Question Paper solved : BCS051 :::::

Explain Rapid Application Development (RAD) approach for Software Development. What are the advantages and disadvantages of RAD approach over Waterfall model.

- Rapid Application Development (RAD) is a software development approach that emphasizes quick development and iteration of prototypes over rigorous planning and testing. It focuses on user feedback and active user involvement throughout the development process, enabling faster delivery of functional software.
- The RAD approach typically involves the following stages:
    - **Requirement Planning**: Gathering initial requirements and identifying the core features needed for the application.
    - **User Design**: Creating prototypes based on user feedback and requirements, allowing users to interact with early versions of the software.
    - **Construction**: Developing the actual software through iterative cycles, refining prototypes, and incorporating user feedback.
    - **Cutover**: Transitioning from development to production, including testing and implementation.
- Advantages of RAD over the Waterfall model:
    - **Faster Development**: RAD allows for quicker delivery of functional prototypes, enabling faster time-to-market compared to the more linear Waterfall model.
    - **User Feedback**: Continuous user involvement and feedback lead to better alignment with user needs, resulting in more satisfactory final products.
    - **Flexibility**: Changes can be incorporated more easily during the development process, allowing teams to adapt to evolving requirements without significant delays.
    - **Reduced Risk**: Early prototypes help identify issues and risks sooner, enabling teams to address them before full-scale development.
    - **Enhanced Collaboration**: Emphasizes teamwork and communication among developers, stakeholders, and users, leading to better project outcomes.
- Disadvantages of RAD compared to the Waterfall model:
    - **Less Predictability**: RAD's iterative nature can make it harder to predict timelines and budgets, potentially leading to scope creep if not managed properly.
    - **Requires User Commitment**: RAD relies heavily on user involvement and feedback, which may not always be available or consistent, impacting the development process.

- **Quality Control Challenges**: Rapid iterations may lead to less emphasis on documentation and formal testing, risking the overall quality of the final product.
- **Limited Scalability**: RAD is often more suitable for smaller projects; larger projects may struggle with coordination and management in an iterative approach.
- **Technical Debt**: The focus on speed may lead to shortcuts in coding and design, resulting in technical debt that could affect maintenance and future development.

---

Explain any five key features of Function Oriented Design.

**Function-Oriented Design (FOD) is a traditional approach in software engineering where the primary focus is on the functions or processes that the system will perform. It is often used in procedural programming, where the system is broken down into a set of functions that perform specific tasks.**

## Five Key Features of Function-Oriented Design:

- **Modularity:**
  **The design is divided into smaller, independent modules or functions, each responsible for performing a specific task or operation. These modules can be developed, tested, and maintained separately.**
- **Top-down Approach:**
  **Function-oriented design follows a top-down approach, where the system is first designed as a whole, and then broken down into smaller and more manageable parts or functions. This helps in simplifying the complexity of the system.**
- **Data Flow Representation:**
  **It emphasizes the flow of data between functions. In FOD, data is passed between functions to carry out operations, which is represented using flowcharts or data flow diagrams (DFDs).**
- **Sequential Process:**
  **In FOD, processes are executed in a sequential manner, meaning each function performs its task in a specific order. The flow of execution is linear, and control moves from one function to the next in a step-by-step process.**
- **Emphasis on Functions:**
  **The focus is on what the system should do rather than how the system will be structured or organized. Functions define the operations, and the design is built around these functions with minimal concern for the underlying data structures.**

- **Modularity**: Function-oriented design emphasizes breaking down a system into smaller, manageable modules or functions. Each module performs a specific task, which promotes reusability and easier maintenance.
- **Top-Down Approach**: This design methodology typically follows a top-down approach, where the overall system is first outlined, and then the functions are developed in a hierarchical manner. High-level functions are decomposed into lower-level sub-functions, making it easier to understand and implement.
- **Data Flow**: Function-oriented design emphasizes the flow of data between functions. Data is processed by various functions, and the design highlights how data moves through the system, often illustrated using data flow diagrams (DFDs).
- **Control Structures**: The design incorporates well-defined control structures (like loops, conditionals, and sequences) that dictate how functions are executed. This ensures that the logical flow of the program is clear and structured, making it easier to follow and maintain.
- **Function Reusability**: The design encourages the reuse of functions across different parts of the application or even in different projects. By creating generalized functions that can be called multiple times, development efficiency increases, and redundancy is minimized.

---

Define Data Dictionary. Consider a Data Element Student. What information about Student is stored in Data Dictionary ? 10

- A data dictionary is a centralized repository that contains definitions, descriptions, and characteristics of data elements within a database or information system. It provides a comprehensive overview of the data structure, ensuring consistency, clarity, and effective management of data across applications.
- For the data element "Student," the following information is typically stored in the data dictionary:

1. **Data Element Name**: The name of the data element (e.g., "Student").
2. **Data Type**: Specifies the type of data stored (e.g., integer, string, date) for attributes like student ID, name, date of birth, etc.

3. **Length**: Defines the maximum length of the data element, particularly for string types (e.g., name could be up to 50 characters).
4. **Description**: A brief description of what the data element represents, such as "Information related to individual students enrolled in the system."
5. **Constraints**: Specifies any restrictions or rules applicable to the data element, such as "Student ID must be unique" or "Date of birth cannot be a future date."
6. **Default Value**: Indicates any default value assigned to the data element if no specific value is provided (e.g., status could default to "active").
7. **Domain**: Defines the valid values or range for the data element (e.g., enrollment status could be limited to "active," "inactive," or "graduated").
8. **Relationships**: Information about how this data element relates to other elements or entities in the system (e.g., a Student may have a relationship with Courses or Instructors).
9. **Usage**: Details about where and how the data element is used within the application (e.g., in student enrollment forms, reports, etc.).
10. **Audit Information**: Metadata about the data element, including who created it, when it was last modified, and any relevant change history to track updates over time.

---

Explain the methodology for identification of objects in Object Oriented Design with an example.

- The methodology for identifying objects in Object-Oriented Design (OOD) involves analyzing the problem domain to determine the key entities and their relationships. This process typically follows several steps:
- **Understanding Requirements**: Begin by gathering and analyzing the requirements of the system. This includes understanding what the system is supposed to do, who the users are, and what problems need to be solved.
- **Identifying Key Use Cases**: Determine the key use cases that represent the interactions between users and the system. Use cases help in identifying the main functionalities required and provide context for the objects.
- **Analyzing the Problem Domain**: Examine the problem domain to identify the main entities involved. Look for nouns in the requirements and use cases, as these often represent potential objects or classes.
- **Defining Attributes and Behaviors**: For each identified object, define its attributes (data members) and behaviors (methods or functions). This helps to clarify what each object will represent and how it will interact with other objects.

- **Establishing Relationships**: Determine the relationships between identified objects. This includes associations, aggregations, and inheritances that define how objects interact with one another within the system.
- **Iterating and Refining**: The identification process is iterative. As the design evolves, revisit and refine the identified objects based on further analysis and feedback. This ensures that all necessary objects are included and accurately defined.

## Example:

Consider the development of a library management system:

1. **Understanding Requirements**: Gather requirements such as managing books, tracking users, and handling loans.
2. **Identifying Key Use Cases**: Identify use cases like "Borrow a Book," "Return a Book," and "Search for Books."
3. **Analyzing the Problem Domain**: Key entities in this scenario might include:
    - **Book**
    - **User**
    - **Loan**
4. **Defining Attributes and Behaviors**:
    - **Book**:
        - Attributes: title, author, ISBN, publication year
        - Behaviors: checkAvailability(), borrow(), return()
    - **User**:
        - Attributes: userID, name, contactInfo
        - Behaviors: register(), updateProfile(), viewLoanHistory()
    - **Loan**:
        - Attributes: loanID, book, user, dueDate
        - Behaviors: createLoan(), closeLoan(), checkOverdue()
5. **Establishing Relationships**:
    - A **User** can have multiple **Loans** (one-to-many relationship).
    - A **Loan** is associated with one **Book** and one **User**.
6. **Iterating and Refining**: Review the identified objects and their relationships. If more requirements arise, like adding reservations or fines, new objects can be defined or existing ones modified accordingly.

By following this methodology, the design team can systematically identify and define the objects necessary for the library management system, leading to a well-structured and efficient object-oriented design.

---

Explain the terms ‘Module Testing’ and ‘Integration Testing’.

**Module Testing**:

- Also known as unit testing, module testing involves testing individual components or modules of a software application in isolation. The main goal is to verify that each module functions correctly according to its specifications.
- Each module is tested independently to ensure it performs its intended tasks and meets the required criteria.
- It focuses on the internal logic and behavior of the module, including its inputs, outputs, and handling of various scenarios.
- Module testing is typically conducted by developers and may involve writing test cases that cover various aspects of the module’s functionality.
- This type of testing helps identify bugs early in the development process, making it easier and less costly to fix issues before moving on to larger components.

**Integration Testing**:

- Integration testing is the process of testing the interactions and interfaces between multiple modules or components after they have been combined. The aim is to ensure that the modules work together as intended and that data flows correctly between them.
- It checks for issues that may arise when different modules are integrated, such as data format mismatches, communication problems, or interface errors.
- Integration testing can be done incrementally, where individual modules are combined and tested step by step, or through a big bang approach, where all modules are integrated at once and tested together.
- This type of testing is crucial for identifying problems that may not be evident during module testing, as it focuses on the combined behavior of the modules rather than their individual performance.
- Integration testing is usually performed by a dedicated testing team and can involve various techniques, such as top-down, bottom-up, or sandwich (mixed) approaches.

---

Explain various challenges for successful debugging.

- **Complexity of Code**: Modern software systems often have complex architectures and interdependencies, making it difficult to track down the source of bugs. Understanding how different components interact can be challenging.
- **Incomplete or Missing Documentation**: Lack of proper documentation can hinder debugging efforts. Without clear information on the code's intended behavior or design, developers may struggle to identify where things are going wrong.
- **Inadequate Testing**: Insufficient testing may result in undetected bugs. If edge cases or specific scenarios are not tested, debugging becomes more difficult when those cases arise in production.
- **Dynamic Behavior**: Software may exhibit different behaviors based on user input or environment, leading to intermittent bugs that are hard to reproduce. This dynamic nature complicates the debugging process.
- **Concurrency Issues**: In multi-threaded or distributed systems, race conditions and deadlocks can occur. These issues can be difficult to diagnose, as they may not manifest consistently during testing.
- **Tool Limitations**: Debugging tools may have limitations in terms of features, ease of use, or compatibility with the codebase. If the tools do not provide sufficient insights, developers may face challenges in identifying and resolving issues.
- **Resistance to Change**: Developers may be reluctant to change or refactor code that is not behaving correctly, fearing it could introduce new bugs. This resistance can hinder the debugging process and prolong resolution times.
- **Time Constraints**: Tight deadlines and pressure to deliver can lead to rushed debugging efforts. Under time constraints, developers may miss important clues or skip necessary testing, resulting in unresolved issues.
- **Poorly Written Code**: Legacy code or code that lacks clarity and structure can be difficult to debug. Poor coding practices, such as inconsistent naming conventions or lack of modularity, can obscure the root cause of problems.
- **Insufficient Knowledge or Skills**: Debugging requires a deep understanding of the code, frameworks, and systems involved. If developers lack the necessary knowledge or experience, it can hinder their ability to identify and fix bugs effectively.

---

List any five software testing tools indicating the testing phase in which they can be employed.

1. **Selenium**

    - **Testing Phase**: Functional Testing
    - Selenium is an open-source tool used for automating web browsers. It allows testers to write test scripts in various programming languages to perform functional testing of web applications.
2. **JMeter**
    - **Testing Phase**: Performance Testing
    - Apache JMeter is a tool designed for load testing and performance measurement of web applications. It helps simulate multiple users to assess how the application performs under heavy load.
3. **Postman**
    - **Testing Phase**: API Testing
    - Postman is a collaboration platform for API development that allows testers to create, send, and analyze HTTP requests to ensure that APIs function correctly and meet specified requirements.
4. **JIRA**
    - **Testing Phase**: Bug Tracking and Management
    - JIRA is primarily a project management tool that is widely used for tracking bugs and issues during the software development life cycle. It helps teams manage and prioritize defects for resolution.
5. **JUnit**
    - **Testing Phase**: Unit Testing
    - JUnit is a widely used framework for writing and running repeatable tests in Java. It allows developers to perform unit testing, ensuring that individual components or methods function as intended.

---

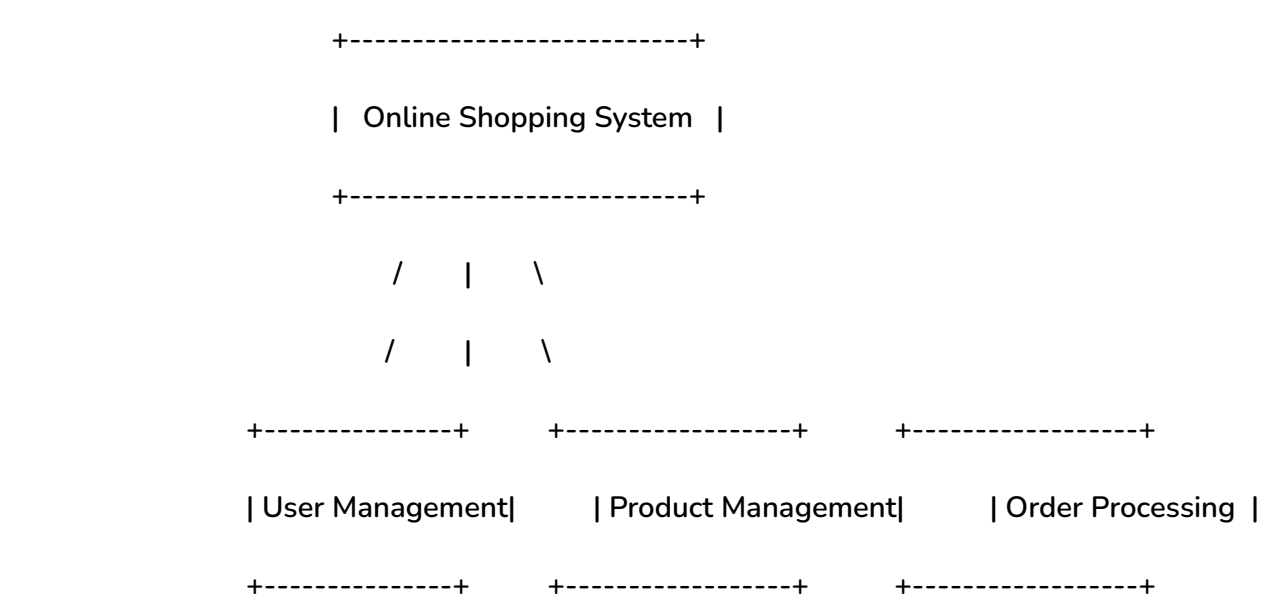
What is a Structure Chart ? Explain with the help of an example.

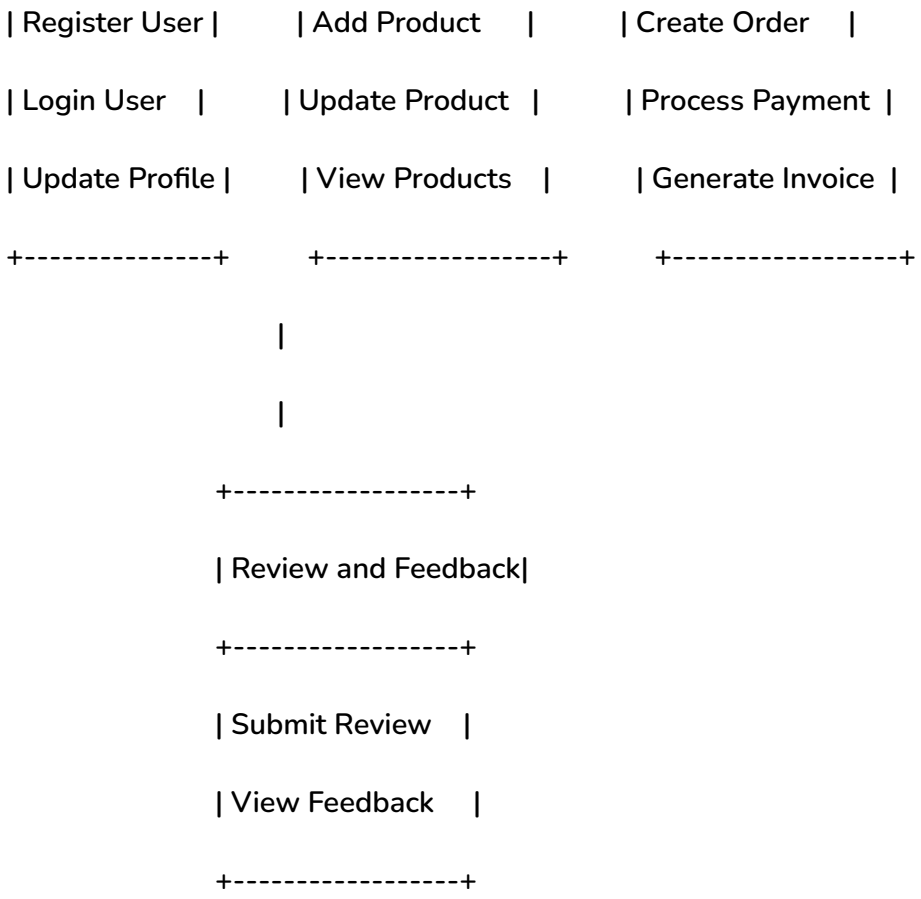
- A structure chart is a visual representation of the hierarchical structure of a software system, focusing on the relationships between various modules or components. It helps in understanding how different parts of the system interact and communicate with each other. Structure charts are commonly used in structured programming and design methodologies to illustrate the organization of functions and modules.
- The structure chart consists of boxes and connecting lines. Each box represents a module or function, while the lines indicate the flow of control and data between these modules. The hierarchy is typically top-down, with the main module at the top and sub-modules branching out below it.

## Example:

Consider a simple online shopping system. The structure chart for this system might look as follows:

1. **Main Module: Online Shopping System**
    - This is the top-level module that controls the overall functionality of the system.
2. **Sub-module 1: User Management**
    - **Functions**:
        - Register User
        - Login User
        - Update Profile
3. **Sub-module 2: Product Management**
    - **Functions**:
        - Add Product
        - Update Product
        - View Products
4. **Sub-module 3: Order Processing**
    - **Functions**:
        - Create Order
        - Process Payment
        - Generate Invoice
5. **Sub-module 4: Review and Feedback**
    - **Functions**:
        - Submit Review
        - View Feedback






---

Explain the terms “Black Box Testing” and “White Box Testing”.

- **Black Box Testing**:
  - Black box testing is a testing method where the tester evaluates the functionality of an application without any knowledge of its internal code structure or implementation.
  - The focus is on the input and output of the software rather than how the system processes those inputs.
  - Testers create test cases based on requirements and specifications, assessing whether the software behaves as expected under various conditions.
  - This approach is useful for functional testing, system testing, and acceptance testing.

    - It allows testers to identify issues related to the user interface, data handling, and overall functionality without needing to understand the underlying code.
- **White Box Testing**:
    - White box testing, also known as clear box testing or structural testing, involves testing the internal workings and structure of an application.
    - Testers have access to the source code and design documents, allowing them to create test cases that validate the logic, flow, and performance of the software.
    - This method focuses on code coverage, testing specific paths, conditions, loops, and branches in the code.
    - White box testing is typically used for unit testing and integration testing, as it helps identify logical errors, security vulnerabilities, and performance bottlenecks within the code.
    - It requires a good understanding of programming languages and code structure, making it more technical than black box testing.

### Key Differences:

- **Perspective**: Black box testing focuses on external behavior; white box testing focuses on internal logic.
- **Knowledge Required**: Black box testing does not require knowledge of code; white box testing requires detailed knowledge of the code.
- **Test Design**: Black box testing is based on requirements and specifications; white box testing is based on code structure and logic.
- **Use Cases**: Black box testing is typically used for functional and system testing; white box testing is used for unit and integration testing.

---

How will you ensure that the software developed by you meets the Quality benchmarks ? Define the term “Software Quality”.

To ensure that the software developed meets quality benchmarks, the following practices can be implemented:

1. **Requirements Analysis**: Clearly define and document the software requirements. Involve stakeholders to ensure that all needs and expectations are captured.

2. **Code Reviews**: Conduct regular code reviews to identify issues early in the development process. Peer reviews help ensure adherence to coding standards and improve code quality.
3. **Testing**: Implement a comprehensive testing strategy that includes unit testing, integration testing, system testing, and user acceptance testing (UAT). Use automated testing tools to increase efficiency and coverage.
4. **Continuous Integration/Continuous Deployment (CI/CD)**: Adopt CI/CD practices to automate the build and testing process. This allows for rapid feedback on code changes and ensures that quality is maintained throughout the development lifecycle.
5. **Quality Metrics**: Define and track quality metrics such as defect density, test coverage, and customer satisfaction. Use these metrics to assess quality and identify areas for improvement.

**Software Quality**:

- Software quality refers to the degree to which a software product meets the specified requirements, satisfies user needs, and adheres to defined standards.
- It encompasses various attributes, including functionality, reliability, usability, efficiency, maintainability, and portability.
- High software quality ensures that the software performs as intended, is free of critical defects, and provides a positive user experience.
- Achieving software quality requires a combination of good development practices, rigorous testing, and continuous improvement throughout the software development lifecycle.

---

In Object Oriented Design, list the common utility objects and criteria for identifying utility objects

**Common Utility Objects in Object-Oriented Design**:

1. **Logger**: Manages logging information for debugging and monitoring purposes, capturing events, errors, and system behavior.
2. **Configuration Manager**: Handles application configuration settings, allowing for dynamic adjustments without changing the code.
3. **Cache Manager**: Provides functionality for caching frequently accessed data to improve performance and reduce load on resources.

4. **Error Handler**: Centralizes error handling and reporting, managing exceptions and ensuring consistent responses to errors.
5. **Date and Time Utilities**: Offers functions for date and time manipulation, such as formatting, parsing, and calculating durations.
6. **Validator**: Validates input data against specific rules or criteria to ensure data integrity and correctness.
7. **Data Access Object (DAO)**: Abstracts the data access layer, providing a standardized way to interact with databases or data sources.
8. **File Manager**: Manages file operations, including reading, writing, and deleting files, while handling file paths and formats.

**Criteria for Identifying Utility Objects**:

1. **Reusability**: Utility objects should be designed to be reused across different parts of the application or in multiple projects. If a function is needed in various modules, it may warrant a utility object.
2. **Common Functionality**: Identify tasks or operations that are commonly needed throughout the application. Utility objects often encapsulate generic functions that are not tied to specific business logic.
3. **Separation of Concerns**: Utility objects should separate common functionality from core business logic. This helps maintain cleaner code and promotes better organization.
4. **Simplicity and Clarity**: Utility objects should have a clear and straightforward interface. They should provide simple methods that are easy to understand and use, enhancing overall code readability.
5. **Statelessness**: Many utility objects are stateless, meaning they do not maintain any internal state between method calls. This makes them easier to use and ensures predictable behavior.

---

Explain various Debugging strategies.

- **Print Debugging**: This strategy involves inserting print statements in the code to display variable values and program flow at various points during execution. It helps to trace how data changes and where the program may be deviating from expected behavior.
- **Interactive Debugging**: Using debugging tools or integrated development environments (IDEs), developers can set breakpoints to pause execution at specific

lines. This allows for inspection of variable states, step-by-step execution, and real-time analysis of the program's behavior.

- **Logging**: Implementing a logging framework enables the application to write detailed logs to a file or console, capturing important information such as errors, warnings, and informational messages. Analyzing logs helps identify issues without modifying the code.
- **Rubber Duck Debugging**: This technique involves explaining the code and the problem to an inanimate object (like a rubber duck) or another person. Articulating the problem often leads to new insights or the realization of mistakes that were previously overlooked.
- **Divide and Conquer**: This strategy involves isolating sections of code to narrow down where the problem lies. By disabling or removing parts of the code, developers can determine which segment is causing the issue and focus their debugging efforts accordingly.
- **Code Reviews**: Engaging peers in reviewing the code can help identify errors or potential issues that the original developer might have missed. Fresh eyes can provide different perspectives and catch bugs more effectively.
- **Automated Testing**: Writing automated tests for code can help catch bugs early. When tests fail, developers can quickly locate the issue based on the failing test case and its associated code.
- **Static Analysis Tools**: These tools analyze code without executing it, identifying potential errors, code smells, or security vulnerabilities. They help enforce coding standards and best practices, reducing the likelihood of bugs.
- **Version Control Comparison**: When a bug is identified, comparing the current code with previous versions in a version control system can help identify what changes might have introduced the issue. This aids in understanding the evolution of the code and pinpointing the origin of the bug.
- **Memory Analysis Tools**: In languages where memory management is manual (like C or C++), tools that analyze memory usage can help identify memory leaks, buffer overflows, and other memory-related issues that can cause unpredictable behavior.

---

What is function oriented design of software system ? Explain its advantages and disadvantages.

- Function-oriented design emphasizes functionality by organizing software around functions or operations.
- The system is viewed as a collection of functions that transform input data into output.
- The design process involves breaking down the system into smaller, manageable functions and defining their interactions.

Advantages:

- Simplicity: Breaks complex systems into smaller functions, making them easier to understand.
- Clear Structure: Provides better documentation and easier navigation through the codebase.
- Ease of Testing: Functions can be tested individually, aiding in bug isolation.
- Reusability: Functions can be reused across different parts of the application, reducing duplication.
- Better Maintenance: Allows changes to specific functions without affecting the entire system.

Disadvantages:

- Limited Focus on Data: May complicate managing complex data structures.
- Tight Coupling: Functions can become closely linked, reducing system flexibility.
- Difficulties in Scaling: Managing function interactions can be challenging in larger systems.
- Inadequate Representation of Real-World Entities: May not accurately model complex relationships.
- Higher Development Costs: Increased complexity in coordinating multiple functions in large projects.

---

Explain unit testing and module testing with the help of suitable example for each.

**Module Testing**

Definition: Module testing, also known as unit testing, is the process of testing individual components or functions of a program separately to ensure they work correctly on their own.

- It focuses on testing specific parts or units of a program in isolation.

- Each module is checked to verify its functionality without interaction with other parts.
- The aim is to catch bugs within individual functions or methods before integrating them.
- Typically performed by developers as they build each module.
- It helps identify small errors early, making them easier to fix before combining with other modules.

**Integration Testing**
Definition: Integration testing is the process of testing how different modules or components of a software application work together after being combined.

- It verifies that modules interact correctly and exchange data as expected.
- Focuses on identifying issues that arise when modules are integrated, such as data flow or compatibility issues.
- Helps to catch errors in how components work together, which might not show up in module testing.
- Often done after module testing to ensure a smooth interaction between parts.
- Ensures the overall system functions as intended with all modules combined.

---

## Explain various challenges for successful debugging

1. Finding the exact cause of a bug can be difficult, especially in complex code with many parts interacting.
2. Some bugs may only appear under specific conditions, making them hard to reproduce and analyze.
3. Debugging can be time-consuming, especially if the codebase is large or not well-documented.
4. Errors in code can sometimes lead to misleading error messages, which complicate the debugging process.
5. Changes made to fix one bug can accidentally introduce new issues elsewhere in the program.
6. Debugging tools or environments may not fully support all code languages or frameworks, limiting their effectiveness.
7. Sometimes, bugs are caused by external dependencies, like libraries or hardware, which are harder to control and fix.
8. Concurrency issues, where multiple parts of a program run simultaneously, can create complex, hard-to-track bugs.

9. It can be challenging to identify the point where a problem started if multiple changes have been made to the code over time.
10. Lack of experience or familiarity with the code or programming language can make debugging more challenging and slow.

---

## Write a short note on Software Configuration Management.

Software Configuration Management (SCM) is a process that tracks and manages changes to software to ensure consistency and reliability. It helps teams control versioning, maintain code integrity, and collaborate effectively during development.

- Software Configuration Management (SCM) helps manage changes to software projects systematically.
- It ensures consistency and integrity of software by tracking and controlling changes.
- SCM maintains a complete history of modifications to the code, aiding in version control.
- It allows multiple developers to work together on the same project without conflicts.
- Change management in SCM ensures updates are reviewed and implemented in a controlled manner.
- Build management automates the process of compiling, testing, and deploying software.
- SCM helps identify and resolve bugs by tracking the source of changes.
- It enhances traceability by documenting who made changes and why they were made.
- Tools like Git, SVN, and Mercurial are commonly used to implement SCM processes.
- SCM improves collaboration, reduces errors, and ensures reliable software delivery.

---

Unified Modeling Language (UML) is a standardized visual language used to design and document software systems. It helps developers and stakeholders visualize, analyze, and communicate system designs effectively.

1. UML provides a way to represent systems through diagrams, making them easier to understand and plan.
2. It supports object-oriented concepts like classes, objects, and inheritance, making it ideal for object-oriented systems.
3. UML includes structural diagrams (like class diagrams and component diagrams) and behavioral diagrams (like use case and sequence diagrams).
4. A **class diagram** shows the structure of a system, including classes, attributes, methods, and relationships.
5. A **use case diagram** illustrates how users interact with the system through various functionalities.
6. A **sequence diagram** represents the flow of messages or interactions between objects over time.
7. UML is widely used for system modeling, software development, and process documentation in industries like IT, engineering, and business.
8. It is platform-independent, meaning it can be applied to any programming language or development environment.
9. UML helps improve communication among team members and stakeholders by providing a shared, visual understanding of the system.
10. Tools like Visual Paradigm, Lucidchart, and Enterprise Architect are commonly used for creating UML diagrams.

---

A **use case diagram** is a type of UML diagram that shows how users interact with a system. It focuses on the system's functionality and identifies the actors (users or external systems) and their relationships with use cases (tasks or actions).

1. It represents the **functional requirements** of a system.
2. **Actors** are external entities (people, devices, or systems) that interact with the system.
3. **Use cases** are the specific tasks or actions performed by the system in response to the actor.
4. The diagram uses ovals for use cases, stick figures for actors, and lines to show interactions between them.
5. Relationships like **include** (common functionality shared between use cases) and **extend** (optional or conditional functionality) can be shown.
6. Helps in understanding the scope of the system by outlining user needs and system functionalities.
7. It is useful in system design, especially during the requirements-gathering phase.

8. Examples include a user logging into a website, placing an order, or generating a report.
9. Use case diagrams focus on *what* the system does, not *how* it performs tasks.
10. Tools like Lucidchart, Microsoft Visio, and StarUML can be used to create use case diagrams.

---