API Design

Why design?

- not only consistent about the software changes, requirment will changes, technology will change so software will be changing

- So developers have to accomdate the future changes in the present software

Based on 3 things

- DRY - Do Not Repeat Yourself - duplicate code will affect the software

- Divide and Conquer - instead of writing entire appl in single time, we can divide into small small modules

- Software changes -

- So we should design the appl to accomdate new changes without affecting the existing working functionality

SOLID Principles - Robert C.Martin

1. S - Single Responsibility principle

2. O - Open closed Principle

3. L - Lisknow Substitution principle

4. I - Interface Segregation Principle

5. D - Dependency Inversion Priniciple

If we didnt follow SOLID Priniciple

1. End up with tight or strong coupling of the code with many other modules

2. Tight coupling causes time to implement new requirement or features since one module depend on another

3. End up with code which is not testable

4. End up with duplication code

5. End up creating new bugs by fixing another bug

6. End up with unknown issues in the application development lifecycle

Advantage

1. Achieve reduction in complexity of code

2. Reduce error and implement reusability

3. Better testability

4. Reduce tightly coupled

1. Single Responsibility principle

- A class should have only one reason to change

Member class - register, can lend the book, can cancel the membership, new offer can be notified to member

Booking class - makeBooking()

1. Each class and module should focus on a single task at a time

2. Everything in the class should be related to that single purpose

3. There can be many members in the class as long as they related to the single responsiblity

4. With SRP, classes become smaller and cleaner

5. Code is less fragile

2. Open closed Principle

- classes,modules, functions etc are open for extension but closed for modification

- any new functionality should be implemented by adding new classes, attributes, methods etc instead of changing the current ones

3. Liskov Subsitution principle - Barbara Liskov in 1987

- Substitutability is a priniciple in OOP and it states that if S is a subtype of T, then objects of type T may be replaced with object of type S.

- One parent with many child and every children should subsitute the parent for providing the functionality provided by parent

- It is an extension of OCP

- No new exception can be thrown by the subtype unless they are part of the existing exception hierarchy

- We should ensure that the client should not know which specific subtype they are calling nor should they need to know that

- New derived classes just extend without replacing the functionality of old class

4. Interface Segregation Principle

- states that "no client shoule be forced to depend on methods it does not use"

- which means instead of one fat interface we have many small interface

5. Dependency Inversion Priniciple

- Higher level modules should not depend on low level modules, both should depend upon abstraction

Controller class -IRepository interface - Repository class

Design Patterns

- well proved solution for solving specific problem - 3 types

1. Creational Design pattern

- concerned with ways of creating objects

a. Singleton design pattern

- consist of class for which their exist only one instance, we cant create more that one instance

- That class provides global point of access to that unique instance

- Thread pooling, Loggers, device manager software like class which access to certain device like printers etc

private static Logger log=Logger.getLogger(UserController.class);

How to implement?

- Make constructor as private, if the constructor is private no other class can directly instantiate the class

- In order to make other class to access this single instance, we create static method which return the same singleton class

b. Factory design pattern

- we create object without exposing the creation logic to client and referred newly created object using common interface

- If we have super class and sub class based on data provided we have to return the object of one of subclasses

c. Abstract Factory design pattern

- It is a super factory which creates other factories. It is called factory of factory

d. Prototype design pattern

- used for creating new objects by cloning other objects and by this we can improve performance

- when creation of object is costly or complex

e. Object pool design pattern

- used to reuse object and share the object that are expensive to create, improves the performance

- Connection object

f. Builder design pattern

- Build complex object using simple object with step by step approach

- The process of constructing an complex object should be generic approach

Structural design pattern

- how classes and objects are composed to form large structure

- simplifies the structure by identfying the relationship

1. Composite pattern

- allows you to have tree structure and ask each node to perform some task

Employee

- SE

- PA

- TL

- TestingLead

2. Adapter pattern

- works as a bridge between two incompatiable interface

3. Proxy design pattern

- It provides a placeholder for another object to control access on it

- RMI

4. Flyweight design pattern

-used when there is a need for large number of objects of almost similar nature. The large number of object consume large amount of memory and this pattern provides a solution for reducing the load on memory by sharing objects

5. Decorator pattern

- Allows to add new functionality to an existing object without altering its structure

6. Bridge pattern

- It separates abstraction and implementation which developed independently,so that any change in one will not affect the development in another

Behavioral design pattern

- concerned with interaction and responsibility of the object

1. Memento pattern

- we can restore the object on its previous state

- ctrl z

2. Template design pattern

- Defines a sequence of steps or an algorithm. The subclasses are allowed to override the steps but not allowed to change the sequence

3. Mediator design pattern

- chatroom

- used to reduce communication complexity between multiple object

4. Startegy design pattern

- Collectios.sort(Comparator)

- Used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime

5. State design pattern

- ATM mc, Vending mc

- allows an object to alter its behaviour when its internal state changes

6. Iterator pattern - forEach()

7. Chain of Responsibilty - try catch, Filter