UNIT - 1


Ans : Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.

Software products may be
• Generic - developed to be sold to a range of different customers
• Bespoke (custom) - developed for a single customer according to their specification

➢ Software engineering is an engineering discipline which is concerned with all aspects of software production.
➢ Software engineers should adopt a systematic and organized approach to their work and use appropriate tools and techniques depending on the problem to be solved, the development constraints and the resources available.

Attributes of a good software :

Software is engineered and not manufactured.
Software does not ware out
Most softwares are custom built rather than being assembled from components

<table>
<thead>
<tr>
<th>Figure 1.5 Essential attributes of good software</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Product characteristic</strong></td>
</tr>
<tr>
<td>---------------------------</td>
</tr>
<tr>
<td>Maintainability</td>
</tr>
<tr>
<td>Dependability</td>
</tr>
<tr>
<td>Efficiency</td>
</tr>
<tr>
<td>Usability</td>
</tr>
</tbody>
</table>
Software engineering in the 21st century faces three key challenges:

1. **The heterogeneity challenge** Increasingly, systems are required to operate as distributed systems across networks that include different types of computers and with different kinds of support systems. It is often necessary to integrate new software with older legacy systems written in different programming languages. The heterogeneity challenge is the challenge of developing techniques for building dependable software that is flexible enough to cope with this heterogeneity.

2. **The delivery challenge** Many traditional software engineering techniques are time-consuming. The time they take is required to achieve software quality. However, businesses today must be responsive and change very rapidly. Their supporting software must change equally rapidly. The delivery challenge is the challenge of shortening delivery times for large and complex systems without compromising system quality.

3. **The trust challenge** As software is intertwined with all aspects of our lives, it is essential that we can trust that software. This is especially true for remote software systems accessed through a web page or web service interface. The trust challenge is to develop techniques that demonstrate that software can be trusted by its users.

2. Describe four Ethics & Professional responsibilities of a software engineer. 

   It goes without saying that you should always uphold normal standards of honesty and integrity. You should not use your skills and abilities to behave in a dishonest way or in a way that will bring disrepute to the software engineering profession. However, there are areas where standards of acceptable behaviour are not bounded by laws but by the more tenuous notion of professional responsibility. Some of these are:

   1. **Confidentiality** You should normally respect the confidentiality of your employers or clients irrespective of whether a formal confidentiality agreement has been signed.

   2. **Competence** You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.

   3. **Intellectual property rights** You should be aware of local laws governing the use of intellectual property such as patents and copyright. You should be careful to ensure that the intellectual property of employers and clients is protected.

   4. **Computer misuse** You should not use your technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).
3. Define socio technical systems and explain emergent system properties with example. [June 2010] [5M]

Systems that include software fall into two categories:

- **Technical computer-based systems** are systems that include hardware and software components but not procedures and processes. Examples of technical systems include televisions, mobile phones and most personal computer software. Individuals and organisations use technical systems for some purpose but knowledge of this purpose is not part of the system. For example, the word processor I am using is not aware that is it being used to write a book.

- **Socio-technical systems** include one or more technical systems but, crucially, also include knowledge of how the system should be used to achieve some broader objective. This means that these systems have defined operational processes, include people (the operators) as inherent parts of the system, are governed by organisational policies and rules and may be affected by external constraints such as national laws and regulatory policies. For example, this book was created through a socio-technical publishing system that includes various processes and technical systems.

The entire properties of a system are called Emergent properties.

There are two types of emergent properties:

1. **Functional emergent properties** 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 from its components.

2. **Non-functional emergent properties** relate to the behaviour of the system in its operational environment. Examples of non-functional properties are reliability, performance, safety and security. These are often critical for computer-based systems, as failure to achieve some minimal defined level in these properties may make the system unusable. Some users may not need some system functions so the system may be acceptable without them. However, a system that is unreliable or too slow is likely to be rejected by all its users.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume</td>
<td>The volume of a system (the total space occupied) varies depending on how the component assemblies are arranged and connected.</td>
</tr>
<tr>
<td>Reliability</td>
<td>System reliability depends on component reliability but unexpected interactions can cause new types of failure and therefore affect the reliability of the system.</td>
</tr>
<tr>
<td>Security</td>
<td>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 safeguard.</td>
</tr>
<tr>
<td>Repairability</td>
<td>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.</td>
</tr>
<tr>
<td>Usability</td>
<td>This property reflects how easy it is to use the system. It depends on the technical system components, its operators and its operating environment.</td>
</tr>
</tbody>
</table>

4. Describe briefly the phases of System engineering process with neat diagram.

[Dec 2010 Dec 2011, Dec 2012] [10M] [5M]
System requirements definition:

This requirements definition phase usually concentrates on deriving three types of requirement:
1. Abstract functional requirements: The basic functions that the system must provide are defined at an abstract level. More detailed functional requirements specification takes place at the sub-system level. For example, in an air traffic control system, an abstract functional requirement would specify that a flight-plan database should be used to store the flight plans of all aircraft entering the controlled airspace. However, you would not normally specify the details of the database unless they affected the requirements of other sub-systems.
2. System properties: These are non-functional emergent system properties such as availability, performance and safety, as I have discussed above. These nonfunctional system properties affect the requirements for all sub-systems.
3. Characteristics that the system must not exhibit It is sometimes as important to specify what the system must not do as it is to specify what the system should do.

For example, if you are specifying an air traffic control system, you might specify that the system should not present the controller with too much information.

System Design:
The activities involved in this process are:
1. Partition requirements You analyse the requirements and organise them into related groups. There are usually several possible partitioning options, and you may suggest a number of alternatives at this stage of the process.
2. Identify sub-systems You should identify sub-systems that can individually or collectively meet the requirements. Groups of requirements are usually related to sub-systems, so this activity and requirements partitioning may be amalgamated.
However, sub-system identification may also be influenced by other organizational or environmental factors.
3. Assign requirements to sub-systems You assign the requirements to sub-systems.
In principle, this should be straightforward if the requirements partitioning is used to drive the sub-system identification. In practice, there is never a clean match between requirements partitions and identified sub-systems.
Limitations of externally purchased sub-systems may mean that you have to change the requirements to accommodate these constraints.
4. Specify sub-system functionality You should specify the specific functions provided by each sub-system. This may be seen as part of the system design phase.
5. Define sub-system interfaces You define the interfaces that are provided and required by each sub-system. Once these interfaces have been agreed upon, it becomes possible to develop these sub-systems in parallel.

System modeling:
During the system requirements and design activity, systems may be modelled as a set of components and relationships between these components. These are normally illustrated graphically in a system architecture model that gives the reader an overview of the system organisation.

The system architecture may be presented as a block diagram showing the major sub-systems and the interconnections between these sub-systems. When drawing a block diagram, you should represent each sub-system using a rectangle, and you should show relationships between the sub-systems using arrows that link these rectangles.

The relationships indicated may include data flow, a uses/' used by' relationship or some other type of dependency relationship.

Sub-system development:

During sub-system development, the sub-systems identified during system design are implemented. This may involve starting another system engineering process for individual sub-systems or, if the sub-system is software, a software process involving requirements, design, implementation and testing.

Systems integration:

During the systems integration process, you take the independently developed sub-systems and put them together to make up a complete system. Integration can be done using a 'big bang' approach, where all the sub-systems are integrated at the same time. However, for technical and managerial purposes, an incremental integration process where sub-systems are integrated one at a time is the best approach, for two reasons:

1. It is usually impossible to schedule the development of all the sub-systems so that they are all finished at the same time.
2. Incremental integration reduces the cost of error location. If many sub-systems are simultaneously integrated, an error that arises during testing may be in any of these sub-systems. When a single sub-system is integrated with an already working system, errors that occur are probably in the newly integrated sub-system or in the interactions between the existing sub-systems and the new sub-system.

System evolution:

Large, complex systems have a very long lifetime. During their life, they are changed to correct errors in the original system requirements and to implement new requirements that have emerged. The system's computers are likely to be replaced with new, faster machines. System evolution is inherently costly for several reasons:

1. Proposed changes have to be analyzed very carefully from a business and a technical perspective. Changes have to contribute to the goals of the system and should not simply be technically motivated.
2. Because sub-systems are never completely independent, changes to one subsystem may adversely affect the performance or behavior of other subsystems.
3. The reasons for original design decisions are often unrecorded. Those responsible for the system evolution have to work out why particular design decisions were made.
4. As systems age, their structure typically becomes corrupted by change so the costs of making further changes increases.
UNIT -2

1. **Explain the different dimensions of dependability with a neat diagram** [Dec 2011, Dec 2010, Jan 2009] [5M].
   There are four principal dimensions to dependability:

   ![Dependability Diagram]

   1. **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.
   2. **Reliability** Informally, the reliability of a system is the probability, over a given period of time, that the system will correctly deliver services as expected by the user.
   3. **Safety** Informally, the safety of a system is a judgment of how likely it is that the system will cause damage to people or its environment.
   4. **Security** Informally, the security of a system is a judgment of how likely it is that the system can resist accidental or deliberate intrusions.

2. **What is process iteration? Explain Boehm’s spiral model.** [Jan 2010, Jan 2009, Dec 2012] [10M]
   Iteration approach interleaves the activities of specification, development and validation. An initial system is rapidly developed from very abstract specifications. This is then refined with customer input to produce a system that satisfies the customer's needs. The system may then be delivered. Alternatively, it may be re-implemented using a more structured approach to produce a more robust and maintainable system.

   In a spiral model, 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.

   This model is developed by Barry Boehm. It is an evolutionary software process model.
The goal of this model is to provide a framework for designing such process, guided by the risk levels in the project at hand. The spiral model may be viewed as a meta-model.

This model focuses on identifying and eliminating high risk problems by careful process design.

3. What are the four basic process activities? Explain the general model of design process with a neat diagram. [Dec 11] [10M]

**Process activities:**

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.

The specific design process activities are:

1. *Architectural design* The sub-systems making up the system and their relationships are identified and documented.
2. *Abstract specification* For each sub-system, an abstract specification of its services and the constraints under which it must operate is produced.
3. **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.

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

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

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

4. **Explain phases in RUP. [June 2010] [SM]**

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.

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.

Figure shows the phases in the RUP.

These are:

1. **Inception** The goal of the inception phase is to establish a business case for the system.
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. Parts of the system are developed in parallel and integrated during this phase.

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.

5. **What is Critical systems? Explain Dimensions of dependability?**

   [Dec 2012][6M]

   For critical systems, it is usually the case that the most important system property is the dependability of the system.

   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.

**Dimensions of dependability**

![Diagram of dimensions of dependability]

- **Availability**: The ability of the system to deliver services when requested.
- **Reliability**: The ability of the system to deliver services as specified?
- **Safety**: The ability of the system to operate without catastrophic failure.
- **Security**: The ability of the system to protect itself against accidental or deliberate intrusion.
UNIT-3

1. **Describe functional and non-functional requirements with examples.**
   
   [Jan 10, Jan 09, June 09, June 10, Dec 11, Dec 2012] [10M]

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

1. **Functional requirements** These are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

2. **Non-functional requirements** These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. Non-functional requirements often apply to the system as a whole. They do not usually just apply to individual system features or services.

**Functional requirements:**

Example, here are examples of functional requirements for a university library system called LIBSYS, used by students and faculty to order books and documents from other libraries.

1. The user shall be able to search either all of the initial set of databases or select a subset from it.
2. The system shall provide appropriate viewers for the user to read documents in the document store.
3. 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.
Non-functional requirements:

Types of Non-functional requirements:

Non-functional requirements are not just concerned with the software system to be developed. Some non-functional requirements may constrain the process that should be used to develop the system. Examples of process requirements include a specification of the quality standards that should be used in the process, a specification that the design must be produced with a particular CASE toolset and a description of the process that should be followed.

2. What are the metrics used to specify non-functional system properties?
   [June 10, Dec 2012] [5 M]

<table>
<thead>
<tr>
<th>SI No</th>
<th>Property</th>
<th>Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Speed</td>
<td>Processed transaction/second User/Event response time Screen refresh time</td>
</tr>
<tr>
<td>2</td>
<td>Size</td>
<td>kilo bytes Number of Ram chips</td>
</tr>
<tr>
<td>3</td>
<td>Ease of Use</td>
<td>Training time Number of help frames</td>
</tr>
<tr>
<td>4</td>
<td>Reliability</td>
<td>Mean time to failure</td>
</tr>
</tbody>
</table>
2. Explain the IEEE standard format for requirement document in detail [June 10, Jan 09, Dec 11, Dec 2012] [6M]

IEEE standard suggests the following structure for requirements documents:

1. Introduction
   1.1 Purpose of the requirement:nts document
   1.2 Scope of the product
   1.3 Definitions, acronyms and abbreviations
   1.4 References
   1.5 Overview of the remainder of the document
2. General description
   2.1 Product perspective
   2.2 Product functions
   2.3 User characteristics
   2.4 General constraints
   2.5 Assumptions and dependencies
3. Specific requirements cover functional, non-functional and interface requirements.
   This is obviously the most substantial part of the document but because of the wide variability in organizational practice, it is not appropriate to define a standard structure for this section.
4. Appendices
5. Index

The structure of a requirements document:
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.</td>
</tr>
<tr>
<td>Introduction</td>
<td>This should describe the need for the system. It should briefly describe its functions and explain how it will work with other systems. It should describe how the system fits into the overall business or strategic objectives of the organisation commissioning the software.</td>
</tr>
<tr>
<td>Glossary</td>
<td>This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.</td>
</tr>
<tr>
<td>User requirements</td>
<td>The services provided for the user and the non-functional system requirements should be described in this section. This description may use natural language, diagrams or other notations that are understandable by customers. Product and process standards which must be followed should be specified.</td>
</tr>
<tr>
<td>definition</td>
<td></td>
</tr>
<tr>
<td>System architecture</td>
<td>This chapter should present a high-level overview of the anticipated system architecture showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.</td>
</tr>
<tr>
<td>System requirements</td>
<td>This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements, e.g. interfaces to other systems may be defined.</td>
</tr>
<tr>
<td>specification</td>
<td></td>
</tr>
<tr>
<td>System models</td>
<td>This should set out one or more system models showing the relationships between the system components and the system and its environment. These might be object models, data-flow models and semantic data models.</td>
</tr>
<tr>
<td>System evolution</td>
<td>This should describe the fundamental assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs, etc.</td>
</tr>
<tr>
<td>Appendices</td>
<td>These should provide detailed, specific information which is related to the application which is being developed. Examples of appendices that may be included are hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organisation of the data used by the system and the relationships between data.</td>
</tr>
<tr>
<td>Index</td>
<td>Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, etc.</td>
</tr>
</tbody>
</table>

4. Explain requirements elicitation and analysis process and its activities.
   [Jan 10, June 10, Dec 11] [10M]
The term *stakeholder* is used to refer to any person or group who will be affected by the system, directly or indirectly. Stakeholders include end-users who interact with the system and everyone else in an organisation that may be affected by its installation. Other system stakeholders may be engineers who are developing or maintaining related systems, business managers, domain experts and trade union representatives.

Eliciting and understanding stakeholder requirements is difficult for several reasons:
1. Stakeholders often don't know what they want from the computer system except in the most general terms. They may find it difficult to articulate what they want the system to do or make unrealistic demands because they are unaware of the cost of their requests.
2. Stakeholders naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, must understand these requirements.
3. Different stakeholders have different requirements, which they may express in different ways. Requirements engineers have to consider all potential sources of requirements and discover commonalities and conflict.
4. Political factors may influence the requirements of the system. For example, managers may demand specific system requirements that will increase their influence in the organisation.
5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. Hence the importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

A very general process model of the elicitation and analysis process is shown in figure:
5. Why requirements need to be validated? Explain the check made in requirement validation. [June 10, July 09] [6M]

Requirements validation is concerned with showing that the requirements actually define the system that the customer wants. Requirements validation is important because errors in a requirements document can lead to extensive rework costs when they are discovered during development or after the system is in service. The cost of fixing a requirements problem by making a system change is much greater than repairing design or coding errors.

The reason for this is that a change to the requirements usually means that the system design and implementation must also be changed and then the system must be tested again.

During the requirements validation process, checks should be carried out on the requirements in the requirements document. These checks include:

1. **Validity checks** A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with distinct needs, and any set of requirements is inevitably a compromise across the stakeholder community.

2. **Consistency checks** Requirements in the document should not conflict. That is, there should be no contradictory constraints or descriptions of the same system function.

3. **Completeness checks** The requirements document should include requirements, which define all functions, and constraints intended by the system user.

4. **Realism checks** Using knowledge of existing technology, the requirements should be checked to ensure that they could actually be implemented. These checks should also take account of the budget and schedule for the system development.

5. **Verifiability** To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

6. What are the enduring and violate requirements? Also give the classification of violate requirement with brief explanation. [Jan 09, Dec 11] [10M]

1. **Enduring requirements** These are relatively stable requirements that derive from the core activity of the organisation and which relate directly to the domain of the system. For example, in a hospital, there will always be requirements concerned with patients, doctors, nurses and treatments. These requirements may be derived from domain models that show the entities and relations that characterize an application domain.

2. **Volatile requirements** are requirements that are likely to change during the system development process or after the system has been become operational. An example would be requirements resulting from government healthcare policies.
<table>
<thead>
<tr>
<th>Requirement Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutable requirements</td>
<td>Requirements which 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.</td>
</tr>
<tr>
<td>Emergent requirements</td>
<td>Requirements which emerge as the customer’s understanding of the system develops during the system development. The design process may reveal new emergent requirements.</td>
</tr>
<tr>
<td>Consequential requirements</td>
<td>Requirements which result from the introduction of the computer system. Introducing the computer system may change the organisation’s processes and open up new ways of working which generate new system requirements.</td>
</tr>
<tr>
<td>Compatibility requirements</td>
<td>Requirements which 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.</td>
</tr>
</tbody>
</table>
UNIT-4

1. **What is data-flow model? With an example show the notations used in data flow model.** [Jan 10] [10M]

Data-flow models are an intuitive way of showing how data is processed by a system. At the analysis level, they should be used to model the way in which data is processed in the existing system. Data-flow models are used to show how data flows through a sequence of processing steps. For example, a processing step could be to filter duplicate records in a customer database. The data is transformed at each step before moving on to the next stage. These processing steps or transformations represent software processes or functions when data-flow diagrams are used to document a software design. A data-flow model, which shows the steps involved in processing an order for goods (such as computer equipment) in an organisation, is illustrated in figure.

![Diagram](image)

This particular model describes the data processing in the Place equipment order activity in the overall process model shown in fig. The model shows how the order for the goods moves from process to process. It also shows the data stores (Orders file and Budget file) that are involved in this process.

2. **Explain the object aggregation with example.** [Jan 09] [5M]

**Object aggregation:** an object is an aggregate of a set of other objects. The classes representing these objects may be modeled using an object aggregation model, as shown in figure.
Example: Model of a library item, which is a study pack for a university course. This study pack includes lecture notes, exercises, sample solutions, copies of transparencies used in lectures, and videotapes. The UML notation for aggregation is to represent the composition by including a diamond shape on the source of the link. Therefore figure can be read as 'A study pack is composed of one of more assignments, OHP slide packages, lecture notes and videotapes.
2. **Draw and explain state transition model of a simple microwave oven.**  
   [July 09] [10M]

A state machine model describes how a system responds to internal or external events. The state machine model shows system states and events that cause transitions from one state to another. It does not show the flow of data within the system. This type of model is often used for modeling real-time systems because these systems are often driven by stimuli from the system's environment.

Sequence of actions in using the microwave is:
1. Select the power level (either half-power or full-power).
2. Input the cooking time.
3. Press Start, and the food is cooked for the given time.

4. **Describe the three categories of risks. Explain different stages in risk management.** [Jan 10] [10M]

There are three related categories of risk:
1. **Project risks** are risks that affect the project schedule or resources. An example might be the loss of an experienced designer.
2. **Product risks** are risks that affect the quality or performance of the software being developed.
   
   Example: the failure of a purchased component to perform as expected.
3. **Business risks** are risks that affect the organisation developing or procuring the software. For example, a competitor introducing a new product is a business risk.
The process of risk management is illustrated in figure.

It involves the stages:
1. **Risk identification** Possible project, product and business risks are identified.
2. **Risk analysis** The likelihood and consequences of these risks are assessed.
3. **Risk planning** Plans to address the risk either by avoiding it or minimizing its effects on the project are drawn up.
4. **Risk monitoring** The risk is constantly assessed and plans for risk mitigation are revised as more information about the risk becomes available.

5. **Describe different types of system models.** [Dec 2012][5M]

System modeling: System modeling 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
- Behavioral perspective showing the behavior of the system
- Structural perspective showing the system or data architecture

**Context models**
- Context models are used to illustrate the boundaries of a system
- Social and organizational concerns may affect the decision on where to position system boundaries
- Architectural models show the a system and its relationship with other systems

**Behavioural models**
- Behavioural models are used to describe the overall behaviour of a system
- Two types of behavioural model are shown here
- 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
- Both of these models are required for a description of the system’s behaviour

**Structured methods**
- Structured methods incorporate system modeling as an inherent part of the method
- Methods define a set of models, a process for deriving these models and rules and guidelines that should apply to the models
- CASE tools support system modeling as part of a structured method
UNIT - 5

1. Explain why it is necessary to design the system architecture. What are the system factors affected by system architecture. [Jan 10] [10M]

1. *Stakeholder communication* The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.
2. *System analysis* Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether the system can meet critical requirements such as performance, reliability and maintainability.
3. *Large-scale reuse* A system architecture model is a compact, manageable description of how a system is organised and how the components interoperate. The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse.

**The system architecture affects the performance, robustness, distributability and maintainability of a system.**

1. *Performance* If performance is a critical requirement, the architecture should be designed to localise critical operations within a small number of subsystems, with as little communication as possible between these sub-systems.
2. *Security* If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers and with a high level of security validation applied to these layers.
3. *Safety* If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single sub-system or in a small number of sub-systems.
4. *Availability* If availability is a critical requirement, the architecture should be designed to include redundant components and so that it is possible to replace and update components without stopping the system.
5. *Maintainability* If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be changed.

2. With an example describe the repository model and give its advantages and disadvantages. [Jan 09, Dec 11, Dec 2012] [5M]

A system model based on a shared database is called a *repository model*. Below figure is an example of a CASE toolset architecture based on a shared repository.
The advantages and disadvantages of a shared repository are as follows:
1. It is an efficient way to share large amounts of data. There is no need to transmit data explicitly from one sub-system to another.
2. However, sub-systems must agree on the repository data model. Inevitably, this is a compromise between the specific needs of each tool. Performance may be adversely affected by this compromise. It may be difficult or impossible to integrate new sub-systems if their data models do not fit the agreed schema.
3. Sub-systems that produce data need not be concerned with how that data is used by other sub-systems.
4. However, evolution may be difficult as a large volume of information is generated according to an agreed data model. Translating this to a new model will certainly be expensive; it may be difficult or even impossible.
5. Activities such as backup, security, access control and recovery from error are centralised. They are the responsibility of the repository manager. Tools can focus on their principal function rather than be concerned with these issues.
6. However, different sub-systems may have different requirements for security, recovery and backup policies. The repository model forces the same policy on all sub-systems.
7. The model of sharing is visible through the repository schema. It is straightforward to integrate new tools given that they are compatible with the agreed data model.
8. However, it may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralised repository, there may be problems with data redundancy and inconsistency.

3. Explain two types of control styles [June 10] [10M]

Control models at the architectural level are concerned with the control flow between sub-systems. There are two generic control styles that are used in software systems:
1. Centralised control One sub-system has overall responsibility for control and starts and stops other sub-systems. It may also devolve control to another subsystem but will expect to have this control responsibility returned to it.
2. **Event-based control** Rather than control information being embedded in a subsystem, each sub-system can respond to externally generated events. These events might come from other sub-systems or from the environment of the system. In a centralised control model, one sub-system is designated as the system controller and has responsibility for managing the execution of other sub-systems. Centralised control models fall into two classes, depending on whether the controlled sub-systems execute sequentially or in parallel.

1. **The call-return model** This is the familiar top-down subroutine model where control starts at the top of a subroutine hierarchy and, through subroutine calls, passes to lower levels in the tree. The subroutine model is only applicable to sequential systems.

2. **The manager model** This is applicable to concurrent systems. One system component is designated as a system manager and controls the starting, stopping and coordination of other system processes. A process is a sub-system or module that can execute in parallel with other processes. A form of this model may also be applied in sequential systems where a management routine calls particular sub-systems depending on the values of some state variables. This is usually implemented as a case statement.

A centralised control model for a real-time system:

![Centralised Control Model Diagram]

Event-driven control models are driven by externally generated events. The term event in this context does not just mean a binary signal. Two event-driven control models:

1. **Broadcast models** In these models, an event is broadcast to all sub-systems. Any sub-system that has been programmed to handle that event can respond to it.

2. **Interrupt-driven models** These are exclusively used in real-time systems where external interrupts are detected by an interrupt handler. They are then passed to some other component for processing.

A control model based on selective broadcasting:
4. Explain the stages in object oriented design process. [Jan 10, Jan 09] [6M]

Stages:
1. Understand and define the context and the modes of use of the system.
2. Design the system architecture.
3. Identify the principal objects in the system.
4. Develop design models.
5. Specify object interfaces.

**The system context and the model of system:**
Use to represent two complementary models of the relationships between a system and its environment:
1. The system context is a static model that describes the other systems in that environment.
2. The model of the system use is a dynamic model that describes how the system actually interacts with its environment.

**Architectural design:**
Example: The three layers in weather station software:
1. The interface *layer* that is concerned with all communications with other parts of the system and with providing the external interfaces of the system;
2. The *data collection layer* that is concerned with managing the collection of data from the instruments and with summarizing the weather data before transmission to the mapping system;
3. The *instruments layer* that is an encapsulation of all of the instruments used to collect raw data about the weather conditions.

**Object identification:**
There have been various proposals made about how to identify object classes:
1. Use a grammatical analysis of a natural language description of a system.
2. Use tangible entities (things) in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings,
3. Use a behavioral approach where the designer first understands the overall behavior of the system.
4. Use a scenario-based analysis where various scenarios of system use are identified and analysed in turn.

**Design models:**
Design models show the objects or object classes in a system and, where appropriate, the relationships between these entities. Design models essentially are the design. They are the bridge between the requirements for the system and the system implementation. This means that there are conflicting requirements on these models. There are two types of design models that should normally be produced to describe an object-oriented design:
1. **Static models** describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalisation relationships, uses/used-by relationships and composition relationships.
2. **Dynamic models** describe the dynamic structure of the system and show the interactions between the system objects (not the object classes). Interactions that may be documented include the sequence of service requests made by objects and the way in which the state of the system is related to these object interactions.

**Object interface specification:** An important part of any design process is the specification of the interfaces between the components in the design. You need to specify interfaces so that objects and sub-systems can be designed in parallel. Once an interface has been specified, the developers of other objects may assume that interface will be implemented.

5. **Write short notes on client server model, layered model. [Dec 2012] [10M]**

**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 organization. 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
Layered model

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data display</td>
<td>Data display layer where objects are concerned with preparing and presenting the data in a human-readable form</td>
</tr>
<tr>
<td>Data archiving</td>
<td>Data archiving layer where objects are concerned with storing the data for future processing</td>
</tr>
<tr>
<td>Data processing</td>
<td>Data processing layer where objects are concerned with checking and integrating the collected data</td>
</tr>
<tr>
<td>Data collection</td>
<td>Data collection layer where objects are concerned with acquiring data from remote sources</td>
</tr>
</tbody>
</table>

System context and models of use

Develop an understanding of the relationships between the software being designed and its external environment

- **System context**: A static model that describes other systems in the environment. Use a subsystem model to show other systems. Following slide shows the systems around the weather station system.
- **Model of system use**: A dynamic model that describes how the system interacts with its environment. Use use-cases to show interactions
UNIT – 6

1. What are the characteristics of Rapid software development? [July 09] [5M]

Rapid application development (RAD) techniques evolved from so-called fourth-generation languages in the 1980s and are used for developing applications that are data-intensive. Consequently, they are usually organised as a set of tools that allow data to be created, searched, displayed and presented in reports.

Figure illustrates a typical organisation for a RAD system.

![RAD System Diagram]

The tools that are included in a RAD environment are:
1. A database programming language that embeds knowledge of the database structure; and includes fundamental database manipulation operations. SQL is the standard database programming language. The SQL commands may be input directly or generated automatically from forms filled in by an end-user.
2. An interface generator, which is used to create forms for data input and display.
3. Links to office applications such as a spreadsheet for the analysis and manipulation of numeric information or a word processor for report template creation.
4. A report generator, which is used to define and create reports from information in the database.

Many business applications rely on structured forms for input and output, so RAD environments provide powerful facilities for screen definition and report generation. Screens are often defined as a series of linked forms, so the screen generation system must provide for:
1. Interactive form definition where the developer defines the fields to be displayed and how these are to be organised.
2. Form linking where the developer can specify that particular inputs cause further forms to be displayed.
3. Field verification where the developer defines allowed ranges for values input to form fields.
2. What are agile methods? Explain the principles of Agile methods. [Jan 09, July 09, June 10, Dec 2012] [7M]

Agile method is a group of software development methods based on iterative and incremental development.

Principles of Agile methods:

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

3. Write a note on extreme programming and discuss advantages of pair programming. [June 10, Jan 09] [6M]

Extreme programming (XP) is perhaps the best known and most widely used of the agile methods. The extreme programming release cycle:

Extreme programming involves a number of agile methods:

1. Incremental development is supported through small, frequent releases of the system and by an approach to requirements description based on customer stories or scenarios that can be the basis for process planning.
2. Customer involvement is supported through the full-time engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.

3. People, not process, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.

4. Change is supported through regular system releases, test-first development and continuous integration.

5. Maintaining simplicity is supported through constant refactoring to improve code quality and using simple designs that do not anticipate future changes to the system.

Pair programming:
The idea is that pairs are created dynamically so that all team members may work with other members in a programming pair during the development process.
The use of pair programming has a number of advantages:
1. It supports the idea of common ownership and responsibility for the system.
2. It acts as an informal review process because each line of code is looked at by at least two people.
3. It helps support refactoring, which is a process of software improvement. A principle of XP is that the software should be constantly refactored.

4. What is software prototyping? Explain its benefits. [July 09, June 10] [5M]

A prototype: is an initial version of a software system that is used to demonstrate concepts, try out design options and, generally, to find out more about the problem and its possible solutions. A software prototype can be used in a software development process in following ways:
1. In the requirements engineering process, a prototype can help with the elicitation and validation of system requirements.
2. In the system design process, a prototype can be used to explore particular software solutions and to support user interface design.
3. In the testing process, a prototype can be used to run back-to-back tests with the system that will be delivered to the customer. benefits of using prototyping are:
   1. Improved system usability
   2. A closer match of the system to users needs
   3. Improved design quality
   4. Improved maintainability
   5. Reduced development effort
A process model for prototype development is shown in figure.
5. What are the different types of software maintenance? What are the key factors that distinguish development and maintenance? [Jan 10, Jan 09, July 09] [10M]

There are three different types of software maintenance:
1. **Maintenance to repair software faults** Coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve rewriting several program components. Requirements errors are the most expensive to repair because of the extensive system redesign that may be necessary.
2. **Maintenance to adapt the software to a different operating environment** This type of maintenance is required when some aspect of the system's environment such as the hardware, the platform operating system or other support software changes. The application system must be modified to adapt it to cope with these environmental changes.
3. **Maintenance to add to or modify the system's functionality** This type of maintenance is necessary when the system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance.

The **key factors** that distinguish development and maintenance:
1. **Team stability:** After a system has been delivered, it is normal for the development team to be broken up and people work on new projects. The new team or the individuals responsible for system maintenance do not understand the system or the background to system design decisions.
2. **Contractual responsibility:** The contract to maintain a system is usually separate from the system development contract. The maintenance contract may be given to a different company rather than the original system developer. This factor, along with the lack of team stability, means that there is no incentive for a development team to write the software so that it is easy to change.
3. **Staff skills** Maintenance: Staffs are often relatively inexperienced and unfamiliar with the application domain. Maintenance has a poor image among software engineers. It is seen as a less skilled process than system development and is often allocated to the most junior staff.
4. Program age and structure: As programs age, their structure tends to be
change, so they become harder to understand and modify. Some systems have
been
developed without modem software engineering techniques. They may never have
been
well structured and were perhaps optimized for efficiency rather than
understandability.

6. With a neat diagram describe the system evolution process. [Dec 2012][7M]
The system evolution process

Change implementation

Urgent change requests
Urgent changes may have to be implemented without going through all stages
of the software engineering process
- If a serious system fault has to be repaired;
- If changes to the system’s environment (e.g. an OS upgrade) have
  unexpected effects;
- If there are business changes that require a very rapid response (e.g.
  the release of a competing product).
7. **Explain re-engineering process. [Dec 2012][7M]**

System re-engineering

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

Advantages of reengineering

Reduced risk

- There is a high risk in new software development. There may be development problems, staffing problems and specification problems.

Reduced cost

- The cost of re-engineering is often significantly less than the costs of developing new software.
1. What is verification and validation? Explain. [June 10] [5M]

Verification and validation (V & V) is the process of checking and analysis. Meaning of Verification and validation:
'Validation: Are we building the right product?'
'Verification: Are we building the product right?'
Static and dynamic verification and validation:

Within the V & V process, there are two complementary approaches to system checking and analysis:
1. Software inspections or peer reviews analyse and check system representations such as the requirements document, design diagrams and the program source code. You can use inspections at all stages of the process. Inspections may be supplemented by some automatic analysis of the source text of a system or associated documents. Software inspections and automated analyses are static V & V techniques, as you don't need to run the software on a computer.

2. Software testing involves running an implementation of the software with test data. You examine the outputs of the software and its operational behavior to check that it is performing as required. Testing is a dynamic technique of verification and validation.

2. Write a note on software inspection process. [Jan 10] [5M]

Software inspection is a static V & V process in which a software system is reviewed to find errors, omissions and anomalies. Generally, inspections focus on source code.
There are three major advantages of inspection over testing:
1. During testing, errors can mask (hide) other errors. Once one error is discovered, you can never be sure if other output anomalies are due to a new error or are side effects of the original error. Because inspection is a static process, you don't have to be concerned
with interactions between errors. Consequently, a single inspection session can discover many errors in a system.

2. Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialised test harnesses to test the parts that are available. This obviously adds to the system development costs.

3. As well as searching for program defects, an inspection can also consider broader quality attributes of a program such as compliance with standards, portability and maintainability. You can look for inefficiencies, inappropriate algorithms and poor programming style that could make the system difficult to maintain and update.

3. Describe the characteristics of clean room software development with neat diagram. [Jan 09, Jan 10] [5M]

The objective of this approach to software development is zero-defect software. The name 'Clean-room was derived by analogy with semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere. The Clean-room approach to software development is based on five key strategies:

1. **Formal specification:** The software to be developed is formally specified. A state transition model that shows system responses to stimuli is used to express the specification.

2. **Incremental development:** The software is partitioned into increments that are developed and validated separately using the Clean-room process. These increments are specified, with customer input, at an early stage in the process.

3. **Structured programming:** Only a limited number of control and data abstraction constructs are used. The program development process is a process of stepwise refinement of the specification. A limited number of constructs are used and the aim is to systematically transform the specification to create the program code.

4. **Static verification:** The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.

5. **Statistical testing of the system:** The integrated software increment is tested statistically to determine its reliability. These statistical tests are based on an operational profile, which is developed in parallel with the system specification as shown in figure.
4. Explain with illustration: i. Integration testing, ii. Release testing
[Jan 10, Dec 2012] [12M]

4. 1. Integration testing:

The process of system integration involves building a system from its components and testing the resultant system for problems that arise from component interactions. The components that are integrated may be off-the-shelf components, reusable components that have been adapted for a particular system or newly developed components. For many large systems, all three types of components are likely to be used. Integration testing checks that these components actually work together, are called correctly and transfer the right data at the right time across their interfaces.

*Top-down integration:* the overall skeleton of the system is developed first, and components are added to it.

*Bottom-up integration:* first integrate infrastructure components that provide common services, such as network and database access, then add the functional components.

A major problem that arises during integration testing is localising errors. There are complex interactions between the system components and, when an anomalous output is discovered, you may find it hard to identify where the error occurred. To make it easier to locate errors, you should always use an incremental approach to system integration and testing.
In the example shown in figure, A, B, C and D are components and n to T5 are related sets of tests of the features incorporated in the system. n, T2 and T3 are first run on a system composed of component A and component B (the minimal system). If these reveal defects, they are corrected. Component C is integrated and n, T2 and T3 are repeated to ensure that there have not been unexpected interactions with A and B. If problems arise in these tests, this probably means that they are due to interactions with the new component. The source of the problem is localised, thus simplifying defect location and repair. Test set T4 is also run on the system. Finally, component D is integrated and tested using existing and new tests (T5).

2. Release Testing:

Release testing is the process of testing a release of the system that will be distributed to customers. The primary goal of this process is to increase the supplier's confidence that the system meets its requirements. If so, it can be released as a product or delivered to the customer. Release testing is usually a black-box testing process where the tests are derived from the system specification. The system is treated as a black box whose behavior can only be determined by studying its inputs and the related outputs. Another name for this is functional testing because the tester is only concerned with the functionality and not the implementation of the software. Figure illustrates the model of a system that is assumed in black-box testing.
The tester presents inputs to the component or the system and examines the corresponding outputs. If the outputs are not those predicted (i.e., if the outputs are in set $O_e$) then the test has detected a problem with the software. $I_e$: inputs that have a high probability of generating system failures.
UNIT – 8

1. Explain factors governing staff selection. [June 10, Dec 2012] [10M]

Application domain experience: For a project to develop a successful system, the developers must understand the application domain. It is essential that some members of a development team have some domain experience.

Platform experience: This may be significant if low-level programming is involved. Otherwise, this is not usually a critical attribute.

Programming language experience: This is normally only significant for short-duration projects where there is not enough time to learn a new language. While learning a language itself is not difficult, it takes several months to become proficient in using the associated libraries and components.

Problem solving ability: This is very important for software engineers who constantly have to solve technical problems. However, it is almost impossible to judge without knowing the work of the potential team member.

Educational background: This may provide an indicator of what the candidate knows and his or her ability to learn. This factor becomes increasingly irrelevant as engineers gain experience across a range of projects.

Communication ability: Project staff must be able to communicate orally and in writing with other engineers, managers and customers.

Adaptability: Adaptability may be judged by looking at the experience that candidates have had. This is an important attribute, as it indicates an ability to learn.

Attitude: Project staff should have a positive attitude toward their work and should be willing to learn new skills. This is an important attribute but often very difficult to assess.

Personality: This is an important attribute but difficult to assess. Candidates must be reasonably compatible with other team members. No particular type of personality is more or less suited to software engineering.
2. Explain Maslow’s human-needs hierarchy of motivating people. [Jan 10][10M]

![Maslow's Hierarchy of Needs](image)

Maslow (Maslow 1954) suggests that people are motivated by satisfying their needs and that needs are arranged in a series of levels. The lower levels of this hierarchy represent fundamental needs for food, sleep, and so on, as well as the need to feel secure in an environment. Social needs are concerned with the need to feel part of a social grouping. Esteem needs are the need to feel respected by others, and self-realisation needs are concerned with personal development. Human priorities are to satisfy lower-level needs such as hunger before the more abstract, higher-level needs.

Ensuring the satisfaction of social, esteem and self-realisation needs is most significant from a management point of view.

1. To satisfy social needs, you need to give people time to meet their co-workers and to provide: places for them to meet. This is relatively easy when all of the members of a development team work in the same place but, increasingly, team members are not located in the same building or even the same town or state.

They may work for different organizations or from home most of the time. Electronic communications such as e-mail and teleconferencing may be used to support this remote working. However, my experience with electronic communications is that they do not really satisfy social needs. If your team is distributed, you should arrange periodic face-to-face meetings so that people experience direct interaction with other members of the team. Through this direct interaction, people become part of a social group and may be motivated by the goal and priorities of that group.

2. To satisfy esteem needs, you need to show people that they are valued by the organization. Public recognition of achievements is a simple yet effective way of doing this. Obviously, people must also feel that they are paid at a level that reflects their skills and experience.

3. Finally, to satisfy self-realization needs, you need to give people responsibility for their work, assign them demanding (but not impossible) tasks and provide a training programme where people can develop their skills.
3. Describe with block diagram, SEI people- CMM [Jan 09, June 10, Dec 2012]|10M|

The Software Engineering Institute (SEI) in the United States is engaged in a long term programme of software process improvement. Part of this programme is the Capability Maturity Model (CMM) for software processes. The P-CMM can be used as a framework for improving the way in which an organisation manages its human assets.

The five levels are:
1. **Initial** Ad hoc, informal people management practices
2. **Repeatable** Establishment of policies for developing the capability of the staff
3. **Defined** Standardization of best people management practice across the organisation
4. **Managed** Quantitative goals for people management
5. **Optimizing** Continuous focus on improving individual competence and workforce motivation

The strategic objectives of the P-CMM are:
1. To improve the capability of software organisations by increasing the capability of their workforce
2. To ensure that software development capability is an attribute of the organization rather than of a few individuals
3. To align the motivation of individuals with that of the organisation
4. To retain valuable human assets (i.e., people with critical knowledge and skills) within the organisation.

4. What are the factors affecting software pricing? What are the two types of metric used? Explain. [Jan 10] [10M]

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

Two types of metric that have been used:
1. Size-related metrics: These are related to the size of some output from an activity. The most commonly used size-related metric is lines of delivered source code. Other metrics that may be used are the number of delivered object code instructions or the number of pages of system documentation.
2. Function-related metrics: These are related to the overall functionality of the delivered software. Productivity is expressed in terms of the amount of useful functionality produced in some given time. Function points and object points are the best-known metrics of this type.
5. Explain COCOMO model. [Jan 09] [10M]

A number of algorithmic models have been proposed as the basis for estimating the effort, schedule and costs of a software project. These are conceptually similar but use different parameter values. The model that I discuss here is the COCOMO model. The COCOMO model is an empirical model that was derived by collecting data from a large number of software projects. These data were analysed to discover formulae that were the best fit to the observations. These formulae link the size of the system and product, project and team factors to the effort to develop the system.

It is chosen to use the COCOMO model for several reasons:

1. It is well documented, available in the public domain and supported by public domain and commercial tools.
2. It has been widely used and evaluated in a range of organisations.
3. It has a long pedigree from its first instantiation in 1981 (Boehm, 1981), through a refinement tailored to Ada software development (Boehm and Royce, 1989), to its most recent version, COCOMO II, published in 2000 (Boehm, et al. 2000),
6. Describe different cost estimation techniques. [Dec 2012][10M]

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithmic cost</td>
<td>A model is developed using historical cost information that relates some software metric (usually its size) to the project cost. An estimate is made of that metric and the model predicts the effort required.</td>
</tr>
<tr>
<td>modelling</td>
<td></td>
</tr>
<tr>
<td>Expert judgement</td>
<td>Several experts on the proposed software development techniques and the application domain are consulted. They each estimate the project cost. These estimates are compared and discussed. The estimation process iterates until an agreed estimate is reached.</td>
</tr>
<tr>
<td>Estimation by analogy</td>
<td>This technique is applicable when other projects in the same application domain have been completed. The cost of a new project is estimated by analogy with these completed projects. Myers (Myers, 1989) gives a very clear description of this approach.</td>
</tr>
<tr>
<td>Parkinson's Law</td>
<td>Parkinson’s Law states that work expands to fill the time available. The cost is determined by available resources rather than by objective assessment. If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 person-months.</td>
</tr>
<tr>
<td>Pricing to win</td>
<td>The software cost is estimated to be whatever the customer has available to spend on the project. The estimated effort depends on the customer’s budget and not on the software functionality.</td>
</tr>
</tbody>
</table>