Q.1 a. Explain any four generic process framework activities applicable to majority of software projects.

Answer:
The following generic process framework is applicable to vast majority of software projects:

1. Communication: This framework activity involves heavy communication and collaboration with the customer and other stakeholders and encompasses requirements gathering and other related activities.

2. Planning: This activity establishes a plan for the software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required the work products to be produced, and a work schedule.

3. Modeling: This activity encompasses the creation of models that allow the developer and the customer to better understand software requirements and the design that will achieve those requirements.

4. Construction: This activity combines code generation (either manual or automated) and testing that is required to uncover errors in the code.

5. Deployment: The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on evaluation.

b. What is requirement engineering? Why is it important?

Answer:
Requirement engineering helps software engineers to better understand the problem they will work to solve. It encompasses the set of tasks that lead to an understanding of what the business impact of the software will be, what the customer wants, and how end-user will interact with the software.

Designing and building an elegant computer program that solves the wrong problems serves no one's needs. For this reason it is important to understand the needs of the customer before designing and developing software.

c. Explain the concept of modularity in the context of software design. What are the benefits of modularity?

Answer:
Modularity is the concept in which the software is divided into separately named and addressable components that are integrated to satisfy problem requirements. Modularity is a single attribute of software that allows a program to be intellectually manageable. A large program composed of a single module cannot be grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to

impossible. It also follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. Though the effort (cost) to develop an individual software module does decrease as the total number of modules increases, but the cost associated with integrating the modules increases. The benefits of modularizing a design are:

- Development can be more easily planned.
- Software increments can be defined and delivered.
- Changes can be more easily accommodated.
- Testing and debugging can be conducted more efficiently.
- Long-term maintenance can be conducted without serious side effects.

d. Explain the process of program debugging.

Answer:
Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is an action that results in the removal of error. Debugging is not testing but always occurs as a consequence of testing. The debugging process begins with the execution of a test case. Results are assessed and the lack of correspondence between expected and actual performance is encountered. In many cases, the non-corresponding data are symptom of an underlying cause as yet hidden. Debugging attempts to match symptom with cause, thereby leading to error correction. Debugging will have one of the two outcomes: (1) the cause will be found and corrected, or(2) the cause will not be found. In the later case, the person performing debugging may suspect a cause, design one or more test cases to help validate that suspicion, and work toward error correction in an iterative fashion.

e. Explain software project scheduling.

Answer:
Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. It is important to note, however, that the schedule evolves over time. During early stages of project planning, a macroscopic schedule is developed. This type of schedule identifies all major process framework activities and product functions to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule. Here, specific software tasks (required to accomplish an activity) are identified and scheduled.

Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end-date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but the end-date is set by the software engineering organization. Effort is distributed to make best use of resources and an end-date is defined after carefully analysis of the software.

f. Explain the concept of software reliability along with an example.

Answer:
Software reliability, unlike many other quality factors can be measured directed and estimated using historical and developmental data. Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time". For example, program X is estimated to have a reliability of 0.96 over eight elapsed processing hours. In other words, if program X were to be executed 100 times and require a total of eight hours of elapsed processing time (execution time), it is likely to operate correctly (without failure) 96 times.

g. Explain software reverse-engineering.

Answer:
Reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code. Reverse engineering is the process of design recovery. Reverse engineering tools extract data, architectural, and procedural design information from an existing program. Reverse engineering is performed to understand the internal workings of a program.

Q.2 a. What is software engineering? Explain the three layers of software engineering.

Answer: Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines. It is the application of systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software. The three layers of software engineering are:

1. **Process:** The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software project and establishes the context in which technology methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.
2. **Methods:** Software engineering methods provide the technical "how to' s" for building software. Methods encompass a broad array of tasks that include communication, testing, and support of technology and include modeling activities and other descriptive techniques.
3. **Tools:** Software engineering tools provide automated or semi-automated support for the process and the methods. When tools are integrated so that information crated by one tool can be used by another, a system for the support of software development, called computer-aided-software engineering, is established.

b. How do process models differ from one another?

Answer:
The process models differ from one another fundamentally in:

1. The overall flow of activities and tasks and the interdependencies among activities and tasks.
2. The degree to which work tasks are defined within each framework activity.
3. The degree to which work products are identified and required.
4. The manner in which quality assurance activities are applied.
5. The manner in which project tracking and control activities are applied.
6. The overall degree of detail and rigor with which the process is described.
7. The degree to which customer and other stakeholders are involved with the project.
8. The level of autonomy given to the software project team.
9. The degree to which team organization and roles are prescribed.

c. Describe the waterfall model for software development. Why does waterfall model sometimes fail?

Answer:
The waterfall model, sometimes called the classic life cycle, suggests a systemic sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in on-going support of the completed software. The waterfall model sometimes fails due to the following reasons:

1. Real projects rarely follow the sequential that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
2. It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.
3. The customer must have patience. A working version of the software will not be available until late in the project time-span.
4. A major blunder, if undetected until the working software is reviewed, can be disastrous. Problems are often not discovered until system testing.
5. This model is very resource intensive. A tremendous amount of time must be spent on gathering information and preparing specifications and sign-off documents.

Q.3 a. Why is it difficult to gain a clear understanding of what the customer wants? How this problem can be overcome?

Answer:
Following are the reasons due to which it becomes difficult to gain a clear understanding of customer wants:

1. **Problems of scope:** The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
2. **Problems of understanding:** The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, do not have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.
3. **Problems of volatility:** The requirements change over time.

These problems can be overcome by requirement engineers by approaching the requirements gathering activity in an organized manner.

b. What is requirement analysis? What are the primary objectives that must be achieved by the analysis model?

Answer:
Requirement analysis allows the software engineer to elaborate on basic requirements established during earlier requirement engineering tasks and build models that depict user scenarios, functional activities, problem classes and their relationships, system and class behavior, and the flow of data as it is transformed. Requirements analysis provides the software designer with a representation of information, function, and behaviour that can be translated to architectural, interface, and component-level designs. The analysis model bridges the gap between a system-level description that describes overall system functionality. Finally, the analysis model and requirement specification provide the customer and the developer with the means to access quality once software is built. Throughout analysis modeling, the software engineer's primary focus is on *what*, not *how*. The primary objectives that must be achieved by the analysis model are:

1. To describe what the customer requires.
2. To establish a basis for the creation of software design.
3. To define a set of requirements that can be validated once the software is built.

c. Define the four characteristics of a well-formed design class.

Answer:
The four characteristics of a well-formed design class are:

1. **Complete and sufficient:** A design class should be complete encapsulation of all attributes and methods that can reasonably be expected to exist for the class. Sufficiency ensures that design class contains only those methods that are sufficient to achieve the intent of the class, no more and no less.

2. **Primitiveness:** Methods associated with the design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplishing the same thing.
3. **High cohesion:** A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.
4. **Low coupling:** Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design is highly coupled (all design classes collaborate with other design classes), the system is difficult to implement, to test, and to maintain over time. In general, design classes within a sub-system should have only limited knowledge of classes in other sub-systems. This restriction, called the Law of Demeter, suggests that a method should only send messages to methods in neighboring classes.

Q.4 a. What is coupling? Describe five different types of coupling.

Answer:
Coupling is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more independent, an important objective in component-level design is to keep coupling as low as possible. The different types of coupling are:

1. **Content coupling:** Occurs when one component "surreptitiously modifies data that is internal to another component". This violates information hiding-a basic design concept.
2. **Common coupling:** Occurs when a number of components all make use of a global variable. Although, this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application), common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.
3. **Control coupling:** Occurs when *operation A()* invokes *operation B()* and passes a control flag to *B*. The control flag then "directs" logical flow within *B*. The problem with this type of coupling is that an unrelated change in *B* can result in the necessity to change the meaning of the control that *A* passes. If this is overlooked, an error will result.
4. **Stamp coupling:** Occurs when **ClassB** is declared as a type for an argument of an operation of **ClassA**. Because **ClassB** is now a part of the definition of **ClassA**, modifying the system becomes more difficult.
5. **Data coupling:** Occurs when operations pass long strings of data arguments. The "bandwidth" of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.

6. **Routine call coupling:** Occurs when one operation invokes another. This level of coupling is common and is often quite necessary. However, it does increase the connectedness of a system.
7. **Type use coupling:** Occurs when component **A** uses a data type defined in component **B** (e.g., this occurs whenever "a class declares an instance variable or a local variable as having another class of its type". If the type definition changes, every component that uses the definition must also change.
8. **Inclusion or import coupling:** Occurs when component **A** imports or includes a package or the content of component **B**.
9. **External coupling:** Occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). It should be limited to a small number of components or classes within a system.

Q.5 b. What are the basic principles that guide software project scheduling?

Answer:
The basic principles that guide software development scheduling are:

1. **Compartmentalization:** The project must be compartmentalized into a number of manageable activities, actions, and tasks. To accomplish compartmentalization, both the product and the process are decomposed.
2. **Interdependency:** The interdependency of each compartmentalized activity, action, or task must be determined. Some tasks must occur in sequence while others can occur in parallel. Some actions or activities cannot commence until the work product produced by another is available. Other actions or activities can occur independently.
3. **Time allocation:** each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.
4. **Effort validation:** Every project has a defined number of people on the software team. As time allocation occurs, the project manager must ensure that no more than the allocated number of people has been scheduled at any given time.
5. **Defined responsibilities:** Every task that is scheduled should be assigned to a specific team member.
6. **Defined outcomes:** Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a module) or a part of a work product. Work products are often combined in deliverables.
7. **Defined milestones:** Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

Q.6 a. What are the activities defined in the software reengineering process model?

Answer:
The following six activities are defined in the software reengineering process:

1. **Inventory analysis:** Every software organization should have an inventory of all applications. The inventory can be nothing more than a spreadsheet model containing information that provides a detailed description (e.g., size, age, business criticality) of every active application. By sorting this information according to business criticality, longevity, current maintainability, and other locally important criteria, candidates for reengineering appear. Resources can then be allocated to candidate applications for reengineering work.
2. **Document restructuring:** Weak documentation is the trade mark of many legacy systems. The following options can be used:
    - **Creating documentation is far too time consuming:** If the system works, we will live with what we have. In some cases this is the correct approach. It is not possible to recreate documentation for hundreds of computer programs. If a program is relatively static, is coming to the end of its useful life, and is unlikely to undergo significant change, let it be!
    - **Documentation must be updated, but we have limited resources:** It may not be necessary to fully redocument an application. Rather, those portions of the system that are currently undergoing change are fully documented. Over time, a collection of useful and relevant documentation will evolve.
    - **The system is business critical and must be fully redocumented:** Even in this case, an intelligent approach is to pare documentation to an essential minimum.

Each of these options is viable. A software organization must choose the one that is most appropriate for each case.

3. **Reverse engineering:** Reverse engineering for software is the process of analyzing a program in an effort to create a representation of the program at a higher level of abstraction than source code. Reverse engineering is the process of design recovery. Reverse engineering tools extract data, architectural, and procedural design information from an existing program. Reverse engineering is performed to understand the internal workings of a program.
4. **Code restructuring:** Some legacy systems have relatively solid program architecture, but individual modules were coded in a way that makes them difficult to understand, test, and maintain. To accomplish this activity, the source code is analyzed using a restructuring tool. Violations of structured programming constructs are noted, and code is then restructured. The resultant code is reviewed and tested to ensure that no anomalies have been introduced. Internal code documentation is updated.
5. **Data restructuring:** A program with weak data architecture will be difficult to adapt and enhance. Data restructuring is a full-scale reengineering activity. In most cases, data restructuring begins with a reverse engineering activity. Current data architecture is

dissected, and necessary data models are defined. Data objects and attributes are identified, and existing data structures are reviewed for quality.

6. **Forward engineering**: forward engineering, also called renovation or reclamation, not only recovers design information from existing software, but uses this information to alter or reconstitute the existing system in an effort to improve its overall quality. In most cases, reengineered software reimplements the function of the existing system and also adds new functions and/or improves overall performance.

b. Describe the software testing principles.

Answer:
Following are the software testing principles:

1. **All tests should be traceable to customer requirements:** The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.
2. **Tests should be planned long before testing begins:** Test planning can begin as soon as the analysis is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.
3. **The pareto principle applies to software testing:** The pareto principle implies that 80% of all errors uncovered during testing will likely be traceable to 20% all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.
4. **Testing should begin "in the small" and progress toward testing "in the large":** The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts an attempt to find errors in integrated clusters of components and ultimately in the entire system.
5. **Exhaustive testing is not possible:** The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

Q.7 a. What is system testing? Describe the different types of system tests.

Answer:
System testing is a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that the system elements have been properly integrated and perform allocated functions. System processing is a process in which the software and other system elements are tested as a whole. The different types of system tests are:

1. **Recovery testing:** Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that the recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair is evaluated to determine whether it is within acceptable limits.
2. **Security testing:** The system's security must be tested for invulnerability from frontal attacks as well as invulnerability from flank or rear attack. This is the process to determine that the system protects data and maintains functionality as intended. During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. The tester may attempt to acquire passwords through external clerical means, may attack the system with custom software designed to break down any defenses that have been constructed, may purposely cause system errors, hoping to penetrate during recovery, may browse through insecure data, hoping to find the key to system entry.
3. **Stress testing:** Stress testing is used to determine the stability of a given system or entity. It involves testing beyond normal operational capacity, often to a breaking point, in order to observe the results. These tests are designed to confront programs with abnormal situations. For example, special tests may be designed that generate 10 interrupts when one or two is the average rate; input data rates may be increased by an order of magnitude to determine how input function will perform; test cases that require maximum memory or other resources are executed; test cases that may cause memory management problems are designed; or the test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to overwhelm the program.
4. **Performance testing:** Performance testing is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted.

   However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.

b. What are the uses of function point metric?

Answer:

The function point metric (FP) can be used effectively for measuring the functionality delivered by a system. Using historical data, the FP can be used to:

1. Estimate the cost or effort required to design, code, and test the software.
2. Predict the number of errors that will be encountered during testing.
3. Forecast the number of components and/or the number of projected source lines in the implemented system.

c. Explain PERT or CPM.

Answer:
PERT and CPM are two project scheduling methods that can be applied to software development. Both PERT and CPM provide quantitative tools that allow the software planner to:

1. Determine the critical path-the chain of tasks that determines the duration of the project.
2. Establish "most likely" time estimates for individual tasks by applying statistical models.
3. Calculate "boundary times" that define a time "window" for a particular task.

Text Books

1. Roger Pressman, Software Engineering: A Practitioners approach, Sixth Edition, McGraw-Hill International Edition

2. Pankaj Jalote, An Integrated Approach to Software Engineering, Third Edition, Narosa Publishing House, 2008.