# Module 1

## Critical Systems

**Definition:**

There are some systems where failure can result in significant losses, physical damage or threats to human life, these systems are usually called **critical system.**

There are three main types of critical systems:

- **Safety-critical systems**
  - Failure results in loss of life, injury or damage to the environment.
    **Ex:** Chemical plant protection system.
- **Mission-critical systems**
  - Failure results in failure of some goal-directed activity.
    **Ex:** Spacecraft navigation system.
- **Business-critical systems**
  - Failure results in high economic losses.
    **Ex:** Customer accounting system in a bank;

**System dependability:**

There are several reasons why dependability is the most important emergent property for critical systems:

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

**There are three 'system components' where critical systems failures may occur:**

- Systems that are not dependable and are unreliable, unsafe or insecure may be rejected by their users.
- The costs of system failure may be very high.
- Undependable systems may cause information loss with a high consequent recovery cost.
- System hardware may fail because of mistakes in its design, because components fail as a result of manufacturing errors, or because the components have reached the end of their natural life.

- System software may fail because of mistakes in its specifications, design or implementations.
- Human operator of the system may fail to operate the system correctly. As hardware and software have become more reliable, failures in operations are now probably the largest single cause of system failures.

**Socio-technical critical systems**

- Hardware failure
  - Operational failure
- Software failure
  - Software fails due to errors in its specification, design or implementation.
- Operational failure
  - Human operators make mistakes. Now perhaps the largest single cause of system failures.

## A simple safety-critical system

**A software-controlled insulin pump:**

- Used by diabetics to simulate the function of the pancreas which manufactures insulin, an essential hormone that metabolises blood glucose.
- Measures blood glucose (sugar) using a micro-sensor and computes the insulin dose required to metabolise the glucose.

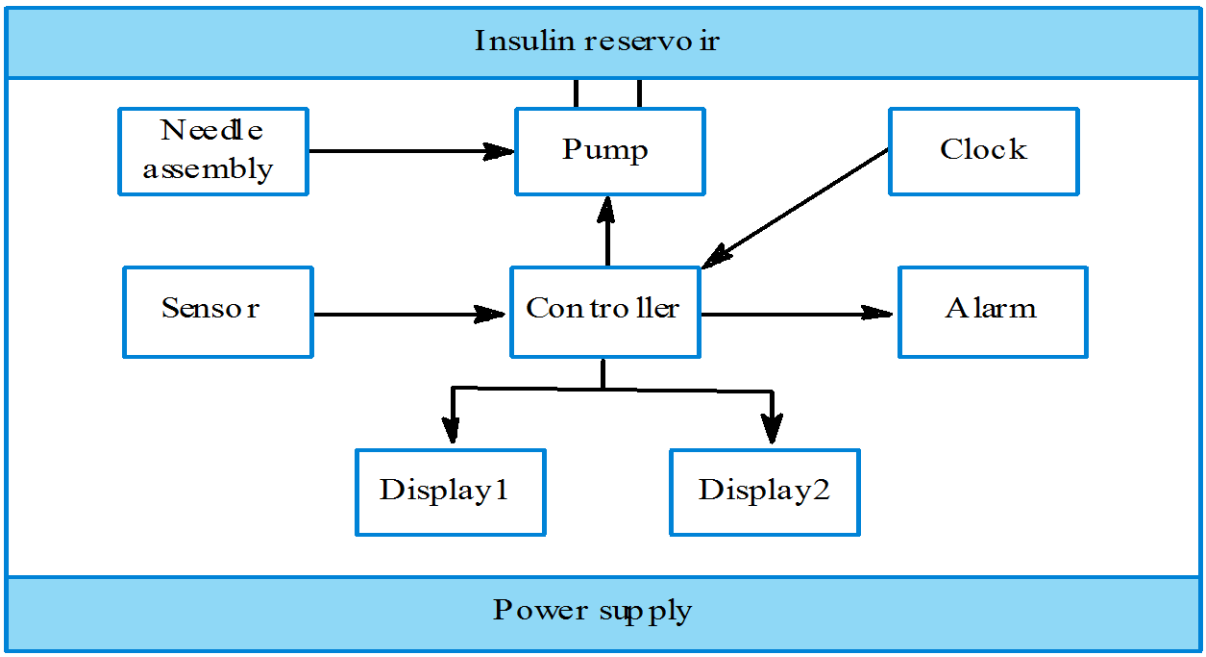



Figure: Insulin pump structure

- Three dimensions of dependability apply to this insulin delivery system:
  - **Availability:** it is important that the system should be available to deliver insulin when required.
  - **Reliability:** It is important that the system performs reliably and delivers the correct amount of insulin to counter the current level of blood sugar.
  - **Safety:** failure of the system could, in principle, cause excessive does of insulin to be delivered and this would threaten the life of the user.

Insulin pump data-flow:

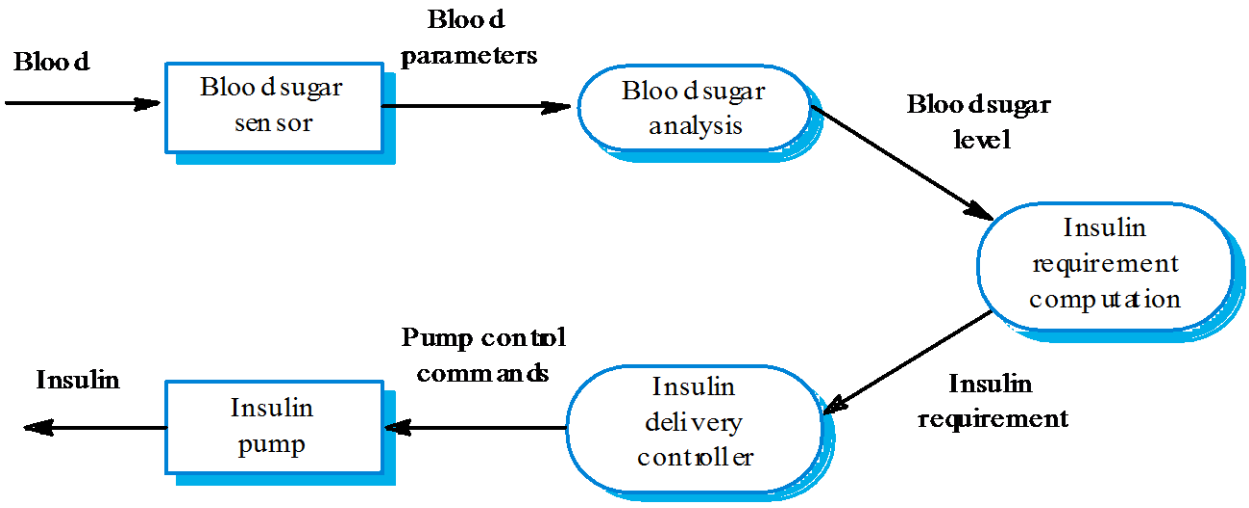



Figure: Data-flow model of the insulin pump

## System dependability

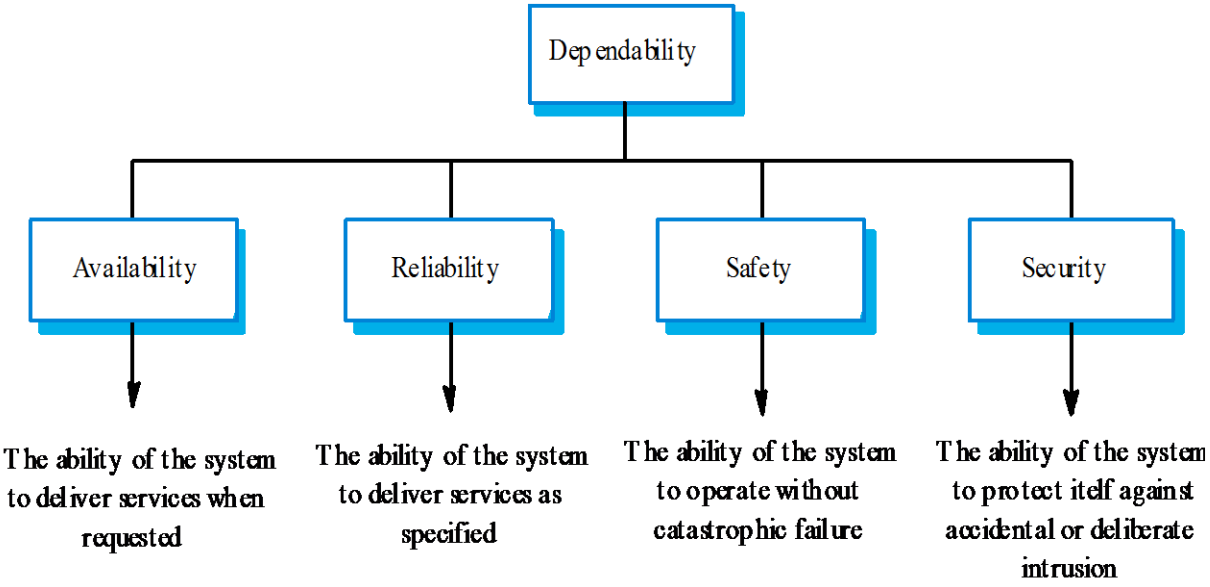
### Dependability requirements

- The system shall be available to deliver insulin when required to do so.
- The system shall perform reliability 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.

### Dependability

- The dependability of a system equates to its trustworthiness.
- A dependable system is a system that is trusted by its users

**Dimensions of dependability**




There are four principal dimensions to dependability, Principal dimensions of dependability are:

- Availability
- Reliability
- Safety
- Security

As well as these four main dimensions, other system properties can also be considered under the heading of dependability:

- **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 whilst under hostile attack.
- **Error tolerance**
  - Reflects the extent to which user input errors can be avoided and tolerated.

**Maintainability**

- A system attribute that is concerned with the ease of repairing the system after a failure has been discovered or changing the system to include new features

- Very important for critical systems as faults are often introduced into a system because of maintenance problems
- Maintainability is distinct from other dimensions of dependability because it is a static and not a dynamic system attribute. I do not cover it in this course.

**Survivability**

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

**Dependability vs Performance:**

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

**Dependability 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

Figure 3.4 shows the relationship between costs and incremental improvements in dependability.

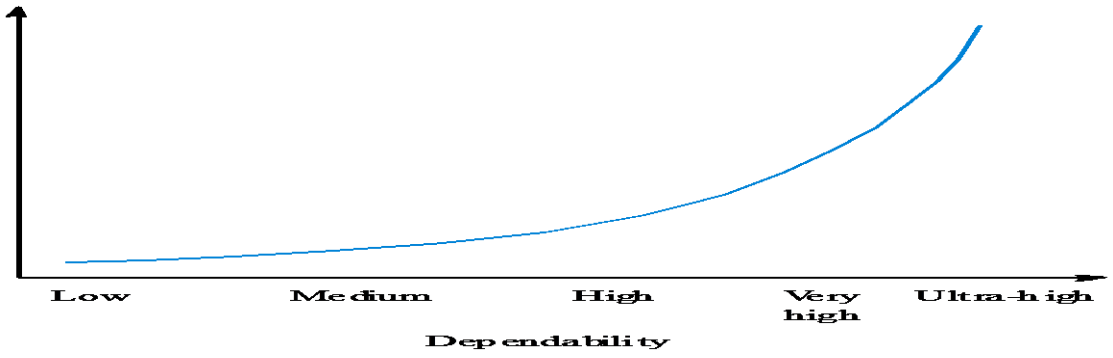



Figure: Cost/dependability curve

- The higher the dependability that you need, the more that you have to spend on testing to check that you have reached that level.

## Availability and reliability

System reliability and availability may be defined more precisely as follows:

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

**Reliability terminology**

| Term | Description |
|---|---|
| System failure | An event that occurs at some point in time when the system does not deliver a service as expected by its users |
| System error | 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. |

**Figure: Reliability terminology**

This distinction between the terms shown in Figure above helps us identify three complementary approaches that are used to improve the reliability of a system:

**Faults 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.

**Perceptions of reliability:**

- The formal definition of reliability does not always reflect the user's perception of a system's reliability
  - The assumptions that are made about the environment where a system will be used may be incorrect
    - Usage of a system in an office environment is likely to be quite different from usage of the same system in a university environment

- The consequences of system failures affects the perception of reliability
  - Unreliable windscreen wipers in a car may be irrelevant in a dry climate.
  - Failures that have serious consequences (such as an engine breakdown in a car) are given greater weight by users than failures that are inconvenient.

**Reliability achievement**

- **Fault avoidance**
  - Development technique are used that either minimize 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

# Software Processes

**Objectives:**

- To introduce software process models
- To describe three generic process models and when they may be used
- To describe outline process models for requirements engineering, software development, testing and evolution
- To explain the Rational Unified Process model
- To introduce CASE technology to support software process activities

Although there are many software processes, some fundamental activities are common to all software processes:

- A structured set of activities required to develop a software system
  - Specification: The functionality of the software and constraints on its operation must be defined.
  - Design: The software to meet the specification must be produced.
  - Validation: The software must be validated to ensure that it does what the customer wants.
  - Evolution: The software must evolve to meet changing customer needs.

## Software process models

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

**Generic software process models:**

- **The waterfall model:** This takes the fundamental process activities of specification, development, validation and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing and so on.
- **Evolutionary development:** This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from abstract specifications. This is then refined with customer input to produce a system that satisfies the customer s needs.
- **Component-based software engineering:** This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.
- There are many variants of these models e.g. formal development where a waterfall-like process is used but the specification is a formal specification that is refined through several stages to an implementable design.

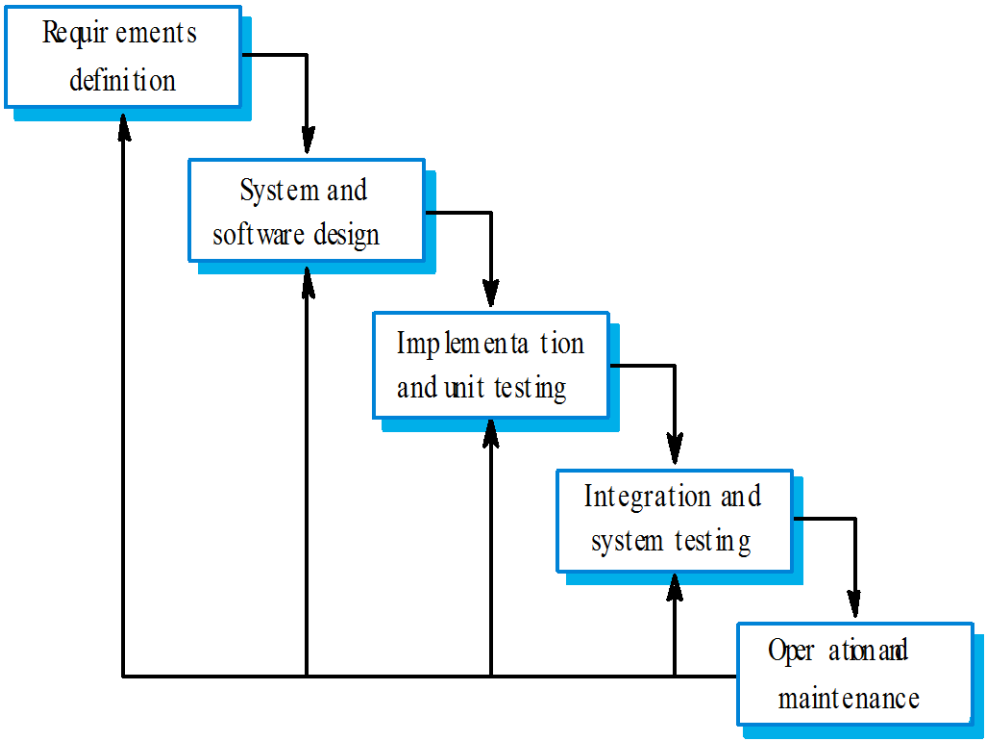



**Figure: The software life cycle**

## The waterfall model

Waterfall model phases:

- **Requirements analysis and definition:**
  - The system's services, constraints and goals are established by consultation with system users.
- **System and software design:**
  - The systems design process partitions the requirements to either hardware or software systems.
  - It establishes overall system architecture.
  - Software design involves identifying and describing the fundamental software system abstractions and their relationships.
- **Implementation and unit testing:**
  - During this stage, the software design is realized as a set of programs or program units.
  - Unit testing involves verifying that each unit meets its specification.
- **Integration and system testing:**
  - The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met.
  - After testing, the software system is delivered to the customer.
- **Operation and maintenance:**
  - Normally (although not necessarily) this is the longest life-cycle phase.
  - The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

**Waterfall model problems:**

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

- Few business systems have stable requirements.
- The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.

## Evolutionary development

There are two fundamental types of evolutionary development:

- **Exploratory development**
  - Objective is to work with customers and to evolve a final system from an initial outline specification.
  - The development starts with well-understood requirements and adds new features as proposed by the customer.
- **Throw-away prototyping**
  - Objective is to understand the system requirements. Should start with poorly understood requirements to clarify what is really needed.

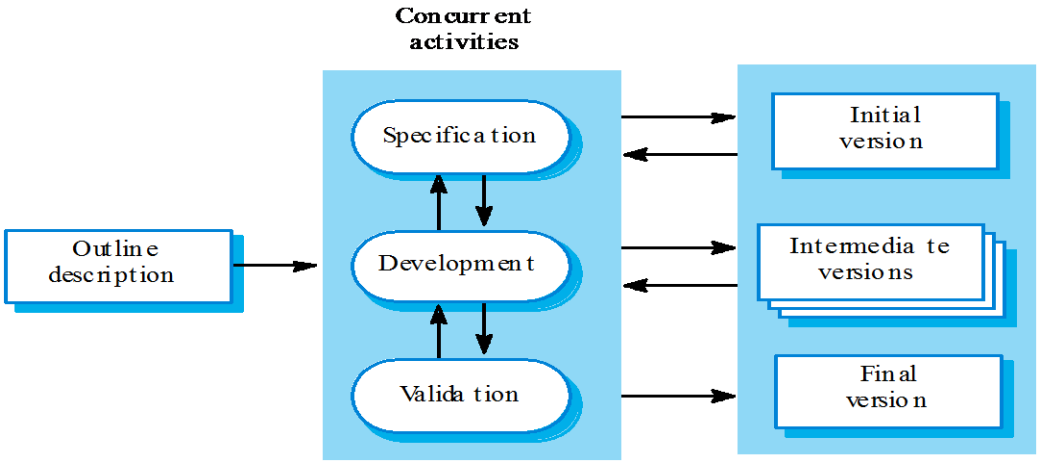



**Figure: Evolutionary development**

**The evolutionary approach has two problems:**

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

**Systems are often poorly structured**: Continual change tends to corrupt the software structure. Incorporating software changes becomes increasingly difficult and costly.

- Special skills: These allow for rapid development but they may be incompatible with other tools or techniques & relatively few people may have the skills which are needed to use them.

**Applicability**

- For small or medium-size interactive systems.
- For parts of large systems (e.g. the user interface).
- For short-lifetime systems.

## Component based software engineering

- Based on systematic reuse where systems are integrated from existing components or COTS (Commercial-off-the-shelf) systems.
- Process stages
  - **Component analysis:** Given the requirements specification, a search is made for components to implement that specification.
  - **Requirements modification:** During this stage, the requirements are analysed using information about the components that have been discovered. They are then modified to reflect the available components.
  - **System design with reuse:** During this phase, the framework of the system is designed or an existing framework is reused. Some new software may have to be designed if reusable components are not available.
  - **Development and integration:** Software that cannot be externally procured is developed, and the components and COTS systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.

**Reuse-oriented development**

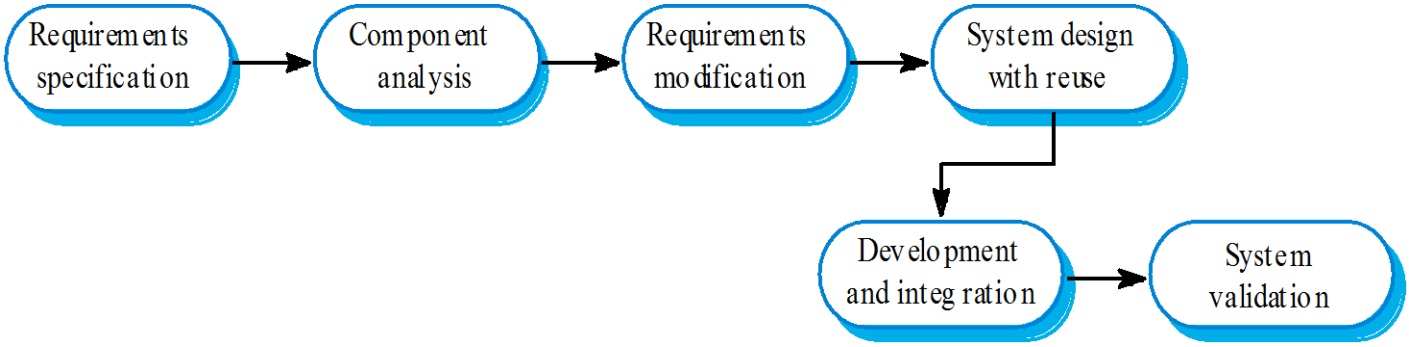



**Figure: Component-based software engineering**

## Process iteration

- System requirements always evolve in the course of a project so process iteration where earlier stages are reworked is always part of the process for large systems.
- Iteration can be applied to any of the generic process models.
- There are two process models that have been explicitly designed to support process iteration:
- Two (related) approaches
  - Incremental delivery
  - Spiral development

## Incremental delivery:

- Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- User requirements are prioritized and the highest priority requirements are included in early increments.
- Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.

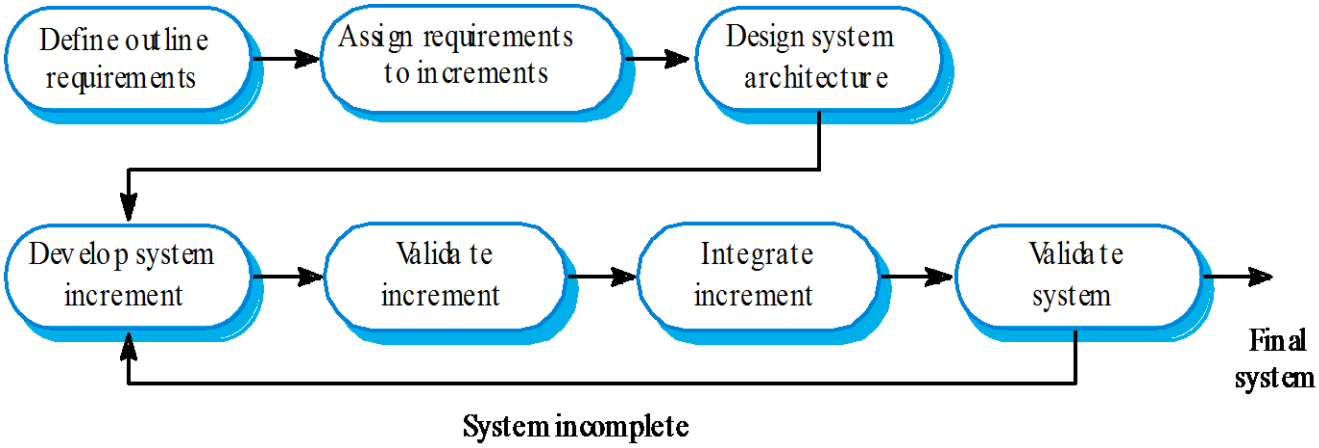



**Figure:; Incremental delivery**

### Incremental development advantages:

- Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements so they can use the software immediately.
- Customers can use the early increments as prototypes and gain experience that informs

their requirements for later system increments.

- There is a lower risk of overall project failure.
- As the highest priority services are delivered first, and later increments are integrated with them, it is inevitable that the most important system services receive the most testing.

A variant of this incremental approach called **extreme programming** has been developed:

- An approach to development based on the development and delivery of very small increments of functionality.
- Relies on constant code improvement, user involvement in the development team and pairwise programming.

## Spiral development:

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

**Spiral model of the software process:**

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

- **Objective setting**: Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified.
- **Risk assessment and reduction:** For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk.
- **Development and validation:** After risk evaluation, a development model for the system is chosen.
- **Planning:** The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

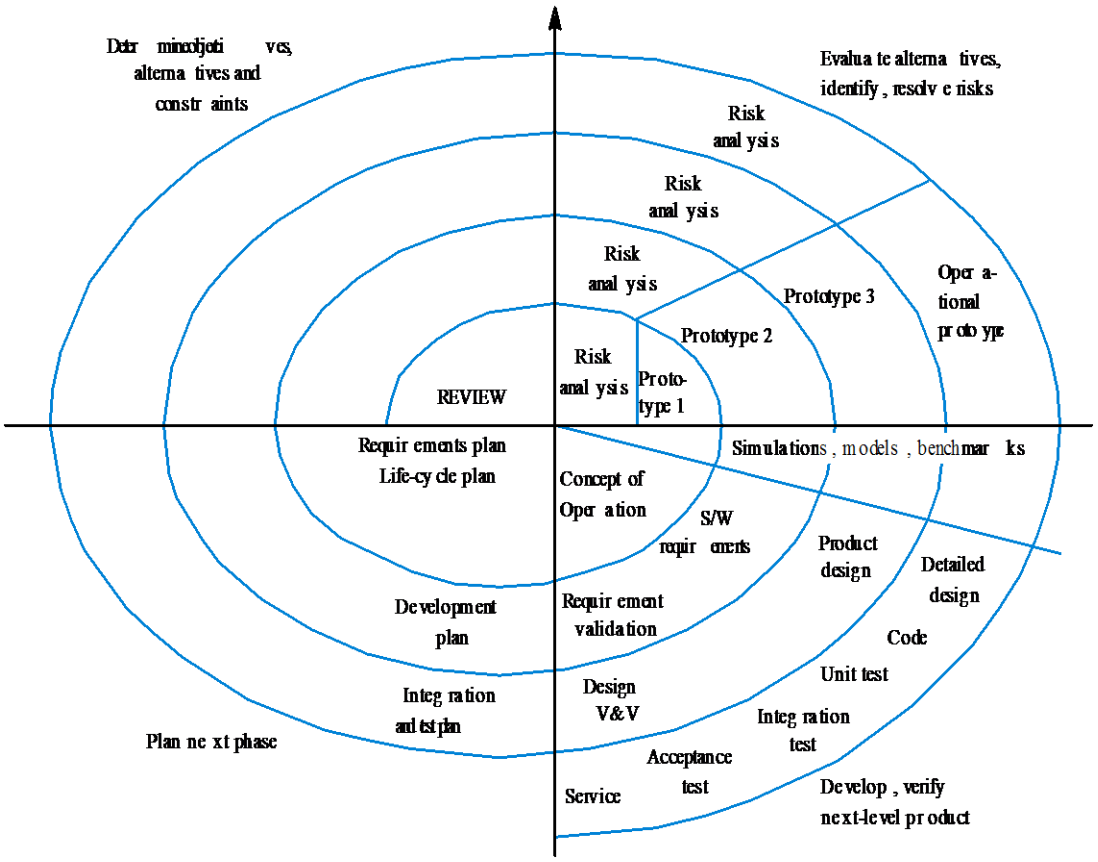



**Figure: Boehm's spiral model of the software process**

## Process activities:

- Software specification
- Software design and implementation
- Software validation
- Software evolution

## Software specification:

- The process of establishing what services are required and the constraints on the system's operation and development.
- There are four main phases in the requirements engineering process:

➢ **Feasibility study:** An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study considers whether the proposed system will be cost-effective from a business point of view and whether it can be developed within existing budgetary constraints.

➢ **Requirements elicitation and analysis:** This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis and so on.

- **Requirements specification:** The activity of translating the information gathered during the analysis activity into a document that defines a set of requirements.
- Two types of requirements may be included in this document. **User requirements** are abstract statements of the **system requirements** for the customer and end-user of the system.
- **Requirements validation:** This activity checks the requirements for realism, consistency and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

## The requirements engineering process:

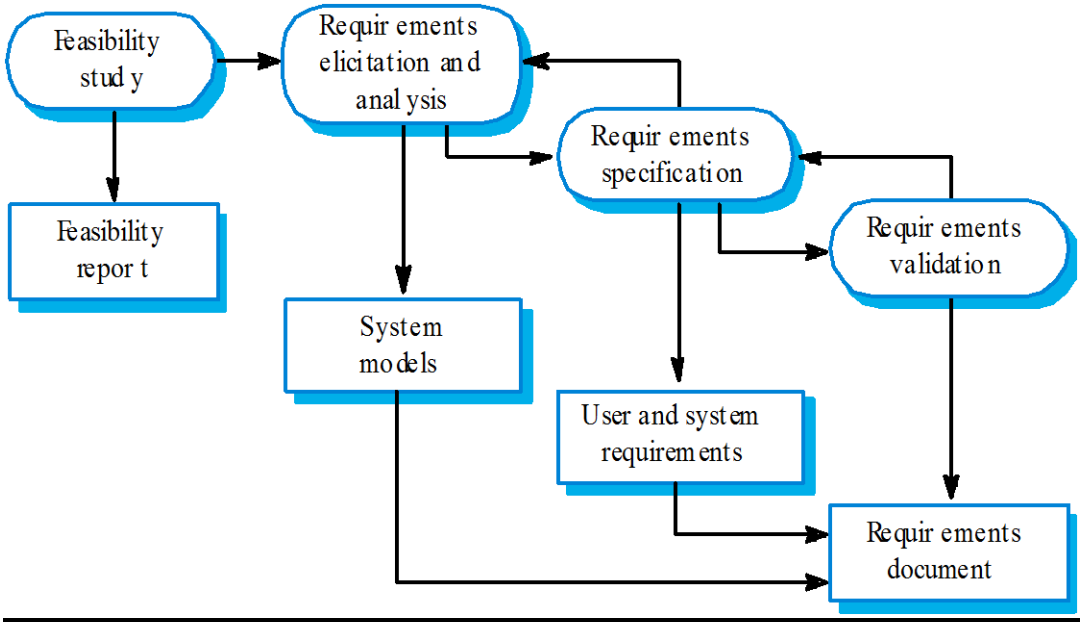



**Figure: The requirements engineering process**

## Software design and implementation:

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

- **Software design**
  - Design a software structure that realizes the specification
- **Implementation**
  - Translate this structure into an executable program
- The activities of design and implementation are closely related and may be inter-leaved.

**The specific design process activities are:**

- Architectural design: The sub-systems making up the system and their relationships are identified and documented.
- **Abstract specification:** For each sub-system an abstract specification of its services and the constraints under which it must operate is produced.
- **Interface design:** For each sub-system, its interface with other sub-systems is designed and documented. This interface specification must be unambiguous as it allows the sub-system to be used without knowledge of the sub-system operation.
- **Component design:** Services are allocated to components and the interfaces of these components are designed.
- **Data structure design:** The data structures used in the system implementation are designed in detail and specil1ed.
- **Algorithm design:** The algorithms used to provide services are designed in detail and specified.

**The software design process:**

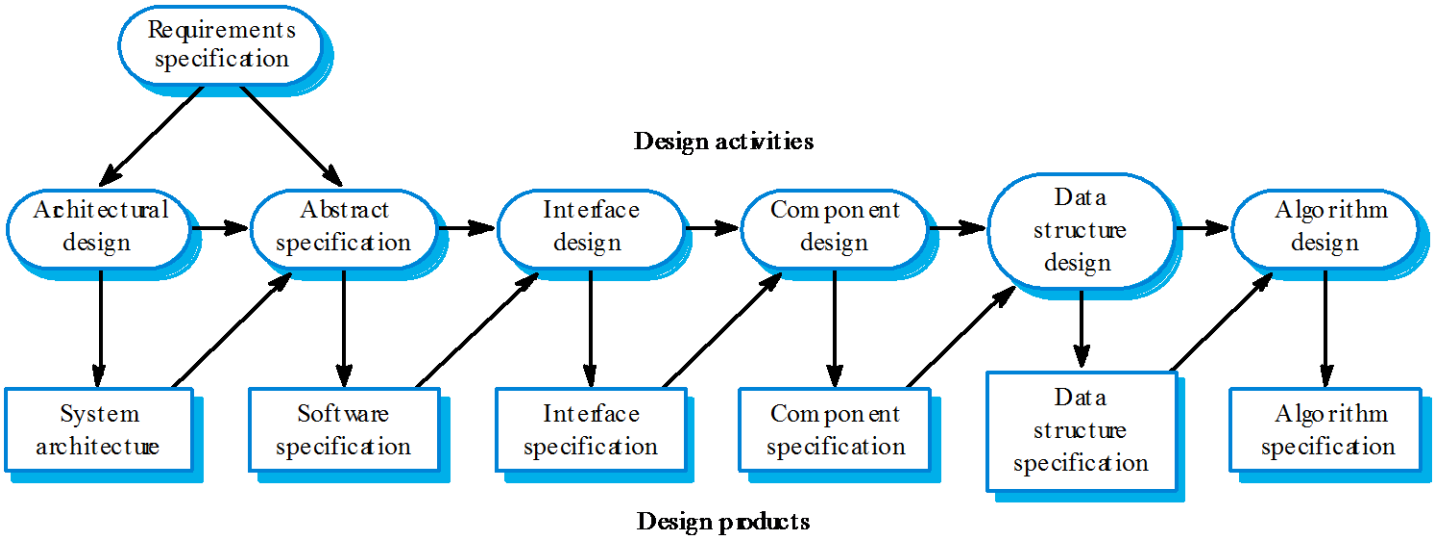



**Figure: A general model of the design process**

This is a general model of the design process and real, practical processes may adapt it in different ways.

**Possible adaptations are**:

- The last two stages of design-data structure and algorithm design-may be delayed until the implementation process.
- If an exploratory approach to design is used, the system interfaces may be designed after

the data structures have been specified.

- The abstract specification stage may be skipped, although it is usually an essential part of critical systems design.

A contrasting approach is taken by structured methods for design that rely on producing graphical models of the system (Structured methods):

- Systematic approaches to developing a software design.
- The design is usually documented as a set of graphical models.
- Structured methods may support some or all of the following models of a system:
  - Object model
  - Sequence model
  - State transition model
  - Structural model
  - Data-flow model

## Programming and debugging:

- Translating a design into a program and removing errors from that program.
- Programming is a personal activity - there is no generic programming process.
- Programmers carry out some program testing to discover faults in the program and remove these faults in the debugging process.
- The debugging process:

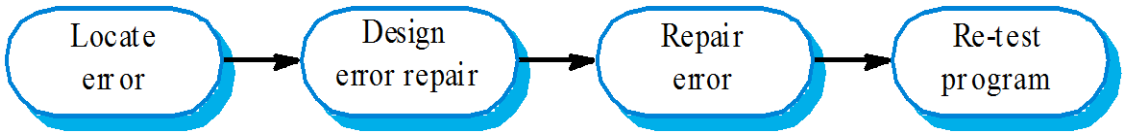



**Figure: The debugging process**

## Software validation:

- Verification and validation (V & V) is intended to show that a system conforms to its specification and meets the requirements of the system customer.
- Involves checking and review processes and system testing.
- System testing involves executing the system with test cases that are derived from the specification of the real data to be processed by the system.

## The testing process:

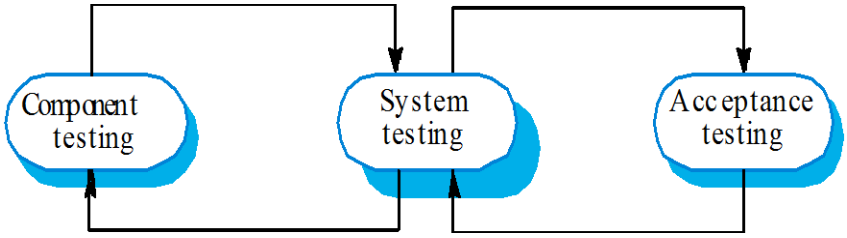



**Figure: The testing process**

## Testing stages:

- **Component or unit testing**
  - Individual components are tested independently.
  - Components may be functions or objects or coherent groupings of these entities.
- **System testing**
  - Testing of the system as a whole. Testing of emergent properties is particularly important.
  - This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems.
  - It is also concerned with validating that the system meets its functional and non-functional requirements and testing the emergent system properties.
- **Acceptance testing**
  - This is the final stage in the testing process before the system is accepted for operational use.
  - The system is tested with data supplied by the system customer rather than with simulated test data.
  - Acceptance testing may also reveal requirements problems where the system's facilities do not really meet the user s needs or the system performance is unacceptable.

## Testing phases:

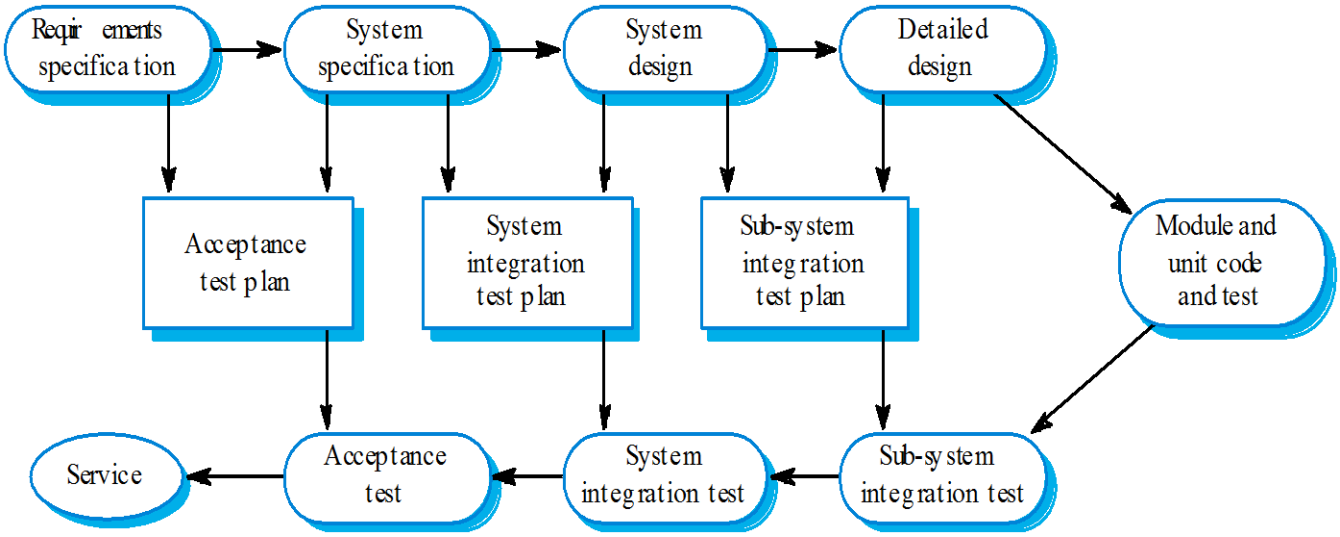



**Figure: Testing phases in the software process**

## Software evolution:

- Software is inherently flexible and can change.
- As requirements change through changing business circumstances, the software that supports the business must also evolve and change.
- Although there has been a demarcation between development and evolution (maintenance) this is increasingly irrelevant as fewer and fewer systems are completely new.

## System evolution:

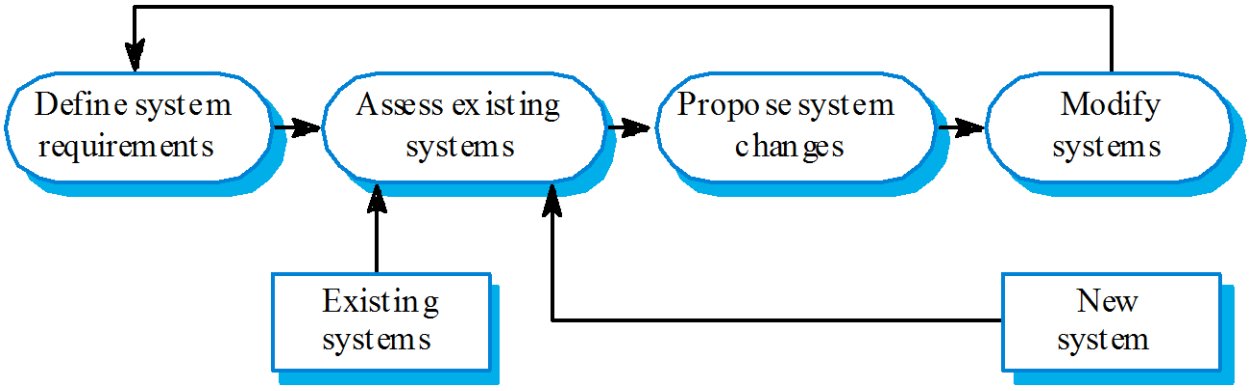



**Figure: System evolution**