2nd unit

3.Critical systems

Contents

3.1 A simple safety-critical system

3.2 System dependability

3.3 Availability and reliability

3.4 Safety

3.5 Security

Citical Systems? :are those whose failure can result in significant economic losses, physical damage or threats to human life.

Critical systems are technical or socio-technical systems that people or businesses depend on. If these systems fail to deliver their services as expected then serious problems and significant losses may result.

There are three main types of critical systems:

1. ***Safety-critical systems*** A system whose failure may result in injury, loss of life or serious environmental damage.

Example: control system for chemical manufacturing plant.

2. ***Mission-critical systems*** A system whose failure may result in the failure of some goal-directed activity.

Example : navigational system for a spacecraft.

3. ***Business-critical systems*** A system whose failure may result in very high costs for the business using that system.

Example :customer accounting system in a bank.

<u>System Dependability</u>

1 .Most important property of a critical system is the dependability.

2. The term *dependability* was proposed by Laprie (Laprie 1995) to cover the related systems attributes of availability, reliability, safety and security.

3 .Dependability of a system reflects the user's degree of trust in that system

<u>Importance of dependability</u>

***a).Systems that are unreliable, unsafe or insecure are often rejected by their users*.**

If users don't trust a system, they will refuse to use it and buy products from the same company as the untrustworthy system

b).System failure costs may be enormous.

Example:For some applications, such as a an aircraft navigation system, the cost of system failure is High than the cost of the control system.

c).Untrustworthy systems may cause information loss. Data is very expensive to collect and maintain. A great deal of effort and money may have to be spent duplicating valuable data to guard against data corruption.

4. Critical systems are usually developed using well-tried techniques, whose strengths and weaknesses are understood rather than newer techniques that have not been subject to extensive practical experience.

5. Critical systems are socio-technical systems where people monitor and control the operation of computer-based systems. The costs of critical systems failure are usually so high that we need people in the system who can cope with unexpected situations, and who can often recover from difficulties when things go wrong.

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

a). System hardware failure This may because of mistakes in its design, and manufacturing errors, or because the components have reached the end of their natural life.

b). System software failure This may because of mistakes in its specification, design or implementation.

c). **Operational Failure** Human operators of the system may fail to operate the system correctly. As hardware and software have become more reliable, failures in operation are now probably the largest single cause of system failures.

These failures can be interrelated.

-A failed hardware component may mean system operators have to cope with an unexpected situation and additional workload.

-This puts them under stress—and people under stress often make mistakes. This can cause the software to fail, which means more work for the operators, more stress, and so on.

3.1 A simple safety-critical system

1-Here A medical system that simulates the operation of the pancreas is chosen as a example to specify why safety and reliability are so important for this type of critical system.

2-The system chosen is intended to help people who suffer from diabetes.

-Diabetes is a relatively common condition where the human pancreas is unable to produce sufficient quantities of a hormone called *insulin*.

-Insulin metabolises glucose in the blood.

-Since level of insulin in the blood does not just depend on the blood glucose level but is a function of the time when the insulin injection was taken. This can lead to very low levels of blood glucose (if there is too muchinsulin) or very high levels of blood sugar (if there is too little insulin). This results in more serious health problems

3. These problems results in development of safety critical system

-A software-controlled insulin delivery system might work by using a micro-sensor embedded in the patient to measure blood parameter that is proportional to the sugar level.

- This is then sent to the pump controller.

-This controller computes the sugar level and the amount of insulin that is needed.

- It then sends signals to a miniaturised pump to deliver the insulin via a permanently attached needle.

4. Figure 3.1 shows the components and organisation of the insulin pump. Figure 3.2 is a data-flow model that illustrates how an input blood sugar level is transformed to a sequence of pump control commands.

Figure 3.1 Insulin pump structure

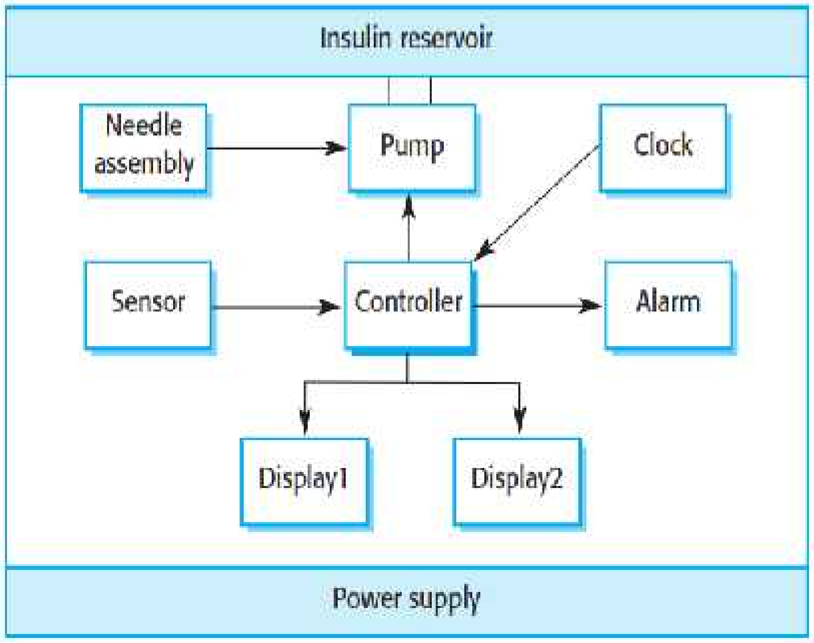
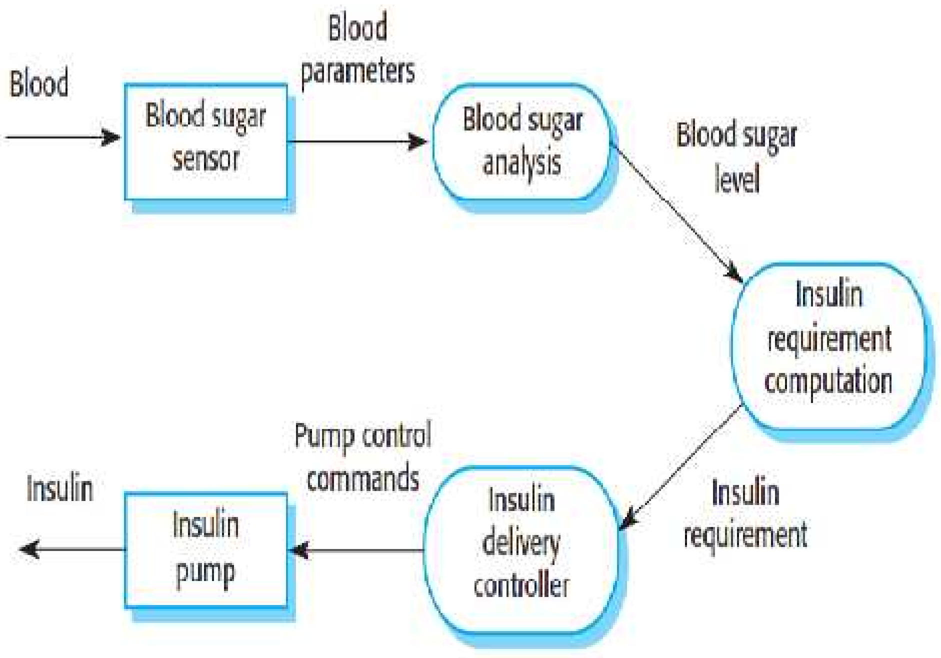


Figure 3.2 Data-flow model of the insulin pump



***5**. There are two high-level dependability requirements for this insulin pump system:*

a).The system shall be available to deliver insulin when required.

b).The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.

3.2 System dependability

1-The dependability of a computer system is a property of the system that equates to its trustworthiness.

2- Trustworthiness essentially means the degree of user confidence that the system will operate as they expect and that the system will not 'fail' in normal use.

3-There are four principal dimensions to dependability, as shown in Figure 3.3:

a). ***Availability*** Informally, the availability of a system is the probability that it will be up and running and able to deliver useful services at any given time.

b). ***Reliability*** Informally, the reliability of a system is the probability, over given period of time, that the system will correctly deliver services as expected by the user.

c) ***Safety*** Informally, the safety of a system is a judgement of how likely it is that the system will cause damage to people or its environment.

d). *Security* Informally, the security of a system is a judgement of how likely it is that the system can resist accidental or deliberate intrusions.

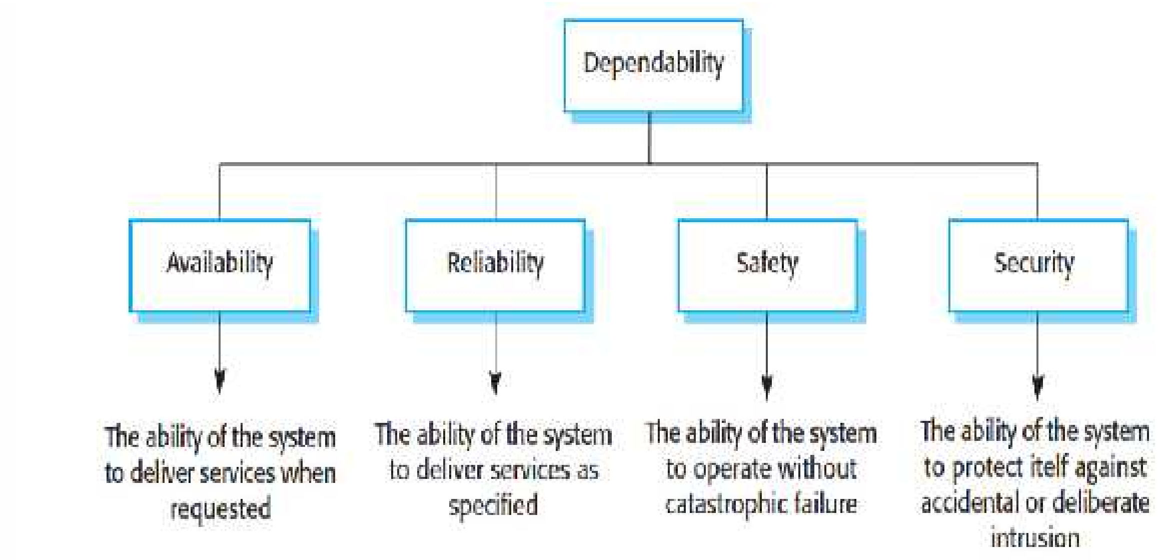

Dependability
Availability
Reliability
Safety
Security
The ability of the system to deliver services when requested
The ability of the system to deliver services as specified
The ability of the system to operate without catastrophic failure
The ability of the system to protect itelf against accidental or deliberate intrusion

Figure 3.3
Dimensions of dependability

These are complex properties that can be decomposed into a number of other,simpler properties.

For example, ***security*** includes

Integrity-ensuring that the systems program and data are not damaged

confidentiality-ensuring that information can only be accessed by people who are authorised.

Reliability includes

correctness -ensuring the system services are as specified

precision-ensuring information is delivered at an appropriate level of detail

timeliness -ensuring that information is delivered when it is required.

The dependability properties of availability, security, reliability and safety are all interrelated.

4. As well as these four main dimensions, other system properties can also be considered under dependability:

a) ***Repairability*** System failures are inevitable, but the disruption caused by failure can be minimised if the system can be repaired quickly.

b). ***Maintainability*** As systems are used, new requirements emerge. It is important to maintain the usefulness of a system by changing it to accommodate these new requirements.

c) ***Survivability*** It is the ability of a system to continue to deliver service whilst it is under attack and, potentially, while part of the system is disabled.

d). ***Error tolerance*** This property can be considered as part of usability , reflects the extent to which the system has been designed so that user input error are avoided and tolerated. When user errors occur, the system should, as far as possible, detect these errors and either fix them automatically or request the user to re-input their data.

5. <u>**Relationship between cost and dependability**</u>

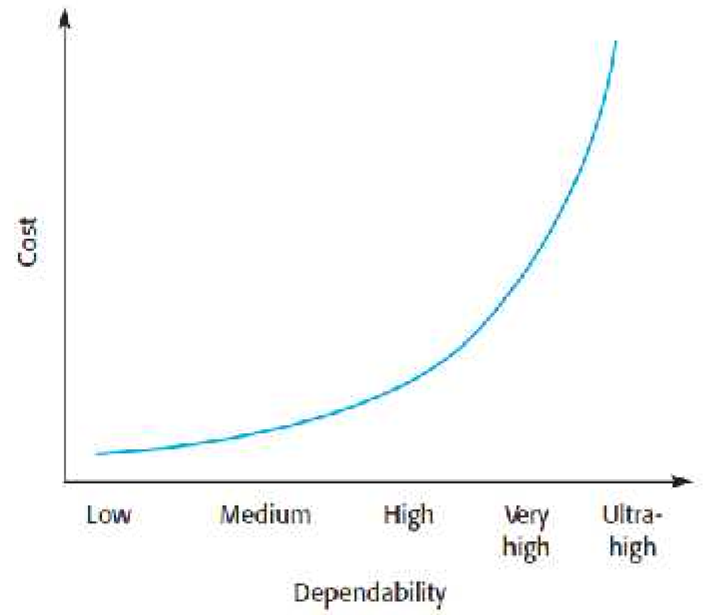
A .Dependable software includes extra, often redundant, code to perform the necessary checking for exceptional system states and to recover from system faults.

B. This reduces system performance and increases the costs of system development.

C. Because of extra, additional design, implementation and validation costs, increasing the dependability of a system can significantly increase development costs

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

Figure 3.4
Cost/dependability
curve



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

F. Because of the exponential nature of this cost/dependability curve, it is not possible to demonstrate that a system is 100% dependable, as the costs of dependability assurance would then be infinite.

3.3 Availability and reliability

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

Reliability The probability of failure-free 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.

2. Difference between Availability and Reliability

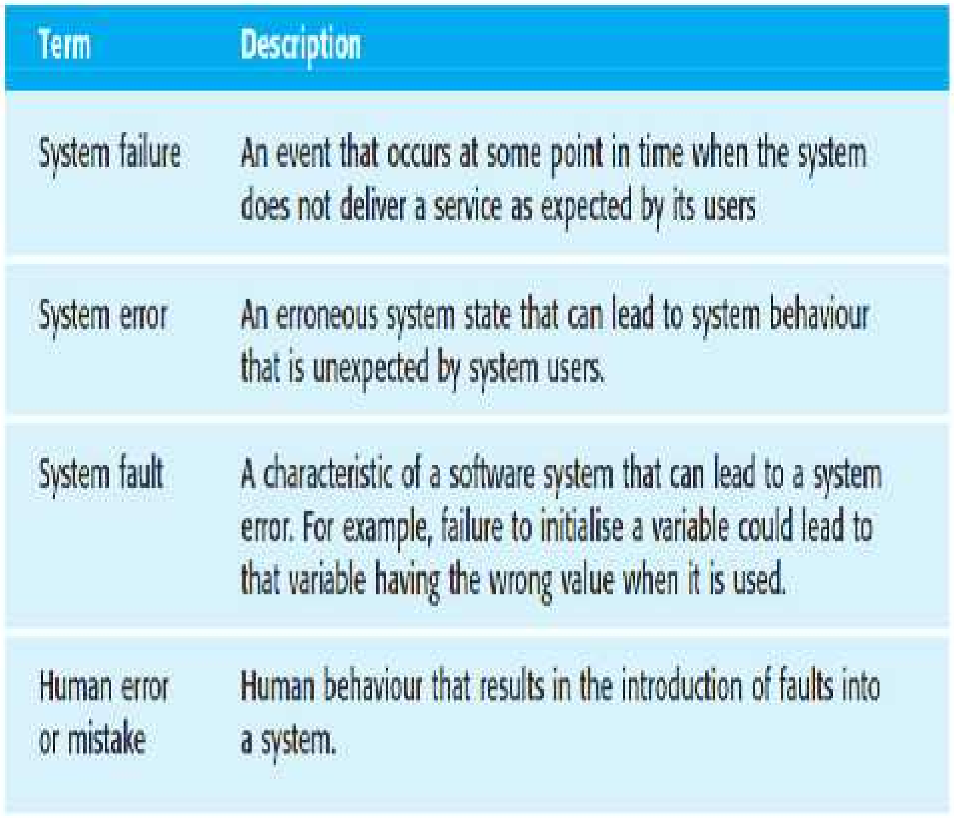
Availability	**Reliability**
Availability does not simply depend on the system itself but also on the time needed to repair the faults that make the system unavailable. *Example*: assume that system A takes three days to restart after a failure, whereas system B takes 10 minutes to restart. The availability of system B over the year (120 minutes of down time) is much better than that of system A	Reliability depends on the system itself. *Example*: system A fails once per year, and system B fails once per month, then A is clearly more reliable then B.
The probability that a system, at a point in time, will be operational and able to deliver the requested services.	The probability of failure-free operation over a specified time in a given environment for a specific purpose.
Environment is not taken into consideration ,the system should be operational to deliver the requested service	Environment in which the system is used and the purpose that it is used for must be taken into account. -reliability in one environment, can't be the same in another environment where the system is used in a different way. Example:,reliability of a word processor in an office environment where most users do not try to experiment with the

	system. Where as in a university environment, students may explore the boundaries of the system and use the system in unexpected ways. These may result in system failures that did not occur in the more constrained office environment.
Human perceptions and patterns of use are not significant. The significant action is to be operational, to deliver the requested service	Human perceptions and patterns of use are also significant. Example, say a car has a fault in its windscreen wiper system that results in failures of the wipers to operate correctly in heavy rain. The reliability of that system as perceived by a driver depends on where they live and use the car. A driver in Wet climate will probably be more affected by this failure and it will make system unreliable than a driver in dry climate who will not notice this problem only.

3 -Reliability and availability are compromised by system failures. These may be a failure to provide a service, a failure to deliver a service as specified, or the delivery of a service in such a way that is unsafe or insecure.

4. When discussing reliability, t is helpful to distinguish between the terms *fault*, *error* and *failure*, defined these terms in Figure 3.5

Figure 3.5 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.

5. Complementary approaches that are used to improve the reliability of a system are:

a) ***Fault avoidance*** Development techniques are used that either minimise the possibility of mistakes and/or that trap mistakes before they result in the introduction of system faults.

Examples : avoiding error-prone programming language constructs such as pointers and the use of static analysis to detect program anomalies.

b) ***Fault detection and removal*** The use of verification and validation techniques that increase the chances that faults will be detected and removed before the system is used.

Example : Systematic system testing and debugging

c). ***Fault tolerance*** Techniques that ensure that faults in a system do not result in system errors or that ensure that system errors do not result in system failures.

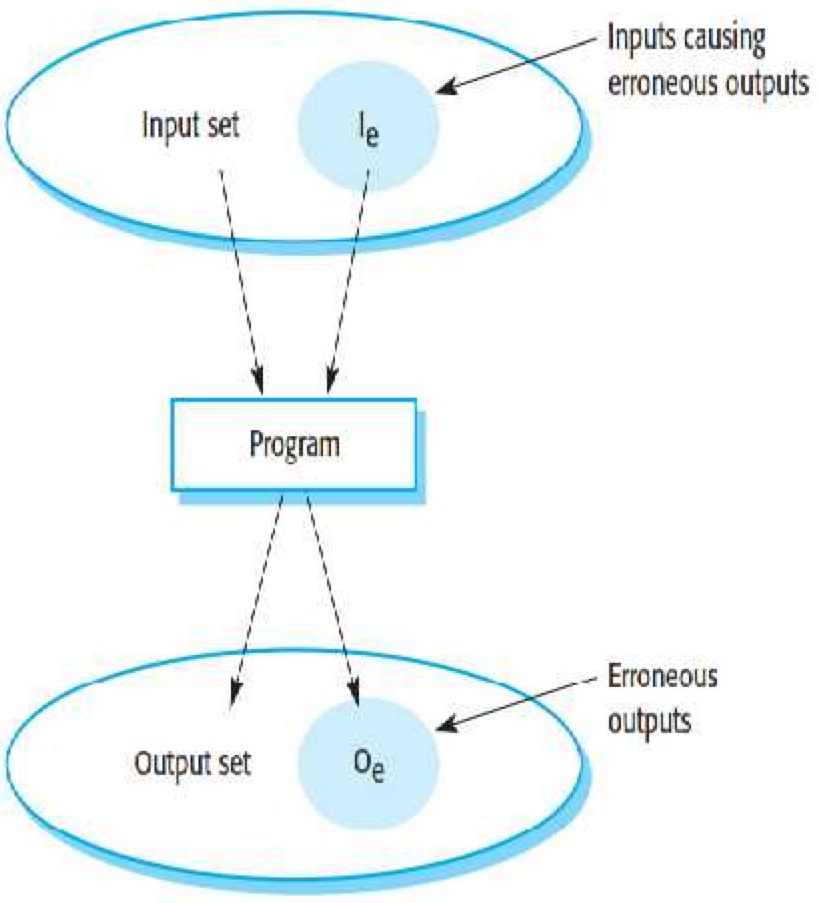
Examples: The incorporation of self-checking facilities in a system and the use of redundant system modules.

Reliability Modelling

-Software faults cause software failures when the faulty code is executed with a set of inputs.

- Figure 3.6, derived from Littlewood shows a software system as a mapping of an input to an output set. Given an input or input sequence, the program responds by producing a corresponding output

Figure 3.6 A system as an input/output mapping



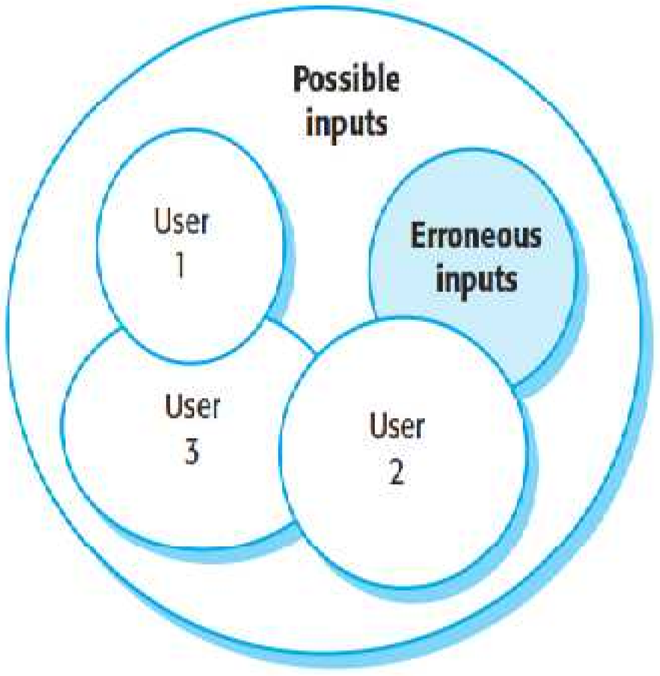
1.Some of these inputs or input combinations, shown in the shaded ellipse in Figure 3.6, cause erroneous outputs to be generated.

2.If an input causing an erroneous output is associated with a frequently used part of the program, then failures will be frequent. However, if it is associated with rarely used code, then users will hardly ever see failures.

3. Each user of a system uses it in different ways. Faults that affect the reliability of the system for one user may never be revealed under someone else's mode of working .

4. In Figure 3.7, the set of erroneous inputs correspond to the shaded ellipse in Figure 3.6. The set of inputs produced by User 2 intersects with this erroneous input set. User 2 will therefore experience some system failures. User 1 and User 3, however, never use inputs from the erroneous set. For them, the software will always be reliable.

Figure 3.7 Software usage patterns



2nd Unit

4. Software processes

Contents

4.1 Software process models

4.2 Process iteration

4.3 Process activities

4.4 The Rational Unified Process

4.5 Computer-Aided Software Engineering

A software process is a set of activities that leads to the production of a software product.

These activities involve

1.The development of software from scratch in a standard programming language like Java or C.

2. or new software is developed by extending and modifying existing systems and by configuring and integrating off-the-shelf software or system components.

some fundamental activities of software processes are:

1. *Software specification* The functionality of the software and constraints on its operation must be defined.

2. *Software design and implementation* The software to meet the specification must be produced.

3. *Software validation* The software must be validated to ensure that it does what the customer wants.

4. ***Software evolution*** The software must evolve to meet changing customer needs.

4.1 Software process models

A software process model is an abstract representation of a software process. Each process model represents a process from a particular perspective, and thus provides only partial information about that process.

The process models are:

1. ***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.

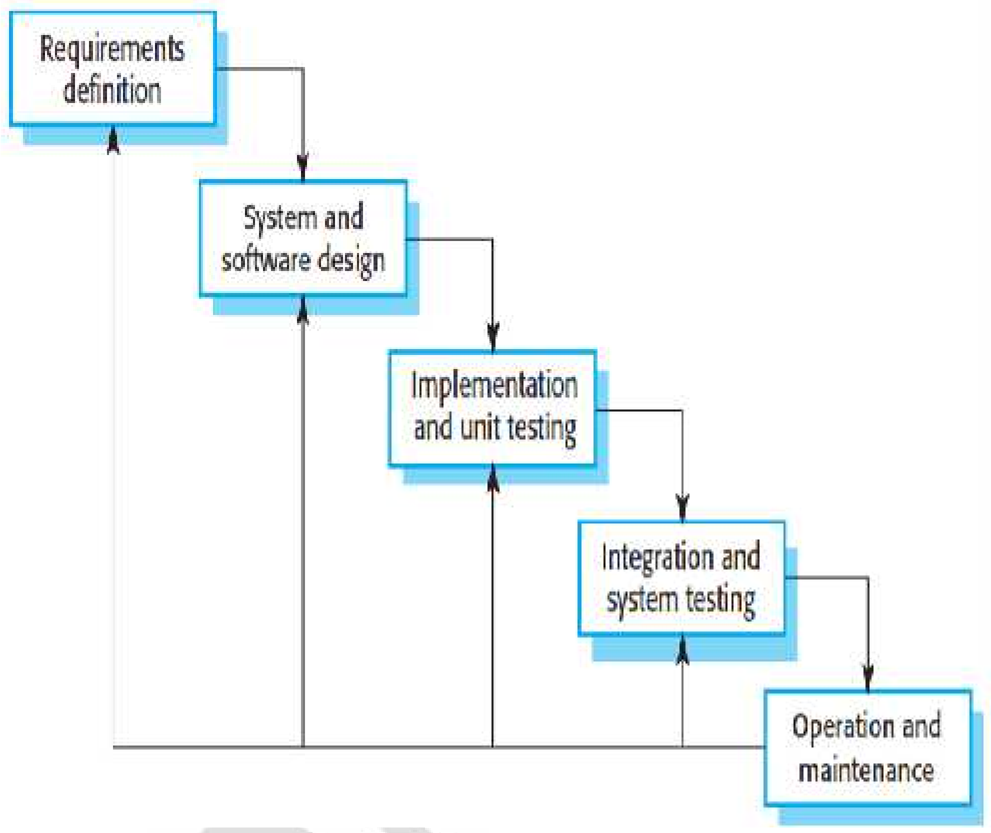
2. ***Evolutionary development*** This approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from abstract specifications. This is then refined with customer input to produce a System that satisfies the customer's needs.

3. 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.

4.1.1 The waterfall model

The first published model of the software development process was derived from more general system engineering processes. Because of the cascade from one phase to another, this model is known as the waterfall model or software life cycle. This is illustrated in Figure 4.1.

Figure 4.1 The software life cycle


Requirements definition
System and software design
Implementation and unit testing
Integration and system testing
Operation and maintenance

The principal stages of the model map onto fundamental development activities:

1. ***Requirements analysis and definition*** The system's services, constraints and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.

2. ***System and software design*** The systems design process partitions the requirements to either hardware or software systems. It establishes an overall system architecture.

Software design involves identifying and describing the fundamental software system abstractions and their relationships.

3. ***Implementation and unit testing*** During this stage, the software design is realised as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

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

5. ***Operation and maintenance*** Normally (although not necessarily) this is the longest life-cycle phase. The system is installed and put into practical use.

Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

The result of each phase is one or more documents that are approved ('signed off'). The next phase should not start until the previous phase has finished.

Advantages

1. waterfall model will be used when the requirements are well understood and unlikely to change radically during system development.
2. It is a simplest model
3. Documentation done at each phase

Disadvantages

1.Reusability is not encouraged.

2.It is very difficult to make any changes after the process is completed.

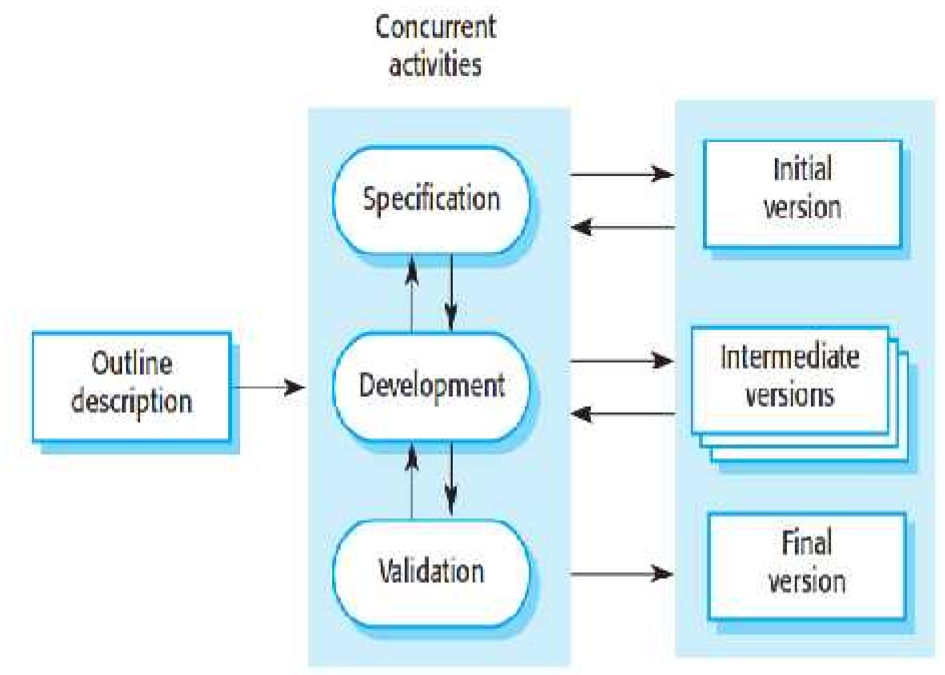
3.Inflexible portioning of project into distinct stages makes it difficult to response to changing customer requirements.

4.one phase has to complete before moving to next phase.

4.1.2 Evolutionary development

Evolutionary development is based on the idea of developing an initial implementation, exposing this to user comment and refining it through many versions until an adequate system has been developed (Figure 4.2).

Figure 4.2 Evolutionary development



There are two fundamental types of evolutionary development:

1. ***Exploratory development*** where the objective of the process is to work with the customer to explore their requirements and deliver a final system. The development starts with the parts of the system that are understood. The system evolves by adding new features proposed by the customer.

2. ***Throwaway prototyping*** where the objective of the evolutionary development process is to understand the customer's requirements and hence develop a better requirements definition for the system. The prototype concentrates on experimenting with the customer requirements that are poorly understood.

Advantages

1. The advantage of a software process that is based on an evolutionary approach is that the specification can be developed incrementally.
2.In evolutionary approach users develop a better understanding of their problem, this can be reflected in the software system.
3.Suitable For small and medium-sized systems

Disadvantages

1. *The process is not visible* Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
2. *Systems are often poorly structured* Continual change tends to corrupt the software structure. Incorporating software changes becomes increasingly difficult and costly.
3.The problems of evolutionary development become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system. It is difficult to establish a stable system architecture using this approach, which makes it hard to integrate contributions from the teams.

For large systems, mixed process that incorporates the best features of the waterfall and the evolutionary development models are recommended

Applicability of Evolutionary development approach

1.for short –lifetime systems
2.for small-medium sized interactive systems
3.for parts of large systems

2.3 Component-based software engineering

1. It is based on systematic reuse where systems are integrated from existing components or COTS(commercial off the shelf)systems
2. In the majority of software projects, there is some software reuse. This usually happens informally when people working on the project know of designs or code which is similar to that required. They look for these, modify them as needed and incorporate them into their system.

3 This reuse-oriented approach relies on a large base of reusable software components and some integrating framework for these components.

4. The generic process model for CBSE is shown in Figure 4.3.

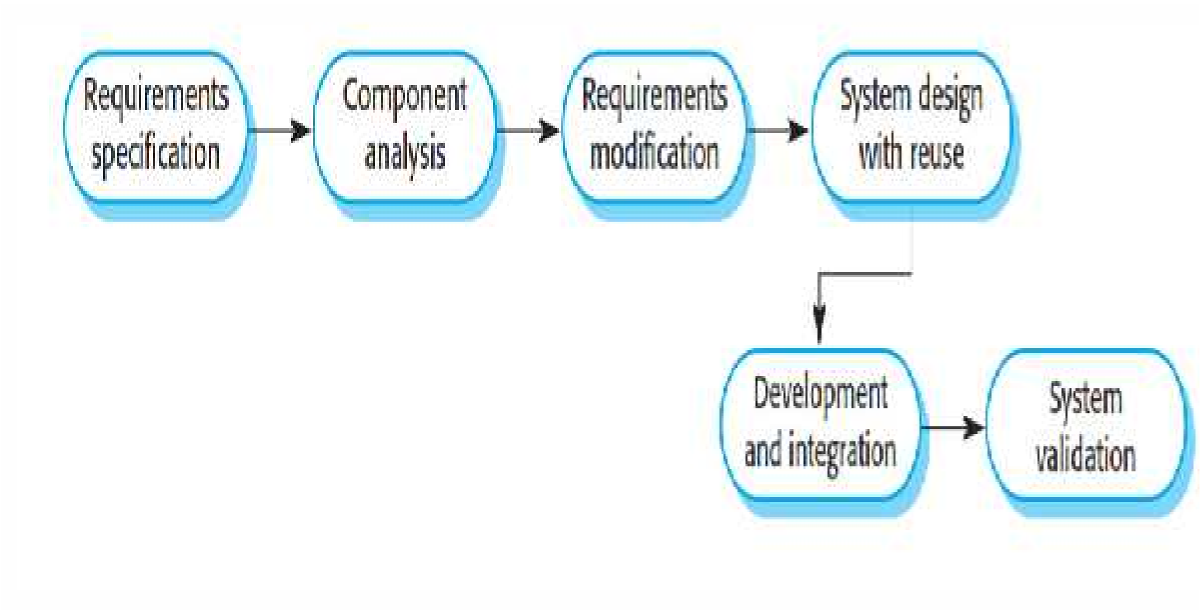

Requirements specification
Component analysis
Requirements modification
System design with reuse
Development and integration
System validation

Figure 4.3
Component-based software engineering

5. The initial requirements specification stage and the validation stage are comparable with other processes, the intermediate stages in a reuse-oriented process are different.

These stages are:

Component analysis Given the requirements specification, a search is made for components to implement that specification. Usually, there is no exact match, and the components that may be used only provide some of the functionality required.

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. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.

System design with reuse During this phase, the framework of the system is designed or an existing framework is reused. The designers take into account the components that are reused and organise the framework to cater to this. 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.

Advantages

1.reduces the amount of software to be developed

2.reduces cost and risks

3.leads to faster delivery of the software. However, requirements compromises

Disadvantages

1.As requirements are compromised, the evolved system does not meet the real need of users.

2.This also may affect control over system evolution /control over them is lost.

4.2 Process iteration

The software process is not a one-off process; rather, the process activities are regularly repeated as the system is reworked in response to change requests, As new technologies become available.

Two process models that have been explicitly designed to support process iteration:

1. ***Incremental delivery*** The software specification, design and implementation are broken down into a series of increments that are each developed in turn.

2. ***Spiral development*** The development of the system spirals outwards from an initial outline through to the final developed system.

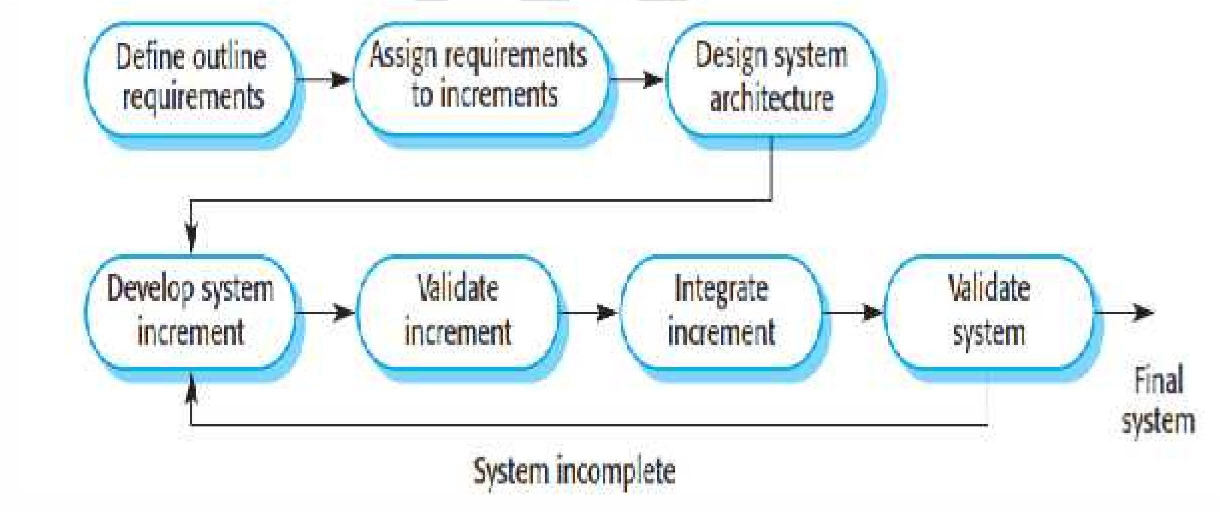


Figure 4.4
Incremental delivery

4.2.1 Incremental delivery

1. In an incremental development process (Figure 4.4), customers identify, in outline, the services to be provided by the system.

2. They identify which of the services are most important and which are least important to them.

3. A number of delivery increments are then defined, with each increment providing a sub-set of the system functionality.

4. Once the system increments have been identified, the requirements for the services to be delivered in the first increment are defined in detail, and that increment is developed.

5. During development, further requirements analysis for later increments Can take place, but requirements changes for the current increment are not accepted.

6. Once an increment is completed and delivered, customers can put it into service.

7. This means that they take early delivery of part of the system functionality. They can experiment with the system that helps them clarify their requirements for later increments and for later versions of the current increment.

8. As new increments are completed, they are integrated with existing increments so that the system functionality improves with each delivered increment.

9. This incremental development process has a number of advantages:

a). 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.

b). Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments.

c). There is a lower risk of overall project failure. Although problems may be encountered in some increments, it is likely that some will be successfully delivered to the customer.

d). 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. This means that customers are less likely to encounter software failures in the most important parts of the system.

10. However, there are problems with incremental delivery.

a). Increments should be relatively small (no more than 20,000 lines of code), and each increment should deliver some system functionality. It can be difficult to map the customer’s requirements onto increments of the right size.

b). Most systems require a set of basic facilities that are used by different parts of the system. As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.

4.2.2 Spiral development

The spiral model of the software process (Figure 4.5) was originally proposed by Boehm (Boehm, 1988).

Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design and so on.

Each loop in the spiral is split into four sectors:

1. ***Objective setting*** Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.

2. *Risk assessment and reduction* For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk.

Example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

3.Development and validation After risk evaluation, a development model for the system is chosen.

Example, if user interface risks are dominant, an appropriate development model might be evolutionary prototyping.

4. ***Planning*** The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

The main difference between the spiral model and other software process models is the explicit recognition of risk in the spiral model.

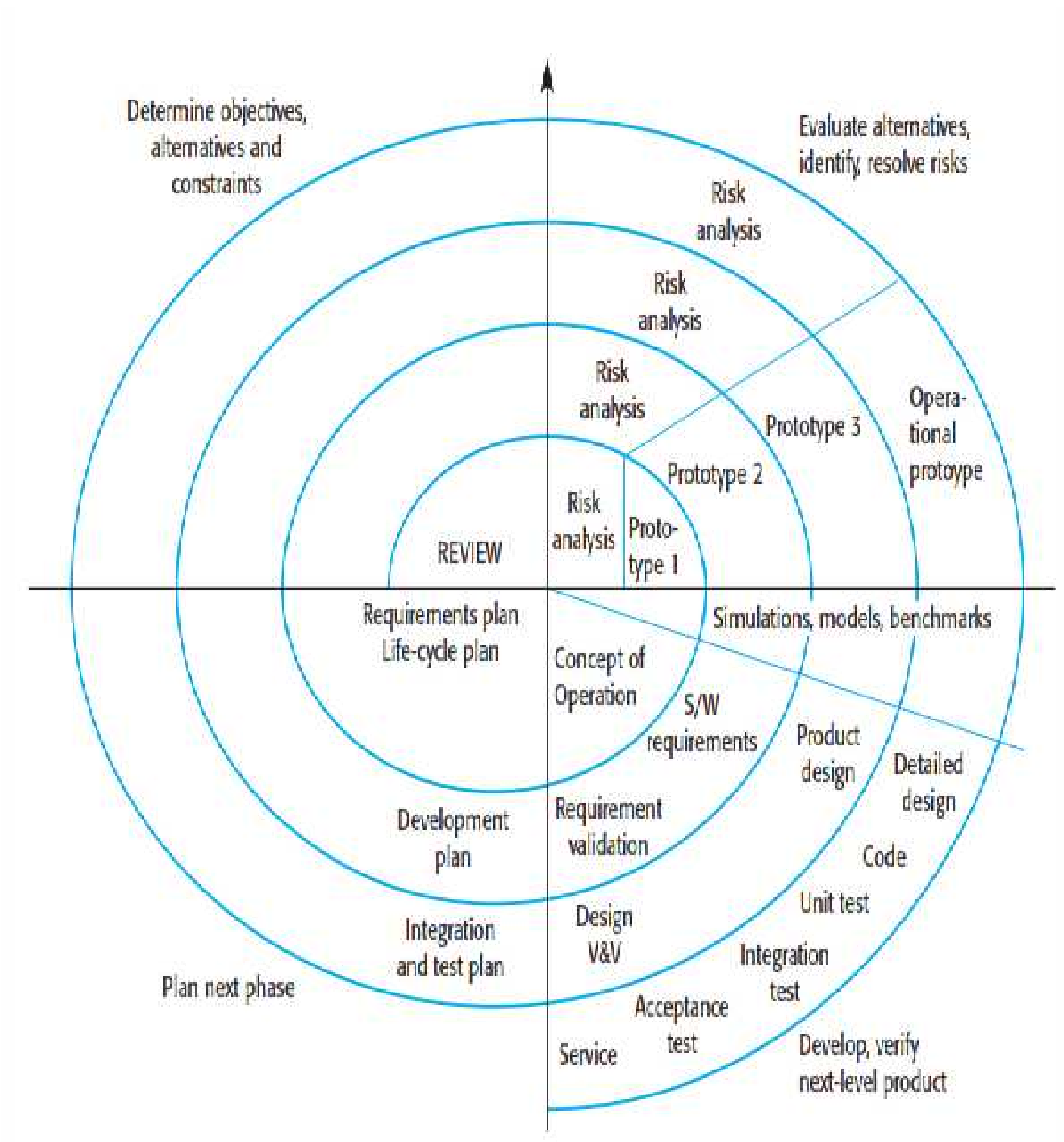


Figure 4.5 Boehm's spiral model of the software process (©IEEE, 1988)

4.3 Process activities

The four basic process activities are

1.Software specification
2.software development and implementation
3.Software validation
4.Software evolution

4.3.1 Software specification

1. Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development.

2. Requirements engineering is a particularly critical stage of the software process as errors at this stage inevitably lead to later problems in the system design and implementation.

3. This process leads to the production of a requirements document that is the specification for the system. Requirements are usually presented at two levels of detail in this document. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.

4. The requirements engineering process is shown in Figure 4.6.

5. There are four main phases in the requirements engineering process:

a). *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.

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

c).Requirements specification The activity of translating the information gathered during the analysis activity into a document that defines a set of requirements.

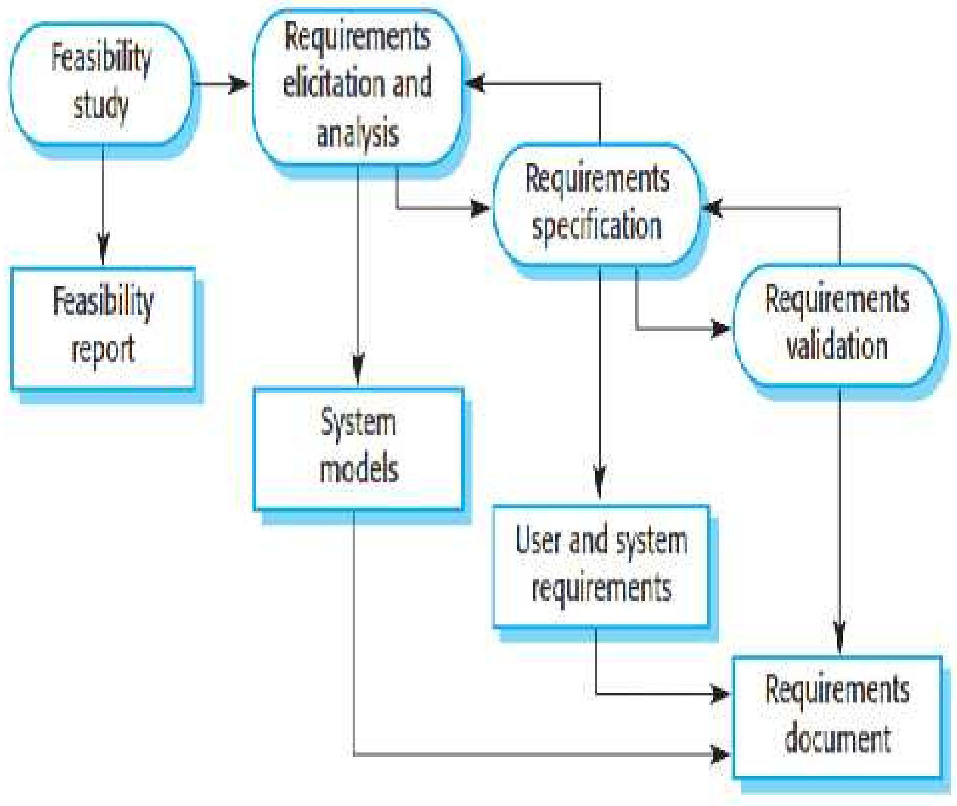
Two types of requirements may be included in this document.

User requir*ements* are abstract statements of the system requirements for the customer and end-user of the system

system requirements are a more detailed description of the functionality to be provided.

d)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.

Figure 4.6 The requirements engineering process



4.3.2 Software design and implementation

1. The implementation stage of software development is the process of converting a system specification into an executable system.

2. A software design is a description of the structure of the software to be implemented, the data which is part of the system, the interfaces between system components and, sometimes, the algorithms used.

3. The design process involves adding formality and detail as the design is developed with constant backtracking to correct earlier designs.

4. Figure 4.7 is a model of this design process are sequential process showing the design descriptions.

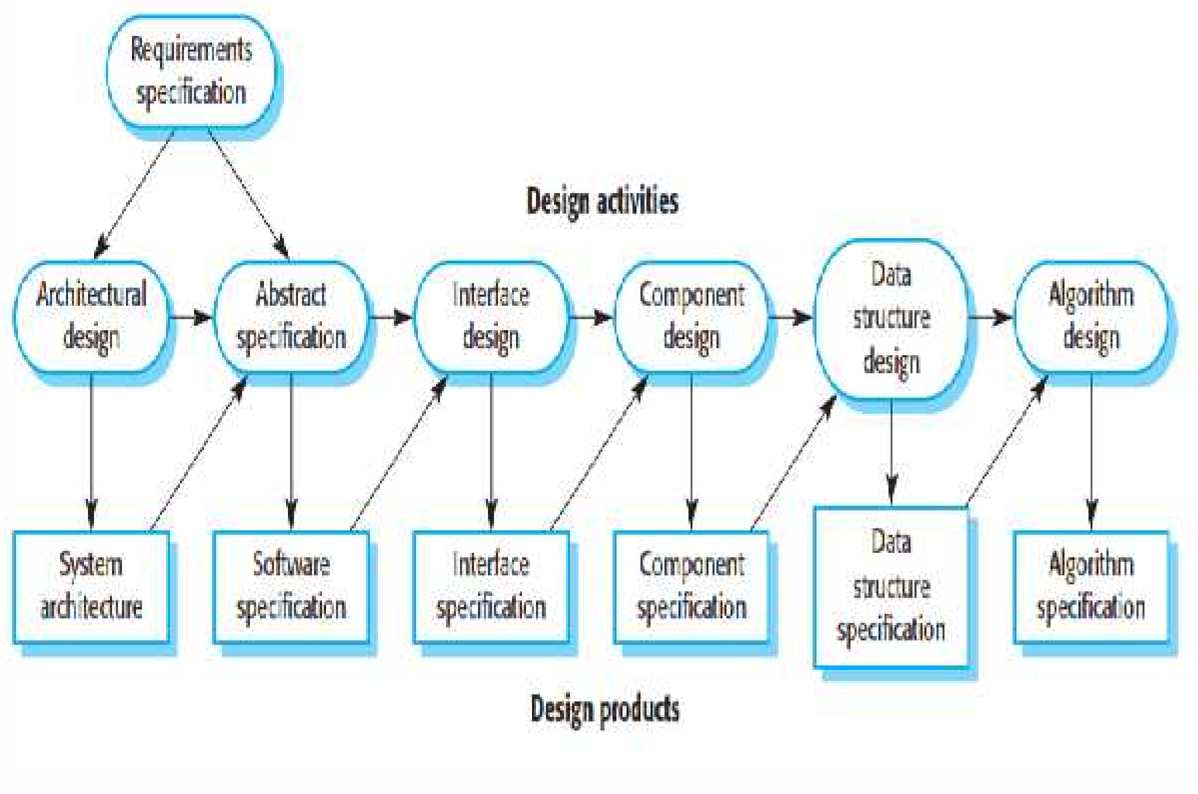


Figure 4.7 A general model of the design process

5. The specific design process activities are:

a). *Architectural design* The sub-systems making up the system and their relationships are identified and documented.

b). *Abstract specification* For each sub-system, an abstract specification of its services and the constraints under which it must operate is produced.

c). *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.

d). *Component design* Services are allocated to components and the interfaces of these components are designed.

e). *Data structure design* The data structures used in the system implementation are designed in detail and specified.

f). *Algorithm design* The algorithms used to provide services are designed in detail and specified.

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

a). The last two stages of design—data structure and algorithm design—may be delayed until the implementation process.

b). If an exploratory approach to design is used, the system interfaces may be designed after the data structures have been specified.

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

Structured Methods

structured methods for design rely on producing graphical models of the system, automatically generating code from these models.

A structured method includes a design process model, notations to represent the design, report formats, rules and design guidelines.

Structured methods may support some or all of the following models of a system:

1. An object model that shows the object classes used in the system and their dependencies.

2. A sequence model that shows how objects in the system interact when the system is executing.

3. A state transition model that shows system states and the triggers for the transitions from one state to another.

4. A structural model where the system components and their aggregations are documented.

5. A data flow model where the system is modelled using the data transformations that take place as it is processed. This is not normally used in object-oriented Methods but is still frequently used in real-time and business system design.

Programming and Debugging

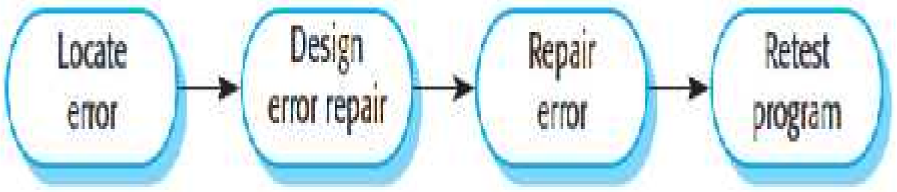
1-After design process programming is done for implementing the design ideas

2-Programming is a personal activity and there is no general process that is usually followed.

3-After programming programmers carry out some testing of the code they have developed. This often reveals program defects that must be removed from the program. This is called *debugging*

4.Figure 4.8 illustrates the stages of debugging. Defects in the code must be located
and the program modified to meet its requirements. Testing must then be repeated to ensure that the change has been made correctly. Thus the debugging process is part of both software development and software testing.

Figure 4.8 The debugging process



4.3.3 Software validation

1. Software validation or, more generally, verification and validation (V & V) is intended to show that a system conforms to its specification and that the system meets the expectations of the customer buying the system.

2. Figure 4.9 shows a three-stage testing process where system components are tested, the integrated system is tested and, finally, the system is tested with the customer's data.

3. The stages in the testing process are:

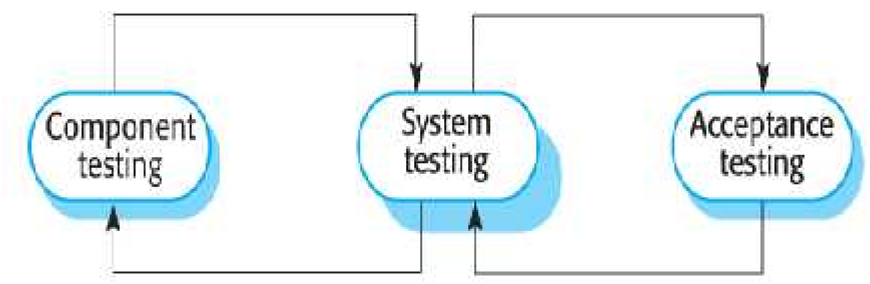
a) **Component (or unit) testing** Individual components are tested to ensure that they operate correctly. Each component is tested independently, without other system components. Components may be simple entities such as functions or object classes, or may be coherent groupings of these entities.

b) ***System testing*** The components are integrated to make up the system. This process is concerned with finding errors that result from unanticipated interactions

4. 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.

a)Acceptance testing/Alpha testing This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system customer rather than with simulated test data. Acceptance testing may Reveal errors and omissions in the system requirements definition. 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.

Figure 4.9 The testing process



5. Programmers make up their own test data and incrementally test the code as it is developed.

6-If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment.

7-In extreme programming, tests are developed along with the requirements before development starts.

8. This helps the testers and developers to understand the requirements and ensures that there are no delays as test cases are created.

9-Later stages of testing involve integrating work from a number of programmers and must be planned in advance. An independent team of testers should work from preformulated test plans that are developed from the system specification and design.

10. Figure 4.10 illustrates how test plans are the link between testing and development
activities.

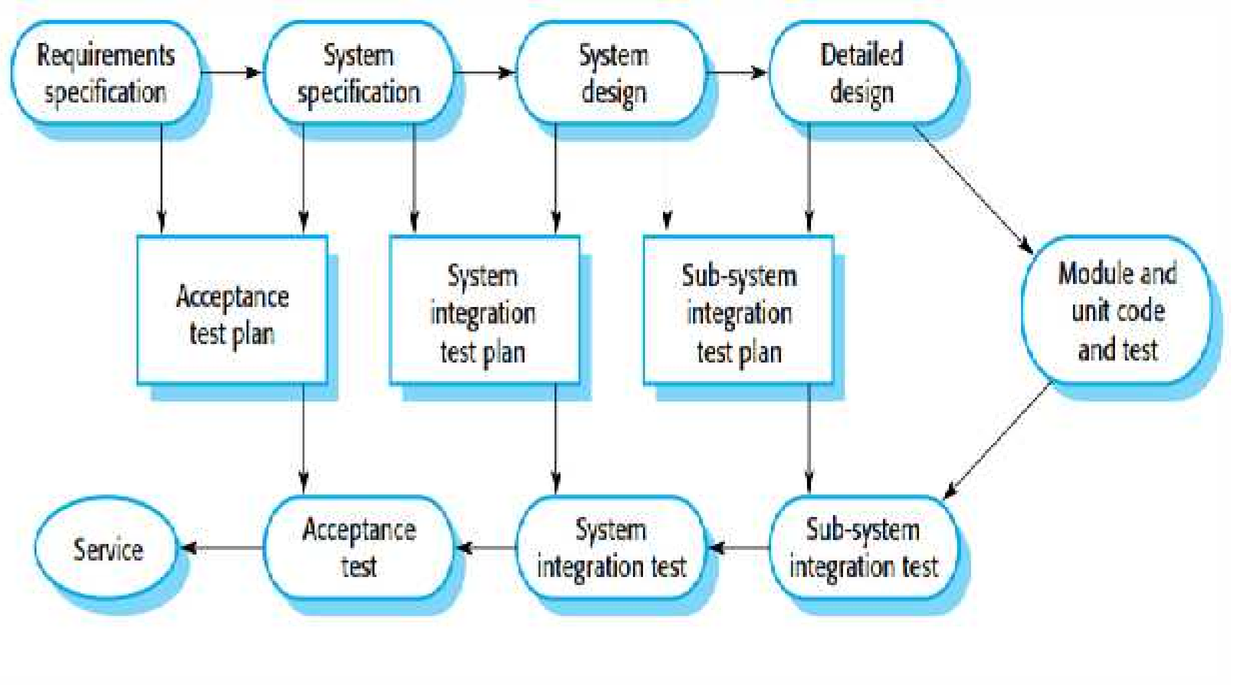

Requirements specification
System specification
System design
Detailed design
Acceptance test plan
System integration test plan
Sub-system integration test plan
Module and unit code and test
Service
Acceptance test
System integration test
Sub-system integration test

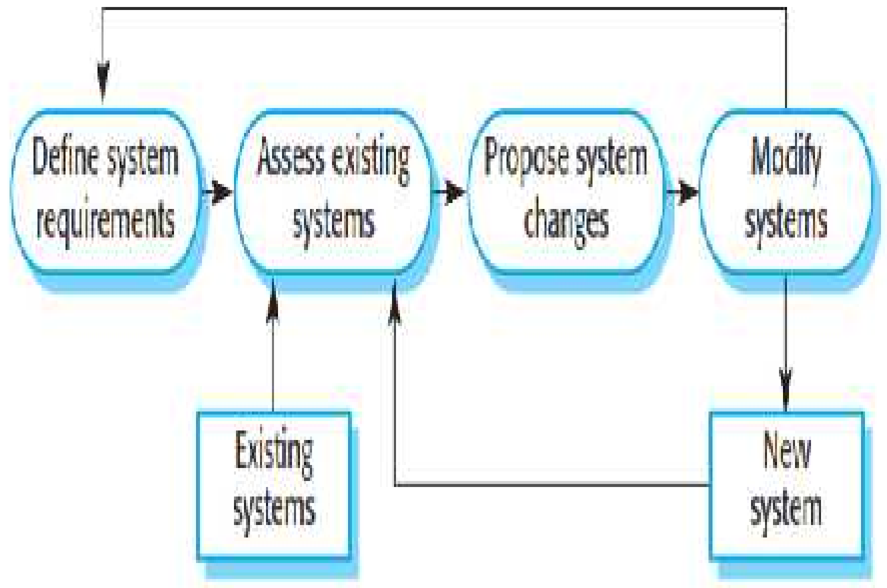
Figure 4.10 Testing phases in the software process

4.3.4 Software evolution

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

-Thus requirement change through changing business circumstances the software that supports the business must also evolve and change.

Figure 4.11 System evolution



4.4 The Rational Unified Process

The Rational Unified Process (RUP) is an example of a modern process model that has been derived from work on the UML and the associated Unified Software Development Process.

The RUP is normally described from three perspectives:

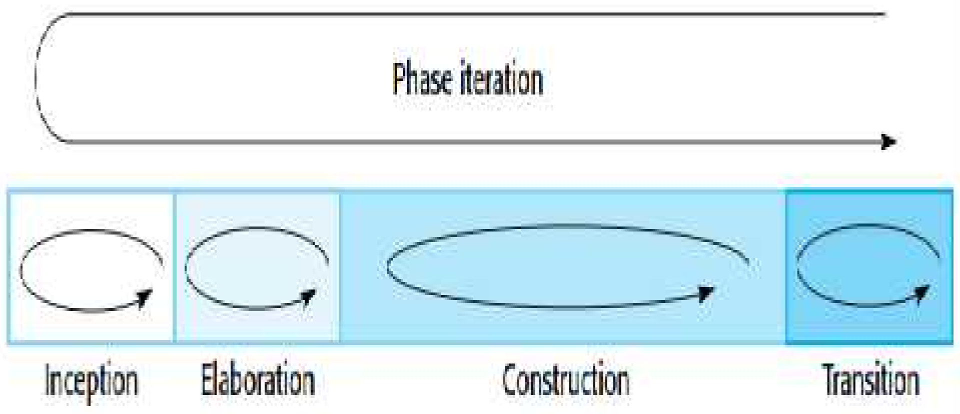
1. A dynamic perspective that shows the phases of the model over time.
2. A static perspective that shows the process activities that are enacted.

3. A practice perspective that suggests good practices to be used during the process.

The RUP is a phased model that identifies four discrete phases in the software process. The phases in the RUP are more closely related to business rather
Than technical concerns

Figure 4.12 shows the phases in the RUP.

Figure 4.12 Phases in the Rational Unified Process



These are:

1.Inception The goal of the inception phase is to establish a business case for the system. Identify all external entities (people and systems) that will interact with the system and define these interactions. Then use this information to assess the contribution that the system makes to the business. If this contribution is minor, then the project may be cancelled after this phase.

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.

On completion of this phase, requirements model for the system (UML use cases are specified), an architectural description and a development plan for the software will be ready.

3.Construction The construction phase is essentially concerned with system design, programming and testing. Parts of the system are developed in parallel and integrated during this phase.

On completion of this phase, you should have a working software system and associated documentation that is ready for delivery to users.

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.

On completion of this phase, you should have a documented software system That is working correctly in its operational environment.

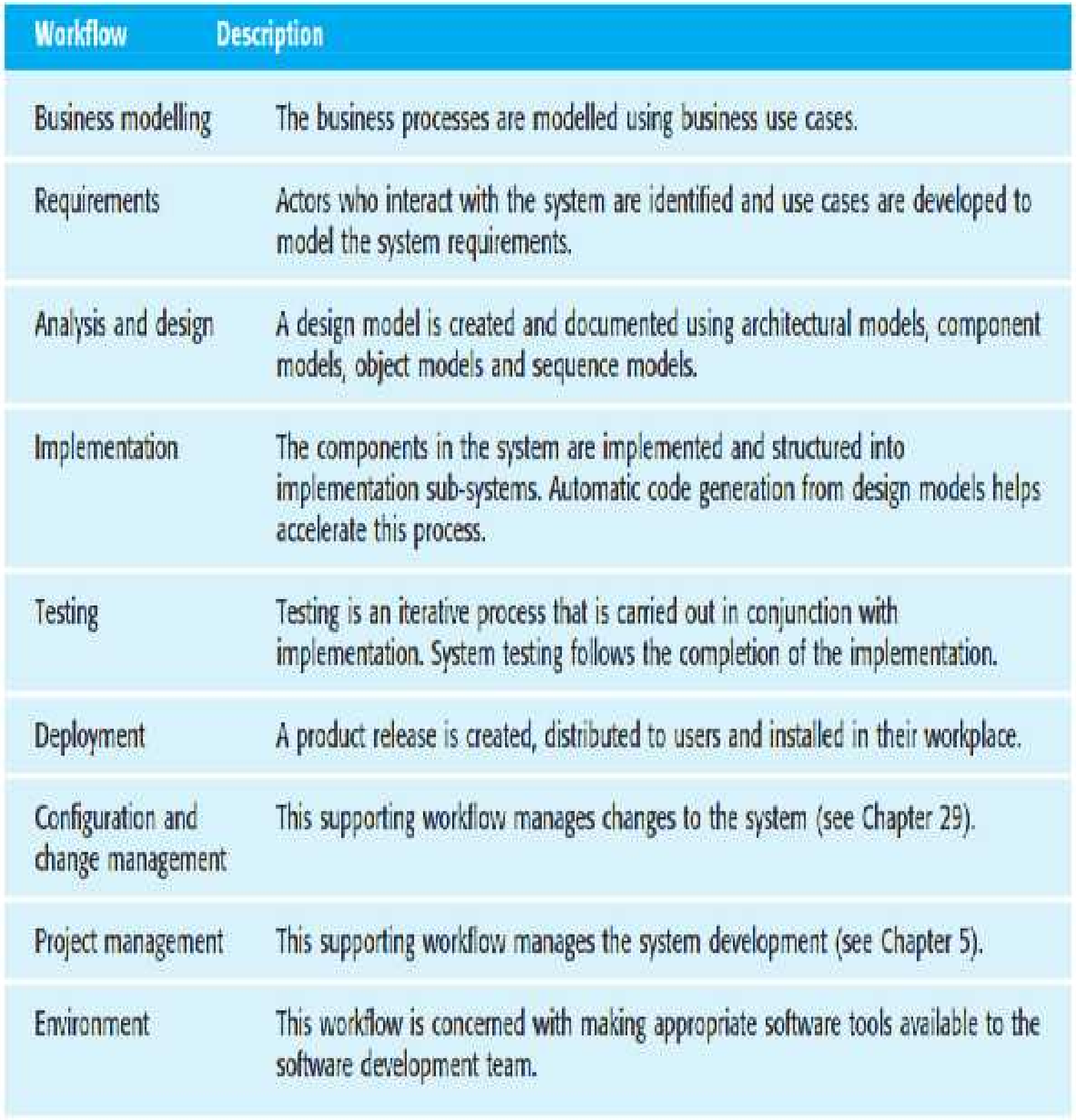
Iteration within the RUP is supported in two ways, as shown in Figure 4.12. Each phase may be enacted in an iterative way with the results developed incrementally.

In addition, the whole set of phases may also be enacted incrementally, as shown by the looping arrow from Transition to Inception in Figure 4.12.

Workflows in RUP

1. 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.

2. The core engineering and support workflows are described in Figure 4.13.



Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users and installed in their workplace.
Configuration and change management	This supporting workflow manages changes to the system (see Chapter 29).
Project management	This supporting workflow manages the system development (see Chapter 5).
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

Figure 4.13 Static workflows in Rational Unified Process

Engineering practices

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. Analyse the impact of changes on the system before accepting them.

3. *Use component-based architectures*. Structure the system architecture into components as discussed earlier in this chapter.

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 .

4.5 Computer-Aided Software Engineering

1. Computer-Aided Software Engineering (CASE) is the name given to software used to support software process activities such as requirements engineering, design, program development and testing.

2 .CASE tools therefore include design editors, data dictionaries, compilers, debuggers, system building tools and so on.

3. CASE technology provides software process support by automating some process activities and by providing information about the software that is being developed.

Examples of activities that can be automated using CASE include:

a. The development of graphical system models as part of the requirements specification or the software design.

b. Understanding a design using a data dictionary that holds information about the entities and relations in a design.

c. The generation of user interfaces from a graphical interface description that is created interactively by the user.

d. Program debugging through the provision of information about an executing program.

e. The automated translation of programs from an old version of a programming language such as COBOL to a more recent version.

4. CASE technology is now available for most routine activities in the software process. This has led to some improvements in software quality and productivity, although these have been less than predicted.

5. The improvements from the use of CASE are limited by two factors/Disadvantages of CASE:

a). Software engineering is, essentially, a design activity based on creative thought. Existing CASE systems automate routine activities but attempts to harness artificial intelligence technology to provide support for design have not been successful.

b). In most organisations, software engineering is a team activity, and software engineers spend quite a lot of time interacting with other team members. CASE technology does not provide much support for this.

4.5.1 CASE classification

CASE classifications help us understand the types of CASE tools and their role in supporting software process activities.

CASE tools are classified from three of these perspectives:

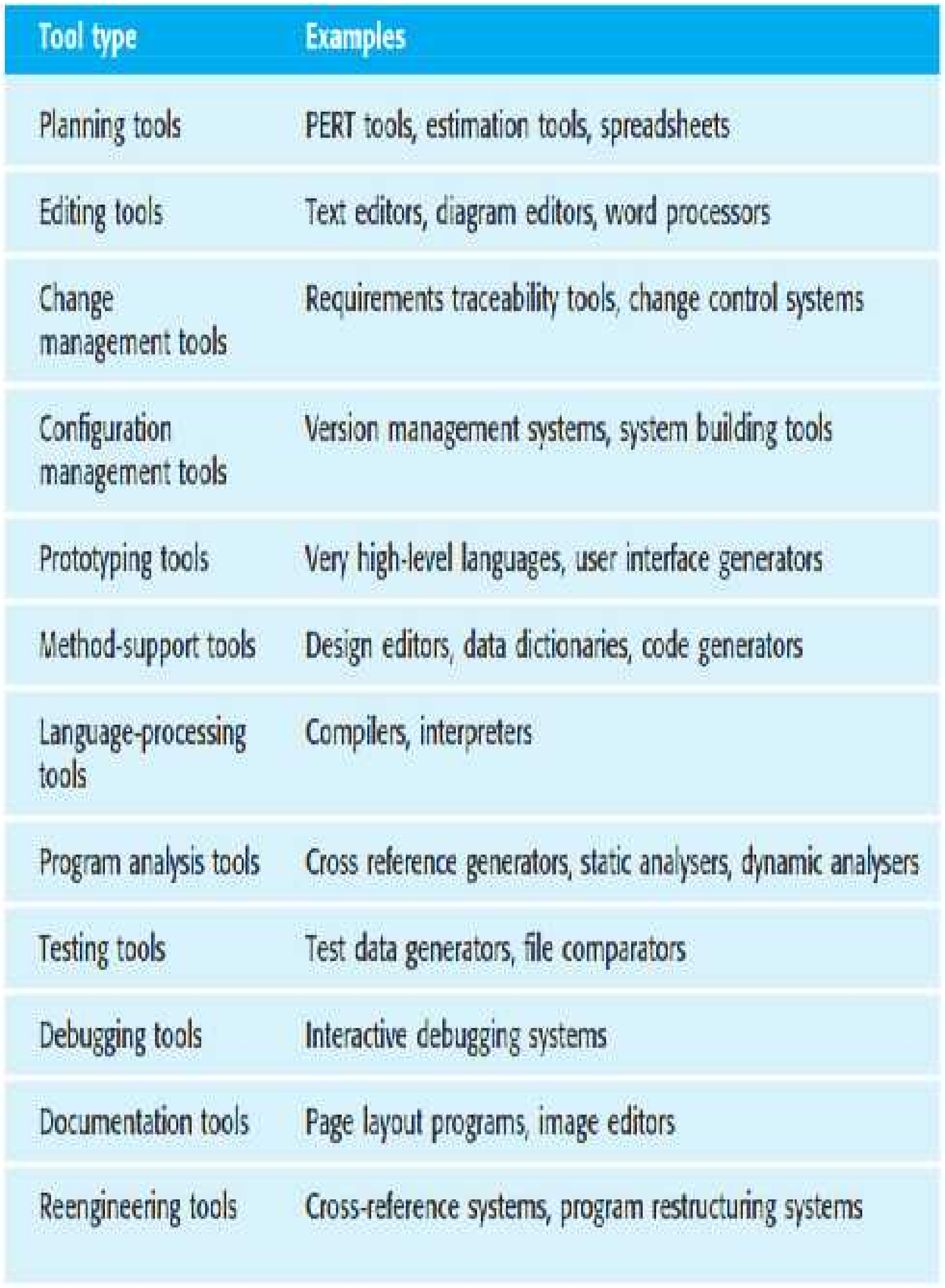
1. ***A functional perspective*** where CASE tools are classified according to their specific function.

2. ***A process perspective*** where tools are classified according to the process activities that they support.

3. ***An integration perspective*** where CASE tools are classified according to how they are organised into integrated units that provide support for one or more process activities.

Figure 4.14 is a classification of CASE tools according to function. This table lists a number of different types of CASE tools and gives specific examples of each

Figure 4.14 Functional classification of CASE tools



Tool type	Examples
Planning tools	PERT tools, estimation tools, spreadsheets
Editing tools	Text editors, diagram editors, word processors
Change management tools	Requirements traceability tools, change control systems
Configuration management tools	Version management systems, system building tools
Prototyping tools	Very high-level languages, user interface generators
Method-support tools	Design editors, data dictionaries, code generators
Language-processing tools	Compilers, interpreters
Program analysis tools	Cross reference generators, static analysers, dynamic analysers
Testing tools	Test data generators, file comparators
Debugging tools	Interactive debugging systems
Documentation tools	Page layout programs, image editors
Reengineering tools	Cross-reference systems, program restructuring systems

Figure 4.15 presents an alternative classification of CASE tools. It shows the process phases supported by a number of types of CASE tools. Tools for planning And estimating, text editing, document preparation and configuration management May be used throughout the software process.

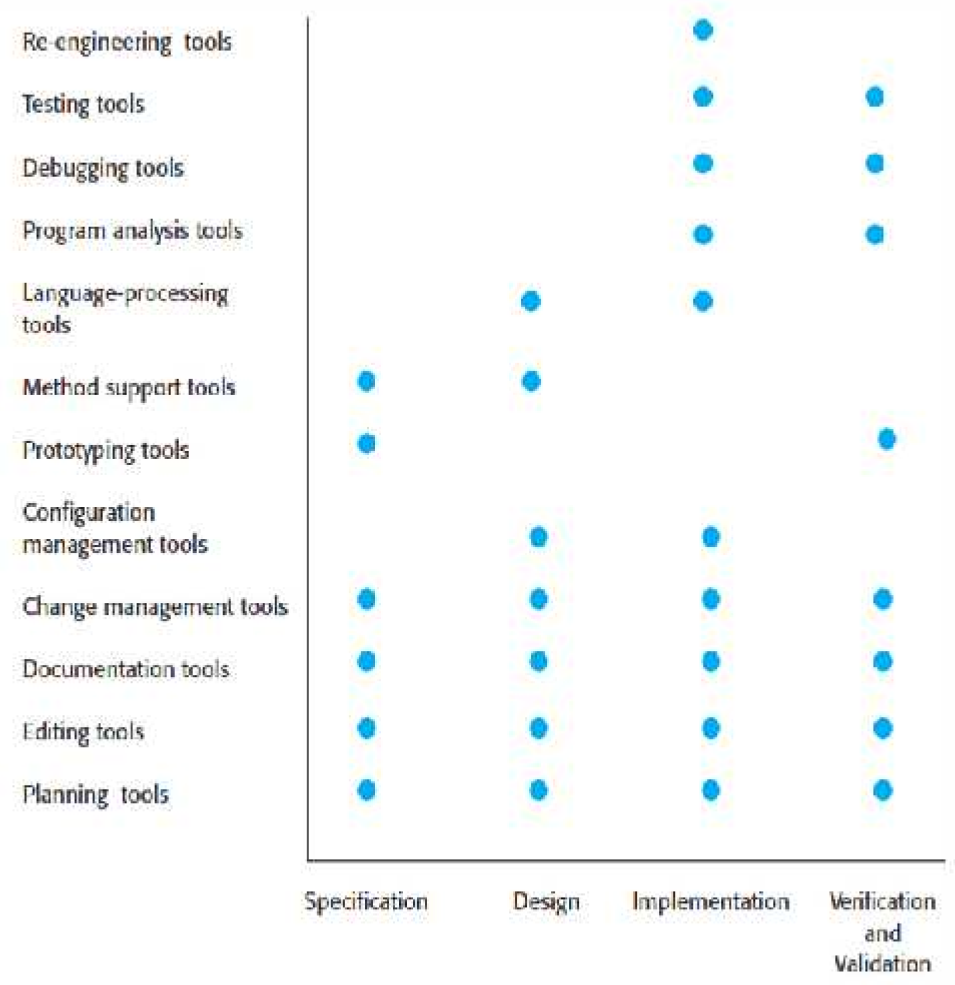
The breadth of support for the software process offered by CASE technology is another possible classification dimension. Fuggetta (Fuggetta, 1993) proposes that CASE systems should be classified in three categories:

1. *Tools* support individual process tasks such as checking the consistency of a design, compiling a program and comparing test results. Tools may be general-purpose, standalone tools (e.g., a word processor) or grouped into workbenches.

2. *Workbenches* support process phases or activities such as specification, design, etc. They normally consist of a set of tools with some greater or lesser degree of integration.

3. *Environments* support all or at least a substantial part of the software process. They normally include several integrated workbenches.

Figure 4.15 Activity-based classification of CASE tools



	Specification	Design	Implementation	Verification and Validation
Re-engineering tools			●	
Testing tools			●	●
Debugging tools			●	●
Program analysis tools			●	●
Language-processing tools		●	●	
Method support tools	●	●		
Prototyping tools	●			●
Configuration management tools		●	●	
Change management tools	●	●	●	●
Documentation tools	●	●	●	●
Editing tools	●	●	●	●
Planning tools	●	●	●	●

Figure 4.16 illustrates this classification and shows some examples of these classes of CASE support.

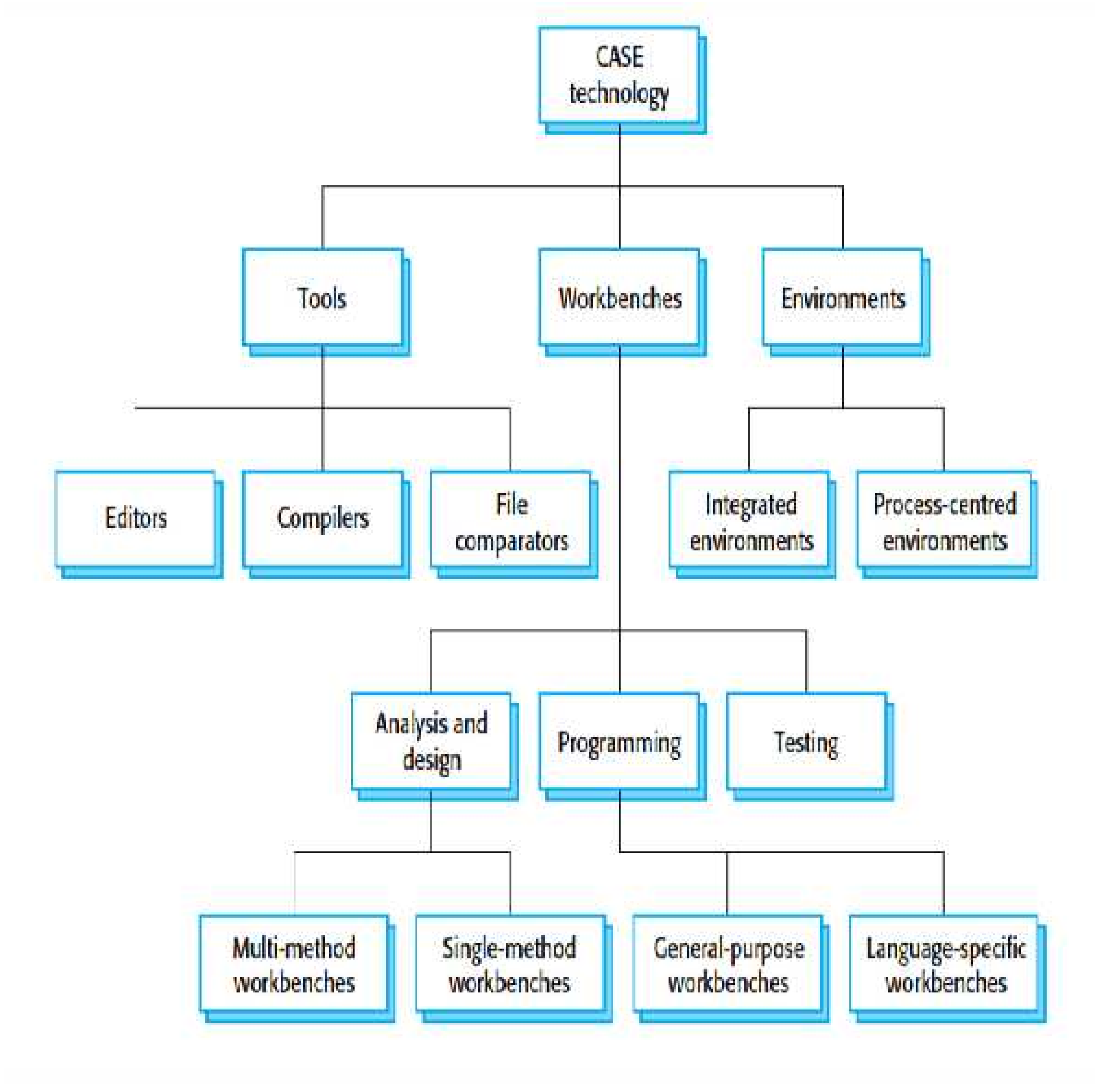


Figure 4.16 Tools, workbenches and environments