OOAD-4 : UML Deployment and Component Diagram, GOF Design patterns and Iterative planning. Introduction to GRASP – Methodological approach to OO design, Architectural analysis and UML package Design.

GOF Design and Patterns:

* Gang of Four (GOF) Design patterns provide a catalog of proven solutions to common design problems in software development.
* The GOF Design patterns encourage best practices, code reusability, and the separation of concerns, aiding in the development of robust and scalable applications.
* The GOF Design patterns is a set of solutions to common problems we encounter in SDD.
* They were first introduced in the book "Design patterns: Elements of Reusabe object-oriented software", published in 1994.

Why they're called Gang of Four?

* The Gang of Four are four smart people who wrote a book about clever ways to solve common problems in computer programming.

Types of Gang of Four Design Patterns:

* The GOF patterns are set of 23 common software design patterns.

These patterns categorize into three main groups:

1. Creational Patterns.
2. Structural Patterns.
3. Behavioral Patterns.

* These patterns provide solutions to common design problems and

help make software systems more modular, flexible and maintainable.

1. Creational Design Patterns:

* These patterns help us create objects in a smart and organized way.
* They focus on the process of object creation in software development.
* These patterns make sure that we create things in a way that's not only easy but also flexible, so we can change them later if we need too.
* They hide the complicated details of how we put pieces together.

Types of Creational Design Patterns:

1. Factory Method Pattern :-
   → to make objects with flexibility.
   → having a blueprint for creating things.
   → Interface is defined to create different types of subclass.
   Ex: car manufacture factory.

2. Abstract Factory pattern :-
   → provides a way to create families of objects
   → Ensuring that everything you create fits together seamlessly.
   Ex: various models of same car.

3. Singleton pattern:
   → all about exclusivity.
   → It ensures that a class has just one instance.
   → The instance can be accessed from anywhere, making it handy for situations where you want a single point of control or coordination in your application.

4. Prototype Pattern:
   → Instead of creating something from scratch, use the existing one, saving time and resources.

5. Builder pattern:
   → like a set of instructions for making something complex.
   → It helps you create that complex thing step by step, one piece at a time.

6. Object pool pattern:
   → resource manager for reusable items.
   → The object pool keeps a collection of objects, like database connections or threads, and hand them out when needed.
   → This saves time and resources compared to creating and destroying objects frequently.

2. Structural Design Patterns:

* The pattern for putting together different objects and classes to build a bigger structure.
* Following a blueprint to construct a house.
* These patterns teach us how to combine the unique parts of a system in a way that's easy to change / expand without

affecting the entire system.

Types of Structural Design Patterns:

1. Adapter: Bridges incompatible interfaces.
2. Bridge: Separates abstraction from implementation.
3. Composite: Treats individual and grouped objects uniformly.
4. Decorator: Dynamically adds behavior to objects.
5. Facade: Simplifies complex system interactions.
6. Flyweight: Reduces memory usage by sharing objects.
7. Proxy: Controls access to another object.

Behavioral Design Patterns:

* These patterns define communication between objects, improving flexibility and modularity.
* These patterns enhance code organizations, flexibility and maintainability.

Types:-

1. Chain of Responsibility: passes requests through handlers.
2. Command: Encapsulates requests as objects.
3. Iterator: Sequentially accesses collection elements.
4. Mediator: Centralizes communication between objects.
5. Memento: Saves an object's state for future restoration.
6. Observer: Notifies dependent objects of changes.
7. State: changes behavior based on internal state.
8. Strategy: switches between algorithms dynamically.
9. Template Method: Defines a skeleton process with customizable steps.

GRASP:

Methodological approach to OO Design.

GRASP Design principles in OOAD:

* In OOAD General Responsibility Assignment Software Patterns (GRASP) play a crucial role in designing effective and maintainable software systems.
* GRASP offers a set of guidelines to and developers in assigning responsibilities to classes and objects in a way that promotes low coupling, high cohesion and overall robustness.
* By applying GRASP principles, developers can create software solutions that are flexible, scalable, and easier to maintain overtime.

GRASP principles:

* It includes several principles that guide the allocation of responsibilities in object-oriented design.

1. Creator:
   → Assign the responsibility of creating instances of a class to the class that has the most knowledge about when and how to create them.
2. Information Expert:
   → Assign a responsibility to the class that has the necessary information to fulfil it.
3. Low coupling:
   → Aim for classes to have minimal dependencies on each other.

1. High cohesion :
→ Ensure that responsibilities within a class are closely related and focused.

5. Controller :
→ Assign the responsibility of handling system events & coordinating activities to a controller class.

6. Pure Fabrication :
→ Introduce new classes to fulfill responsibilities without violating cohesion and coupling principles.

7. Indirection :
→ Use intermediaries & abstractions to decouple classes and promote flexibility in design.

8. Polymorphism :
→ Utilize inheritance and interfaces to enable multiple implementations of behaviors.

Benefits of GRASP :
* clarity and Understandability.
* Flexibility and Adaptability.
* promotion of Best practices.
* Maintainability and Scalability.
* Enhanced Reusability.

Challenges of GRASP :-
* Complexity
* Subjectivity.
* Trade-offs.
* Context Sensitivity.
* Maintanance Overhead.

Architectural Analysis :

Def :- Architectural Analysis is the process of evaluating and defining the structure of a software system to ensure it meets functional and non-functional requirements.

Key aspects :

1. Identifying Requirements –
→ understanding system needs, including performance, security, and scalability.

2. Defining Architecture Styles –
→ choosing suitable architecture patterns
Ex :- Layered, microservices, client-server.

3. Evaluating Design Decisions –
→ Assessing trade-offs b/w different architectural approaches.

4. Ensuring Feasibility –
→ Verifying that the chosen architecture supports future growth and maintainability.

Importance:

* Enhances system efficiency and reliability.
* Improves maintainability and scalability.
* Helps in risk identification and mitigation.

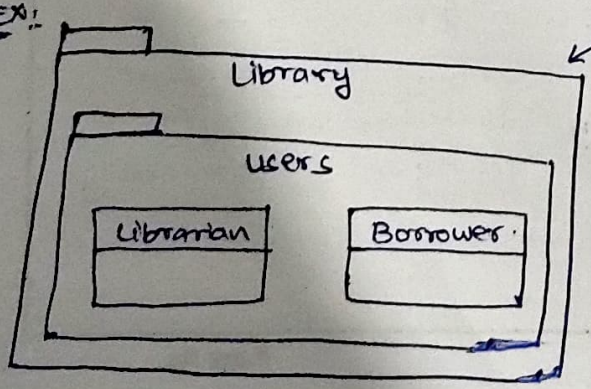
UML Package Design:

* In UML, the partitions/sub-systems is called packages.
* A package is a grouping of model elements, and, such, it is a UML construct used as also in other UML diagrams.
* Package themselves may be nested within other packages that basically the UML version of a directory, a place to put things.

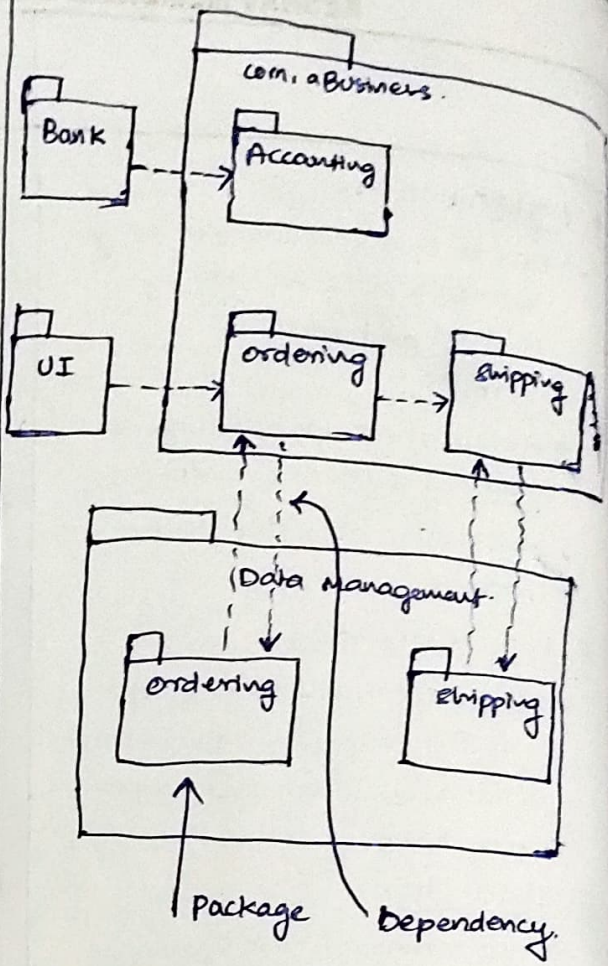
Package diagram use cases:

* provide a way to group related UML elements and to scope their names.
* provide a way to visualize dependencies b/w parts of system.
* Vulnerable to changes (in other packages).
* provide support for analysis.
* Determine compilation order.

Ex:



Ex:

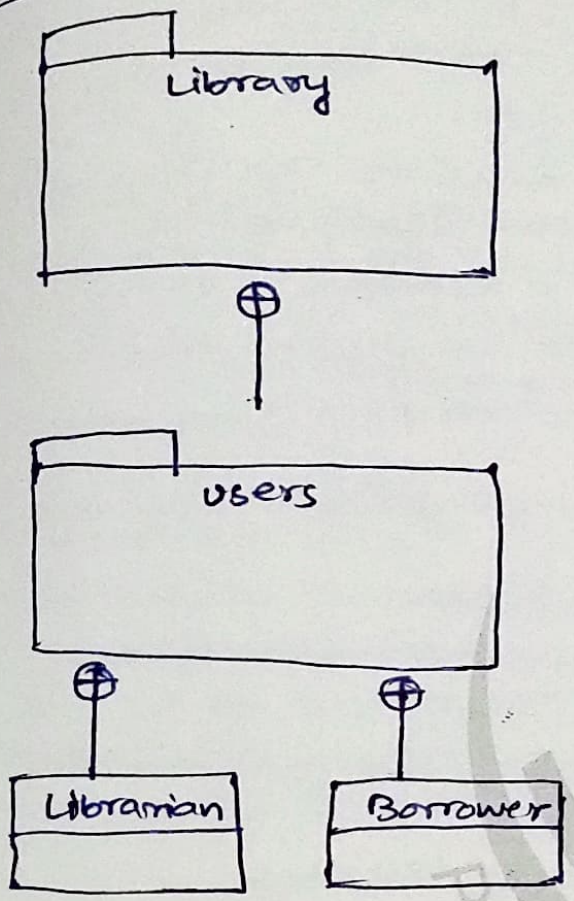


Package Diagram Namespace:

* A UML package establishes a namespace for specifying the context of a UML.
* A package defines what is known as an encapsulated namespace.
* When an element in one space needs to refer to an element in a different namespace, it has to specify both the name of the element it wants and the qualified name/pathname of the element.

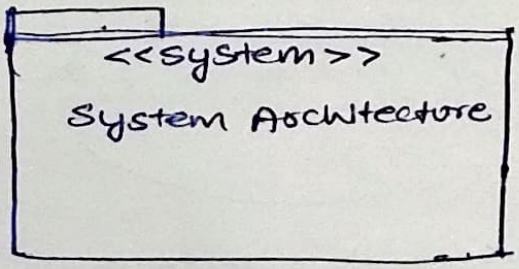
Ex: packageName::className.

Alternative Representation of Package:



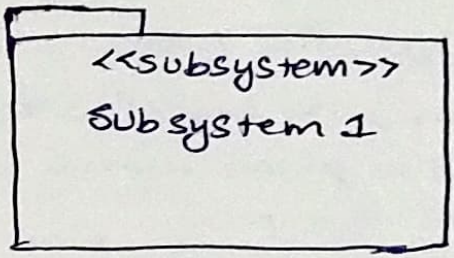
System and Subsystems:

* A System is represented as a package with the stereotype of <<system>>.
* The system represents all the model elements that pertain to the particular project.
* Break a system to make them smaller and more workable.



Subsystem: A subsystem is a grouping of model elements that are part of the overall system.

⇒ Also like systems, are stereotyped packages.



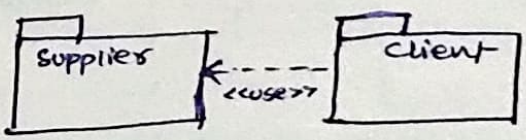
Stereotype:

Stereotypes are a high-level classification of an object that gives you some indication of the kind of object is.

* Classes can be grouped under stereotypes, written between guillemots (<<>>), over the class name.
* A stereotype enables you to extend the UML to fit your modeling needs more specifically.
* A stereotype is a UML modeling element that extends the existing elements.

Package Diagram - Dependency Notation:

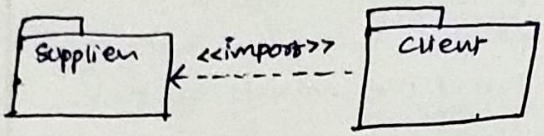
①



- The client package uses a public element in the supplier package.
- The client depends on the supplier.

"<<use>>"

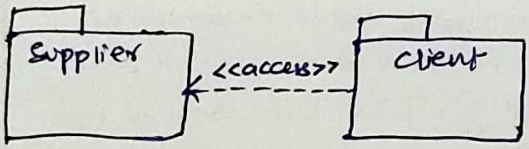
(2)



- Elements in the client can access all public elements in the supplier using unqualified names.
- public elements of supplier namespace are added as private elements to the client namespace.

"<<import>>"

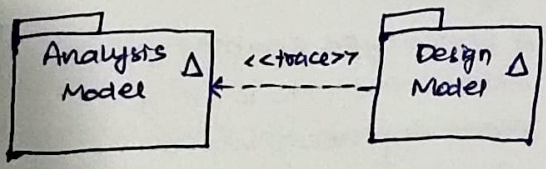
(3)



- public elements of the supplier name space are added as private elements to the client namespace.

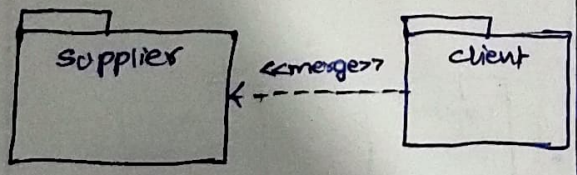
"<<access>>"

(4)



- <<trace>> usually represents a historical development of one element into another more developed version.
- It is usually a relationship b/w models rather than elements.

(5)



- public elements of the supplier package are merged with elements of the client package.

OOAD-5 :- Test Driven Development and Agile concepts, Documenting Architecture Case studies.

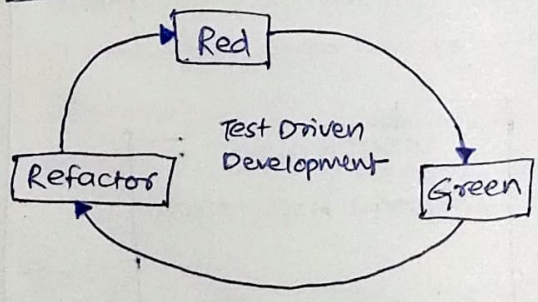
Test Driven Development:

- It is the process in which test cases are written before the code that validates those cases.
- It depends on the repetition of a very short development cycle.
- TDD is a technique in which automated unit test are used to drive the design and free decoupling of dependencies.

The following seqcence of steps followed:

1. Add a testcase
   - ↳ that describe the function completely.
   - ↳ The developer must understand the features and requirements.
2. Run all the testcases
   - ↳ make sure that new test case fails.
3. write the code that passes the test case.
4. Run the test cases.
5. Refactor code - to remove duplication of code.
6. Repeat the steps again and again.

Motto:-



1. Red - create a test case and make it fail.
2. Green - Make the test case pass by any means.

Date

3. Refactor - change the code to remove duplicate/redundancy.

Benefits :

* Unit test provides constant feedback about the functions.
* Quality of design increases which further helps in proper maintenance.
* TDD act as a saftey net against the bugs.
* TDD ensures that application actually meets requirements defined for it.
* TDD have very short development life cycle.
* Efficient approach that drives positive results.

⇒ It is an iterative approach combining programming, unit test creation, and refactoring.

⇒ TDD approach originates from the Agile manifesto principles and Extreme programming.

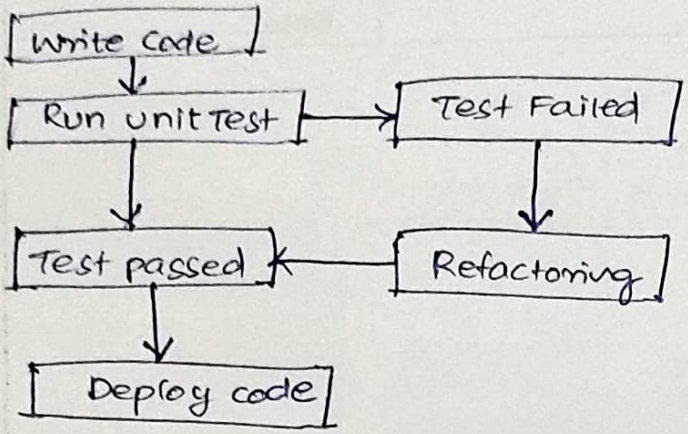
Ex: 1. Calculator Function.
2. User Authentication.
3. E-commerce website.

Feature	TDD	Traditional testing.
Approach	Tests are written before code development	Testing occurs after code is written.
Testing Scope	Focuses on unit testing small code segments	Covers system, integration, and functional testing.
Process	Iterative write test → Develop code → Refine.	code is tested once and refined based on results
Debugging	Detects errors early, simplifying debugging	Errors are found, making debugging harder
Documentation	Focuses on test cases and results	Includes detailed reports on testing process and environment.

Three phases of TDD :-

1. Create precise tests
   ↳ to verify the functionality of specific features.
2. Correcting the code ; once a test fails, developers must make the minimal changes required to update the code to run successfully when re-executed.
3. Refactor the code : After test
   ↳ check for redundancy and any possible code optimizations to enhance overall performance.

High level TDD Approach toward development:



How TDD fits in Agile Development

1. Enables Feedback-Driven Development.
2. Handles Changing Requirements.
3. Prevents Critical Bottlenecks.
4. Ensures Continous Evolution.
5. Enhances Collaboration.
6. Reduces Testing overhead.

Best practices for TDD:-

1. Start with a clear understanding of requirements.
2. Write atomic tests.
3. Write the simplest test case first.
4. Write tests for edge cases.
5. Refactor regularly.
6. Maintain a fast feedback loop.
7. Automate your tests.
8. Follow the Red-Green-Refactor cycle.
9. Continoosly run tests.
10. Test failures should guide development.

Document Architecture

* Document Architecture refers to the structured way of organizing, storing and managing documents in a system.
* It defines how documents are created, formatted, accessed and shared to ensure consistency, security, and ease of use.
* It helps teams understand system design, communicate technical decisions, and maintain documentation for future reference.

Importance / Reasons to Document Architecture:

1. Whiteboard Designs Are not persistant.
2. Scaling teams and Knowledge Shared.
3. Preserving Design Rationale.
4. Addressing Staff Turnover.
5. Catering to Different Stakeholders.
6. Visualizing and Planning.
7. Handling complexity in Modern Systems.

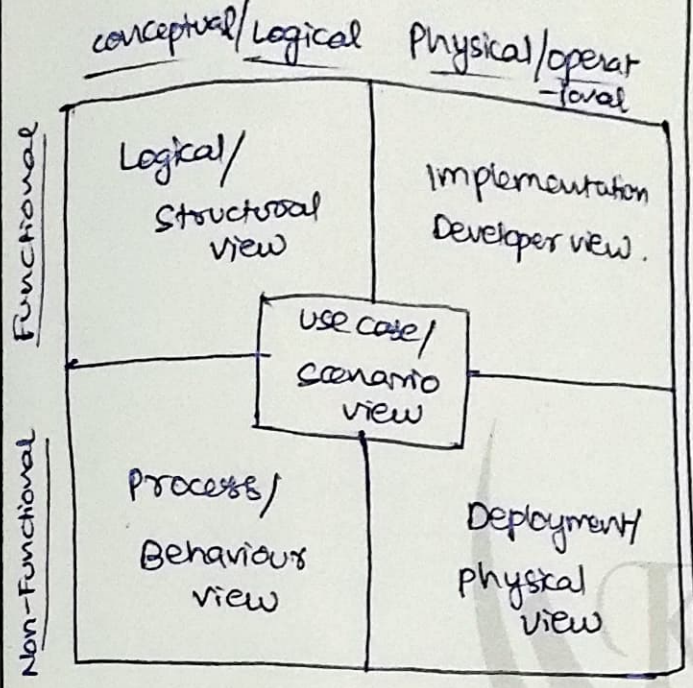
Architecture document include:

* Scope and Summary.
* Identify Stakeholders.
* Architecture views.
* Document Interfaces
* Key Decisions.
* Non-Functional Requirements (NFRs)
* Standard Templates
* Review Process.

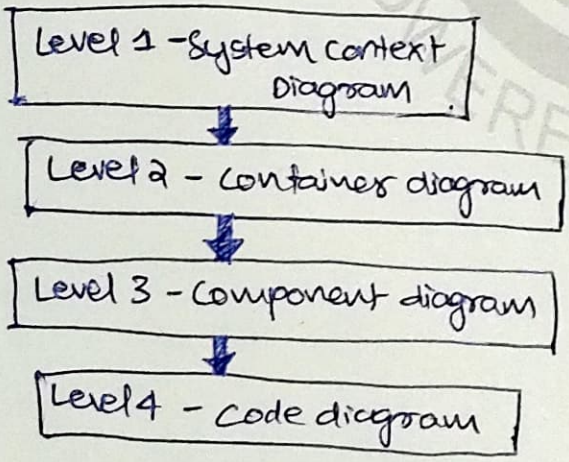
EXPT. No.
Date

Common architecture view models:

1) 4+1 Architectural view model



2) C4-Model



3) Sivens Four view Model.

Architecture Decision Records (ADR)

* Logs significant architectural decisions with their context and impact.
* Helps teams understand why certain decisions were made.

Case Studies

* It is an in-depth analysis of a real world scenario, problem or project.
* It helps understand concepts by applying them to practical situations.
* Case studies demonstrate how object-oriented principles and UML diagrams are used to design software solutions.

Why are Case studies Important?

* Practical Learning.
* problem solving.
* Improved understanding.
* Decision Making.
* Standardization.

How are Case studies Conducted?

1. Identify the problem.
2. Analyze Requirements.
3. Design the solution.
4. Implement the System.
5. Evaluate Results.

EX:- Banking System.

1. Identify users (Customers, Bank Employees).
2. Define actions (Deposit, withdraw, Transfer).
3. Create UML diagrams (Class, Sequence, state).