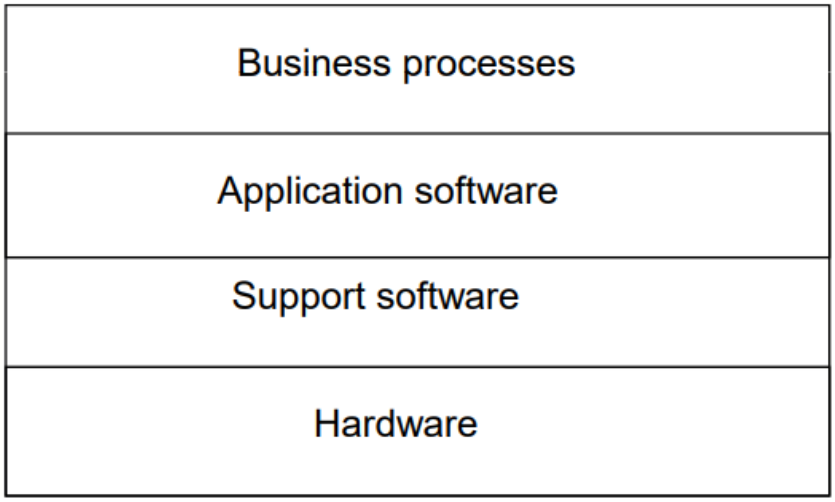
Legacy system components



Legacy system components

- System Hardware - may be obsolete mainframe hardware.
- Support software - may rely on support software from suppliers who are no longer in business.
- Application software - may be written in obsolete programming languages.
- Application data - often incomplete and inconsistent.
- Business processes - may be constrained by software structure and functionality.
- Business policies and rules - may be implicit and embedded in the system software.
- Layered model of legacy system
- Changing in one layer requires changes in both neighbor layers

Socio-technical system



Critical Systems

Objectives

• To explain what is meant by a critical system where system failure can have severe human or economic consequence.
• To explain four dimensions of dependability - availability, reliability, safety and security.
• To explain that, to achieve dependability, you need to avoid mistakes, detect and remove errors and limit damage caused by failure.

Critical Systems

• If the system failure results in significant economic losses, physical damages or threats to human life than the system is called critical systems. 3 types of it are:
• Safety-critical systems
  – Failure results in loss of life, injury or damage to the environment;
  – Chemical plant protection system;
• Mission-critical systems
  – Failure results in failure of some goal-directed activity;
  – Spacecraft navigation system;
• Business-critical systems
  – Failure results in high economic losses;
  – Customer accounting system in a bank;

System dependability

• The most important emergent property of a critical system is its dependability. It covers the related system attributes of availability, reliability, safety & security.
• Importance of dependability
  – Systems that are unreliable, unsafe or insecure are often rejected by their users(refuse to the product from the same company).
  – System failure costs may be very high.(reactor / aircraft navigation)
  – Untrustworthy systems may cause information loss with a high consequent recovery cost.

Development methods for critical systems

• Trusted methods and technique must be used.
• These methods are not cost-effective for other types of system.
• The older methods strengths & weaknesses are understood
• Formal methods reduce the amount of testing required.
Example :
  – Formal mathematical method

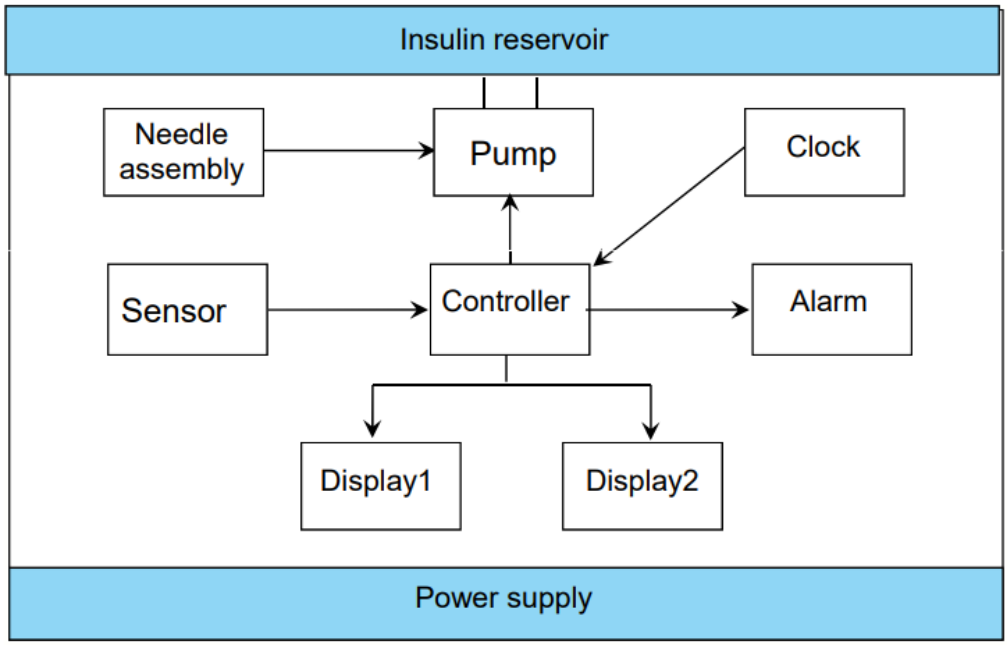
Socio-technical critical systems Failures

- Hardware failure
  - Hardware fails because of design and manufacturing errors or because components have reached the end of their natural life.
- Software failure
  - Software fails due to errors in its specification, design or implementation.
- Human Operator failure
  - Fail to operate correctly.
  - Now perhaps the largest single cause of system failures.

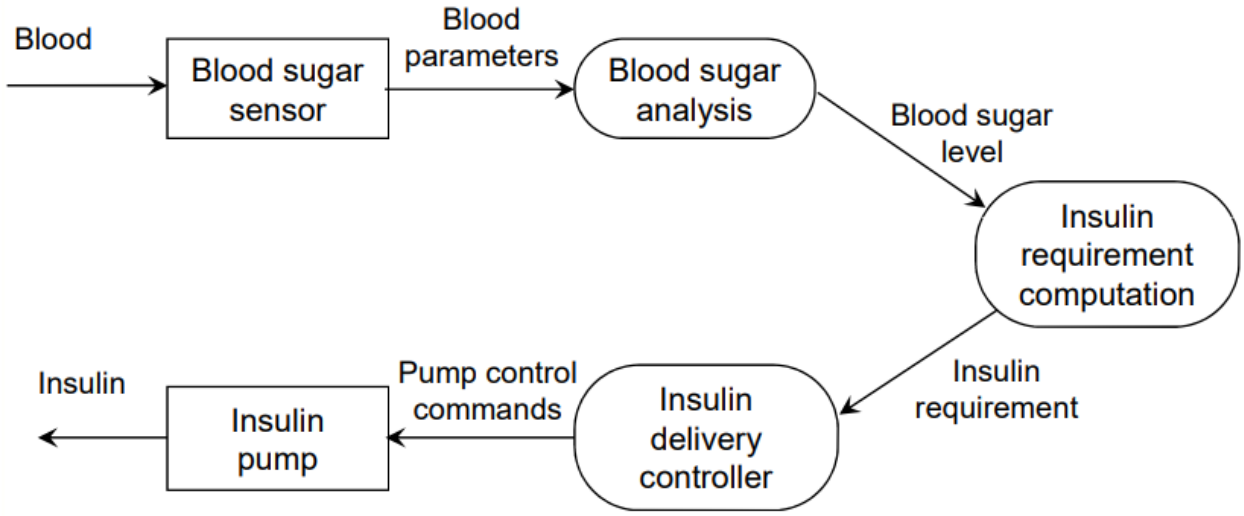
A Simple safety Critical System

- Example of software-controlled insulin pump.
- Used by diabetics to simulate the function of insulin, an essential hormone that metabolises blood glucose.
- Measures blood glucose (sugar) using a microsensor and computes the insulin dose required to metabolise the glucose.

Insulin pump organization



Insulin pump data-flow



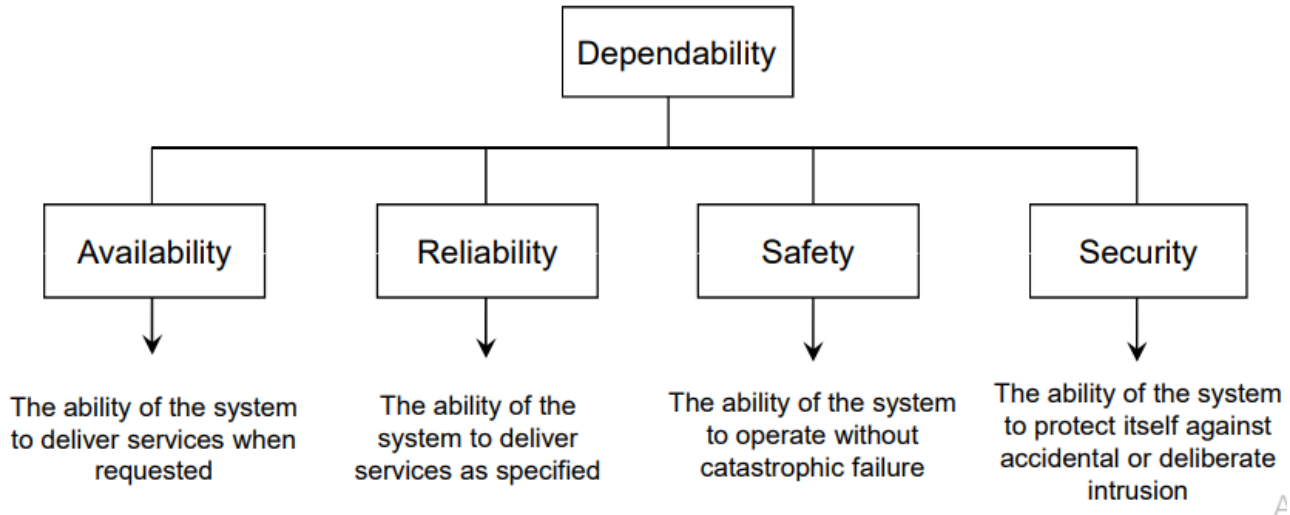
Dependability requirements

• The system shall be available to deliver insulin when required to do so.
• The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.
• The essential safety requirement is that excessive doses of insulin should never be delivered as this is potentially life threatening.

System Dependability

• The dependability of a system equates to its trustworthiness.
• A dependable system is a system that is trusted by its users.
• Principal dimensions of dependability are:
– **Availability** :- Probability that it will be up & running & able to deliver at any given time ;
– **Reliability** :-Correct delivery of services as expected by user over a given period of time;
– **Safety** :-A Judgment of how likely the system will cause damage to people or its environment;
– **Security** :- A Judgment of how likely the system can resist accidental or deliberate intrusions;

Dimensions of dependability



Other dependability properties

- **Repairability**
  - Reflects the extent to which the system can be repaired in the event of a failure
- **Maintainability**
  - Reflects the extent to which the system can be adapted to new requirements;
- **Survivability**
  - Reflects the extent to which the system can deliver services while it is under hostile attack;
- **Error tolerance**
  - Reflects the extent to which user input errors can be avoided and tolerated.

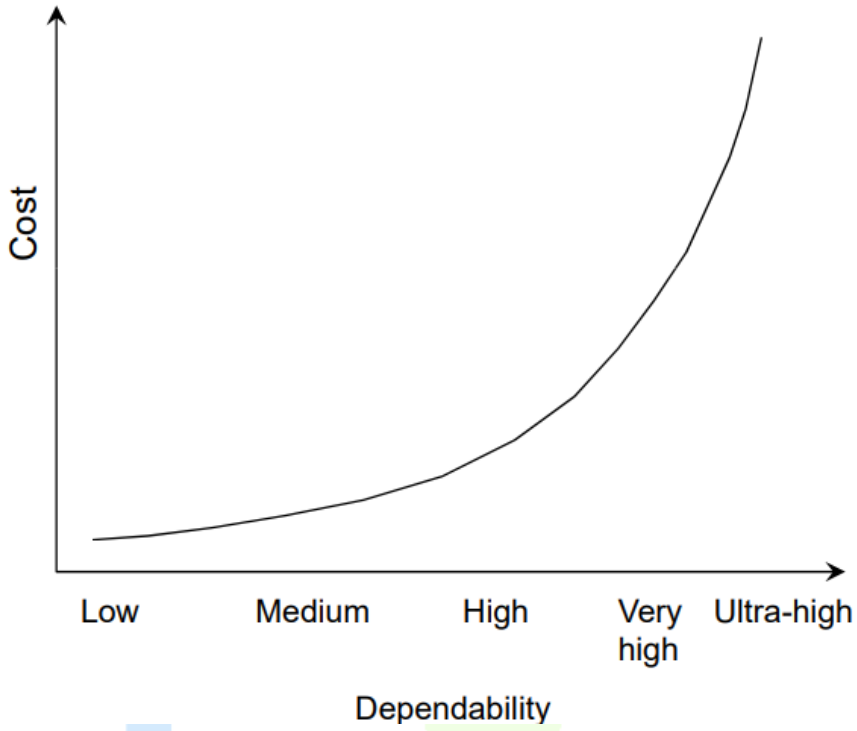
Dependability vs performance

- It is very difficult to tune systems to make them more dependable
- High level dependability can be achieved by expense of performance. Because it include extra/ redundant code to perform necessary checking
- It also increases the cost.

Dependability costs

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

Costs of increasing dependability



Availability and reliability

- **Reliability**
  - The probability of failure-free system operation over a specified time in a given environment for a specific purpose
- **Availability**
  - The probability that a system, at a point in time, will be operational and able to deliver the requested services
- Both of these attributes can be expressed quantitatively
- It is sometimes possible to include system availability under system reliability
  - Obviously if a system is unavailable it is not delivering the specified system services
- However, it is possible to have systems with low reliability that must be available. So long as system failures can be repaired quickly and do not damage data, low reliability may not be a problem
- Availability takes repair time into account

Faults, Errors and failures

- Failures are a usually a result of system errors that are derived from faults in the system
- However, faults do not necessarily result in system errors
  - The faulty system state may be transient and 'corrected' before an error arises
- Errors do not necessarily lead to system failures
  - The error can be corrected by built-in error detection and recovery

– The failure can be protected against by built-in protection facilities. These may, for example, protect system resources from system errors

Reliability terminology

Term	Description
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users
System error	An erroneous system state that can lead to system behaviour that is unexpected by system users.
System fault	A characteristic of a software system that can lead to a system error. For example, failure to initialise a variable could lead to that variable having the wrong value when it is used.
Human error or mistake	Human behaviour that results in the introduction of faults into a system.

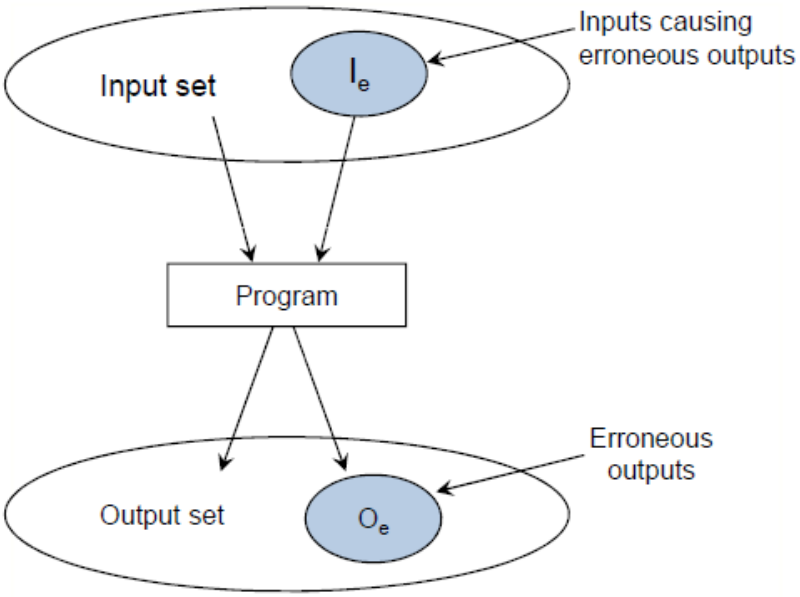
Reliability Improvement

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

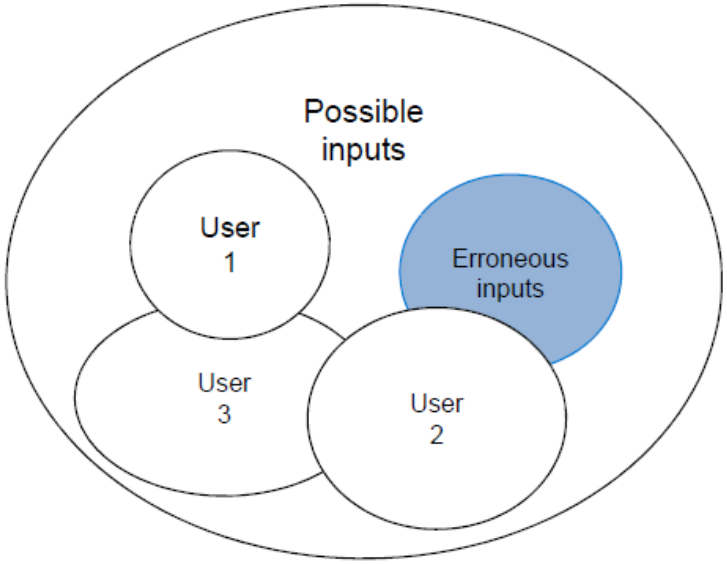
Reliability modelling

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

Input/output mapping



Reliability perception



Reliability improvement

• Removing X% of the faults in a system will not necessarily improve the reliability by X%. A study at IBM showed that removing 60% of product defects resulted in a 3% improvement in reliability

• Program defects may be in rarely executed sections of the code so may never be encountered by users. Removing these does not affect the perceived reliability

• A program with known faults may therefore still be seen as reliable by its users

Safety

• Safety is a property of a system that reflects the system should never damage people or the system's environment
• For example control & monitoring systems in aircraft
• It is increasingly important to consider software safety as more and more devices incorporate software-based control systems
• Safety requirements are exclusive requirements i.e. they exclude undesirable situations rather than specify required system services
• Safety critical software are 2 types

Types of Safety-critical software

• Primary safety-critical systems
  – Embedded software systems whose failure can cause hardware malfunction which results inhuman injury or environmental damage.
• Secondary safety-critical systems
  – Systems whose failure indirectly results in injury.
  – Eg. Medical Database holding details of drugs
• Discussion here focuses on primary safety-critical systems

Safety and reliability

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

Unsafe reliable systems

• Reasons why reliable system are not necessarily safe:
  • Specification errors
    • It does not describe the required behaviour in some critical situations
  • Hardware failures generating spurious inputs
    • Hard to anticipate in the specification
  • Operator error
    • Context-sensitive commands i.e. issuing the right command at the wrong time

Safety terminology

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

Ways to achieve Safety

- Hazard avoidance
  - The system is designed so that hazard simply cannot arise.
  - Eg. Press 2 buttons at the same time in a cutting machine to start
- Hazard detection and removal
  - The system is designed so that hazards are detected and removed before they result in an accident.
  - Eg. Open relief valve on detection over pressure in chemical plant.
- Damage limitation
  - The system includes protection features that minimise the damage that may result from an accident
  - Automatic fire safety system in aircraft.

Security

- Security is a system property that reflects the ability to protect itself from accidental or deliberate external attack.
- Security is becoming increasingly important as systems are networked so that external access to the system through the Internet is possible
- Security is an essential pre-requisite for availability, reliability and safety
- **Example :** Viruses, unauthorised use of service/data modification

Security terminology

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

Damage from insecurity

- Denial of service
  - – The system is forced into a state where normal services become unavailable.
- Corruption of programs or data
  - – The system components of the system may altered in an unauthorised way, which affect system behaviour & hence its reliability and safety
- Disclosure of confidential information
  - – Information that is managed by the system may be exposed to people who are not authorised to read or use that information

Security assurance

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

Requirements Engineering Processes

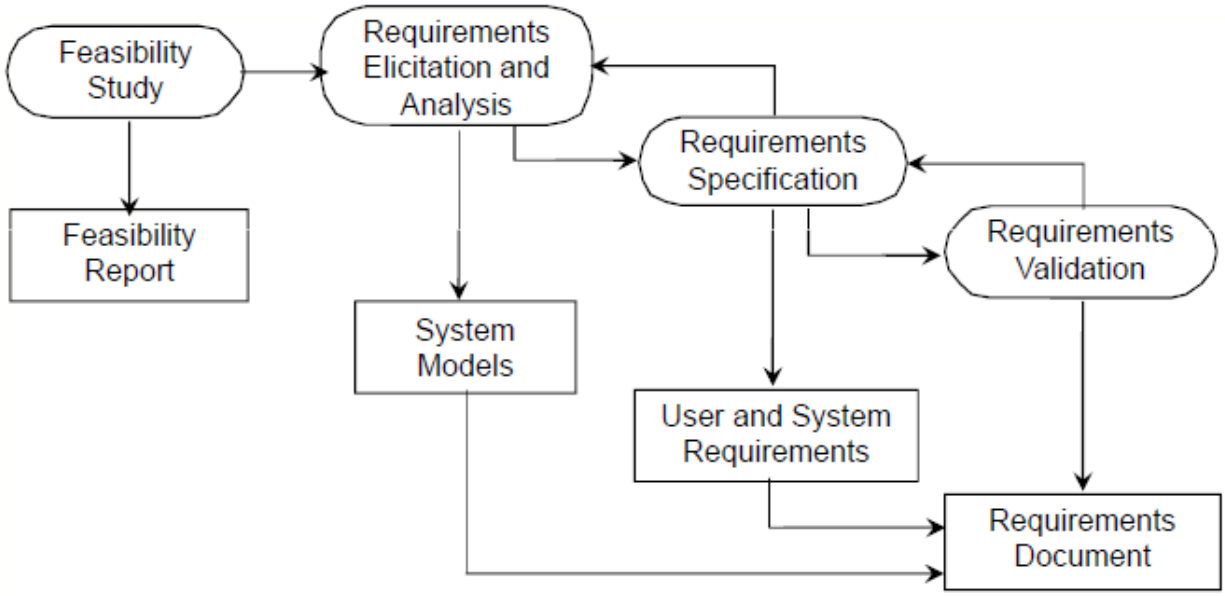
Objectives

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

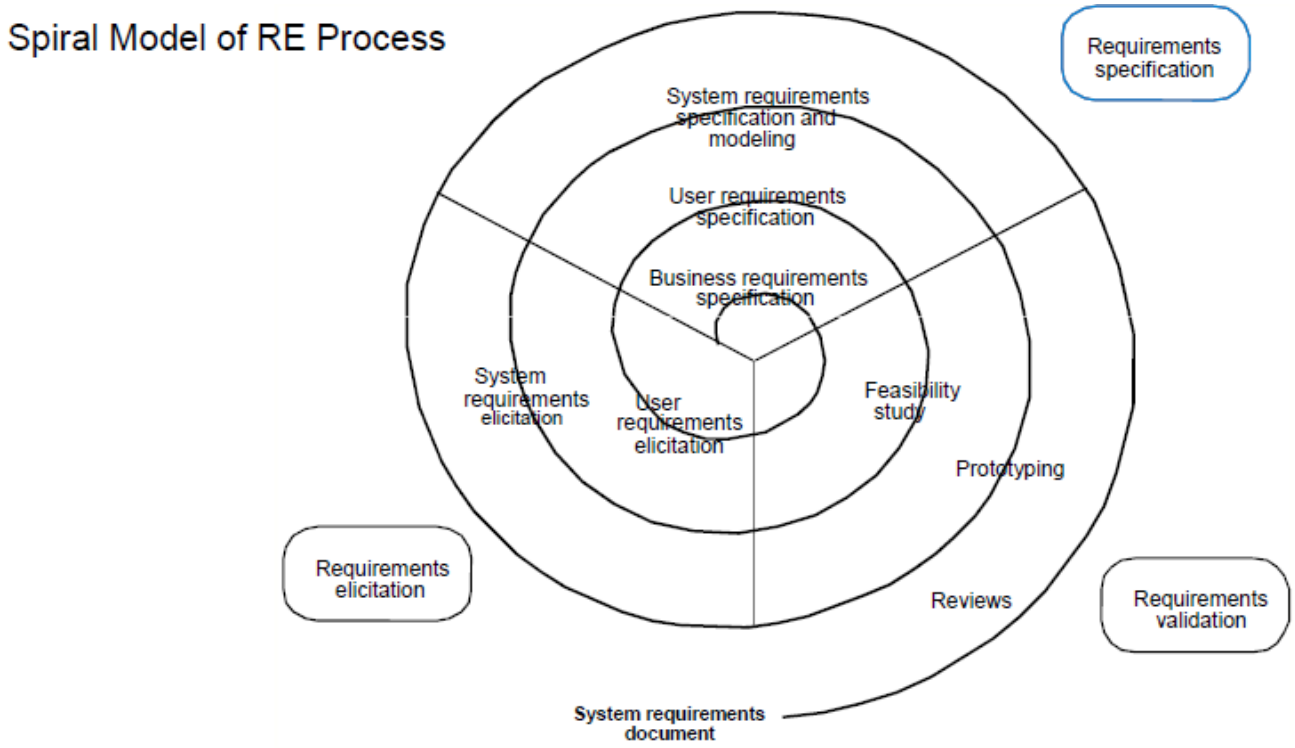
Requirements engineering processes

• The goal of RE Process is to create & maintain a system requirements documents.
• Includes 4 High level RE Sub-processes:
  – Feasibility study : usefulness to the business ;
  – Elicitation and analysis : discovering requirements;
  – Specification: conversion of requirement into some standard form;
  – Validation : check the requirements which defines the system that the customer wants;

The requirements engineering process



Alternative perspective of RE Process



Feasibility studies

- A feasibility study decides whether or not the proposed system is worthwhile.
- A short focused study that checks
  - If the system contributes to organisational objectives;
  - If the system can be implemented using current technology, within given cost and schedule constraints;
  - If the system can be integrated with other systems that are already in place.

Feasibility study implementation

- Feasibility study involves information assessment (what is required), information collection and report writing.
- Questions for people in the organisation for information assessment and collection:
  - What if the system wasn't implemented?
  - What are current process problems?
  - How will the proposed system help?
  - What will be the integration problems?
  - Is new technology needed?What skills?
  - What facilities must be supported by the proposed system?
- Feasibility study report should make a recommendation about the development to continue or not.

Requirements elicitation and analysis

• Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.

• May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders.*

• Eliciting & understanding stakeholder requirement is difficult due to the following reasons:

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

E&A Process activities

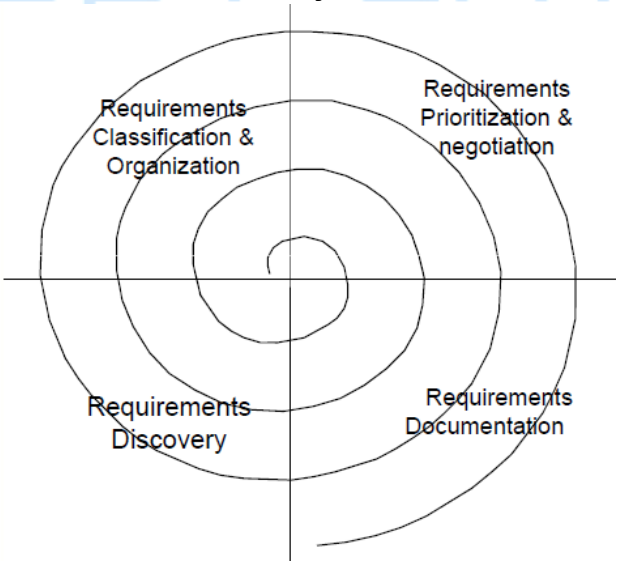
• Requirements discovery
  – Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.

• Requirements classification and organisation
  – Group related requirements and organises them into coherent clusters.

• Prioritisation and negotiation
  – Prioritising requirements and finding and resolving requirements conflicts.

• Requirements documentation
  – Requirements are documented and input into the next round of the spiral.

The requirements elicitation & analysis Process



Requirements discovery

• The process of gathering information about the proposed and existing systems and distilling the user and system requirements from this information.
• Sources of information include documentation, system stakeholders and the specifications of similar systems.
• Approaches & Techniques of requirements discovery are:
  – Viewpoints(approach)
  – Interviewing
  – Scenario
  – Use-cases
  – ethnography

Example : ATM stakeholders

• Bank customers – who receive services
• Representatives of other banks – who have reciprocal agreements for ATM usage
• Bank branch managers –who obtain management information
• Counter staff – who involved in day-to-day running of system
• Database administrators - who responsible for integrating system with customers database
• Bank Security managers
• The Bank's Marketing department
• Hardware and software maintenance engineers
• Banking regulators Who ensures system conforms to banking regulations

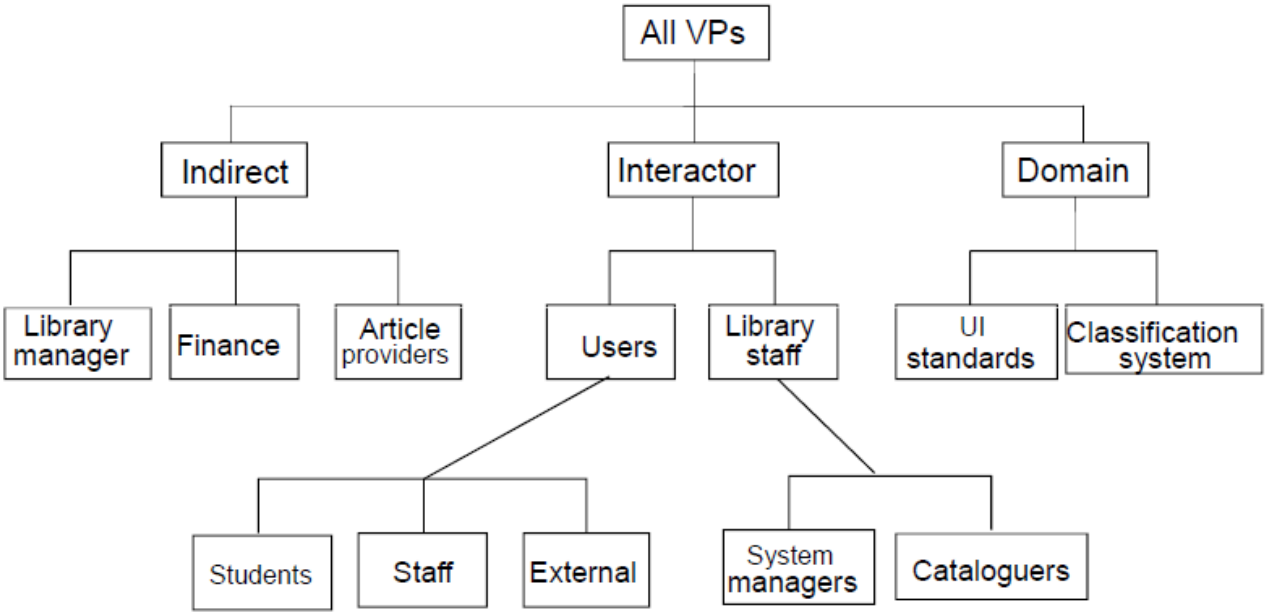
Viewpoints

• Recognizes multiple perspectives and provides framework for discovering conflicts in the requirements proposed by different stakeholders
• Viewpoints(VP) are used as a way of classifying Stakeholders and other sources of requirements.
• **There are 3 generic types of VP:**
• Interactor viewpoints
– People or other systems that interact directly with the system. In an ATM, the customer's and the account database are interactor VPs.
• Indirect viewpoints
– Stakeholders who do not use the system themselves but who influence the requirements. In an ATM, management and security staff are indirect viewpoints.
• Domain viewpoints
– Domain characteristics and constraints that influence the requirements. In an ATM, an example would be standards for inter-bank communications.

Viewpoint identification

- Identify viewpoints using following more specific VP types:
  - Providers and receivers of system services;
  - Systems that interact directly with the system being specified;
  - Regulations and standards that apply to the system;
  - Sources of business and non-functional requirements.
  - Engineers who have to develop and maintain the system;
  - Marketing and other business viewpoints.

LIBSYS viewpoint hierarchy



Interviewing

- Used for getting an overall understanding of what stakeholders do, how they interact with system and difficulties they face with current system
- In formal or informal interviewing, the RE team puts questions to stakeholders about the system that they use and the system to be developed.
- There are two types of interview
  - Closed interviews where a pre-defined set of questions are answered.
  - Open interviews where there is no pre-defined agenda and a range of issues are explored with stakeholders.

Interviews in practice

- Normally a mix of closed and open-ended interviewing.
- Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
- Interviews are not good for understanding domain requirements
  - Requirements engineers cannot understand specific domain terminology;

– Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.

Effective interviewers Characteristics

• Interviewers should be open-minded, willing to listen to stakeholders and should not have preconceived ideas about the requirements.
• They should prompt the interviewee with a question or a proposal and should not simply expect them to respond to a question such as 'what do you want'.

Scenarios

• Scenarios are real-life examples of how a system can be used.
• They should include
  – A description of what the system & user expect 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 concurrent activities;
  – A description of the state when the scenario finishes.

LIBSYS scenario...

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

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

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

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

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

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

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

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

Other activities: Simultaneous downloads of other articles.

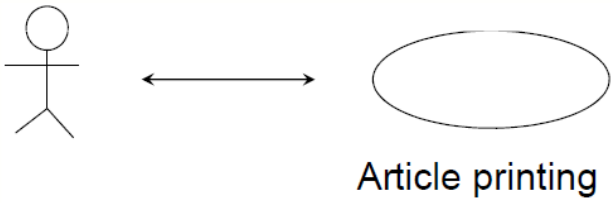
System state on completion: User is logged on. The downloaded article has been deleted from LIBSYS workspace if it has been flagged as print-only.

Use cases

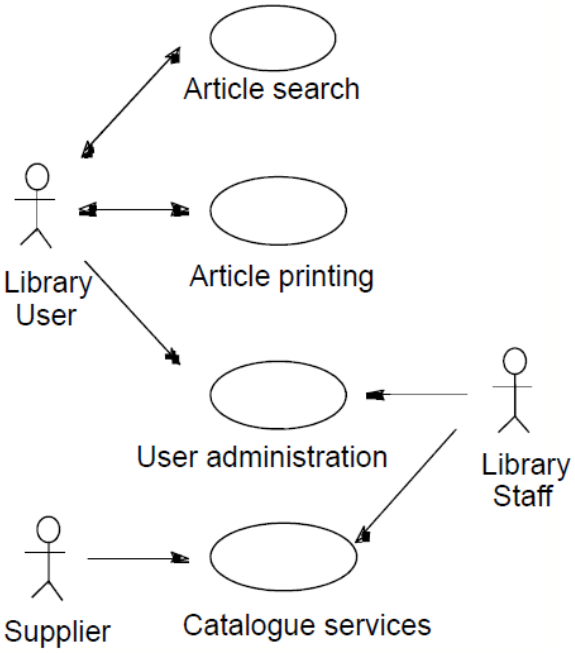
• Use-cases are a scenario based technique in the UML which identify the actors in an interaction and which describe the interaction itself.
• A set of use cases should describe all possible interactions with the system.
• Sequence diagrams may be used to add detail to use-cases by showing the sequence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

Simple Use-case for Article printing

• Actors is stick figures, each class of interaction is named ellipse.



Use cases for library system



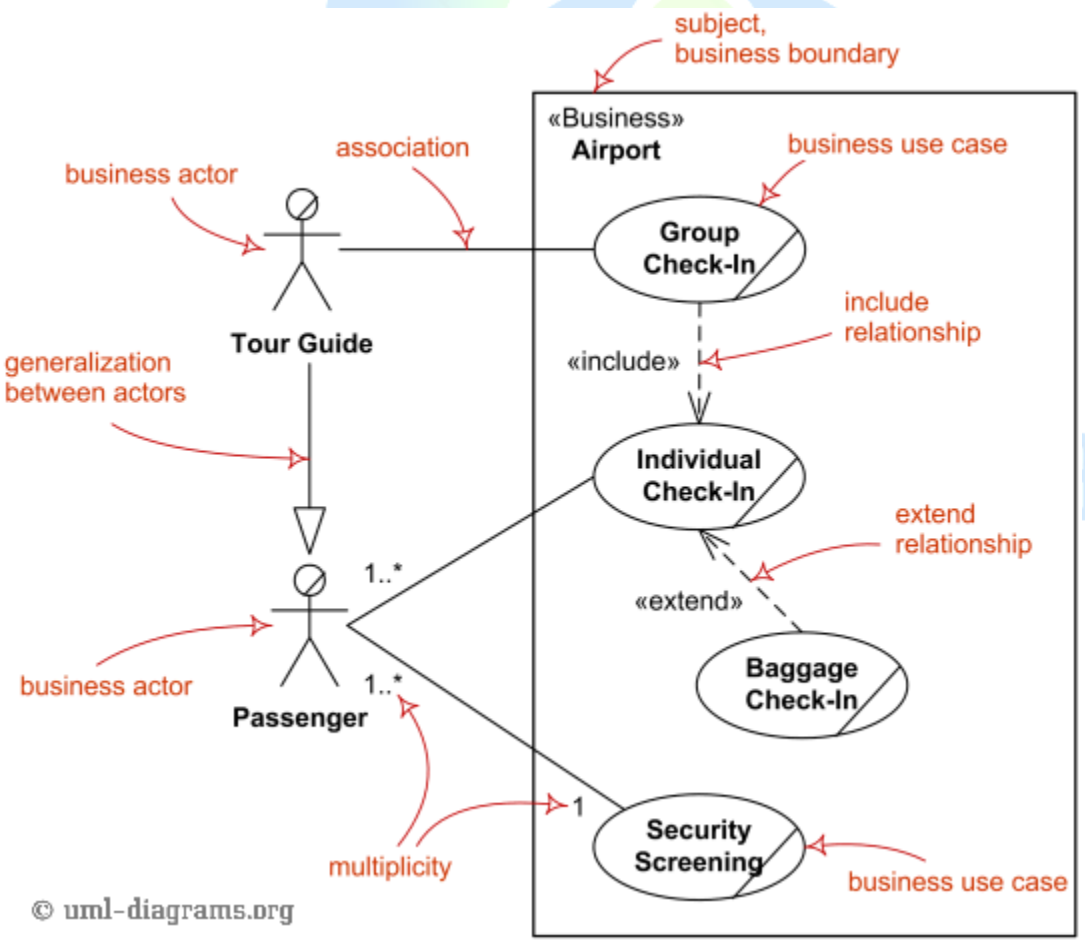
UML Use Case Diagrams

Use case diagrams are usually referred to as **behavior diagrams** used to describe a set of actions (**use cases**) that some system or systems (**subject**) should or can perform in collaboration with one or more **external users** of the system (**actors**). Each use case should provide some observable and valuable result to the actors or other stakeholders of the system. Note, that UML 2.0 to 2.4 specifications also described **use case diagram** as a specialization of a **class diagram**, and class diagram is a **structure diagram**.

Use case diagrams are in fact twofold - they are both **behavior diagrams**, because they describe behavior of the system, and they are also **structure diagrams** - as a special case of class diagrams where classifiers are restricted to be either **actors** or **use cases** related to each other with **associations**.

Business Use Case Diagrams

While support for **business modeling** was declared as one of the goals of the UML, UML specification provides **no notation** specific to business needs.
Business use cases were introduced in **Rational Unified Process** (RUP) to represent business function, process, or activity performed in the modeled **business**. A **business actor** represents a role played by some person or system external to the modeled business, and interacting with the business. Business use case should produce a result of observable value to a business actor.
Major elements of the business use case diagram are shown on the picture below. Note again, both business use case as well as business actor are not defined in UML standard, so you will either need to use some UML tool supporting those or create your own business modeling stereotypes.



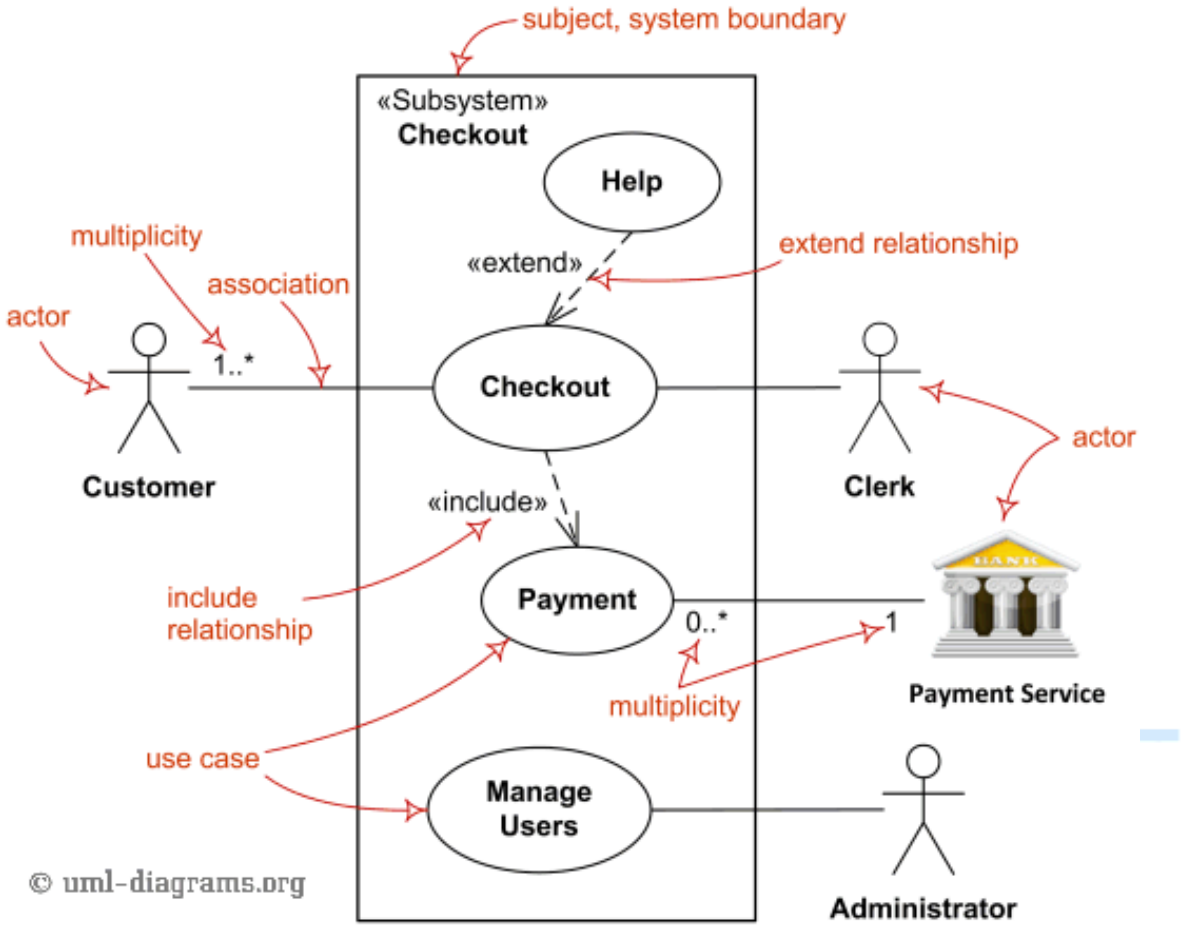
*Major elements of business use case diagram - **business actor**, **business use case**, **business boundary**, **include** and **extend** relationships.*

System Use Case Diagrams

(System) Use case diagrams are used to specify:

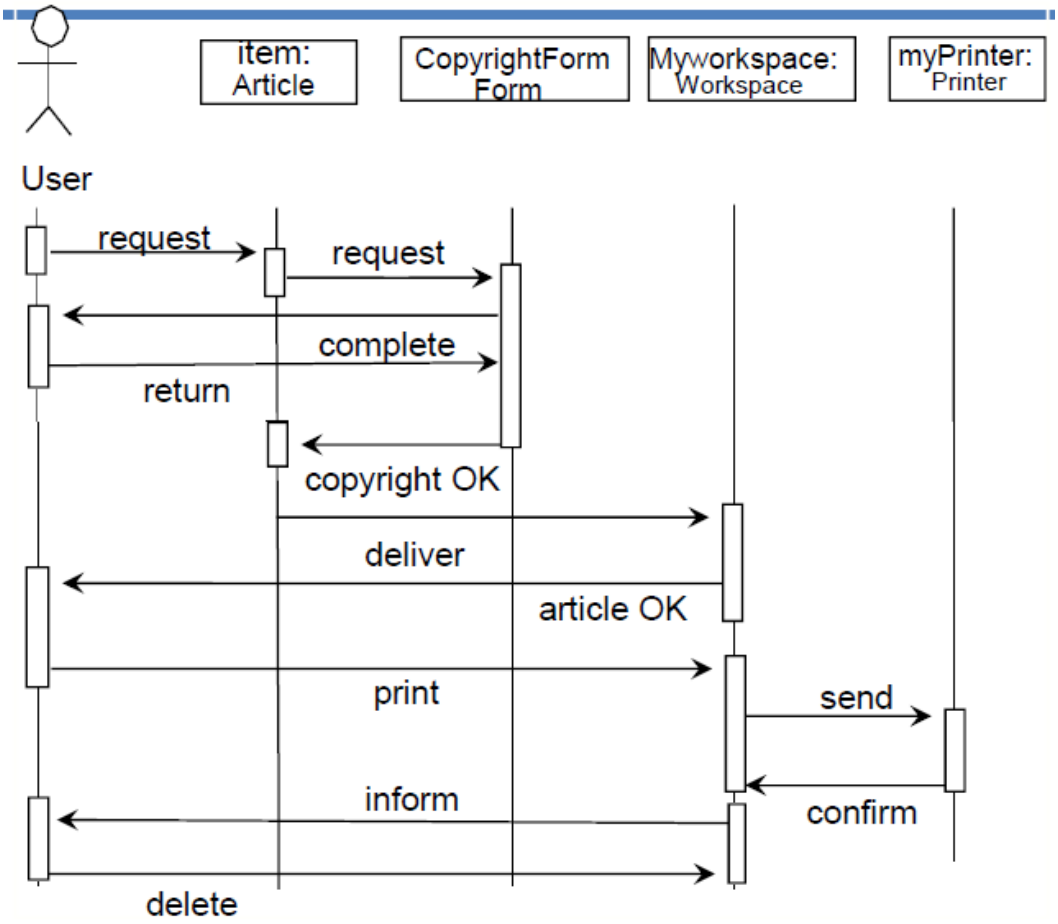
- (external) **requirements**, required usages of a system under design or analysis (**subject**) - to capture what the system is supposed to do;
- the **functionality** offered by a subject – what the system can do;
- requirements the specified subject poses on its **environment** - by defining how environment should interact with the subject so that it will be able to perform its services.

Major elements of the UML use case diagram are shown on the picture below.



*Major elements of UML use case diagram - **actor**, **use case**, **subject**, **include** and **extend** relationships.*

System interactions for article Printing



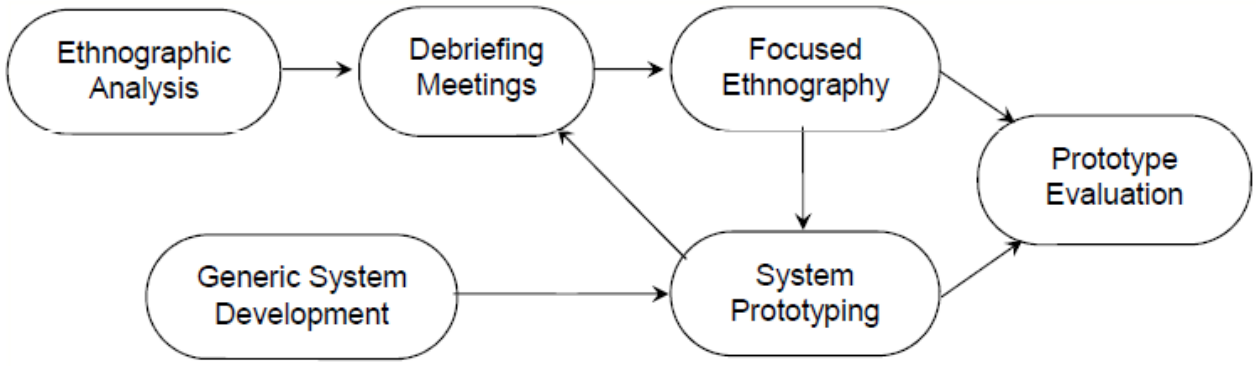
Ethnography

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

Focused ethnography

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

Ethnography and prototyping



Ethnography discovers 2 types of requirements

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

Requirements validation

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

Requirements checking

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

Requirements validation techniques

• Requirements reviews
  – Systematic manual analysis of the requirements.
• Prototyping
  – Using an executable model of the system to check requirements.
• Test-case generation
  – Developing tests for requirements to check testability.
  – If test is difficult or impossible to design for the requirement means it is difficult to implement that requirement

Requirements reviews

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

Review checks

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

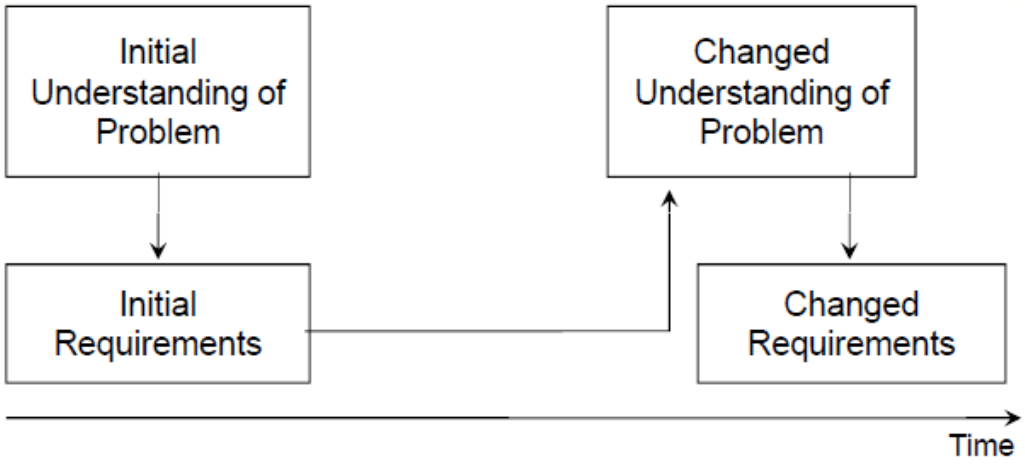
Requirements management

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

Requirements change

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

Requirements evolution



Enduring and volatile requirements

• Enduring requirements: Stable requirements derived from the core activity of the customer organisation. E.g. a hospital will always have doctors, nurses, etc. May be derived from domain models

• Volatile requirements: Requirements which change during development or when the system is in use. In a hospital, requirements derived from health-care policy

Requirements classification

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

Requirements management planning

• During the requirements engineering process, you have to plan:

– Requirements identification

• How requirements are individually identified;

– A change management process

• The process followed when analysing a requirements change;

– Traceability policies

• The amount of information about requirements relationships that is maintained;

– CASE tool support

• The tool support required to help manage requirements change;

Traceability

• Traceability is concerned with the relationships between requirements, their sources and the system design

• Source traceability

– Links from requirements to stakeholders who proposed these requirements;

• Requirements traceability

– Links between dependent requirements;

• Design traceability

– Links from the requirements to the design;

A traceability matrix

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

CASE tool support

• Requirements storage

– Requirements should be managed in a secure, managed data store.

• Change management

– The process of change management is a workflow process whose stages can be defined and information flow between these stages partially automated.

• Traceability management

– Automated retrieval of the links between requirements.

Requirements change management

• Should apply to all proposed changes to the requirements.

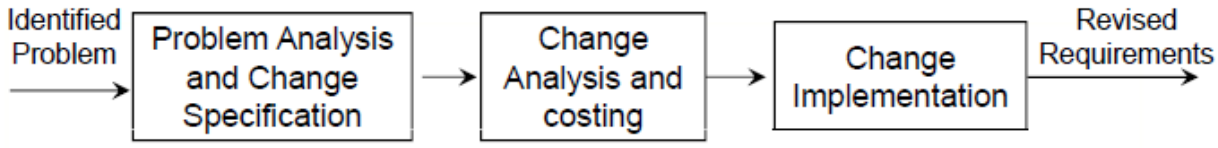
• Principal stages

– Problem analysis: Discuss requirements problem and propose change;

– Change analysis and costing: Assess effects of change on other requirements;

– Change implementation: Modify requirements document and other documents to reflect change.

Change management



System models

Objectives

- **To explain why the context of a system should be modelled as part of the RE process**
- **To describe behavioural modelling, data modelling and object modelling**
- **To show how CASE workbenches support system modelling**

System modelling

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

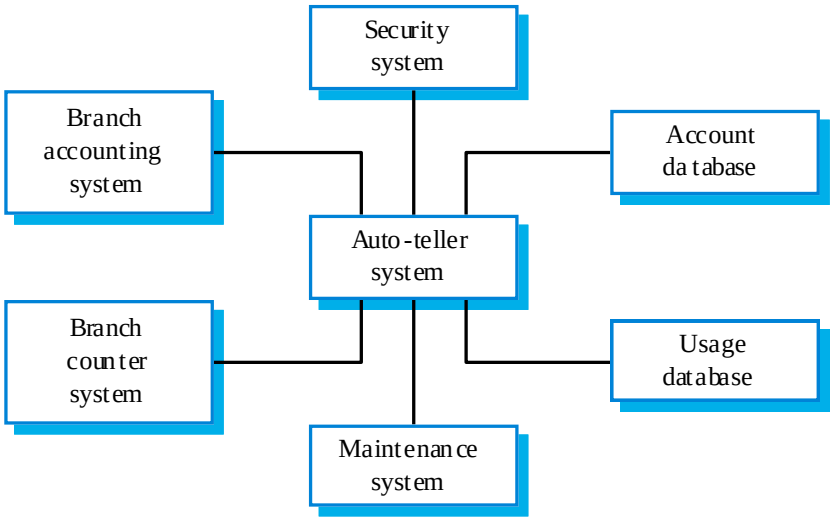
Model types

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

Context models

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

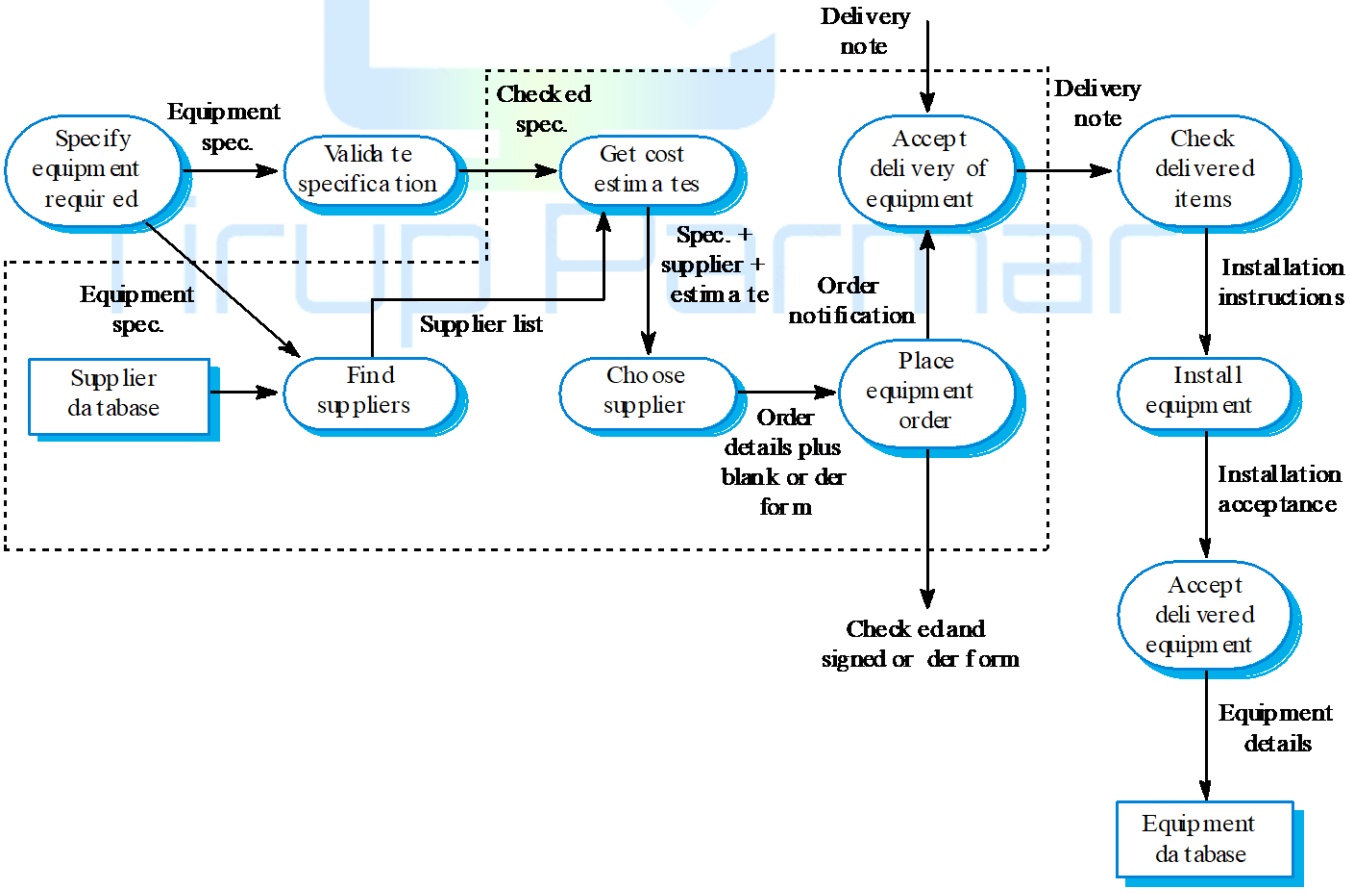
The context of an ATM system



Showing context in process models

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

Equipment procurement process



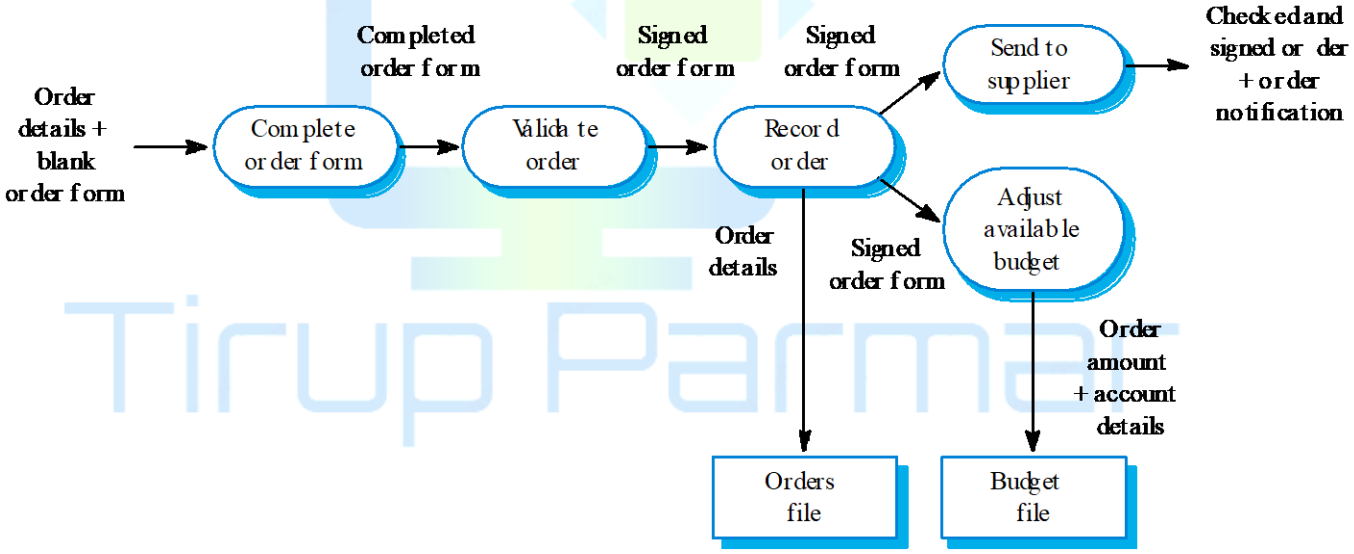
Behavioural models

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

Data-processing models

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

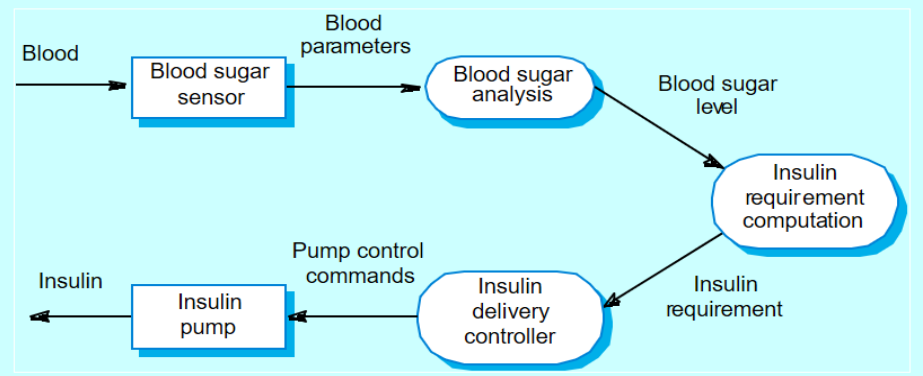
Order processing DFD



Data flow diagrams

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

Insulin pump DFD



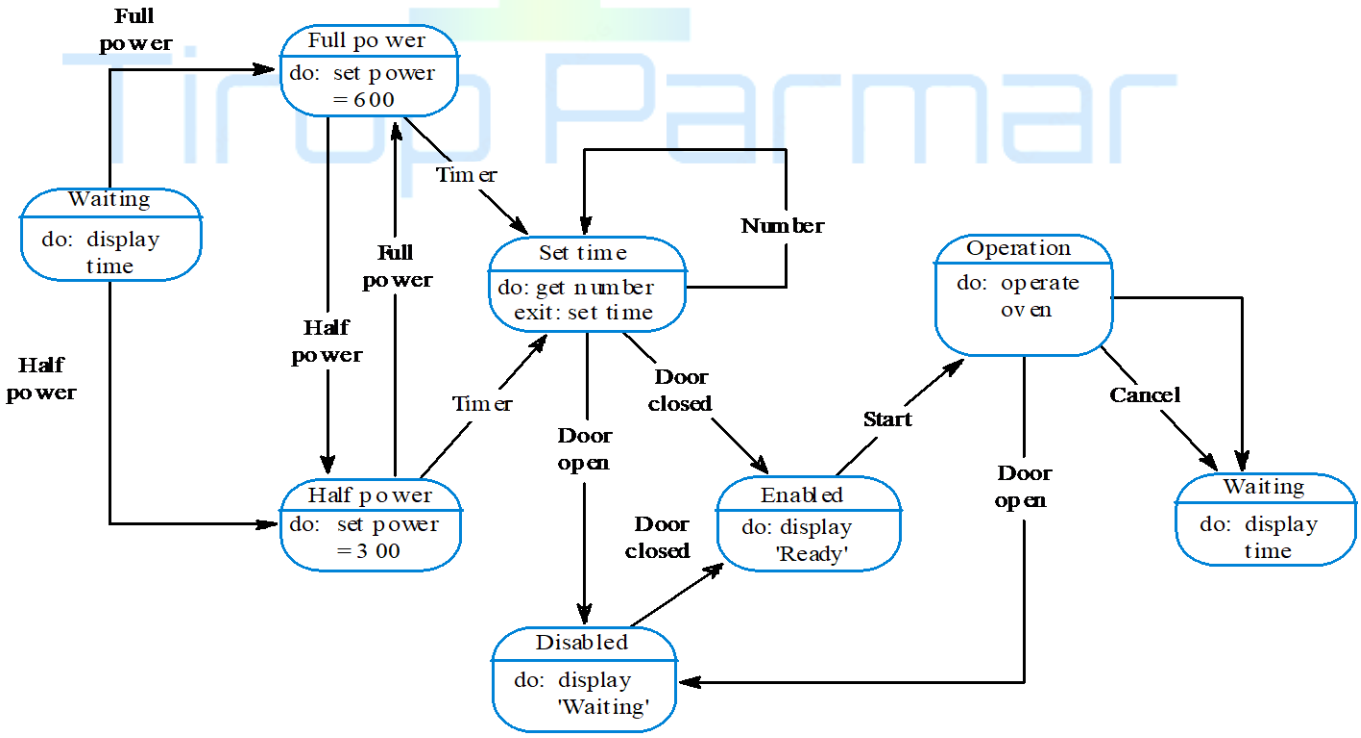
State machine models

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

Statecharts

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

Microwave oven model



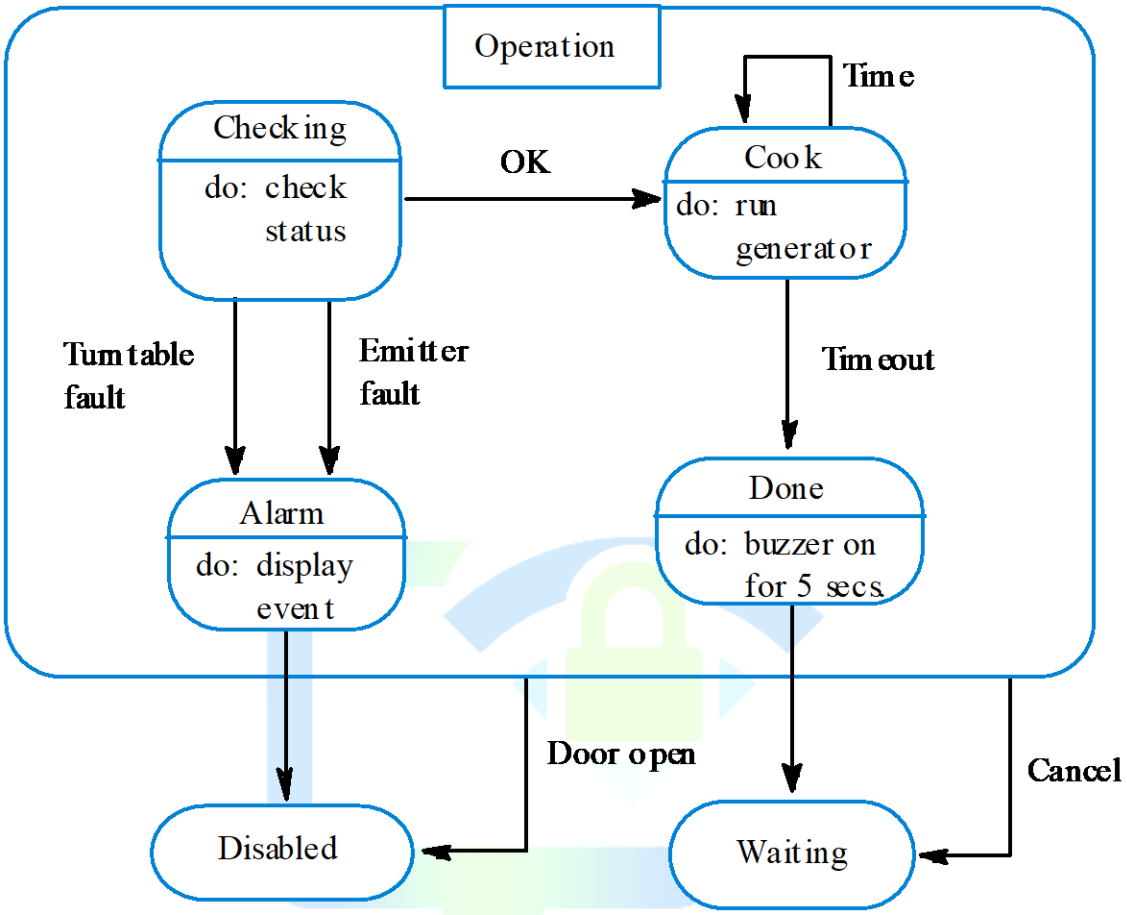
Microwave oven state description

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

Microwave oven stimuli

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

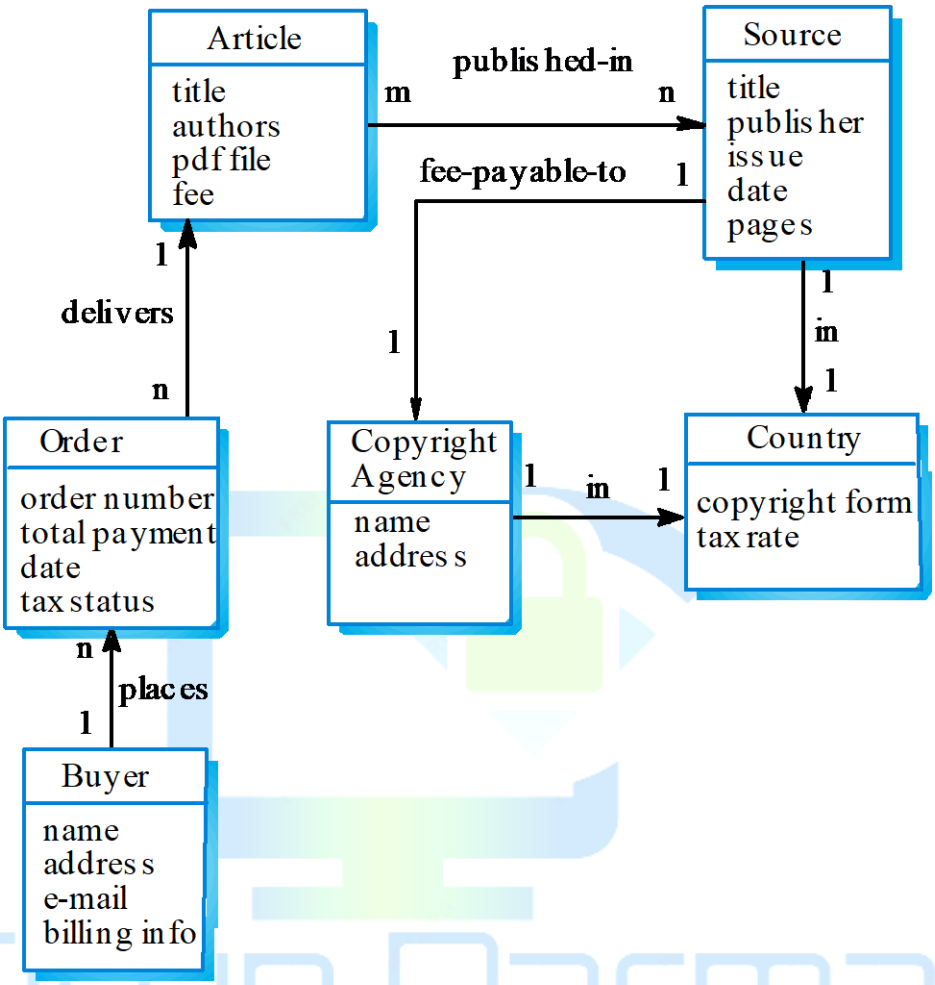
Microwave oven operation



Semantic data models

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

Library semantic model



Data dictionaries

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

Data dictionary entries

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

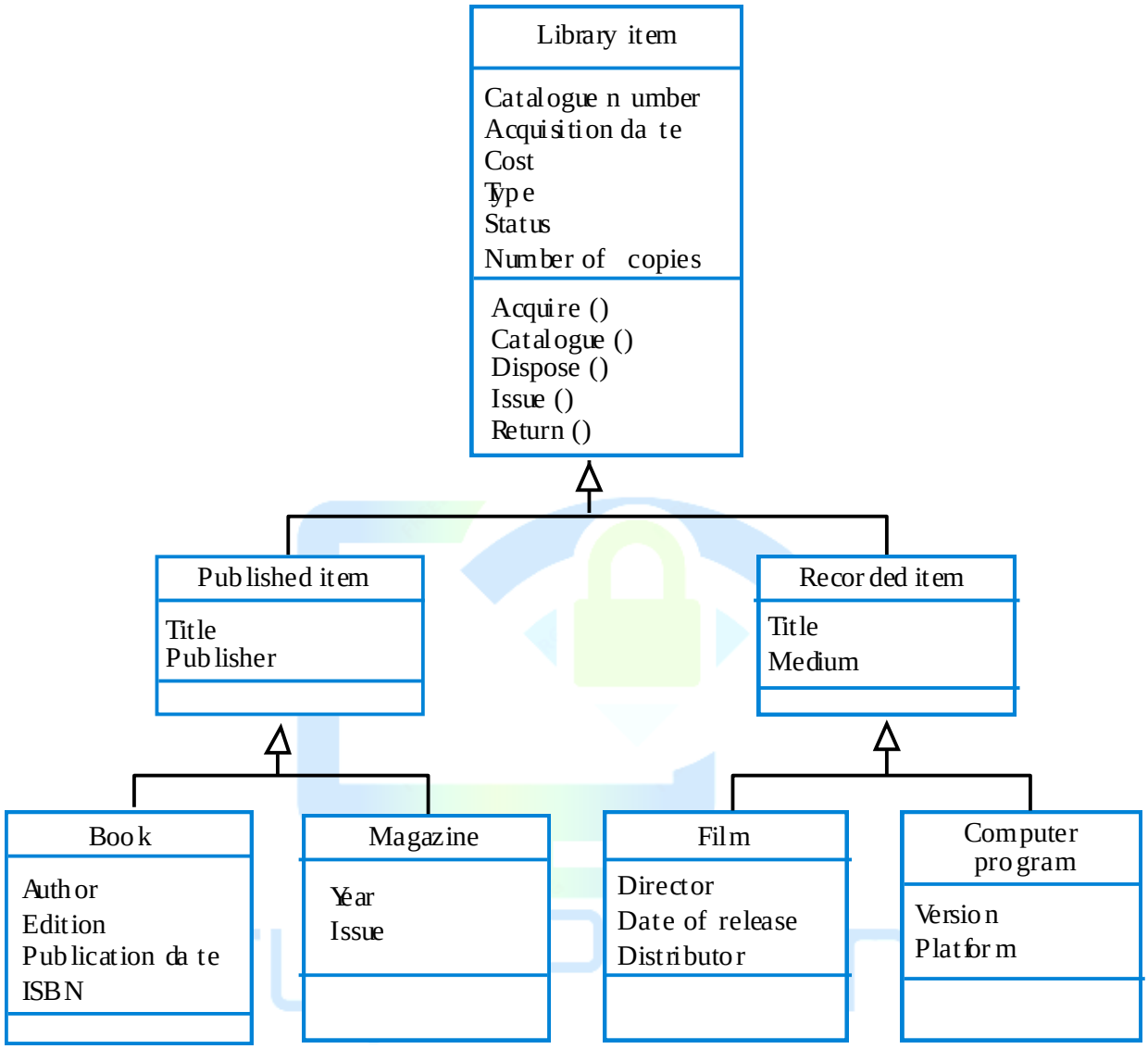
Object models

- Object models describe the system in terms of object classes and their associations.
- The semantic data model is a specialized form of object model.
- An object class is an abstraction over a set of objects with common attributes and the services (operations) provided by each object.
- Various object models may be produced
    1. Inheritance models;
    2. Aggregation models;
    3. Interaction models.
- Natural ways of reflecting the real-world entities manipulated by the system
- Object class identification is recognised as a difficult process requiring a deep understanding of the application domain
- Object classes reflecting domain entities are reusable across systems

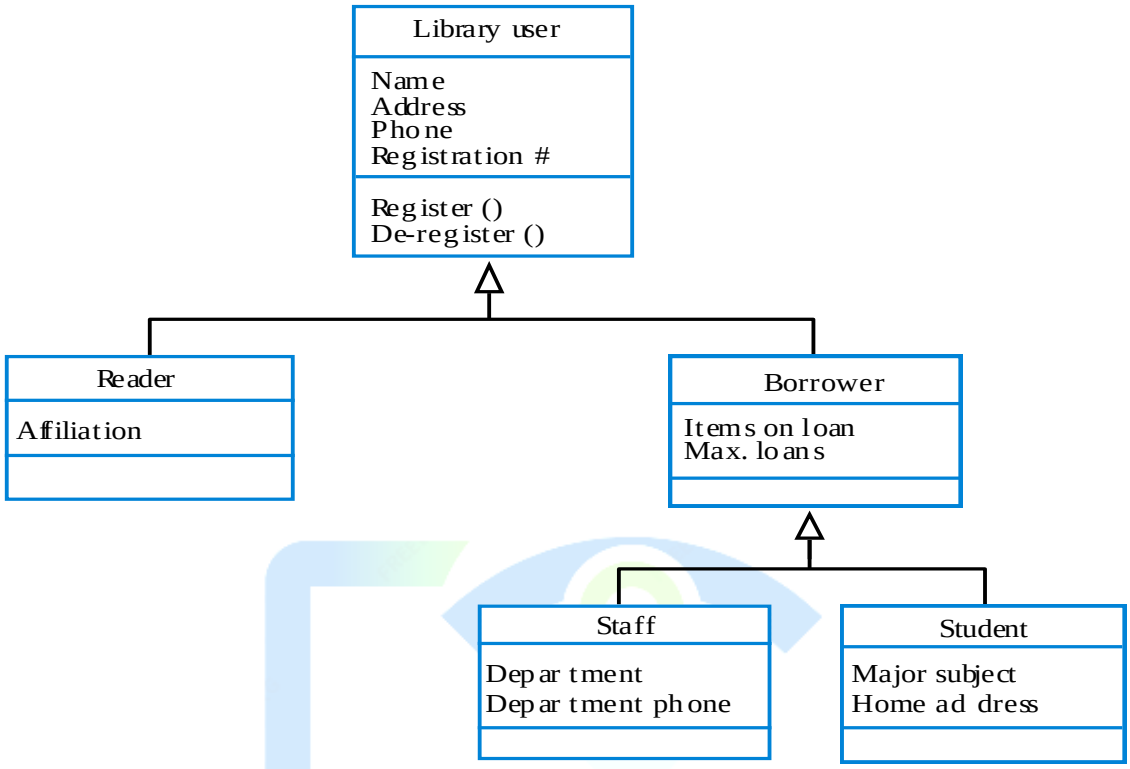
1. Inheritance models

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

Library class hierarchy

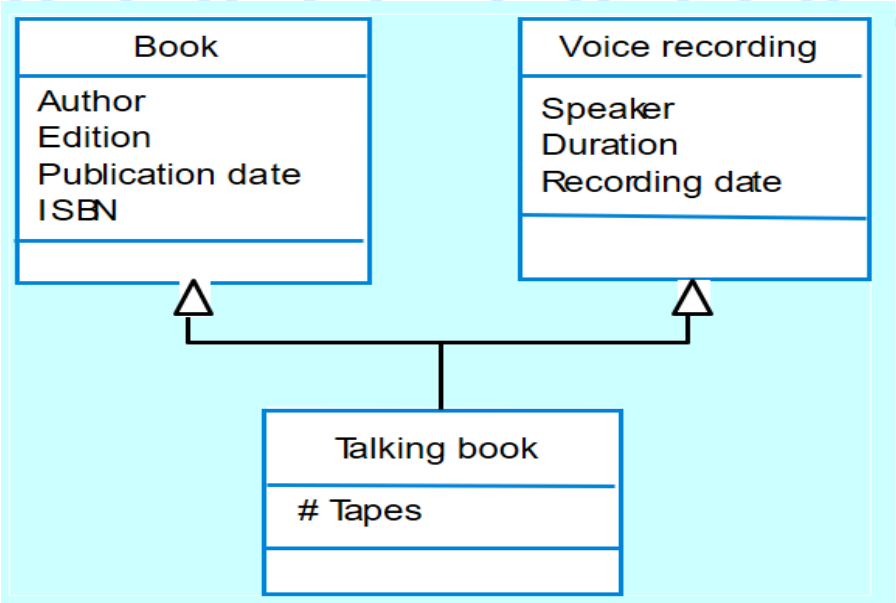


User class hierarchy



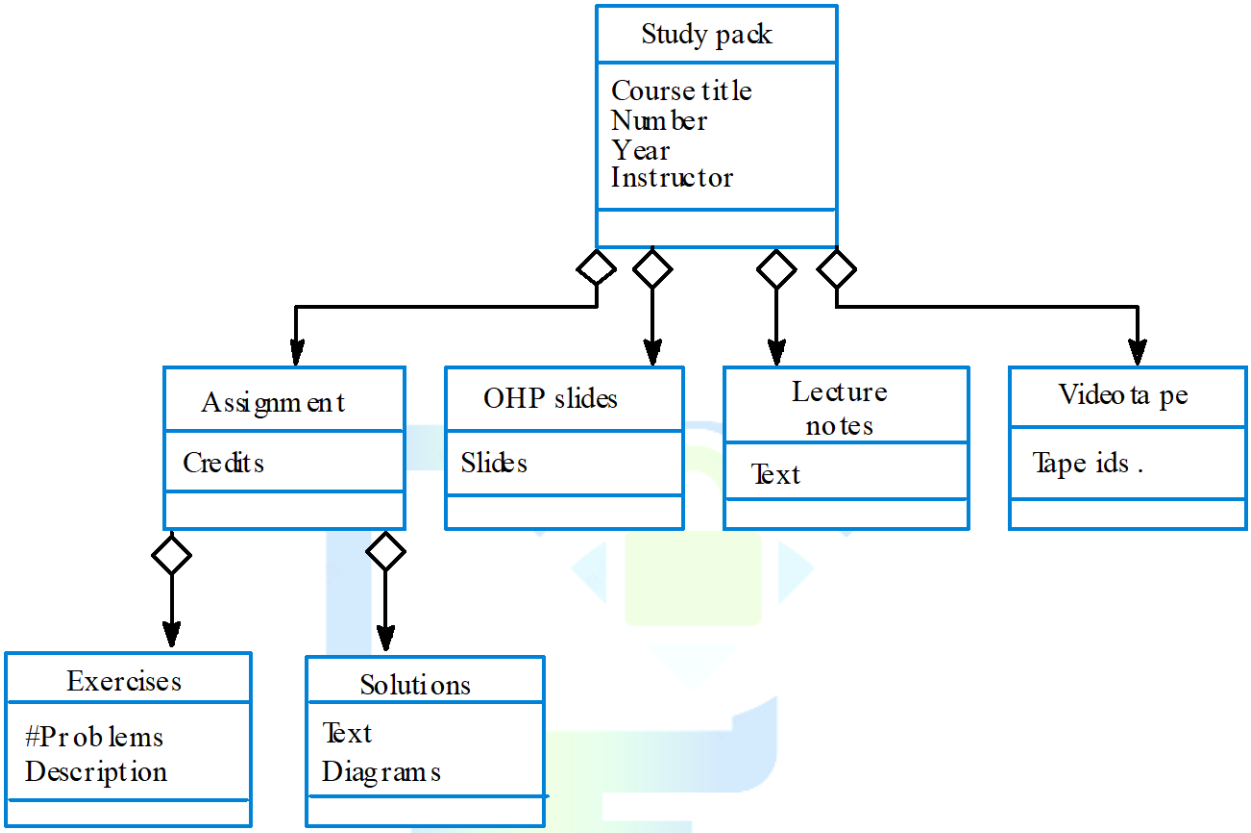
1.1. Multiple inheritance

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



Object aggregation

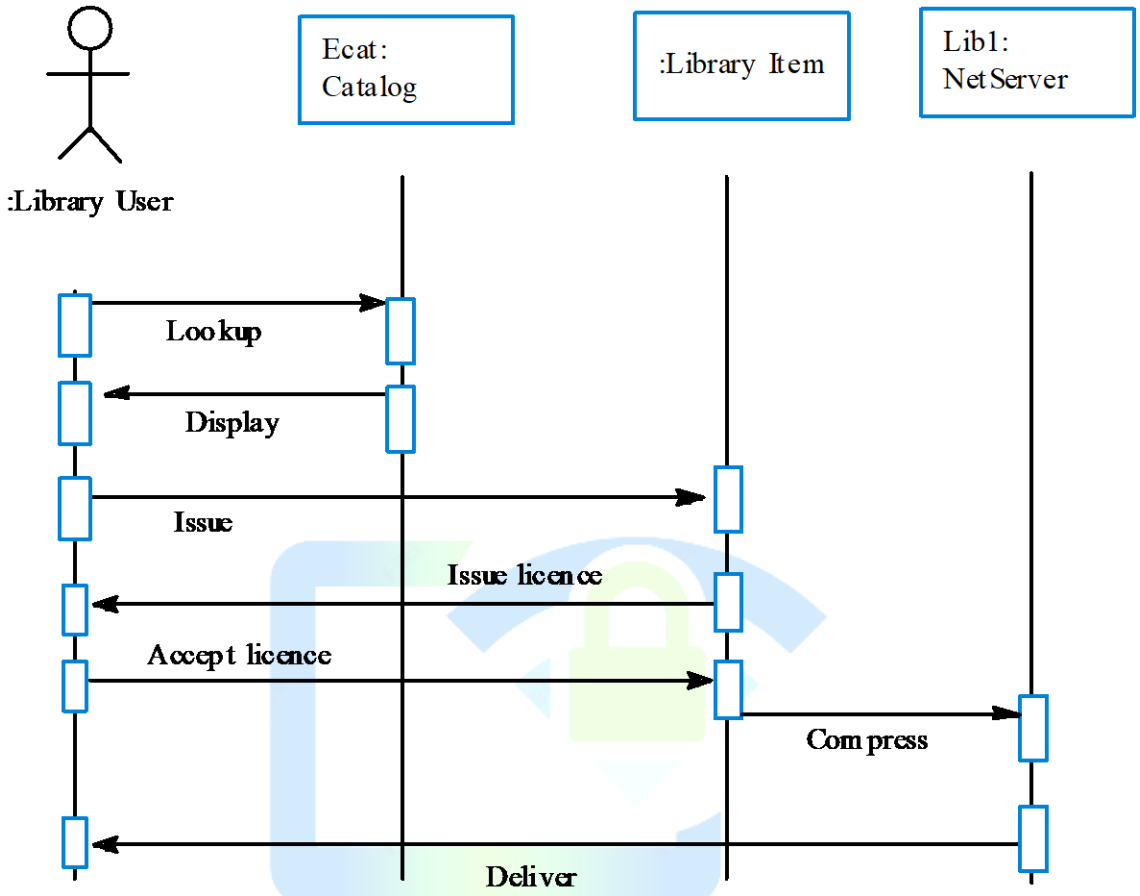
- An aggregation model shows how classes that are collections are composed of other classes.
- Aggregation models are similar to the part-of relationship in semantic data models.



Object behaviour modelling

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

Issue of electronic items



An object-oriented method for requirements engineering

- Elicit requirements
  - Identify viewpoints
  - Conduct interviews and ethnographies
  - Draw up scenarios and use cases
- Refine use cases with functional decomposition and identify objects involved in interactions
- Build object model
- Create sequence diagrams for each use case
- Create state diagrams for each object
- Group related objects into subsystems

Putting it all together

- Preface
- Introduction
- Glossary
- User requirements definition
  - Initial requirements from the user (functional and nonfunctional) in natural language
- System architecture
  - The initial subsystem decomposition
  - A description of each subsystem's purposes and services offered
  - Used to organize the system requirements
- System requirements specification
  - Derive from user requirements, use case models, and other system models a list of requirements in natural language
  - These requirements are a refinement of the user requirements.
  - These can also specify requirements for each of the subsystems.
- System models
  - From previous slide
- System evolution
- Appendices
- Index

Structured methods

- Structured methods
  - Guide software development from requirements to implementation by stepwise refinement
  - Incorporate system modelling as an inherent part of the method.
- Methods
  - a set of models
  - a process for deriving these models
  - rules and guidelines that should apply to the models.
- CASE tools support system modelling as part of a structured method.

Method weaknesses

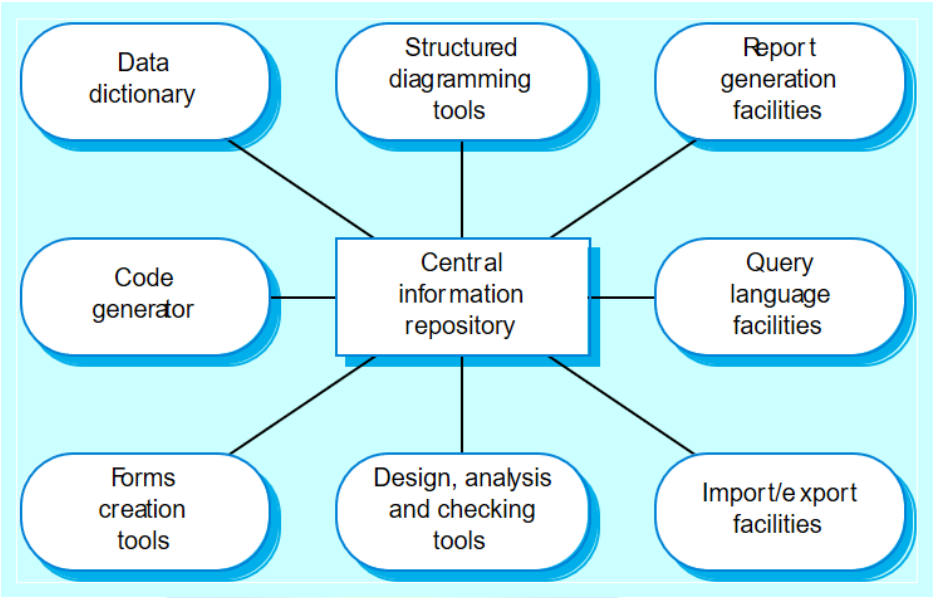
- They do not model non-functional system requirements.
- They do not usually include information about whether a method is appropriate for a given problem.
- They may produce too much documentation.
- The system models are sometimes too detailed and difficult for users to understand.

CASE workbenches

- A coherent set of tools that is designed to support related software process activities such as analysis, design or testing.

- Analysis and design workbenches support system modelling during both requirements engineering and system design.
- Support iterations of modelling activities.
- These workbenches may support a specific design method or may provide support for a creating several different types of system model.

An analysis and design workbench



Analysis workbench components

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

Key points

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

- Object models describe logical system entities, their classification and aggregation.
- Sequence models show the interactions between actors and the system objects that they use.
- Structured methods provide a framework for developing system models.

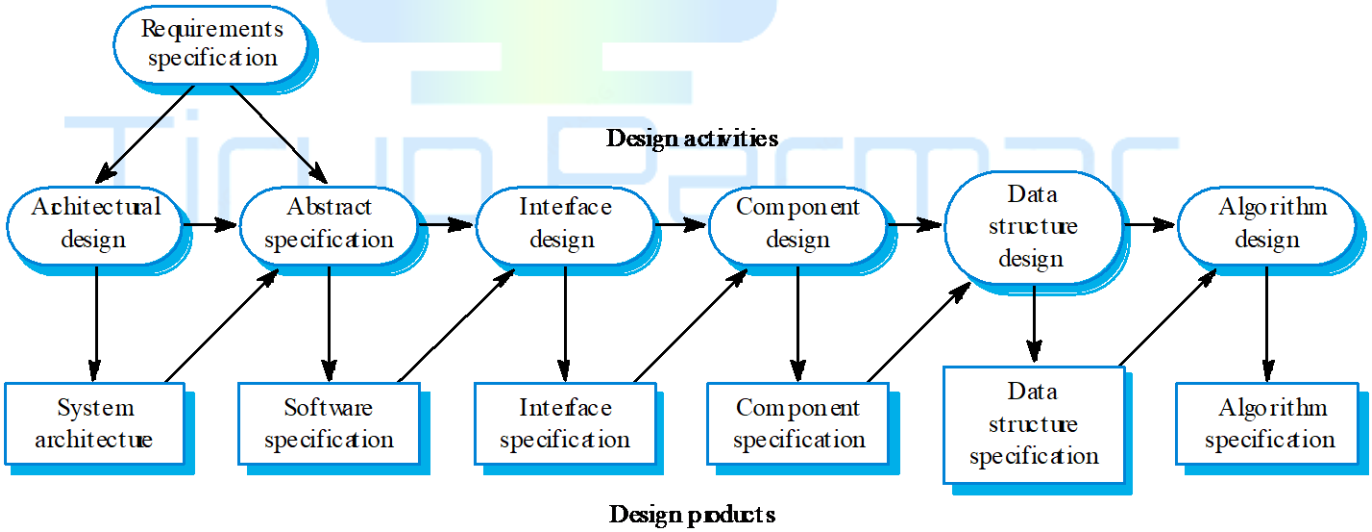
UNIT III

Architectural Design

Objectives

- To introduce architectural design and to discuss its importance
- To explain the architectural design decisions that have to be made
- To introduce three complementary architectural styles covering organisation, decomposition and control
- To discuss reference architectures are used to communicate and compare architectures

The software design process



Software architecture

- The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is architectural design.
- The output of this design process is a description of the software architecture.

Architectural design

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

Advantages of explicit architecture

- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
  - The architecture may be reusable across a range of systems.

Architecture and system characteristics

- Performance
  - Localise critical operations and minimise communications. Use large rather than fine-grain components.
- Security
  - Use a layered architecture with critical assets in the inner layers.
- Safety
  - Localise safety-critical features in a small number of sub-systems.
- Availability
  - Include redundant components and mechanisms for fault tolerance.
- Maintainability
  - Use fine-grain, replaceable components.

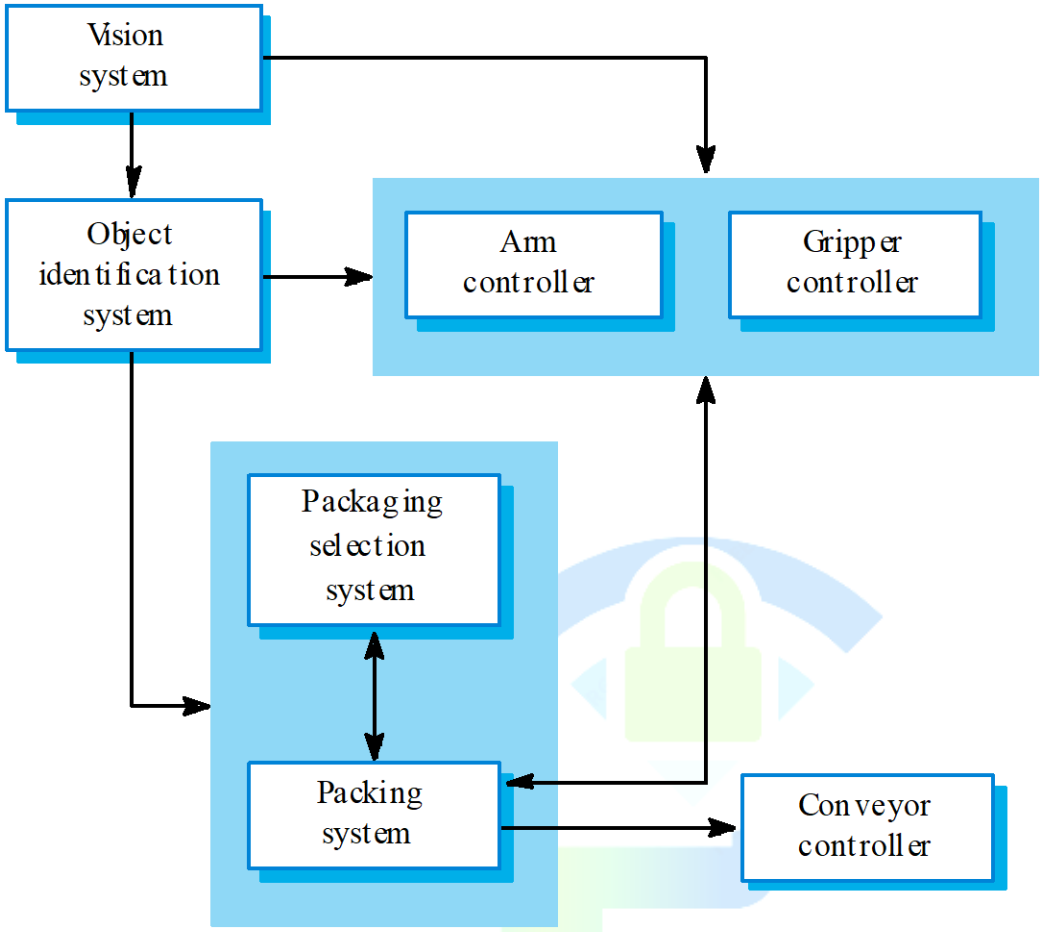
Architectural conflicts

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

Subsystem decomposition

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

Packing robot control system



Box and line diagrams

- Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- However, useful for communication with stakeholders and for project planning.

Architectural design decisions

- Architectural design is a creative process so the process differs depending on the type of system being developed.
- However, a number of common decisions span all design processes.
- Is there a generic application architecture that can be reused? How will the subsystems be distributed?
    - Distributed architectures
    - Application architectures
    - Product line architectures
    - Reference architectures

- What architectural styles are appropriate? What approach will be used to structure the system?
    - Repository, client-server, layered
- How will the system be decomposed into modules?
    - Object-oriented decomposition
    - Functional/dataflow decomposition
- What control strategy should be used?
    - Centralized, event-driven
- How will the architectural design be evaluated?
- How should the architecture be documented?

Outline

- Architecture reuse
    - Application architectures
    - Reference architectures
    - Product line architectures
- Architectural styles
    - Repository, client-server, layered
- Modular decomposition styles
    - Object-oriented decomposition, pipes and filters
- Control strategies
    - Centralised, event-driven
    - Distributed systems

Architecture reuse

- Systems in the same domain often have similar architectures that reflect domain concepts.
- Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- Application architectures are covered in Chapter 13 and product lines in Chapter 18.

Architectural models

- Used to document an architectural design.
- Static structural model that shows the major system components.
- Dynamic process model that shows the process structure of the system.
- Interface model that defines sub-system interfaces.
- Relationships model such as a data-flow model that shows sub-system relationships.
- Distribution model that shows how sub-systems are distributed across computers.

System organisation

Architectural styles

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

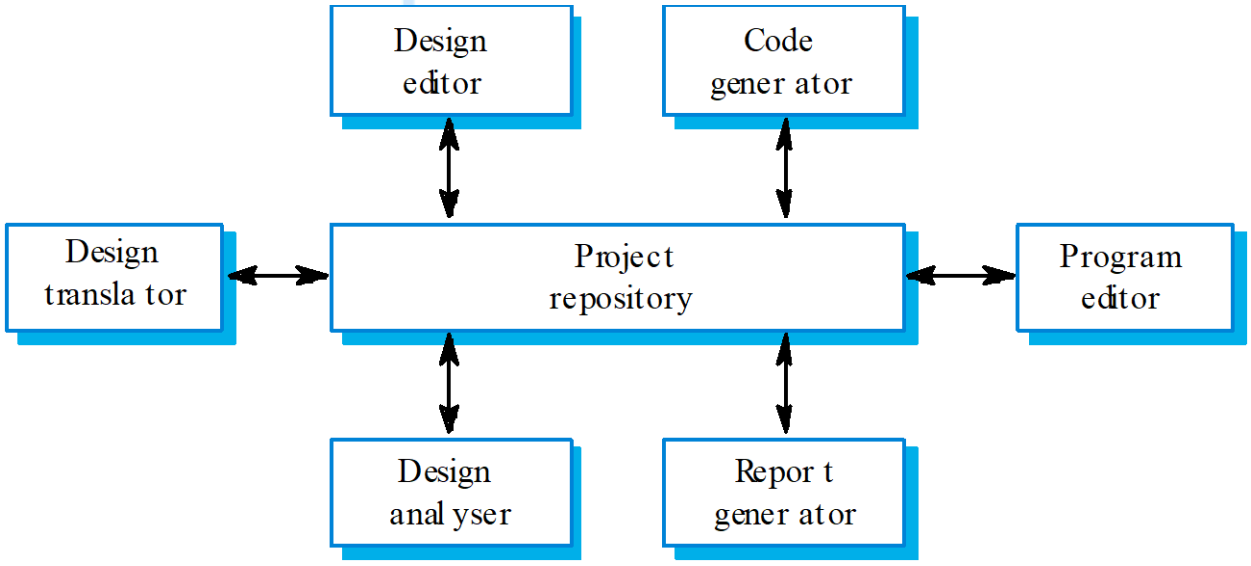
System organization

- Reflects the basic strategy that is used to organize a system.
- Three architectural styles are widely used:
  - A shared data repository style;
  - A shared services and servers style;
  - An abstract machine or layered style.

The repository model

- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used.

CASE toolset architecture



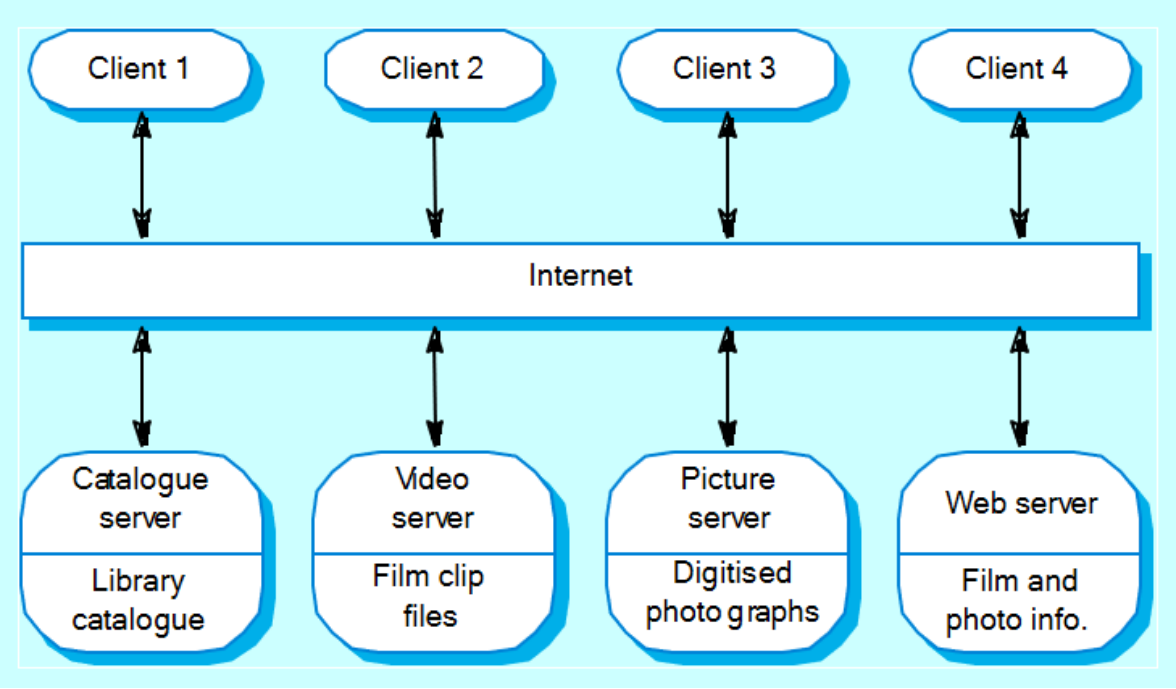
Repository model characteristics

- Advantages
  - Efficient way to share large amounts of data;
  - Sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security, etc.
  - Sharing model is published as the repository schema.
- Disadvantages
  - Sub-systems must agree on a repository data model. Inevitably a compromise;
  - Data evolution is difficult and expensive;
  - No scope for specific management policies;
  - Difficult to distribute efficiently.

Client-server model

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

Film and picture library



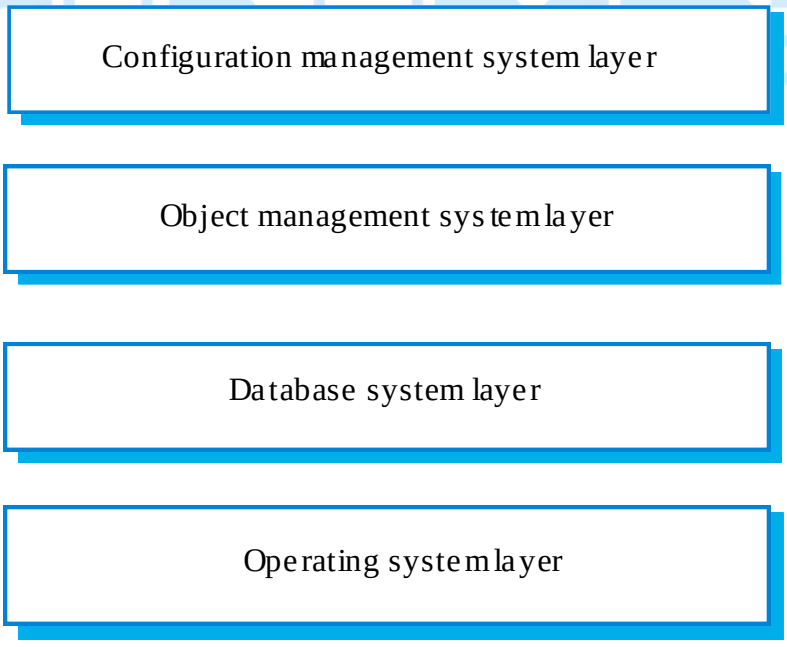
Client-server characteristics

- Advantages
  - Distribution of data is straightforward;
  - Makes effective use of networked systems. May require cheaper hardware;
  - Easy to add new servers or upgrade existing servers.
- Disadvantages
  - No shared data model so sub-systems use different data organisation. Data interchange may be inefficient;
  - Redundant management in each server;
  - No central register of names and services - it may be hard to find out what servers and services are available.

Abstract machine (layered) model

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often difficult to structure systems in this way. New services may require changes that cut across multiple layers.
- Performance can be a problem as a request may have to go through several layers before being processed.

Version management system



Decomposition styles

Modular decomposition styles

- Styles of decomposing sub-systems into modules.
- No rigid distinction between system organisation and modular decomposition.

Sub-systems and modules

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

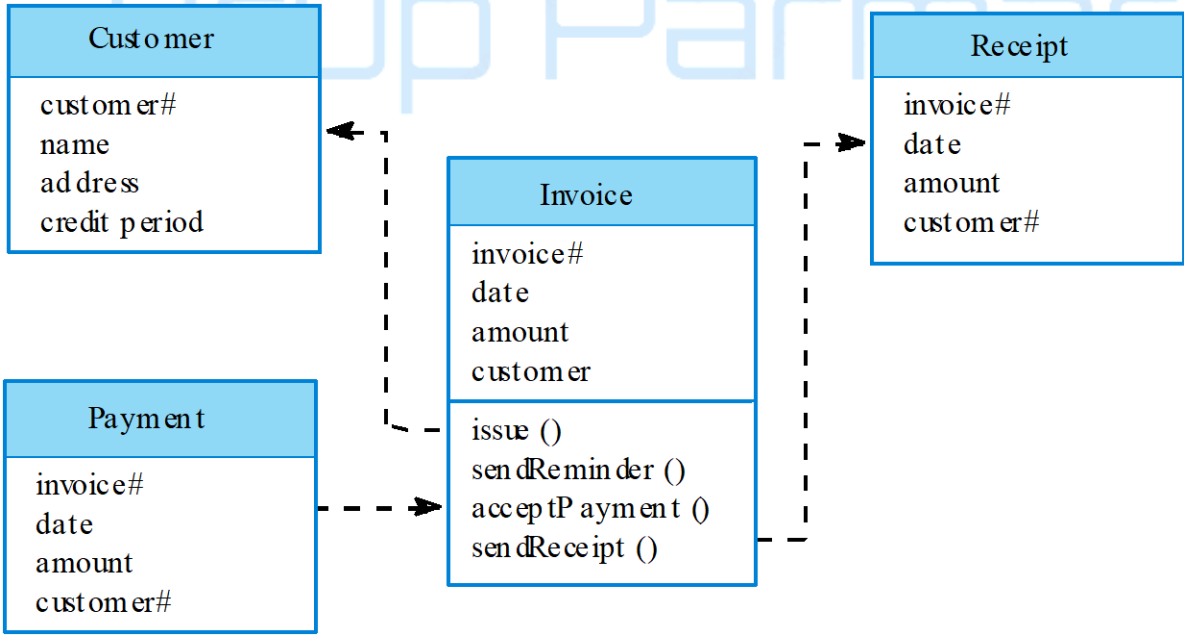
Modular decomposition

- Another structural level where sub-systems are decomposed into modules.
- Two modular decomposition models covered
  - An object model where the system is decomposed into interacting object;
  - A pipeline or data-flow model where the system is decomposed into functional modules which transform inputs to outputs.
- If possible, decisions about concurrency should be delayed until modules are implemented.

Object models

- Structure the system into a set of loosely coupled objects with well-defined interfaces.
- Object-oriented decomposition is concerned with identifying object classes, their attributes and operations.
- When implemented, objects are created from these classes and some control model used to coordinate object operations.

Invoice processing system



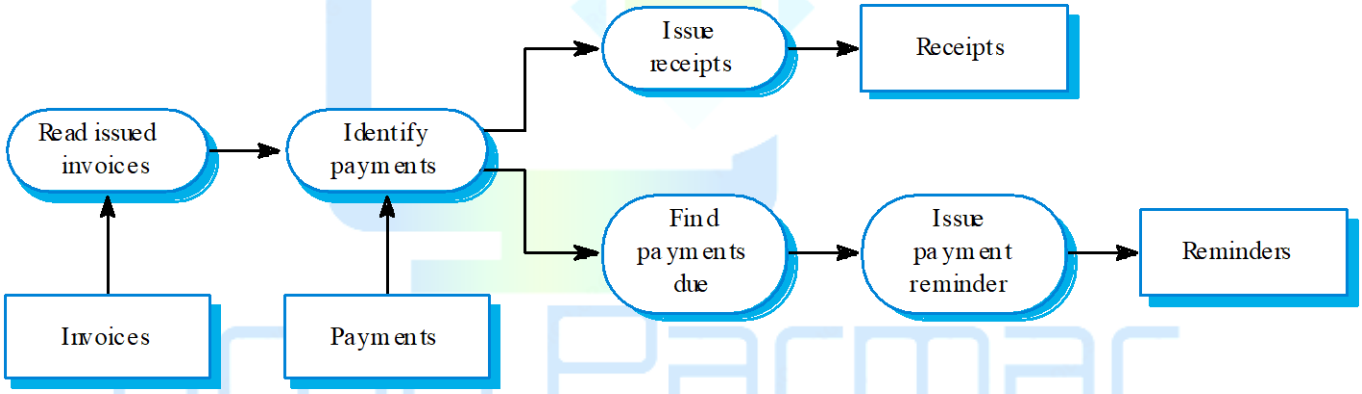
Object model advantages and disadvantages

- Objects are loosely coupled so their implementation can be modified without affecting other objects.
- The objects may reflect real-world entities.
- OO implementation languages are widely used.
- However, object interface changes may cause problems and complex entities may be hard to represent as objects.

Function-oriented pipelining

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

Invoice processing system



Pipeline model advantages and disadvantages

- Supports transformation reuse.
- Intuitive organisation for stakeholder communication.
- Easy to add new transformations.
- Relatively simple to implement as either a concurrent or sequential system.
- However, requires a common format for data transfer along the pipeline and difficult to support event-based interaction.

Control styles

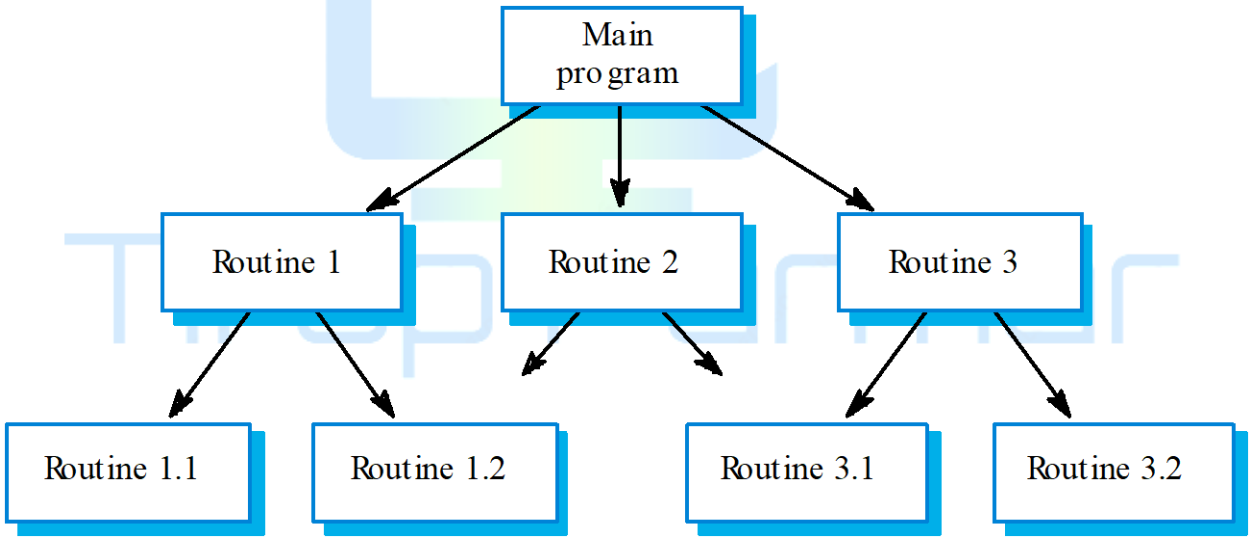
- Are concerned with the control flow between sub-systems. Distinct from the system decomposition model.

- Centralised control
  - One sub-system has overall responsibility for control and starts and stops other sub-systems.
- Event-based control
  - Each sub-system can respond to externally generated events from other sub-systems or the system's environment.

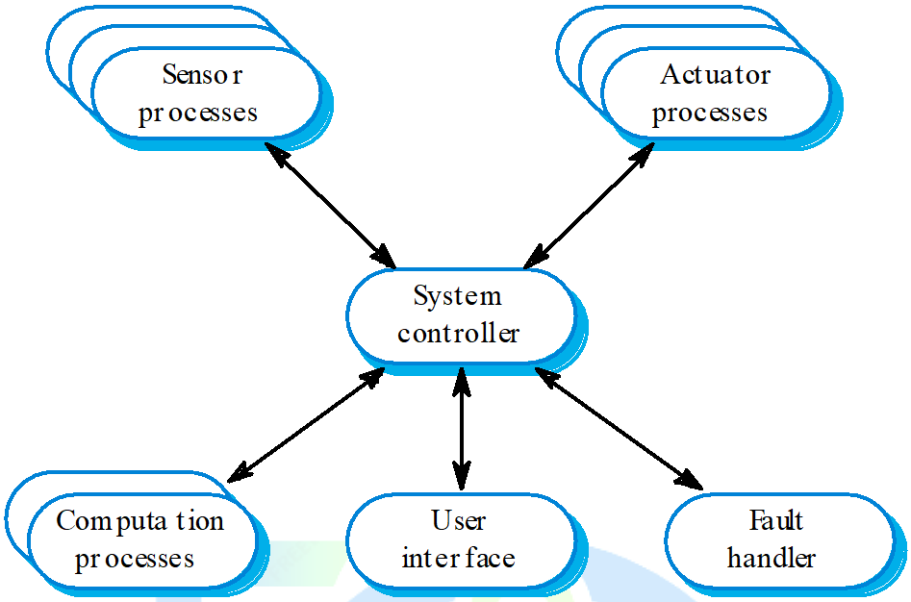
Centralised control

- A control sub-system takes responsibility for managing the execution of other sub-systems.
- Call-return model
  - Top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards. Applicable to sequential systems.
- Manager model
  - Applicable to concurrent systems. One system component controls the stopping, starting and coordination of other system processes. Can be implemented in sequential systems as a case statement.

Call-return model



Real-time system control



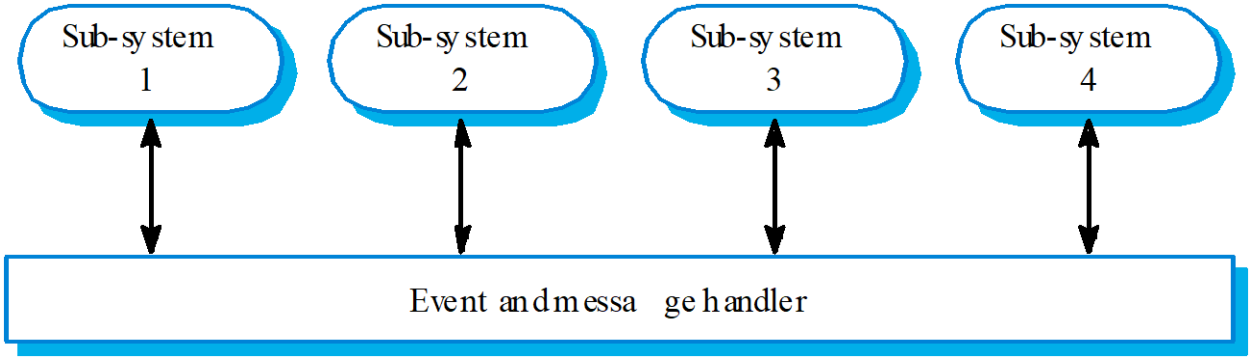
Event-driven systems

- Driven by externally generated events where the timing of the event is outwith the control of the sub-systems which process the event.
- Two principal event-driven models
  - Broadcast models. An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so;
  - Interrupt-driven models. Used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing.
- Other event driven models include spreadsheets and production systems.

Broadcast model

- Effective in integrating sub-systems on different computers in a network.
- Sub-systems register an interest in specific events. When these occur, control is transferred to the sub-system which can handle the event.
- Control policy is not embedded in the event and message handler. Sub-systems decide on events of interest to them.
- However, sub-systems don't know if or when an event will be handled.

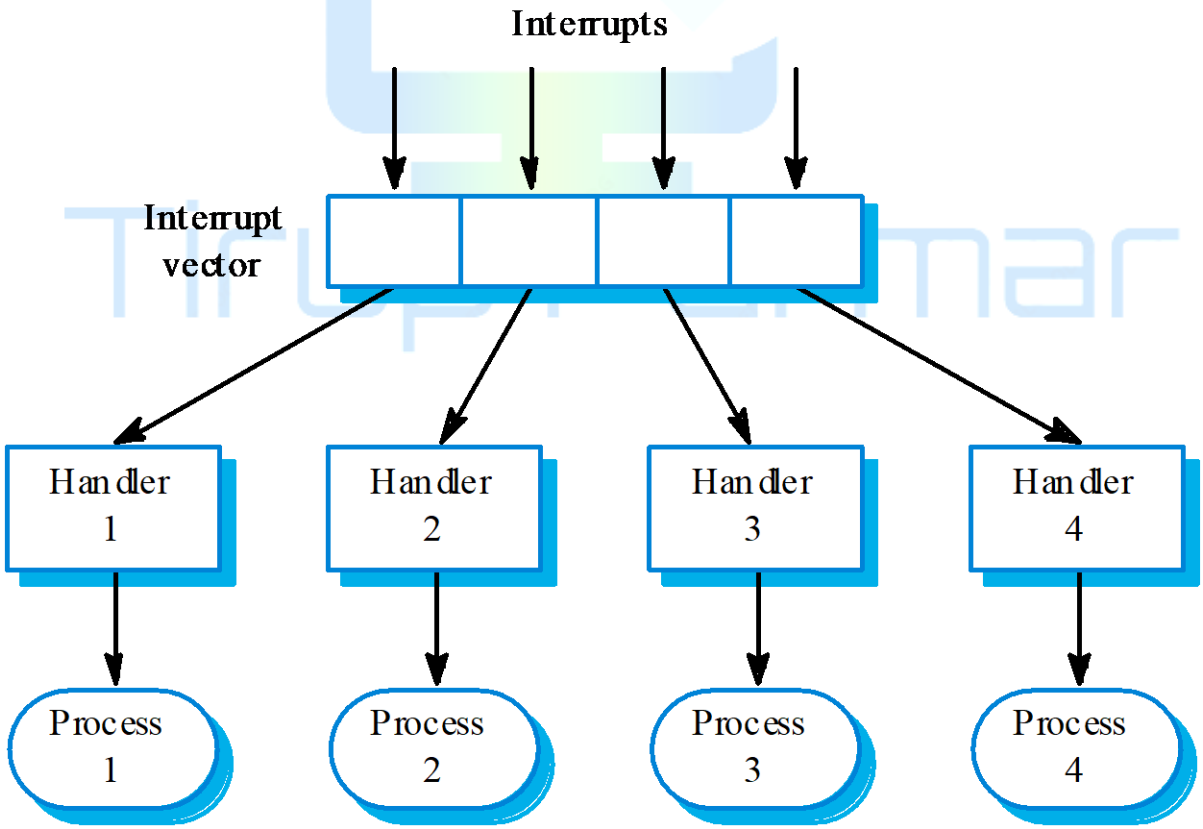
Selective broadcasting



Interrupt-driven systems

- Used in real-time systems where fast response to an event is essential.
- There are known interrupt types with a handler defined for each type.
- Each type is associated with a memory location and a hardware switch causes transfer to its handler.
- Allows fast response but complex to program and difficult to validate.

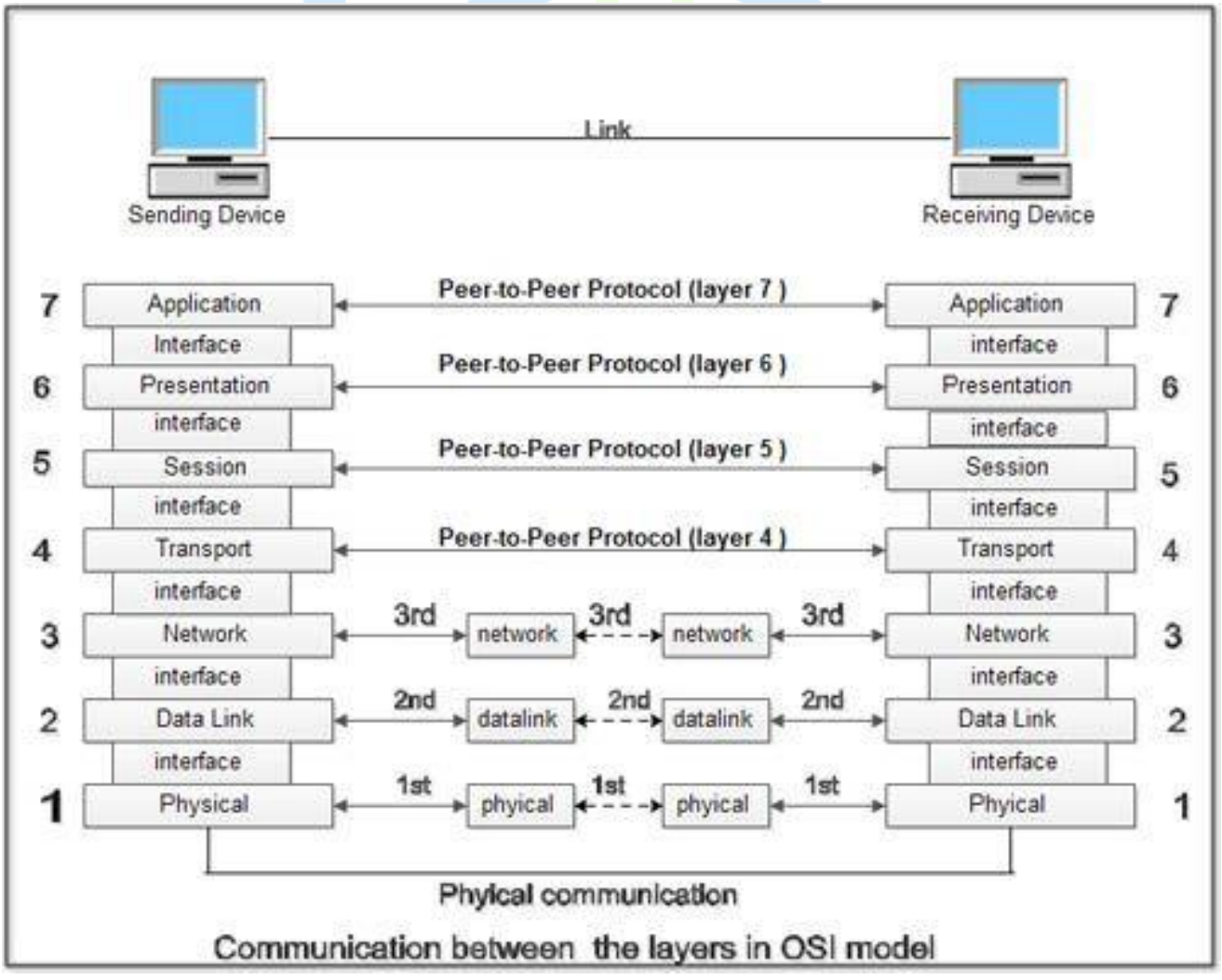
Interrupt-driven control



Reference architectures

- Architectural models may be specific to some application domain.
- Two types of domain-specific model
  - Generic models which are abstractions from a number of real systems and which encapsulate the principal characteristics of these systems.
  - Reference models which are more abstract, idealised model. Provide a means of information about that class of system and of comparing different architectures.
- Generic models are usually bottom-up models; Reference models are top-down models.
- Reference models are derived from a study of the application domain rather than from existing systems.
- May be used as a basis for system implementation or to compare different systems. It acts as a standard against which systems can be evaluated.
- OSI model is a layered model for communication systems.

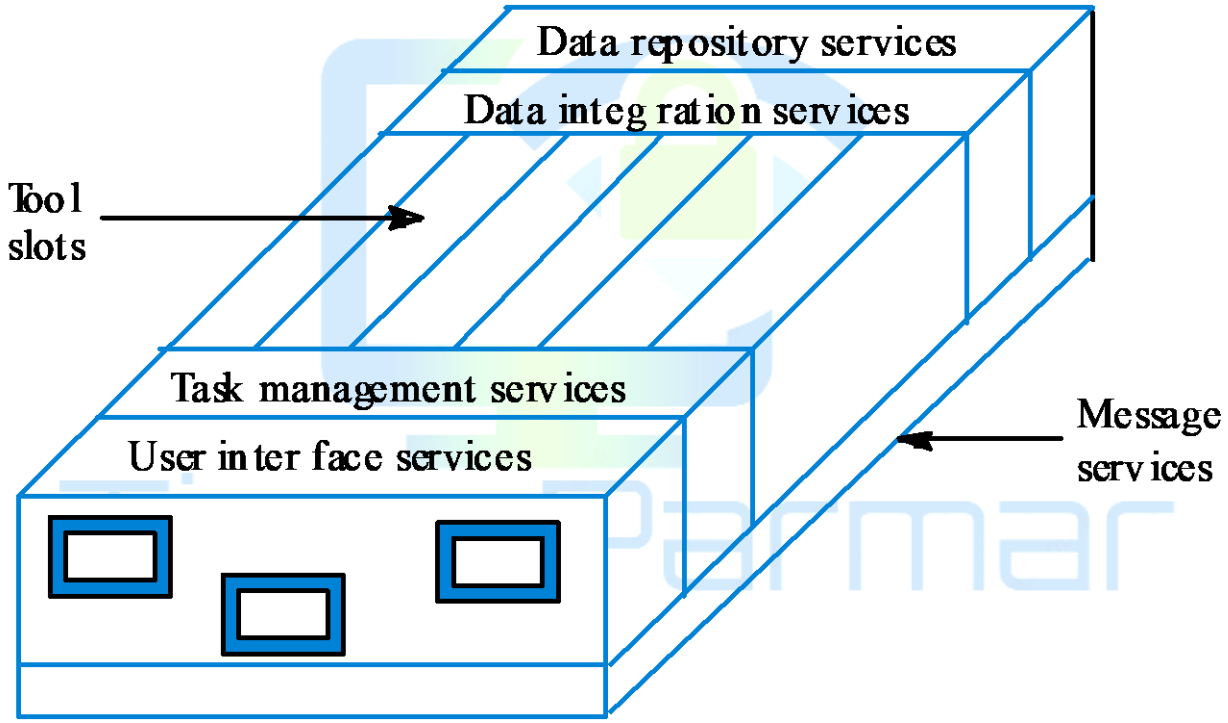
OSI reference model



Case reference model

- Data repository services
  - Storage and management of data items.
- Data integration services
  - Managing groups of entities.
- Task management services
  - Definition and enaction of process models.
- Messaging services
  - Tool-tool and tool-environment communication.
- User interface services
  - User interface development.

The ECMA reference model



Key points

- The software architecture is the fundamental framework for structuring the system.
- Architectural design decisions include decisions on the application architecture, the distribution and the architectural styles to be used.
- Different architectural models such as a structural model, a control model and a decomposition model may be developed.
- System organisational models include repository models, client-server models and abstract machine models.
- Modular decomposition models include object models and pipelining models.
- Control models include centralised control and event-driven models.

- Reference architectures may be used to communicate domain-specific architectures and to assess and compare architectural designs.

Architectural models

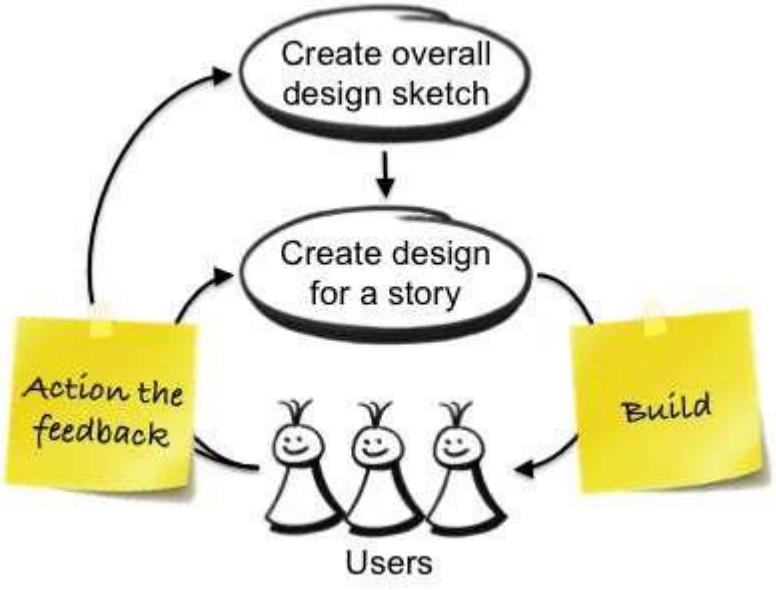
- Different architectural models may be produced during the design process
- Each model presents different perspectives on the architecture

Architecture attributes

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

User Interface Design

Introduction



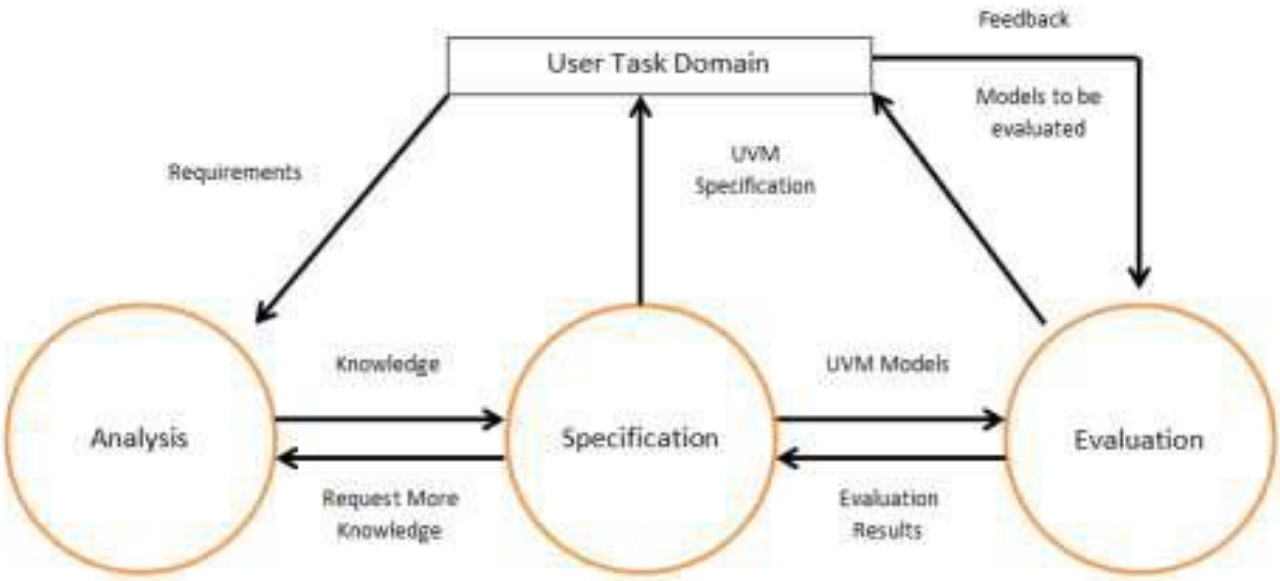
Why Should You Care about Interface Design Principles?

• Interface inconsistency can cost a big company millions of dollars in lost productivity and increased support costs.
• The software becomes more popular if its user interface is:
  – Attractive
  – Simple to use
  – Responsive in short time
  – Clear to understand
  – Consistent on all interfacing screens

Background

• Interface design focuses on the following
  – The design of interfaces between software components
  – The design of interfaces between the software and other nonhuman producers and consumers of information
  – The design of the interface between a human and the computer
• Graphical user interfaces (GUIs) have helped to eliminate many of the most horrific interface problems
• However, some are still difficult to learn, hard to use, confusing, counterintuitive, unforgiving, and frustrating
• User interface analysis and design has to do with the study of people and how they relate to technology

Interface Design Process



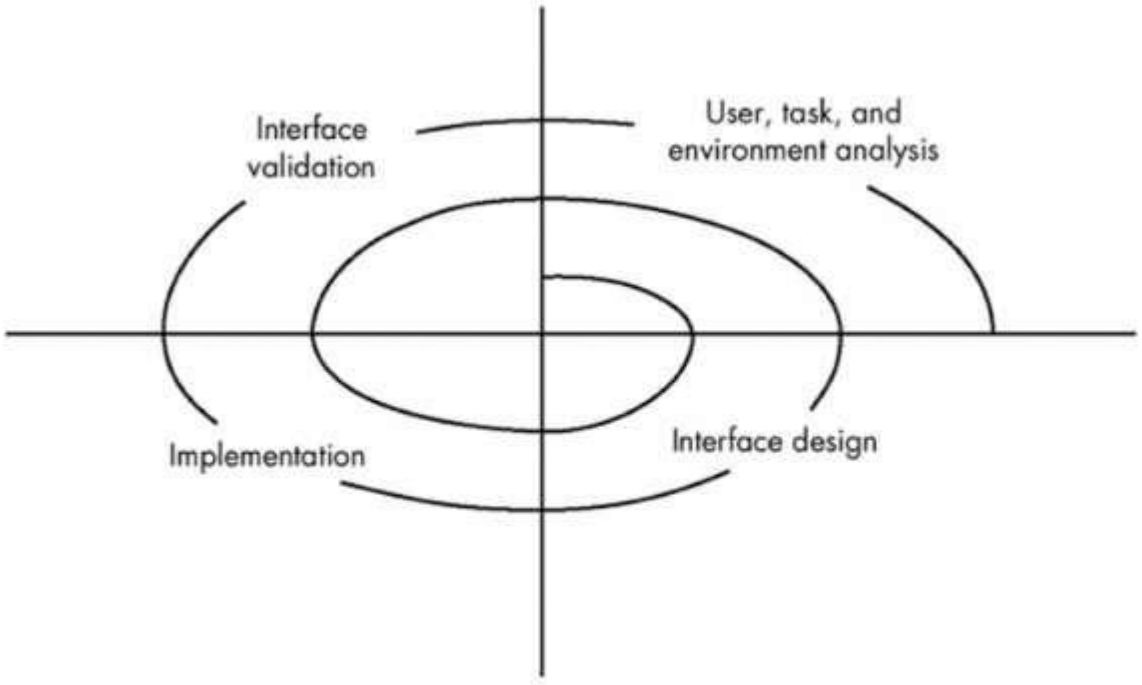
• User interface development follows a spiral process
  – Interface analysis (user, task, and environment analysis)
    • Focuses on the profile of the users who will interact with the system
    • Concentrates on users, tasks, content and work environment
    • Studies different models of system function (as perceived from the outside)

• Delineates the human- and computer-oriented tasks that are required to achieve system function

– Interface design
• Defines a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system

– Interface construction
• Begins with a prototype that enables usage scenarios to be evaluated
• Continues with development tools to complete the construction

– Interface validation, focuses on
• The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements
• The degree to which the interface is easy to use and easy to learn
• The users' acceptance of the interface as a useful tool in their work



The Golden Rules of User Interface Design

The three areas of user interface design principles are:
– **1.** Place users in control of the interface
– **2.** Reduce users' memory load
– **3.** Make the user interface consistent.

1. Place the User in Control

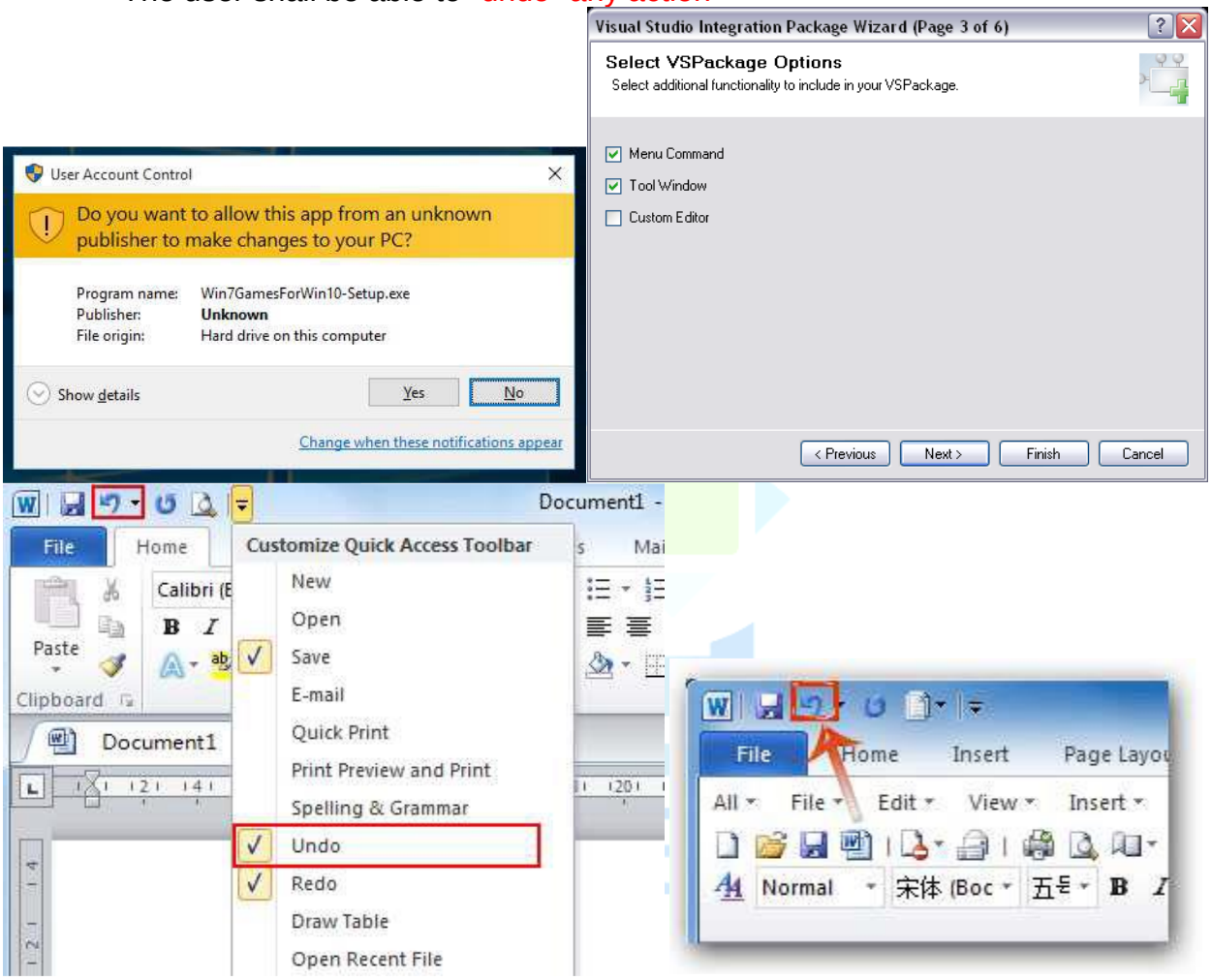
• Define interaction modes in a way that does not force a user into unnecessary or undesired actions
– The user shall be able to enter and exit a mode with little or no effort (e.g., spell check → edit text → spell check)
• Provide for flexible interaction

– The user shall be able to perform the same action via keyboard commands, mouse movement, or voice recognition

• Allow user interaction to be interruptible and "undo"able

– The user shall be able to easily interrupt a sequence of actions to do something else (without losing the work that has been done so far)

– The user shall be able to "undo" any action



• Streamline interaction as skill levels advance and allow the interaction to be customized

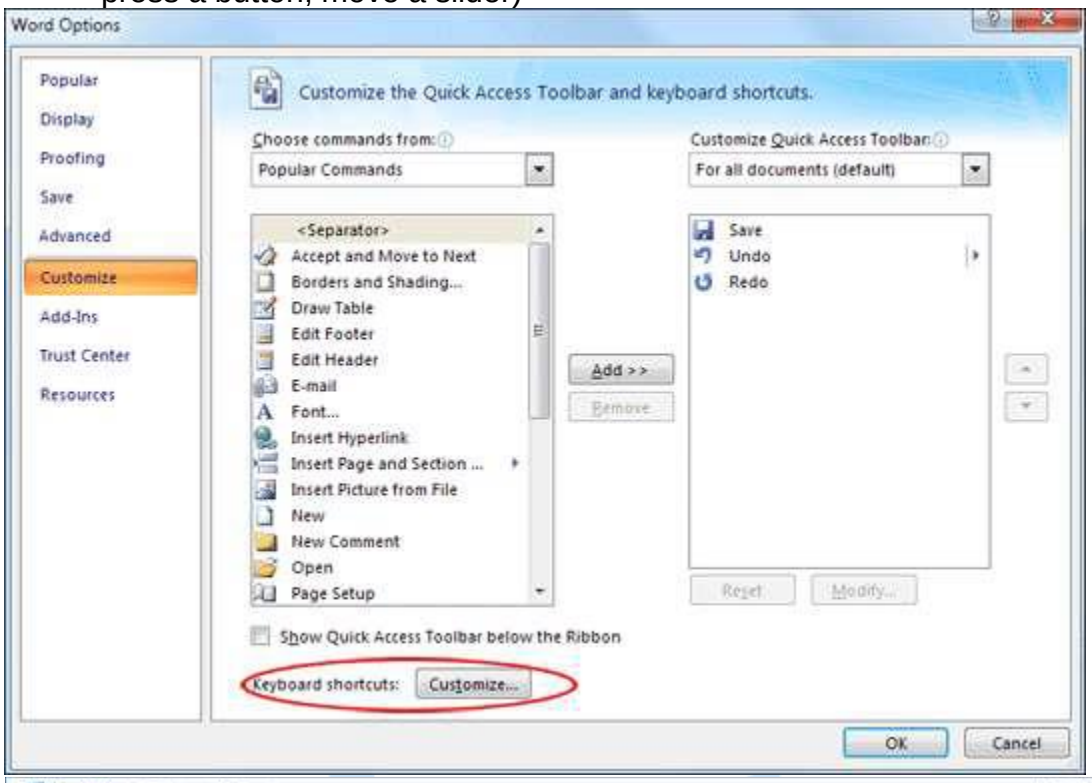
– The user shall be able to use a macro mechanism to perform a sequence of repeated interactions and to customize the interface

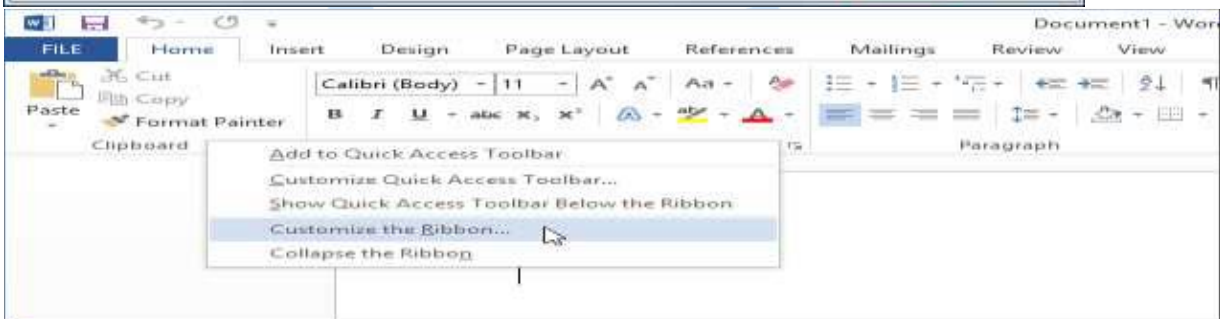
• Hide technical internals from the casual user

– The user shall not be required to directly use operating system, file management, networking. etc., commands to perform any actions. Instead, these operations shall be hidden from the user and performed "behind the scenes" in the form of a real-world abstraction

• Design for direct interaction with objects that appear on the screen

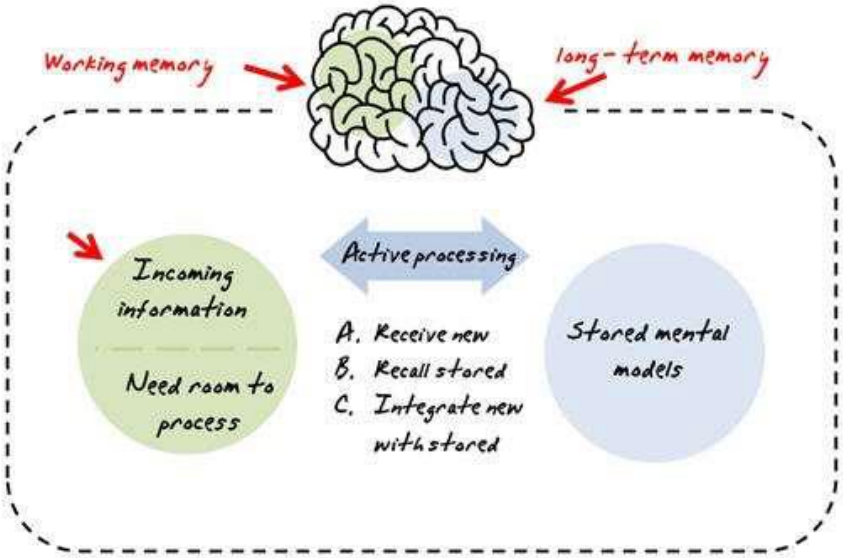
– The user shall be able to manipulate objects on the screen in a manner similar to what would occur if the object were a physical thing (e.g., stretch a rectangle, press a button, move a slider)





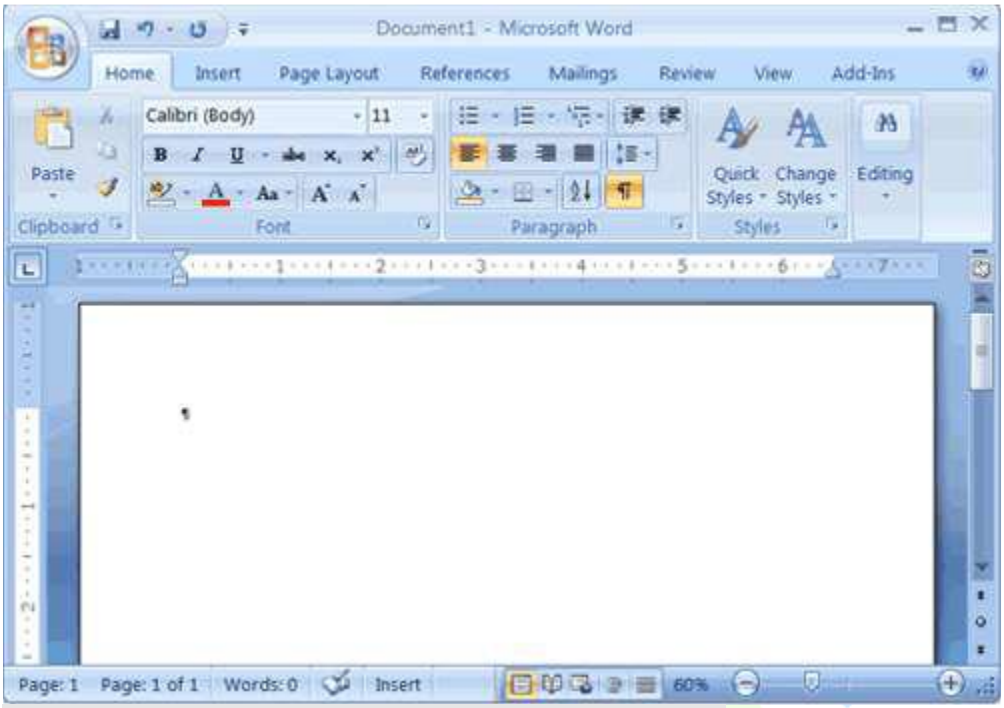
2. Reduce the User's Memory Load

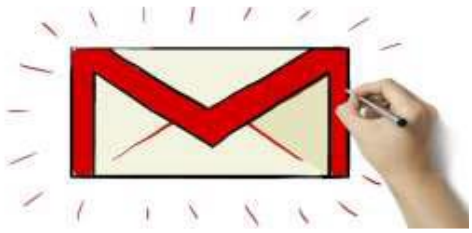
- Reduce demand on short-term memory
  - – The interface shall reduce the user's requirement to remember past actions and results by providing visual cues of such actions
- Establish meaningful defaults
  - – The system shall provide the user with default values that make sense to the average user but allow the user to change these defaults
  - – The user shall be able to easily reset any value to its original default value
- Define shortcuts that are intuitive
  - – The user shall be provided mnemonics (i.e., control or alt combinations) that tie easily to the action in a way that is easy to remember such as the first letter



The more a user has to remember, the more error-prone interaction with the system will be

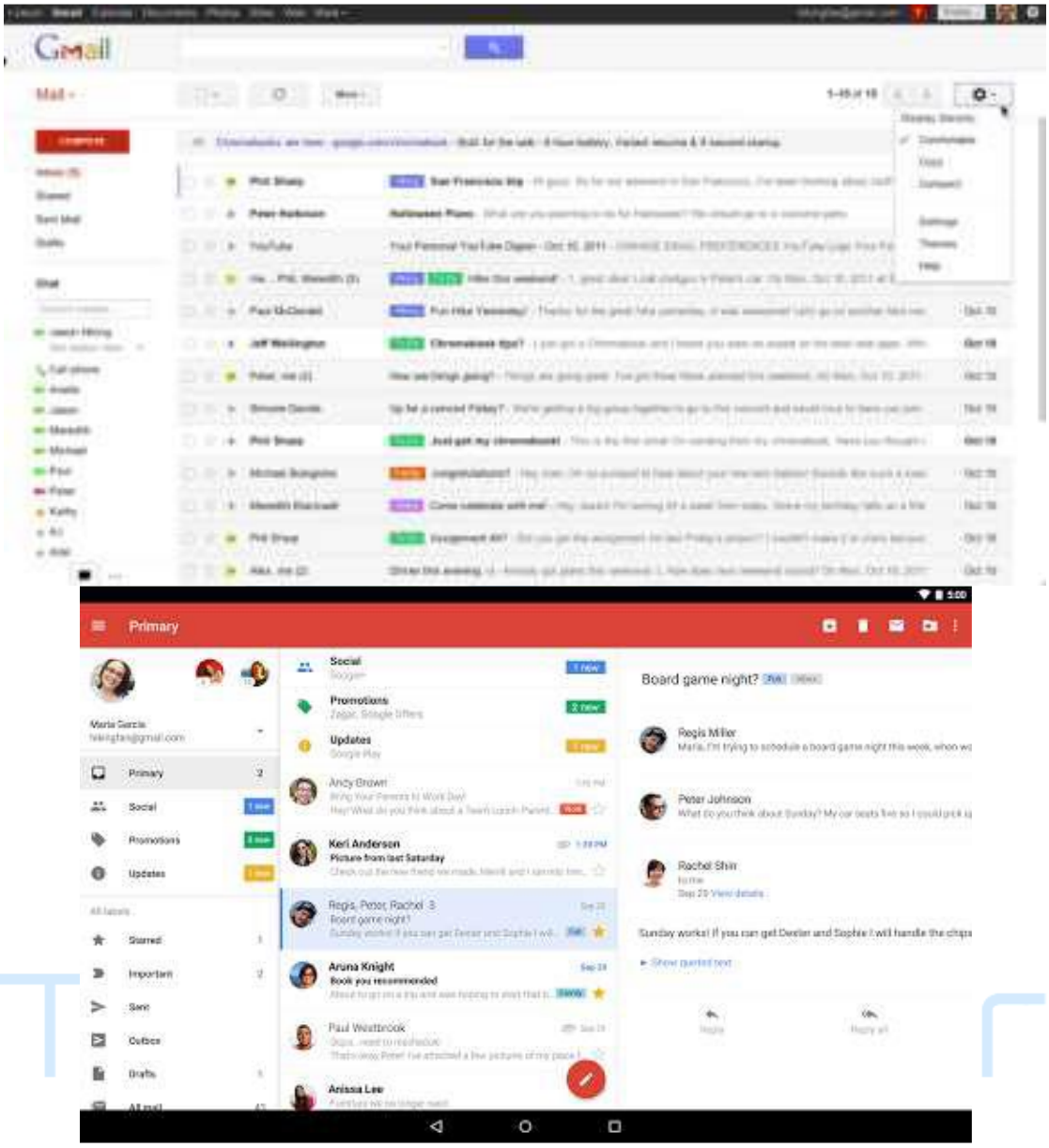
- The visual layout of the interface should be based on a real world metaphor
  - – The screen layout of the user interface shall contain well-understood visual cues that the user can relate to real-world actions
- Disclose information in a progressive fashion
  - – When interacting with a task, an object or some behavior, the interface shall be organized hierarchically by moving the user progressively in a step-wise fashion from an abstract concept to a concrete action (e.g., text format options
  - → format dialog box)



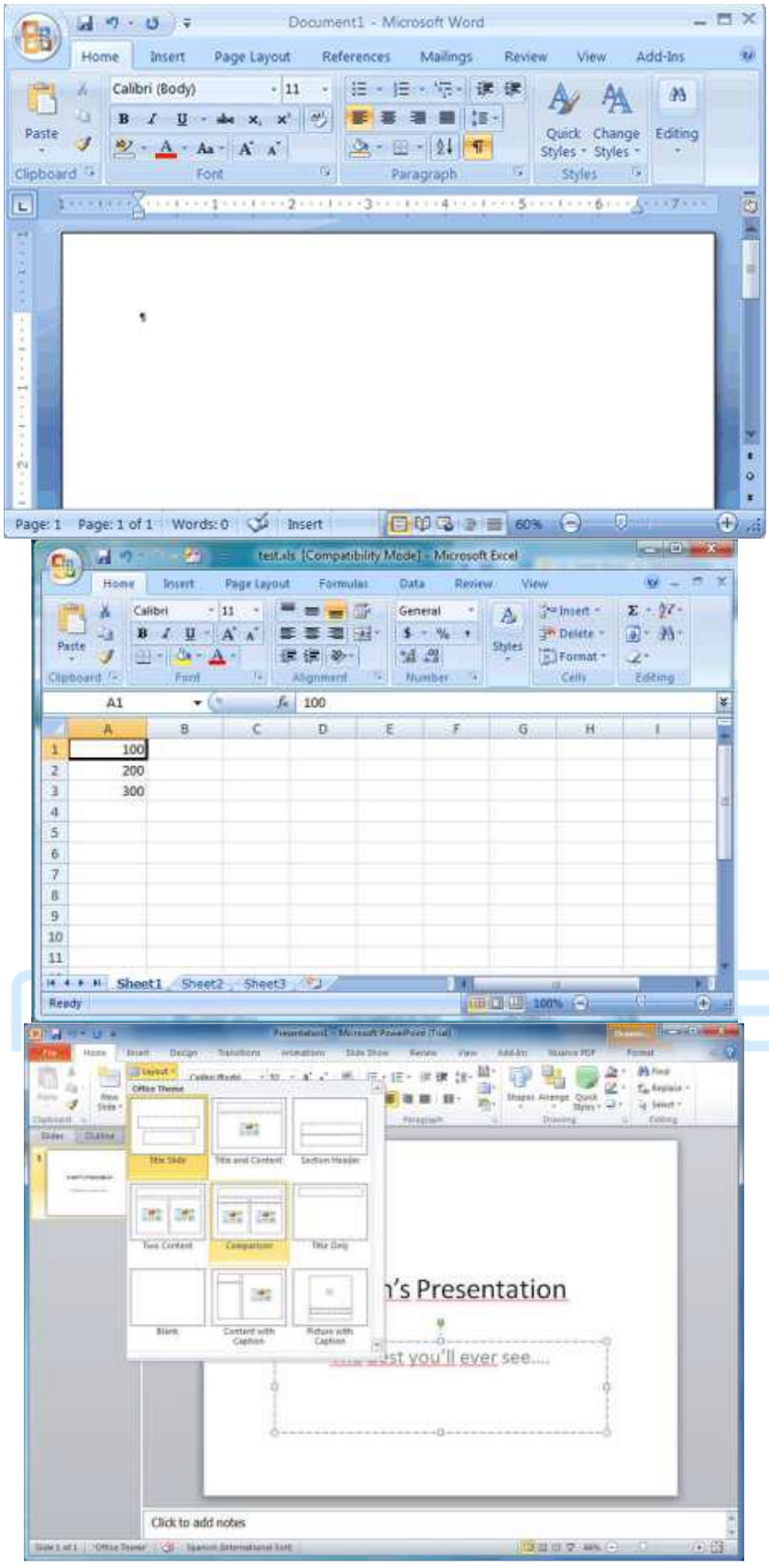


3.Make the Interface Consistent

- The interface should present and acquire information in a consistent fashion
  - – All visual information shall be organized according to a design standard that is maintained throughout all screen displays
  - – Input mechanisms shall be constrained to a limited set that is used consistently throughout the application
  - – Mechanisms for navigating from task to task shall be consistently defined and implemented
- Allow the user to put the current task into a meaningful context
  - – The interface shall provide indicators (e.g., window titles, consistent color coding) that enable the user to know the context of the work at hand
  - – The user shall be able to determine where he has come from and what alternatives exist for a transition to a new task



• Maintain consistency across a family of applications
  – A set of applications performing complimentary functionality shall all implement the same design rules so that consistency is maintained for all interaction
• If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so
  – Once a particular interactive sequence has become a de facto standard (e.g., alt-S to save a file), the application shall continue this expectation in every part of its functionality


Document1 - Microsoft Word
Home
Insert
Page Layout
References
Mailings
Review
View
Add-Ins
Paste
Calibri (Body)
Clipboard
Font
Paragraph
Styles
Editing
Page: 1
Page: 1 of 1
Words: 0
Insert
60%
test.xls [Compatibility Mode] - Microsoft Excel
Formulas
Data
Review
View
General
Insert
Delete
Format
Cells
A1
100
200
300
Sheet1
Sheet2
Sheet3
Ready
's Presentation
Click to add notes

Summary: Golden Rules

•*Place User in Control*
- •Define interaction in such a way that the user is not forced into performing unnecessary or undesired actions
- •Provide for flexible interaction (users have varying preferences)
- •Allow user interaction to be interruptible and reversible
- •Streamline interaction as skill level increases and allow customization of interaction
- •Hide technical internals from the casual user
- •Design for direct interaction with objects that appear on the screen

•*Reduce User Cognitive (Memory) Load*
- •Reduce demands on user's short-term memory
- •Establish meaningful defaults
- •Define intuitive short-cuts
- •Visual layout of user interface should be based on a familiar real world metaphor
- •Disclose information in a progressive fashion

•*Make Interface Consistent*
- •Allow user to put the current task into a meaningful context
- •Maintain consistency across a family of applications
- •If past interaction models have created user expectations, do not make changes unless there is a good reason to do so

User Interface Analysis and Design Models

Concepts of Good/Bad Design

•**Affordances**
Perceived properties of an artifact that determines how it can be used (e.g knobs/buttons/slots)

•**Constraints**
Physical, semantic, cultural, and logical factors that encourage proper actions

•**Conceptual Models**
Mental model of system which allows users to:
- understand the system
- predict the effects of actions
- interpret results

•**Mappings**
Describe relationship between controls and their effects on system

•**Visibility**
The system shows you the conceptual model by showing its state and actions that can be taken

•**Feedback**
Information about effects of user's actions

Introduction

• Four different models come into play when a user interface is analyzed and designed
  – User profile model – Established by a human engineer or software engineer
  – Design model – Created by a software engineer
  – Implementation model – Created by the software implementers
  – User's mental model – Developed by the user when interacting with the application
• The role of the interface designer is to reconcile these differences and derive a consistent representation of the interface

1. User Profile Model

• Establishes the profile of the end-users of the system
  – Based on age, gender, physical abilities, education, cultural or ethnic background, motivation, goals, and personality
• Considers syntactic knowledge of the user
  – The mechanics of interaction that are required to use the interface effectively
• Considers semantic knowledge of the user
  – The underlying sense of the application; an understanding of the functions that are performed, the meaning of input and output, and the objectives of the system
• Categorizes users as
  – Novices
    • No syntactic knowledge of the system, little semantic knowledge of the application, only general computer usage
  – Knowledgeable, intermittent users
    • Reasonable semantic knowledge of the system, low recall of syntactic information to use the interface
  – Knowledgeable, frequent users
    • Good semantic and syntactic knowledge (i.e., power user), look for shortcuts and abbreviated modes of operation

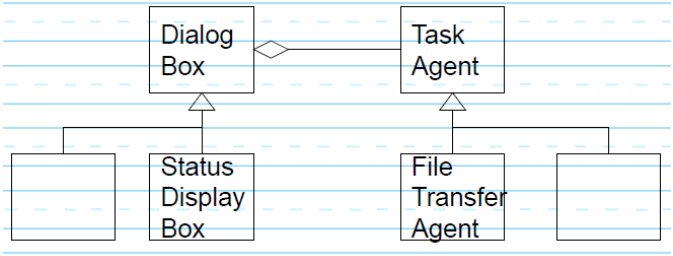
Norman's *Seven Stages of Action* that explain how people do things:

1. Form a goal
2. Form the intention
3. Specify an action
4. Execute the action
5. Perceive the state of the world
6. Interpret the state of the world
7. Evaluate the outcome

2. Design Model

• Derived from the analysis model of the requirements
• Incorporates data, architectural, interface, and procedural representations of the software
• Constrained by information in the requirements specification that helps define the user of the system

• Normally is incidental to other parts of the design model
  – But in many cases it is as important as the other parts

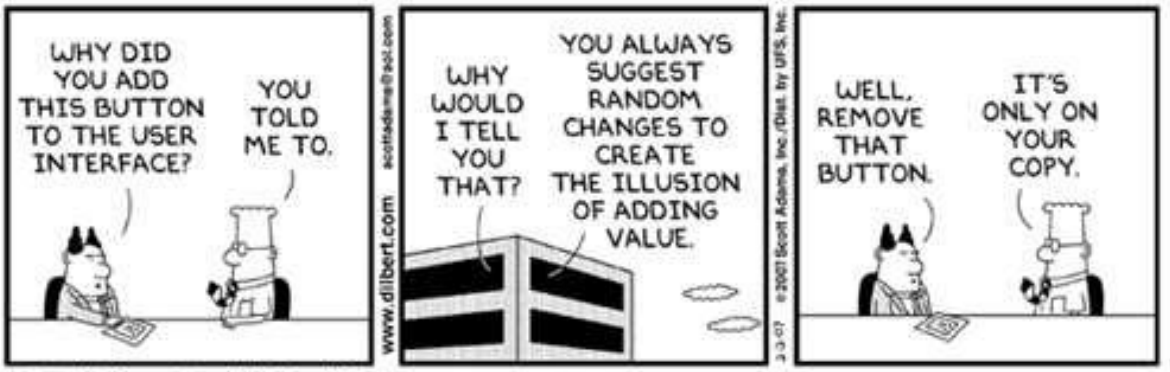


3.Implementation Model

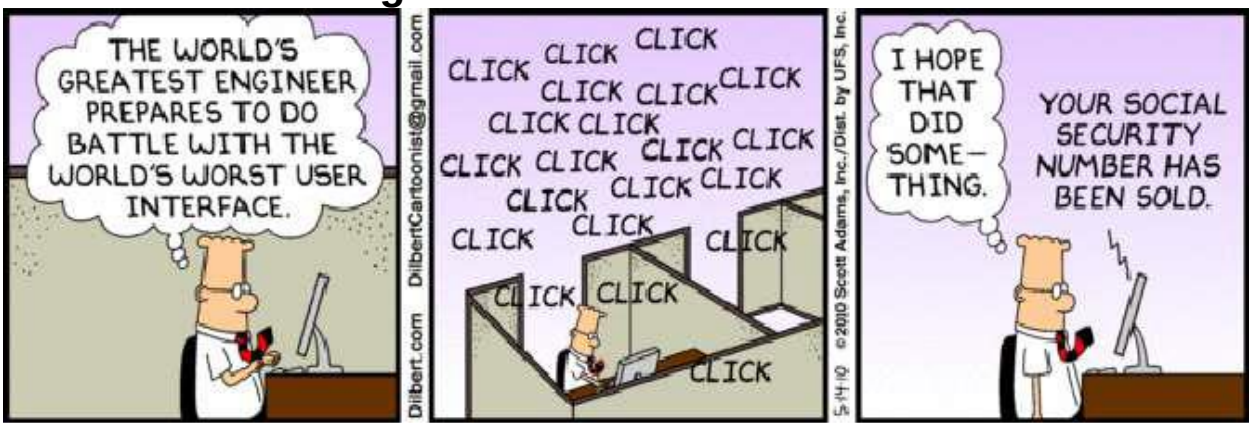
• Consists of the look and feel of the interface combined with all supporting information (books, videos, help files) that describe system syntax and semantics
• Strives to agree with the user's mental model; users then feel comfortable with the software and use it effectively
• Serves as a translation of the design model by providing a realization of the information contained in the user profile model and the user's mental model

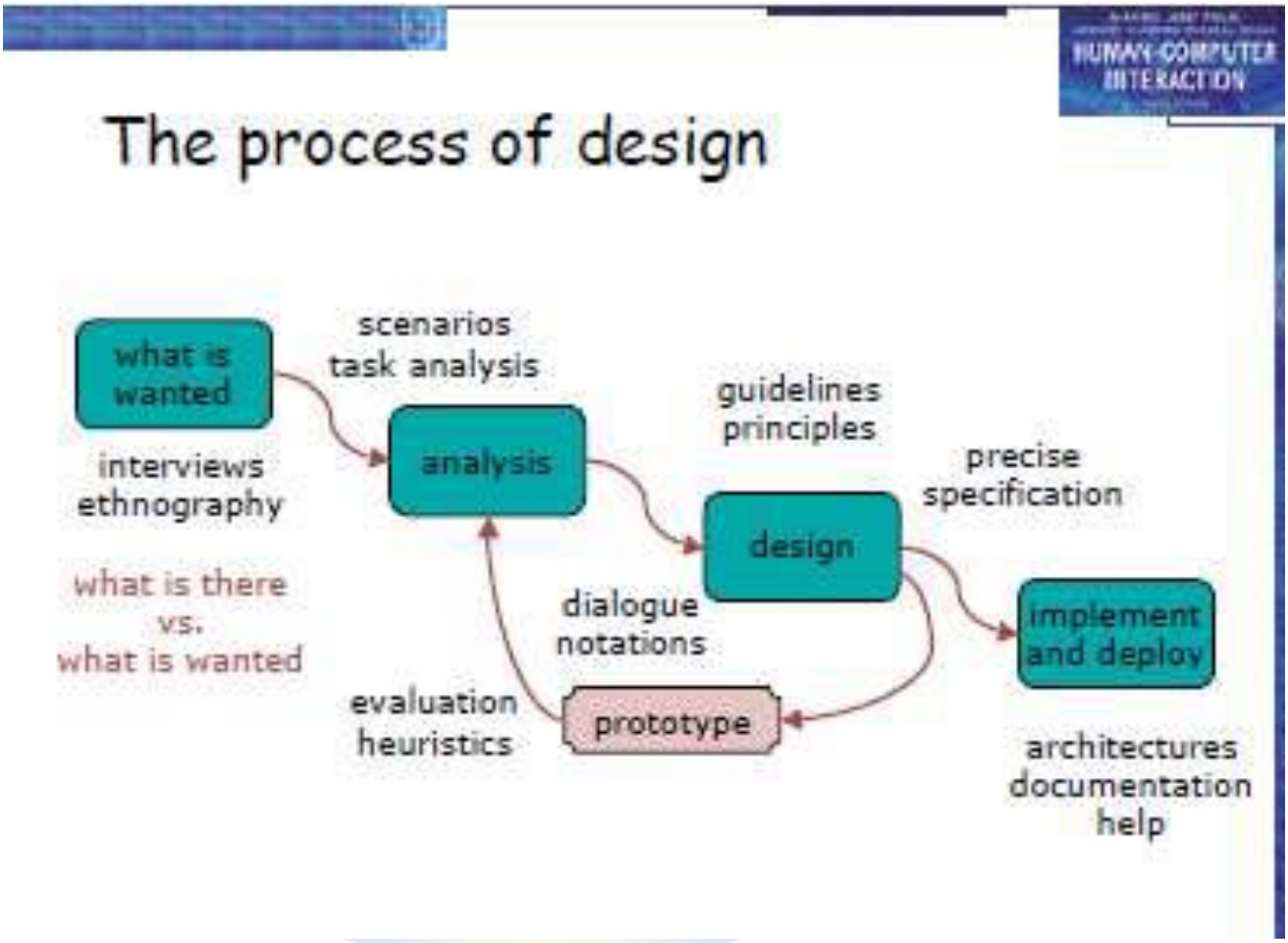
4.User's Mental Model

• Often called the user's system perception
• Consists of the image of the system that users carry in their heads
• Accuracy of the description depends upon the user's profile and overall familiarity with the software in the application domain



Process of Creating User Interfaces





Concepts of Good/Bad Design

•Affordances

Perceived properties of an artifact that determines how it can be used (e.g knobs/buttons/slots)

•Constraints

Physical, semantic, cultural, and logical factors that encourage proper actions

•Conceptual Models

Mental model of system which allows users to:

- understand the system
- predict the effects of actions
- interpret results

•Mappings

Describe relationship between controls and their effects on system

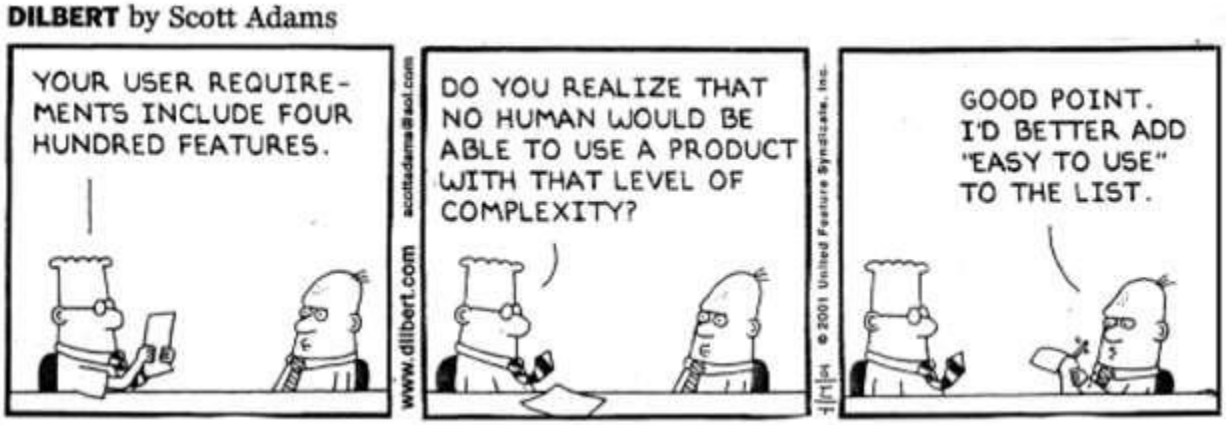
•Visibility

The system shows you the conceptual model by showing its state and actions that can be taken

•Feedback

Information about effects of user's actions

A.User Interface Analysis



Elements of the User Interface

• To perform user interface analysis, the practitioner needs to study and understand four elements

– The users who will interact with the system through the interface
– The tasks that end users must perform to do their work
– The content that is presented as part of the interface
– The work environment in which these tasks will be conducted

1.User Analysis

• The analyst strives to get the end user's mental model and the design model to converge by understanding

– The users themselves
– How these people use the system

• Information can be obtained from

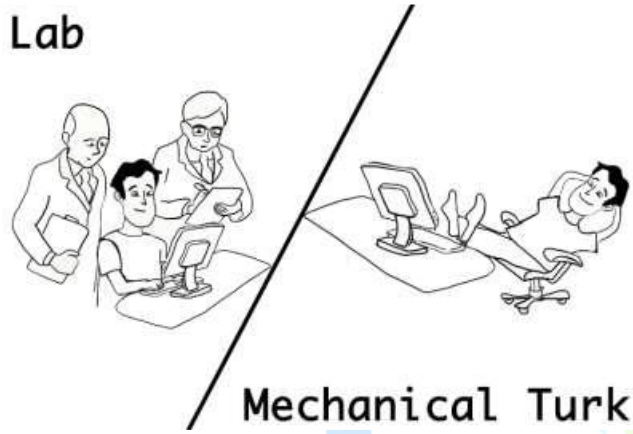
– User interviews with the end users
– Sales input from the sales people who interact with customers and users on a regular basis
– Marketing input based on a market analysis to understand how different population segments
might use the software
– Support input from the support staff who are aware of what works and what doesn't, what users like and dislike, what features generate questions, and what features are easy to use

• A set of questions should be answered during user analysis

Idea Generation
•observation
•listening
•brainstorming
•metaphor
•sketching

- scenario creations
- free association
- mediation
- juxtaposition
- searching for patterns
- "lateral thinking"

User Analysis Questions



1) Are the users trained professionals, technicians, clerical or manufacturing workers?
2) What level of formal education does the average user have?
3) Are the users capable of learning on their own from written materials or have they expressed a desire for classroom training?
4) Are the users expert typists or are they keyboard phobic?
5) What is the age range of the user community?
6) Will the users be represented predominately by one gender?
7) How are users compensated for the work they perform or are they volunteers?
8) Do users work normal office hours, or do they work whenever the job is required?
9) Is the software to be an integral part of the work users do, or will it be used only occasionally?
10) What is the primary spoken language among users?
11) What are the consequences if a user makes a mistake using the system?
12) Are users experts in the subject matter that is addressed by the system?
13) Do users want to know about the technology that sits behind the interface?

2. Task Analysis and Modeling

- Task analysis strives to know and understand
  - The work the user performs in specific circumstances
  - The tasks and subtasks that will be performed as the user does the work
  - The specific problem domain objects that the user manipulates as work is performed
  - The sequence of work tasks (i.e., the workflow)
  - The hierarchy of tasks

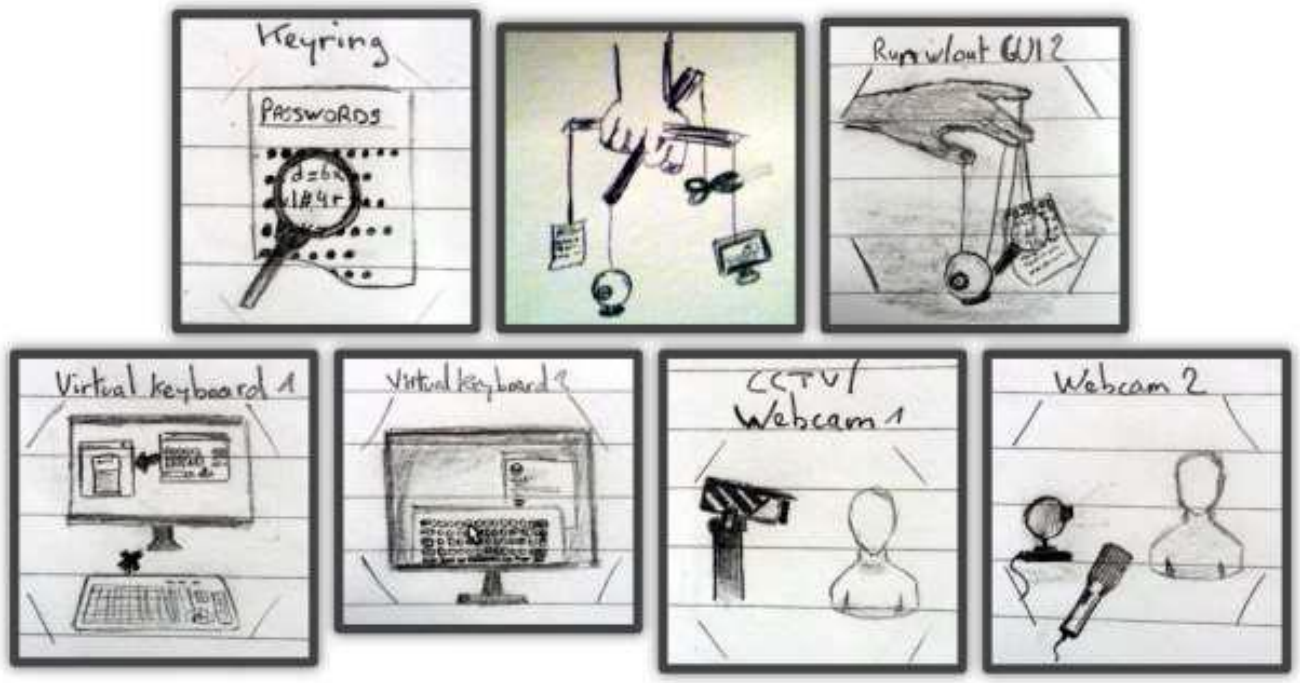
• Use cases
  – Show how an end user performs some specific work-related task
  – Enable the software engineer to extract tasks, objects, and overall workflow of the interaction
  – Helps the software engineer to identify additional helpful features

3.Content Analysis

• The display content may range from character-based reports, to graphical displays, to multimedia information
• Display content may be
  – Generated by components in other parts of the application
  – Acquired from data stored in a database that is accessible from the application
  – Transmitted from systems external to the application in question
• The format and aesthetics of the content (as it is displayed by the interface) needs to be considered
• A set of questions should be answered during content analysis
  1) Are various types of data assigned to consistent locations on the screen (e.g., photos always in upper right corner)?
  2) Are users able to customize the screen location for content?
  3) Is proper on-screen identification assigned to all content?
  4) Can large reports be partitioned for ease of understanding?
  5) Are mechanisms available for moving directly to summary information for large collections of data?
  6) Is graphical output scaled to fit within the bounds of the display device that is used?
  7) How is color used to enhance understanding?
  8) How are error messages and warnings presented in order to make them quick and easy to see and understand?

4.Work Environment Analysis

• Software products need to be designed to fit into the work environment, otherwise they may be difficult or frustrating to use
• Factors to consider include
  – Type of lighting
  – Display size and height
  – Keyboard size, height and ease of use
  – Mouse type and ease of use
  – Surrounding noise
  – Space limitations for computer and/or user
  – Weather or other atmospheric conditions
  – Temperature or pressure restrictions
  – Time restrictions (when, how fast, and for how long)



B.User Interface Designs

Introduction

• User interface design is an iterative process, where each iteration elaborate and refines the information developed in the preceding step

• General steps for user interface design

1) Using information developed during user interface analysis, define user interface objects and actions (operations)
2) Define events (user actions) that will cause the state of the user interface to change; model this behavior
3) Depict each interface state as it will actually look to the end user
4) Indicate how the user interprets the state of the system from information provided through the interface

• During all of these steps, the designer must
  – Always follow the three golden rules of user interfaces
  – Model how the interface will be implemented
  – Consider the computing environment (e.g., display technology, operating system, development tools) that will be used

1.Interface Objects and Actions

• Interface objects and actions are obtained from a grammatical parse of the use cases and the software problem statement
• Interface objects are categorized into types: source, target, and application
  – A source object is dragged and dropped into a target object such as to create a hardcopy of a report
  – An application object represents application-specific data that are not directly manipulated as part of screen interaction such as a list
• After identifying objects and their actions, an interface designer performs screen layout which involves
  – Graphical design and placement of icons
  – Definition of descriptive screen text
  – Specification and titling for windows
  – Definition of major and minor menu items
  – Specification of a real-world metaphor to follow

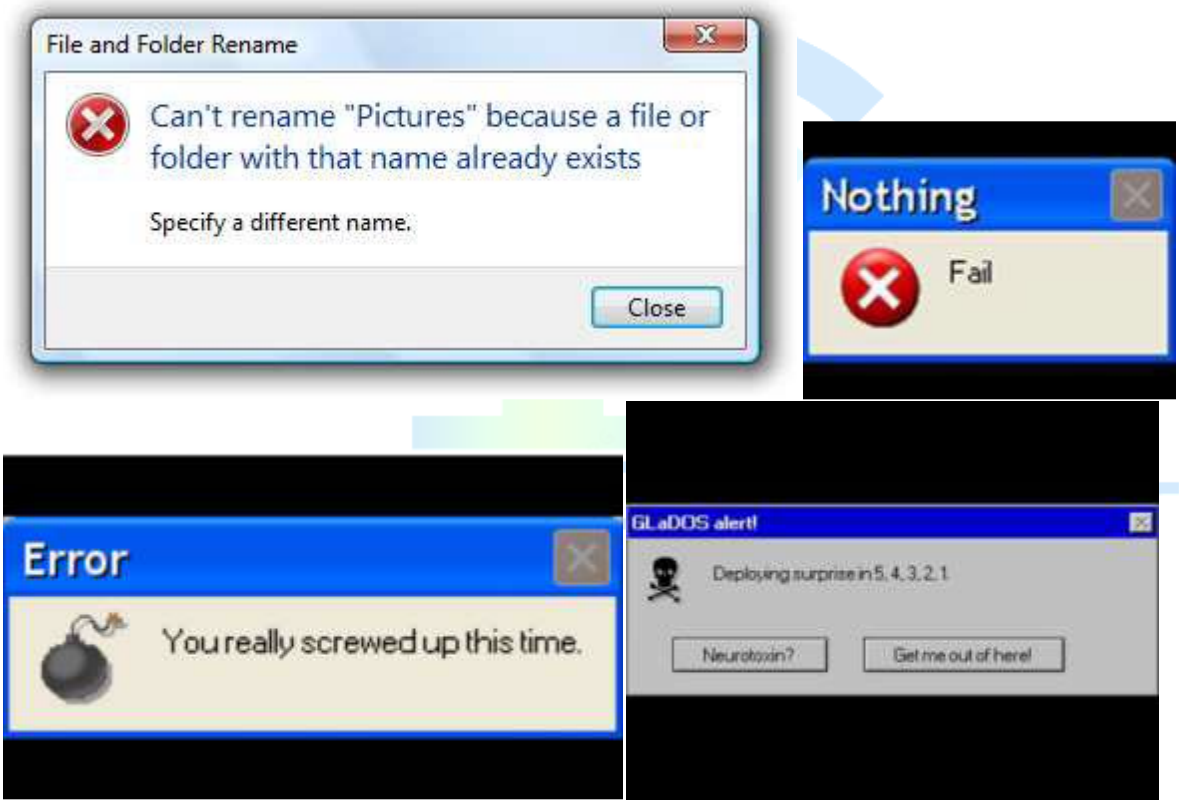
Design Issues to Consider

• Four common design issues usually surface in any user interface
  – System response time (both length and variability)
  – User help facilities
• When is it available, how is it accessed, how is it represented to the user, how is it structured, what happens when help is exited
  – Error information handling
  – How meaningful to the user, how descriptive of the problem
  – Menu and command labeling (more on upcoming slide)
• Consistent, easy to learn, accessibility, internationalization
• Many software engineers do not address these issues until late in the design or construction process
  – This results in unnecessary iteration, project delays, and customer frustration

Guidelines for Error Messages

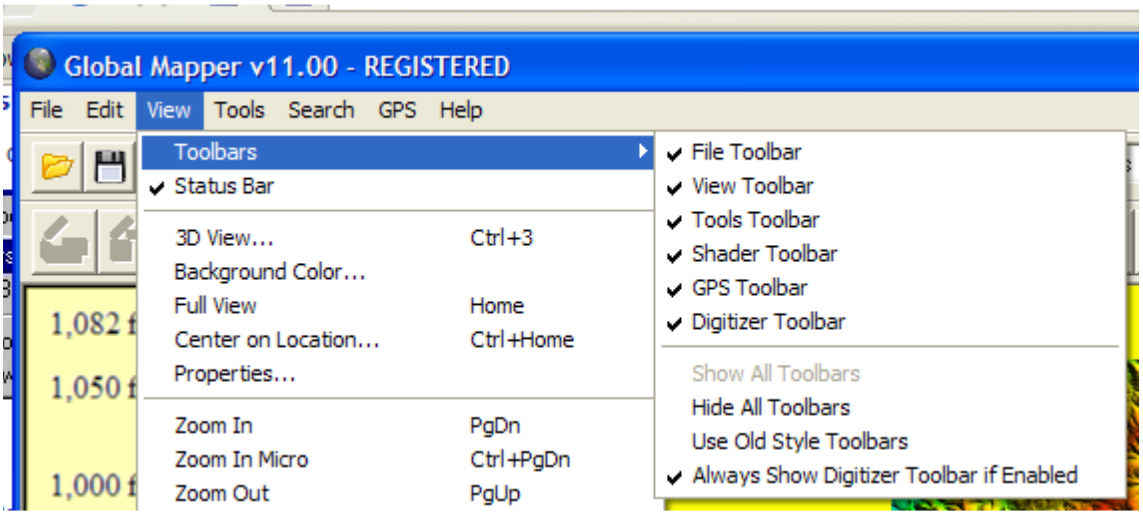
An effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur

• The message should describe the problem in plain language that a typical user can understand
• The message should provide constructive advice for recovering from the error
• The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have)
• The message should be accompanied by an audible or visual cue such as a beep, momentary flashing, or a special error color
• The message should be non-judgmental
  – The message should never place blame on the user



Questions for Menu Labeling and Typed Commands

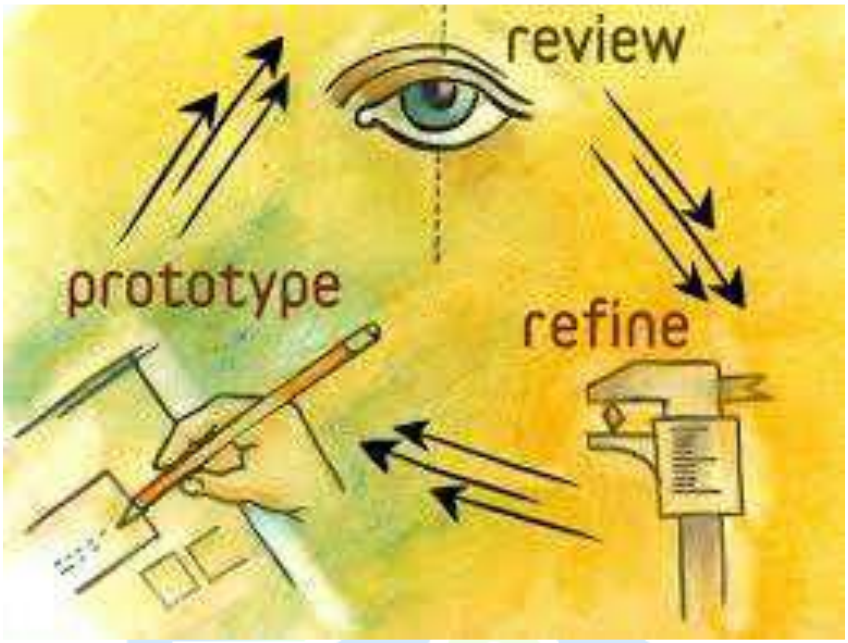
• Will every menu option have a corresponding command?
• What form will a command take? A control sequence? A function key? A typed word?
• How difficult will it be to learn and remember the commands?
• What can be done if a command is forgotten?
• Can commands be customized or abbreviated by the user?
• Are menu labels self-explanatory within the context of the interface?
• Are submenus consistent with the function implied by a master menu item?



C. User Interface Evaluation

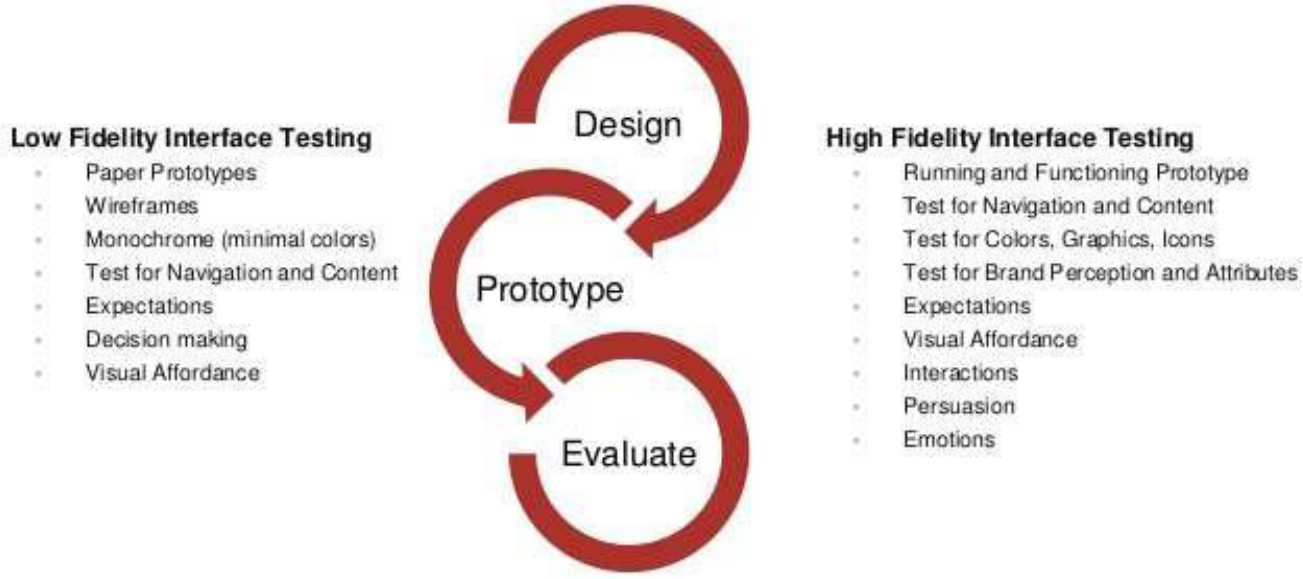
Design and Prototype Evaluation

- Before prototyping occurs, a number of evaluation criteria can be applied during design reviews to the design model itself
  - The amount of learning required by the users
- Derived from the length and complexity of the written specification and its interfaces
  - The interaction time and overall efficiency
- Derived from the number of user tasks specified and the average number of actions per task
  - The memory load on users
- Derived from the number of actions, tasks, and system states
  - The complexity of the interface and the degree to which it will be accepted by the user
- Derived from the interface style, help facilities, and error handling procedures



• Prototype evaluation can range from an informal test drive to a formally designed study using statistical methods and questionnaires
• The prototype evaluation cycle consists of prototype creation followed by user evaluation and back to prototype modification until all user issues are resolved
• The prototype is evaluated for
  – Satisfaction of user requirements
  – Conformance to the three golden rules of user interface design
  – Reconciliation of the four models of a user interface

UI EVALUATION



Summary: Interface Creation

1. User, task, and environment analysis and modeling
    - Where will the interface be located physically?
    - Will the user be sitting, standing, or performing other tasks unrelated to the interface?
    - Does the interface hardware accommodate space, light, or noise constraints?
    - Are there special human factors considerations driven by environmental factors?
2. Interface design
    - define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system
3. Interface construction
4. Interface Evaluation
    - the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements
    - the degree to which the interface is easy to use and easy to learn
    - the users' acceptance of the interface as a useful tool in their work

FAQs on Interface Design

- Define (in context of Interface design): User interface, models, design, evaluation, prototype
- Describe process of UI Design
- Golden Rules of Design
- Process of : analysis, design, evaluation
- Ways to know users
- Need of effective UI design
- Justification and examples of golden rules
- And similar kinds…

Project Management

Software project management

✧ Concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organisations developing and procuring the software.

✧ Project management is needed because software development is always subject to budget and schedule constraints that are set by the organisation developing the software.

Success criteria

✧ Deliver the software to the customer at the agreed time.
✧ Keep overall costs within budget.
✧ Deliver software that meets the customer's expectations.
✧ Maintain a happy and well-functioning development team.

Software management distinctions

✧ The product is intangible.
  - Software cannot be seen or touched. Software project managers cannot see progress by simply looking at the artefact that is being constructed.

✧ Many software projects are 'one-off' projects.
  - Large software projects are usually different in some ways from previous projects. Even managers who have lots of previous experience may find it difficult to anticipate problems.

✧ Software processes are variable and organization specific.
  - We still cannot reliably predict when a particular software process is likely to lead to development problems.

Management activities

✧ *Project planning*
  - Project managers are responsible for planning. estimating and scheduling project development and assigning people to tasks.

✧ *Reporting*
  - Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software.

✧ *Risk management*
  - Project managers assess the risks that may affect a project, monitor these risks and take action when problems arise.

✧ *People management*
  - Project managers have to choose people for their team and establish ways of working that leads to effective team performance

- *Proposal writing*
  - The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out.

Risk management

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

Examples of common project, product, and business risks

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

The risk management process

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

Risk identification

- ✧ May be a team activities or based on the individual project manager's experience.
- ✧ A checklist of common risks may be used to identify risks in a project
  - Technology risks.
  - People risks.
  - Organisational risks.
  - Requirements risks.
  - Estimation risks.

Examples of different risk types

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. (1) Reusable software components contain defects that mean they cannot be reused as planned. (2)
People	It is impossible to recruit staff with the skills required. (3) Key staff are ill and unavailable at critical times. (4) Required training for staff is not available. (5)
Organizational	The organization is restructured so that different management are responsible for the project. (6) Organizational financial problems force reductions in the project budget. (7)
Tools	The code generated by software code generation tools is inefficient. (8) Software tools cannot work together in an integrated way. (9)
Requirements	Changes to requirements that require major design rework are proposed. (10) Customers fail to understand the impact of requirements changes. (11)
Estimation	The time required to develop the software is underestimated. (12) The rate of defect repair is underestimated. (13) The size of the software is underestimated. (14)

Risk analysis

- ✧ Assess probability and seriousness of each risk.
- ✧ Probability may be very low, low, moderate, high or very high.

- Risk consequences might be catastrophic, serious, tolerable or insignificant.

Risk types and examples

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget (7).	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project (3).	High	Catastrophic
Key staff are ill at critical times in the project (4).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused. (2).	Moderate	Serious
Changes to requirements that require major design rework are proposed (10).	Moderate	Serious
The organization is restructured so that different management are responsible for the project (6).	High	Serious
The database used in the system cannot process as many transactions per second as expected (1).	Moderate	Serious
The time required to develop the software is underestimated (12).	High	Serious
Software tools cannot be integrated (9).	High	Tolerable
Customers fail to understand the impact of requirements changes (11).	Moderate	Tolerable
Required training for staff is not available (5).	Moderate	Tolerable
The rate of defect repair is underestimated (13).	Moderate	Tolerable
The size of the software is underestimated (14).	High	Tolerable
Code generated by code generation tools is inefficient (8).	Moderate	Insignificant

Risk planning

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

Strategies to help manage risk

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of a program generator.

Risk monitoring

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

Risk indicators

Risk type	Potential indicators
Technology	Late delivery of hardware or support software; many reported technology problems.
People	Poor staff morale; poor relationships amongst team members; high staff turnover.
Organizational	Organizational gossip; lack of action by senior management.
Tools	Reluctance by team members to use tools; complaints about CASE tools; demands for higher-powered workstations.
Requirements	Many requirements change requests; customer complaints.
Estimation	Failure to meet agreed schedule; failure to clear reported defects.

Key points

✧ Good project management is essential if software engineering projects are to be developed on schedule and within budget.
✧ Software management is distinct from other engineering management. Software is intangible. Projects may be novel or innovative with no body of experience to guide their management. Software processes are not as mature as traditional engineering processes.
✧ Risk management is now recognized as one of the most important project management tasks.
✧ Risk management involves identifying and assessing project risks to establish the probability that they will occur and the consequences for the project if that risk does arise. You should make plans to avoid, manage or deal with likely risks if or when they arise.

Managing people

✧ People are an organisation's most important assets.
✧ The tasks of a manager are essentially people-oriented. Unless there is some understanding of people, management will be unsuccessful.
✧ Poor people management is an important contributor to project failure.

People management factors

✧ Consistency
  - Team members should all be treated in a comparable way without favourites or discrimination.
✧ Respect
  - Different team members have different skills and these differences should be respected.
✧ Inclusion
  - Involve all team members and make sure that people's views are considered.
✧ Honesty
  - You should always be honest about what is going well and what is going badly in a project.

Motivating people

✧ An important role of a manager is to motivate the people working on a project.
✧ Motivation means organizing the work and the working environment to encourage people to work effectively.
  - If people are not motivated, they will not be interested in the work they are doing. They will work slowly, be more likely to make mistakes and will not contribute to the broader goals of the team or the organization.
✧ Motivation is a complex issue but it appears that their are different types of motivation based on:
  - Basic needs (e.g. food, sleep, etc.);
  - Personal needs (e.g. respect, self-esteem);
  - Social needs (e.g. to be accepted as part of a group).

Human needs hierarchy

Need satisfaction

✧ In software development groups, basic physiological and safety needs are not an issue.

✧ Social
- Provide communal facilities;
- Allow informal communications e.g. via social networking

✧ Esteem
- Recognition of achievements;
- Appropriate rewards.

✧ Self-realization
- Training - people want to learn more;
- Responsibility.

Individual motivation

Alice is a software project manager working in a company that develops alarm systems. This company wishes to enter the growing market of assistive technology to help elderly and disabled people live independently. Alice has been asked to lead a team of 6 developers than can develop new products based around the company's alarm technology.

Alice's assistive technology project starts well. Good working relationships develop within the team and creative new ideas are developed. The team decides to develop a peer-to-peer messaging system using digital televisions linked to the alarm network for communications. However, some months into the project, Alice notices that Dorothy, a hardware design expert, starts coming into work late, the quality of her work deteriorates and, increasingly, that she does not appear to be communicating with other members of the team.
Alice talks about the problem informally with other team members to try to find out if Dorothy's personal circumstances have changed, and if this might be affecting her work. They don't know of anything, so Alice decides to talk with Dorothy to try to understand the problem.

After some initial denials that there is a problem, Dorothy admits that she has lost interest in the job. She expected that she would be able to develop and use her hardware interfacing skills. However, because of the product direction that has been chosen, she has little opportunity for this. Basically, she is working as a C programmer with other team members.

Although she admits that the work is challenging, she is concerned that she is not developing her interfacing skills. She is worried that finding a job that involves hardware interfacing will be difficult after this project. Because she does not want to upset the team by revealing that she is thinking about the next project, she has decided that it is best to minimize conversation with them.

Personality types

- The needs hierarchy is almost certainly an oversimplification of motivation in practice.
- Motivation should also take into account different personality types:
  - Task-oriented;
  - Self-oriented;
  - Interaction-oriented.
- Task-oriented.
  - The motivation for doing the work is the work itself;
- Self-oriented.
  - The work is a means to an end which is the achievement of individual goals - e.g. to get rich, to play tennis, to travel etc.;
- Interaction-oriented
  - The principal motivation is the presence and actions of co-workers. People go to work because they like to go to work.

Motivation balance

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

Teamwork

- Most software engineering is a group activity
  - The development schedule for most non-trivial software projects is such that they cannot be completed by one person working alone.
- A good group is cohesive and has a team spirit. The people involved are motivated by the success of the group as well as by their own personal goals.
- Group interaction is a key determinant of group performance.
- Flexibility in group composition is limited
  - Managers must do the best they can with available people.

Group cohesiveness

✧ In a cohesive group, members consider the group to be more important than any individual in it.

✧ The advantages of a cohesive group are:

- Group quality standards can be developed by the group members.
- Team members learn from each other and get to know each other's work; Inhibitions caused by ignorance are reduced.
- Knowledge is shared. Continuity can be maintained if a group member leaves.
- Refactoring and continual improvement is encouraged. Group members work collectively to deliver high quality results and fix problems, irrespective of the individuals who originally created the design or program.

Team spirit

Alice, an experienced project manager, understands the importance of creating a cohesive group. As they are developing a new product, she takes the opportunity of involving all group members in the product specification and design by getting them to discuss possible technology with elderly members of their families. She also encourages them to bring these family members to meet other members of the development group.

Alice also arranges monthly lunches for everyone in the group. These lunches are an opportunity for all team members to meet informally, talk around issues of concern, and get to know each other. At the lunch, Alice tells the group what she knows about organizational news, policies, strategies, and so forth. Each team member then briefly summarizes what they have been doing and the group discusses a general topic, such as new product ideas from elderly relatives.

Every few months, Alice organizes an 'away day' for the group where the team spends two days on 'technology updating'. Each team member prepares an update on a relevant technology and presents it to the group. This is an off-site meeting in a good hotel and plenty of time is scheduled for discussion and social interaction.

The effectiveness of a team

✧ The people in the group

- You need a mix of people in a project group as software development involves diverse activities such as negotiating with clients, programming, testing and documentation.

✧ The group organization

- A group should be organized so that individuals can contribute to the best of their abilities and tasks can be completed as expected.

✧ Technical and managerial communications

- Good communications between group members, and between the software engineering team and other project stakeholders, is essential.

Selecting group members

✧ A manager or team leader's job is to create a cohesive group and organize their group so that they can work together effectively.
✧ This involves creating a group with the right balance of technical skills and personalities, and organizing that group so that the members work together effectively.

Assembling a team

✧ May not be possible to appoint the ideal people to work on a project
- Project budget may not allow for the use of highly-paid staff;
- Staff with the appropriate experience may not be available;
- An organisation may wish to develop employee skills on a software project.

✧ Managers have to work within these constraints especially when there are shortages of trained staff.

Group composition

✧ Group composed of members who share the same motivation can be problematic
- Task-oriented - everyone wants to do their own thing;
- Self-oriented - everyone wants to be the boss;
- Interaction-oriented - too much chatting, not enough work.

✧ An effective group has a balance of all types.
✧ This can be difficult to achieve software engineers are often task-oriented.
✧ Interaction-oriented people are very important as they can detect and defuse tensions that arise.

In creating a group for assistive technology development, Alice is aware of the importance of selecting members with complementary personalities. When interviewing potential group members, she tried to assess whether they were task-oriented, self-oriented, or interaction-oriented. She felt that she was primarily a self-oriented type because she considered the project to be a way of getting noticed by senior management and possibly promoted. She therefore looked for one or perhaps two interaction-oriented personalities, with task-oriented individuals to complete the team. The final assessment that she arrived at was:

Alice—self-oriented
Brian—task-oriented
Bob—task-oriented
Carol—interaction-oriented
Dorothy—self-oriented
Ed—interaction-oriented
Fred—task-oriented

Group organization

✧ The way that a group is organized affects the decisions that are made by that group, the ways that information is exchanged and the interactions between the development group and external project stakeholders.

- Key questions include:
  - Should the project manager be the technical leader of the group?
  - Who will be involved in making critical technical decisions, and how will these be made?
  - How will interactions with external stakeholders and senior company management be handled?
  - How can groups integrate people who are not co-located?
  - How can knowledge be shared across the group?

✧ Small software engineering groups are usually organized informally without a rigid structure.

✧ For large projects, there may be a hierarchical structure where different groups are responsible for different subprojects.

✧ Agile development is always based around an informal group on the principle that formal structure inhibits information exchange

Informal groups

✧ The group acts as a whole and comes to a consensus on decisions affecting the system.

✧ The group leader serves as the external interface of the group but does not allocate specific work items.

✧ Rather, work is discussed by the group as a whole and tasks are allocated according to ability and experience.

✧ This approach is successful for groups where all members are experienced and competent.

Group communications

✧ Good communications are essential for effective group working.

✧ Information must be exchanged on the status of work, design decisions and changes to previous decisions.

✧ Good communications also strengthens group cohesion as it promotes understanding.

✧ Group size
  - The larger the group, the harder it is for people to communicate with other group members.

✧ Group structure
  - Communication is better in informally structured groups than in hierarchically structured groups.

✧ Group composition
  - Communication is better when there are different personality types in a group and when groups are mixed rather than single sex.

✧ The physical work environment
  - Good workplace organisation can help encourage communications.

Key points

✧ People are motivated by interaction with other people, the recognition of management and their peers, and by being given opportunities for personal development.

✧ Software development groups should be fairly small and cohesive. The key factors that influence the effectiveness of a group are the people in that group, the way that it is organized and the communication between group members.

✧ Communications within a group are influenced by factors such as the status of group members, the size of the group, the gender composition of the group, personalities and available communication channels.

Quality Management & Process Improvement

Objectives

- To introduce the quality management process and key quality management activities
- To explain how software process factors influence software quality and productivity
- To explain the concept of a software metric, predictor metrics and control metrics
- To explain the principles of software process improvement and key process improvement activities
- To explain the notion of process capability and the CMMI process improvement model

Topics

- Process and product quality
- Quality management activities
  - Quality assurance and standards
  - Quality planning
  - Quality control
- Process improvement activities
  - Process measurement
  - Process analysis and modelling
  - Process change
- The CMMI process improvement framework

Software quality management

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

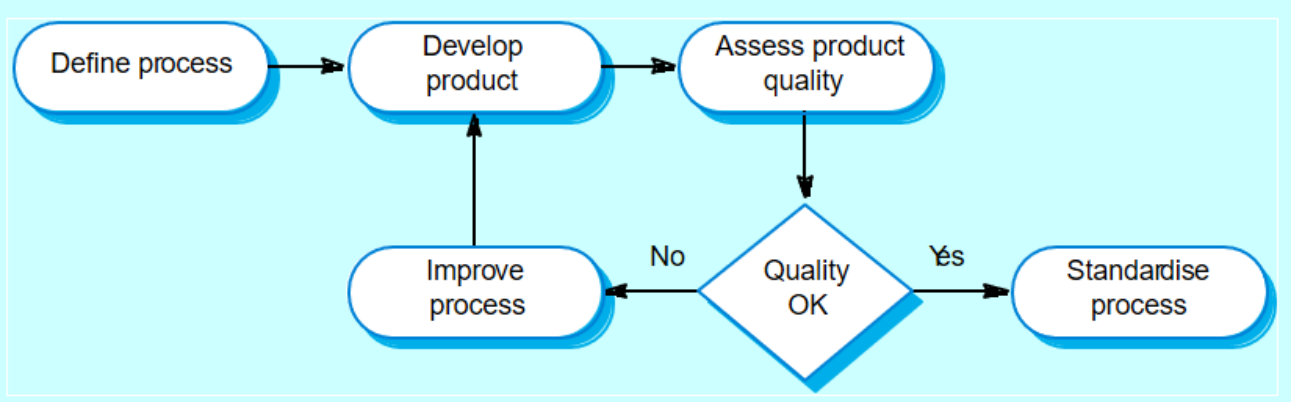
What is quality?

- Quality, simplistically, means that a product should meet its specification.
- This is problematical for software systems
  - There is a tension between customer quality requirements (efficiency, reliability, etc.) and developer quality requirements (maintainability, reusability, etc.);
  - Some quality requirements are difficult to specify in an unambiguous way;
  - Software specifications are usually incomplete and often inconsistent.
- Software quality management procedures cannot rely on having perfect specifications.

Process and product quality

- The quality of a developed product is influenced by the quality of the production process.
- A good process is usually required to produce a good product.
  - For manufactured goods, this link is straightforward.
  - But for software, this link is more complex and poorly understood.

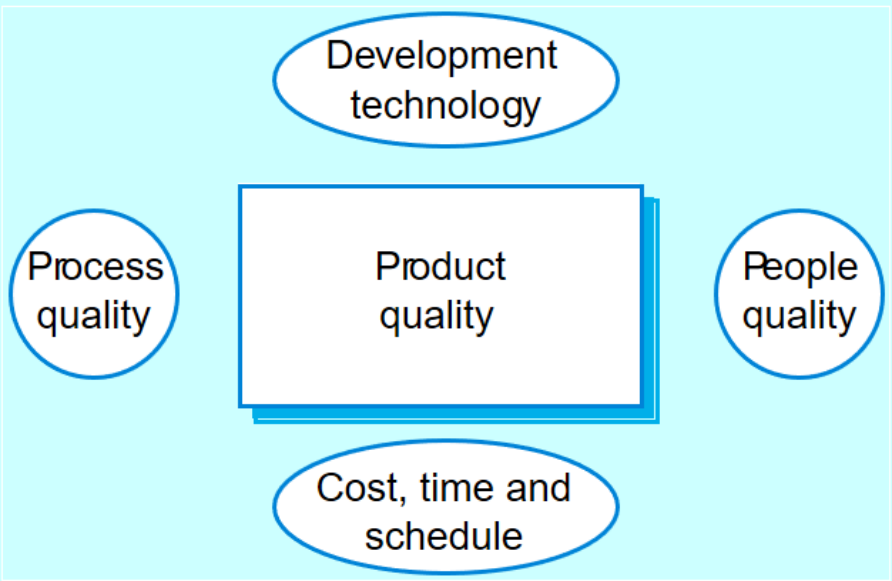
Process-based product quality



- More complex for software because:
  - The application of individual creative skills and experience is particularly important in software development;
  - Other factors also play a significant role in product quality;
  - Software product quality attributes are hard to assess.

- Care must be taken not to impose inappropriate process standards - these could reduce rather than improve the product quality.

Principal product quality factors



Quality factors

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

Practical process quality

- Define process standards such as how reviews should be conducted, configuration management, etc.
- Monitor the development process to ensure that standards are being followed.
- Report on the process to project management and software procurer.
- Don't use inappropriate practices simply because standards have been established.

- **Quality management activities**
  - Quality assurance and standards
  - Quality planning
  - Quality control

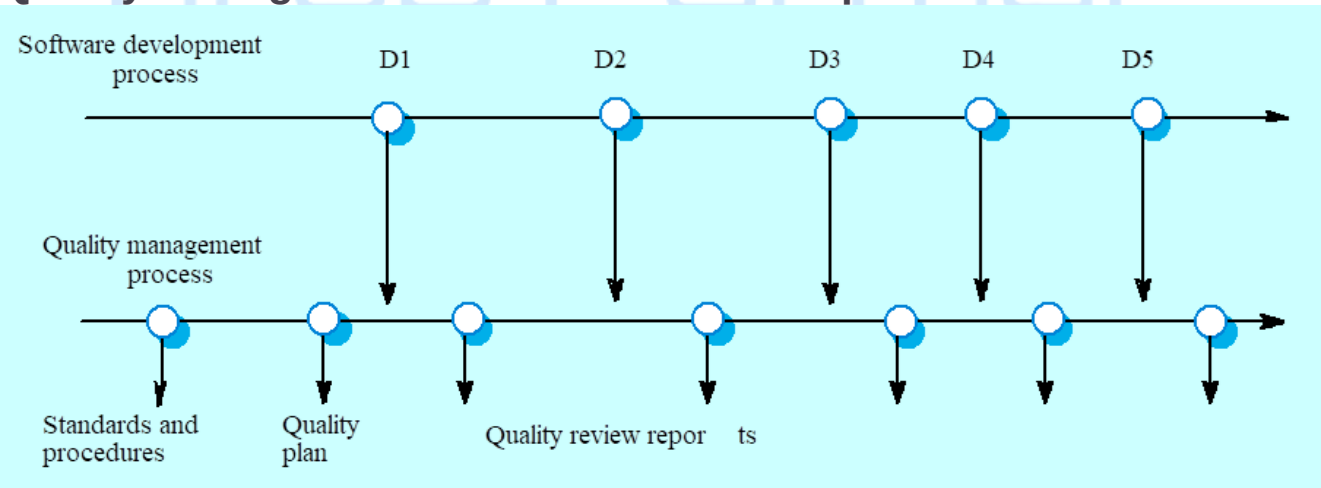
Scope of quality management

- Quality management is particularly important for large, complex systems. The quality documentation is a record of progress and supports continuity of development as the development team changes.
- For smaller systems, quality management needs less documentation and should focus on establishing a quality culture.

Quality management activities

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

Quality management and software development



Quality assurance and standards

- Standards are important for effective quality management.
  - Encapsulation of best practice – avoids repetition of past mistakes.
  - They are a framework for quality assurance processes – they involve checking compliance to standards.
  - They provide continuity – new staff can understand the organisation by understanding the standards that are used.

Quality standards

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

Product and process standards

Product standards	Process standards
Design review form	Design review conduct
Requirements document structure	Submission of documents to CM
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

Problems with standards

- They may not be seen as relevant and up-to-date by software engineers.
- They often involve too much bureaucratic form filling.
- If they are unsupported by software tools, tedious manual work is often involved to maintain the documentation associated with the standards.

Standards development

- Involve practitioners in development. Engineers should understand the rationale underlying a standard.

- Review standards and their usage regularly. Standards can quickly become outdated and this reduces their credibility amongst practitioners.
- Detailed standards should have associated tool support. Excessive clerical work is the most significant complaint against standards.

ISO 9000

- An international set of standards for quality management.
- Applicable to a range of organisations from manufacturing to service industries.
- ISO 9001 applicable to organisations which design, develop and maintain products.
- ISO 9001 is a generic model of the quality process that must be instantiated for each organisation using the standard.

ISO 9001

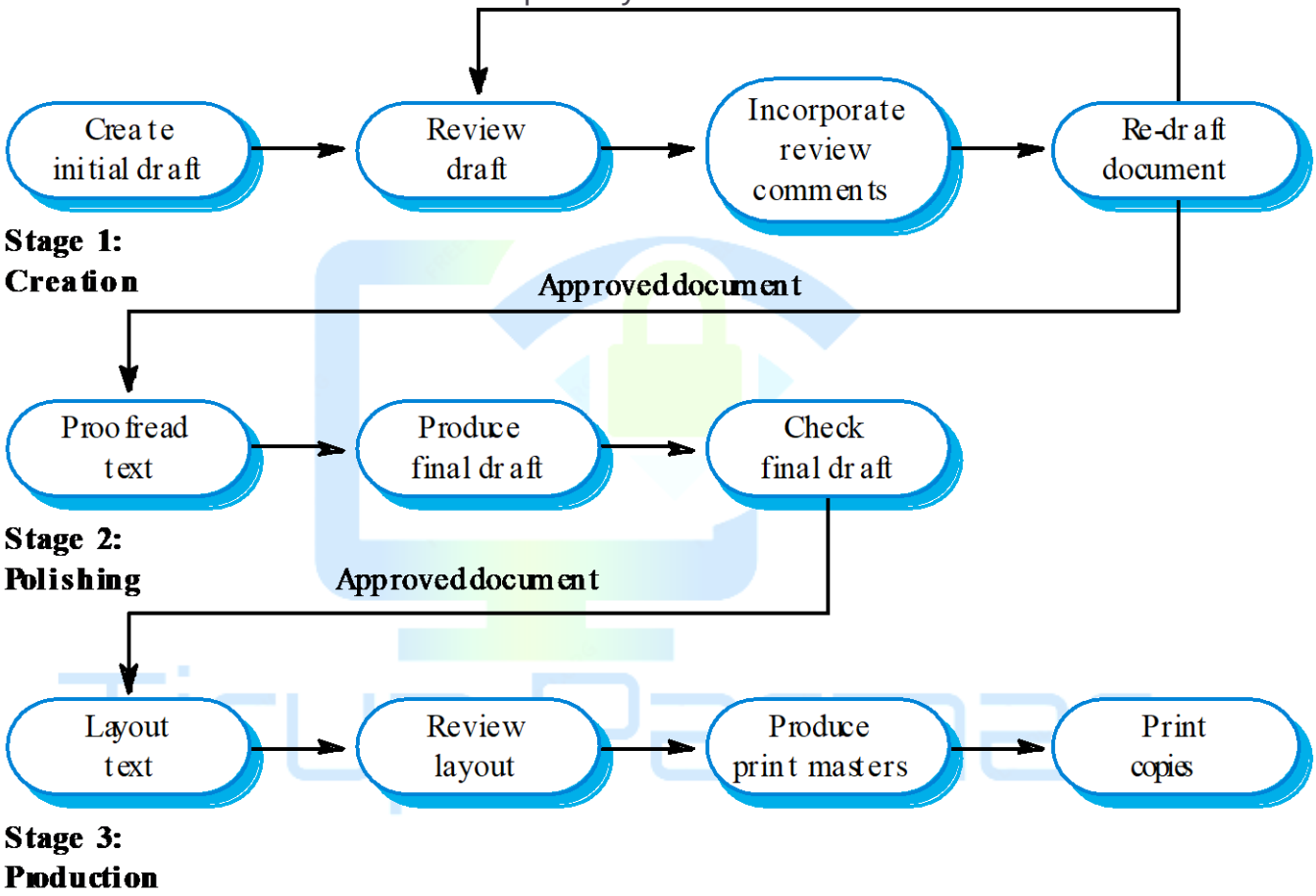
Management responsibility	Quality system
Control of non-conforming products	Design control
Handling, storage, packaging and delivery	Purchasing
Purchaser-supplied products	Product identification and traceability
Process control	Inspection and testing
Inspection and test equipment	Inspection and test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

ISO 9000 certification

- Quality standards and procedures should be documented in an organisational quality manual.
- An external body may certify that an organisation's quality manual conforms to ISO 9000 standards.
- Some customers require suppliers to be ISO 9000 certified although the need for flexibility here is increasingly recognised.

Documentation standards

- Particularly important – documents are the tangible manifestation of the software.
- Documentation process standards
  - Concerned with how documents should be developed, validated and maintained.
- Document standards
  - Concerned with document contents, structure, and appearance.
- Document interchange standards
  - Concerned with the compatibility of electronic documents.



Document standards

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

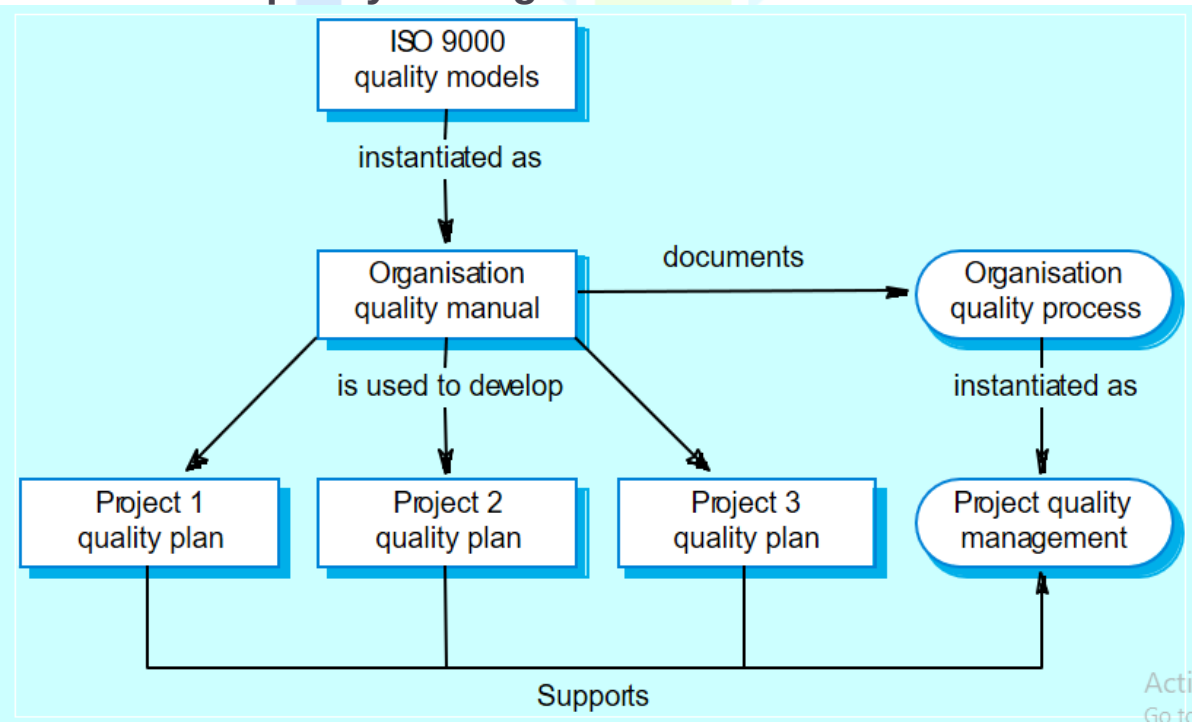
Document interchange standards

- Interchange standards allow electronic documents to be exchanged, mailed, etc.
- Documents are produced using different systems and on different computers. Even when standard tools are used, standards are needed to define conventions for their use e.g. use of style sheets and macros.
- Need for archiving. The lifetime of word processing systems may be much less than the lifetime of the software being documented. An archiving standard may be defined to ensure that the document can be accessed in future.

Quality planning

- The process of developing a quality plan for a project.
- A quality plan sets out the desired product qualities and how these are assessed and defines the most significant quality attributes.
- The quality plan should define the quality assessment process.
- It should set out which organisational standards should be applied and, where necessary, define new standards to be used.

ISO 9000 and quality management



Quality plans

- Quality plan structure
  - Product introduction;
  - Product plans;
  - Process descriptions;
  - Quality goals;

  - Risks and risk management.
- Quality plans should be short, succinct documents
  - If they are too long, no-one will read them.

Software quality attributes

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

Quality control

- This involves checking the software development process to ensure that procedures and standards are being followed.
- There are two approaches to quality control
  - Quality reviews;
  - Objective software assessment and software measurement.

Quality reviews

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

Types of review

Review type	Principal purpose
Design or program inspections	To detect detailed errors in the requirements, design or code. A checklist of possible errors should drive the review.
Progress reviews	To provide information for management about the overall progress of the project. This is both a process and a product review and is concerned with costs, plans and schedules.
Quality reviews	To carry out a technical analysis of product components or documentation to find mismatches between the specification and the component design, code or documentation and to ensure that defined quality standards have been followed.

Quality reviews

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

Review functions

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

Quality reviews

- The objective is the discovery of system defects and inconsistencies.
- Any documents produced in the process may be reviewed.
- Review teams should be relatively small and reviews should be fairly short.
- Records should always be maintained of quality reviews.

Review results

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

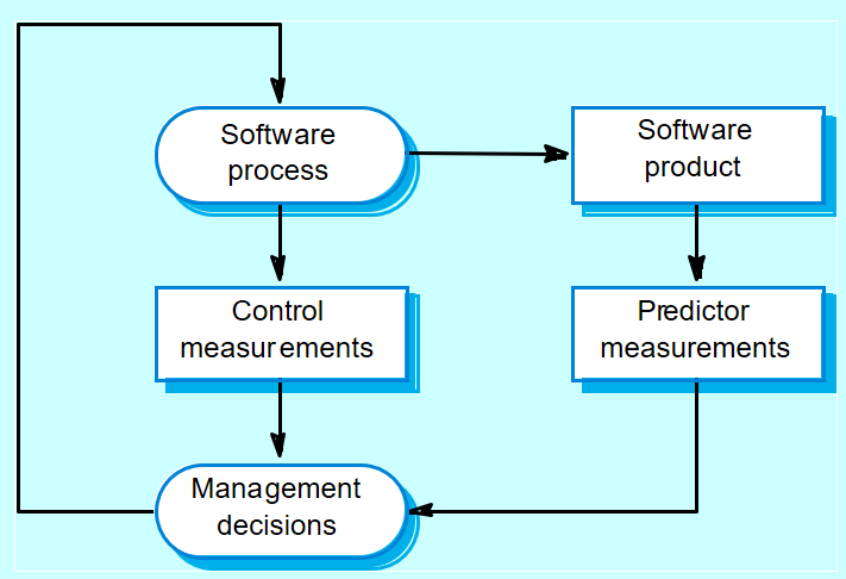
Software measurement and metrics

- Software measurement is concerned with deriving a numeric value for an attribute of a software product or process.
- This allows for objective comparisons between techniques and processes.
- Although some companies have introduced measurement programmes, most organisations still don't make systematic use of software measurement.
- There are few established standards in this area.

Software metric

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

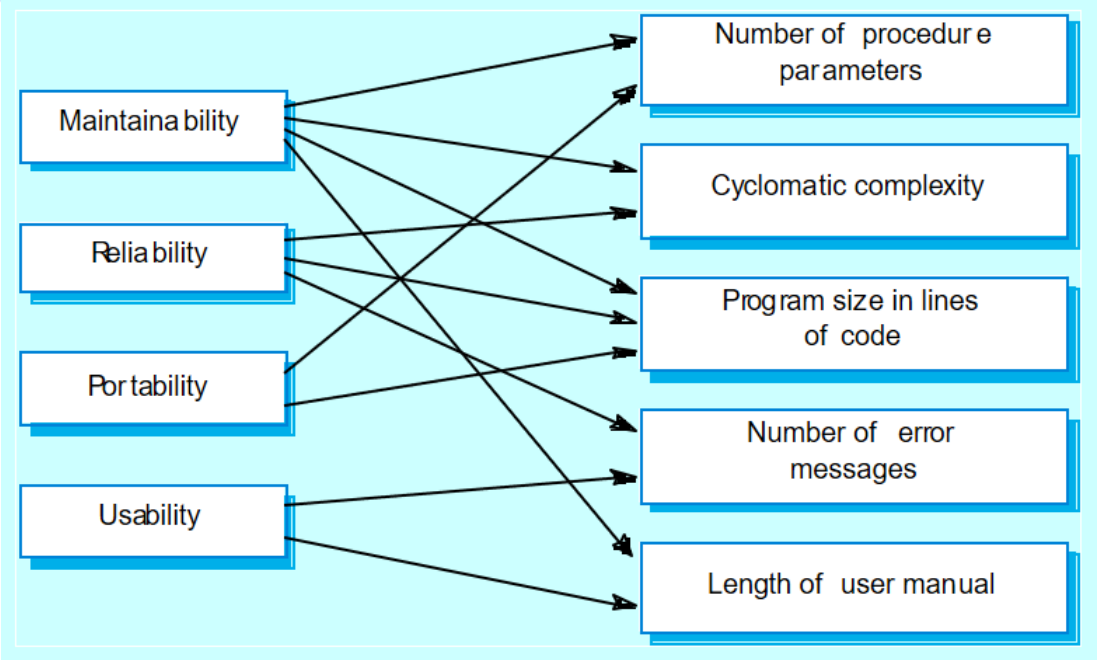
Predictor and control metrics



Metrics assumptions

- A software property can be measured.
- The relationship exists between what we can measure and what we want to know. We can only measure internal attributes but are often more interested in external software attributes.
- This relationship has been formalised and validated.
- It may be difficult to relate what can be measured to desirable external quality attributes.

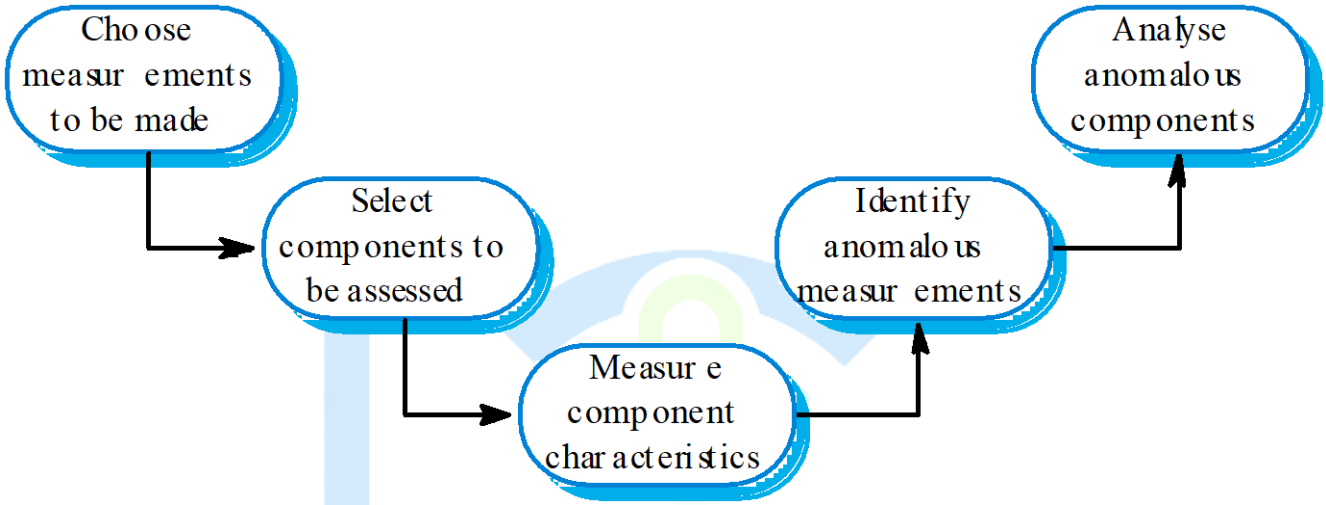
Internal and external attributes



The measurement process

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

Product measurement process



Data collection

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

Data accuracy

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

Product metrics

- A quality metric should be a predictor of product quality.

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

Dynamic and static metrics

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

Software product metrics

Software metric	Description
Fan in/Fan-out	Fan-in is a measure of the number of functions or methods that call some other function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
Length of code	This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss how to compute cyclomatic complexity in Chapter 22.
Length of identifiers	This is a measure of the average length of distinct identifiers in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.
Depth of conditional nesting	This is a measure of the depth of nesting of if-statements in a program. Deeply nested if statements are hard to understand and are potentially error-prone.
Fog index	This is a measure of the average length of words and sentences in documents. The higher the value for the Fog index, the more difficult the document is to understand.

Object-oriented metrics

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

Measurement analysis

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

Measurement surprises

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

- **Process improvement activities**
  - Process measurement
  - Process analysis and modelling
  - Process change

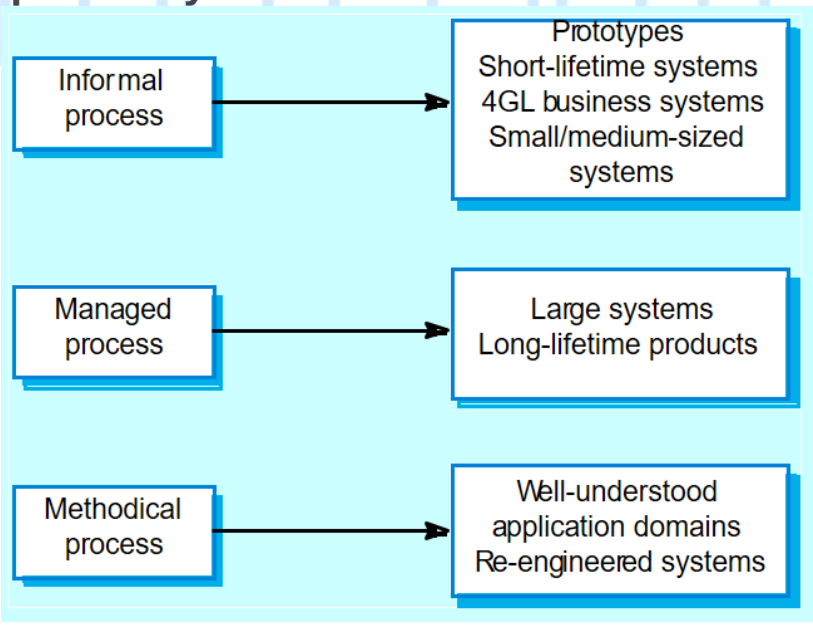
Process improvement

- Understanding existing processes and introducing process changes to improve product quality, reduce costs or accelerate schedules.
- Most process improvement work so far has focused on defect reduction. This reflects the increasing attention paid by industry to quality.
- However, other process attributes can also be the focus of improvement

Process classification

- Informal
  - No detailed process model. Development team chose their own way of working.
- Managed
  - Defined process model which drives the development process.
- Methodical
  - Processes supported by some development method such as the RUP.
- Improving
  - Continuous learning is built into the process.

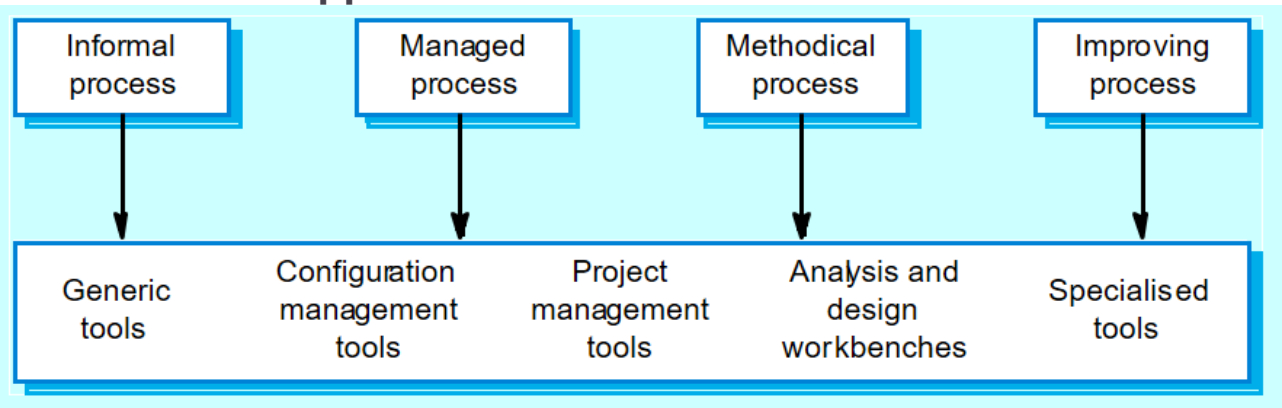
Process applicability



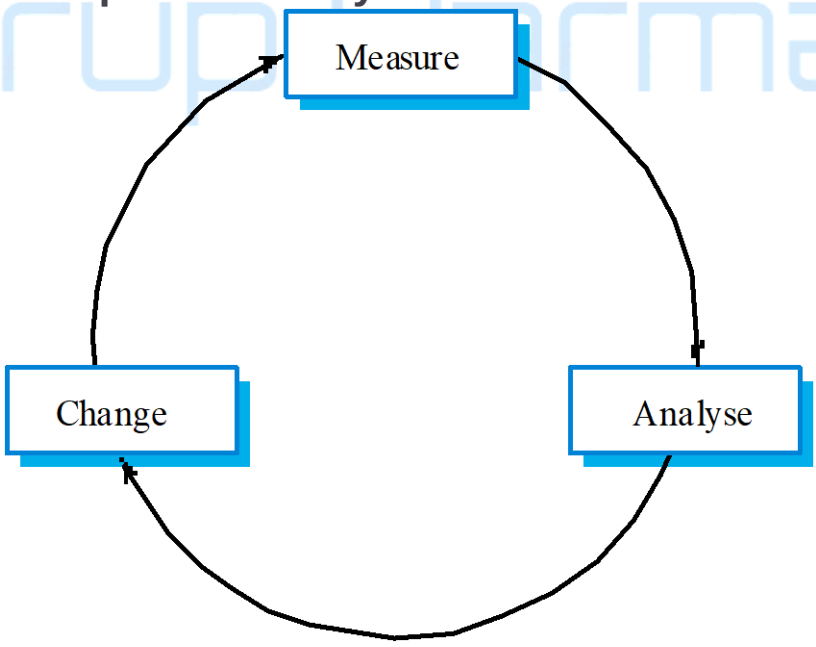
Process choice

- Process used should depend on type of product which is being developed
  - For large systems, management is usually the principal problem so you need a strictly managed process;
  - For smaller systems, more informality is possible.
- There is no uniformly applicable process which should be standardised within an organisation
  - High costs may be incurred if you force an inappropriate process on a development team;
  - Inappropriate methods can also increase costs and lead to reduced quality.

Process tool support



The process improvement cycle



Process improvement activities

- **Process measurement**
  - Attributes of the current process are measured. These are a baseline for assessing improvements.
- **Process analysis**
  - The current process is assessed and bottlenecks and weaknesses are identified.
- **Process change**
  - Changes to the process that have been identified during the analysis are introduced.

Process measurement

- Wherever possible, quantitative process data should be collected
  - However, where organisations do not have clearly defined process standards this is very difficult as you don't know what to measure. A process may have to be defined before any measurement is possible.
- Process measurements should be used to assess process improvements
  - But this does not mean that measurements should drive the improvements. The improvement driver should be the organizational objectives.

Process attributes

Process characteristic	Description
Understandability	To what extent is the process explicitly defined and how easy is it to understand the process definition?
Visibility	Do the process activities culminate in clear results so that the progress of the process is externally visible?
Supportability	To what extent can CASE tools be used to support the process activities?
Acceptability	Is the defined process acceptable to and usable by the engineers responsible for producing the software product?
Reliability	Is the process designed in such a way that process errors are avoided or trapped before they result in product errors?
Robustness	Can the process continue in spite of unexpected problems?

Maintainability	Can the process evolve to reflect changing organisational requirements or identified process improvements?
Rapidity	How fast can the process of delivering a system from a given specification be completed?

Classes of process measurement

- Time taken for process activities to be completed
  - E.g. Calendar time or effort to complete an activity or process.
- Resources required for processes or activities
  - E.g. Total effort in person-days.
- Number of occurrences of a particular event
  - E.g. Number of defects discovered.

Goal-Question-Metric Paradigm

- Goals
  - What is the organisation trying to achieve? The objective of process improvement is to satisfy these goals.
- Questions
  - Questions about areas of uncertainty related to the goals. You need process knowledge to derive these.
- Metrics
  - Measurements to be collected to answer the questions.

Process analysis and modelling

- Process analysis
  - The study of existing processes to understand the relationships between parts of the process and to compare them with other processes.
- Process modelling
  - The documentation of a process which records the tasks, the roles and the entities used;
  - Process models may be presented from different perspectives.
- Study an existing process to understand its activities.
- Produce an abstract model of the process. You should normally represent this graphically. Several different views (e.g. activities, deliverables, etc.) may be required.
- Analyse the model to discover process problems. This involves discussing process activities with stakeholders and discovering problems and possible process changes.

Process analysis techniques

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

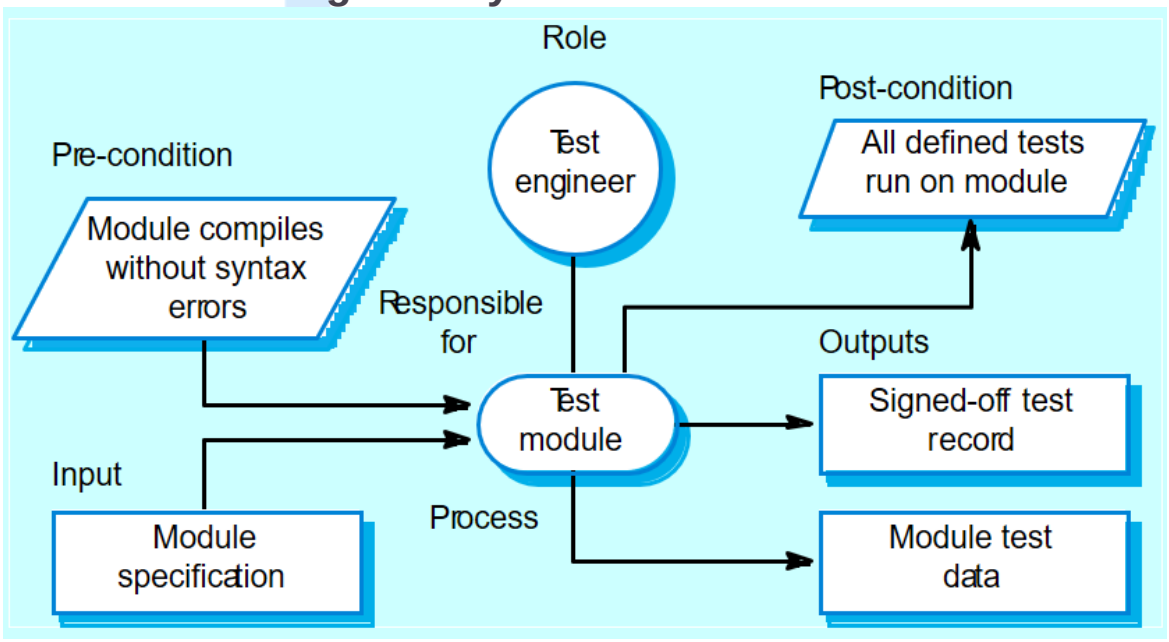
Process model elements 1

Activity (shown as a round-edged rectangle with no drop shadow)	An activity has a clearly defined objective, entry and exit conditions. Examples of activities are preparing a set of test data to test a module, coding a function or a module, proof-reading a document, etc. Generally, an activity is atomic i.e. it is the responsibility of one person or group. It is not decomposed into sub-activities.
Process (shown as a round-edged rectangle with drop shadow)	A process is a set of activities which have some coherence and whose objective is generally agreed within an organisation. Examples of processes are requirements analysis, architectural design, test planning, etc.
Deliverable (shown as a rectangle with drop shadow)	A deliverable is a tangible output of an activity that is predicted in a project plan.
Condition (shown as a parallelogram)	A condition is either a pre-condition that must hold before a process or activity can start or a post-condition that holds after a process or activity has finished.

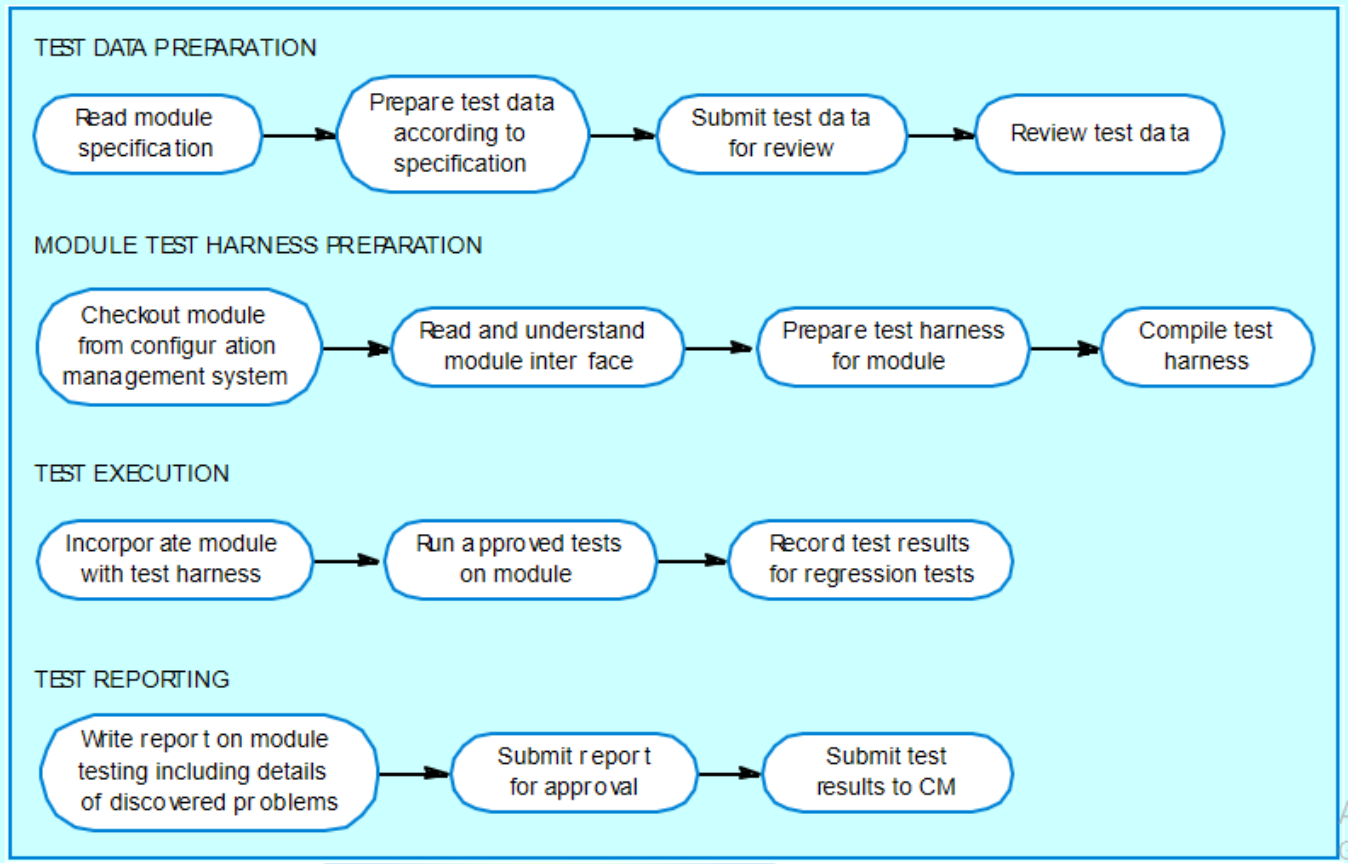
Process model elements 2

Role (shown as a circle with drop shadow)	A role is a bounded area of responsibility. Examples of roles might be configuration manager, test engineer, software designer, etc. One person may have several different roles and a single role may be associated with several different people.
Exception (not shown in examples here but may be represented as a double edged box)	An exception is a description of how to modify the process if some anticipated or unanticipated event occurs. Exceptions are often undefined and it is left to the ingenuity of the project managers and engineers to handle the exception.
Communication (shown as an arrow)	An interchange of information between people or between people and supporting computer systems. Communications may be informal or formal. Formal communications might be the approval of a deliverable by a project manager; informal communications might be the interchange of electronic mail to resolve ambiguities in a document.

The module testing activity



Activities in module testing



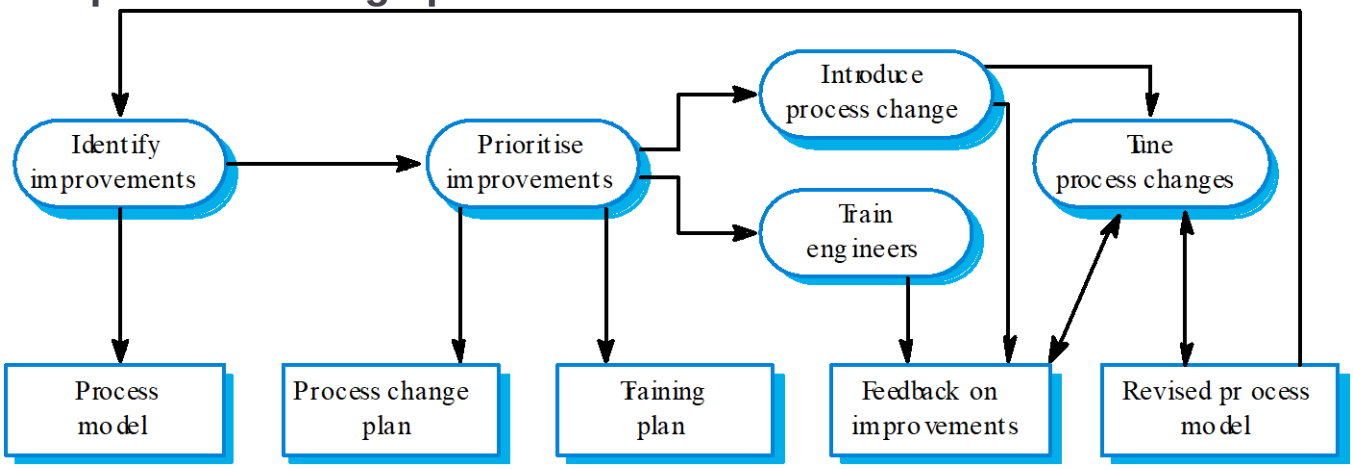
Process exceptions

- Software processes are complex and process models cannot effectively represent how to handle exceptions:
  - Several key people becoming ill just before a critical review;
  - A breach of security that means all external communications are out of action for several days;
  - Organisational reorganisation;
  - A need to respond to an unanticipated request for new proposals.
- Under these circumstances, the model is suspended and managers use their initiative to deal with the exception.

Process change

- Involves making modifications to existing processes.
- This may involve:
  - Introducing new practices, methods or processes;
  - Changing the ordering of process activities;
  - Introducing or removing deliverables;
  - Introducing new roles or responsibilities.
- Change should be driven by measurable goals.

The process change process



Process change stages

- Improvement identification.
- Improvement prioritisation.
- Process change introduction.
- Process change training.
- Change tuning.