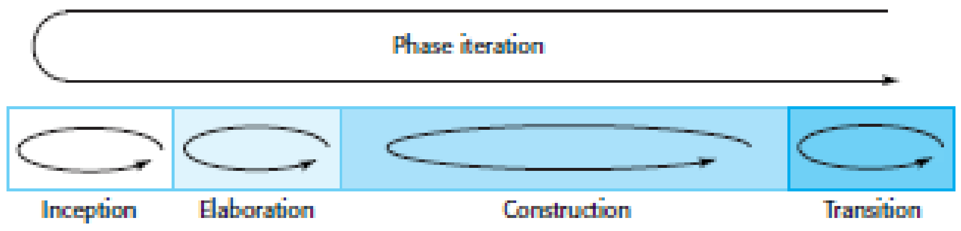
Q.2 a. What is Rational Unified Process? With the help of suitable figure, explain the different phases of rational unified process.

Answer:
The **Rational Unified Process** (RUP) is an example of a modern process model that has been derived from work on the UML and the associated Unified Software Development Process.
The RUP recognises that conventional process models present a single view of the process. In contrast, the RUP is normally described from three perspectives:
1. A dynamic perspective that shows the phases of the model over time.
2. A static perspective that shows the process activities that are enacted.
3. A practice perspective that suggests good practices to be used during the process.

Most descriptions of the RUP attempt to combine the static and dynamic perspectives in a single diagram. The RUP is a phased model that identifies four discrete phases in the software process. However, unlike the waterfall model where phases are equated with process activities, the phases in the RUP are more closely related to business rather than technical concerns. Figure below shows the phases in the RUP.



Phases in the Rational Unified Process

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

2. *Elaboration* The goals of the elaboration phase are to develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan and identify key project risks. On completion of this phase, you should have a requirements model for the system (UML use cases are specified), an architectural description and a development plan for the software.

3. *Construction* The construction phase is essentially concerned with system design, programming and testing. Parts of the system are developed in parallel and integrated during this phase. On completion of this phase, you should have a working software system and associated documentation that is ready for delivery to users.

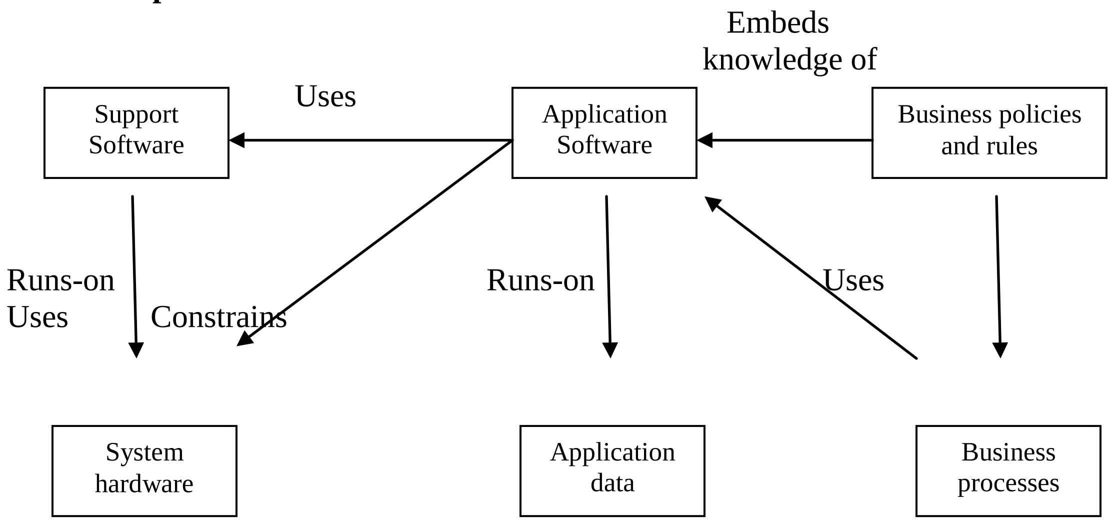
4. *Transition* The final phase of the RUP is concerned with moving the system from the development community to the user community and making it work in a real environment. This is something that is ignored in most software process models but is, in fact, an expensive and sometimes problematic activity.
On completion of this phase, you should have a documented software system that is working correctly in its operational environment.

Iteration within the RUP is supported in two ways, as shown in Figure. Each phase may be enacted in an iterative way with the results developed incrementally. In addition, the whole set of phases may also be enacted incrementally, as shown by the looping arrow from Transition to Inception in Figure.

b. **Define Legacy system. With the help of figure, explain the logical parts of a legacy system and their relationships.**

Answer:
Legacy systems:- Are socio-technical computer-based systems that have been developed in the past, often using older or obsolete technology. These systems include not only hardware and software but also legacy processes and procedures, old ways of doing things that are difficult to change because they rely on legacy software. Changes to one part of the system inevitably involve changes to other components. Legacy systems are often business-critical systems. They are maintained because it is too risky to replace them.

The following figure illustrates the logical parts of a legacy system and their relationships:



Legacy System Components

1. ***System hardware*** In many cases, legacy systems have been written for mainframe hardware that is no longer available, that is expensive to maintain and that may not be compatible with current organisational IT purchasing policies.
2. ***Support software*** The legacy system may rely on a range of support software from the operating system and utilities provided by the hardware manufacturer through to the compilers used for system development. Again, these may be obsolete and no longer supported by their original providers.
3. ***Application software*** The application system that provides the business services is usually composed of a number of separate programs that have been developed at different times. Sometimes the term legacy system means this application software system rather the entire system.
4. ***Application data*** These are the data that are processed by the application system. In many legacy systems, an immense volume of data has accumulated over the lifetime of the system. This data may be inconsistent and may be duplicated in several files.
5. ***Business processes*** These are processes that are used in the business to achieve some business objective. Business processes may be designed around a legacy system and constrained by the functionality that it provides.

Business policies and rules These are definitions of how the business should be carried out and constraints on the business. Use of legacy application system may be embedded in these policies and rules.

Q.3 a. What are the different types of requirements? Distinguish between function and non-functional requirements.

Answer: Following are three different types of requirements:

- Known requirements : Something a stakeholder believes to be implemented.
- Unknown requirements : forgotten by the stakeholder because they are not needed right now or needed only by another stakeholder.
- Undreamt requirements : Stakeholder may not be able to think of new new requirements due to limited domain knowledge.

Software requirements are broadly classified as functional and non-functional requirements:

Functional requirements: These are related to the expectations from the intended software. They describe what the software has to do. They are also called product features. Sometimes, functional requirements may also specify what the software should not do.

Non-Functional requirements: Non-functional requirements are mostly quality requirements that stipulate how well the software does what it has to do. Non-functional requirements that are especially important to users include specifications of desired performance, availability, usability and flexibility. Non-functional requirements for developers are maintainability, portability and testability.

The functional requirements are directly related to customer's expectations and are essential for the acceptance of product. However, non-functional requirements may make the customer happy and satisfied. These requirements are important for the success of any product. Sometimes, distinction between functional and non-functional may not be easy. If we want to develop a secure system, security becomes crucial to us. It is non-functional requirements apparently, but when design is carried out, it may have serious implications. These implications may also be reflected in functional requirements and should therefore find a place in the functional requirements category of SRS document.

b. What is Data dictionary? What are the advantages of using data dictionary?

Answer:
A **data dictionary** is an alphabetic list of the names included in the system models. As well as the name, the dictionary should include an associated description of the named entity and, if the name represents a composite object, a description of the composition. Other information such as the date of creation, the creator and the representation of the entity may also be included depending on the type of model being developed.

The advantages of using a data dictionary are:

1. *It is a mechanism for name management*. Many people may have to invent names for entities and relationships when developing a large system model. These names should be used consistently and should not clash. The data dictionary software can check for name uniqueness where necessary and warn requirements analysts of name duplications.

2. *It serves as a store of organizational information*. As the system is developed, information that can link analysis, design, implementation and evolution is added to the data dictionary, so that all information about an entity is in one place.

c. Write short notes on Ethnography.

Answer:
Ethnography: - is an observational technique that can be used to understand social and organizational requirements. An analyst immerses him or herself in the working environment where the system will be used. He or she observes the day-to-day work and notes made of the actual tasks in which participants are involved. The value of ethnography is that it helps analysts discover implicit system requirements that reflect the actual rather than the formal processes in which people are involved.
People often find it very difficult to articulate details of their work because it is second nature to them. They understand their own work but may not understand its relationship with other work in the organization. Social and organizational factors that affect the work but that are not obvious to individuals may only become clear when noticed by an unbiased observer.

Suchman used ethnography to study office work and found that the actual work practices were far richer, more complex and more dynamic than the simple models assumed by office automation systems. The difference between the assumed and the actual work was the most important reason why these office systems have had no significant effect on productivity. Other ethnographic studies for system requirements understanding have included work on air traffic control, underground railway control rooms, financial systems and various design activities.

Ethnography is particularly effective at discovering two types of requirements:

1. *Requirements that are derived from the way in which people actually work* rather than the way in which process definitions say they ought to work. For example, air traffic controllers may switch off an aircraft conflict alert system that detects aircraft with intersecting flight paths even though normal control procedures specify that it should be used. Their control strategy is designed to ensure that these aircraft are moved apart before problems occur and they find that the conflict alert alarm distracts them from their work.

2. *Requirements that are derived from cooperation and awareness of other people's activities*. For example, air traffic controllers may use an awareness of other controllers' work to predict the number of aircraft that will be entering their control sector. They then modify their control strategies depending on that predicted workload. Therefore, an automated ATC system should allow controllers in a sector to have some visibility of the work in adjacent sectors.

Ethnography may be combined with prototyping. The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer. He or she should then look for the answers to these questions during the next phase of the system study.
Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques. However, because of its focus on the end user, this approach is not appropriate for discovering organizational or domain requirements. Ethnographic studies cannot always identify new features that should be added to a system. Ethnography is not, therefore, a complete approach to elicitation on its own, and it should be used to complement other approaches, such as use-case analysis.

Q.4 a. Explain what is a software prototype. Identify three reasons for the necessity of developing a prototype during software development. Identify when a prototype needs to be developed.

Answer:
A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a

function, for example, may produce the desired results by using a table look-up instead of performing the actual computations. A prototype usually turns out to be a very crude version of the actual system.

Necessity of developing a prototype

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs:

- how screens might look like
- how the user interface would behave
- how the system would produce outputs

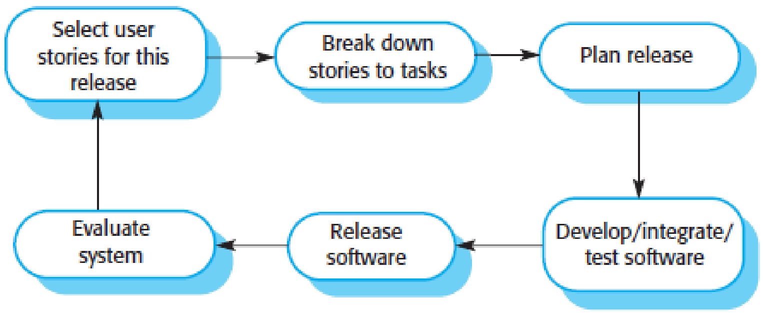
This is something similar to what the architectural designers of a building do; they show a prototype of the building to their customer. The customer can evaluate whether he likes it or not and the changes that he would need in the actual product. A similar thing happens in the case of a software product and its prototyping model.

Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.

Need for Development
A prototype can be developed when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with the product development. Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.

b. What is Extreme programming (XP)? What are the numbers of practices based on which extreme programming fits into the principles of agile method?

Answer:
Extreme programming :- (XP) is perhaps the best known and most widely used of the agile methods. The name was coined by Beck (Beck, 2000) because the approach was developed by pushing recognized good practice, such as iterative development, and customer involvement to 'extreme' levels.
In extreme programming, all requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks. Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system. There is a short time gap between releases of the system. Figure below illustrates the XP process to produce an increment of the system that is being developed.


Select user stories for this release
Break down stories to tasks
Plan release
Develop/integrate/ test software
Release software
Evaluate system

The extreme programming release cycle

Extreme programming involves a number of practices that fit into the principles of agile methods:
1. Incremental development is supported through small, frequent releases of the system and by an approach to requirements description based on customer stories or scenarios that can be the basis for process planning.
2. Customer involvement is supported through the full-time engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.
3. People, not process, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.
4. Change is supported through regular system releases, test-first development and continuous integration.
5. Maintaining simplicity is supported through constant refactoring to improve code quality and using simple designs that do not anticipate future changes to the system.

Q.5 a. Write the advantages and disadvantages of a shared-repository model?

Answer:
The advantages and disadvantages of a shared repository are as follows:

1. It is an efficient way to share large amounts of data. There is no need to transmit data explicitly from one sub-system to another.

2. However, sub-systems must agree on the repository data model. Inevitably, this is a compromise between the specific needs of each tool. Performance may be adversely affected by this compromise. It may be difficult or impossible to integrate new sub-systems if their data models do not fit the agreed schema.

3. Sub-systems that produce data need not be concerned with how that data is used by other sub-systems.

4. However, evolution may be difficult as a large volume of information is generated according to an agreed data model. Translating this to a new model will certainly be expensive; it may be difficult or even impossible.

5. Activities such as backup, security, access control and recovery from error are centralized. They are the responsibility of the repository manager. Tools can focus on their principal function rather than be concerned with these issues.

6. However, different sub-systems may have different requirements for security, recovery and backup policies. The repository model forces the same policy on all sub-systems.

7. The model of sharing is visible through the repository schema. It is straightforward to integrate new tools given that they are compatible with the agreed data model.

8. However, it may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralized repository, there may be problems with data redundancy and inconsistency.

b. What are the differences between the service model and the distributed object approach to distributed systems architectures?

Answer:
The differences between the service model and the distributed object approach to distributed systems architectures are:

• Services can be offered by any service provider inside or outside of an organization. Assuming these conform to certain standards, organizations can create applications by integrating services from a range of providers. For example, a manufacturing company can link directly to services provided by its suppliers.
• The service provider makes information about the service public so that any authorized user can use it. The service provider and the service user do not need to negotiate about what the service does before it can be incorporated in an application program.
• Applications can delay the binding of services until they are deployed or until execution. Therefore, an application using a stock price service (say) could dynamically change service providers while the system was executing.
• Opportunistic construction of new services is possible. A service provider may recognize new services that can be created by linking existing services in innovative ways.
• Service users can pay for services according to their use rather than their provision. Therefore, instead of buying an expensive component that is rarely used, the application writer can use an external service that will be paid for only when required.
• Applications can be made smaller (which is important if they are to be embedded in other devices) because they can implement exception handling as external services.
• Applications can be reactive and adapt their operation according to their environment by binding to different services as their environment changes.

c. Give two application uses for each of the following client-server architectures:

(i) Two-tier C/S architecture with thin client

Answer:

- Legacy system applications where separating application processing and data management is impractical.
- Computationally-intensive applications such as compilers with little or no data management.
- Data-intensive applications (browsing and querying) with little or no application processing.

(ii) Two-tier C/S architecture with fat client

Answer:

- Applications where application processing is provided by off-the-shelf software (e.g. Microsoft Excel) on the client.
- Applications where computationally-intensive processing of data (e.g. data visualisation) is required.
- Applications with relatively stable end-user functionality used in an environment with well-established system management.

(iii)Three-tier or multi-tier C/S architecture

Answer:

- Large-scale applications with hundreds or thousands of clients.
- Applications where both the data and the application are volatile.
- Applications where data from multiple sources are integrated.

Q.6 a. "Boehm and Abts" discuss four problems with COTS system integration. What are those four problems?

Answer:
The four problems discussed by Boehm and Abts with COTS system integration are as follows:

1. *Lack of control over functionality and performance* Although the published interface of a product may appear to offer the required facilities, these may not be properly implemented or may perform poorly. The product may have hidden operations that interfere with its use in a specific situation. Fixing these problems may be a priority for the COTS product integrator but may not be of real concern to the product vendor. Users may simply have to find workarounds to problems if they wish to reuse the COTS product.

2. *Problems with COTS system interoperability* It is sometimes difficult to get COTS products to work together because each product embeds its own assumptions about how it will be used. Garlan et al., reporting on their experience of trying to integrate four COTS products, found that three of these products were event-based but each used a different model of events and assumed that it had exclusive access to the event queue. As a consequence, the project required five times as much effort as originally predicted and the schedule grew to two years rather than the predicted six months.

3. *No control over system evolution* Vendors of COTS products make their own decisions on system changes in response to market pressures. For PC products in particular, new versions are often produced frequently and may not be compatible with all previous versions. New versions may have additional unwanted functionality, and previous versions may become unavailable and unsupported.

4. *Support from COTS vendors* The level of support available from COTS vendors varies widely. Because these are off-the-shelf systems, vendor support is particularly important when problems arise because developers do not have access to the source code and detailed documentation of the system. While vendors may commit to providing support, changing market and economic circumstances may make it difficult for them to deliver this commitment. For example, a COTS system vendor may decide to discontinue a product because of limited demand or may be taken over by another company that does not wish to support all of its current products.

b. What is design model? What are the two types of design models used to describe an object-oriented design?

Answer:
Design models show the objects or object classes in a system and, where appropriate, the relationships between these entities. Design models essentially are the design. They are the bridge between the requirements for the system and the system implementation. This means that there are conflicting requirements on these models. They have to be abstract so that unnecessary detail doesn't hide the relationships between them and the system requirements. However, they also have to include enough detail for programmers to make implementation decisions.
In general, you get around this conflict by developing models at different levels of detail. Where there are close links between requirements engineers, designers and programmers, then abstract models may be all that are required. Specific design decisions may be made as the system is implemented. When the links between system specifies, designers and programmers are indirect (e.g., where a system is being designed in one part of an organization but implemented elsewhere), then more detailed models may be required.
An important step in the design process, therefore, is to decide which design models that you need and the level of detail of these models. This depends on the type of system that is being developed. A sequential data processing system will be designed in a different way from an embedded real-time system, and different design models will therefore be used. There are very few systems where all models are necessary. Minimizing the

number of models that are produced reduces the costs of the design and the time required to complete the design process.

There are two types of design models that should normally be produced to describe an object-oriented design:

1. *Static models* describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalization relationships, uses/used-by relationships and composition relationships.
2. *Dynamic models* describe the dynamic structure of the system and show the interactions between the system objects (not the object classes). Interactions that may be documented include the sequence of service requests made by objects and the way in which the state of the system is related to these objects interactions.

c. Define component? How components are different from objects, taking account into account that component are generally developed using object-oriented approach?

Answer:

Council and Heineman define a *component* as:

"A software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."

This definition is essentially based on standards—a software unit that conforms to these standards is a component. Szyperski, however, does not mention standards in his definition of a component but focuses instead on the key characteristics of components:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Components are usually developed using an object-oriented approach, but they differ from objects in a number of important ways:

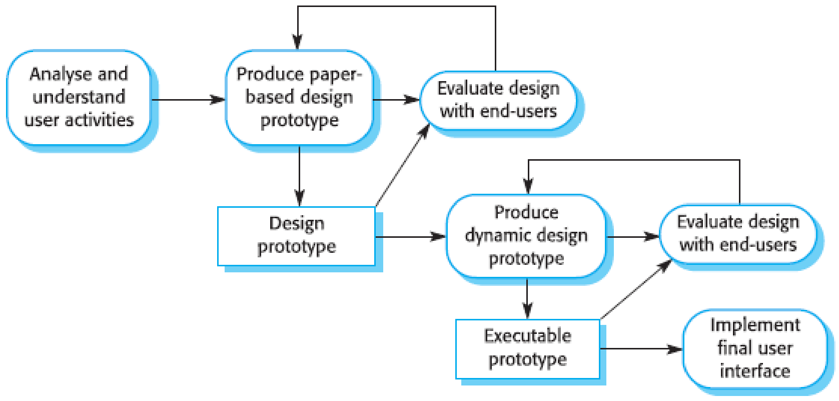
1. *Components are deployable entities* That is, they are not compiled into an application program but are installed directly on an execution platform. The methods and attributes defined in their interfaces can then be accessed by other components.
2. *Components do not define types* A class definition defines an abstract data type and objects are instances of that type. A component is an instance, not a template that is used to define an instance.
3. *Component implementations are opaque* Components are, in principle at least, completely defined by their interface specification. The implementation is hidden from component users. Components are often delivered as binary units so the buyer of the component does not have access to the implementation.
4. *Components are language-independent* Object classes have to follow the rules of a particular object-oriented programming language and, generally, can only interoperate with other classes in that language. Although components are usually implemented using object-oriented languages such as Java, you can implement them in non-object-oriented programming languages.

5. *Components are standardized* Unlike object classes that you can implement in any way, components must conform to some component model that constrains their implementation.

Q.7 a. What do you mean by UI design process? With the help of suitable figure, describe the core activities of UI design process.

Answer: User interface (UI) design is an iterative process where users interact with designers and interface prototypes to decide on the features, organisation and the look and feel of the system user interface. Sometimes, the interface is separately prototyped in parallel with other software engineering activities. More commonly, especially where iterative development is used, the user interface design proceeds incrementally as the software is developed. In both cases, however, before you start programming, you should have developed and, ideally, tested some paper-based designs.

The overall UI design process is illustrated in Figure below. **There are three core activities in this process:**


Analyse and understand user activities
Produce paper-based design prototype
Evaluate design with end-users
Design prototype
Produce dynamic design prototype
Evaluate design with end-users
Executable prototype
Implement final user interface

The UI design process

1. *User analysis* In the user analysis process, you develop an understanding of the tasks that users do, their working environment, the other systems that they use, how they interact with other people in their work and so on. For products with a diverse range of users, you have to try to develop this understanding through focus groups, trials with potential users and similar exercises.

2. *System prototyping* User interface design and development is an iterative process. Although users may talk about the facilities they need from an interface, it is very difficult for them to be specific until they see something tangible. Therefore, you have to develop prototype systems and expose them to users, who can then guide the evolution of the interface.

3. *Interface evaluation* Although you will obviously have discussions with users during the prototyping process, you should also have a more formalized evaluation activity where you collect information about the users’ actual experience with the interface.

b. What are the different approaches that can be used for user interface prototyping?

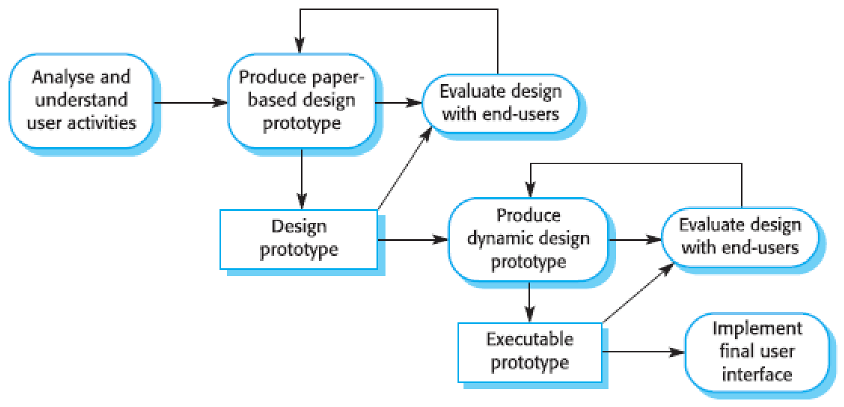
Answer:
There are three approaches that can be used for user interface prototyping:

1. *Script-driven approach* If you simply need to explore ideas with users, you can use a script-driven approach such as you'd find in Macromedia Director. In this approach, you create screens with visual elements, such as buttons and menus, and associate a script with these elements. When the user interacts with these screens, the script is executed and the next screen is presented, showing them the results of their actions. There is no application logic involved.

2. *Visual programming languages* Visual programming languages, such as Visual Basic, incorporate a powerful development environment, access to a range of reusable objects and a user-interface development system that allows interfaces to be created quickly, with components and scripts associated with interface objects.

3. *Internet-based prototyping* These solutions, based on web browsers and languages such as Java, offer a ready-made user interface. You add functionality by associating segments of Java programs with the information to be displayed. These segments (called applets) are executed automatically when the page is loaded into the browser. This approach is a fast way to develop user interface prototypes, but there are inherent restrictions imposed by the browser and the Java security model.
Prototyping is obviously closely associated with interface evaluation. Formal evaluation is unlikely to be cost-effective for early prototypes, so what you are trying to achieve at this stage is a 'formative evaluation' where you look for ways in which the interface can be improved. As the prototype becomes more complete, you can use systematic evaluation techniques User interface (UI) design is an iterative process where users interact with designers and interface prototypes to decide on the features, organisation and the look and feel of the system user interface. Sometimes, the interface is separately prototyped in parallel with other software engineering activities. More commonly, especially where iterative development is used, the user interface design proceeds incrementally as the software is developed. In both cases, however, before you start programming, you should have developed and, ideally, tested some paper-based designs.

The overall UI design process is illustrated in Figure below. **There are three core activities in this process:**


Analyse and understand user activities
Produce paper-based design prototype
Evaluate design with end-users
Design prototype
Produce dynamic design prototype
Evaluate design with end-users
Executable prototype
Implement final user interface

The UI design process

1. *User analysis* In the user analysis process, you develop an understanding of the tasks that users do, their working environment, the other systems that they use, how they interact with other people in their work and so on. For products with a diverse range of users, you have to try to develop this understanding through focus groups, trials with potential users and similar exercises.

2. *System prototyping* User interface design and development is an iterative process. Although users may talk about the facilities they need from an interface, it is very difficult for them to be specific until they see something tangible. Therefore, you have to develop prototype systems and expose them to users, who can then guide the evolution of the interface.

3. *Interface evaluation* Although you will obviously have discussions with users during the prototyping process, you should also have a more formalized evaluation activity where you collect information about the users' actual experience with the interface.

c. What is fault tolerance? What are the different aspects related to fault-tolerance? Describe the two approaches to software fault tolerance.

Answer:
A fault-tolerant system can continue in operation after some system faults have occurred. The fault-tolerance mechanisms in the system ensure that these system faults do not cause system failure. You may need fault tolerance in situations where system failure could cause a catastrophic accident or where a loss of system operation would cause large economic losses. For example, the computers in an aircraft must carry on working until the aircraft has landed; the computers in an air traffic control system must be continuously available while planes are in the air.

There are four aspects to fault-tolerance:
1. *Fault detection* The system must detect a fault that could lead to a system failure. Generally, this involves checking that the system state is consistent.

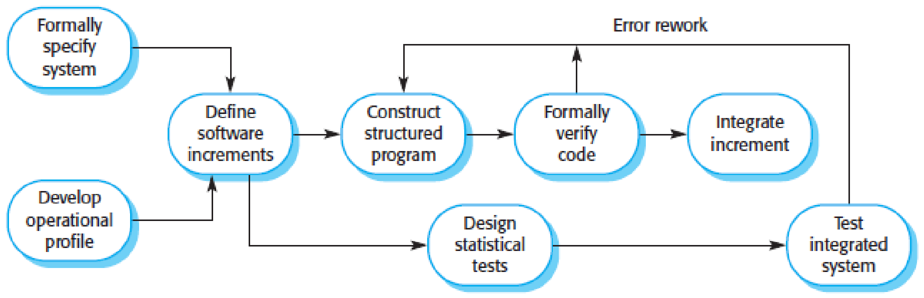
2. *Damage assessment* The parts of the system state that have been affected by the fault must be detected.
3. *Fault recovery* The system must restore its state to a known 'safe' state. This may be achieved by correcting the damaged state (forward error recovery) or by restoring the system to a known 'safe' state (backward error recovery).
4. *Fault repair* This involves modifying the system so that the fault does not recur. However, many software faults manifest themselves as transient states. They are due to a peculiar combination of system inputs. No repair is necessary and normal processing can resume immediately after fault recovery.

The two approaches to software fault tolerance are:
1. *N-version programming* Using a common specification, the software system is implemented in a number of versions by a number of teams. These versions are executed in parallel on separate computers. Their outputs are compared using a voting system and inconsistent outputs or outputs that are not produced in time are rejected. At least three versions of the system should be available so that two versions should be consistent in the event of a single failure. This is the most commonly used approach to software fault tolerance. It has been used in railway signaling systems, in aircraft systems and in reactor protection systems.

2. *Recovery blocks* In this approach, each program component includes a test to check that the component has executed successfully. It also includes alternative code that allows the system to back up and repeat the computation if the test detects a failure. The implementations are deliberately different interpretations of the same specification. They are executed in sequence rather than in parallel, so replicated hardware is not required. In N-version programming, the implementations may be different but it is not uncommon for two or more development teams to choose the same algorithms to implement the specification.

Q.8 a. What is Cleanroom software development approach? Discuss the key strategies on which the Cleanroom software development is based. Also describe the different teams involved when the Cleanroom process is used for large system development.

Answer:
Cleanroom software development:- is a software development philosophy that uses formal methods to support rigorous software inspection. A model of the Cleanroom process is shown in figure below. The objective of this approach to software development is zero-defect software. The name 'Cleanroom' was derived by analogy with semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere. Cleanroom development is particularly relevant to this chapter because it has replaced the unit testing of system components by inspections to check the consistency of these components with their specifications.



The Cleanroom development process

Use of the Cleanroom approach has generally led to software with very few errors. Cobb and Mills discuss several successful Cleanroom development projects that had a uniformly low failure rate in delivered systems. The costs of these projects were comparable with other projects that used conventional development techniques. The approach to incremental development in the Cleanroom process is to deliver critical customer functionality in early increments. Less important system functions are included in later increments. The customer therefore has the opportunity to try these critical increments before the whole system has been delivered. If requirements problems are discovered, the customer feeds back this information to the development team and requests a new release of the increment.

Rigorous program inspection is a fundamental part of the Cleanroom process. A state model of the system is produced as a system specification. This is refined through a series of more detailed system models to an executable program. The approach used for development is based on well-defined transformations that attempt to preserve the correctness at each transformation to a more detailed representation. At each stage, the new representation is inspected, and mathematically rigorous arguments are developed that demonstrate that the output of the transformation is consistent with its input.

<u>The Cleanroom approach to software development is based on five key strategies:</u>

1. *Formal specification* The software to be developed is formally specified. A state transition model that shows system responses to stimuli is used to express the specification.
2. *Incremental development* The software is partitioned into increments that are developed and validated separately using the Cleanroom process. These increments are specified, with customer input, at an early stage in the process.
3. *Structured programming* Only a limited number of control and data abstraction constructs are used. The program development process is a process of stepwise refinement of the specification. A limited number of constructs are used and the aim is to systematically transform the specification to create the program code.
4. *Static verification* The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.
5. *Statistical testing of the system* The integrated software increment is tested statistically to determine its reliability. These statistical tests are based on an operational profile, which is developed in parallel with the system specification as shown in figure.

There are three teams involved when the Cleanroom process is used for large system development:

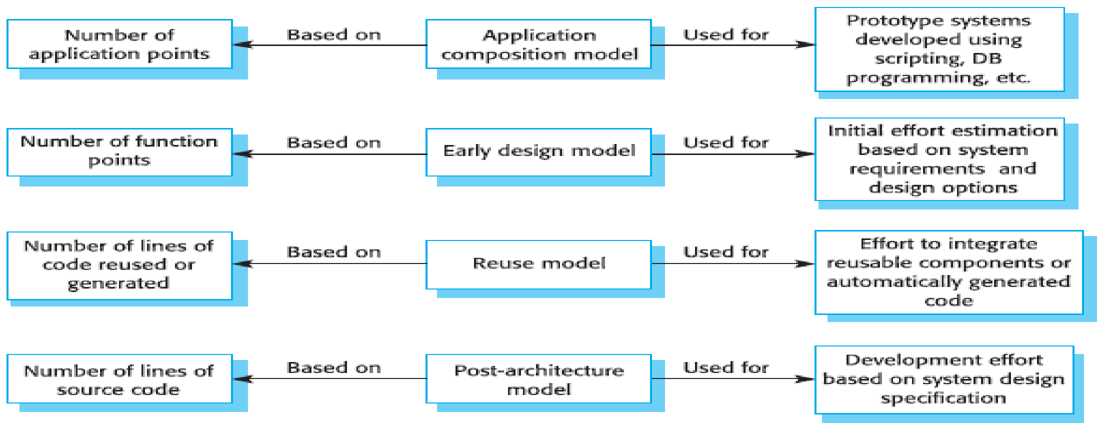
1. *The specification team* This group is responsible for developing and maintaining the system specification. This team produces customer-oriented specifications (the user requirements definition) and mathematical specifications for verification. In some cases, when the specification is complete, the specification team also takes responsibility for development.

2. *The development team* This team has the responsibility of developing and verifying the software. The software is not executed during the development process. A structured, formal approach to verification based on inspection of code supplemented with correctness arguments is used.

3. *The certification team* This team is responsible for developing a set of statistical tests to exercise the software after it has been developed. These tests are based on the formal specification. Test case development is carried out in parallel with software development. The test cases are used to certify the software reliability. Reliability growth models may be used to decide when to stop testing.

b. With the help of a figure, describe COCOMO II sub-model and explain where they are used.

Answer:
The COCOMO II model recognizes different approaches to software development such as prototyping, development by component composition and use of database programming. COCOMO II supports a spiral model of development and embeds several sub-models that produce increasingly detailed estimates. These can be used in successive rounds of the development spiral. The sub-models that are part of the COCOMO II model are shown in figure below:


Number of application points
Based on
Application composition model
Used for
Prototype systems developed using scripting, DB programming, etc.
Number of function points
Based on
Early design model
Used for
Initial effort estimation based on system requirements and design options
Number of lines of code reused or generated
Based on
Reuse model
Used for
Effort to integrate reusable components or automatically generated code
Number of lines of source code
Based on
Post-architecture model
Used for
Development effort based on system design specification

The COCOMO II models

1. *An application-composition model* This assumes that systems are created from reusable components, scripting or database programming. It is designed to make estimates of prototype development. Software size estimates are based on application points, and a simple size/productivity formula is used to estimate the effort required. Application points are the same as object points but the name was changed to avoid confusion with objects in object-oriented development.

2. *An early design model* This model is used during early stages of the system design after the requirements have been established. Estimates are based on function points, which are then converted to number of lines of source code. The formula follows the standard form with a simplified set of seven multipliers.

3. *A reuse model* This model is used to compute the effort required to integrate reusable components and/or program code that is automatically generated by design or program translation tools. It is usually used in conjunction with the post-architecture model.

4. *A post-architecture model* Once the system architecture has been designed, a more accurate estimate of the software size can be made. Again this model uses the standard formula for cost estimation discussed above. However, it includes a more extensive set of 17 multipliers reflecting personnel capability and product and project characteristics.

Of course, in large systems, different parts may be developed using different technologies, and you may not have to estimate all parts of the system to the same level of accuracy. In such cases, you can use the appropriate sub-model for each part of the system and combine the results to create a composite estimate.

Q.9 a. What is static product metrics? Describe static product metrics that have used for quality assessment.

Answer:
Static metrics :- are collected by measurements made of representations of the system such as the design, program or documentation. Static metrics help to assess the complexity, understandability and maintainability of a software system. Static metrics, on the other hand, have an indirect relationship with quality attributes. A large number of these metrics have been proposed, and many experiments have tried to derive and validate the relationships between these metrics and system complexity, understandability and maintainability. Several static product metrics that have been used for quality assessment are explained below. Of these, program or component length and control complexity seem to be the most reliable predictors of understandability, system complexity and maintainability.

Fan-in/Fan-out:-

- *Fan-in* is a measure of the number of functions or methods that call some other function or method (say X). *Fan-out* is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.

Length of code:-

- This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components.

Cyclomatic complexity

- This is a measure of the control complexity of a program. This control complexity may be related to program understandability.

Length of identifiers

- This is a measure of the average length of distinct identifiers in a program. The longer the identifiers, the more likely they are to be meaningful and hence the more understandable the program.

Depth of conditional nesting

- This is a measure of the depth of nesting of if-statements in a program. Deeply nested if statements are hard to understand and are potentially error-prone.

Fog index

- This is a measure of the average length of words and sentences in documents. The higher the value for the Fog index, the more difficult the document is to understand.

b. What is Software configuration management? Explain major tasks and important concepts of SCM

Answer:

Software configuration management **(SCM)** is an umbrella activity that is applied throughout the software process. Because change can occur at any time, SCM activities are developed to (1) identify change, (2) control change, (3) ensure that change is being properly implemented, and (4) report changes to others who may have an interest. The output of the software process is information that may be divided into three broad categories: (1) computer programs (both source level and executable forms); (2) documents that describe the computer programs (targeted at both technical practitioners and users), and (3) data (contained within the program or external to it). The items that comprise all information produced as part of the software process are collectively called a *software configuration.*

As the software process progresses, the number of *software configuration items* (SCIs) grows rapidly. A *System Specification* spawns a *Software Project Plan* and *Software Requirements Specification* (as well as hardware related documents). These in turn spawn other documents to create a hierarchy of information. If each SCI simply spawned other SCIs, little confusion would result. Unfortunately, another variable enters the process—*change.* Change may occur at any time, for any reason.

Major SCM tasks and important concept are explained below.
Baselines: Change is a fact of life in software development. Customers want to modify requirements. Developers want to modify the technical approach. Managers want to modify the project strategy. Why all this modification? The answer is really quite simple.
As time passes, all constituencies know more (about what they need, which approach would be best, how to get it done and still make money). This additional knowledge is the driving force behind most changes and leads to a statement of fact that is difficult for many software engineering practitioners to accept: Most changes are justified!

A *baseline* is a software configuration management concept that helps us to control change without seriously impeding justifiable change. The IEEE defines a baseline as:
A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.
In the context of software engineering, a baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that is obtained through a formal technical review . For example, the elements of a *Design Specification* have been documented and reviewed. Errors are found and corrected. Once all parts of the specification have been reviewed, corrected and then approved, the *Design Specification* becomes a baseline. Further changes to the program architecture (documented in the *Design Specification*) can be made only after each has been evaluated and approved. Although baselines can be defined at any level of detail, Software

Configuration Items
A software configuration item may be defined as information that is created as part of the software engineering process. In the extreme, a SCI could be considered to be a single section of a large specification or one test case in a large suite of tests. More realistically, an SCI is a document, a entire suite of test cases, or a named program component (e.g., a C++ function or an Ada package).

In addition to the SCIs that are derived from software work products, many software engineering organizations also place software tools under configuration control. That is, specific versions of editors, compilers, and other CASE tools are "frozen" as part of the software configuration. Because these tools were used to produce documentation, source code, and data, they must be available when changes to the software configuration are to be made. Although problems are rare, it is possible that a new version of a tool (e.g., a compiler) might produce different results than the original version. For this reason, tools, like the software that they help to produce, can be baselined as part of a comprehensive configuration management process.

In reality, SCIs are organized to form *configuration objects* that may be catalogued in the project database with a single name. A configuration object has a name, attributes, and is "connected" to other objects by relationships. The configuration objects, **Design Specification, data model, component, source code** and **Test Specification** are each defined separately. However, each of the objects is related to the others.

TEXT BOOK

Software Engineering, Ian Sommerville, 7th Edition, Pearson Education, 2004.