SE (unit 1) / (unit 2 upto engineering requirements)

1	①	Suggest any 5 essential attributes of good software and explain with suitable examples.
almost not there	②	Write functional and non functional and domain requirements for Jewellery mart management system.
X	③	Write a neat diagram. briefly explain the system engineering process.
	④	Explain how to accomodate various software process models in Incremental delivery model.
II	⑤	Explain the Boehm's spiral model of software process in detail. mention benefits of this model over the others.
II	⑥	Explain how both waterfall model an the prototyping model can be accomodated in the incrementary model of the software process.
	⑦	Describe the essence of SE practice.
	⑧	Briefly explain various specialized process models.
	⑨	12 agility principles.
	⑩	Describle overall flow of SCRUM process with a neat diagram
	⑩	problem that occur when requirements must be elicitated from different customers.
	⑫	Define requirement engineering. List and explain seven distinct tasks of requirement engineering.
	⑬	Analyze the generic process model for the software development process with a neat representation.
	⑭	Explain best practices of extreme programming method (wt is XP)
	⑮	With neat diagram explain process flow of software process
1	⑯	Suggest who might be stakeholders in a Hospital management system. Explain why it is almost inevitable that the requirements of different stakeholders will conflict in some way.
	⑰	What are the Key challenges facing software engineering (explain with examples)
	⑱	Explain rational unified process model for developing business applications.

(19) compare and contrast waterfall and evolutionary model of software development.

(20) Explain component based software engineering method.

(21) Explain the ethical and professional responsibilities of a software engineer.

SE(SOFTWARE ENGINEERING)

Important Questions and Answers

KRUTHIK B

1DA19CS077

B SECTION

UNIT 1 AND 2(UPTO ENGINEERING REQUIREMENTS)

1)

1)**Functionality:** A good software must be able to do what it was designed to do. The software requirements must guide the design and implementation of the software.

2)**Usability:** The software must be usable; the users must not find it difficult to figure out how a good software works. A good software is user-centered and user-friendly.

3)**Efficiency:** Efficiency means that perform it's operations with minimal time and processing power. A good software uses the least amount of processing power and memory needed to achieve the desired result.

4)**Maintainability:** A good software must evolve with changing requirements.

5)**Security:** A good software must be secure. It should not cause physical or economic damage in the event of a system failure. Unauthorized users must not be allowed access to the system.

6)**Reliability:** A reliable system will rarely fail, and even when it does fail, there are recovery mechanisms in the software to recover from the failure with minimal losses.

(Examples just give the software that u r aware off and explain it in a simple way)

KRUTHIK B(1DA19CS077 B SECTION)

17)

Key challenges facing software engineering:

1.Developing systems that are energy-efficient. This makes them more usable on low power mobile devices and helps reduce the overall carbon footprint of IT equipment.

2.Developing validation techniques for simulation systems (which will be essential in predicting the extent and planning for climate change).

3.Developing systems for multicultural use

4.Developing systems that can be adapted quickly to new business needs

5.Designing systems for outsourced development

6.Developing systems that are resistant to attack

7.Developing systems that can be adapted and configured by end-users

8.Finding ways of testing, validating and maintaining end-user developed systems

9.Developing systems that are trusted by user

KRUTHIK B(1DA19CS077 B SECTION)

18)

Rational Unified Process(RUP)

The Rational Unified Process (RUP) is an adaptable process model that has been derived from work on the UML and the associated Unified Software Development Process . It brings together elements from all of the generic process models, illustrates good practice in specification and and supports prototyping and incremental delivery

The **RUP** recognizes that conventional process models present a single view of the process.

In contrast, the RUP is normally described from three perspectives:

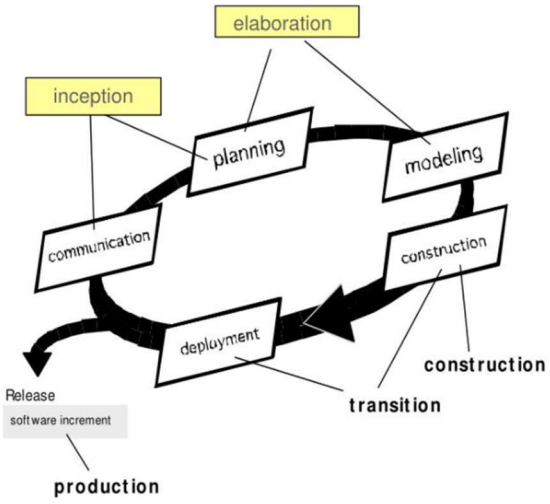
1.A *dynamic perspective,* which shows the phases of the model over time.

2.A *static perspective*, which shows the process activities that are enacted.

A practice perspective, which suggests good practices to be used during the process.

Dynamic perspective

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.



Static perspective

The static view of the RUP focuses on the activities that take place during the development process. These are called workflows in the RUP description. There are six core process workflows identified in the process and three core supporting workflows.

1)Business Modeling
2)Requirements
3)Analysis and Design
4)Implementation
5)Testing
6)Deployment
7)Configuration and management
8)Project management
9)Envorinment

The advantage in presenting dynamic and static views is that phases of the development process are not associated with specific workflows. In principle at least, all of the RUP workflows may be active at all stages of the process.

KRUTHIK B(1DA19CS077 B SECTION)

20)

Component-Based Development

The *component-based development model* incorporates many of the characteristics of the spiral model. It is evolutionary in nature [Nie92], demanding an iterative approach to the creation of software. However, the component based development model comprises applications from prepackaged software components.

Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps
1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

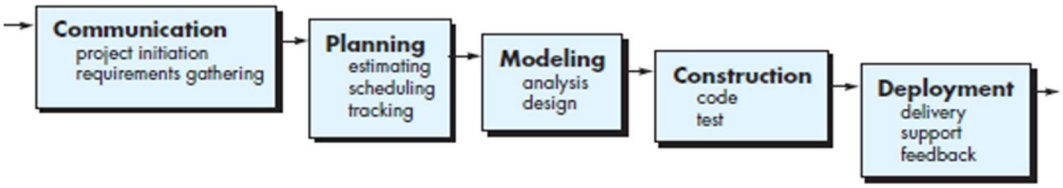
The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits including a reduction in development cycle time and a reduction in project cost if component reuse becomes part of your organization's culture.

KRUTHIK B(1DA19CS077 B SECTION)

19) / 4) / 6)
For all the questions just explain the model that they have asked and try to build ur own connection between them

The Waterfall Model

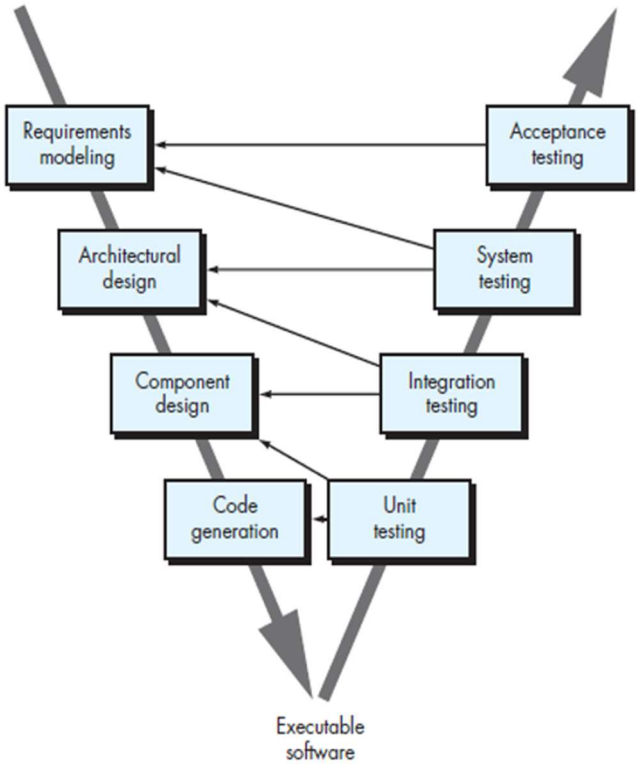
1)The waterfall is a universally accepted SDLC model. In this method, the whole process of software development is divided into various phases.
2)The waterfall model is a continuous software development model in which development is seen as flowing steadily downwards (like a waterfall) throughthe steps of requirements analysis, design, implementation, testing (validation), integration, and maintenance.
3)Linear ordering of activities has some significant consequences. First, to identify the end of a phase and the beginning of the next, some certification techniques have to be employed at the end of each step.



The V model

1)A variation in the representation of the waterfall model is called the V-model.
2)As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.
3)Once code has been generated, the team moves up the right side of the V,

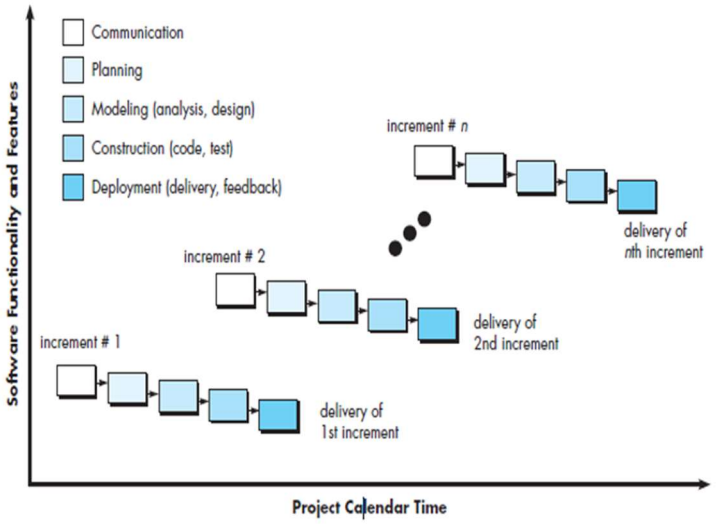
essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moves down the left side.
4)The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.



INCREMENTAL Model

The incremental model combines the elements' linear and parallel process flows the incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable "increments" of the software

When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/ or evaluation, a plan is developed for the next increment.



Evolutionary Process Models

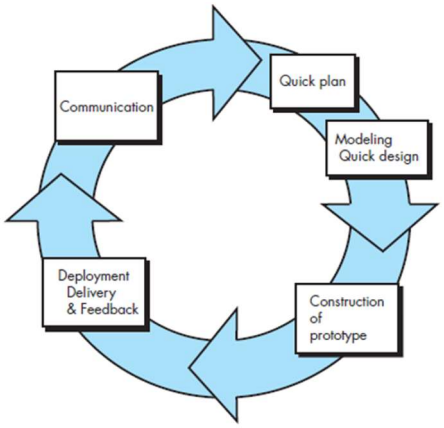
Software, like all complex systems, evolves over a period of time. Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. There are two common evolutionary models

PROTOTYPING model

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models noted in this chapter.

The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

Ideally, the prototype serves as a mechanism for identifying software requirements. If a working prototype is to be built, you can make use of existing program fragments or apply tools that enable working programs to be generated quickly.

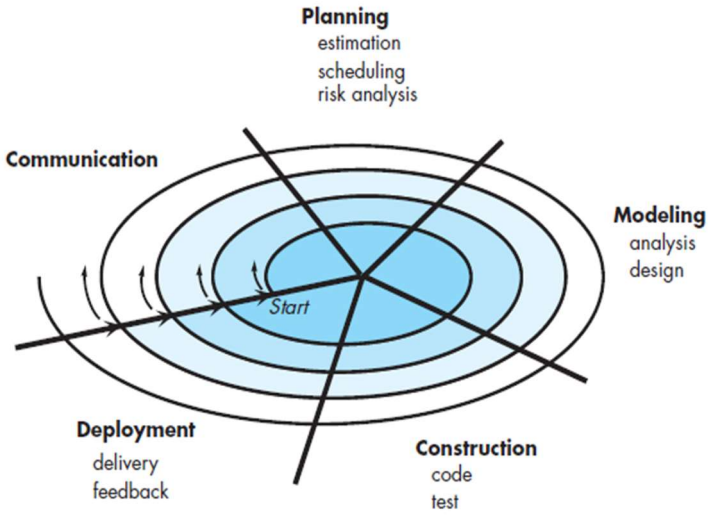


SPIRAL Model

The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.

One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.



KRUTHIK B(1DA19CS077 B SECTION)

16)

Stakeholders as those entities that are integrally involved in the healthcare system and would be substantially affected by reforms to the system. The major stakeholders in the healthcare system are **patients, physicians, employers, insurance companies, pharmaceutical firms and government.**

As stakeholders are everyone who will be affected in someway, it's fairly hard to list them all, butsome examples would be students, teachers, other faculty members relating to the records such as the billing department, government officials in charge of the financial aid given to students who are below a certain number of hours, people of that nature.

The requirements that stakeholders have will conflict in various ways, as different people have different priorities, and some of the stakeholders who influence the design of a system more than others may not know exactly what they really want and need.

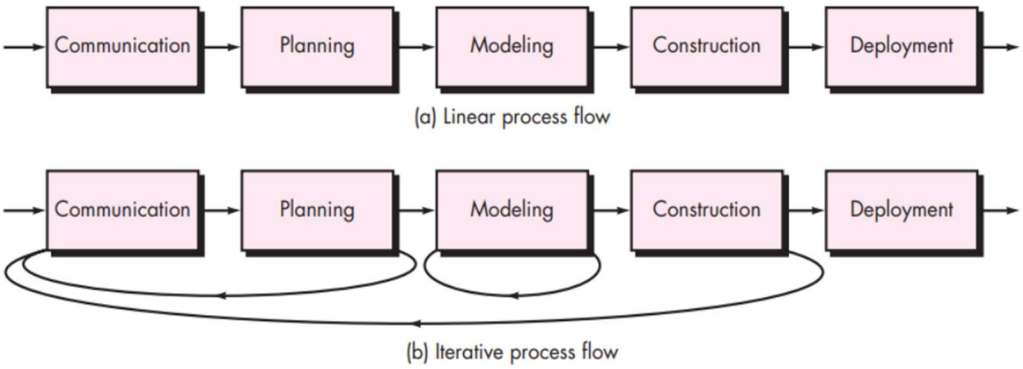
In the **student records system**, the requirements of students may become thing along the lines of they want to see at any given time what classes are remaining in their curriculum .

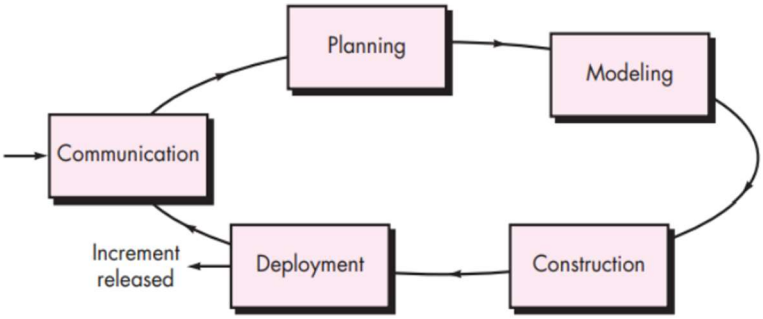
The **government officials** may want to see at any given time the hours of the courses already taken.

In an extremely simple records system, these would conflict with each other, and would need each to be built up onto meet the requirements of both parties

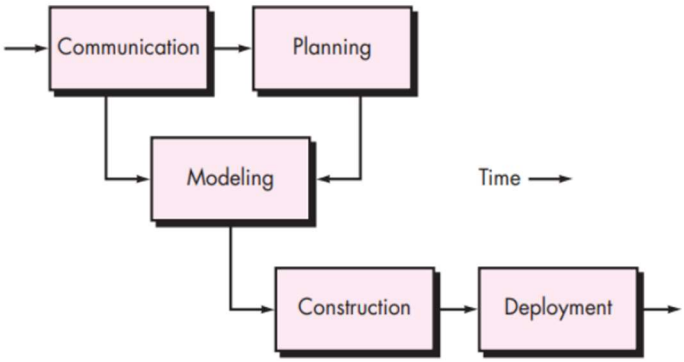
KRUTHIK B(1DA19CS077 B SECTION)

15)





(c) Evolutionary process flow



(d) Parallel process flow

A **linear process flow** executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.

An **iterative process flow** repeats one or more of the activities before proceeding to the next.

An **evolutionary process flow** executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software.

A **parallel process flow** executes one or more activities in parallel with other activities (e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software).

(Only five keywords required to be known just write ur own explanation after seeing the models)

KRUTHIK B(1DA19CS077 B SECTION)

14)

<u>The XP Process</u>

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that

occur within the context of four framework activities: planning, design, coding, and testing.

PLANNING

"The planning activity (also called the planning game) begins with listening —a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality. Listening leads to the creation of a set of " stories " (also called user stories) that describe required output, features, and functionality for software to be built.Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team". Once the commitment is completed ths stories will be dveloped in the following three ways

1) all stories will be implemented immediately (within a few weeks)

2) the stories with highest value will be moved up in the schedule and implemented first.

3) the riskiest stories will be moved up in the schedule and implemented first.

DESIGN

XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality (because the developer assumes it will be required later) is discouraged.

CODING

After stories are developed and preliminary design work is done, the team does not move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment). Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the test.

TESTING

The unit tests that are created should be implemented using a framework that enables them to be automated (hence, they can be executed easily and

repeatedly). This encourages a regression testing strategy whenever code is modified.

KRUTHIK B(1DA19CS077 B SECTION)

12)

The broad spectrum of tasks and techniques that lead to an understanding of requirements is called requirements engineering. From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity.

1)***Inception***

• Inception is a task where the requirement engineering asks a set of questions to

establish a software process.

• In this task, it understands the problem and evaluates with the proper solution.

• It collaborates with the relationship between the customer and the developer.

2)***Elicitation***

Elicitation means to find the requirements from anybody.

The requirements are difficult because the following problems occur in elicitation.

Problem of understanding: Poor understanding between the customer and the developer regarding various aspect of the project like capability, limitation of the computing environment.

Problem of volatility: In this problem, the requirements change from time to time and it is difficult while developing the project.

3)***Elaboration***

• In this task, the information taken from user during inception and elaboration and are expanded and refined in elaboration.

• Its main task is developing pure model of software using functions, feature and constraints of a software.

4)***Negotiation***

• In negotiation task, a software engineer decides the how will the project be achieved with limited business resources.

• To create rough guesses of development and access the impact of the requirement on the project cost and delivery time.

5)***Specification***

• In this task, the requirement engineer constructs a final work product.

• The work product is in the form of software requirement specification.

• In this task, formalize the requirement of the proposed software such as informative,functional and behavioral.

6)***Validation***

• The work product is built as an output of the requirement engineering and that is accessed for the quality through a validation step.

• The formal technical reviews from the software engineer, customer and other stakeholders helps for the primary requirements validation mechanism.

7)***Requirement management***

• It is a set of activities that help the project team to identify, control and track the requirements and changes can be made to the requirements

at any time of the ongoing project.

• After finalizing the requirement traceability table is developed.

• The examples of traceability table are the features, sources, dependencies, subsystems and interface of the requirement.

KRUTHIK B(1DA19CS077 B SECTION)

11)

The following are the problems that occur when requirements must be elicited from three or four different customers.

• The requirements of the customer cannot be understood easily. The customer's requirements will change over time such that a customer with a set of requirements at one time can include another set of requirements afterward.

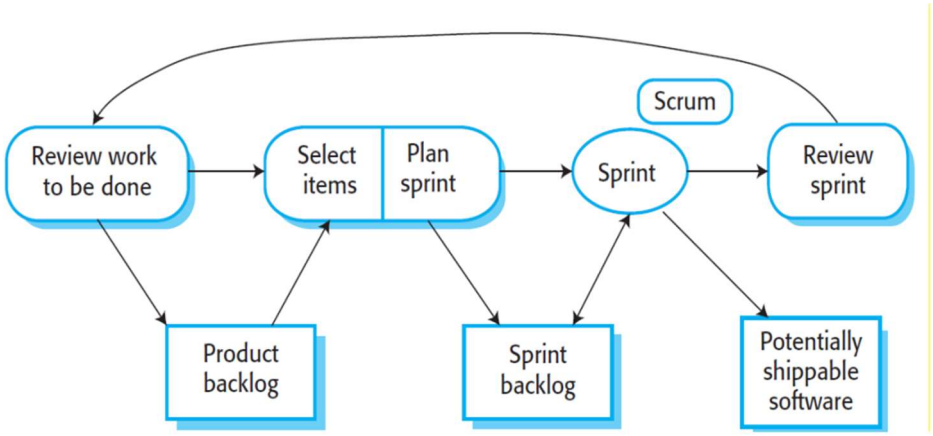
• It is very difficult to understand the requirements of the customers.

• The customers will have a wide range of expectations such that it may lead to disappointments at most of the time.

• The customers will change their requirements rapidly.

KRUTHIK B(1DA19CS077 B SECTION)

10)



Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery.

Backlog

A prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time.

Sprints

Consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box 10 (typically 30 days). Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.

Scrum meetings

Are short (typically 15-minute) meetings held daily by the Scrum team. A team leader, called a Scrum master, leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible.

Demos

Deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer.

KRUTHIK B(1DA19CS077 B SECTION)

9)

Agile software engineering combines a philosophy and a set of development guidelines. The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity. The development guidelines stress delivery over analysis and design (although these activities are not discouraged), and active and continuous communication between developers and customers.

12 AGILE PRINCIPLES

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7. Working software is the primary measure of progress.

8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9. Continuous attention to technical excellence and good design enhances agility.

10. Simplicity—the art of maximizing the amount of work not done—is essential.

11. The best architectures, requirements, and designs emerge from self organizing teams.

12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

KRUTHIK B(1DA19CS077 B SECTION)

8)

Component-Based Development:

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software.

The component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for the application domain in question.

2. Component integration issues are considered.

3. A software architecture is designed to accommodate the components.

4. Components are integrated into the architecture.

5. Comprehensive testing is conducted to ensure proper functionality. The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.

The Formal Methods Model:

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable you to specify, develop, and verify a computer-based system by applying a

rigorous, mathematical notation. A variation on this approach, called cleanroom software engineering, is currently applied by some software development organizations.

Its applicability in a business environment has been voiced:

• The development of formal models is currently quite time consuming and expensive.

• Because few software developers have the necessary background to apply formal methods, extensive training is required.

• It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

Aspect-Oriented Software Development:

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions, and information content. These localized software characteristics are modeled as components (e.g., object oriented classes) and then constructed within the context of a system architecture. As modern computer-based systems become more sophisticated (and complex), certain concerns—customer required properties or areas of technical interest—span the entire architecture

KRUTHIK B(1DA19CS077 B SECTION)

7)

1. ***Understand the problem (communication and analysis):***

Who has a stake in the solution to the problem? That is, who are the stakeholders?

What are the unknowns? What data, functions, and features are required to properly solve the problem?

Can the problem be compartmentalized? Is it possible to represent smaller problems that may be easier to understand?

Can the problem be represented graphically? Can an analysis model be created?

2. ***Plan a solution (modelling a software design):***

Have you seen similar problems before?

Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?

Has a similar problem been solved? If so, are elements of the solution reusable?

Can sub problems be defined? If so, are solutions readily apparent for the sub problems?

Can you represent a solution in a manner that leads to effective implementation? Can a design model be created?

3. ***Carry out a plan (code generation):***

The design we create serves as a road map the system we want to create.

Does the solution conform the plan? Is the source code traceable to the design model?

Is each part of the component part of the solution provably correct? Has the design and code had correctness proofs been applied to the algorithm?

4. ***Examine the result for accuracy (testing and quality assurance):***

You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

Is it possible to test each component part of the solution? Has a reasonable testing strategy been implemented?

Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements?

(understand and write the questions in ur own way but two questions min for case)

KRUTHIK B(1DA19CS077 B SECTION)

5)

Barry Boehm (Boehm, 1988) proposed a risk-driven software process framework (the spiral model) that integrates risk management and incremental development. The software process is represented as a spiral rather than a sequence of activities with some backtracking from one activity

to another. Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design and so on.

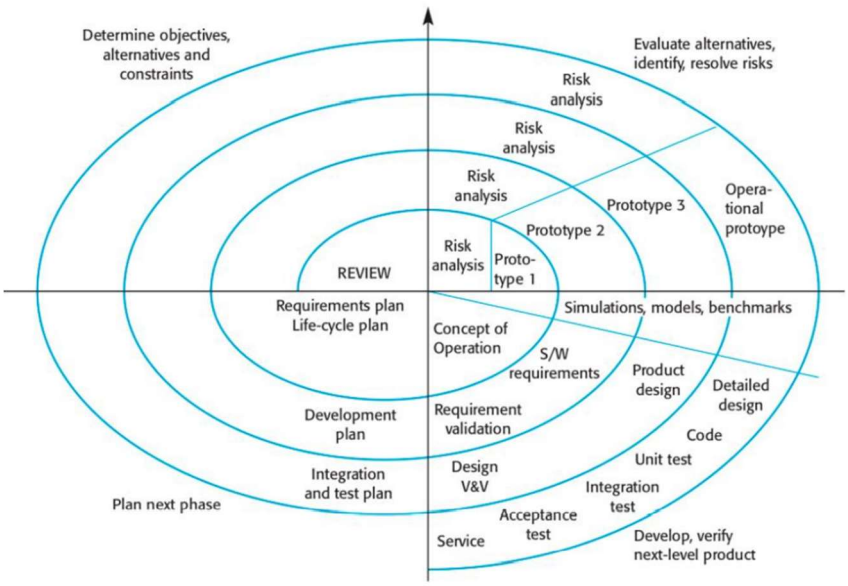
Each loop in the spiral is split into four sectors

1. **Objective setting**: Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.

2. **Risk assessment and reduction**: For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

3. **Development and validation**: After risk evaluation, a development model for the system is chosen. For example, throw-away prototyping may be the best development approach if user interface risks are dominant. If safety risks are the main consideration, development based on formal transformations may be the most appropriate process, and so on.

4. **Planning**: The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.



Software engineering.

CBSE activity.

3rd unit page 4 good design.

1. Propose software quality guidelines of attributes for a
2. Briefly explain the taxanomy of architectural style.
3. Draw the architectural context diagram for hospital. solved mapp page 26
4. Describe cohesion and coupling with example. 3rd unit page 8
5. Four elements of design model. 3rd unit 11-12
6. concepts in design i) design patterns 6 ii) separation and concern 6 3rd unit page iii) refinement and refactoring. 9
7. What is meant by design class ? List and explain characteristics 8 of a well defined design class. 3rd unit 9-10
8. Suggest & justify architectural style for the online Jewellery mart system, also identify major components used in designing online Jewellery mart system.
9. Basic design principles applicable to component level diagram. 3rd unit 33-34
10. Illustrate dimensions of design technique with diagram 3rd unit 10-11
11. Architectural context diagram for i) online gaming | ii) agricultural product system.
12. Component based development in detail. 3rd unit 43-44
13. Fundamentals of software design process. 3rd unit 3|4|5 explain any 6
14. Architectural design for safe house product, also identify various components used in safe home product. types genres
15. Suitable examples, describe architectural styles & architectural 16 grp unit 3rd 17
16. Design architectural context diagram for iii) stock market trading system iv) consumer product management system
17. content type used in software design. (styles) process
18. Explain different types of design classes used in system design same as (7)
19. conducting component level design (VVIP) 3rd unit page 14
20. compare and contrast shared repository & client-server system organizn models.
21. With component based development explain CBSE activity in same as (12) detail.
22. Use case diagram for (university library management | Buying a stock online brokerage a/c | use credit card at a restaurant)

Solve MAP 17 page

(23) purpose of domain analysis ? How it is related to concept of requirement patterns.

(24) Use case and Activity diagram for PTRS PathDale tracking and repair system. solved MAP 18th page

(25) component level design for Hospital management system.

(26) 10 object classes for employee management system.

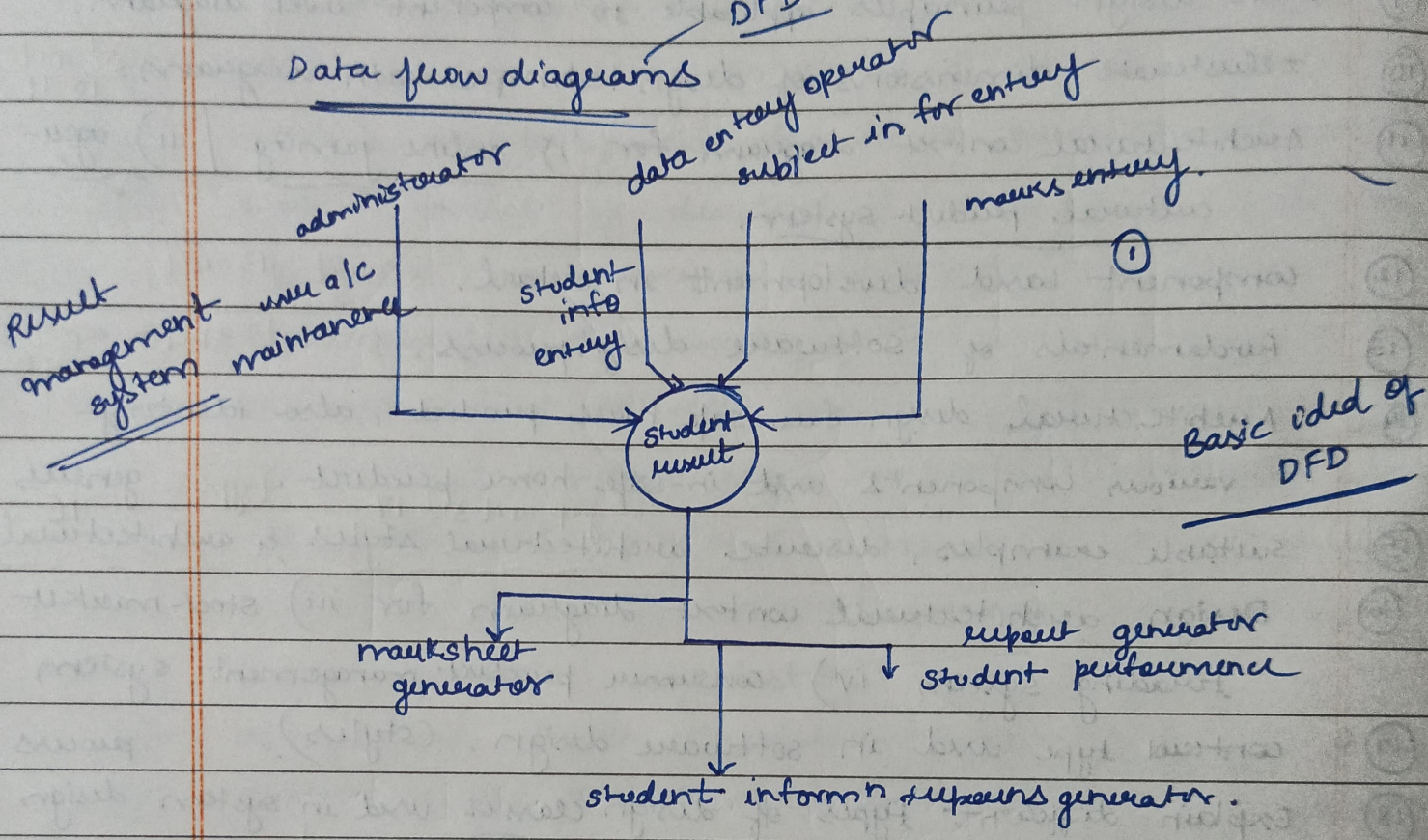
(27) use case and a sequence diagram that could serve as a understanding the requirements for a Bank ATM system. OOP concept

(28) context model for patient health care system.

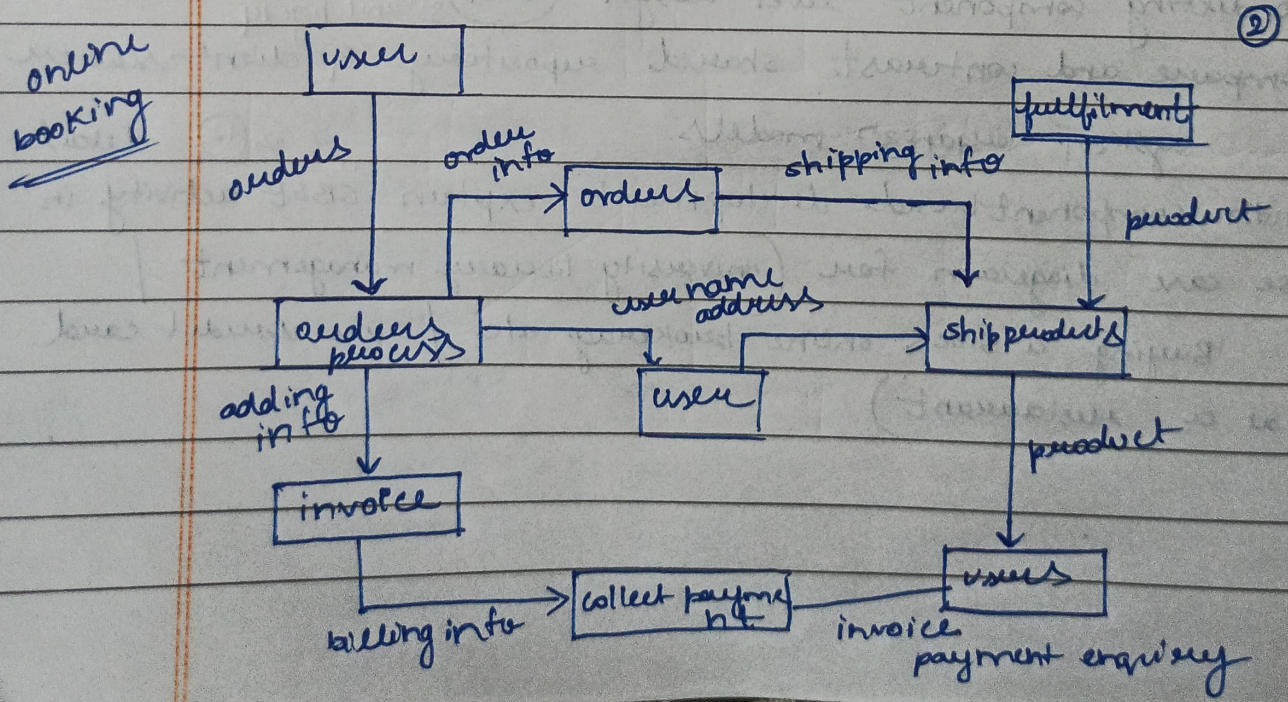
for reference 2nd unit pg 22

DFD

Data flow diagrams

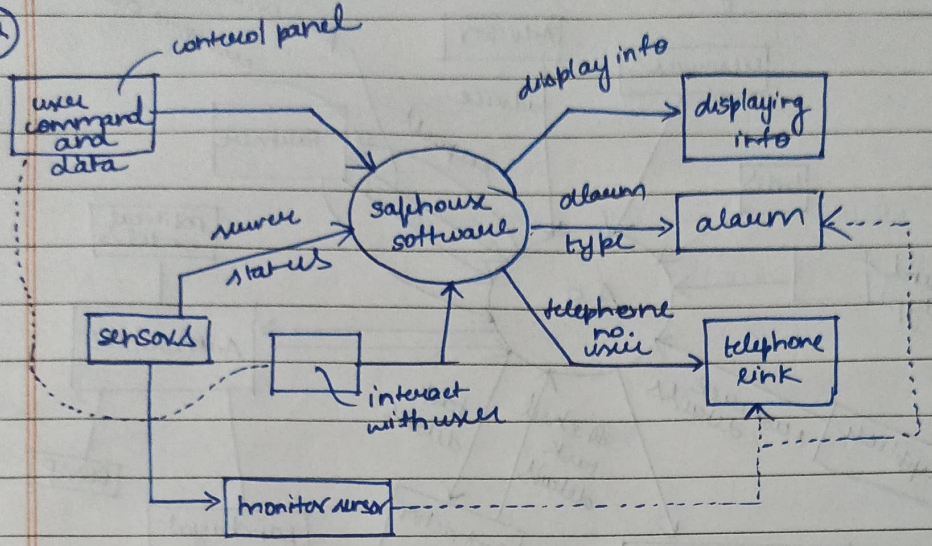


Basic idea of DFD

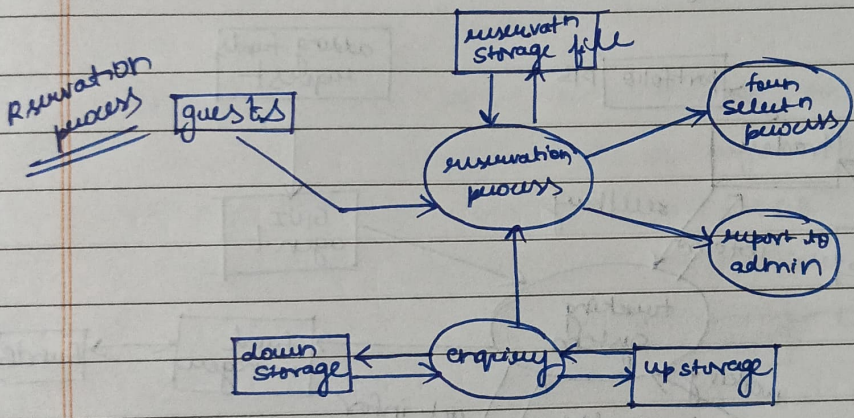


process diagrams | safehouse software with DFD

(2)



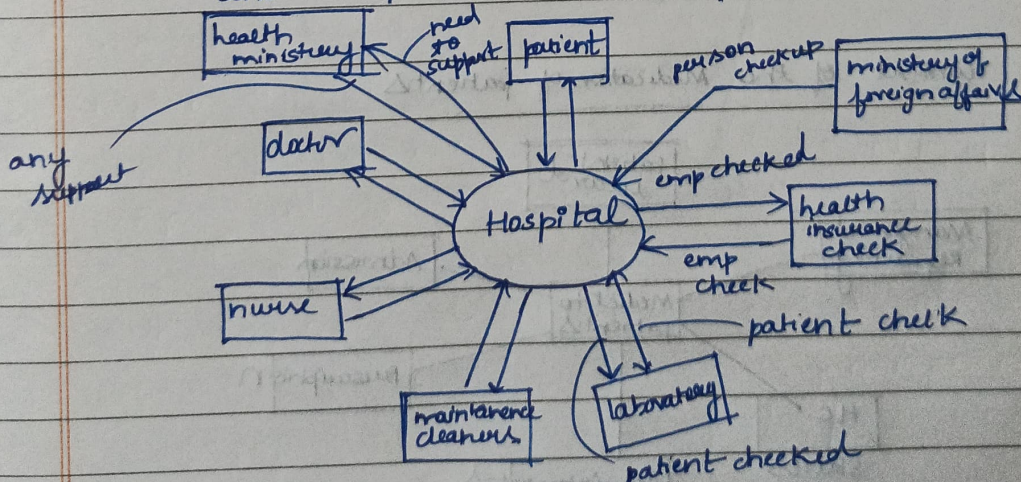
(3) → data flow | process | source/sink | data store



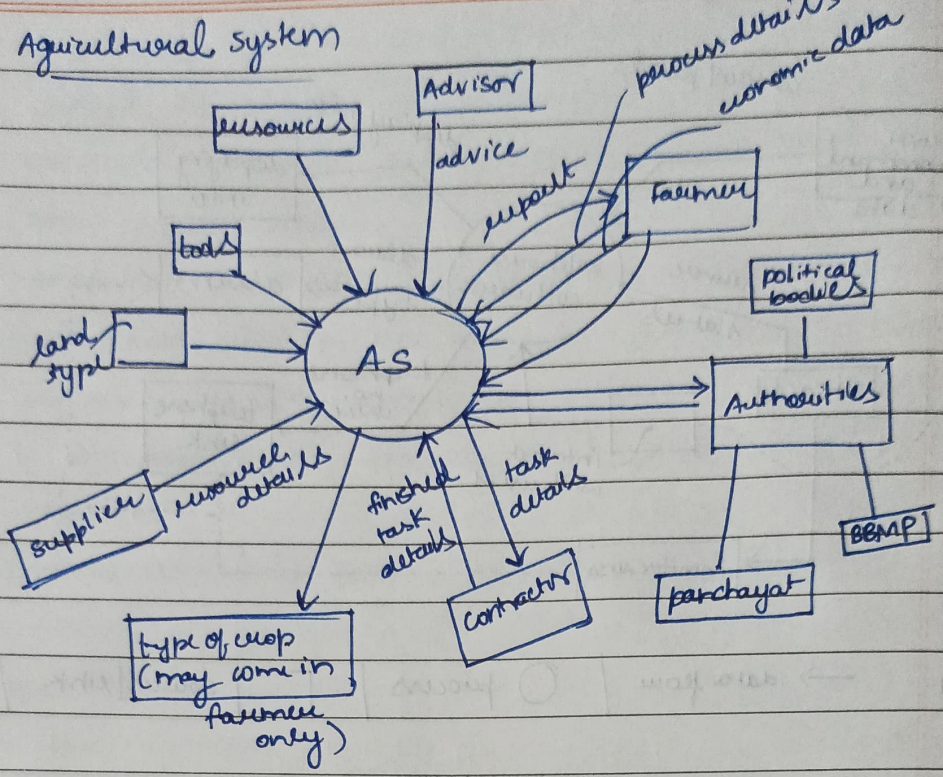
context diagram

online gaming | agriculture | stock grading market

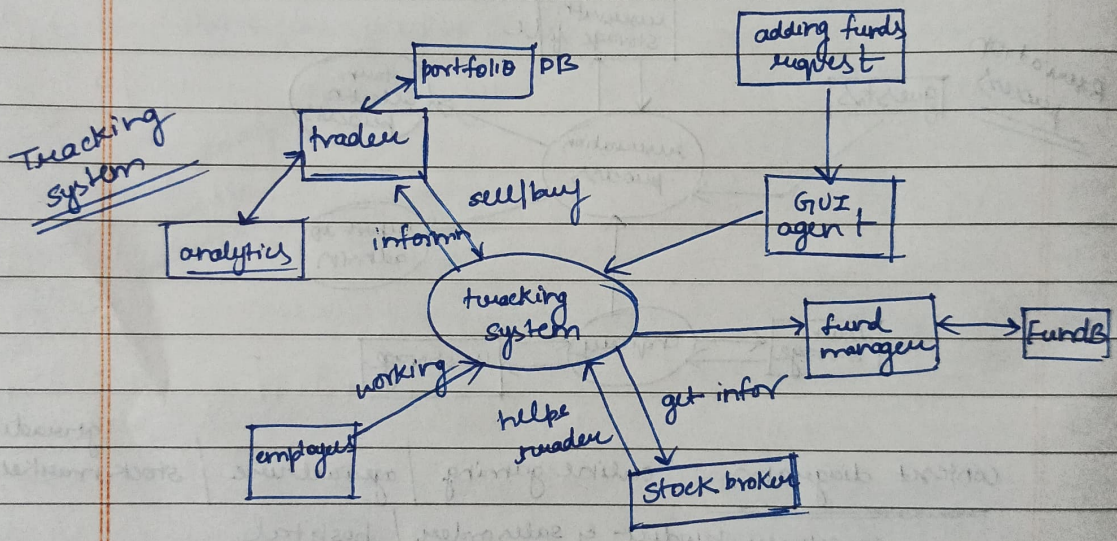
consumer product & sales order | hospital



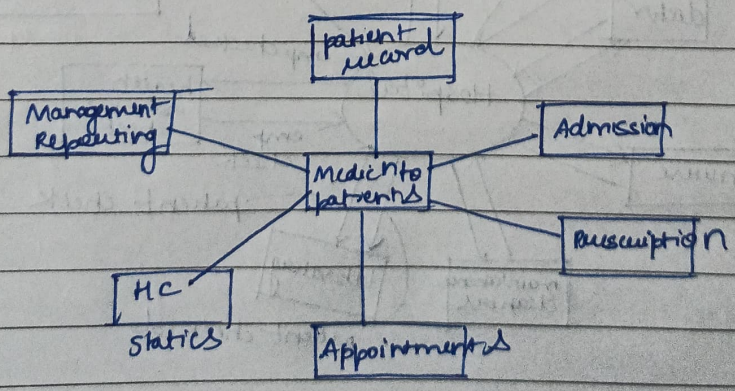
Agricultural system



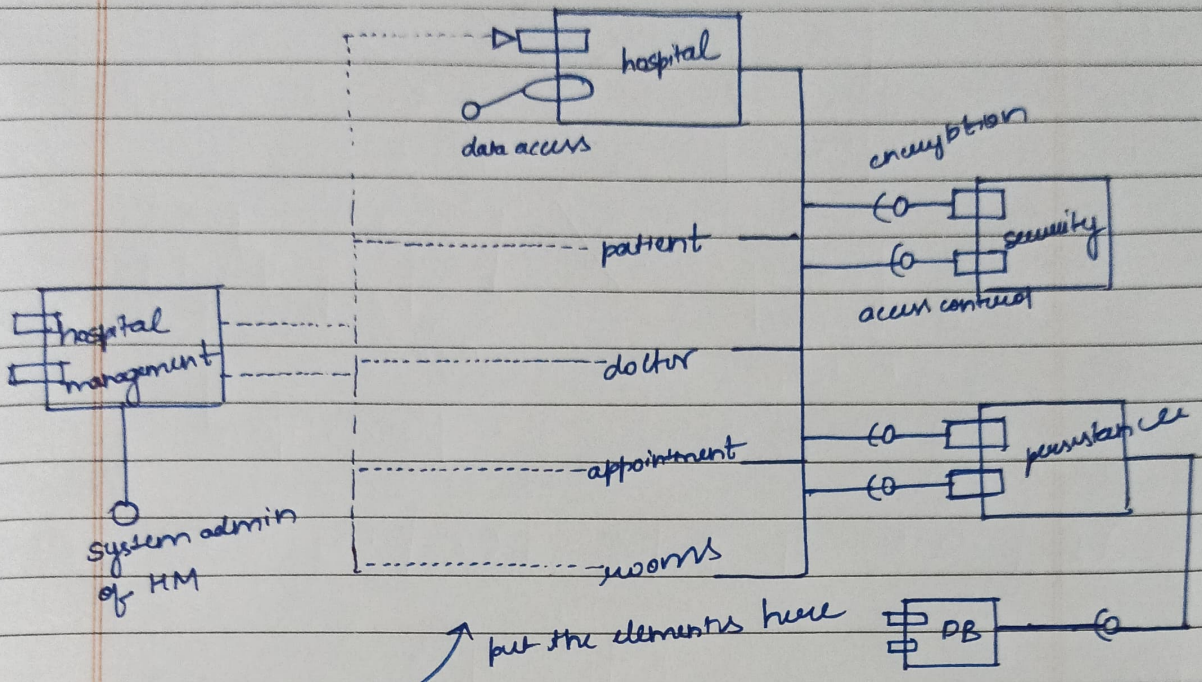
Tracking system



Context of the Medication of patients

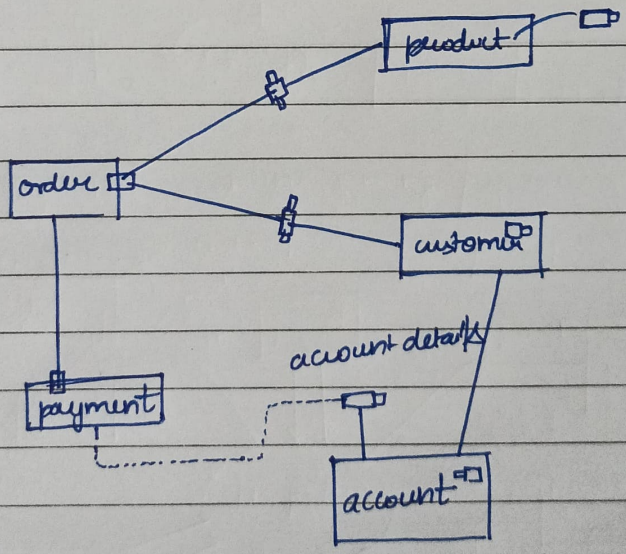


Component diagram (Hospital Management system)



CD for ticketsystem bus, booking, ticket, customer, route, bus class, facility, mode of payment.

CD for library management - LMDB, Bank, member, transaction, packagename, DB, maintenance.



SE

Makeup(cie-3) SE/IPR/AI

Testing (almost covered separately also.)

~ || (1) ✓ Distinguish blw white box and black box testing.

~ | (2) ✓ Discuss the importance of determinants for software quality and organizational effectiveness with a neat diagram.

~ | (3) ✓ With a neat diagram, describe the art debugging process.

| (4) ○ Defect | Structural | Inspection checks | Path Testing | Inspection team

(5) ✓ Identify different test strategies for conventional software and explain any one strategy in detail.

~ (6) ✓ Explain Mccabe's path testing along with graph

(7) ✓ With black box testing for boundary values analysis, find test cases for a function of, two variables

(8) ✗ major challenges faced during finalizing project resources in estimating software products (projects)

~ (9) ✓ Illustrate the concept of integrating metrics within the software process using collection process.

(10) ✗ Effective software management focusses on four (Ps) Justify

(11) ✗ Explain why program inspections are an effective technique for discovering errors in a program. wt types of errors are unlikely to be discovered through inspections.

(12) ✓ (1) question and how they are used together in defect testing process with ex.

(13) ✗ what are the processes of change identification and system evolution ? Explain.

(14) ✗ Four strategic options of legacy system evolution ?

| (15) ✗ Explain clean room software development method.

~ | (16) ✓ Difference blw verifict | validn, explain why validn is particularly a different process.

(17) ✓ Integration testing (types | advantages)

(18) ✗ Software inspections with ex in detail.

(19) ✗ Release testing (exp and example)

~ (20) ✓ factors governing staffg collection | How to motivate ppl in software industry.

~ (21) ✓ Explain ppl capability mature model (neat deagram)

(22) ✗ Rapid applⁿ development process (examples)

✗ not in syllabus
~ stress more
✓ answers are there

(5th unit so many questions they can ask but cant do much abt it coz we are ******* screwed)

☺

~| (23) ✓ Strategic approach to software testing

(24) ✓ contrast top-down and bottom-up integration testing

(25) ✓ Briefly explain management spectrum in software project management

~| (26) ✓ Establish a software metrics program (various steps and goals) — explain four metrics (software measurement)

(27) ✗ Three major categories of software engineering process.

~ (28) ✓ Empirical estimating models.

Software testing fundamentals

Testability (7 characteristics)

Test characteristics (4 attributes)

(1) ~ ✓ External view of testing (Black box)

~ ✓ Internal —— 11 —— (White box)

~ ✓ Basis path testing

~ ✓ control structure testing

SOFTWARE ENGINEERING (18CS51)

(Cie 3 Important questions and Answers)

KRUTHIK B

1DA19CS077

B SECTION B2 BATCH

1) and 7) and 12)(I have went all out on white and black box testing)

Black Box Testing	White Box Testing
It is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it.	It is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software.
It is mostly done by software testers.	It is mostly done by software developers.
No knowledge of implementation is needed.	Knowledge of implementation is required.
It can be referred as outer or external software testing.	It is the inner or the internal software testing.
It is functional test of the software.	It is structural test of the software.
This testing can be initiated on the basis of requirement specifications document.	This type of testing of software is started after detail design document.
No knowledge of programming is required.	It is mandatory to have knowledge of programming.
It is the behavior testing of the software.	It is the logic testing of the software.
It is applicable to the higher levels of testing of software.	It is generally applicable to the lower levels of software testing.
It is also called closed testing.	It is also called as clear box testing.
It is least time consuming.	It is most time consuming.
It is not suitable or preferred for algorithm testing.	It is suitable for algorithm testing.
Can be done by trial and error ways and methods.	Data domains along with inner or internal boundaries can be better tested.
Example: search something on google by using keywords	**Example:** by input to check and verify loops
Types of Black Box Testing: • A. Functional Testing • B. Non-functional testing • C. Regression Testing	**Types of White Box Testing:** • A. Path Testing • B. Loop Testing • C. Condition testing

Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than whitebox methods.

Black-box testing attempts to find errors in the following categories:

(1) incorrect or missing functions,

(2) interface errors,

(3) errors in data structures or external database access,

(4) behavior or performance errors, and

(5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, blackbox testing tends to be applied during later stages of testing. Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

• How is functional validity tested?

• How are system behavior and performance tested?

• What classes of input will make good test cases?

• Is the system particularly sensitive to certain input values?

• How are the boundaries of a data class isolated?

• What data rates and data volume can the system tolerate?

• What effect will specific combinations of data have on system operation?

By applying black-box techniques, you derive a set of test cases that satisfy the following criteria

(1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and

(2) test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Example,

A simple login screen of software or a web application will be tested for seamless user login. The login screen has two fields, username and password as an input and the output will be to enable access to the system.

A black box testing will not consider the specifications of the code, and it will test the valid username and password to login to the right account.

This form of testing technique will check the input and output.

A user logged in when inputs a present username and correct password

A user receives an error message when enters username and incorrect password

The black box testing is also known as an opaque, closed box, function-centric testing. It emphasizes on the behavior of the software. Black box testing checks scenarios where the system can break.

For example, a user might enter the password in the wrong format, and a user might not receive an error message on entering an incorrect password.

White-box testing, sometimes called glass-box testing, is a testcase design philosophy that uses the control structure described as part of component-level design to derive test cases. Using white-box testing methods, you can derive test cases that

(1) guarantee that all independent paths within a module have been exercised at least once,

(2) exercise all logical decisions on their true and false sides,

(3) execute all loops at their boundaries and within their operational bounds, and

(4) exercise internal data structures to ensure their validity.

Example,

Consider the following piece of code

```
Printme (int a, int b) {                    ----------  Printme is a function
   int result = a+ b;
   If (result> 0)
        Print ("Positive", result)
   Else
        Print ("Negative", result)
   }                                  ----------   End of the source code
```

The goal of WhiteBox testing in software engineering is to verify all the decision branches, loops, statements in the code.

To exercise the statements in the above white box testing example, WhiteBox test cases would be

A = 1, B = 1

A = -1, B = -3

Following are important WhiteBox Testing Techniques:

- Statement Coverage
- Decision Coverage
- Branch Coverage
- Condition Coverage
- Multiple Condition Coverage
- Finite State Machine Coverage
- Path Coverage
- Control flow testing
- Data flow testing

*Understand the concept and try to apply both the testing in one example

*Boundary value analysis given below

Boundary Value Analysis

Boundary value analysis is one of the widely used case design technique for black box testing. It is used to test boundary values because the input values near the boundary have higher chances of error.

Whenever we do the testing by boundary value analysis, the tester focuses on, while entering boundary value whether the software is producing correct output or not.

Boundary values are those that contain the upper and lower limit of a variable. Assume that, age is a variable of any function, and its minimum value is 18 and the maximum value is 30, both 18 and 30 will be considered as boundary values.

The basic assumption of boundary value analysis is, the test cases that are created using boundary values are most likely to cause an error.

There is 18 and 30 are the boundary values that's why tester pays more attention to these values, but this doesn't mean that the middle values like 19, 20, 21, 27, 29 are ignored. Test cases are developed for each and every value of the range.

Imagine, there is a function that accepts a number between 18 to 30, where 18 is the minimum and 30 is the maximum value of valid partition, the other values of this partition are 19, 20, 21, 22, 23, 24, 25, 26, 27, 28 and 29. The invalid partition consists of the numbers which are less than 18 such as 12, 14, 15, 16 and 17, and more than 30 such as 31, 32, 34, 36 and 40. Tester develops test cases for both valid and invalid partitions to capture the behavior of the system on different input conditions.

Invalid test cases	Valid test cases	Invalid test cases
11, 13, 14, 15, 16, 17	18, 19, 24, 27, 28, 30	31, 32, 36, 37, 38, 39

The software system will be passed in the test if it accepts a valid number and gives the desired output, if it is not, then it is unsuccessful. In another scenario, the software system should not accept invalid numbers, and if the entered number is invalid, then it should display error massage.

If the software which is under test, follows all the testing guidelines and specifications then it is sent to the releasing team otherwise to the development team to fix the defects.

*Create test cases by understanding the problem ***KRUTHIK B(1DA19CS077 B SECTION)***

2)

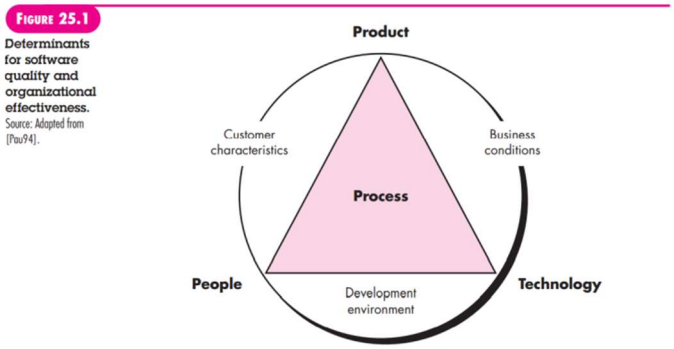


FIGURE 25.1 Determinants for software quality and organizational effectiveness. Source: Adapted from [Pau94].

The only rational way to improve any process is to measure specific attributes of the process, develop a set of meaningful metrics based on these attributes, and then use the metrics to provide indicators that will lead to a strategy for improvement.

But before we discuss software metrics and their impact on software process improvement, it is important to note that process is only one of a number of "controllable factors in improving software quality and organizational performance".

Referring to Figure 25.1, process sits at the center of a triangle connecting three factors that have a profound influence on software quality and organizational performance.

The skill and motivation of people has been shown to be the single most influential factor in quality and performance.

The complexity of the product can have a substantial impact on quality and team performance.

The technology (i.e., the software engineering methods and tools) that populates the process also has an impact.

In addition, the process triangle exists within a circle of environmental conditions that include the development environment (e.g., integrated software tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication and collaboration). You can only measure the efficacy of a software process indirectly.

That is, you derive a set of metrics based on the outcomes that can be derived from the process.

Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end users, work products delivered (productivity), human effort expended, calendar time expended, schedule conformance, and other measures. We can also derive process metrics by measuring the characteristics of specific software engineering tasks.

3)

The debugging process begins with the execution of a test case.

The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will usually have one of two outcomes:

(1) the cause will be found and corrected or

(2) the cause will not be found.

Few characteristics of bugs provide some clues:

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed.

2. The symptom may disappear (temporarily) when another error is corrected.

3. The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).

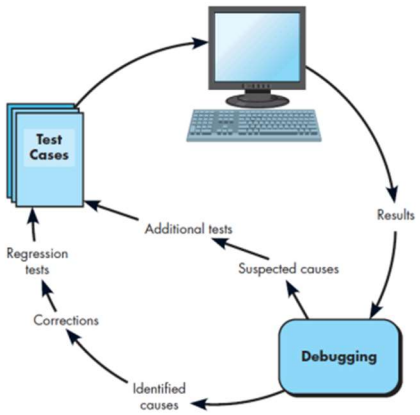
4. The symptom may be caused by human error that is not easily traced.

5. The symptom may be a result of timing problems, rather than processing problems.

6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.

The symptom may be due to causes that are distributed across a number of tasks running on different processors.



4) *Understand in ur own words and write wt comes to mind

A **Defect Based Testing** Technique is a technique where test cases are derived on the basis of defects. Instead of using the traditional requirements documents or the use cases (Specification-based techniques), this strategy uses the defects to base their test cases.

The categorized list of Defects (called Defect Taxonomy) is being used. The coverage using this technique is not very systematic, hence deriving the base of your test cases on this technique only, may not solve the purpose of the quality deliverable.

This technique can complement your test deriving conditions and can be taken as one of the options to increase the testing coverage. Or in some other sense – This technique can be applied when all the test conditions and test cases are identified and we need some extra coverage or insight into testing.

Defect based technique can be used at any level of testing, but it is best suited in Systems Testing. We should base our test cases from the available defect taxonomies as well. These defects can be the production ones or historical ones. Root cause analysis can also be used to baseline your test cases.

Structural testing is the type of testing carried out to test the structure of code.

It is also known as White Box testing or Glass Box testing.

This type of testing requires knowledge of the code, so, it is mostly done by the developers.

It is more concerned with how system does it rather than the functionality of the system.

It provides more coverage to the testing.

For ex, to test certain error message in an application, we need to test the trigger condition for it, but there must be many trigger for it. It is possible to miss out one while testing the requirements drafted in SRS. But using this testing, the trigger is most likely to be covered since structural testing aims to cover all the nodes and paths in the structure of code.

An **inspection checklist** is simply an assurance that specific software product has been inspected.

An inspection checklist should be developed by discussion with some experienced staff and as well as regularly updated as more experience is gained from inspection process. Guidebook generally includes checklist simply for various artifacts such as design documents, requirements, etc.

Path Testing is a method that is used to design the test cases.

In path testing method, the control flow graph of a program is designed to find a set of linearly independent paths of execution.

In this method Cyclomatic Complexity is used to determine the number of linearly independent paths and then test cases are generated for each path.

It give complete branch coverage but achieves that without covering all possible paths of the control flow graph.

McCabe's Cyclomatic Complexity is used in path testing. It is a structural testing method that uses the source code of a program to find every possible executable path.

Inspection team consists of:

• Author: The person who created the work product being inspected.

• Moderator: This is the leader of the inspection. The moderator plans the inspection and coordinates it.

• Reader: The person reading through the documents, one item at a time. The other inspectors then point out defects.

• Recorder/Scribe: The person that documents the defects that are found during the inspection.

• Inspector: The person that examines the work product to identify possible defects.

5)

Following are the four strategies for conventional software:

1) Unit testing 2) Integration testing 3) Regression testing 4) Smoke testing

1) Unit testing

• Unit testing focus on the smallest unit of software design, i.e module or software component.

• Test strategy conducted on each module interface to access the flow of input and output.

• The local data structure is accessible to verify integrity during execution.

• Boundary conditions are tested.

• In which all error handling paths are tested.

• An Independent path is tested.

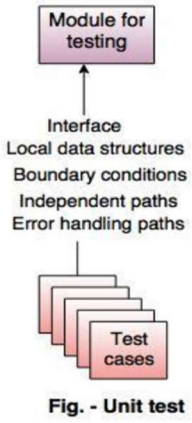


Fig. - Unit test

2) Integration testing

Integration testing is used for the construction of software architecture.

There are two approaches of incremental testing are:

i) Non incremental integration testing

• Combines all the components in advanced.

• A set of error is occurred then the correction is difficult because isolation cause is complex.

ii) Incremental integration testing

• The programs are built and tested in small increments.

• The errors are easier to correct and isolate.

• Interfaces are fully tested and applied for a systematic test approach to it.

3) Regression testing

• In regression testing the software architecture changes every time when a new module is added as part of integration testing.

4)Smoke Testing

• Smoke testing is an integration testing approach that is commonly used when product software is developed

• Smoke testing is performed by developers before releasing the build to the testing team and after releasing the build to the testing team it is performed by testers whether to accept the build for further testing or not.

• It is designed as a pacing (Speedy) mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.

6)

Basis path testing is a white-box testing technique fi rst proposed by Tom McCabe [McC76].

The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

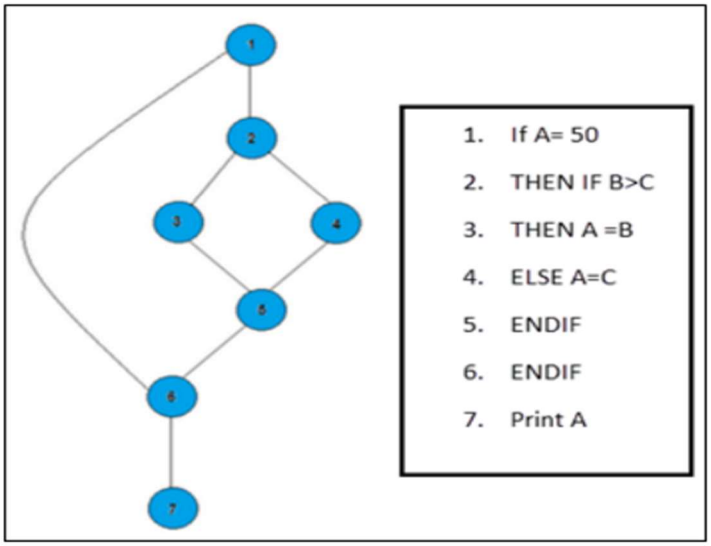
Steps for basis path testing:

Draw a control graph (to determine different program paths)

Calculate Cyclomatic complexity (metrics to determine the number of independent paths)

Find a basis set of paths

Generate test cases to exercise each path



Basis path testing involves execution of all possible blocks in a program and achieves maximum path coverage with the least number of test cases.

In the above example, we can see there are few conditional statements that is executed depending on what condition it suffice. Here there are 3 paths or condition that need to be tested to get the output,

• Path 1: 1,2,3,5,6, 7

• Path 2: 1,2,4,5,6, 7

• Path 3: 1, 6, 7

9)

The majority of software developers still do not measure, and sadly, most have little desire to begin.

By establishing a metrics baseline, benefits can be obtained at the process, project, and product (technical) levels.

Yet the information that is collected need not be fundamentally different.

The same metrics can serve many masters.

The metrics baseline consists of data collected from past software development projects and can be as simple or as complex as a comprehensive database containing dozens of project measures and the metrics derived from them.

To be an effective aid in process improvement and/or cost and effort estimation, baseline data must have the following attributes:

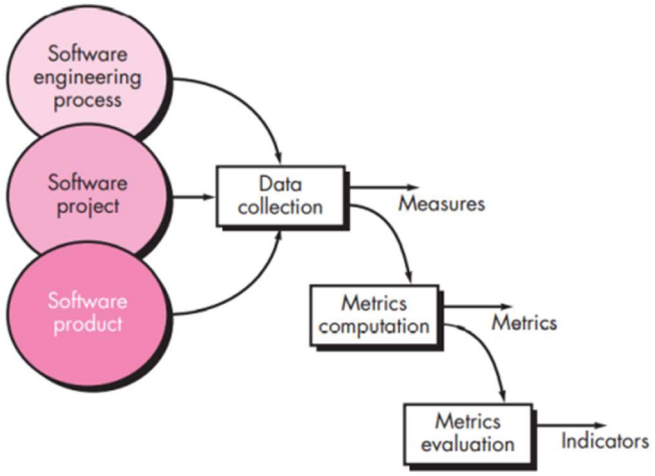
(1) data must be reasonably accurate—"guestimates" about past projects are to be avoided,

(2) data should be collected for as many projects as possible,

(3) measures must be consistent (for example, a line of code must be interpreted consistently across all projects for which data are collected),

(4) applications should be similar to work that is to be estimated—it makes little sense to use a baseline for batch information systems work to estimate a real-time, embedded application.

FIGURE 25.3
Software metrics collection process

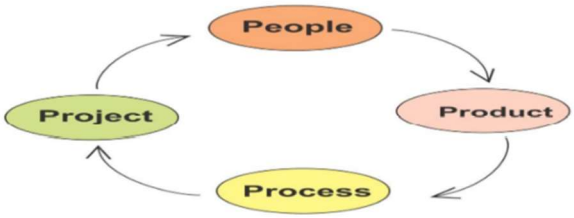


25) and 10) *It's There

The management spectrum describes the management of a software project or how to make a project successful.

It focuses on the four P's; people, product, process and project.

Here, the manager of the project has to control all these P's to have a smooth flow in the project progress and to reach the goal.



The People:

People of a project includes from manager to developer, from customer to end user.

But mainly people of a project highlight the developers.

It is so important to have highly skilled and motivated developers that the Software Engineering Institute has developed a

People Management Capability Maturity Model (PM-CMM), "to enhance the readiness of software organizations to undertake increasingly complex applications by helping to attract, grow, motivate, deploy, and retain the talent needed to improve their software development capability".

Organizations that achieve high levels of maturity in the people management area have a higher likelihood of implementing effective software engineering practices.

The Product:

Product is any software that has to be developed.

To develop successfully, product objectives and scope should be established, alternative solutions should be considered, and technical and management constraints should be identified.

Without this information, it is impossible to define reasonable and accurate estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks or a manageable project schedule that provides a meaningful indication of progress.

The Process:

A software process provides the framework from which a comprehensive plan for software development can be established.

A number of different tasks sets— tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team.

Finally, umbrella activities overlay the process model.

Umbrella activities are independent of any one framework activity and occur throughout the process.

The Project:

Here, the manager has to do some job.

The project includes all and everything of the total development process and to avoid project failure the manager has to take some steps, has to be concerned about some common warnings etc.

*for 10th one explain the below para and give some random explanation about the four p's at the top

Effective software project management focuses on the four P's: people, product, process, and project.

The order is not arbitrary.

The manager who forgets that software engineering work is an intensely human endeavor will never have success in project management.

A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem.

The manager who pays little attention to the processs runs the risk of inserting competent technical methods and tools into a vacuum.

The manager who embarks without a solid project plan jeopardizes the success of the project

KRUTHIK B(1DA19CS077 B SECTION)

16)

Sr. No.	Key	Verification	Validation
1	Definition	Verification is the process in which product or system is evaluated in development phase to find out whether it meets the specified requirements or not.	On other hand Validation is the process in which product or system is evaluated at the end of the development process to determine whether software meets the customer expectations and requirements or not.
2	Objective	The main objective of Verification process is to make sure that the system being develop is as per the requirements and design specifications of the customer and if it deviates from it then make it correct in the development phase itself.	On other hand the objective of Validation is to make sure that the product which has been developed is actually meet up the user's requirements or not. And if it is not then make it to the level of acceptance in re development phase.
3	Activities	Main activities which defines the Verification process are Reviews of specification and product development, Meetings about diversification and inspections.	On other hand activities under Validation process are typically different type of testing such as Black Box testing, White Box testing, Grey box testing etc. which ensure the defect free delivery of product as per specification document.
4	Type	Verification is the process where execution of code is not take place and hence it comes under static testing.	On other hand during Validation execution of code take place and thus it comes under dynamic testing.
5	Sequence	Verification is carried out before the Validation.	On other hand Validation activity is carried out just after the Verification.
6	Performer	Verification is carried out by Quality assurance team.	On other hand Validation is executed on software code with the help of testing team.

*just explain some random s**t that validation is different than verification after studying the differences. Explain in ur own words.

KRUTHIK B(1DA19CS077 B SECTION)

17) and 24)

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.

The objective is to take unit-tested components and build a program structure that has been dictated by design.

There is often a tendency to attempt non incremental integration; that is, to construct the program using a “big bang” approach.

All components are combined in advance and the entire program is tested as a whole.

Chaos usually results! Errors are encountered, but correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.

Incremental integration is the antithesis of the big bang approach.

ADVANTAGES

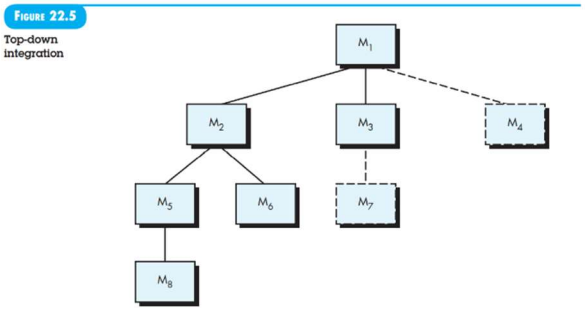
Integration testing for different modules at the same time is easy.

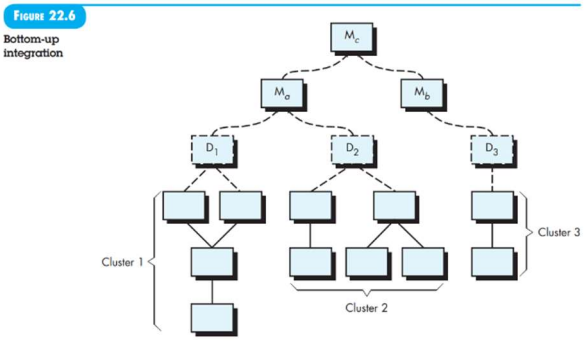
It can be used in the early as well as later stages of the testing process.

Code length coverage is more as compared with other software testing techniques as both the bottom-up and top-down approaches can be used.

According to the changes in the requirements, the development varies, so the testing of modules at different levels becomes important, for which Integration testing can be used easily.

Top down and Bottom up are the two types of integration





*Explain in ur own words with diagram after studying the differences, differences are in next page

Top-Down integration testing	Bottom-up integration testing
Integration testing takes place from top to bottom i.e., it begins with the top-level module.	Integration testing takes place from bottom to top i.e., it begins with the lowest level module.
In this testing higher level modules are tested first and then lower-level modules and then the modules are integrated accordingly.	In this testing lower level modules are tested first and then top level modules and then the modules are integrated accordingly.
Here stubs are used to simulate the submodule. If the invoked submodule is not developed means stub works as a momentary replacement.	Here drivers are used to simulate the submodule. If the invoked submodule is not developed means driver works as a momentary replacement.
Beneficial if the significant defect occurs toward the top of the program.	Beneficial if the significant defect occurs toward the bottom of the program.
The main module is designed first and then the submodules are called from it.	The submodules are designed first and then are integrated into the main function.
It is implemented on structural-oriented programming.	It is implemented on object-oriented programming.
The complexity of this testing is simple.	The complexity of this testing is complex and data intensive.
It works on big to small components.	It works on small to big components.
Stub modules must be produced	Driver modules must be produced.

20) and 21)

Level	Focus	Process Areas
Optimizing	***Continuous process improvement***	**Organizational innovation and deployment** **Causal analysis and resolution**
Quantitatively managed	***Quantitative management***	**Organizational process performance** **Quantitative project management**
Defined	***Process standardization***	**Requirements development** **Technical solution** **Product integration** **Verification** **Validation** **Organizational process focus** **Organizational process definition** **Organizational training** **Integrated project management** **Integrated supplier management** **Risk management** **Decision analysis and resolution** **Organizational environment for integration** **Integrated teaming**
Managed	***Basic project management***	**Requirements management** **Project planning** **Project monitoring and control** **Supplier agreement management** **Measurement and analysis** **Process and product quality assurance** **Configuration management**
Performed		

*Write if asked the table explain various levels in ur own words

But the next explanation and diagram is imp for 21st question

CMM was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University in 1987.

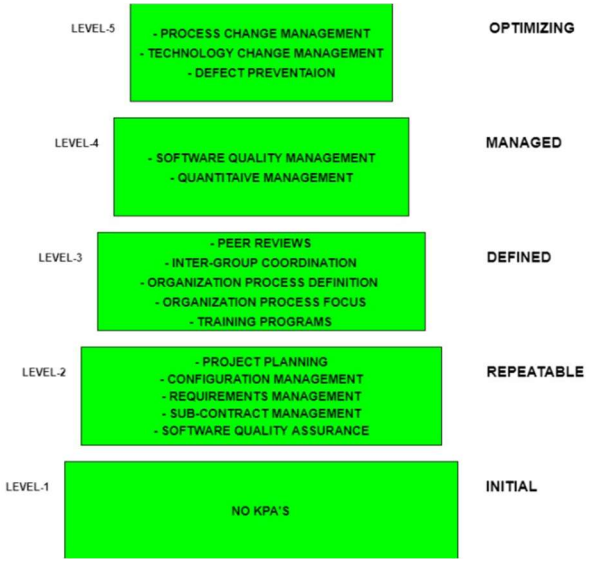
It is not a software process model. It is a framework that is used to analyze the approach and techniques followed by any organization to develop software products.

It also provides guidelines to further enhance the maturity of the process used to develop those software products.

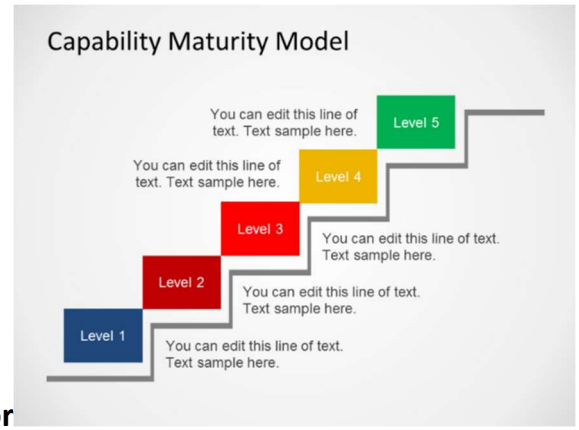
It is based on profound feedback and development practices adopted by the most successful organizations worldwide.

This model describes a strategy for software process improvement that should be followed by moving through 5 different levels.

Each level of maturity shows a process capability level. All the levels except level-1 are further described by Key Process Areas (KPA's).



or



Key Process Areas (KPA's)

Each of these KPA's defines the basic requirements that should be met by a software process in order to satisfy the KPA and achieve that level of maturity.

Conceptually, key process areas form the basis for management control of the software project and establish a context in which technical methods are applied, work products like models, documents, data, reports, etc. are produced, milestones are established, quality is ensured and change is properly managed.

*Write smthng abt CMM and CMM on how to integrate it in ur own words for 20th question and start writing the answer below

A software process, no matter how well conceived, will not succeed without talented, motivated software people.

The People Capability Maturity Model “is a road map for implementing workforce practices that continuously improve the capability of an organization’s workforce”.

Developed in the mid-1990s and refined over the intervening years, the goal of the People CMM is to encourage continuous improvement of generic workforce knowledge (called “core competencies”), specifi c software engineering and project management skills (called “workforce competencies”), and process-related abilities.

Like the CMM, CMMI, and related SPI frameworks, the People CMM defines a set of five organizational maturity levels that provide an indication of the relative sophistication of workforce practices and processes.

These maturity levels are tied to the existence (within an organization) of a set of key process areas (KPAs).

An overview of organizational levels and related KPAs is shown in The People CMM complements any SPI framework by encouraging an organization to nurture and improve its most important asset—its people.

As important, it establishes a workforce atmosphere that enables a software organization to “attract, develop, and retain outstanding talent”.

CMM-Capability Mature Model

CMMI-Capability Mature Model Intergration

SPI-Software Process Improvement

KRUTHIK B(1DA19CS077 B SECTION)

23)*explain that spiral model in ur own words to make the answer look big

Verification and validation:

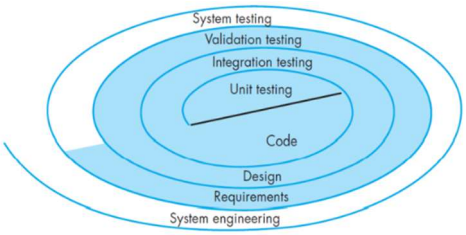
- Verification refers to the set of tasks that ensure that software correctly implements a specific function.
- Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
- Verification and validation include a wide array of SQA activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing

Organizing for software testing:

- The software developer is always responsible for testing the individual units of the program, ensuring that each performs the function or exhibits the behavior for which it was designed.
- The role of an independent test group (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present.
- The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted.

Software Testing Strategy:

- Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established.
- Moving inward along the spiral, you come to design and finally to coding.
- A strategy for software testing may also be viewed in the context of the spiral



Criteria for completion of testing:

- The cleanroom software engineering approach suggests statistical use techniques that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population.

26)

First half

STEPS

To establish a software metrics program, these are the steps:

1. Identify your business goals.

2. Identify what you want to know or learn.

3. Identify your sub goals.

4. Identify the entities and attributes related to your sub goals.

5. Formalize your measurement goals.

6. Identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals.

7. Identify the data elements that you will collect to construct the indicators that help answer your questions.

8. Define the measures to be used, and make these definitions operational.

9. Identify the actions that you will take to implement the measures.

10. Prepare a plan for implementing the measures.

Software supports business functions, differentiates computer-based systems or products, or acts as a product in itself, goals defined for the business can almost always be traced downward to specific goals at the software engineering level. For example, consider the Safe Home product.

Working as a team, software engineering and business managers develop a list of prioritized business **GOALS**:

1. Improve our customers' satisfaction with our products.

2. Make our products easier to use.

3. Reduce the time it takes us to get a new product to market.

4. Make support for our products easier.

5. Improve our overall profitability.

*For second half refer text book(from page 709 to 714)

27) Is there

Human Resources

The planner begins by evaluating software scope and selecting the skills required to complete development.

Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client-server) are specified.

For relatively small projects (a few person-months), a single individual may perform all software engineering tasks, consulting with specialists as required.

For larger projects, the software team may be geographically dispersed across a number of different locations.

Reusable Software

Resources Component-based software engineering (CBSE) emphasizes reusability—that is, the creation and reuse of software building blocks.

Such building blocks, often called components, must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

Bennatan suggests four software resource categories that should be considered as planning proceeds: Off-the-shelf components.

Existing software that can be acquired from a third party or from a past project.

COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.

Full-experience components Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project.

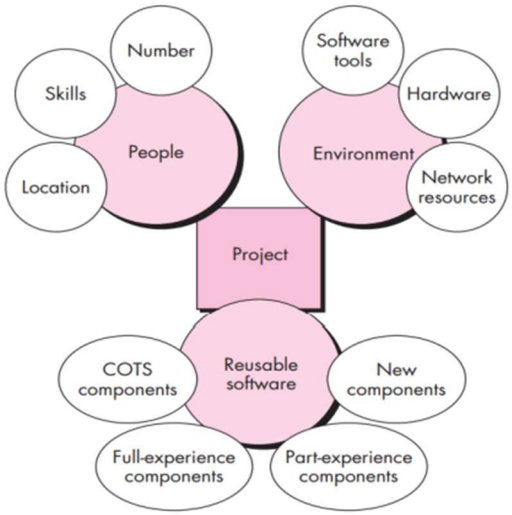
Environmental Resources

The environment that supports a software project, often called the software engineering environment (SEE), incorporates hardware and software.

Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.

Because most software organizations have multiple constituencies that require access to the SEE, you must prescribe the time window required for hardware and software and verify that these resources will be available.

FIGURE 26.1
Project resources



28)

The empirical data that support most estimation models are derived from a limited sample of projects. For this reason, no estimation model is appropriate for all classes of software and in all development environments. Therefore, you should use the results obtained from such models judiciously

An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC(Line of Code) or FP(Function Point Metric)

The Structure of Estimation Models

A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form given below

$$E = A + B \times (e_v)^C$$

where A, B, and C are empirically derived constants, E is effort in person-months, and e (v) is the estimation variable (either LOC or FP).

In addition to the relationship noted in Equation above, the majority of estimation models have some form of project adjustment component that enables E to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment).

A quick examination of any empirically derived model indicates that it must be calibrated for local needs.

The COCOMO II Model

In his classic book on software engineering economics, Barry Boehm [Boe81] introduced a hierarchy of software estimation models bearing the name COCOMO, for COnstructive COst MOdel. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry.

It has evolved into a more comprehensive estimation model, called COCOMO II.

Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address different "stages" of the software process.

Like all estimation models for software, the COCOMO II models require sizing information.

Three different sizing options are available as part of the model hierarchy: object points, 10 function points, and lines of source code.

The Software Equation

The software equation is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project.

The model has been derived from productivity data collected for over 4,000 contemporary software projects. Based on these data, we derive an estimation model of the form given below

$$E = \frac{\text{LOC} \times B^{0.333}}{P^3} \times \frac{1}{t^4}$$

Where,

E effort in person-months or person-years

t project duration in months or years

B "special skills factor" 11

P "productivity parameter" that refl ects: overall process maturity and management practices, the extent to which good software engineering practices are used, the level of programming languages used, the state of the software environment, the skills and experience of the software team, and the complexity of the application

Typical values might be P = 2,000 for development of real-time embedded software, P = 10,000 for telecommunication and systems software, and P = 28,000 for business systems applications. ***KRUTHIK B(1DA19CS077 B SECTION)***

The productivity parameter can be derived for local conditions using historical data collected from past development efforts.

KINDA IMPORTANT

BASIS PATH TESTING

Four topic in it:

Flow graph Notation

Independent Program Paths

Deriving test Cases(4 steps)-PDL and Flow Graph

Graph Matrices

*Refer Txt book page no(500-507) right overview of each

CONTROL STRUCTURE TESTING

*Simple text book page number (507 and 508)

KRUTHIK B(1DA19CS077 B SECTION)

Extra Questions

(KRUTHIK B 1DA19CS077 B SECTION)

1)Will exhaustive testing guarantee that the program is 100 percent correct? Illustrate with suitable examples.

A)No, even exhaustive testing will not guarantee that the program is 100 percent correct.

There are too many variables to consider.

Consider this...

Installation testing - did the program install according to the instructions?

Integration testing - did the program work with all of the other programs on the system without interference, and did the installed modules of the program integrate and workwith other installed modules?

Function testing - did each of the program functions work properly?

Unit testing - did the unit work as a standalone as designed, and did the unit work when placed in the overall process?

User Acceptance Testing - did the program fulfill all of the user requirements and work per the user design?

Performance testing - did the program perform to a level that was satisfactory and could it carry the volume load placed upon it?

While these are just the basic tests for an exhaustive testing scenario, you could keep testing beyond these tests using destructive methods, white box internal program testing, establish program exercises using automated scripts, etc.

The bottom line is... testing has to stop at some point in time.

Either the time runs out that was allotted for testing, or you gain a confidence level that the program is going to work. (Of course, the more you test, the higher your confidence level).

I don't know anyone that would give a 100% confidence level that the program is 100% correct, (to do so is to invite people to prove you wrong and they will come back with all kinds of bugs you never even considered).

However, you may be 95% confident that you found most all of the major bugs. Based upon this level of confidence, you would then place the program into production use - always expecting some unknown bug to be found.

2)The decisions made by senior management can have a significant impact on the effectiveness of a software engineering team. Provide five examples to illustrate that this is true.

A)*Don't know

3)Describe any three system testing types with real time examples.

A)**Usability Testing**– mainly focuses on the user's ease to use the application, flexibility in handling controls and ability of the system to meet its objectives

Load Testing– is necessary to know that a software solution will perform under real-life loads.

Regression Testing– involves testing done to make sure none of the changes made over the course of the development process have caused new bugs. It also makes sure no old bugs appear from the addition of new software modules over time.

Recovery testing – is done to demonstrate a software solution is reliable, trustworthy and can successfully recoup from possible crashes.

Migration testing- is done to ensure that the software can be moved from older system infrastructures to current system infrastructures without any issues.

Functional Testing – Also known as functional completeness testing, Functional Testing involves trying to think of any possible missing functions. Testers might make a list of additional functionalities that a product could have to improve it during functional testing.

Hardware/Software Testing – IBM refers to Hardware/Software testing as "HW/SW Testing". This is when the tester focuses his/her attention on the interactions between the hardware and software during system testing.

*Write some examples of ur own

4)Briefly explain various decomposition techniques used during software project estimations.

Software Sizing –

The accuracy of a software project estimate is predicated on a number of things:

(1) the degree to which you have properly estimated the size of the product to be built;

(2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects);

(3) the degree to which the project plan reflects the abilities of the software team; and

(4) the stability of product requirements and the environment that supports the software engineering effort.

Because a project estimate is only as good as the estimate of the size of the work to be accomplished, software sizing represents your first major challenge as a planner.

In the context of project planning, size refers to aquantifiable outcome of the software project.

Problem-Based Estimation –

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common.

We begin with a bounded statement of software scope and from this statement attempt to decompose the statement of scope into problem functions that can each be estimated individually.

LOC or FP (the estimation variable) is then estimated for each function.

Alternatively, you may choose another component for sizing, such as classes or objects, changes, or business processes affected.

Baseline productivity metrics are then applied to the appropriate estimation variable, and cost or effort for the function is derived.

Function estimates are combined to produce an overall estimate for the entire project.

Process-Based Estimation –

The most common technique for estimating a project is to base the estimate on the process that will be used.

That is, the process is decomposed into a relatively small set of activities, actions, and tasks and the effort required to accomplish each is estimated.

Like the problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope.

A series of framework activities must be performed for each function.

Once problem functions and process activities are melded, you estimate the effort (e.g., person-months) that will be required to accomplish each software process activity for each software function.

Estimation with Use Cases –

Use cases provide a software team with insight into software scope and requirements.

Once use cases have been developed they can be used to estimate the projected "size" of a software project.

However, developing an estimation approach with use cases presents challenge.

Use cases are described using many different formats and styles and represent an external view (the user's view) of the software.

Therefore, they can be written at many different levels of abstraction.

Use cases do not address the complexity of the functions and features that are described, and they can describe complex behavior (e.g., interactions) that involve many functions and features.

Reconciling Estimates –

The estimation techniques discussed in the preceding sections result in multiple estimates that must be reconciled to produce a single estimate of effort, project duration, or cost.

What happens when agreement between estimates is poor? The answer to this question requires a reevaluation of information used to make the estimates. Widely divergent estimates can often be traced to one of two causes:

(1) the scope of the project is not adequately understood or has been misinterpreted by the planner, or

(2) productivity data used for problem-based estimation techniques is inappropriate for the application, obsolete (in that it no longer accurately reflects the software engineering organization), or has been misapplied.

We should determine the cause of divergence and then reconcile the estimates.

Extra Questions

(KRUTHIK B 1DA19CS077 B SECTION)

1)Will exhaustive testing guarantee that the program is 100 percent correct? Illustrate with suitable examples.

A)No, even exhaustive testing will not guarantee that the program is 100 percent correct.

There are too many variables to consider.

Consider this...

Installation testing - did the program install according to the instructions?

Integration testing - did the program work with all of the other programs on the system without interference, and did the installed modules of the program integrate and workwith other installed modules?

Function testing - did each of the program functions work properly?

Unit testing - did the unit work as a standalone as designed, and did the unit work when placed in the overall process?

User Acceptance Testing - did the program fulfill all of the user requirements and work per the user design?

Performance testing - did the program perform to a level that was satisfactory and could it carry the volume load placed upon it?

While these are just the basic tests for an exhaustive testing scenario, you could keep testing beyond these tests using destructive methods, white box internal program testing, establish program exercises using automated scripts, etc.

The bottom line is... testing has to stop at some point in time.

Either the time runs out that was allotted for testing, or you gain a confidence level that the program is going to work. (Of course, the more you test, the higher your confidence level).

I don't know anyone that would give a 100% confidence level that the program is 100% correct, (to do so is to invite people to prove you wrong and they will come back with all kinds of bugs you never even considered).

However, you may be 95% confident that you found most all of the major bugs. Based upon this level of confidence, you would then place the program into production use - always expecting some unknown bug to be found.

2)The decisions made by senior management can have a significant impact on the effectiveness of a software engineering team. Provide five examples to illustrate that this is true.

A)*Don't know

3)Describe any three system testing types with real time examples.

A)**Usability Testing**– mainly focuses on the user's ease to use the application, flexibility in handling controls and ability of the system to meet its objectives

Load Testing– is necessary to know that a software solution will perform under real-life loads.

Regression Testing– involves testing done to make sure none of the changes made over the course of the development process have caused new bugs. It also makes sure no old bugs appear from the addition of new software modules over time.

Recovery testing – is done to demonstrate a software solution is reliable, trustworthy and can successfully recoup from possible crashes.

Migration testing- is done to ensure that the software can be moved from older system infrastructures to current system infrastructures without any issues.

Functional Testing – Also known as functional completeness testing, Functional Testing involves trying to think of any possible missing functions. Testers might make a list of additional functionalities that a product could have to improve it during functional testing.

Hardware/Software Testing – IBM refers to Hardware/Software testing as "HW/SW Testing". This is when the tester focuses his/her attention on the interactions between the hardware and software during system testing.

*Write some examples of ur own

4)Briefly explain various decomposition techniques used during software project estimations.

Software Sizing –

The accuracy of a software project estimate is predicated on a number of things:

(1) the degree to which you have properly estimated the size of the product to be built;

(2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects);

(3) the degree to which the project plan reflects the abilities of the software team; and

(4) the stability of product requirements and the environment that supports the software engineering effort.

Because a project estimate is only as good as the estimate of the size of the work to be accomplished, software sizing represents your first major challenge as a planner.

In the context of project planning, size refers to aquantifiable outcome of the software project.

Problem-Based Estimation –

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common.

We begin with a bounded statement of software scope and from this statement attempt to decompose the statement of scope into problem functions that can each be estimated individually.

LOC or FP (the estimation variable) is then estimated for each function.

Alternatively, you may choose another component for sizing, such as classes or objects, changes, or business processes affected.

Baseline productivity metrics are then applied to the appropriate estimation variable, and cost or effort for the function is derived.

Function estimates are combined to produce an overall estimate for the entire project.

Process-Based Estimation –

The most common technique for estimating a project is to base the estimate on the process that will be used.

That is, the process is decomposed into a relatively small set of activities, actions, and tasks and the effort required to accomplish each is estimated.

Like the problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope.

A series of framework activities must be performed for each function.

Once problem functions and process activities are melded, you estimate the effort (e.g., person-months) that will be required to accomplish each software process activity for each software function.

Estimation with Use Cases –

Use cases provide a software team with insight into software scope and requirements.

Once use cases have been developed they can be used to estimate the projected "size" of a software project.

However, developing an estimation approach with use cases presents challenge.

Use cases are described using many different formats and styles and represent an external view (the user's view) of the software.

Therefore, they can be written at many different levels of abstraction.

Use cases do not ad dress the complexity of the functions and features that are described, and they can describe complex behavior (e.g., interactions) that involve many functions and features.

Reconciling Estimates –

The estimation techniques discussed in the preceding sections result in multiple estimates that must be reconciled to produce a single estimate of effort, project duration, or cost.

What happens when agreement between estimates is poor? The answer to this question requires a reevaluation of information used to make the estimates. Widely divergent estimates can often be traced to one of two causes:

(1) the scope of the project is not adequately understood or has been misinterpreted by the planner, or

(2) productivity data used for problem-based estimation techniques is inappropriate for the application, obsolete (in that it no longer accurately reflects the software engineering organization), or has been misapplied.

We should determine the cause of divergence and then reconcile the estimates.