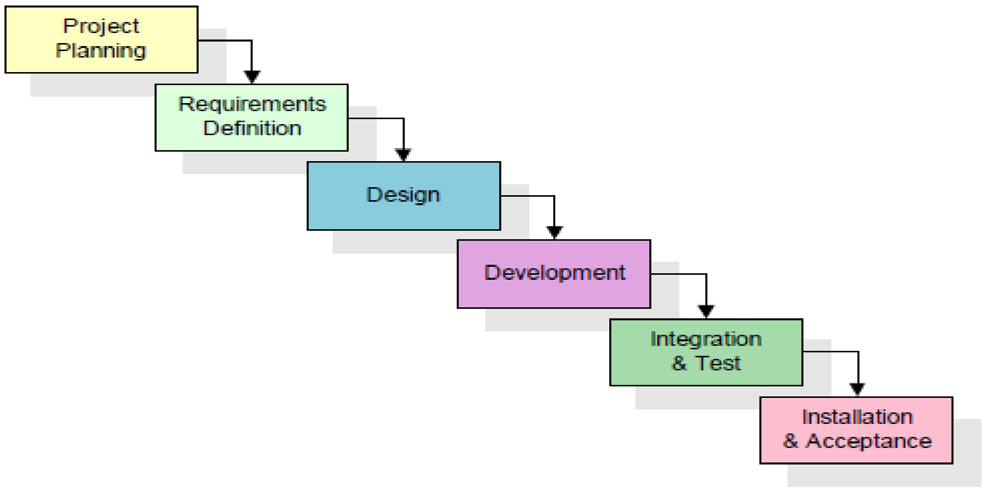
Q.2 a. With the help of a suitable diagram, explain the software lifecycle. (8)

Answer:



The relationship of each stage to the others can be roughly described as a waterfall, where the outputs from a specific stage serve as the initial inputs for the following stage.

During each stage, additional information is gathered or developed, combined with the inputs, and used to produce the stage deliverables. It is important to note that the additional information is restricted in scope; "new ideas" that would take the project in directions not anticipated by the initial set of high-level requirements are not incorporated into the project. Rather, ideas for new capabilities or features that are out-of-scope are preserved for later consideration.

After the project is completed, the Primary Developer Representative (PDR) and Primary End-User Representative (PER), in concert with other customer and development team personnel develop a list of recommendations for enhancement of the current software.

b. Explain the CASE toolset architecture. (4)

Answer:
Computer-aided software engineering (CASE) is the scientific application of a set of tools and methods to a software system which is meant to result in high-quality, defect-free, and maintainable software products.[1] It also refers to methods for the development of information systems together with automated tools that can be used in the software development process

All aspects of the software development life cycle can be supported by software tools, and so the use of tools from across the spectrum can, arguably, be described as CASE; from project management software through tools for business and functional analysis, system design, code storage, compilers, translation tools, test software, and so on.

However, tools that are concerned with analysis and design, and with using design information to create parts (or all) of the software product, are most frequently thought of as CASE tools. CASE applied, for instance, to a database software product, might normally involve:

c. Describe briefly the process of risk management. (4)

Q.3 a. What do you mean by requirement engineering? Explain its activities in details. (8)

Answer:

Requirements engineering

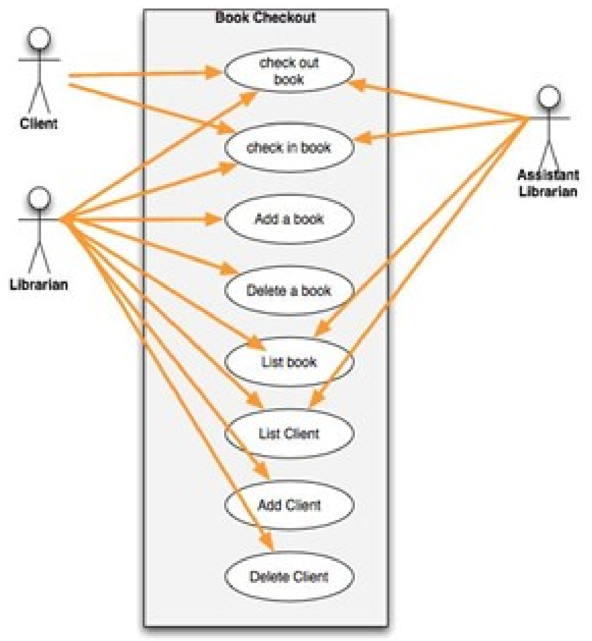
Systematic requirements analysis is also known as requirements engineering. It is sometimes referred to loosely by names such as requirements gathering, requirements capture, or requirements specification. The term requirements analysis can also be applied specifically to the analysis proper, as opposed to elicitation or documentation of the requirements, for instance. Requirements Engineering can be divided into discrete chronological steps:

1. Requirements elicitation,
2. Requirements analysis and negotiation,
3. Requirements specification,
4. System modeling,
5. Requirements validation,
6. Requirements management.

Requirement engineering is a systems engineering and software engineering that is concerned with determining the goals, functions, and constraints of hardware and software systems." In some life cycle models, the requirement engineering process begins with a feasibility study activity, which leads to a feasibility report. If the feasibility study suggests that the product should be developed, then requirement analysis can begin. If requirement analysis precedes feasibility studies, which may foster outside the box thinking, then feasibility should be determined before requirements are finalized

b. Draw the Use Case Diagram for a Library Management System. (4)

Answer:



c. Differentiate between the functional and non functional requirements. (4)

Answer:
Basically, functional requirements directly support the user requirements by describing the "processing" of the information or materials as inputs or outputs. Nonfunctional requirements generally support all users in that they describe the business standards and the business environment, as well as the overall user's experience (user attributes).

Functional	**Nonfunctional**
Product features	Product properties
Describe the work that is done	Describe the character of the work
Describe the actions with which the work is concerned	Describe the experience of the user while doing the work
Characterized by verbs	Characterized by adjectives

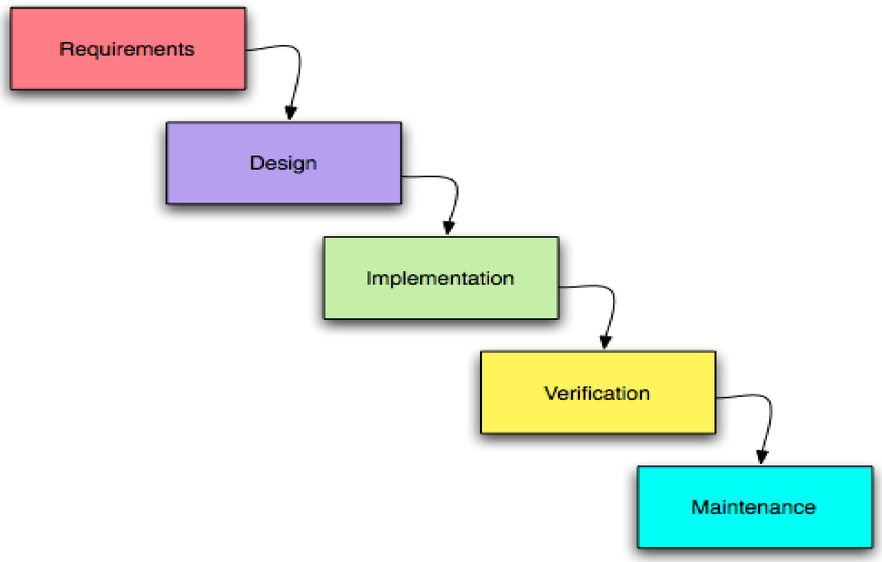
The functional requirements specify what the product must do. They relate to the actions.

Q.4 a. Explain the RAD technique in detail. (8)

Answer:
Rapid application development (RAD) refers to a type of software development methodology that uses minimal planning in favor of rapid prototyping. The "planning" of software developed using RAD is interleaved with writing the software itself. The lack of extensive pre-planning generally allows software to be written much faster, and makes it easier to change requirements.
"Rapid Application
Development (RAD) is a development lifecycle designed to give much faster development and higher-quality results than those achieved with the traditional lifecycle. It is designed to take the maximum advantage of powerful development software that has evolved recently."
RAD (rapid application development) is a concept that products can be developed faster and of higher quality through:
Gathering requirements using workshops or focus groups
Prototyping and early, reiterative user testing of designs
The re-use of software components
A rigidly paced schedule that defers design improvements to the next product version
Less formality in reviews and other team communication
Rapid Application Development encouraged the creation of quick-and-dirty prototype-style software which fulfilled most of the user's requirements but not necessarily all. Development would take place in a series of short cycles, called time boxes, each of which would deepen the functionality of the application a little more. Features to be implemented in each time box were agreed in advance and this game plan rigidly adhered to. The strong emphasis on this point came from unhappy experience with other development practices in which new requirements would tend to be added as the project was evolving, caused massive chaos and disrupting the already carefully prepared plans and development schedules. Rapid Application Development methodology advocated that development be undertaken by small, experienced teams using CASE (Computer Aided Software Engineering) tools to enhance their productivity.

b. Explain the various stages of software specification and its interface with the design process. (4)

Answer:
A software development process, also known as a software development lifecycle, is a structure imposed on the development of a software product. Similar terms include software life cycle and *software process*. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process. Some people consider a lifecycle model a more general term and a software development process a more specific term. For example, there are many specific software development processes that 'fit' the spiral lifecycle model.



c. What do you mean by prototype of a software? What are the benefits of making a prototype in software development? (4)

Answer:
The idea behind software prototyping is to allow people who have some 'stake' or interest in a system to 'test drive' designs, rather than having to interpret those designs based on some other means. Though not all prototypes are interactive, the most useful application of prototyping is based upon providing a similation of some behaviour and functionality.
The conventional purpose of a prototype is to allow users of the software to evaluate developers' proposals for the design of the eventual product by actually trying them out, rather than having to interpret and evaluate the design based on descriptions.

Benefits of Prototyping

Reduced time and costs:
Prototyping can improve the quality of requirements and specifications provided to developers. Because changes cost exponentially more to implement as they are detected later in development,

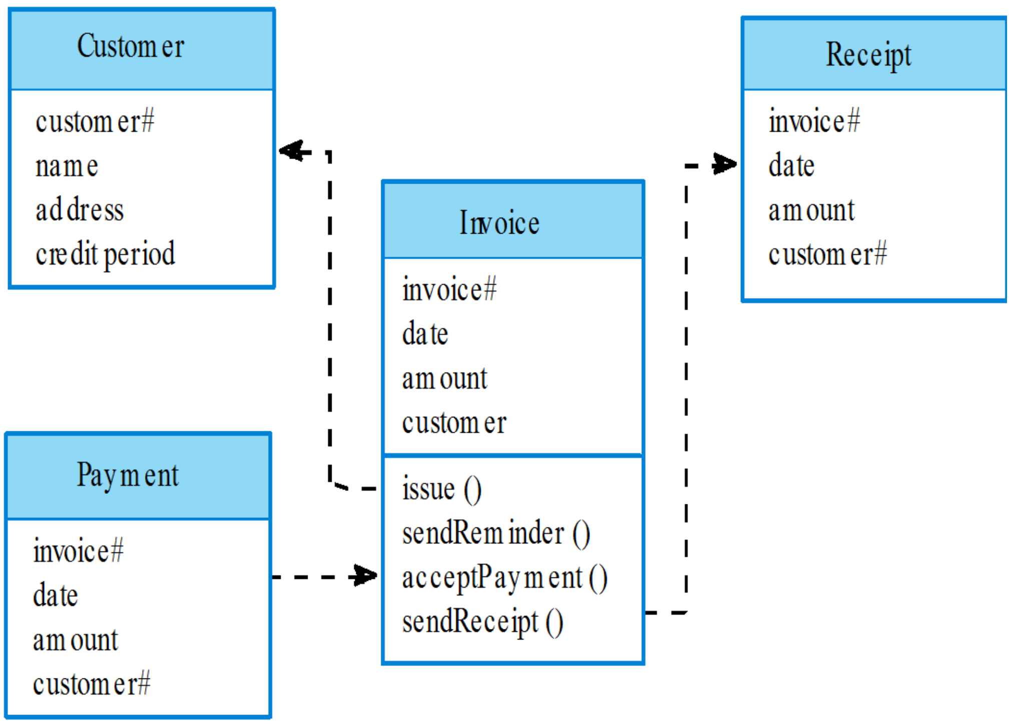
the early determination of what the user really wants can result in faster and less expensive software.
Improved and increased user involvement:
Prototyping requires user involvement and allows them to see and interact with a prototype allowing them to provide better and more complete feedback and specifications.
Feedback
The main benefit of software prototyping is in obtaining feedback on a proposed design at an early stage in a project. This feedback can be used to help refine project requirements specifications, establish usability and gain stakeholder support
Visualisation of an design
By creating prototype models and simulations we can improve our understanding of what is to be developed. Once a prototype has been created for a project, it is easy for everyone to gain a 'sneak preview' of what the end system will look like and what it will do.

Q.5 a. What are the likely limits on the scalability of a distributed system? Explain. (5)

Answer:

b. With the help of a diagram, explain object oriented architectural model of an invoice processing system. (6)

Answer:
System is structured as a set of loosely coupled objects with well defined interfaces. Object-oriented decomposition identifies object classes, attributes, and operations .Objects are instantiated from classes, and some control model coordinates how the objects operate.
Invoice Processing System



•Advantages:
–Loose coupling ensures that changes in one object class does not affect other objects
–Since objects tend to reflect real-world entities, object models are easy to understand

Disadvantages:
–Changes to the interface of an object have an impact on other users of the object
–Complex entities may be difficult to represent as objects

c. Explain the use of different client-server architectures. (5)

Answer:
Distributed system model organized as a set of services provided by servers, and clients that access and use those services

Components:
–Servers: Provide services to other sub-systems (i.e. print and file servers)
–Clients: Call on services provided by servers
–Network: Allows clients to access services

Advantages
–Data distribution is straightforward
–Makes effective use of networked systems
–Easy to add new servers or upgrade existing servers

Disadvantages
–No shared data model, sub-systems may use different data organization, leading to inefficient data interchange
–Redundant management in each server
–No central registry of names and services, making it hard to find out what servers and services are available

Q.6 a. What are the benefits and problems of software reuse? (8)

Answer:
Benefits of reuse
Increased dependability
Reused software, that has been tried and tested in working systems, should be m ore dependable than new software. The initial use of the software reveals any design and implementation faults. These are then fixed, thus reducing the number of failures when the software is reused.

Reduced process risk
If software exists, there is less uncertainty in the costs of reusing that software than in the costs of development. This is an important factor for project management as it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as sub-systems are reused.

Effective use of specialists

Instead of application specialists doing the same work on different projects, these specialists can develop reusable software that encapsulate their knowledge.

Standards compliance
Some standards, such as user interface standards, can be implemented as a set of standard reusable components. For example, if menus in a user interfaces are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability as users are less likely to make mistakes when presented with a familiar interface.

Accelerated development
Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time should be reduced.

Reuse problems
Increased maintenance Costs
If the source code of a reused software system or component is not available then maintenance costs may be increased as the reused elements of the system may become increasingly incompatible with system changes.

Lack of tool support
CASE toolsets may not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account.

Not-invented-here Syndrome
Some software engineers sometimes prefer to re-write components as they believe that they can improve on the reusable component. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.

Creating and maintaining a component library
Populating a reusable component library and ensuring the software developers can use this library can be expensive. Our current techniques for classifying, cataloguing and retrieving software components are immature.

b. Explain the basic elements of a component model. (4)

Answer:
Software component technology, which is based on building software systems from reusable components, has attracted attention because it is capable of reducing developmental costs. In a narrow sense, a software component is defined as a unit of composition, and can be independently exchanged in the form of an object code without source codes. The internal structure of the component is not available to the public.

The characteristics of the component-based development are the following:

- Black-box reuse
- Reactive-control and component's granularity
- Using RAD (rapid application development) tools
- Contractually specified interfaces
- Introspection mechanism provided by the component systems
- Software component market (CALS)

c. Write a note on "Component Based Software Engineering". (4)

Answer:
Component-based software engineering (CBSE) (also known as component-based development (CBD)) is a branch of software engineering which emphasizes the separation of concerns in respect of the wide-ranging functionality available throughout a given software system. This practice aims to bring about an equally wide-ranging degree of benefits in both the short-term and the long-term for the software itself and for organizations that sponsor such software.

Software engineers regard components as part of the starting platform for service-orientation. Components play this role, for example, in Web Services, and more recently, in Service-Oriented Architecture (SOA) - whereby a component is converted into a service and subsequently inherits further characteristics beyond that of an ordinary component.

Components can produce events or consume events and can be used for event driven architecture (EDA).

Q.7 a. What are the various characteristics of dependable processes? (8)

Answer:
Dependable process characteristics

Documentable	The process should have a defined process model that sets out the activities in the process and the documentation that is to be produced during these activities.
Standardised	A comprehensive set of software development standards that define how the software is to be produced and documented should be available.
Auditable	The process should be understandable by people apart from process participants who can check that process standards are being followed and make suggestions for process improvement.
Diverse	The process should include redundant and diverse verification and validation activities.
Robust	The process should be able to recover from failures of individual process activities.

b. Explain the general principles of user interface design. (8)

Answer:

1. The structure principle. Your design should organize the user interface purposefully, in meaningful and useful ways based on clear, consistent models that are apparent and recognizable to users, putting related things together and separating unrelated things, differentiating dissimilar things and making similar things resemble one another. The structure principle is concerned with your overall user interface architecture.
2. The simplicity principle. Your design should make simple, common tasks simple to do, communicating clearly and simply in the user's own language, and providing good shortcuts that are meaningfully related to longer procedures.
3. The visibility principle. Your design should keep all needed options and materials for a given task visible without distracting the user with extraneous or redundant information. Good designs don't overwhelm users with too many alternatives or confuse them with unneeded information.
4. The feedback principle. Your design should keep users informed of actions or interpretations, changes of state or condition, and errors or exceptions that are relevant and of interest to the user through clear, concise, and unambiguous language familiar to users.
5. The tolerance principle. Your design should be flexible and tolerant, reducing the cost of mistakes and misuse by allowing undoing and redoing, while also preventing errors wherever possible by tolerating varied inputs and sequences and by interpreting all reasonable actions reasonable.
6. The reuse principle. Your design should reuse internal and external components and behaviors, maintaining consistency with purpose rather than merely arbitrary consistency, thus reducing the need for users to rethink and remember.

Q.8 a. Discuss the differences between verification and validation, and explain why validation is particularly a difficult process? (8)

Answer:

Verification is quality control in which we only take corrective actions and Verification is quality assurance which involves preventive actions. Software companies combine both verification and validation in their software quality assurance departments to produce high quality software. Software Quality Assurance is an essential part of Software Development Life Cycle and Software Testing comes under Software Quality Assurance. Validation and Verification are part of Software Quality Assurance and Software Testing.

To understand the basic concept of Verification and Validation, let's discuss them separately. Verification is a process of ensuring that every product is designed according to the user requirements. Reviews and meetings are conducted to evaluate different documents. These include requirements, specifications, plans and code. Evaluation of all these documents is done via checklists, walkthroughs and inspections. After completion of verification we move one step further and start validation.

Validation involves the actual testing where we check the program design to see if it is according to the intended design. Validation is performed both by software developers and software testers. Unit testing comes under validation and it is performed by Software developers in order to check that their code is working correctly. Alpha and beta testing also comes under validation.

b. Describe two metrics that have been used to measure programmer productivity. (4)

Answer:
Productivity in Lines of Code Per Staff Month
LOC did not include data declarations, comments, or any other lines that did not result in object code.
Disadvantage of using LOC
• It is language dependent
• They also reflect what the system is rather than what it does.
• Measuring software size in terms of LOC is analogous to measuring a car stereo by the number of resistors, capacitors and integrated circuits involved in it production.
Function Count
Function Point Analysis
The principle of Function Point Analysis is that a system is decomposed into functional units

The five function units are divided in two categories:
• Data function types
– Internal Logical files
– External Interface files
• Transactional Function Types
– External Inputs
– External Outputs
– External Inquiry

c. Differentiate between the structural testing and Functional testing. (4)

Answer:
Functional testing (also known as black-box testing), is a software testing approach in which: the tester will have a user perspective in mind, not knowing and doesn't mind how the program works. Input and output are the only things that matter.
The tester acts as if he/she is the final user of the program.

On the other hand, structural testing (also known as white-box testing), is a software testing approach in which: the tester will have a developer perspective in mind, knowing how the program works behind the scene, such that the test will test all algorithm paths in the program. Everything does matter. The tester acts as a developer of the program who knows the internal structure of the program very well.

Q.9 a. What are the different levels of CMM? Explain each in detail. (8)

Answer:
There are many different applications of the Capability Maturity Models. Some of these models target software development, staffing, etc. In this series I will focus on the Systems Engineering Capability Maturity ModelSM (CMM). The CMM is service marked by the Software Engineering

Institute (SEI) of Carnegie Mellon University and was developed by the SEI, the Department of Defense and a host of other entities.

The CMM is designed to be an easy to understand methodology for ranking a company's IT related activities. The CMM has six levels 0 – 5 (see illustration below):

- Level 0 – Not Performed
- Level 1 – Performed Informally
- Level 2 – Planned and Tracked
- Level 3 – Well-Defined
- Level 4 – Quantitatively Controlled
- Level 5 – Continuously Improving

b. What is SQA? Discuss the different software quality factors. (8)

Answer:

Software quality assurance (SQA) consists of a means of monitoring the software engineering processes and methods used to ensure quality. The methods by which this is accomplished are many and varied, and may include ensuring conformance to one or more standards, such as ISO 9000 or a model such as CMMI

A software quality factor is a non-functional requirement for a software program which is not called up by the customer's contract, but nevertheless is a desirable requirement which enhances the quality of the software program. Note that none of these factors are binary; that is, they are not "either you have it or you don't" traits. Rather, they are characteristics that one seeks to maximize in one's software to optimize its quality. So rather than asking whether a software product "has" factor *x*, ask instead the *degree* to which it does (or does not).

Some software quality factors are listed here:

Understandability

Clarity of purpose - This goes further than just a statement of purpose; all of the design and user documentation must be clearly written so that it is easily understandable. This is obviously subjective in that the user context must be taken into account: for instance, if the software product is to be used by software engineers it is not required to be understandable to the layman.

Completeness - Presence of all constituent parts, with each part fully developed. This means that if the code calls a subroutine from an external library, the software package must provide reference to that library and all required parameters must be passed. All required input data must also be available.

Conciseness -Minimization of excessive or redundant information or processing. This is important where memory capacity is limited, and it is generally considered good practice to keep lines of code to a minimum. It can be improved by replacing repeated functionality by one subroutine or function which achieves that functionality. It also applies to documents.

TEXT BOOK

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