**Q.2** a. What is a risk? Is it economical to do risk management? What is the effect of this activity on the overall cost of the project?

**Answer:**

The ISO 31000 (2009) /ISO Guide 73:2002 definition of risk is the 'effect of uncertainty on objectives'. In this definition, uncertainties include events (which may or not happen) and uncertainties caused by ambiguity or a lack of information. It also includes both negative and positive impacts on objectives. Many definitions of risk exist in common usage, however this definition was developed by an international committee representing over 30 countries and is based on the input of several thousand subject matter experts. Many, however, disagree with any alteration of the common definition of the word "risk" (possibility of loss or injury). Trying to include "positive impacts on objectives" within the definition of the word "risk" causes confusion and great difficulty communicating concepts.

Risk management is simply a practice of systematically selecting cost effective approaches for minimizing the effect of threat realization to the organization. All risks can never be fully avoided or mitigated simply because of financial and practical limitations. Therefore all organizations have to accept some level of residual risks.

Whereas risk management tends to be preemptive, business continuity planning (BCP) was invented to deal with the consequences of realized residual risks. The necessity to have BCP in place arises because even very unlikely events will occur if given enough time. Risk management and BCP are often mistakenly seen as rivals or overlapping practices. In fact these processes are so tightly tied together that such separation seems artificial. For example, the risk management process creates important inputs for the BCP (assets, impact assessments, cost estimates etc.). Risk management also proposes applicable controls for the observed risks. Therefore, risk management covers several areas that are vital for the BCP process. However, the BCP process goes beyond risk management's preemptive approach and assumes that the disaster **will** happen at some point.

In the more general case, every probable risk can have a pre-formulated plan to deal with its possible consequences (to ensure *contingency* if the risk becomes a *liability*).

From the information above and the average cost per employee over time, or cost accrual ratio, a project manager can estimate:

- the cost associated with the risk if it arises, estimated by multiplying employee costs per unit time by the estimated time lost (*cost impact*, *C* where *C = cost accrual ratio * S*).
- the probable increase in time associated with a risk (*schedule variance due to risk*, *Rs* where Rs = P * S):
  - Sorting on this value puts the highest risks to the schedule first. This is intended to cause the greatest risks to the project to be attempted first so that risk is minimized as quickly as possible.
  - This is slightly misleading as *schedule variances* with a large P and small S and vice versa are not equivalent.
- the probable increase in cost associated with a risk (*cost variance due to risk*, *Rc* where Rc = P*C = P*CAR*S = P*S*CAR)

- sorting on this value puts the highest risks to the budget first.
- see concerns about *schedule variance* as this is a function of it, as illustrated in the equation above.

  Risk in a project or process can be due either to Special Cause Variation or Common Cause Variation and requires appropriate treatment. That is to re-iterate the concern about extremal cases not being equivalent in the list immediately above.

b. Differentiate between generic and customised software products. Also give a brief note on legacy system.

**Answer:**

Generic software development is the one done for "General Purpose" audience whist "Custom Software Development" is done to satisfy a particular need of a particular client. General Purpose software development is tough as compared with Custom made; not by the skills required to develop the application but by the design and marketing point of view.

In General Purpose application/product design and development, you will need to "imagine" what an end-user require. Here, the term "end - user" has no face; you have to imagine it. Market Surveys and general Customer Demand analysis may help a company to reduce the risk factor and think about some innovations over existing similar solutions.

On the other hand, in Custom Made application/product development, you have a specific face of "end - user" in front of you. You know whom you need to satisfy. Understanding the need and Analyzing it to get the best out of it is a challenge here. Planning and reaching the goals within dead-line adds a value to your software development excellence as a professional service provider.

By Buyer's (or Client's) point of view, now a days it's preferred to have a Custom Made application developed rather than buying a General Purpose software. You get the application done that exactly matches your requirements and also most importantly your style of computing. If you manage to find a quality Freelance Developer (or team) you're 50% done! If you successfully locate desired talent in Asian countries (such as India) then it's a bonus; because $1 is still Re.40 here. So, the overall development cost for US and Europian companies is likely to get slashed down by at least 40%.

A **legacy system** is an old method, technology, computer system, or application program. The legacy system may or may not remain in use. Even if it is no longer used, it may continue to impact the organization due to its historical role. Historic data may not have been converted into the new system format and may exist within the new system with the use of a customized schema crosswalk, or may exist only in a data warehouse. In either case, the effect on business intelligence and operational reporting can be significant. For a variety of reasons, a legacy system may continue to be used, sometimes well past its vendor-supported lifetime, resulting in support and maintenance challenges. It may be that the system still provides for the users' needs, even though newer technology or more efficient methods of performing a task are now available. However, the decision to keep an old system may be influenced by economic reasons such as return on investment challenges or vendor lock-in, the inherent challenges of change management, or a variety of other reasons other than functionality. A legacy system may include procedures or terminology which are no longer relevant in the current context, and may hinder or confuse understanding of the methods or technologies used.

The term "legacy" may have little to do with the size or age of the system — mainframes run 64-bit Linux and Java alongside 1960s vintage code.

Although the term is most commonly used to describe computers and software, it may also be used to describe human behaviors, methods, and tools. For example, timber framing using wattle and daub is a legacy building construction method.

Organizations can have compelling reasons for keeping a legacy system, such as:

- The system works satisfactorily, and the owner sees no reason for changing it.
- The costs of redesigning or replacing the system are prohibitive because it is large, monolithic, and/or complex.
- Retraining on a new system would be costly in lost time and money, compared to the anticipated appreciable benefits of replacing it (which may be zero).
- The system requires near-constant availability, so it cannot be taken out of service, and the cost of designing a new system with a similar availability level is high. Examples include systems to handle customers' accounts in banks, computer reservation systems, air traffic control, energy distribution (power grids), nuclear power plants, military defense installations, and systems such as the TOPS database.
- The way that the system works is not well understood. Such a situation can occur when the designers of the system have left the organization, and the system has either not been fully documented or documentation has been lost.
- The user expects that the system can easily be replaced when this becomes necessary.

c. Describe the various phases of a software development life cycle model. Compare evolutionary development model and iterative enhancement model.

**Answer:**

Software Development Life Cycle (SDLC) adheres to important phases that are essential for developers, such as planning, analysis, design, and implementation, and are explained in the section below.It include evaluation of present system, information gathering, feasibility study and request approval. A number of system development life cycle (SDLC) models have been created: waterfall, fountain, spiral, build and fix, rapid prototyping, incremental, and synchronize and stabilize. The oldest of these, and the best known, is the waterfall model: a sequence of stages in which the output of each stage becomes the input for the next. These stages can be characterized and divided up in different ways, including the following

- **Preliminary Analysis**: The objective of phase 1 is to conduct a preliminary analysis, propose alternative solutions, describe costs and benefits and submit a preliminary plan with recommendations.

  Conduct the preliminary analysis: in this step, you need to find out the organization's objectives and the nature and scope of the problem under study. Even if a problem refers only to a small segment of the organization itself then you need to find out what the objectives of the organization itself are. Then you need to see how the problem being studied fits in with them.
  Propose alternative solutions: In digging into the organization's objectives and specific problems, you may have already covered some solutions. Alternate proposals may come from interviewing employees, clients, suppliers, and/or consultants. You can also study what competitors are doing. With this data, you will have three choices: leave the system as is, improve it, or develop a new system.
  Describe the costs and benefits.

- **Systems analysis, requirements definition**: Defines project goals into defined functions and operation of the intended application. Analyzes end-user information needs.

- **Systems design**: Describes desired features and operations in detail, including screen layouts, business rules, process diagrams, pseudocode and other documentation.

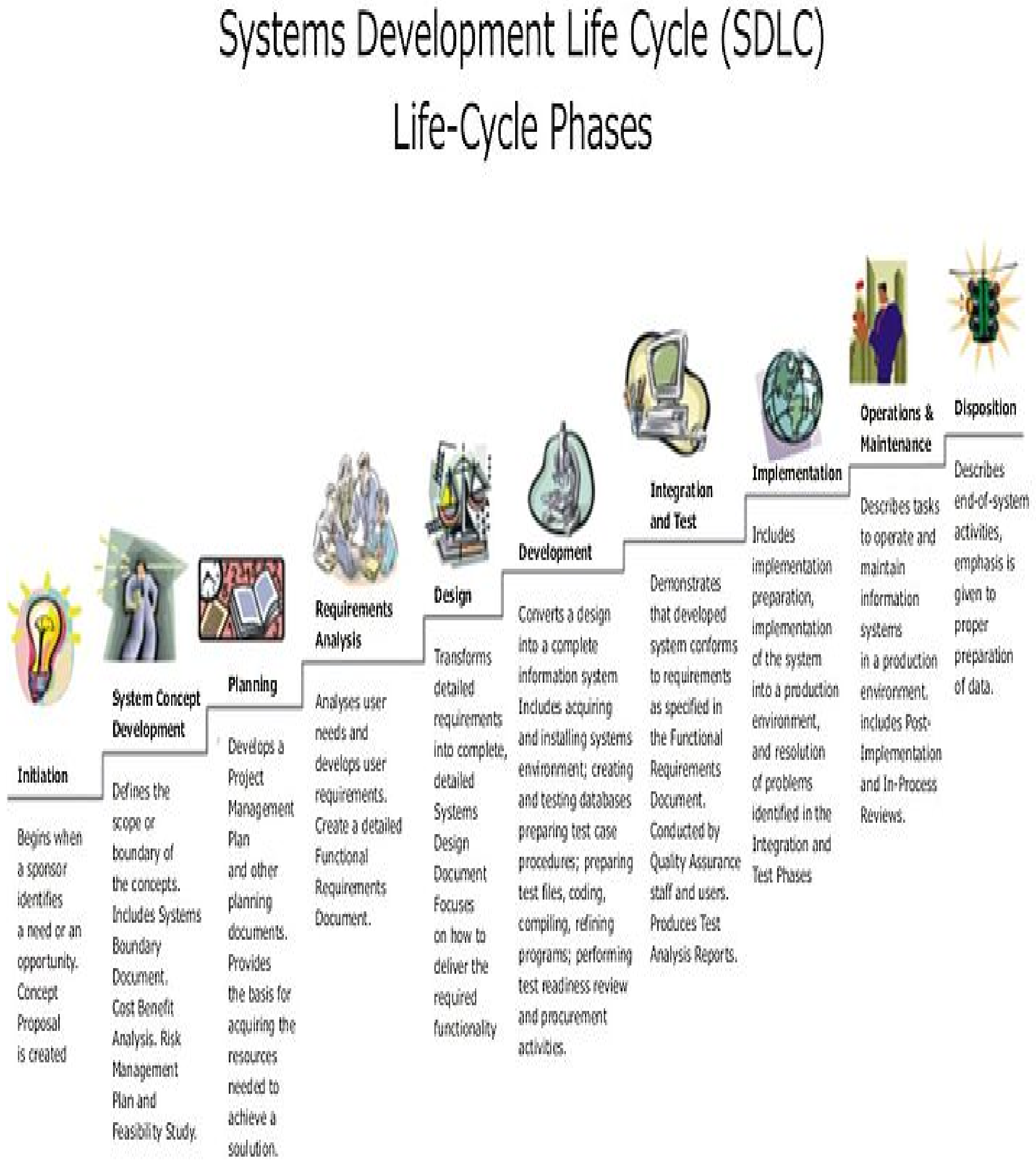
- **Development**: The real code is written here.

- **Integration and testing**: Brings all the pieces together into a special testing environment, then checks for errors, bugs and interoperability.

- **Acceptance, installation, deployment**: The final stage of initial development, where the software is put into production and runs actual business.

- **Maintenance**: What happens during the rest of the software's life: changes, correction, additions, moves to a different computing platform and more. This is often the longest of the stages.

In the following example (see picture) these stage of the systems development life cycle are divided in ten steps from definition to creation and modification of IT work products:


Systems Development Life Cycle (SDLC)

Life-Cycle Phases

**Initiation**

Begins when a sponsor identifies a need or an opportunity. Concept Proposal is created

**System Concept Development**

Defines the scope or boundary of the concepts. Includes Systems Boundary Document. Cost Benefit Analysis. Risk Management Plan and Feasibility Study.

**Planning**

Develops a Project Management Plan and other planning documents. Provides the basis for acquiring the resources needed to achieve a soulution.

**Requirements Analysis**

Analyses user needs and develops user requirements. Create a detailed Functional Requirements Document.

**Design**

Transforms detailed requirements into complete, detailed Systems Design Document Focuses on how to deliver the required functionality

**Development**

Converts a design into a complete information system Includes acquiring and installing systems environment; creating and testing databases preparing test case procedures; preparing test files, coding, compiling, refining programs; performing test readiness review and procurement activities.

**Integration and Test**

Demonstrates that developed system conforms to requirements as specified in the Functional Requirements Document. Conducted by Quality Assurance staff and users. Produces Test Analysis Reports.

**Implementation**

Includes implementation preparation, implementation of the system into a production environment, and resolution of problems identified in the Integration and Test Phases

**Operations & Maintenance**

Describes tasks to operate and maintain information systems in a production environment. includes Post-Implementation and In-Process Reviews.

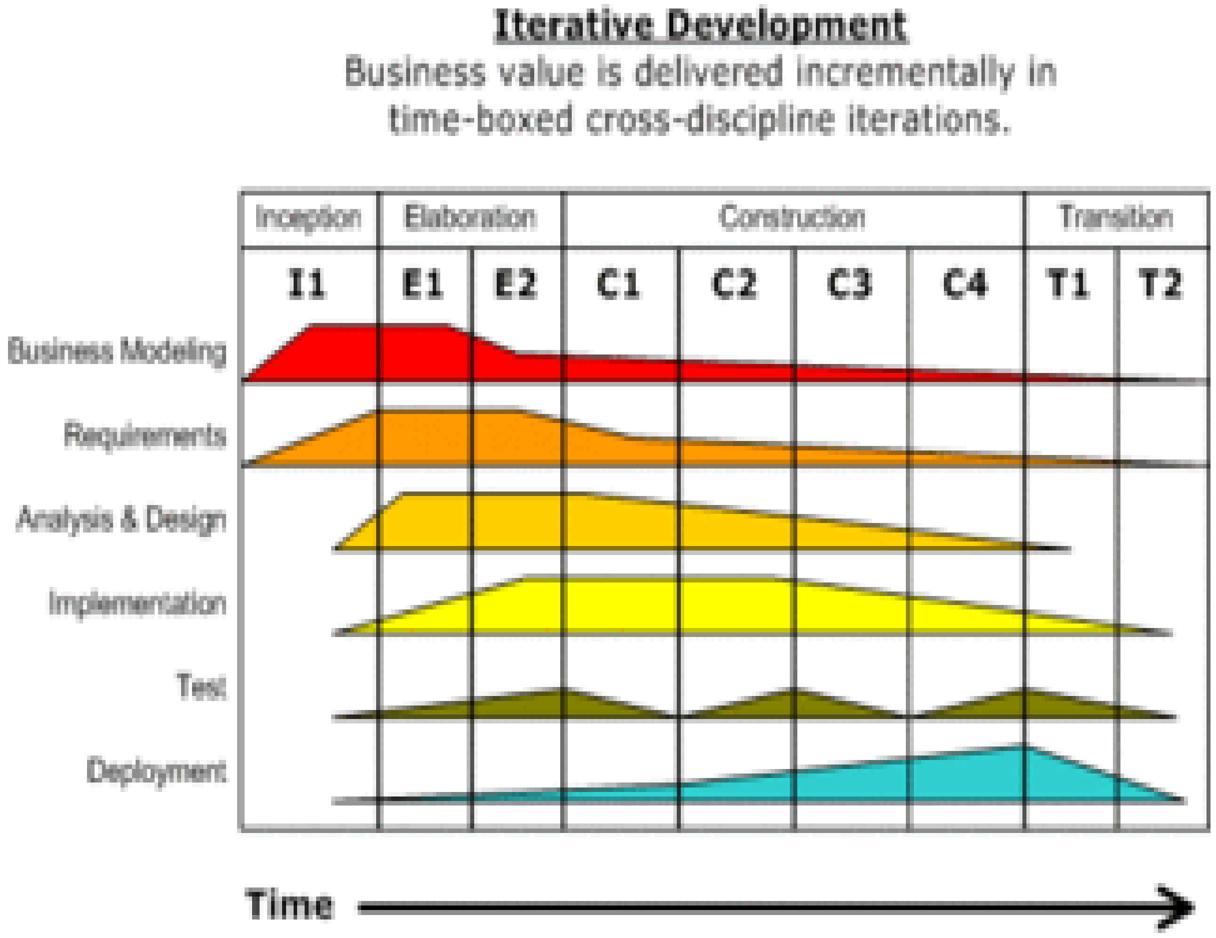
**Disposition**

Describes end-of-system activities, emphasis is given to proper preparation of data.

Incremental development slices the system functionality into increments (portions). In each increment, a slice of functionality is delivered through cross-discipline work, from the requirements to the deployment. The unified process groups increments/iterations into phases: inception, elaboration, construction, and transition.

- Inception identifies project scope, requirements (functional and non-functional) and risks at a high level but in enough detail that work can be estimated.
- Elaboration delivers a working architecture that mitigates the top risks and fulfills the non-functional requirements.
- Construction incrementally fills-in the architecture with production-ready code produced from analysis, design, implementation, and testing of the functional requirements.
- Transition delivers the system into the production operating environment.

Each of the phases may be divided into 1 or more iterations, which are usually time-boxed rather than feature-boxed. Architects and analysts work one iteration ahead of developers and testers to keep their work-product backlog full.



Software Products can be perceived as evolving over a time period.

However, neither the Linear Sequential Model nor the Prototype Model apply this aspect to software production. The Linear Sequential Model was designed for straight-line development. The Prototype Model was designed to assist the customer in understanding requirements and is designed to produce a visualization of the final system.

But the Evolutionary Models take the concept of “evolution” into the engineering paradigm. Therefore Evolutionary Models are iterative. They are built in a manner that enables software engineers to develop increasingly more complex versions of the software.

The Spiral Model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the Linear Sequential Model. Using the Spiral Model the software is developed in a series of incremental releases. Unlike the Iteration Model where in the first product is a core product, in the Spiral Model the early iterations could result in a paper model or a prototype. However, during later iterations more complex functionalities could be added.

A Spiral Model, combines the iterative nature of prototyping with the controlled and systematic aspects of the Waterfall Model, therein providing the potential for rapid development of incremental versions of the software. A Spiral Model is divided into a number of framework activities, also called task regions. These task regions could vary from 3-6 in number and they are:

- **Customer Communication -** tasks required to establish effective communication between the developer and customer.

- **Planning -** tasks required to define resources, timelines and other project related information /items.

- **Risk Analysis -** tasks required to assess the technical and management risks.

- **Engineering -** tasks required to build one or more representation of the application.

- **Construction & Release -** tasks required to construct, test and support (eg. Documentation and training)

- **Customer evaluation -** tasks required to obtain periodic customer feedback so that there are no last minute surprises.

**Q.3** a. Discuss the general format of a SRS document and discuss its main components in detail.

**Answer:**

**REVISION HISTORY** ............................................... ERROR! BOOKMARK NOT DEFINED.

**DOCUMENT APPROVAL**......................................... ERROR! BOOKMARK NOT DEFINED.

**1. INTRODUCTION**............................................................................................................ **9**

1.1 PURPOSE ............................................................................................................ 9
1.2 SCOPE ................................................................................................................ 9
1.3 DEFINITIONS, ACRONYMS, AND ABBREVIATIONS.................................................... 9

1.4 REFERENCES ........ 9
1.5 OVERVIEW ........ 9

**2. GENERAL DESCRIPTION ........ 9**

2.1 PRODUCT PERSPECTIVE ........ 9
2.2 PRODUCT FUNCTIONS ........ 10
2.3 USER CHARACTERISTICS ........ 10
2.4 GENERAL CONSTRAINTS ........ 10
2.5 ASSUMPTIONS AND DEPENDENCIES ........ 10

**3. SPECIFIC REQUIREMENTS ........ 10**

3.1 EXTERNAL INTERFACE REQUIREMENTS ........ 11
*3.1.1 User Interfaces ........ 11*
*3.1.2 Hardware Interfaces ........ 11*
*3.1.3 Software Interfaces ........ 11*
*3.1.4 Communications Interfaces ........ 11*
3.2 FUNCTIONAL REQUIREMENTS ........ 11
*3.2.1 <Functional Requirement or Feature #1> ........ 11*
*3.2.2 <Functional Requirement or Feature #2> ........ 11*
3.3 USE CASES ........ 11
*3.3.1 Use Case #1 ........ **Error! Bookmark not defined.***
*3.3.2 Use Case #2 ........ **Error! Bookmark not defined.***
3.4 CLASSES / OBJECTS ........ **ERROR! BOOKMARK NOT DEFINED.**
*3.4.1 <Class / Object #1> ........ **Error! Bookmark not defined.***
*3.4.2 <Class / Object #2> ........ **Error! Bookmark not defined.***
3.5 NON-FUNCTIONAL REQUIREMENTS ........ 11
*3.5.1 Performance ........ 11*
*3.5.2 Reliability ........ 11*
*3.5.3 Availability ........ 11*
*3.5.4 Security ........ 11*
*3.5.5 Maintainability ........ 11*
*3.5.6 Portability ........ 11*
3.6 INVERSE REQUIREMENTS ........ 11
3.7 DESIGN CONSTRAINTS ........ 12
3.8 LOGICAL DATABASE REQUIREMENTS ........ 12
3.9 OTHER REQUIREMENTS ........ 12

**4. ANALYSIS MODELS ........ 12**

4.1 SEQUENCE DIAGRAMS ........ 12
4.3 DATA FLOW DIAGRAMS (DFD) ........ 12
4.2 STATE-TRANSITION DIAGRAMS (STD) ........ 12

**5. CHANGE MANAGEMENT PROCESS ........ 12**

**A. APPENDICES ........ 12**

A.1 APPENDIX 1 ........ **ERROR! BOOKMARK NOT DEFINED.**
A.2 APPENDIX 2 ........ **ERROR! BOOKMARK NOT DEFINED.**

**1. Introduction**

*The introduction to the Software Requirement Specification (SRS) document should provide an overview of the complete SRS document. While writing this document please remember that this document should contain all of the information needed by a software engineer to adequately design and implement the software product described by the requirements listed in this document. (Note: the following subsection annotates are largely taken from the IEEE Guide to SRS).*

**1.1 Purpose**

What is the purpose of this SRS and the (intended) audience for which it is written.

**1.2 Scope**

This subsection should:

(1) Identify the software product(s) to be produced by name; for example, Host DBMS, Report Generator, etc

(2) Explain what the software product(s) will, and, if necessary, will not do

(3) Describe the application of the software being specified. As a portion of this, it should:

 (a) Describe all relevant benefits, objectives, and goals as precisely as possible. For example, to say that one goal is to provide effective reporting capabilities is not as good as saying parameter-driven, user-definable reports with a 2 h turnaround and on-line entry of user parameters.

 (b) Be consistent with similar statements in higher-level specifications (for example, the System Requirement Specification) , if they exist.What is the scope of this software product.

**1.3 Definitions, Acronyms, and Abbreviations**

This subsection should provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS. This information may be provided by reference to one or more appendixes in the SRS or by reference to other documents.

**1.4 References**

This subsection should:

(1) Provide a complete list of all documents referenced elsewhere in the SRS, or in a separate, specified document.

(2) Identify each document by title, report number - if applicable - date, and publishing organization.

(3) Specify the sources from which the references can be obtained.

This information may be provided by reference to an appendix or to another document.

**1.5 Overview**

This subsection should:

(1) Describe what the rest of the SRS contains

(2) Explain how the SRS is organized.

**2. General Description**

This section of the SRS should describe the general factors that affect 'the product and its requirements. It should be made clear that this section does not state specific requirements; it only makes those requirements easier to understand.

**2.1 Product Perspective**

*This subsection of the SRS puts the product into perspective with other related products or*
*projects. (See the IEEE Guide to SRS for more details).*

**2.2 Product Functions**

This subsection of the SRS should provide a summary of the functions that the software will perform.

**2.3 User Characteristics**

This subsection of the SRS should describe those general characteristics of the eventual users of the product that will affect the specific requirements. (See the IEEE Guide to SRS for more details).

**2.4 General Constraints**

*This subsection of the SRS should provide a general description of any other items that will*
*limit the developer's options for designing the system. (See the IEEE Guide to SRS for a partial list of possible general constraints).*

**2.5 Assumptions and Dependencies**

This subsection of the SRS should list each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software but are, rather, any changes to them that can affect the requirements in the SRS. For example, an assumption might be that a specific operating system will be available on the hardware designated for the software product. If, in fact, the operating system is not available, the SRS would then have to change accordingly.

**3. Specific Requirements**

This will be the largest and most important section of the SRS. The customer requirements will be embodied within Section 2, but this section will give the D-requirements that are used to guide the project's software design, implementation, and testing.

Each requirement in this section should be:

- Correct
- Traceable (both forward and backward to prior/future artifacts)
- Unambiguous
- Verifiable (i.e., testable)
- Prioritized (with respect to importance and/or stability)
- Complete
- Consistent
- Uniquely identifiable (usually via numbering like 3.4.5.6)

Attention should be paid to the carefuly organize the requirements presented in this section so that they may easily accessed and understood. Furthermore, this SRS is not the software design document, therefore one should avoid the tendency to over-constrain (and therefore design) the software project within this SRS.

**3.1 External Interface Requirements**

**3.1.1 User Interfaces**

**3.1.2 Hardware Interfaces**

**3.1.3 Software Interfaces**

**3.1.4 Communications Interfaces**

**3.2 Functional Requirements**

This section describes specific features of the software project. If desired, some requirements may be specified in the use-case format and listed in the Use Cases Section.

**3.2.1 <Functional Requirement or Feature #1>**

3.2.1.1 Introduction
3.2.1.2 Inputs
3.2.1.3 Processing
3.2.1.4 Outputs
3.2.1.5 Error Handling

**3.2.2 <Functional Requirement or Feature #2>**

…

**3.3 Use Cases**

**3.5 Non-Functional Requirements**

Non-functional requirements may exist for the following attributes. Often these requirements must be achieved at a system-wide level rather than at a unit level. State the requirements in the following sections in measurable terms (e.g., 95% of transaction shall be processed in less than a second, system downtime may not exceed 1 minute per day, > 30 day MTBF value, etc).

**3.5.1 Performance**

**3.5.2 Reliability**

**3.5.3 Availability**

**3.5.4 Security**

**3.5.5 Maintainability**

**3.5.6 Portability**

**3.6 Inverse Requirements**

State any *useful* inverse requirements.

**3.7 Design Constraints**
Specify design constrains imposed by other standards, company policies, hardware limitation, etc. that will impact this software project.

**3.8 Logical Database Requirements**
Will a database be used? If so, what logical requirements exist for data formats, storage capabilities, data retention, data integrity, etc.

**3.9 Other Requirements**
Catchall section for any additional requirements.

**4. Analysis Models**
List all analysis models used in developing specific requirements previously given in this SRS. Each model should include an introduction and a narrative description. Furthermore, each model should be traceable the SRS's requirements.

**4.1 Sequence Diagrams**
**4.3 Data Flow Diagrams (DFD)**
**4.2 State-Transition Diagrams (STD)**
**5. Change Management Process**
Identify and describe the process that will be used to update the SRS, as needed, when project scope or requirements change. Who can submit changes and by what means, and how will these changes be approved.

**A. Appendices**
Appendices may be used to provide additional (and hopefully helpful) information. If present, the SRS should explicitly state whether the information contained within an appendix is to be considered as a part of the SRS's overall set of requirements.

*Example Appendices could include (initial) conceptual documents for the software project, marketing materials, minutes of meetings with the customer(s), etc.*

b. Explain the types of feasibility in requirement engineering? Discuss the validity of using cost benefit analysis, especially in socially useful applications.

**Answer:**
A **feasibility study** is an evaluation and analysis of the potential of the proposed project which is based on extensive investigation and research to give full comfort to the decisions makers. Feasibility studies aim to objectively and rationally uncover the strengths and weaknesses of an existing business or proposed venture, opportunities and threats as presented by the environment, the resources required to carry through, and ultimately the prospects for success. In its simplest terms, the two criteria to judge feasibility are cost required and value to be attained.[ As such, a well-designed feasibility study should provide a historical background of the business or project, description of the product or service, accounting statements, details of the operations and management, marketing research and policies, financial data, legal requirements and tax obligations. Generally, feasibility studies precede technical development and project implementation.

The acronym TELOS refers to the five areas of feasibility - Technical, Economic, Legal, Operational, and Scheduling.

## Technology and system feasibility

The assessment is based on an outline design of system requirements, to determine whether the company has the technical expertise to handle completion of the project. When writing a feasibility report, the following should be taken to consideration:

- A brief description of the business to assess more possible factor/s which could affect the study
- The part of the business being examined
- The human and economic factor
- The possible solutions to the problems

At this level, the concern is whether the proposal is both *technically* and *legally* feasible (assuming moderate cost).

## Legal feasibility

Determines whether the proposed system conflicts with legal requirements, e.g. a data processing system must comply with the local Data Protection Acts..

## Operational feasibility

Operational feasibility is a measure of how well a proposed system solves the problems, and takes advantage of the opportunities identified during scope definition and how it satisfies the requirements identified in the requirements analysis phase of system development.

## Schedule feasibility

A project will fail if it takes too long to be completed before it is useful. Typically this means estimating how long the system will take to develop, and if it can be completed in a given time period using some methods like payback period. Schedule feasibility is a measure of how reasonable the project timetable is. Given our technical expertise, are the project deadlines reasonable? Some projects are initiated with specific deadlines. You need to determine whether the deadlines are mandatory or desirable.

The following is a list of steps that comprise a generic cost-benefit analysis.[5]

a. List alternative projects/programs.
b. List stakeholders.
c. Select measurement(s) and measure all cost/benefit elements.
d. Predict outcome of cost and benefits over relevant time period.
e. Convert all costs and benefits into a common currency.
f. Apply discount rate.
g. Calculate net present value of project options.
h. Perform sensitivity analysis.

i. Adopt recommended choice.

The value of a cost–benefit analysis depends on the accuracy of the individual cost and benefit estimates. Comparative studies indicate that such estimates are often flawed,

Causes of these inaccuracies include

a) Overreliance on data from past projects (often differing markedly in function or size and the skill levels of the team members)
b) Use of subjective impressions by assessment team members
c) Inappropriate use of heuristics to derive money cost of the intangible elements
d) Confirmation bias among project supporters (looking for reasons to proceed).

Interest groups may attempt to include or exclude significant costs from an analysis to influence the outcome.

In the case of the Ford Pinto (where, because of design flaws, the Pinto was liable to burst into flames in a rear-impact collision), the company's decision was not to issue a recall. Ford's cost–benefit analysis had estimated that based on the number of cars in use and the probable accident rate, deaths due to the design flaw would cost it about $49.5 million to settle wrongful death lawsuits versus recall costs of $137.5 million. Ford overlooked (or considered insignificant) the costs of the negative publicity that would result, which forced a recall *and* damaged sales.[35]

In health economics, some analysts think cost–benefit analysis can be an inadequate measure because willingness-to-pay methods of determining the value of human life can be influenced by income level. They support use of variants such as cost–utility analysis and quality-adjusted life year to analyze the effects of health policies.[36]

In environmental and occupational health regulation, it has been argued that if modern cost-benefit analyses had been applied prospectively to decisions such as whether to mandate the removal of lead from gasoline, build the Hoover Dam in the Grand Canyon, and regulate workers' exposure to vinyl chloride, these measures would not have been implemented even though they are considered to be highly successful in retrospect.[37] The Clean Air Act has been cited in retrospective studies as a case where benefits exceeded costs, but the knowledge of the benefits (attributable largely to the benefits of reducing particulate pollution) was not available until many years later.

c. Differentiate between a context model and a data model. How are they represented?

**Answer:**

A high-level data model in business or for any functional area is an abstract model that documents and organizes the business data for communication between functional and technical people. It is used to show the data needed and created by business processes.

A data model in software engineering is an abstract model that documents and organizes the business data for communication between team members and is used as a plan for developing applications, specifically how data are stored and accessed.

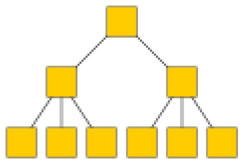
According to Hoberman (2009), "A data model is a way finding tool for both business and IT professionals, which uses a set of symbols and text to precisely explain a subset of real information to improve communication within the organization and thereby lead to a more flexible and stable application environment."

A data model explicitly determines the structure of data or *structured data*. Typical applications of data models include database models, design of information systems, and enabling exchange of data. Usually data models are specified in a data modeling language.

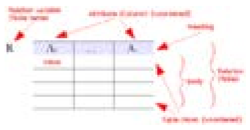
**Types of data models**

## Database model

A database model is a specification describing how a database is structured and used. Several such models have been suggested. Common models include:

- 

  Hierarchical model

- 

  Relational model

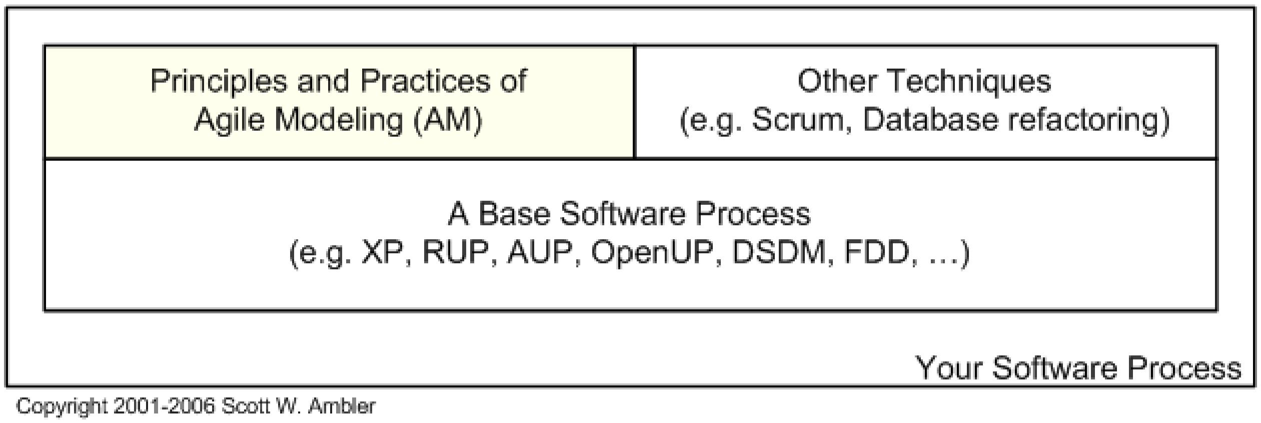
- **Flat model**: This may not strictly qualify as a data model. The flat (or table) model consists of a single, two-dimensional array of data elements, where all members of a given column are assumed to be similar values, and all members of a row are assumed to be related to one another.
- **Hierarchical model:** In this model data is organized into a tree-like structure, implying a single upward link in each record to describe the nesting, and a sort field to keep the records in a particular order in each same-level list.
- **Network model:** This model organizes data using two fundamental constructs, called records and sets. Records contain fields, and sets define one-to-many relationships between records: one owner, many members.
- **Relational model**: is a database model based on first-order predicate logic. Its core idea is to describe a database as a collection of predicates over a finite set of predicate variables, describing constraints on the possible values and combinations of values.

- Object-relational model: Similar to a relational database model, but objects, classes and inheritance are directly supported in database schemas and in the query language.
- Star schema is the simplest style of data warehouse schema. The star schema consists of a few "fact tables" (possibly only one, justifying the name) referencing any number of "dimension tables". The star schema is considered an important special case of the snowflake schema.

**Q.4** a. Give key characteristics of agile models. How is extreme programming an important tool in software engineering discipline? Explain.

**Answer:**

Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. As you see in Figure 1 AM is meant to be tailored into other, full-fledged methodologies such as XP or RUP, enabling you to develop a software process which truly meets your needs. Figure 1. AM enhances other software processes.


Principles and Practices of Agile Modeling (AM)
Other Techniques (e.g. Scrum, Database refactoring)
A Base Software Process (e.g. XP, RUP, AUP, OpenUP, DSDM, FDD, …)
Your Software Process
Copyright 2001-2006 Scott W. Ambler

The values of AM, adopting and extending those of eXtreme Programming v1, are communication, simplicity, feedback, courage, and humility. The keys to modeling success are to have effective communication between all project stakeholders, to strive to develop the simplest solution possible that meets all of your needs, to obtain feedback regarding your efforts often and early, to have the courage to make and stick to your decisions, and to have the humility to admit that you may not know everything, that others have value to add to your project efforts.

AM is based on a collection of principles, such as the importance of assuming simplicity when you are modeling and embracing change as you are working because requirements will change over time. You should recognize that incremental change of your system over time enables agility and that you should strive to obtain rapid feedback on your work to ensure that it accurately reflects the needs of your project stakeholders. You should model with a purpose, if you don't know why you are working on something or you don't know what the audience of the model/document actually requires then you shouldn't be working on it. Furthermore, you need multiple models in your intellectual toolkit to be effective. A critical concept is that models are not necessarily documents, a realization that enables you travel light by discarding most of your models once they have fulfilled their purpose. Agile modelers believe that content is more important than representation, that there are many ways you can model the same concept yet still get it right. To be an effective modeler you need to recognize that open and honest communication is often the best policy to follow to ensure effective teamwork. Finally, a focus on quality work is important because nobody likes to produce sloppy work and that local adaptation of AM to meet the exact needs of your environment is important.

To model in an agile manner you will apply AM's practices as appropriate. Fundamental practices include creating several models in parallel, applying the right artifact(s) for the situation, and iterating to another artifact to continue moving forward at a steady pace. Modeling in small increments, and not attempting to create the magical "all encompassing model" from your ivory tower, is also fundamental to your success as an agile modeler. Because models are only abstract representations of software, abstractions that may not be accurate, you should strive to prove it with code to show that your ideas actually work in practice and not just in theory Active stakeholder participation is critical to the success of your modeling efforts because your project stakeholders know what they want and can provide you with the feedback that you require. The principle of assume simplicity is a supported by the practices of creating simple content by focusing only on the aspects that you need to model and not attempting to creating a highly detailed model, depicting models simply via use of simple notations, and using the simplest tools to create your models. You travel light by single sourcing information, discarding temporary models and updating models only when it hurts. Communication is enabled by displaying models publicly, either on a wall or internal web site, through collective ownership of your project artifacts, through applying modeling standards, and by modeling with others. Your development efforts are greatly enhanced when you apply patterns gently. Because you often need to integrate with other systems, including legacy databases as well as web-based services, you will find that you need to formalize contract models with the owners of those systems. Read this article for a better understanding of how AM's practices fit together.

## AM and XP?

**AM should be tailored into an existing, full lifecycle methodology, in order to improve its approach to modeling. Because modeling is clearly a part of XP, see above, the potential exists for AM to add value to an XP project. This assumes of course that there is possible to tailor AM into XP, I believe it is and argue so below, and that you can do so without detracting from what currently exists within XP. In particular, XP's practices of refactoring and test-first development clearly do a very good job of filling in for two critical goals – promoting clean design and thinking through your design before writing code – that are typically associated with traditional modeling processes. My experience is that both refactoring and test-first**

**development are complementary to AM and arguably enablers of several AM practices, as I argue below.**

How should you approach modeling during development on an XP project? Beck suggests that you should apply the XP practice of Small Initial Investment and draw a few pictures at a time. He states that the XP strategy is that anyone can design with pictures all they want, but as soon as a question is raised that can be answered with code then the designers must turn to code for the answer. In other works you should then seek Rapid Feedback to discover whether your pictures are on target by following the AM practice Prove it With Code.

When should you consider modeling during development on an XP project? Whenever creating a model is more effective that writing code. In other words, follow the AM principle maximize stakeholder ROI and the AM practice Apply the Right Artifact(s).

How should you model? Follow AM's practice Use the Simplest Tools and prefer tools such as index cards, whiteboards, and Post It notes over more complicated CASE tools. Simple tools tend to promote interaction and communication, two factors that are critical to your success. Although XP favors the use of index cards to record user stories, CRC models, and story tasks there is nothing wrong with using a CASE tool as long as its use provides positive value to your effort.

XP developers need to recognize that you can model on an XP project, that modeling is in fact a part of XP already with its existing application of user stories and CRC cards. More importantly, XP developers must abandon any preconceived notions that they may have about modeling - that big modeling up front (BMUF) is the only approach to modeling, that models are permanent documents that must always be updated, that you need to use complex CASE tools to model, and that the UML defines the only models available to you - and approach modeling from a new perspective. One such perspective was presented in this article, that you can tailor Agile Modeling (AM) into a software process based on extreme Programming (XP) and still remain effective as software developers.

b. What is a formal specification in software engineering? Give its uses and limitations.

**Answer:**

**formal specifications** are mathematically based techniques whose purpose are to help with the implementation of systems and software. They are used to describe a system, to analyze its behavior, and to aid in its design by verifying key properties of interest through rigorous and effective reasoning tools.These specifications are *formal* in the sense that they have a syntax, their semantics fall within one domain, and they are able to be used to infer useful information. In each passing decade computer systems have become increasingly more powerful and as a result they have become more impactful to society. Because of this, better techniques are needed to assist in the design and implementation of reliable software. Established engineering disciplines use mathematical analysis as the foundation of creating and validating product design. Formal specifications are one such way to achieve this in software engineering. Though useful, formal methods did not really take over as the standard for software reliability as once predicted. Other methods such as testing are more commonly used to enhance code quality

Testing finds errors (or bugs) in the implementation. It is best to find these as early as possible because the farther along in a project a bug is found, the more costly it is to fix. The idea with formal specifications is to minimize the creation of such errors. This is done by reducing the ambiguity of informal system requirements. By creating a formal specification, the designers are forced to make a detailed system analysis early on in the project. This analysis will usually reveal errors or inconsistencies that exist in the informal system requirements.As a result the chance of subtle errors being introduced and going undetected in complex software systems is reduced. Finding and correcting these kinds of errors early in the design stage will help to prevent expensive fixes that may arise in the future.

Testing and QA contribute to more than 50% of the total development cost of some projects; through the use of formal specifications certain testing processes may be automated leading to better and more cost-effective testing.

**Uses**

Given such a specification, it is possible to use formal verification techniques to demonstrate that a system design is correct with respect to its specification. This allows incorrect system designs to be revised before any major investments have been made into an actual implementation. Another approach is to use provably correct refinement steps to transform a specification into a design, which is ultimately transformed into an implementation that is correct by construction.

It is important to note that a formal specification is *not* an implementation, but rather it may be used to develop an implementation. Formal specifications describe *what* a system should do, not *how* the system should do it.

A good specification must have some of the following attributes: adequate, internally consistent, unambiguous, complete, satisfied, minimal

A good specification will have:

- Constructability, manageability and evolvability
- Usability
- Communicability
- Powerful and efficient analysis

One of the main reasons there is interest in formal specifications is that they will provide an ability to perform proofs on software implementations.These proofs may be used to validate a specification, verify correctness of design, or to prove that a program satisfies a specification.

**Limitations**

A design (or implementation) cannot ever be declared "correct" on its own. It can only every be "correct with respect to a given specification". Whether the formal specification correctly describes the problem to be solved is a separate issue. It is also a difficult issue to address, since it ultimately concerns the problem constructing abstracted formal representations of an informal concrete problem domain, and such an abstraction step is not amenable to formal proof. However, it is possible to validate a specification by proving "challenge" theorems concerning properties that the specification is expected to exhibit. If correct, these theorems reinforce the specifier's understanding of the specification and its relationship with the underlying problem domain. If not, the specification probably needs to be changed to better reflect the domain understanding of those involved with producing (and implementing) the specification.Formal methods of software development are not widely used in industry. Most companies do not consider it cost-effective to apply them in their software development processes. This may be for a variety of reasons, some of which are:

- Time
  - High initial start up cost with low measurable returns
- Flexibility
  - A lot of software companies use agile methodologies that focus on flexibility. Doing a formal specification of the whole system up front is often perceived as being the opposite of flexible. However, there is some research into the benefits of using formal specifications with "agile" development
- Complexity
  - They require a high level of mathematical expertise and the analytical skills to understand and apply them effectively
  - A solution to this would be to develop tools and models that allow for these techniques to be implemented but hide the underlying mathematics
- Limited scope
  - They do not capture properties of interest for all stakeholders in the project
  - They do not do a good job of specifying user interfaces and user interaction
- Not cost-effective
  - This is not entirely true, by limiting their use to only core parts of critical systems they have shown to be cost-effective

c. Perform the requirements workflow for a library management system with respect to a rapid application model.

**Answer:**

Methodology is processes that focus on what information is collected and how to analyze it. In our Development and Connect multiple databases we will implement the RAD methodology. RAD stands for Rapid Application Development as long as product can be implemented rapidly with high quality. RAD produce the quick and dirty prototype application or software which satisfied the customer needs and requirements. In RAD the software developed in series of cycles which probably know time boxes. The main advantage of RAD is that customer gives the feedback on each completion of phase of software. Without completion of previous phase the next phase cannot be started. The expected outcomes of development and connect multiple databases is that it provides the online Library Management System in which user can get registration and search the different kind of books on base of different parameter such as BookID, Author Name etc as long as confirm the booking of book in advance. By connecting the more than two databases with each other, system will execute the efficient, reliable and reduce the database terrific as well as database performance will be increased.

**Q.5** a. Explain distributed object architectures with the help of an example. Write down advantages of the distributed object model.

**Answer: Page Number 299-300 of Text Book**

b. Briefly describe event-driven control models. Differentiate between following two event-driven control models: Broadcast models and Interrupt-driven models.

**Answer: Page Number 282 of Text Book**

**Q.6** a. Differentiate between function oriented design and object oriented design. Explain various stages of an object-oriented design.

**Answer:**

The differences have to be stated in the following manner:

1. Classes and objects are the building blocks of an object oriented design, whereas functions and procedures are the building blocks for a function oriented design
2. Objects are the entities that encapsulate some state and provide services to be used by a client which could be another object, a program or a user. Due to encapsulation preserving data integrity is easier in object oriented design.
3. An important property of objects is that this **state persists** in contrast to the data defined in a function or a procedure which is generally lost once the function stops being active.

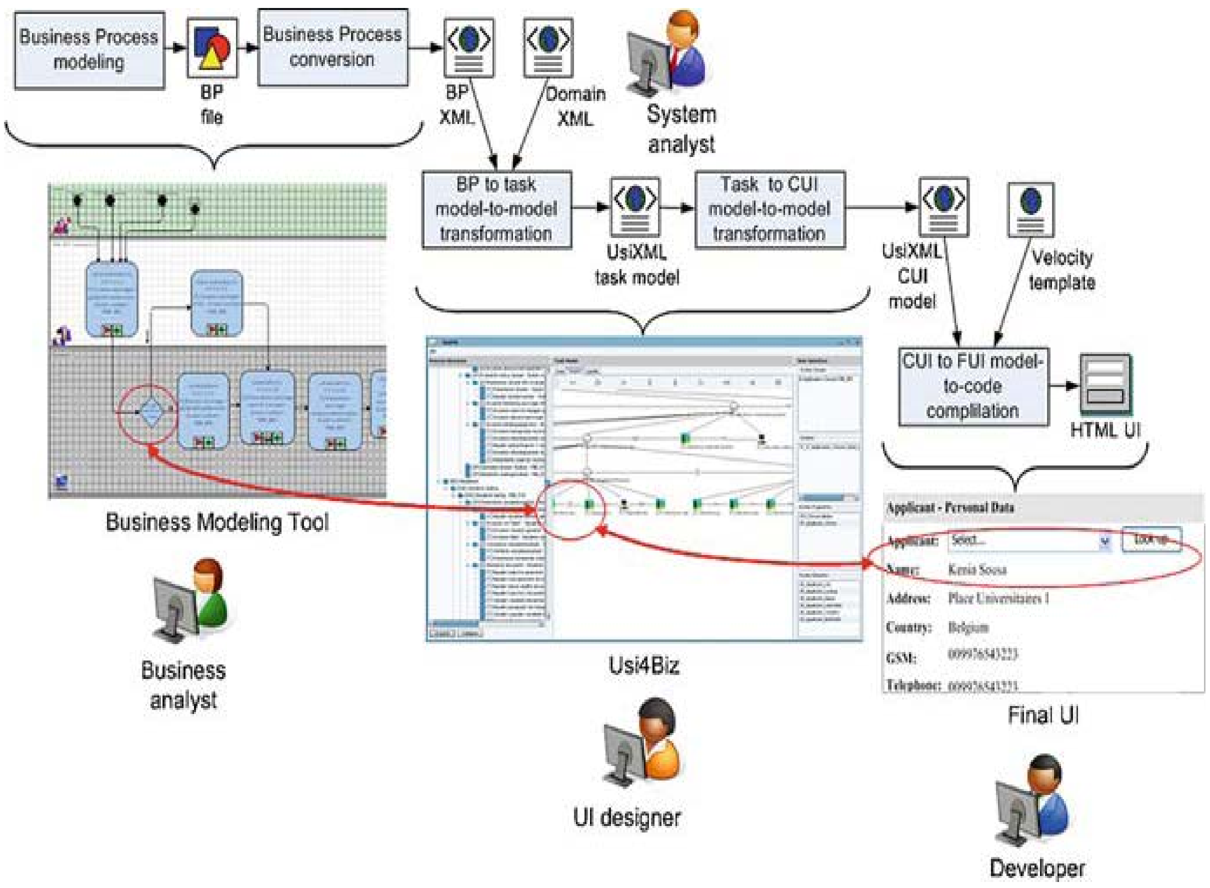
Differences have to be elaborated for the above points.

b. What do you mean by component composition? Explain different types of composition.

**Answer: Page Number 477 of Text Book**

**Q.7** a. Provide a diagram for user interface design life cycle. Explain its steps in brief.

**Answer:**



b. Write about various essential characteristics of dependable processes.

**Answer: Page Number 490 of Text Book**

c. Give a detailed description on fault tolerant architectures.

**Answer:**

Fault Tolerant Architectures:

Fault tolerance, being one of the four means for guaranteeing dependability, is intended to ensure the delivery of the correct services in the presence of active faults. It is implemented by error detection and subsequent system recovery. Error detection finds an erroneous system state. Following system recovery transforms the system state that contains one or more errors and (possibly) faults into a state without detected errors and faults (fault handling). Exceptions and exception handling provide a general framework for structuring the fault tolerance activities in a system, by focusing on the concept of exceptional/abnormal behaviour (as opposed to normal behaviour), exception handling enables specifying actions to be undertaken in the presence of abnormal events. While typical solutions focus on fault tolerance (and specifically, exception handling) during the design and implementation phases of the software life-cycle (e.g., Java and Windows NT exception handling), more recently the need for explicit exception handling solutions during the entire life cycle has been advocated by some researchers. Several solutions have been proposed for fault tolerance via exception handling at the software architecture and component levels. This tutorial describes how the two concepts of fault tolerance and software architectures have been integrated so far. It is structured in two parts (Overview on Fault Tolerance and Exception Handling, and Integrating Fault Tolerance into Software Architecture) and is based on a survey study on architecting fault tolerant systems where more than fifteen approaches have been analyzed and classified. The tutorial concludes identifying those issues that remain still open and require deeper investigation.

A lockstep fault-tolerant machine uses replicated elements operating in parallel. At any time, all the replications of each element should be in the same state. The same inputs are provided to each replication, and the same outputs are expected. The outputs of the replications are compared using a voting circuit. A machine with two replications of each element is termed Dual Modular Redundant (DMR). The voting circuit can then only detect a mismatch and recovery relies on other methods. A machine with three replications of each element is termed Triple Modular Redundancy (TMR). The voting circuit can determine which replication is in error when a two-to-one vote is observed. In this case, the voting circuit can output the correct result, and discard the erroneous version. After this, the internal state of the erroneous replication is assumed to be different from that of the other two, and the voting circuit can switch to a DMR mode. This model can be applied to any larger number of replications.

Lockstep fault tolerant machines are most easily made fully synchronous, with each gate of each replication making the same state transition on the same edge of the clock, and the clocks to the replications being exactly in phase. However, it is possible to build lockstep systems without this requirement.

Bringing the replications into synchrony requires making their internal stored states the same. They can be started from a fixed initial state, such as the reset state. Alternatively, the internal state of one replica can be copied to another replica.

One variant of DMR is pair-and-spare. Two replicated elements operate in lockstep as a pair, with a voting circuit that detects any mismatch between their operations and outputs a signal indicating that there is an error. Another pair operates exactly the same way. A final circuit selects the output of the pair that does not proclaim that it is in error. Pair-and-spare requires four replicas rather than the three of TMR, but has been used commercially.

**Q.8** a. "System testing can be considered as a pure black box test". Justify your answer. What are the drivers and stub modules in the context of integration and unit testing of software product? Why are they required?

**Answer:**

System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic As a rule, system testing takes, as its input, all of the "integrated" software components that have successfully passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called *assemblages*) or between any of the *assemblages* and the hardware. System testing is a more limited type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the

system as a whole. System testing is performed on the entire system in the context of a Functional Requirement Specification(s) (FRS) and/or a System Requirement Specification (SRS). System testing tests not only the design, but also the behaviour and even the believed expectations of the customer. It is also intended to test up to and beyond the bounds defined in the software/hardware requirements specification.
t is always a good idea to develop and test software in "pieces". But, it may seem impossible because it is hard to imagine how you can test one "piece" if the other "pieces" that it uses have not yet been developed (and vice versa). To solve this kindof diffcult problems we use stubs and drivers.

In white-box testing, we must run the code with predetermined input and check to make sure that the code produces predetermined outputs. Often testers write stubs and drivers for white-box testing.

**Driver for Testing:**

Driver is a the piece of code that passes test cases to another piece of code. Test Harness or a test driver is supporting code and data used to provide an environment for testing part of a system in isolation. It can be called as as a software module which is used to invoke a module under test and provide test inputs, control and, monitor execution, and report test results or most simplistically a line of code that calls a method and passes that method a value.

For example, if you wanted to move a fighter on the game, the driver code would be moveFighter(Fighter, LocationX, LocationY);

This driver code would likely be called from the main method. A white-box test case would execute this driver line of code and check "fighter.getPosition()" to make sure the player is now on the expected cell on the board.

**Stubs for Testing:**

A Stub is a dummy procedure, module or unit that stands in for an unfinished portion of a system.

Four basic types of Stubs for Top-Down Testing are:

1 Display a trace message
2 Display parameter value(s)
3 Return a value from a table
4 Return table value selected by parameter

A stub is a computer program which is used as a substitute for the body of a software module that is or will be defined elsewhere or a dummy component or object used to simulate the behavior of a real component until that component has been developed.

Ultimately, the dummy method would be completed with the proper program logic. However, developing the stub allows the programmer to call a method in the code being developed, even if the method does not yet have the desired behavior.

Stubs and drivers are often viewed as throwaway code. However, they do not have to be thrown away: Stubs can be "filled in" to form the actual method. Drivers can become automated test cases.

b. Under what different situations are Cost Estimation Models and COCOMO Models used? Give a brief description of COCOMO-II model.

**Answer:**

The COCOMO cost estimation model is used by thousands of software project managers, and is based on a study of hundreds of software projects. Unlike other cost estimation models, COCOMO is an open model, so all of the details are published, including:

- The underlying cost estimation equations
- Every assumption made in the model (e.g. "the project will enjoy good management")
- Every definition (e.g. the precise definition of the Product Design phase of a project)
- The costs included in an estimate are explicitly stated (e.g. project managers are included, secretaries aren't)

Because COCOMO is well defined, and because it doesn't rely upon proprietary estimation algorithms, Costar offers these advantages to its users:

- COCOMO estimates are more objective and repeatable than estimates made by methods relying on proprietary models
- COCOMO can be calibrated to reflect your software development environment, and to produce more accurate estimates

Costar is a faithful implementation of the COCOMO model that is easy to use on small projects, and yet powerful enough to plan and control large projects.

## Introduction to the COCOMO Model

The most fundamental calculation in the COCOMO model is the use of the Effort Equation to estimate the number of Person-Months required to develop a project. Most of the other COCOMO results, including the estimates for Requirements and Maintenance, are derived from this quantity.

## Source Lines of Code

The COCOMO calculations are based on your estimates of a project's size in Source Lines of Code (SLOC). SLOC is defined such that:

- Only Source lines that are DELIVERED as part of the product are included -- test drivers and other support software is excluded

- SOURCE lines are created by the project staff -- code created by applications generators is excluded
- One SLOC is one logical line of code
- Declarations are counted as SLOC
- Comments are not counted as SLOC

The original COCOMO 81 model was defined in terms of Delivered Source Instructions, which are very similar to SLOC. The major difference between DSI and SLOC is that a single Source Line of Code may be several physical lines. For example, an "if-then-else" statement would be counted as one SLOC, but might be counted as several DSI.

## The Scale Drivers

In the COCOMO II model, some of the most important factors contributing to a project's duration and cost are the Scale Drivers. You set each Scale Driver to describe your project; these Scale Drivers determine the exponent used in the Effort Equation.

The 5 Scale Drivers are:

- Precedentedness
- Development Flexibility
- Architecture / Risk Resolution
- Team Cohesion
- Process Maturity

Note that the Scale Drivers have replaced the Development Mode of COCOMO 81. The first two Scale Drivers, Precedentedness and Development Flexibility actually describe much the same influences that the original Development Mode did.

## Cost Drivers

COCOMO II has 17 cost drivers ï¿½ you assess your project, development environment, and team to set each cost driver. The cost drivers are multiplicative factors that determine the effort required to complete your software project. For example, if your project will develop software that controls an airplane's flight, you would set the Required Software Reliability (RELY) cost driver to Very High. That rating corresponds to an effort multiplier of 1.26, meaning that your project will require 26% more effort than a typical software project.

Click here to see which Cost Drivers are in which Costar models.

COCOMO II defines each of the cost drivers, and the Effort Multiplier associated with each rating. Check the Costar help for details about the definitions and how to set the cost drivers.

## COCOMO II Effort Equation

The COCOMO II model makes its estimates of required effort (measured in Person-Months ï¿½ PM) based primarily on your estimate of the software project's size (as measured in thousands of SLOC, KSLOC)):

Effort = 2.94 * EAF * $(KSLOC)^E$

Where

| | |
|---|---|
| EAF | Is the Effort Adjustment Factor derived from the Cost Drivers |
| E | Is an exponent derived from the five Scale Drivers |

As an example, a project with all Nominal Cost Drivers and Scale Drivers would have an EAF of 1.00 and exponent, E, of 1.0997. Assuming that the project is projected to consist of 8,000 source lines of code, COCOMO II estimates that 28.9 Person-Months of effort is required to complete it:

Effort = 2.94 * (1.0) * $(8)^{1.0997}$ = 28.9 Person-Months

## Effort Adjustment Factor

The Effort Adjustment Factor in the effort equation is simply the product of the effort multipliers corresponding to each of the cost drivers for your project.

For example, if your project is rated Very High for Complexity (effort multiplier of 1.34), and Low for Language & Tools Experience (effort multiplier of 1.09), and all of the other cost drivers are rated to be Nominal (effort multiplier of 1.00), the EAF is the product of 1.34 and 1.09.

Effort Adjustment Factor = EAF = 1.34 * 1.09 = 1.46

Effort = 2.94 * (1.46) * $(8)^{1.0997}$ = 42.3 Person-Months

## COCOMO II Schedule Equation

The COCOMO II schedule equation predicts the number of months required to complete your software project. The duration of a project is based on the effort predicted by the effort equation:

Duration = 3.67 * $(Effort)^{SE}$

Where

| | |
|---|---|
| Effort | Is the effort from the COCOMO II effort equation |
| SE | Is the schedule equation exponent derived from the five Scale Drivers |

Continuing the example, and substituting the exponent of 0.3179 that is calculated from the scale drivers, yields an estimate of just over a year, and an average staffing of between 3 and 4 people:

Duration = 3.67 * $(42.3)^{0.3179}$ = 12.1 months

Average staffing = (42.3 Person-Months) / (12.1 Months) = 3.5 people

## COCOMO II

COCOMO II is tuned to modern software life cycles. The original COCOMO model has been very successful, but it doesn't apply to newer software development practices as well as it does to traditional practices. COCOMO II targets the software projects of the 1990s and 2000s, and will continue to evolve over the next few years.

COCOMO II is really three different models:

- The Application Composition Model

  Suitable for projects built with modern GUI-builder tools. Based on new Object Points.

- The Early Design Model

  You can use this model to get rough estimates of a project's cost and duration before you've determined it's entire architecture. It uses a small set of new Cost Drivers, and new estimating equations. Based on Unadjusted Function Points or KSLOC.

- The Post-Architecture Model

  This is the most detailed COCOMO II model. You'll use it after you've developed your project's overall architecture. It has new cost drivers, new line counting rules, and new equations.

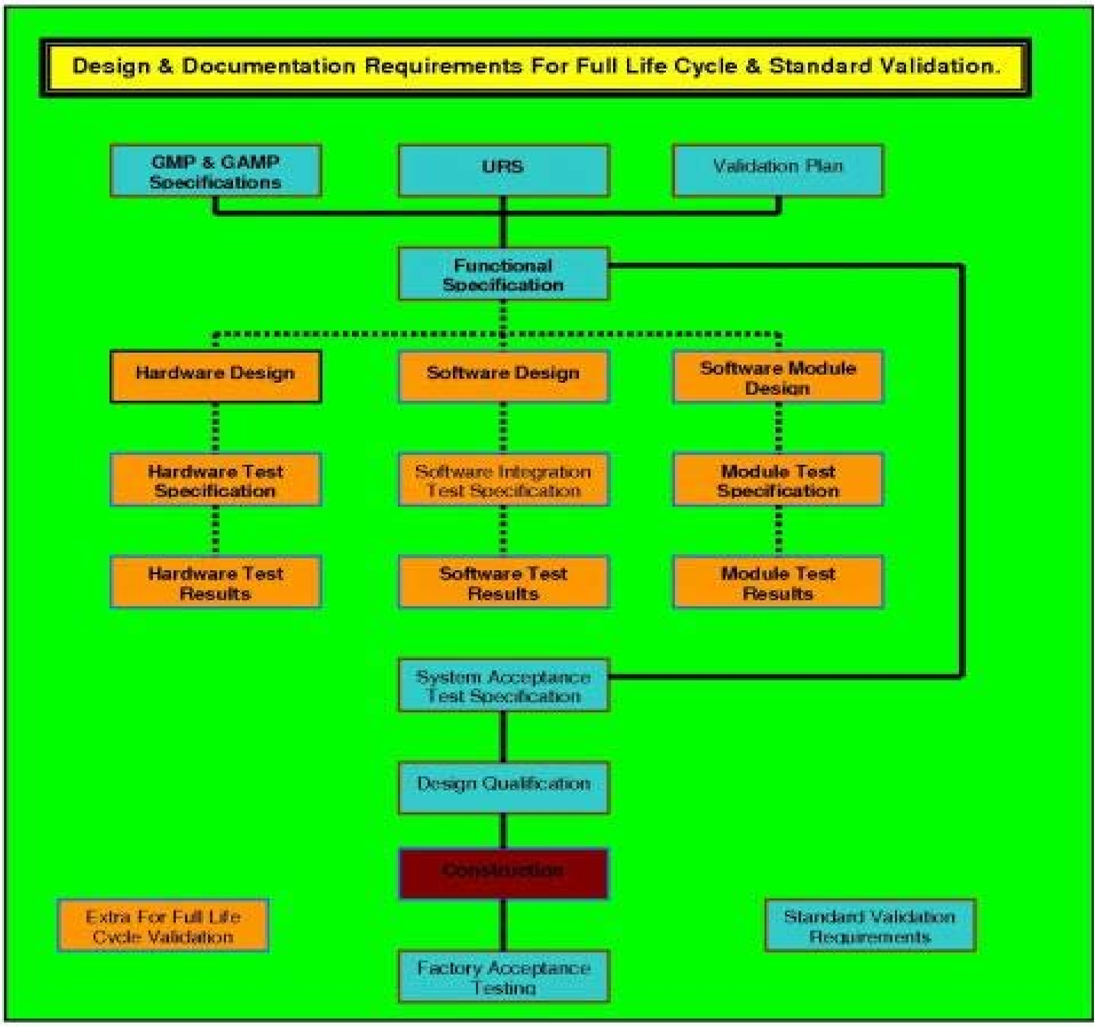
c. Explain, with the help of a diagram the process of software validation.

**Answer:**

Software Validation starts with a user requirement document (URS). When either management by policy; or an employee by cause; requests that a controlled process is instigated or an existing controlled process requires a serious modification (miner modification would be controlled under change control), a User Requirement Specification (URS) document is raised. Over the course of several meeting the URS is fleshed out to document all aspects of the requirement. It is important that the document is properly scoped in order that the procurement , installation, commissioning, validation, user training, maintenance, calibration and cleaning tasks are all investigated and defined adequately.

To scope and define an adequate validation procedure the URS has to be detailed sufficiently for various assessments to be made. The main assessment that concerns us with software validation documentation is the risk assessment. This assessment is only concerned with ensuring that the degree of validation that is proposed; is compliant with the regulatory requirements.

So at this early stage it is required to execute a risk assessment against the URS to see what category the software validation is going to be.


Design & Documentation Requirements For Full Life Cycle & Standard Validation.
GMP & GAMP Specifications
URS
Validation Plan
Functional Specification
Hardware Design
Software Design
Software Module Design
Hardware Test Specification
Software Integration Test Specification
Module Test Specification
Hardware Test Results
Software Test Results
Module Test Results
System Acceptance Test Specification
Design Qualification
Construction
Extra For Full Life Cycle Validation
Standard Validation Requirements
Factory Acceptance Testing

**Q.9** a. Discuss various key process areas of CMM at various maturity levels.

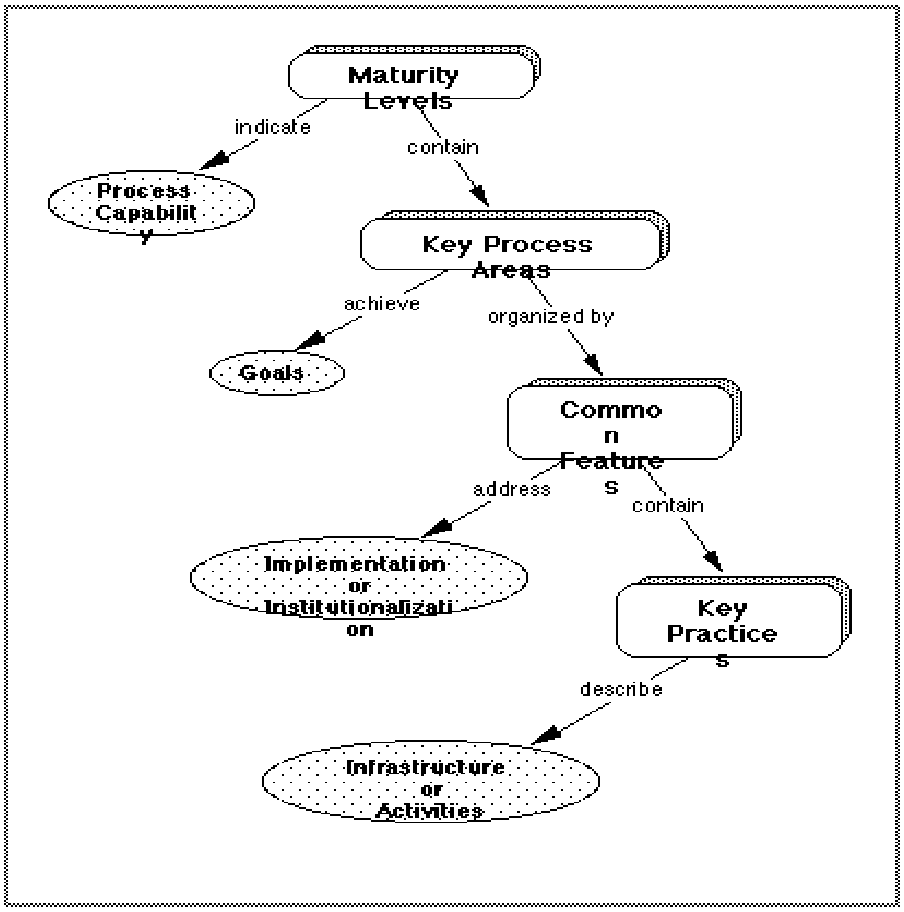
**Answer:**

The Capability Maturity Model for Software (CMM) is a framework that describes the key elements of an effective software process. The CMM describes an evolutionary improvement path from an ad hoc, immature process to a mature, disciplined process.

The CMM covers practices for planning, engineering, and managing software development and maintenance. When followed, these key practices improve the ability of organizations to meet goals for cost, schedule, functionality, and product quality.

The CMM establishes a yardstick against which it is possible to judge, in a repeatable way, the maturity of an organization's software process and compare it to the state of the practice of the industry. The CMM can also be used by an organization to plan improvements to its software process.

The CMM is composed of five maturity levels. With the exception of Level 1, each maturity level is composed of several key process areas. Each key process area is organized into five sections called common features. The common features specify the key practices that, when collectively addressed, accomplish the goals of the key process area. This structure of the CMM is illustrated in Figure



The components of the CMM include:

*Maturity levels*

A maturity level is a well-defined evolutionary plateau toward achieving a mature software process. The five maturity levels provide the top-level structure of the CMM.

*Process capability*

Software process capability describes the range of expected results that can be achieved by following a software process. The software process capability of an organization provides one means of predicting the most likely outcomes to be expected from the next software project the organization undertakes.

*Key process areas*

Each maturity level is composed of key process areas. Each key process area identifies a cluster of related activities that, when performed collectively, achieve a set of goals considered important for establishing process capability at that maturity level. The key process areas have been defined to reside at a single maturity level. For example, one of the key process areas for Level 2 is Software Project Planning.

*Goals*

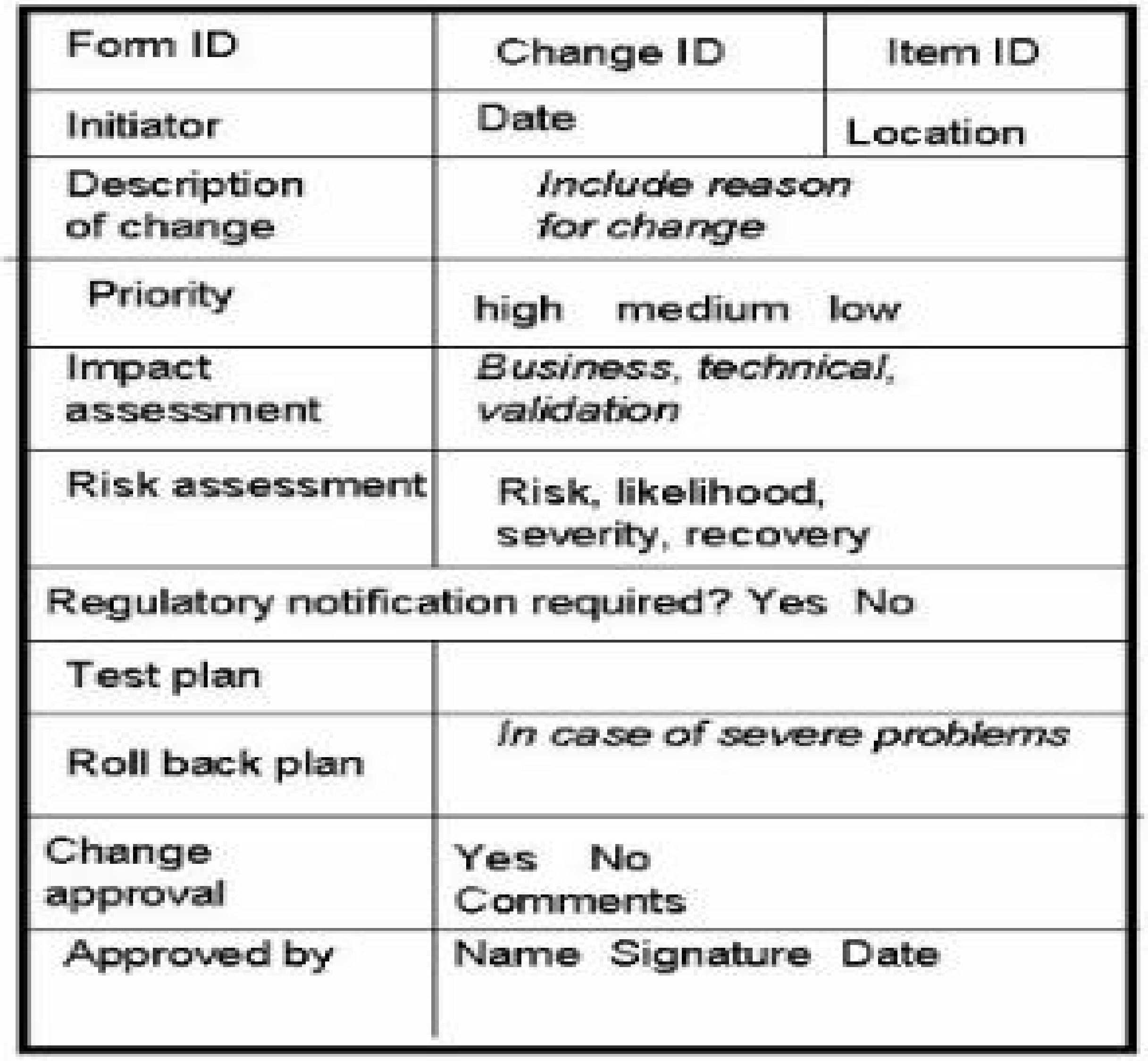
The goals summarize the key practices of a key process area and can be used to determine whether an organization or project has effectively implemented the key process area. The goals signify the scope, boundaries, and intent of each key process area.

b. Describe following static software product metrics:
   (i) Fan-in / Fan-out
   (ii) Length of code
   (iii) Length of identifiers

**Answer: Page Number 684 of Text Book**

c. Explain the following terms:
   (i) Change request form
   (ii) Change Control process

**Answer:**



| | | |
|---|---|---|
| Form ID | Change ID | Item ID |
| Initiator | Date | Location |
| Description of change | *Include reason for change* | |
| Priority | high medium low | |
| Impact assessment | *Business, technical, validation* | |
| Risk assessment | Risk, likelihood, severity, recovery | |
| Regulatory notification required? Yes No | | |
| Test plan | | |
| Roll back plan | *In case of severe problems* | |
| Change approval | Yes No Comments | |
| Approved by | Name Signature Date | |

There are many change control methods a project manager may use. Any good change control methodology will have four (4) components or subsystems. These subsystems are:

1. Change Request Form
2. Change Review and Evaluation
3. Change Priority and Classification
4. Change Approval

Figure 1 illustrates a typical change control process flow. Please note that there are two (2) separate reviews that occur during the change control process. These reviews are:

- First, to validate the change proposal is justified, and
- Second, to access the impact of the change on the project.

**Change Request Form**

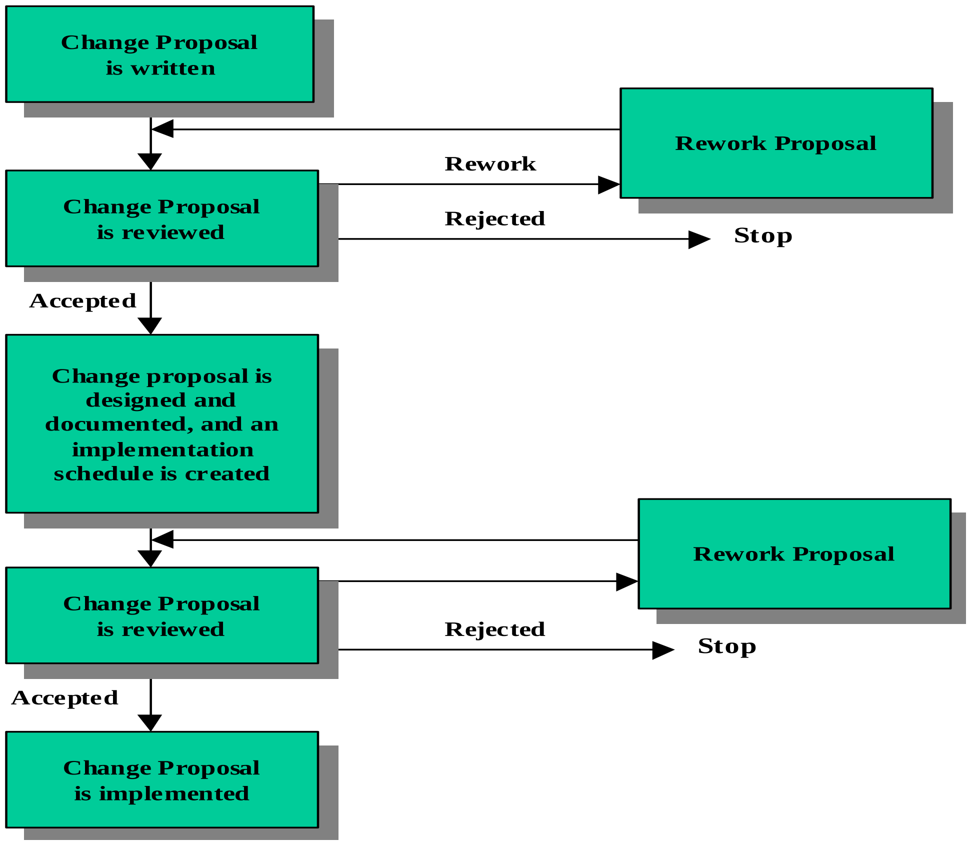
Evaluating the impact of a proposed change takes time. Therefore, it is important that change requests be well thought out before being submitted. The project manager is the filter point at which unnecessary and ill thought out change requests are rejected.

A project change request form should be created and approved prior to the project's startup. At a minimum, the change request form should contain the following information:

- Originator's name and phone number,
- Submission date,
- Description of the problem that the change addresses,
- Description of the change,
- Justification for the change,
- Date the change is needed (if applicable),
- Change category (type of change or reason for change)[1],
- Impact write-up addressing the affect the change has on project schedule and budget,
- Approval disposition (accepted or rejected boxes),
- Project Manager's signature and date,
- Project Sponsor's approval, signature, and date, and
- Date change is updated into the Change Request Log and updated into the project plan.

Under **all** circumstances, the project manager is the channel through which Change Requests are funneled. Any Change Request that does not have the project manager's signature is immediately returned to the submitted.

A **change review committee**, or board, must be established to evaluate all change requests. The number of participants on the committee, or board, is dependent upon the project size and complexity. On a small project, the project manager may review and approve change requests. On larger complex projects, a review committee or board may be more appropriate.The change review committee or board is made up of a cross selection of technical team members. Review committee members should be selected who have the expertise to truly assess the impact of the change request on the project's scope.


Change Proposal is written
Rework Proposal
Rework
Change Proposal is reviewed
Rejected
Stop
Accepted
Change proposal is designed and documented, and an implementation schedule is created
Rework Proposal
Change Proposal is reviewed
Rejected
Stop
Accepted
Change Proposal is implemented

<u>**Text Book**</u>

Software Engineering, Ian Sommerville, 7th edition, Pearson Education, 2004