Objected Oriented Analysis and Design with Java - III

UNIT III - Development process, System Design and Frameworks

▼ Stages of Object oriented methodology

1. System conception - Software development begins with begins with business analysis and formulation requirements.
2. Analysis - Precise abstraction of what the desired system must do. Workflow of the application
3. System design - High level architecture
4. Class design - Add details to the analysis model with system strategy. Focus is on data structures and algorithms
5. Implementation - Translate classes and relationships into particular programming languages, databases and hardware

▼ System Design - Reuse Plans

Reusable things include models, libraries, frameworks and patterns

▼ Libraries

A collection of classes that are useful in many contexts. Classes must be carefully organized. Classes must be coherent, complete, consistent, efficient, extensible and generic.

Performs a set of specific and well defined operations.

Ex: Network protocols, compression, image manipulation, regex, etc

▼ Patterns

A pattern is a best practice, a time tested solution to a recurring problem that's proved to work.

Ex: Software architecture patterns like Layered patter, client server pattern.

Design patterns (GoF) - Creational, Structural and Behavioural

▼ Frameworks

Frameworks provide a skeletal structure or read made architecture for our application.

The skeletal structure provided must be elaborated to build the complete application. Elaboration consists of specialising the abstract classes with behaviour specific to an individual application.

A set of cooperating classes that makes up a reusable design for a specific class of software.

ex: Collections, Swing, AWT, Spring, Hibernate, Grails

▼ Architectural Pattern - MVC

An architectural design pattern for computer software to distinguish between the data model, processing control and the user interface.

Consists of the data model, presentation information and control information.

- Model - Represents data and rules that govern access to updates of data. Software approximation of a real world process.
- View - Render the contents of a model. Presentation layer of the application, visualizes data that the model contains.
- Controller - Translates user's interactions with the view into actions that the model will perform.

→ Loose coupling between components within the system.

▼ GRASP

General Responsibility Assignment Software Patterns

→ Translation of responsibilities into classes and methods is influenced by the granularity of responsibility.

→ Types of responsibility - Knowing and Doing

GRASP Patterns -

- ▼ Creator

  Decide who can be the creator based on the object's association and their interaction.

  Objects interact via messages

- ▼ Information Expert

  Assign a responsibility to the class that has the information needed to respond to it.

- ▼ Low coupling

  Assign responsibility such that dependencies between components are minimized.

- ▼ Controller

  Deals with how to delegate the request from the UI layer objects to domain layer objects. Minimizes dependencies between GUI components.

  Assign responsibility for receiving and handling a system event to an object that represents the overall system (Facade controller) or an object that represents a use case scenario within which the system event occurs (Use case controller)

- ▼ High cohesion

  Cohesion - A measure of how strongly related and focused the responsibilities of an element are.

  Assign responsibility such that the cohesion remains high.

- ▼ Polymorphism

  Types of polymorphism -

  1. Adhoc - Overloading

2. Parametric - Early binding
3. Inclusion - Subtyping
4. Coercion - Casting

▼ Indirection

Assign responsibility to intermediate classes to providing linking between objects and not linking directly.

▼ Controlled/Protected variations

Identify points of predicted variation or instability, assign responsibilities to create a stable interface around them.

- Variation point - branching point on existing system or in requirements
- Evolution point - Future branching point, but not according to existing requirements.

▼ Pure fabrication

Assign a highly cohesive set of responsibilities to an artificial class that does not represent a domain concept.

▼ SOLID principles

Important aspects of bad design -

1. Rigidity
2. Fragility
3. Immobility
4. Viscosity

▼ Single Responsibility Principle (SRP)

Every module or class should have responsibility over a single part of the functionality provided by the software.

▼ Open-Closed Principle (OCP)

Classes, Modules, Micro-services and other code units should be open for extension but closed for modification.

- Liskov Substitution Principle (LSP)

  A child class should never change the characteristic of a parent class; Derived classes should never do less than their base class.

  An object of the superclass should be replaceable by objects of its subclasses without causing issues in the application.

- Interface Segregation Principle (ISP)

  Make fine grained interfaces that are client specific. Clients should not be forced to implement interfaces they don’t use.

- Dependency Inversion Principle (DIP)

  High level modules should not depend on low level modules, both should depend upon abstractions.

  Abstractions should not depend on details, details should depend on abstractions.

  Specific combination of LSP and OCP.