14CS501

III/IV B.Tech (Regular) DEGREE EXAMINATION
COMPUTER SCIENCE AND ENGINEERING

Software Engineering
Schema

1. Answer the following.
   a) What is software process model?
   Process models define a set of distinct activities, actions, tasks, milestones, and work products that are used to design high quality software.
   b) State the design principles.
      1. Design must be traceable to the analysis model
      2. Always consider architecture
      3. Focus on the design of data as it is as important as a design
      4. Interfaces (both user and internal) must be designed
      5. User interface design should be tuned to the needs of the end-user.
      6. Component-level design should exhibit functional independence
      7. Components should be loosely coupled to one another and to the external environment.
      8. Design representation (models) should be easily understood.
      9. The design model should be developed iteratively. With each iteration, the designer should strive for greater simplicity.
      <u>Give marks for any two principles</u>

   c) Why to perform blackbox testing ?
   To uncover the interface errors and incorrect or missing functions, errors in data structures, performance errors.

   d) What is requirement engineering?
   Requirement Engineering helps software engineers to better understand the problem they will work to solve.
   e) Define SAQ group.
   The software quality assurance (SQA) group is a team of people with the necessary training and skills to ensure that all necessary actions are taken during the development process so that the resulting software confirms to established technical requirements.

   f) What is a component?
   Component is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.

   g) What is the importance of user interface?
   In simple words, a great user interface design is one that helps users accomplish any task in the most efficient way possible.

   h) What is ISO 9000 quality standard?
   It describes a quality assurance system in generic terms that can be applied to any business regardless of products or services offered.

i) Explain benchmark testing?
Benchmark testing is a part of the software development life cycle that involves both developers and database administrators (DBAs) to determine current performance and make changes to improve the performances of the same.

j) State the goals of quality assurance?
Goal of quality assurance is to provide management with data necessary to be informed about product quality there by gaining insight and confidence about the product.

k) What are the elements of analysis?
Scenario based elements, flow oriented elements, class based elements, behavioural elements.

l) What is system modelling?
System model is an engineering process which tests whether the focus is on world view or detailed view.

UNIT-1

2.

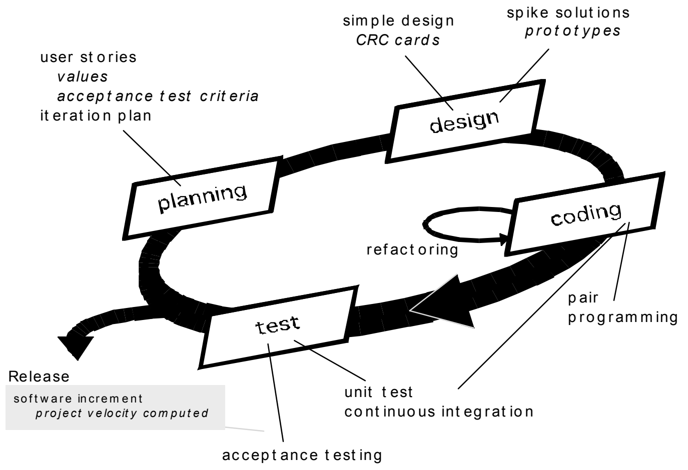
a. **Briefly explain the emergence of agile development approaches**. (6M)

Agile Process Models

- Is driven by customer descriptions of what is required (scenarios)
- Recognizes that plans are short-lived
- Develops software iteratively with a heavy emphasis on construction activities
- Delivers multiple 'software increments'
- Adapts as changes occur

Extreme Programming (XP) :

The most widely used agile process, originally proposed by Kent Beck. It encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding and testing.



Planning: Begins with the creation of "user stories" and then placed on an index card.

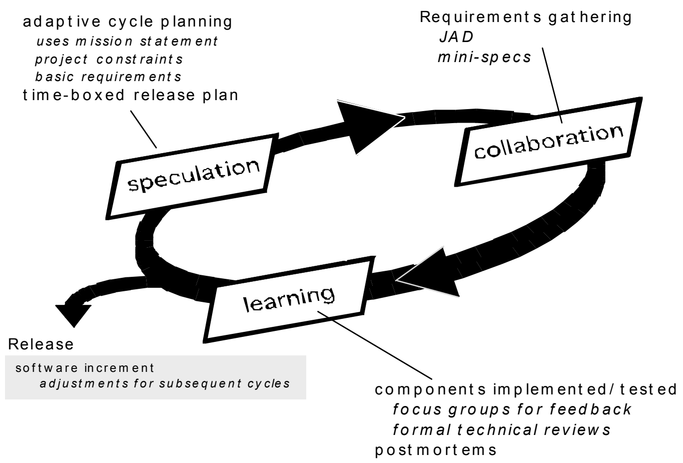
The customer assigns a value to the story based on the overall business value of the function. Agile "XP" team assesses each story and assigns a cost "measured in development weeks."

Design: XP Follows the KIS(S) principle. A simple design is always preferred over a more complex representation. Encourage the use of CRC (class-responsibility collaboration) cards which identify and organize the O-O classes that are relevant to the current software increment. For difficult design problems, suggests the creation of "spike solutions"—a design prototype that is implemented and evaluated. Encourages "refactoring"—an iterative refinement of the internal program design that controls the code modifications by suggesting small design changes that may improve the design.

Coding: Recommends the construction of a unit test for a story before coding commences**.** Encourages "pair programming" where two programmers work together at one workstation to create code for a story. "Two heads better than one."

Testing: All unit tests are executed daily which provides the XP team with a continual indication of progress and also can raise warning flags early if things are going awry. Acceptance tests are defined by the customer "user stories" and executed to assess customer visible functionality.

2. **Adaptive Software Development** (ASD):ASD is a technique that is used for building complex software and systems.
ASD life lifecycle incorporates 3 phases: speculation, collaboration, and learning.
Speculation: An adaptive cycle-planning is conducted where it uses the customer's mission statement, project constraints (delivery dates, user description) and basic requirements.

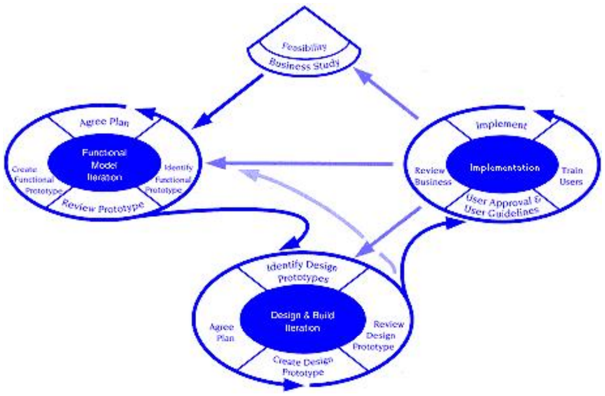


Collaboration: People working together must trust one another to:
1.criticize without animosity
2.assist without resentment
3.work as hard or harder as they do
4.have the skill set to contribute to the work at hand
5.communicate problems or concerns in a way that leads to effective action

Learning: Learning will help them to improve their level of real understanding.

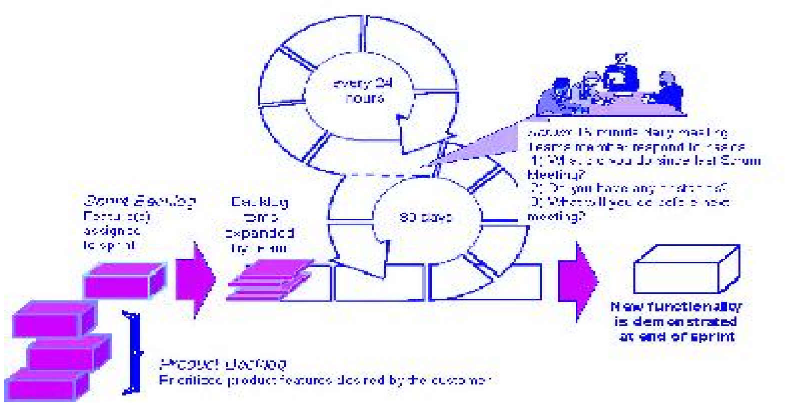
Dynamic System Development Method (DSDM):

Dynamic System Development Method is an agile S/W development approach that provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment.



It defines three different iterative cycles preceded by two additional life cycles activities:
Feasibility Study: Business requirements and constraints.
Business Study: Establishes req. that will allow the application to provide business value.
Functional Model Iteration: Produce iterative prototypes.
Design and build iteration: Revisit prototyping.
Implementation: Include the latest prototype into the operational environment.

Scrum:
Scrum features are
Development work is partitioned into "packets"
Testing and documentation are on-going as the product is constructed
Work occurs in "sprints framework activities" and is derived from a "backlog requirements that provide business values" of existing requirements
Meetings are very short and sometimes conducted without chairs
"**Demos**" are delivered to the customer with the time-box allocated



every 24 hours
30 days
Backlog items expanded by team
Meeting?
meeting?
New functionality is demonstrated at end of sprint

Scrum Process Flow (used with permission)

b. What is the difference between program and software (6M)

The terms software and program are used interchangeably as they often refer to the same thing in daily usage.
Even though they very close to synonymous, there are still minor differences between them should distinguish one from the other.
Software is a very broad term that is used to identify programs, data, and other related files that are used to accomplish certain tasks in a computer or any other device that is performs a computing task.
In this sense, we can say that even a program is also a software.
But in the broader meaning of the word, a program is any set of instructions that are executed by a machine.

3. a. **What are the fundamental activities of a software process? (6M)**

A Process Framework:
Software process models can be prescriptive or agile, complex or simple, all-encompassing or targeted, but in every case, five key activities must occur. The framework activities are applicable to all projects and all application domains, and they are a template for every process model.

```
Software process
 Process framework
         Umbrella activities
                 Framework activity #1
                                 Software Engineering action
```

Each framework activity is populated by a set of S/W eng actions – a collection of related tasks that produces a major S/W eng work product (design is a S/W eng action).
Each action is populated with individual work tasks that accomplish some part of the work implied by the action.
The following generic process framework is applicable to the vast majority of S/W projects.
Communication: involves heavy communication with the customer and other stakeholders and encompasses requirements gathering.

Planning: Describes the technical tasks to be conducted, the risks that are likely, resources that will be required, the work products to be produced and a work schedule.

Modeling: encompasses the creation of models that allow the developer and customer to better understand S/W req. and the design that will achieve those req.

Construction: combines code generation and the testing required uncovering errors in the code.

Deployment: deliver the product to the customer who evaluates the delivered product and provides feedback.

Each S/W eng action is represented by a number of different task sets – each a collection of S/W eng work tasks, related work products, quality assurance points, and project milestones.

The task set that best accommodates the needs of the project and the characteristics of the team is chosen.

The framework described in the generic view of S/W eng is complemented by a number of umbrella activities. Typical activities include:

S/W project tracking and control: allows the team to assess progress against the project plan and take necessary action to maintain schedule.

Risk Management: Assesses the risks that may affect the outcome of the project or the quality.

Software quality assurance: defines and conducts the activities required to ensure software quality.

Formal Technical Review: uncover and remove errors before they propagate to the next action.

Measurement: defines and collects process, project, and product measures that assist the team in delivering S/W that meets customers' needs.

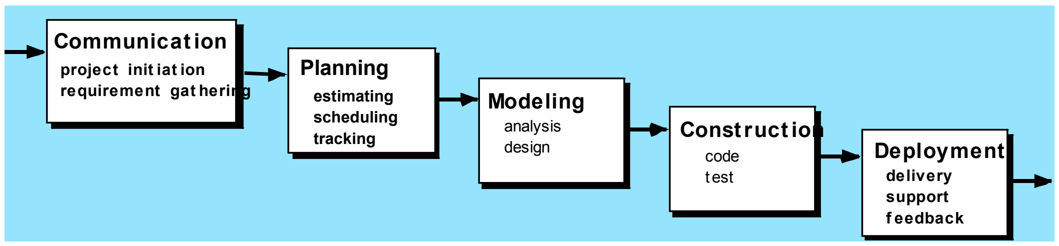
Software configuration management: Manages the effect of change throughout the S/W process

Reusability management: defines criteria for work product reuse.

Work product preparation and production: encompasses the activities required to create work products such as models, documents, etc.

a. Explain waterfall model and reasons for failure of waterfall model. (6M)

The Waterfall Model



Also known as "classic life cycle" means systematic approach and sequential approach.

It will be used when well defined adaptions or enhancements to a existing system.

Many people dismiss the waterfall as obsolete and it certainly does have problems. But this model can still be used in some situations.

Reasons: Among the problems that are sometimes encountered when the *waterfall* model is applied are:

- A Real project rarely follows the sequential flow that the model proposes. Change can cause confusion as the project proceeds.
- It is difficult for the customer to state all the requirements explicitly. The waterfall model requires such demand.
- The customer must have patience. A working of the program will not be available until late in the project time-span.

UNIT-2

4.

a. What is design engineering and what are the underlying concepts that lead to good design? (8M)

The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight.

Design Concepts
1.Abstraction
At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided.
As we move through different levels of abstraction, we work to create procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. An example of a procedural abstraction would be the word *open* for a door.
A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open,* we can define a data abstraction called **door**.
2.Architecture
Software architecture alludes to the "overall structure of the software and the ways in which the structure provides conceptual integrity for a system."
The goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which detailed design activities are constructed.
A set of architectural patterns enable a software engineer to reuse design-level concepts.
The architectural design can be represented using one or more of a number of different models.
Structural models, Framework models, Dynamic models, Process models
Functional models .

3.Patterns
A design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine:

1. whether the pattern is applicable to the current work,
2. whether the pattern can be reused, and
3. whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

4.Modularity

Software architecture and design patterns embody *modularity*; that is, software is divided into separately named and addressable components, sometimes called modules that are integrated to satisfy problem requirements.

It is easier to solve a complex problem when you break it into manageable pieces. "Divide-and-conquer"

Don't over-modularize. The simplicity of each small module will be overshadowed by the complexity of integration "Cost".

5 Information Hiding

It is about controlled interfaces. Modules should be specified and design so that information (algorithm and data) contained within a module is inaccessible to other modules that have no need for such information.

The use of Information Hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to other location within the software.

6 Functional Independence

Functional independence is a key to good design, and design is the key to software quality.
Independence is assessed using two qualitative criteria: **cohesion** and **coupling.**
Cohesion is an indication of the relative functional strength of a module.
Coupling is an indication of the relative interdependence among modules.
A cohesive module should do just one thing.
Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world "lowest possible".

7 Refinement

It is the elaboration of detail for all abstractions. It is a top down strategy.
A program is developed by successfully refining levels of procedural detail.
Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction enables a designer to specify procedure and data and yet suppress low-level details.

Refinement helps the designer to reveal low-level details as design progresses.

8. Refactoring

It is a reorganization technique that simplifies the design of a component without changing its function or behavior. When software is re-factored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed data structures, or any other design failures that can be corrected to yield a better design.

b. **Explain validation**. **(4M)**

The purpose of requirements validation is to make sure that the customer and developer agree on details of the software requirements (or prototype) before beginning the major design work. This implies that both the customer and developer need to be present during the validation process.
As each element of the analysis model is created, it is examined for consistency, omissions, and ambiguity.
The requirements represented by the model are prioritized by the customer and grouped within requirements packages that will be implemented as software increments and delivered to the customer.
A review of the analysis model addresses the following questions:

- Is each requirement consistent with the overall objective for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?

5.

a. **Explain cardinality and modality in data modeling concepts.** (4M)

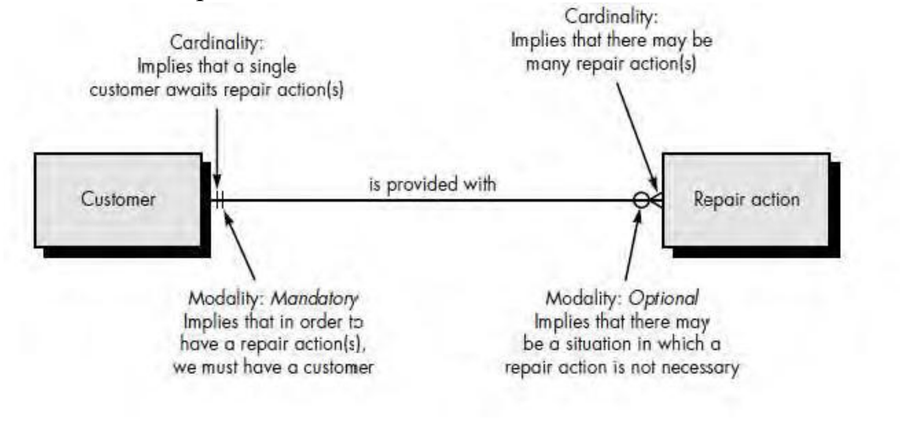
Cardinality:
The data model must be capable of representing the number of occurrences objects in a given relationship. Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object].

Cardinality is usually expressed as simply 'one' or 'many.' For example, a husband can have only one wife, while a parent can have many children.
Taking into consideration all combinations of 'one' and 'many,' two [objects] can be related as

- One-to-one (1:1)—An occurrence of [object] 'A' can relate to one and only one occurrence of [object] 'B,' and an occurrence of 'B' can relate to only one occurrence of 'A.'
- One-to-many (1: N)—One occurrence of [object] 'A' can relate to one or many occurrences of [object] 'B, but an occurrence of ' B' can relate to only one occurrence of 'A.' For example, a mother can have many children, but a child can have only one mother.

- Many-to-many (M:N)—An occurrence of [object] 'A' can relate to one or more occurrences of 'B,' while an occurrence of 'B' can relate to one or more occurrences of 'A.' For example, an uncle can have many nephews, while a nephew can have many uncles.

Modality: The modality of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional.
The modality is 1 if an occurrence of the relationship is mandatory.
Figure illustrates the relationship, cardinality, and modality between the data objects customer and repair action.


Cardinality:
Implies that a single customer awaits repair action(s)
Cardinality:
Implies that there may be many repair action(s)
Customer
is provided with
Repair action
Modality: Mandatory
Implies that in order to have a repair action(s), we must have a customer
Modality: Optional
Implies that there may be a situation in which a repair action is not necessary

b. **Explain Eliciting requirements** (8M)

Collaborative Requirements Gathering all apply some variation on the following guidelines:

- meetings are conducted and attended by both software engineers and customers
- rules for preparation and participation are established
- an agenda is suggested
- a "facilitator" (can be a customer, a developer, or an outsider) controls the meeting
- a "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room or virtual forum) is used
- the goal is to identify the problem, propose elements of the solution negotiate different approaches, and specify a preliminary set of solution requirements

Quality Function Deployment (QFD)
In this technique the customer is asked to prioritize the requirements based on their relative value to the users of the proposed system.

QFD is a technique that translates the needs of the customer into technical requirements for software engineering. QFD identifies three types of requirements:

Normal requirements: These requirements reflect objectives and goals stated for a product or system during meetings with the customer.

Expected requirements:
These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them.

Exciting requirements: These requirements reflect features that go beyond the customer's expectations and prove to be very satisfying when present.
In meeting with the customer, function deployment is used to determine the value of each function that is required for the system.
Information deployment identifies both the data objects and events that the systems must consume and produce.
Finally, task deployment examines the behavior of the system within the context of its environment.
And value analysis determines the relative priority of requirements determined during each of the three deployments.

User Scenarios:
As requirements are gathered an overall version of system functions and features begins to materialize
Developers and users create a set of scenarios that identify a thread of usage for the system to be constructed in order for the software engineering team to understand how functions and features are to be used by end-users.

Elicitation Work Products:
For most systems, the work product includes:

- A statement of need and feasibility.
- A bounded statement of scope for the system or product.
- A list of customers, users, and other stakeholders who participated in requirements elicitation.
- A description of the system's technical environment.
- A list of requirements (preferably organized by function) and the domain constraints that apply to each.
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.

Any prototypes developed to better define requirements.

6.

a. Write examples of three data abstractions. (4M).

Data Abstraction
At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided.
As we move through different levels of abstraction, we work to create procedural and data abstractions.

1. **A procedural abstraction** refers to a sequence of instructions that have a specific and limited function. An example of a procedural abstraction would be the word open for a door.
2. **A data abstraction** is a named collection of data that describes a data object. In the context of the procedural abstraction open, we can define a data abstraction called **door**.
3. **Control Abstraction**: Control abstraction involves the use of subprograms and related concepts control flows

b. Explain software architecture. (8M)

Software Architecture

The term “software architecture” as a framework made up of the system structures that comprise the software components, their properties, and the relationships among these components.

The goal of the architectural model is to allow the software engineer to view and evaluate the system as a whole before moving to component design.

Architecture:

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

(1) Analyze the effectiveness of the design in meeting its stated requirements,

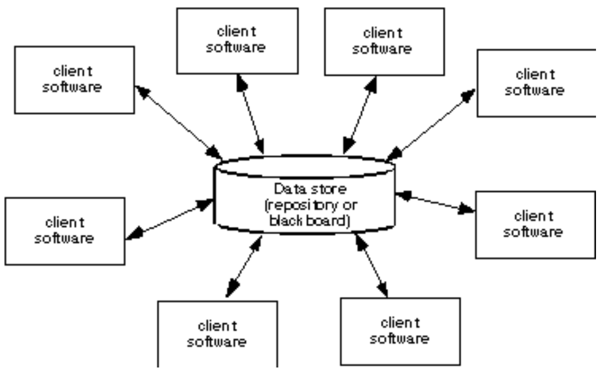
(2) Consider architectural alternatives at a stage when making design changes is still relatively easy, and

(3) Reduce the risks associated with the construction of the software.

A Brief Taxonomy of Architectural Styles

Data-Centered Architecture:

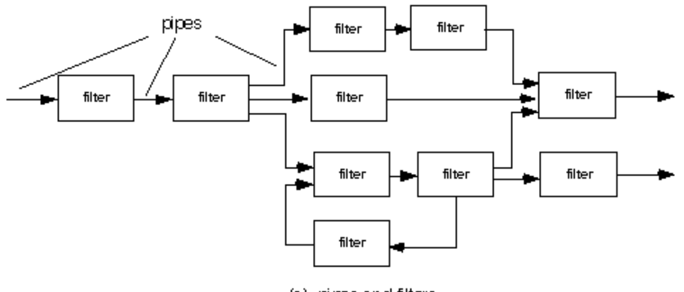
A data store resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. This architecture promotes integrability. Existing components can be changed and new client components can be added to the architecture without concern about other clients.



Data-flow Architecture

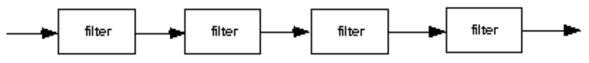
This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data.

A pipe and filter structure has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next.



(a) pipes and filters

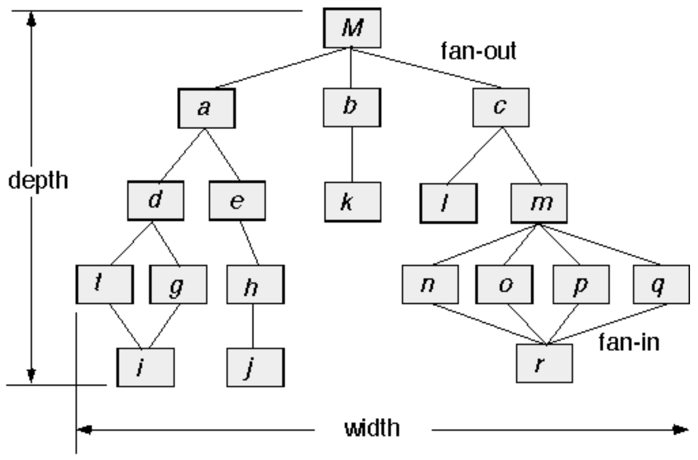
Call and Return architecture



(b) batch sequential

The architectural style enables a S/W designer to achieve a program structure that is relatively easy to modify and scale.

- Main program/subprogram architecture.
- Remote procedure-call architecture.



Architectural Patterns

A S/W architecture may have a number of architectural patterns that address issues such as concurrency, persistence, and distribution.

Concurrency—applications must handle multiple tasks in a manner that simulates parallelism

- *operating system process management* pattern
- *task scheduler* pattern

Persistence—Data persists if it survives past the execution of the process that created it. Persistent data are stored in a database or file and may be read and modified by other processes at a later time.

Two patterns are common:

- a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture
- an *application level persistence* pattern that builds persistence features into the application architecture

Distribution— the manner in which systems or components within systems communicates with one another in a distributed environment, and the nature of the communication that occurs.

- A broker acts as a ‘middle-man’ between the client component and a server component.

7.

a. Explain cohesion and coupling methods?

Cohesion:

Cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself.

Levels of cohesion

a. Functional: occurs when a module performs one and only one computation and then returns a result.
b. Layer: occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

c. Communicational: All operations that access the same data are defined within one class.
d. Sequential: Components or operations are grouped in a manner that allows the first to provide input to the next and so on.
e. Procedural: Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked.
f. Temporal: Operations that are performed to reflect a specific behavior or state.
g. Utility: Components, classes, or operations that exist within the same category but are otherwise unrelated are grouped together.

Coupling:
Conventional view: The degree to which a component is connected to other components and to the external world.
OO View: A qualitative measure of the degree to which classes are connected to one another. Keep coupling as low as possible.
Level of coupling

a. Content: Occurs when one component "superstitiously modifies data that is internal to another component. "Violates Information hiding"
b. Common: Occurs when a number of components all make use of a global variable.
c. Control: Occurs when operation A() invokes operation B() and passes a control flag to B. The control flag then "directs" logical flow within B.
d. Stamp: Occurs when Class B is declared as a type for an argument of an operation of Class A. Because Class B is now a part of the definition of Class A, modifying the system becomes more complex.
e. Data: Occurs when operations pass long strings of data arguments. "Testing and maintenance becomes more difficult."
f. Routine call: Occurs when one operation invokes another.
g. Type use: Occurs when component A uses a data type defined in component B
h. Inclusion or import: Occurs when component A imports or includes a package or the content of component B.
i. External: Occurs when a component communicates or collaborates with infrastructure components (O/S function).

b. What are three types of golden rules?
The three "golden rules" are:
1.Place the user in control
2.Reduce the user's memory load
3.Make the interface consistent
These golden rules actually form the basis for a set of user interface design principles that guide this important software design action.

1. **Place the User in Control**
•Define interaction modes in a way that does not force a user into unnecessary or undesired actions. The user should always be able to enter and exit the mode with little or no effort.
•Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided by using keyboard commands, mouse movements, digitizer pen or voice recognition commands.

•Allow user interaction to be interruptible and undoable. A user should be able to interrupt a sequence of actions to do something else without losing the work that has been done. The user should always be able to "undo" any action.
•Streamline interaction as skill levels advance and allow the interaction to be customized. Allow to design a macro if the user is to perform the same sequence of actions repeatedly.
•Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. A user should never be required to type O/S commands from within application software.
•Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing.

2.**Reduce the User's Memory Load**
Whenever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall.
•Reduce demand on short-term memory. Provide visual cues that enable a user to recognize past actions, rather than having to recall them.
•Establish meaningful defaults. A user should be able to specify individual preferences; however, a reset option should be available to enable the redefinition of original default values.
•Define shortcuts that are intuitive. "Example: Alt-P to print.
•The visual layout of the interface should be based on a real world metaphor. Enable the user to rely on well-understood visual cues, rather than remembering an arcane interaction sequence. For a bill payment system use a check book and check register metaphor to guide the user through the process.
•Disclose information in a progressive fashion. The interface should be organized hierarchically. The information should be presented at a high level of abstraction.
3. **Make the Interface Consistent**
The interface should present and acquire information in a consistent manner:
1.All visual information is organized according to a design standard that is maintained throughout all screen displays,
2.Input mechanisms are constrained to a limited set that is used consistently throughout the application,
3.Mechanisms for navigating from task to task are consistently defined and implemented.
A set of design principles that help make the interface consistent:
Allow the user to put the current task into a meaningful context.
Maintain consistency across a family of applications

8.

a. **List and explain different types of testing done during the testing phase**. (8M)
Types of Testing:
1. White Box Testing
2. Black Box Testing

White Box Testing:

a. Basis Path Testing

• White-box technique usually based on the program flow graph
• The cyclomatic complexity of the program computed from its flow graph using the formula $V(G) = E - N + 2$ or by counting the conditional statements in the

PDL representation and adding 1
• Determine the basis set of linearly independent paths (the cardinality of this set id the program cyclomatic complexity)
• Prepare test cases that will force the execution of each path in the basis set.

Control Structure Testing:
• White-box techniques focusing on control structures present in the software
• Condition testing (e.g. branch testing) focuses on testing each decision statement in a software module, it is important to ensure coverage of all logical combinations of data that may be processed by the module (a truth table may be helpful)
• Data flow testing selects test paths based according to the locations of variable definitions and uses in the program (e.g. definition use chains)
• Loop testing focuses on the validity of the program loop constructs

Black Box Testing:

Graph-based Testing Methods:
• Black-box methods based on the nature of the relationships (links) among the program objects (nodes), test cases are designed to traverse the entire graph
• Transaction flow testing (nodes represent steps in some transaction and links represent logical connections between steps that need to be validated)
• Finite state modeling (nodes represent user observable states of the software and links represent transitions between states)
• Data flow modeling (nodes are data objects and links are transformations from one data object to another)
• Timing modeling (nodes are program objects and links are sequential connections between these objects, link weights are required execution times)

Equivalence Partitioning:
• Black-box technique that divides the input domain into classes of data from which test cases can be derived
• An ideal test case uncovers a class of errors that might require many arbitrary test cases to be executed before a general error is observed

• **Equivalence class guidelines**:
1. If input condition specifies a range, one valid and two invalid equivalence classes are defined
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class is defined
4. If an input condition is Boolean, one valid and one invalid equivalence class is defined

Boundary Value Analysis:
• Black-box technique that focuses on the boundaries of the input domain rather than its center
• **BVA guidelines**:
1. If input condition specifies a range bounded by values a and b, test cases should

include a and b, values just above and just below a and b
2. If an input condition specifies and number of values, test cases should be exercise the minimum and maximum numbers, as well as values just above and just below the minimum and maximum values
3. Apply guidelines 1 and 2 to output conditions, test cases should be designed to produce the minimum and maxim output reports
4. If internal program data structures have boundaries (e.g. size limitations), be certain to test the boundaries

Orthogonal Array Testing:
• Black-box technique that enables the design of a reasonably small set of test cases that provide maximum test coverage
• Focus is on categories of faulty logic likely to be present in the software component (without examining the code)
• Priorities for assessing tests using an orthogonal array
1. Detect and isolate all single mode faults
2. Detect all double mode faults
3. Mutimode faults

b. Explain metrics for software quality. (4M)

A definition of software quality metrics is:- A measure of some property of a piece of software or its specifications. Basically, as applied to the software product, a software metric measures a characteristic of the software.

Some common software metrics are:-
Source lines of code.
Cyclomatic complexity is used to measure code complexity.
Function point analysis (FPA) is used to measure the size (functions) of software.
Bugs per lines of code.
Code coverage, measures the code lines that are executed for a given set of software tests.
Cohesion measures how well the source code in a given module work together to provide a single function.
Coupling measures how well two software components are data related, i.e. how independent they are.
The above list is only a small set of software metrics, the important points to note are:-
They are all measurable, that is they can be quantified.
They are all related to one or more software quality characteristics.

9.
a. Explain formal approaches to SQA (6M)

Software Quality Assurance (SQA) is defined as a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes, and procedures.SQA includes the process of assuring that standards and procedures are established and are followed throughout the software life cycle.
Software Quality Assurance is an umbrella activity that is applied throughout the software process.

SQA activities:

1. Prepares an SQA plan for the project
2. Participates in the development of the projects software process description.
3. Reviews software engineering activities to verify compliance with the defined
4. software process
5. Audits designated software products to verify compliance with those defined as part of the software process.
6. Reviews software engineering activities to verify compliance with the defined software process.
7. Ensures the deviations in software work and work products are documented and handled according to a documented procedure.
8. Records any non compliance to senior management

b. What are advantages and disadvantages of Software testing? (6M)

Advantages:

1) Concise: As simple as possible and no simpler.
2) Self-Checking: Test reports its own results; needs no human interpretation.
3) Repeatable: Test can be run many times in a row without human intervention.
4) Robust: Test produces same result now and forever. Tests are not affected by changes in the external environment.
5) Sufficient: Tests verify all the requirements of the software being tested.
6) Necessary: Everything in each test contributes to the specification of desired behavior.
7) Clear: Every statement is easy to understand.
8) Efficient: Tests run in a reasonable amount of time.
9) Specific: Each test failure points to a specific piece of broken functionality; unit test failures provide "defect triangulation".
10) Independent: Each test can be run by itself or in a suite with an arbitrary set of other tests in any order.
11) Maintainable: Tests should be easy to understand and modify and extend.
12) Traceable: To and from the code it tests and to and from the requirements.

Disadvantages of Automation Testing:
Though the automation testing has many advantages, it has its own disadvantages too.

Some of the disadvantages are:

1) Proficiency is required to write the automation test scripts.

2) Debugging the test script is major issue. If any error is present in the test script, sometimes it may lead to deadly consequences.
3) Test maintenance is costly in case of playback methods. Even though a minor change occurs in the GUI, the test script has to be re-recorded or replaced by a new test script.
4) Maintenance of test data files is difficult, if the test script tests more screens.