Delhi Technological University

SOFTWARE ENGINEERING

(MC 310)

Submitted to:	Submitted by:
Mr. Jamkhongam Touthang	Sachin Duhan
Assistant Professor, DTU	2k17/MC/87

Q1. Explain how the universal use of the Web has changed software systems.

The universal use of web has transformed software systems and software systems engineering. Software engineering has proven to be a continuous development process. In the beginning, the web had very few certain effects on software systems, unlike we have it today.

These days, the high use of web based software development has highly affected the software industry. The points to show this are as given below:

1. In place of monolithic development most of the applications are developed for web users. Web makes faster delivery of the software systems to people anywhere at any time. Web has affected software systems in many terms of delivery, security of the software and advertisement of the new System.
2. The updating and maintenance of web based software is easy. Changes made at one place will effect globally.
3. Software is developed in parts. One part is developed at one place and second is developed at any other place. By using web it becomes easy to collect all parts to make a working application.
4. Popular technologies like HTML5 and AJAX are mostly used for web software development. These are web based technologies that a vast majority of people interact with on a daily basis.

Q2. Suggest why it is important to make a distinction between developing the user requirements and developing system requirements in the requirements engineering process.

There is a fundamental difference between the user and the system requirements that mean they should be considered separately.

1. The user requirements are intended to describe the system's functions and features from a user perspective and it is essential that users understand these requirements. They should be expressed in natural language and may not be expressed in great detail, to allow some implementation flexibility. The people involved in the process must be able to understand the user's environment and application domain.

2. The system requirements are much more detailed than the user requirements and are intended to be a precise specification of the system that may be part of a system contract. They may also be used in situations where development is outsourced and the development team need a complete specification of what should be developed. The system requirements are developed after user requirements have been established.

Q3. Explain why incremental development is the most effective approach for developing business software systems. Why is this model less appropriate for real-time systems engineering?

Incremental model is the most effective model in designing a software system. Below are the reasons justifying it.

1. The software system will be delivered quickly to the user.
2. The cost for each iteration is very less.
3. The testing efforts for each iteration will be less because only few additional functions need to be added in each iteration.
4. The final product produced will be perfect without any defects or issues since we get feedback of the customer during each iteration and developers will resolve these defects/issues.
5. This model allows user to modify/change the requirements according to his convenience at any phase of iteration.

This model is less appropriate to the real-time systems. The following are the reasons justifying it.

1. The overall cost will be more compared to the waterfall model. Same resources may not be available for the next iterations. Hence, new resources should be trained again for working on the application.
2. Concerns regarding the system architecture may arise as all the requirements are not gathered for the software lifecycle at the initial phase.

Q4. Suggest how an engineer responsible for drawing up a system requirements specification might keep track of the relationships between functional and non-functional requirements.

An Engineer would have to make a Systems Requirements Document. An Engineer could provide documents for each functional and non-functional requirement. The Engineer should use natural language for non-functional requirements and structured language for functional requirements. The functional requirements are more for developer’s eyes and to use.

The non-functional requirements are what user wants and what they except to get out of the software being developed. The engineer would also have to make sure that the non-functional requirements don’t conflict with the functional requirements.

The engineer might make a list or draw a graph of some sort linking each functional requirement to one of more non-functional requirements necessary to implement the functional requirement, or vice versa. For example, if a system involved user logins and sessions, the engineer might draw a line between the functional requirement "A user shall be able to login to the system by entering his/her username and password" and the non-functional requirements "A particular user session should not last more than five hours" and "User password should be reset every 150 days to ensure security," to indicate that the two system requirements will have a direct effect on the user requirement.

Q5. Explain all the levels of COCOMO model. Assume that the size of an organic software product has been estimated to be 32,000 lines of code. Determine the effort required to develop the software product and the nominal development time.

COCOMO (Constructive Cost Estimation Model) was proposed by Boehm [1981]. According to Boehm, software cost estimation should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Complete COCOMO. 9 • Basic COCOMO Model The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

Effort = a 1 x (KLOC) a 2 PM

Tdev = b 1 x (Effort) b 2 Months Where • KLOC is the estimated size of the software product expressed in Kilo Lines of Code,

a 1 , a 2 , b 1 , b 2 are constants for each category of software products.

Tdev is the estimated time to develop the software, expressed in months

Effort is the total effort required to develop the software product, expressed in person months (PMs). The effort estimation is expressed in units of person-months (PM). It is the area under the person-month plot as shown in the figure below. It should be carefully noted that an effort of 100 PM does not imply that 100 persons should work for 1 month nor does it imply that 1 person should be employed for 100 months, but it denotes the area under the person-month curve.

Irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC. The values of a 1 , a 2 , b 1 , b 2 for different categories of products (i.e. organic, semidetached, and embedded) as given by Boehm [1981] are summarized below. He derived the above expressions by examining historical data collected from a large number of actual projects. 10 Estimation of development effort.

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic: Effort = 2.4(KLOC) 1.05 PM

Semi-detached: Effort = 3.0(KLOC) 1.12 PM

Embedded: Effort = 3.6(KLOC) 1.20 PM

Estimation of development time For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : Tdev = 2.5(Effort) 0.38 Months

Semi-detached : Tdev = 2.5(Effort) 0.35 Months

Embedded Tdev = 2.5(Effort) 0.32 Months

Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. The figure below shows a plot of estimated effort versus product size. From the figure below, we can observe that the effort is somewhat superlinear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.

From the basic COCOMO estimation formula for organic software:

Effort = 2.4*(32$^{)1.05}$ = 91 PM

Nominal development time = 2.5*(91)*0.38 = 14 months

Q6. Explain why testing can only detect the presence of errors, not their absence.

Testing can detect only the presence of errors, not their absence because the **main goal** of the testing to observe the software behavior to meet its requirement expectation. Testing is a set of activities where the tester(s) try to make the software behave anomalous in order to detect a defect or anomaly to be later fix.

Testing demonstrates to the developer that the software fulfils its requirements and it is a way to find out if it behaves in an incorrect, undesirable or different form from the specifications.

Testing is a part of broader process of software verification and validation. It consists of a set of activities, where the testers try to make the software behave anomalous in order to detect error or anomaly to be later fixed.

Testing cannot detect the absence of errors as effects on the system due to the environment may always be unanticipated and even random. Anticipated scenarios may be tested for absence of errors but it can never be exhaustive. Testing, hence, can only detect presence of errors and not their absence.

Q7. Assume that the initial failure intensity is 20 failures/CPU hr. The failure intensity decay parameter is 0.02/failure. We have experienced 100 failures up to this time.

(i) Determine the current failure intensity.

(ii) Find the decrement of failure intensity per failure.

(iii) Calculate the failures experienced and failure intensity after 20 and 100 CPU hrs. of execution.

(iv) Compute additional failures and additional execution time required to reach the failure intensity objective of 2 failures/CPU hr.

Use Logarithmic Poisson execution time model of software reliability for the calculations.

μ =100 failures

θ = 02.0 /failures

λ =20 failures/CPU hr

(i) **Current failure intensity**: $\lambda(\mu) = \lambda_0 \exp(-\lambda\,\theta) = 20 \exp(-0.02 \times 100)$

$= 2.7$ failures/CPU hr

(ii) **Decrement of failure intensity per failure can be calculated as:**

$(d\lambda)/(d\mu) = -\theta\lambda = -.02 \times 2.7 = -.054$/CPU hr

(iii)(a) Failures experienced & failure intensity after **20 CPU hr:**

$\mu(\tau)= (1/\theta)\mathrm{Ln}(\lambda_0\theta\tau+1) = \mathrm{Ln}(20*0.02*20+1) = 109$ failures

$\lambda(\tau)= \lambda_0/(\lambda_0\theta\tau+1)= (20)/(20*0.02*20+1)= 2.22$ failures/CPU hr

(b) Failures experienced & failure intensity after **100 CPU hr:**

$\mu(\tau)= (1/\theta)\mathrm{Ln}(\lambda_0\theta\tau+1) = \mathrm{Ln}(20*0.02*100+1) = 186$ failures

$\lambda(\tau)= \lambda_0/(\lambda_0\theta\tau+1)= (20)/(20*0.02*100+1)= 0.4878$ failures/CPU hr

(iv) Additional failures and additional execution time required to reach the failure intensity objective of 2 failures/CPU hr

$$\Delta\mu = \frac{1}{\theta}Ln\frac{\lambda_P}{\lambda_F} = \frac{1}{0.02}Ln\left(\frac{2.7}{2}\right) = 15 \text{ failures}$$

$$\Delta\tau = \frac{1}{\theta}\left[\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right] = \frac{1}{0.02}\left[\frac{1}{2} - \frac{1}{2.7}\right] = 6.5 \text{ CPU}$$

Q8. Explain why a software system that is used a real-world environment must change or become progressively less useful.

About concerning system change, Lehman and Belady introduce set of laws. In this "Continuing change" is one.

Law: A system that is used in a real-world environment necessarily must change or become progressively less useful in that environment.

According to this, it states that system maintenance is an inevitable process. As the system's environment changes, new requirements emerge and the system must be modified. When the modified system is re-introduced into the environment, this promotes more environmental changes so the evolution process recycles.

If the number of users may increases, then the burden of the system requiring is expanding to its hardware capability for handle several connections.

The business model of the company may change so the system become obsolete and need for a change to cope its requirements.

The law in the particular country may impose a particular standard to conform legal usable software.

Q9. Explain why the environment in which a computer-based system is installed may have unanticipated effects on the system that lead to system failure. Illustrate your answer with an example.

To assess a software system from a technical perspective, we need to consider both the application system itself and the environment in which the system operates. The environment includes the hardware and all associated support software (compilers, development environments, etc.) that are required to maintain the system. The environment is important because many system changes result from changes to the environment, such as upgrades to the hardware or operating system. Other systems in the system's environment can have unanticipated effects because they have relationships with the system over and above whatever formal relationships (e.g. data exchange) are defined in the system specification.

For example, the system may share an electrical power supply and air conditioning unit, they may be located in the same room (so if there is a fire in one system then the other will be affected) etc.

Q10. Explain why there is a need for risk assessment to be a continuing process from the early stages of requirements engineering through to the operational use of a system.

Dependability and security requirements can be thought of as protection requirements. These specify how a system should protect itself from internal faults, stop system failures causing damage to its environment, stop accidents or attacks from the system's environment damaging the system, and facilitate recovery in the event of failure. To discover these protection requirements, we need to understand the risks to the system and its environment.

A general risk-driven specification process involves understanding the risks faced by the system, discovering their root causes, and generating requirements to manage these risks. The stages in this process are:

1. Risk identification : Potential risks to the system are identified. These are dependent on the environment in which the system is to be used. Risks may arise from interactions between the system and rare conditions in its operating environment.
2. Risk analysis and classification : Each risk is considered separately. Those that are potentially serious and not implausible are selected for further analysis. At this stage, risks may be eliminated because they are unlikely to arise or because they cannot be detected by the software (e.g., an allergic reaction to the sensor in the insulin pump system).
3. Risk decomposition : Each risk is analyzed to discover potential root causes of that risk. Root causes are the reasons why a system may fail. They may be software or hardware errors or inherent vulnerabilities that result from system design decisions.
4. Risk reduction : Proposals for ways in which the identified risks may be reduced or eliminated are made. These contribute to the system dependability requirements that define the defences against the risk and how the risk will be managed.

As different types of risks are presented at different points of Software Life Cycle, risk analysis may be carried out in phases for large projects :

1. Preliminary risk analysis, where major risks from the system's environment are identified. These are independent from the technology used for system development. The aim of preliminary risk analysis is to develop an initial set of security and dependability requirements for the system.

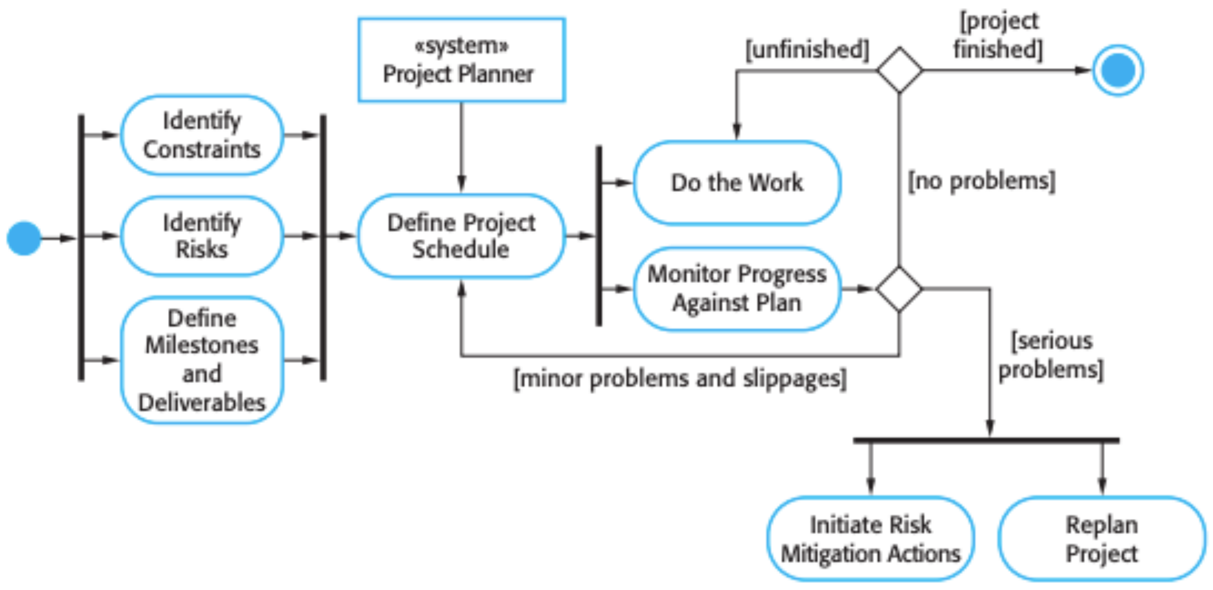
2. Life-cycle risk analysis, which takes place during system development and which is mostly concerned with risks that arise from system design decisions. Different technologies and system architectures have their own associated risks. At this stage, you should extend the requirements to protect against these risks.
3. Operational risk analysis, which is concerned with the system user interface and risks from operator errors. Again, once decisions have been made on the user interface design, further protection requirements may have to be added.

Q11. Explain the steps involved in project planning. Discuss the various factors that affect a project plan.

After the finalization of SRS, we would like to estimate size, cost and development time of the project. Also, in many cases, customer may like to know the cost and development time even prior to finalization of the SRS.

In order to conduct a successful software project, we must understand:

- Scope of work to be done : it must be clear what is encompassed by the project
- The risk to be incurred : risk assessment is essential and determines, often, the extent and utility of a product's life
- The resources required : infrastructure and other requirements should be included in the plan
- The task to be accomplished : the answers to "What needs to be delivered to the client?", "What is the end product of the process?" must be clear
- The cost to be expended : cost estimation is critical in project planning, and must be carried out with great care
- The schedule to be followed: following the schedule may be critical in many cases, hence the plan should consider and stick to the schedule.



Some factors affecting project planning are as follows:

1. Deadline
   Deadline is one of the key aspects that determine how a project is managed. Missing a deadline creates a bad impression for the team. However, completing a project on deadline does not mean that quality may be compromised. We have to be both alert about time and have a keen eye on quality. If the project has narrow deadlines with strict clients or stakeholders, project manager should be alert to all possible hindrances from before and take appropriate precautions, so that on-time delivery of quality products or services can be ensured.
2. Budget
   The budget determines the time and resources allocated for the project, hence being a critical factor. Quality needs to be ensured irrespective of the allocated time or resources, which may pressurise the project manager, and leads to different scenarios to handle.
3. Stakeholders
   Techniques of managing projects vary depending upon the kind of stakeholders for the projects. In case a project has multiple stakeholders from different backgrounds, there is a possibility of disagreement between them. In such cases, project management becomes extremely challenging as unhappy stakeholders and clients cannot be afforded.
4. Project Members
   Project management techniques are also determined by the challenges faced by a project manager which, in turn, depends on the kind of team he or she is handling. If the team consists of members with diverse backgrounds and skills, a gap in terms of team spirit may exist. This obviously impacts work. Therefore, a project manager should apply techniques to bring the team close.
5. Demand
   Demand is another key factor that influences project management techniques. Demand itself depends on a few factors such as type of products or services, usability, etc.
6. Supply
   In order to meet the demand within a stipulated date and time (deadline), supply of resources is necessary. A project manager needs to ensure that supply is adequate, so that deadline is not compromised for want of resources

Q12. Explain the SEI Capability Maturity Model (CMM).

The Capability Maturity Model (CMM) is a methodology used to develop and refine an organization's software development process. The model describes a five-level evolutionary path of increasingly organized and systematically more mature processes. CMM was developed and is promoted by the Software Engineering Institute (SEI).

Characteristics of Capability Maturity Model (CMM):

- It is not a software process model. It is a framework which is used to analyse the approach and techniques followed by any organization to develop a software product.
- It also provides guidelines to further enhance the maturity of those software products.
- It is based on profound feedback and development practices adopted by the most successful organizations worldwide.
- The higher the level, the better the software development process, hence reaching each level is an expensive and time-consuming process.
- Each level of maturity shows a process capability level. All the levels except level-1 are further described by Key Process Areas (KPA's).

CMM's Five Maturity Levels of Software Processes are as follows:

1. **Maturity Level 1**: At the initial level, processes are disorganized, even chaotic. Success is likely to depend on individual efforts, and is not considered to be repeatable, because processes would not be sufficiently defined and documented to allow them to be replicated. Characterization: Adhoc Process

2. **Maturity Level 2**: At the repeatable level, basic project management techniques are established, and successes could be repeated, because the requisite processes would have been made established, defined, and documented. Characterization: Basic Project Management

3. **Maturity Level 3**: At the defined level, an organization has developed its own standard software process through greater attention to documentation, standardization, and integration. Characterization: Process Definition

4. **Maturity Level 4**: At the managed level, an organization monitors and controls its own processes through data collection and analysis. At this level, organization set a quantitative quality goal for both software process and software maintenance. At this maturity level, the performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable. Characterization: Process Measurement

5. **Maturity Level 5**: At the optimizing level, processes are constantly being improved through monitoring feedback from current processes and introducing innovative processes to better serve the organization's particular needs. Characterization: Process Control

Q13. The best programmers do not always make the best software managers." Explain.

Management activities such as proposal writing, project planning and personnel selection require a set of skills including presentation and communication skills, organisational skills and the ability to communicate with other project team members. Programming skills are distinct from these (indeed, it is a common criticism of programmers that they lack human communication skills) so it does not follow that good programmers can re-orient their abilities to be good managers.

Programmer	Manager
They think as Geeks and other people might not be happy with using their software Lack of interface with optimal functionality is a good sign	The kind of people that add niceness Good people skills are necessary for optimal results. Sales oriented people
They know there may exist bugs somewhere and can't sell the software as perfect	Manager and sales- oriented people neither know nor care about perfection
Tasks and mindset include 1. Focus 2. Technical balance of speed, maintainability, reliability 3. Deliver functionality 4. Intellectual 5. Coders	Tasks and mindset include 1. Multitask 2. Project balance of budget, resources, schedule etc 3. Interface between Company and Team 4. Skilled with people 5. Communicators

There is little or no similarity between what you need to think and do to become a great programmer and what's required to be a great manager.

Q14. Explain why the intangibility of software systems poses special problems for software project management.

A system in which its services are not visible or cannot be felt physically is called an intangible system. Intangibility in software system may lead to various major problems in planning, estimating, scheduling and budgeting of the entire product under development for the project management team. Impalpability in programming framework may prompt different serious issues in arranging, evaluating, planning and planning of the whole item a work in progress for the undertaking supervisory crew. Programming advancement which is an extremely protracted procedure may should be refreshed or re-built at different stages, contingent upon the client necessities and desires which are absolutely capricious and shifting.

This issue emerges as the vast majority of the occasions the clients may not know or they don't know pretty much the entirety of their needs and desires from the framework in the early period of formative procedure. They likewise need to ensure that the task meets every one of these limitations and changing prerequisites of its clients on schedule and the advancement procedure is done in wanted way and inside the evaluated or cited spending plan for the equivalent. As building up the elusive programming framework causes all the above complexities in the undertaking director's activity during SDLC, it presents unique issues for programming venture the executives.

Programming, a work in progress, is thought of an Intangible Asset. Since one cannot see it, contact it, remain on it or eat it, one needs to depend on the group to create convenient and precise documentation expected to follow progress and profitability. The immaterial idea of programming messes up the executives in arranging, assessing, planning and planning for bookkeeping purposes.

In the event that there is a product framework that is immaterial in nature, at that point we need to confront different issues. For example, survey programming items and clients, who are unsatisfied with the items they get, clients' failure to realize all the necessities ahead of time. The explanation behind these issues is that the primary result referenced unmistakably gets from a psychological factor i.e. testing programming items isn't a simple or insignificant errand.

The primary explanation is that product is a novel immaterial element with which the human brain has not been formed to adapt during its advancement. Along these lines, our capacity to manage such impalpable relics is restricted.

Programming isn't just an immaterial item; it is additionally executed on a PC.

This implies the created antiquity isn't the one that is really utilized by its clients.

Q15. What is the risk? Identify six possible risks that could arise in software projects. Discuss how you would manage those risks at different phases.

Risk is an expectation of loss, a potential problem that may or may not occur in the future. It is generally caused due to a lack of information, control, or time. A possibility of suffering from loss in the software development process is called a software risk. Loss can be anything, an increase in production cost, development of poor quality software, not being able to complete the project on time. Software risk exists because the future is uncertain and there are many known and unknown things that cannot be incorporated in the project plan. A software risk can be of two types

(a) internal risks that are within the control of the project manager

(b) external risks that are beyond the control of the project manager.

Risk management is carried out to:

1. Identify the risk
2. Reduce the impact of risk
3. Reduce the probability or likelihood of risk
4. Risk monitoring

A project manager has to deal with risks arising from three possible cases:

1. **Known knowns** are software risks that are actually facts known to the team as well as to the entire project. For example not having enough developers can delay the project delivery. Such risks are described and included in the Project Management Plan.
2. **Known unknowns** are risks that the project team is aware of but it is unknown that such risk exists in the project or not. For example, if the communication with the client is not of a good level then it is not possible to capture the requirement properly. This is a fact known to the project team however whether the client has communicated all the information properly or not is unknown to the project.
3. **Unknown Unknowns** are the kind of risks about which the organization has no idea. Such risks are generally related to technology such as working with technologies or tools that you have no idea about because your client wants you to work that way suddenly exposes you to absolutely unknown risks.

Software risk management is all about risk quantification of risk. This includes:

1. Giving a precise description of risk event that can occur in the project
2. Defining risk probability that would explain what are the chances for that risk to occur
3. Defining How much loss a particular risk can cause
4. Defining the liability potential of risk

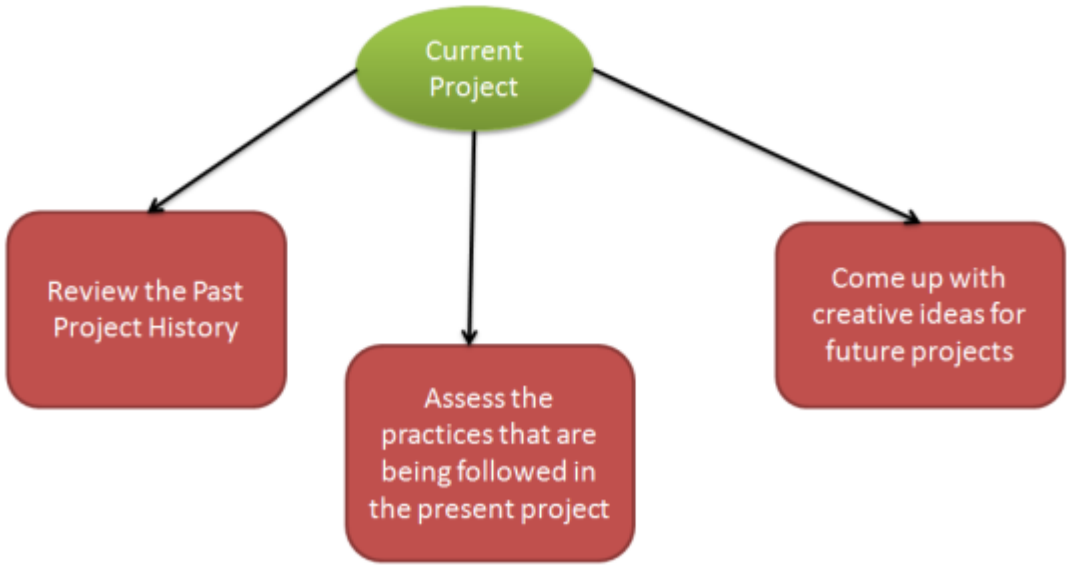
Risk Management comprises of following processes:

1. Software Risk Identification
2. Software Risk Analysis
3. Software Risk Planning
4. Software Risk Monitoring

These Processes are defined below.

Software Risk Identification

In order to identify the risks that your project may be subjected to, it is important to first study the problems faced by previous projects. Study the project plan properly and check for all the possible areas that are vulnerable to some of the other types of risks. The best way of analyzing a project plan is by converting it to a flowchart and examine all essential areas. It is important to conduct a few brainstorming sessions to identify the known unknowns that can affect the project. Any decision taken related to technical, operational, political, legal, social, internal or external factors should be evaluated properly.


Current Project
Review the Past Project History
Assess the practices that are being followed in the present project
Come up with creative ideas for future projects

In this phase of Risk management, you have to define processes that are important for risk identification. All the details of the risk such as unique Id, the date on which it was identified, description, and so on should be clearly mentioned.

Software Risk Analysis

Software Risk analysis is a very important aspect of risk management. In this phase, the risk is identified and then categorized. After the categorization of risk, the level, likelihood (percentage), and impact of the risk are analyzed. The likelihood is defined in percentage after examining what are the chances of risk to occur due to various technical conditions. These technical conditions can be:

1. The complexity of the technology
2. Technical knowledge possessed by the testing team
3. Conflicts within the team
4. Teams being distributed over a large geographical area
5. Usage of poor quality testing tools

With impact we mean the consequence of a risk in case it happens. It is important to know about the impact because it is necessary to know how a business can get affected:

1. What will be the loss to the customer
2. How would the business suffer
3. Loss of reputation or harm to society
4. Monetary losses
5. Legal actions against the company
6. Cancellation of business license

Level of risk is identified with the help of:

Qualitative Risk Analysis: Here you define risk as:

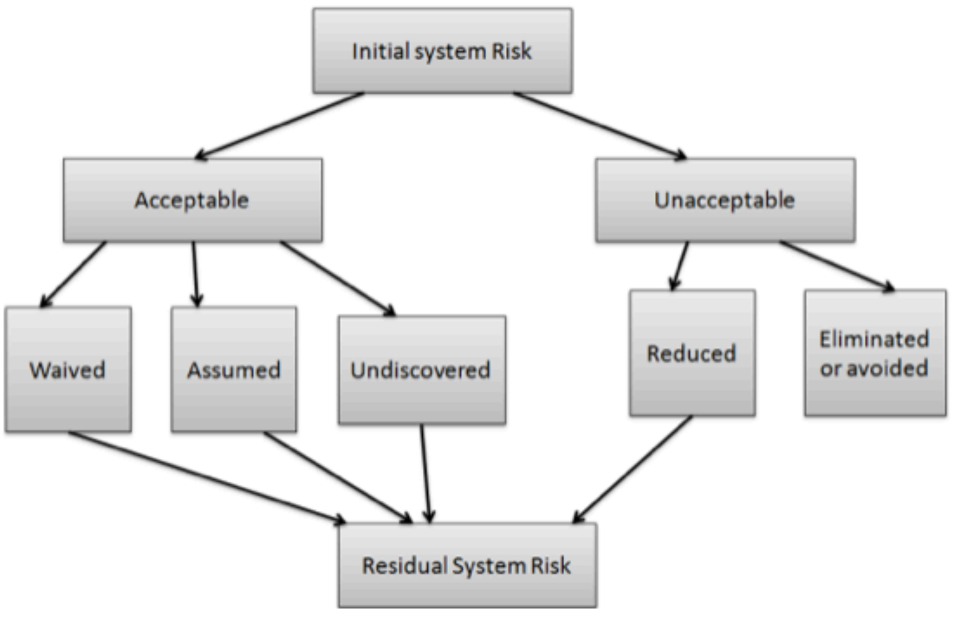
- High
- Low
- Medium

Quantitative Risk Analysis can be used for software risk analysis but is considered inappropriate because risk level is defined in % which does not give a very clear picture.

Software Risk Planning

Software risk planning is all about:

1. Defining preventive measures that would lower down the likelihood or probability of various risks.
2. Define measures that would reduce the impact in case a risk happens.
3. Constant monitoring of processes to identify risks as early as possible.



Software Risk Monitoring

Software risk monitoring is integrated into project activities and regular checks are conducted on top risks. Software risk monitoring comprises of:

- Tracking of risk plans for any major changes in actual plan, attribute, etc.
- Preparation of status reports for project management.
- Review risks and risks whose impact or likelihood has reached the lowest possible level should be closed.
- Regularly search for new risks

Q16. What is software quality? Discuss software quality attributes.

Software quality product is defined in terms of its fitness of purpose. That is, a quality product does precisely what the users want it to do. For software products, the fitness of use is generally explained in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory interpretation of quality for many devices such as a car, a table fan, a grinding machine, etc.for software products, "fitness of purpose" is not a wholly satisfactory definition of quality.

Definition by IEEE

The degree to which a system, component, or process meets specified requirements.

The degree to which a system, component, or process meets customer or user needs or expectations.

Definition by ISTQB

Quality: The degree to which a component, system, or process meets specified requirements and/or user/customer needs and expectations.

software quality: The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs.

Example: Consider a functionally correct software product. That is, it performs all tasks as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally right, we cannot consider it to be a quality product.

Attributes of Software Quality are -

Reliability

- Measure if the product is reliable enough to sustain in any condition. Should give consistently correct results.
- Product reliability is measured in terms of working on the project under different working environments and different conditions.

Maintainability

- Different versions of the product should be easy to maintain. For development it should be easy to add code to the existing system, it should be easy to upgrade for new features and new technologies from time to time.
- Maintenance should be cost-effective and easy. The system is easy to maintain and correcting defects or making a change in the software.

Usability

- This can be measured in terms of ease of use. The application should be user-friendly. It should be easy to learn. Navigation should be simple.
- Easy to use for input preparation, operation, and interpretation of the output.
- Provide consistent user interface standards or conventions with our other frequently used systems.
- Easy for new or infrequent users to learn to use the system.

Portability

- This can be measured in terms of Costing issues related to porting, Technical issues related to porting, Behavioral issues related to porting.

Correctness

- The application should be correct in terms of its functionality, calculations used internally and the navigation should be correct. This means the application should adhere to functional requirements.

Efficiency

- Major system quality attribute. Measured in terms of time required to complete any task given to the system. For example, the system should utilize processor capacity, disk space, and memory efficiently.
- If the system is using all the available resources then the user will get degraded performance failing the system for efficiency. If the system is not efficient then it can not be used in real-time applications.

Integrity or Security

- Integrity comes with security. System integrity or security should be sufficient to prevent unauthorized access to system functions, preventing information loss, ensure that the software is protected from virus infection, and protecting the privacy of data entered into the system.

Testability

- The system should be easy to test and find defects. If required it should be easy to divide into different modules for testing.

Flexibility

- Should be flexible enough to modify. Adaptable to other products with which it needs interaction. It should be easy to interface with other standard 3rd party components.

Reusability

- Software reuse is a good cost-efficient and time-saving development way. Different code libraries classes should be generic enough to use easily in different application modules. Dividing application into different modules so that modules can be reused across the application.

Interoperability

- The interoperability of one system to another should be easy for the product to exchange data or services with other systems. Different system modules should work on different operating system platforms, different databases, and protocols conditions.
- Applying above quality attributes standards we can determine whether the system meets the requirements of quality or not.

Q17. Explain why the process of project planning is iterative and why a plan must be continually reviewed during a software project.

We know that the project plan is developed based on the available information at that moment. Thus, at the beginning of the project, we develop the plan without knowing enough uncertain matters that are relevant to the project. When the project progresses, there are more and more matters relevant to the project become available. Thus, we need to revise the project plan iteratively. The progress of the project through phrases will make some changes to the plan, so the plan needs to be updated to reflect this change to help with monitoring the project.

Every project plan needs to be monitored and updated on a regular basis for the simple reason it is impossible to predict the future with certainty, therefore the plan will be wrong.

The degree of uncertainty varies depending on the project's typology and many other factors which indicates the best approach to maintaining the plan.

In traditional – linear - project management, the approach is to implement the activities under the assumption that all events affecting the project are predictable, that activities are well understood by everybody, and there is no need to revisit the plans. Unfortunately, this approach proves to be not very effective, given the level of uncertainty on many development projects. What happens is that the original assumptions under which the project plan was built change and in some cases in dramatic ways. What was originally assumed to be true is no longer valid. Assumptions about approvals, additional funding, economic and social conditions change dynamically and the project needs to have the flexibility to adapt to these changes.

The project should find these opportunities to review the original assumptions and make the appropriate changes to the plans, specifically in the areas of scheduling, risks, and stakeholders. This approach consists of a series of iterative planning and development cycles, allowing a project team to constantly evaluate the implementation and results of the project and obtain immediate feedback from beneficiaries, or stakeholders.

Q18. Some very large software projects involve writing millions of lines of code. Explain why the effort estimation models, such as COCOMO, might not work well when applied to very large systems.

CoCoMo (Constructive Cost Model) is a regression model based on LOC, i.e number of Lines of Code. It is a procedural cost estimate model for software projects and often used as a process of reliably predicting the various parameters associated with making a project such as size, effort, cost, time, and quality. The key parameters which define the quality of any software products, also an outcome of the CoCoMo are primarily Effort & Schedule:

- Effort: Amount of labor that will be required to complete a task. It is measured in person-months units.
- Schedule: Simply means the amount of time required for the completion of the job, which is, of course, proportional to the effort put. It is measured in the units of time such as weeks, months.

The effort estimation models such as CoCoMo estimate the software size based on the application and function points, and lines of code. For smaller projects, the estimation based on these factors will be effective. But in the case of large systems (millions of lines of code), coding is only a small fraction of the total effort. LOC becomes a minor factor in project estimation. The estimation models may not work well when there is ambiguity in counting codes. Effort estimation depends on various factors such as requirements, plans, people, etc. So for very large systems, effort estimation models such as COCOMO might not work well.

Q19. What are the objectives of testing? Explain why testing can only detect the presence of errors, not their absence.

Software testing is a crucial element in the software development life cycle (SDLC), which can help software engineers save time & money for organizations by finding errors and defects during the early stages of software development. With the assistance of this process, one can examine various components associated with the application and guarantee its appropriateness.

The goals and objectives of software testing are numerous, which when achieved help developers build a defect-less and error-free software and application that has exceptional performance, quality, effectiveness, security, among other things. Though the objective of testing can vary from company to company and project to project, there are some goals that are similar for all. These objectives are:

Verification: A prominent objective of testing is verification, which allows testers to confirm that the software meets the various business and technical requirements stated by the client before the inception of the whole project. These requirements and specifications guide the design and development of the software, hence they are required to be followed rigorously. Moreover, compliance with these requirements and specifications is important for the success of the project as well as to satisfy the client.

Validation: Confirms that the software performs as expected and as per the requirements of the clients. Validation involves checking the comparing the final output with the expected output and then making necessary changes if there is a difference between the two.

Defects: The most important purpose of testing is to find different defects in the software to prevent its failure or crash during implementation or go-live of the project. Defects if left undetected or unattended can harm the functioning of the software and can lead to loss of resources, money, and reputation of the client. Therefore, software testing is executed regularly during each stage of software development to find defects of various kinds. The ultimate source of these defects can be traced back to a fault introduced during the specification, design, development, or programming phases of the software.

Providing Information: With the assistance of reports generated during the process of software testing, testers can accumulate a variety of information related to the software and the steps taken to prevent its failure. These then can be shared with all the stakeholders of the project for a better understanding of the project as well as to establish transparency between members.

Preventing Defects: During the process of testing the aim of testes to identify defects and prevent them from occurring ageing in the future. To accomplish this goal, the software is tested rigorously by independent testers, who are not responsible for software development.

Quality Analysis: Testing helps improve the quality of the software by constantly measuring and verifying its design and coding. Additionally, various types of testing techniques are used by testers, which help them achieve the desired software quality.

Compatibility: It helps validate the application's compatibility with the implementation environment, various devices, Operating Systems, user requirements, among other things.

Optimal User Experience: Easy software and application accessibility and optimum user experience are two important requirements that need to be accomplished for the success of any project as well as to increase the revenue of the client. Therefore, to ensure this software is tested again and again by the testers with the assistance of stress testing, load testing, spike testing, etc.

Verifying Performance & Functionality: It ensures that the software has superior performance and functionality. This is mainly verified by placing the software under extreme stress to identify and measure its all plausible failure modes. To ensure this, performance testing, usability testing, functionality testing, etc. are executed by the testers.

Testing can detect only the presence of errors, not their absence because the main goal of the testing is: to observe the behavior of the particular software and to check whether it meets its requirement expectation or not. ... It is always possible that a test overlooked could discover a further problem with the system.

Q20. What are the different levels of testing and the goals of the different levels? For each level, specify the most suitable testing approach.

There are generally four recognized levels of testing: unit/component testing, integration testing, system testing, and acceptance testing. Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test. Below you can see 4 different levels of testing:

1. Unit/Component Testing
2. Integration testing
3. System testing
4. Acceptance testing

1. Unit/component testing - Unit testing aims to verify each part of the software by isolating it and then perform tests to demonstrate that each individual component is correct in terms of fulfilling requirements and the desired functionality. This type of testing is performed at the earliest stages of the development process, and in many cases, it is executed by the developers themselves before handing the software over to the testing team.

The advantage of detecting any errors in the software early in the day is that by doing so the team minimizes software development risks, as well as time and money wasted in having to go back and undo fundamental problems in the program once it is nearly completed.

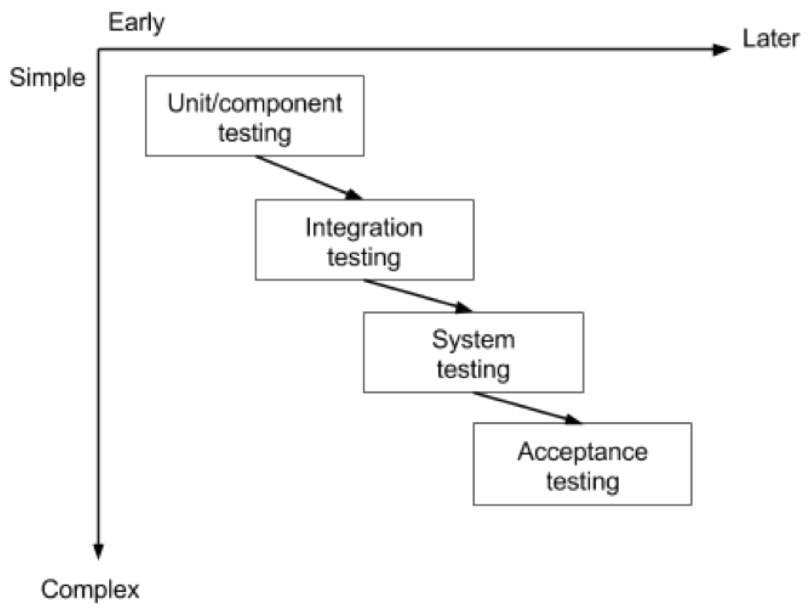
2. Integration testing - Integration testing aims to test different parts of the system in combination in order to assess if they work correctly together. By testing the units in groups, any faults in the way they interact together can be identified. There are many ways to test how different components of the system function at their interface; testers can adopt either a bottom-up or a top-down integration method. In bottom-up integration testing, testing builds on the results of unit testing by testing a higher-level combination of units (called modules) in successively more complex scenarios. It is recommended that testers start with this approach first, before applying the top-down approach which tests higher-level modules first and studies simpler ones later.

3. System testing - The next level of testing is system testing. As the name implies, all the components of the software are tested as a whole in order to ensure that the overall product meets the requirements specified. System testing is a very important step as the software is almost ready to ship and it can be tested in an environment that is very close to that which the user will experience once it is deployed. System testing enables testers to ensure that the product meets business requirements, as well as determine that it runs smoothly within its operating environment. This type of testing is typically performed by a specialized testing team.

4. Acceptance testing - acceptance testing is the level in the software testing process where a product is given the green light or not. The aim of this type of testing is to evaluate whether the system complies with the end-user requirements and if it is ready for deployment. The testing team will utilize a variety of methods, such as pre-written scenarios and test cases to test the software and use the results obtained from these tools to find ways in which the system can be improved. The scope of acceptance testing ranges from simply finding spelling mistakes and cosmetic errors to uncovering bugs that could cause a major error in the application. By performing acceptance tests, the testing team can find out how the product will perform when it is installed on the user's system.

The testing sequence

These four testing types cannot be applied haphazardly during development. There is a logical sequence that should be adhered to in order to minimize the risk of bugs cropping up just before the launch date. Any testing team should know that testing is important at every phase of the development cycle. By progressively testing the simpler components of the system and moving on the bigger, more complex groupings, the testers can rest assured they are thoroughly examining the software in the most efficient way possible.



The four levels of testing shouldn’t only be seen as a hierarchy that extends from simple to complex, but also as a sequence that spans the whole development process from the early to the later stages. Note however that later does not imply that acceptance testing is done only after say 6 months of development work.

Q21. What is software reliability? Discuss the following models of software reliability (a) Basic Execution Time model (b) Jelinski-Moranda model.

Software Reliability is the probability of failure-free software operation for a specified period of time in a specified environment. Software Reliability is also an important factor affecting system reliability. It differs from hardware reliability in that it reflects the design perfection, rather than manufacturing perfection. The high complexity of software is the major contributing factor to Software Reliability problems. Software Reliability is not a function of time - although researchers have come up with models relating the two. The modeling technique for Software Reliability is reaching its prosperity, but before using the technique, we must carefully select the appropriate model that can best suit our case. Measurement in software is still in its infancy. No good quantitative methods have been developed to represent Software Reliability without excessive limitations. Various approaches can be used to improve the reliability of software, however, it is hard to balance development time and budget with software reliability.

Reliability Growth Models -The exponential model can be regarded as the basic form of software reliability growth models. The basic execution model is the most popular and widely used reliability growth model, mainly because:

- It is practical, simple and easy to understand;
- Its parameters clearly relate to the physical world.
- It can be used for accurate reliability prediction. -The basic execution model specifies failure behavior initially using execution time, convertible to calendar time.

- The failure behavior is a non-homogeneous Poisson process, which means the associated probability distribution is a Poisson process whose characteristics vary in time.
- It is equivalent to the M-O logarithmic Poisson execution time model, with different mean value functions.
- The mean value function, in this case, is based on an exponential distribution.

Variables involved in the basic execution model:

- Failure intensity (lambda): number of failures per time unit.
- Execution time (tau): time since the program is running.
- Mean failures experienced (μ): mean failures experienced in a time interval
- In the basic execution model, the mean failures experienced m is expressed in terms of the execution time t as

$$m(t) = n_0 \times \; 1 - e^{-\frac{l_0}{n_0}t}$$

The Jelinski-Moranda (JM) model, which is also a Markov process model, has strongly affected many later models which are in fact modifications of this simple model.

Characteristics of JM Model

Following are the characteristics of JM-Model

- It is a Binomial type model
- It is certainly the earliest and certainly one of the most well-known black-box models.
- The J-M model always yields an over-optimistic reliability prediction.
- JM Model follows a perfect debugging step, i.e., the detected fault is removed with certainty in a simple model.
- The constant software failure rate of the JM model at the i^{th} failure interval is given by:

$\lambda(t_i) = \phi\, [N-(i-1)]$, i=1, 2... N ____________________ equation 1

$\lambda(t_i) = \phi[N - (i - 1)]$

ϕ=a constant of proportionality indicating the failure rate provided by each fault

N=the initial number of errors in the software

ti=the time between $(i - 1)^{th}$ and $(i)^{th}$ failure.

The mean value and the failure intensity methods for this model which belongs to the binomial type can be obtained by multiplying the inherent number of faults by the cumulative failure and probability density functions (pdf) respectively:

$\mu(t_i) = N(1 - e - \phi t_i)$ ___________________________equation 2

And

$\epsilon(t_i) = N\phi e - \phi t_i$ _____________________________equation 3

The assumptions made in the J-M model contains the following:

- The number of initial software errors is unknown but fixed and constant.
- Each error in the software is independent and equally likely to cause a failure during a test.
- Time intervals between occurrences of failure are separate, exponentially distributed random variables.
- The software failure rate remains fixed over the ranges among fault occurrences.
- The failure rate is corresponding to the number of faults that remain in the software.
- A detected error is removed immediately, and no new mistakes are introduced during the removal of the detected defect.
- Whenever a failure appears, the corresponding fault is reduced with certainty.

Q22. List some problems that will come up if the methods used for developing small software are used for developing large software systems.

Agile methods are incremental development methods in which the increments are small and, typically, new releases of the system are created and made available to customers every two or three weeks. They involve customers in the development process to get rapid feedback on changing requirements. They minimize documentation.

In practice, the principles underlying agile methods are sometimes difficult to realize:

1. Although the idea of customer involvement in the development process is an attractive one, its success depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders.Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods, and therefore not interact well with other team members.
2. Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.
3. Maintaining simplicity requires extra work. Under pressure from delivery schedules, the team members may not have time to carry out desirable system simplifications.
4. Many organizations, especially large companies, have spent years changing their culture so that processes are defined and followed. It is difficult for them to move to a working model in which processes are informal and defined by development teams.

Large software system development is different from small system development in a number of ways:

1. Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones. It is practically impossible for each team to have a view of the whole system. Consequently, their priorities are usually to complete their part of the system without regard for wider systems issues.
2. Large systems are 'brownfield systems' i.e. they include and interact with multiple existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development. Political issues can also be significant here—often the easiest solution to a problem is to change an existing system. However, this requires negotiation with the managers of that system to convince them that the changes can be implemented without risk to the system's operation.

3. Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development. This is not necessarily compatible with incremental development and frequent system integration.
4. Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed, that require certain types of system documentation to be produced, etc.
5. Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
6. Large systems usually have a diverse set of stakeholders. For example, nurses and administrators may be the end-users of a medical system but senior medical staff, hospital managers, etc. are also stakeholders in the system. It is practically impossible to involve all of these different stakeholders in the development process.

It is difficult to introduce agile methods into large companies for a number of reasons:

1. Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach, as they do not know how this will affect their particular projects.
2. Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods. Sometimes, these are supported by software tools (e.g., requirements management tools) and the use of these tools is mandated for all projects.
3. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities, and people with lower skill levels may not be effective team members in agile processes.
4. There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

Q23. If the primary goal is to make software maintainable, list some of the things you will do and some of the things you will not do during coding and testing

Some points that a software engineer must keep in mind while working on any software are as follows

- Design for maintainability from the outset
- Iterative development and regular reviews help to improve quality - see the section below
- Readable code is easy to understand ("write programs for people")
- Refactor code to improve its understandability
- Relevant documentation helps developers understand the software
- Automated build make the code easy to compile
- Automated tests make it easy to validate changes
- Continuous integration makes the code easier to build and test
- Version control helps keep code, tests, and documentation up to date and synchronized
- Change the way you work to make maintainability a key goal

Q24. Suggest five possible problems that could arise if a company does not develop effective configuration management policies and processes.

Configuration management is the name given to the general process of managing a changing software system, with the aim to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

Problems that could arise if company does not have effective configuration management policies and processes :

1. New versions of software systems cannot be created effectively as they change. Developers cannot keep track of the changes to the software
2. Controlling the costs and effort involved in making changes to a system is difficult
3. Wrong version of a system may be delivered to the customers or forget where the software source code for a particular version of the system or component is stored
4. If someone leave step company, protecting investments in software and the ability to reproduce a build with the correct components or continue development on a project is difficult
5. Ineffective quality management process because configuration management may be seen as part of a more general quality management process.

Q25. Discuss reverse engineering and re-engineering

RE Engineering:

• Restructuring or rewriting part or all of a system without changing its functionality

• Applicable when some (but not all) subsystems of a larger system require frequent maintenance

• Reengineering involves putting in the effort to make it easier to maintain

• The re-engineered system may also be restructured and should be re-documented

When do you decide to re-engineer?

• When system changes are confined to one subsystem, the subsystem needs to be reengineered

• When hardware or software support becomes obsolete

• When tools to support restructuring are readily available

Re-engineering advantages:

Reduced risk

There is a high risk of 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

The complete Software Reengineering life cycle includes:

- Product Management: Risks analysis, root cause analysis, business analysis, requirements elicitation and management, product planning and scoping, competitive analysis
- Research and Innovation: Definition of a problem, data gathering, and analysis identifying a solution and developing best-of-breed or innovative algorithms, verification of quality for data and results, patent preparation
- Product Development: Technology analysis and selection, software architecture and design, data architecture, deployment architecture, prototyping and production code development, comprehensive software testing, data quality testing, and product packaging and deployment preparation
- Product Delivery and Support: Hardware/Platform analysis and selection, deployment and release procedures definition, installations and upgrades, tracking support issues, organizing maintenance releases.

- Project Management: Brings efficiency and productivity to your software re-engineering project by utilizing modern, practical software project management, software quality assurance, data quality assurance, and advanced risk management techniques.

Reverse Engineering:

Reverse engineering is taking apart an object to see how it works in order to duplicate or enhance the object. The practice, taken from older industries, is now frequently used on computer hardware and software. Software reverse engineering involves reversing a program's machine code (the string of 0s and 1s that are sent to the logic processor) back into the source code that it was written in, using program language statements.

Reverse-engineering is used for many purposes: as a learning tool; as a way to make new, compatible products that are cheaper than what's currently on the market; for making software interoperate more effectively or to bridge data between different operating systems or databases, and to uncover the undocumented features of commercial products.

Following are reasons for reverse engineering a part or product:

1. The original manufacturer of a product no longer produces a product
2. There is inadequate documentation of the original design
3. The original manufacturer no longer exists, but a customer needs the product
4. The original design documentation has been lost or never existed
5. Some bad features of a product need to be designed out. For example, excessive wear might indicate where a product should be improved
6. To strengthen the good features of a product based on long-term usage of the product
7. To analyze the good and bad features of competitors' product
8. To explore new avenues to improve product performance and features
9. To gain competitive benchmarking methods to understand the competitor's products and develop better products
10. The original CAD model is not sufficient to support modifications or current manufacturing methods
11. The original supplier is unable or unwilling to provide additional parts
12. The original equipment manufacturers are either unwilling or unable to supply replacement parts or demand inflated costs for sole-source parts
13. To update obsolete materials or antiquated manufacturing processes with more current, less-expensive technologies.