Lecture # 01.

Date:

Software Development

* Why do we develop software ?
  → To solve frequently occurring problems.

* Technical Aspects
  → Program Constructs
    ① Sequence (order of instructions)
    ② Iteration (repeated execution)
    ③ Selection (path of a program)
  → Program Organization.

* Concept Triad
  ① The term Concept
    → An idea/notion that we apply to things/objects

  ② The intension
    → The internal content of a concept.
    → The set of attributes belonging to all & only those things to which the given term is correctly applied.
    → The test that determines whether or not the concept applies to an object

  ③ The extension.
    → State of being extended.
    → Enlarging scope of concept.
    → The set of all objects to which the concept applies.

* Type
  - Concept.
  - Kind of object.
  - Synonym of concept.

Lecture # 02

(2)

Object - Oriented Concepts

Date: _____

① Abstraction.

→ Purpose is to handle complexity.
→ Hiding unnecessary details from the user.
→ To implement more complex logic on the top of the provided abstraction.

What user knows ?

→ Methods of the objects.
→ Input parameters needed to trigger specific operation.

Java

→ Abstract class / Abstract method
→ Interfaces.

Cannot create object of abstract classes.

② Class.

→ Blueprint for creating objects.
→ User-defined data type.

③ Object

→ Instance of a class

④ Encapsulation

→ Binding the data and the functions.

Date:

⑤ Information hiding.
→ Protecting data/information.
→ Protecting from direct modification

⑥ Inheritance
→ Deriving a class from another class.
→ Hierarchy of classes that share a set of attributes and methods.

⑦ Interface
→ Defines behaviour or method that can be implemented by any class.
→ Allows to specify set of functions signatures and hide the implementation of those functions in an "implementing" class.

⑧ Messaging
→ Visualizing how object-oriented program actually executes and relationship between the abstractions in an OOP.
→ Messaging an object may cause it to change state.

⑨ Polymorphism.
→ Multiple forms

add()
sub()
mull()

calc(a, b, add) → ptr.

⑩ Delegate
→ A pointer to a function.
→ Invoking a piece of code that is unknown until runtime.

→ STC main alk generic calculation ka function bnaya tha.

Date:

(11) Relationships.

→ Association : User defined relationship →

① One-way eg : Student — course

i.e Student class mai course class ka obj hoga.

→ Classification : Applying concept to an object.

→ Generalization / Specialization

→ Aggregation / Composition.

Aggregation : Weak dependency. Has a relationship. ◇

Composition : Strong dependency. Class owns another class ◆

→ Inheritance

→ Realization

Lecture # 03.

Date:

⇒ Specialization

→ Top-down approach ⇒ Already cooked ingredients.

→ Bottom-up approach ⇒ Go & figure out details.

⇒ Generalization.

eg : Library management system. // Top-down approach

→ Identify objects | class

→ Identify attributes

Book ⇒ Copies → tangible

↙ ↘

intangible in general because it doesn't have id.

tangible in library because id is assigned to each book.

→ Identify activity

Issuance → Book copy
→ library member
→ library staff

return book

This is how we design, and perform data modeling.

eg : Leave management system // Bottom-up approach.

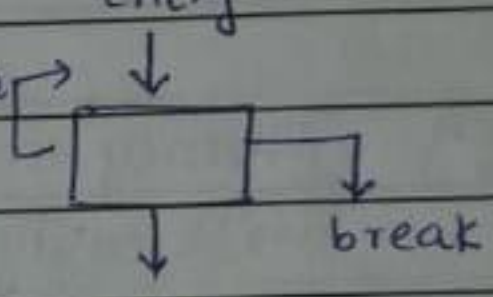
Paradigm → A way of work. How we want to design.
Depends on approach.

Date:

→ Program Structure.

* Don'ts for structured programming

Goto, break, continue, return.



* Don'ts for better programming

Getters & setters ⇒ Make purposeful getter/setter.
Mutation ⇒ Only mutate when there is a purpose

Purpose → Validation & computation.

→ Objects & classes

① Object recognition
- must
- Make relevant objects

② Class creation

③ Top-down approach

④ Bottom-up approach.

⑤ Abstraction.

Date:

(6) Dependency.

It depends on
- Which class is creating an object (Scope)
- When the object is being created (Creation)
- Where it is being created (Location)

Scope → Object defined at class level
→ object defined at method level.

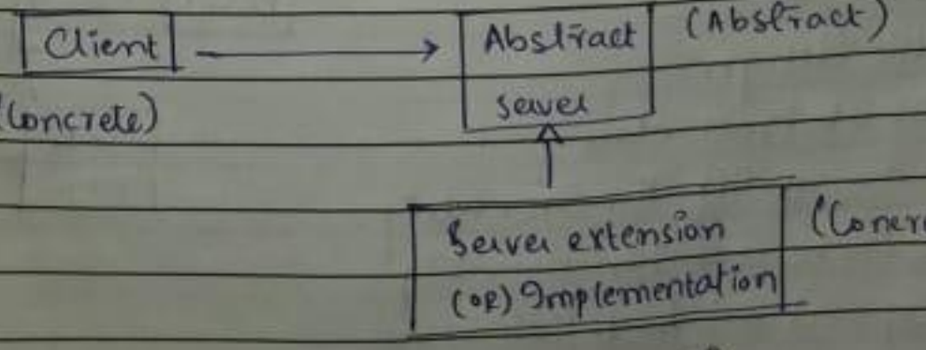
Creation → Class is creating the object where it is defined
→ Class is creating the object and the reference is provided to the class where the object is defined.

Location of creation → Constructor
→ Within the method.

Aim is to minimize dependency.

// Must see examples from slides.

→ Software Design.



Lecture # 04 (8)

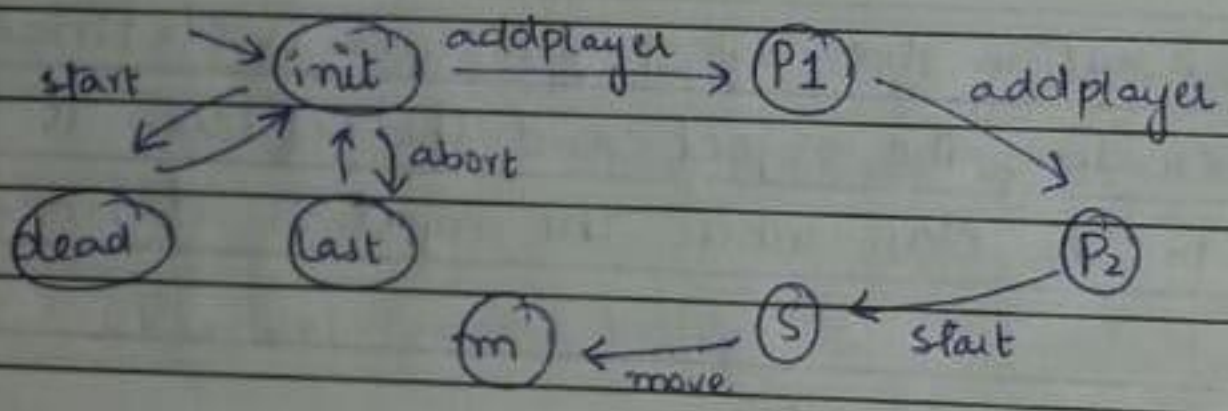
Date:

Software design :

Eg Tic-tac-toe.

Abstract class → Bottom-up approach usually.

Player → Register
→ Unregister
→ Check status.

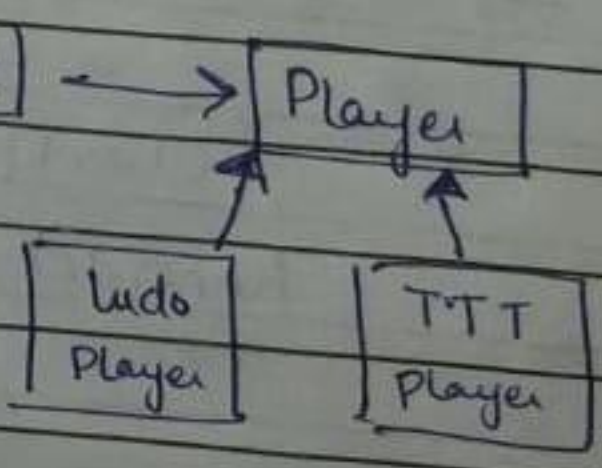


Abstract class
→ Main functionality
→ All exposed functionalities.

Q Are functions reusable?

A: No, functions are not reusable. They are recallable.

→ Reusable design
- An abstract class
- Can modify functionality
- Can borrow functionality
- Exposed methods/behaviour.



Date:

→ Coupling :
The degree of interaction between two modules

We need to reduce coupling.

Six kinds of coupling;

① Content Coupling
Highest coupling
Shouldn't be allowed under any circumstances.
Also called "Pathological Coupling"
Accessing "internal" / private info of the other module
Accessing something which is inside other module's scope.
eg: Module A changes Module B's internal data.
Module A falls into Module B (through GOTO)

② Common Coupling
High coupling.
Undesirable, but unavoidable
"Information hiding" & well-defined interfaces are used to limit coupling effect.
If modules refer to same global data area, then they exhibit Common Coupling.
eg: Use of global variables.

Date:

③ External Coupling

High Coupling

Undesirable, but unavoidable.

Well-defined interfaces are used to limit its effect.

Share direct access to same I/O devices

Tied to same part of environment external to software.

④ Control Coupling

Moderate coupling

Perfectly acceptable.

If one module passes to the other module a piece of information that is intended to control the internal logic of the other, then they exhibit Control Coupling.

eg: Conditions controlled by function parameter

Switch case statements. if-then, while.

⑤ Stamp Coupling

Low coupling / Normal Coupling

Most desirable coupling

Parameters with meaningful structure are passed such as pointers, references, array, tree (not actual data).

If one module directly passes "Composite" piece of data to the other module, then they exhibit Stamp Coupling.

⑥ Data Coupling.

Low coupling / Normal coupling

Most safest and desirable

Minimum dependency.

Date:

One module directly calls other module & communicate using parameters.

Ideal Situation → lowest level of Coupling
Modules do not have direct communication.
Modules are also not "tied-together" by shared access to same global data area or external device.
No coupling at all.

→ Cohesion:
'Functional independence' of a module.
It refers to strength of a method as it relates to the routine within it.

Categories of Cohesion:

⇒ Low Cohesion: Highly undesirable.

① Coincidental Cohesion
All unrelated routines working together without relation.
A coincidental cohesive module is one whose elements contribute to activities with no meaningful relationship to one another.

② Logical Cohesion.
It is a module whose elements contribute to activities of same general category in which activities to be executed are selected from outside the module.

Date:

③ Temporal Cohesion

A method is said to have temporal cohesion when all routines within the method need to occur at the same time, but not necessarily in order.

A module whose elements are involved in activities that are related to time.

eg: "Do all startup activities"
"Do all shutdown activities"

⇒ Moderate Cohesion : Acceptable

④ Procedural Cohesion

A method is said to have procedural cohesion when all all routines within the method needs to occur in specified order.

Routines don't share data. data

Only control dependency, not ↑ dependency.

A module whose elements are involved in different and possibly unrelated activities in which control flows from each activity to next

⑤ Communicational Cohesion

A method is said to have communicational cohesion when it does more than one unrelated thing on the same data.

A module whose elements contribute to activities that use the same input/output data.

eg: Seach for book.
Find title of book.

Date:

⑥ Sequential Cohesion

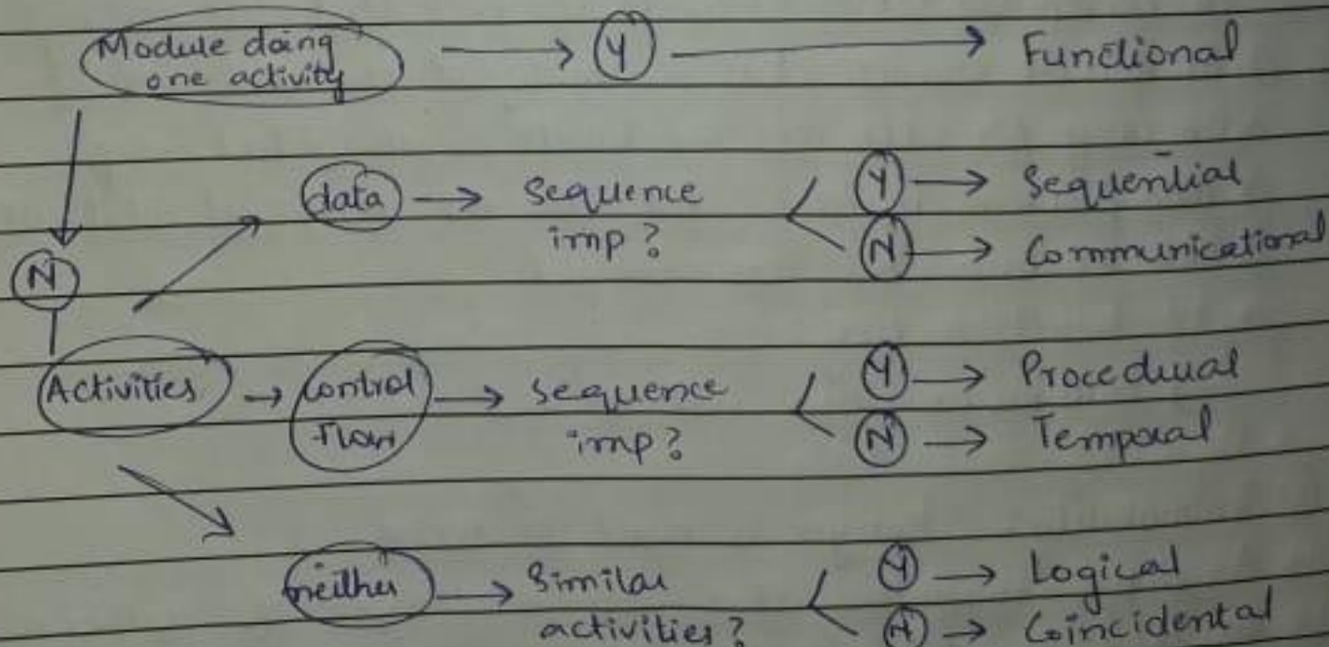
Calling functions in certain sequence & each function is dependent on the data of previous function.
A module whose elements are involved in activities such that output data of one activity serves as input data of the next activity.

⇒ High Cohesion: Desirable.

⑦ Functional Cohesion.

A method with only one responsibility
A method has strong functional cohesion when it does just one thing.

→ Decision Tree For Module Cohesion



Lecture # 05 (14)

Date:

→ Symptoms of Poor design :

① Rigidity : Design is hard to change
→ Unit test ⇒ Test until unit test is passed.
→ The system is hard to change because every changes forces many other changes to other part of the system.
→ Single change causes cascade of subsequent changes in dependent modules
→ Hard coding something & then using it in implementation, so rigidity occurs.
→ Dependency on concrete class creates rigidity.

② Fragility : Design is easy to break.
→ Changes cause the system to break in places that have no conceptual relationship to the part that was change.
→ As soon as concrete class is modified the code breaks.
→ On every fix, the software breaks in unexpected ways.
→ New problems in area that have no conceptual relationship with the area that was changed
→ No bound checking
eg Null ptr exception, not treated.

③ Immobility : Design is hard to reuse.
→ A generic module, that uses any type of database.
→ Service layer, database module should have very less dependency, so that it can be shifted to another project easily.

Date:

→ Configuration of the system.
Create txt/xml file Java properties file key value pairs.
If change then whole program configuration will change.

→ Too much coupling, less immobility.
→ It is hard to disentangle the system into components that can be reused in other systems.
→ It contains part that could be useful in other systems, but the effort & risk involved in seperating those parts from original system is too much.
→ The useful modules have too many dependencies.
→ The cost of rewriting is less compared to the risk to seperate those parts.

④ Viscosity : Hard to ~~write~~ right thing.
→ Two forms of viscosity
Viscosity of design.
Viscosity of environment.
→ When design preserving methods are harder to employ than the hacks, then the viscosity of the design is high.
→ Viscosity of the design comes when the ~~deploy~~ development environment is slow & inefficient.
→ Some procedures/classes are not easy to implement so rather than changing source we look for workarounds

Date:

(5) Needless Complexity: Overdesign
→ Making classes that may not be used, but have been implemented just for the sake that extension exists
→ The design contains infrastructure that adds no direct benefit
→ Adding transaction processing (to prevent data from loss) and something complex in a file system to maintain data flow, will add needless complexity.

(6) Needless Repetition: Mouse abuse
→ The design contains repeating structures that could be unified under a single abstraction.
→ Developer's abuse of cut and paste.

(7) Opacity: Disorganized expression.
→ It is hard to read and understand
→ Poor naming conventions
→ Inconsistent style of code.
→ The code does not express its intent well.

→ Signs of Good design:

① Adaptability → The design is easy to change
② Robustness → The design is hard to break.
③ Reusability → The design can be reused

Date:

④ Fluidity → It is easy to do right thing
⑤ Simplicity → Simplest design that will work
⑥ Terseness → No unneeded duplication of code
⑦ Perspecuity → Organized & clear

→ Design Principles – SOLID

① Single Responsibility Principle. (SRP)

"A class should have only one reason to change"

- If a class has more than one responsibility, then there will be more than one reason for it to change. This coupling lead to fragile design that break in unexpected ways when changed.

- An axis of change is only an axis of change if the changes actually occur.
Otherwise increases Needless Complexity.

Lecture # 06

Date:

→ Primary mechanism: Abstraction & polymorphism.

② Open-Closed Principle (OCP)

"Software entities (classes, modules, functions, etc) should be open for extension, but closed for modification"

- Design modules that never change. When requirements change, you extend the behavior of such module by adding new code, not by changing **old** code that already works.

- Extending behavior of the module or creating new module for additional work follows OCP.

- Changing the source code of existing class violates OCP

③ Liskov Substitution Principle (LSP)

"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."

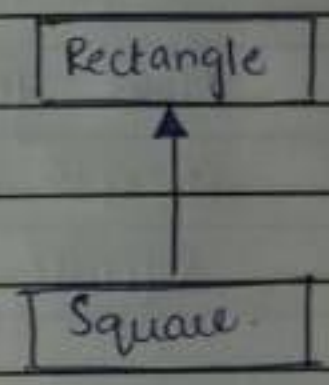
LSP violation example:

```
class Rectangle {
    public:
    void setWidth (double w) { itsWidth = w; }
```

Date:

```
void SetHeight (double h) { itsHeight = h; }
double getHeight ( ) const { return its Height; }
double getWidth ( ) const { return its Width; }.

private :
    double itsWidth;
    double its Height;
};
```




```
class Square : public Rectangle {

public :
    void SetWidth (double w) {
            Rectangle :: SetWidth (w);
            Rectangle :: SetHeight (w);
        }

    void SetHeight (double h) {
            Rectangle :: SetHeight (h);
            Rectangle :: SetWidth (h);

};
```

✓
```
Square s;
s.setWidth (1);      // Sets width & height to 1
s.setHeight (2);     // sets height & width to 2

void f (Rectangle & (r)) { r.Set width (32);
```

Reference to square object

```
// calls Rectangle :: set Width (
```

Date:

This violation can be avoided by declaring SetWidth and SetHeight Virtual in Rectangle class

Now you can pass Square into a function that accepts pointer or reference to a Rectangle, and Square will remain consistent.

```
void g (Rectangle & r)
{
    r.SetWidth (5);
    r.SetHeight (4);
    assert (r.GetWidth() * r.GetHeight() == 20);
}
```

Note:

Do not extend from a Concrete class

Because

① Base can be changed any time.

② Already implemented method is overwritten.

Extension must be from an Abstract class.

Date:

Q Extension v/s delegation? Benefits & disadvantages

④ Interface Segregation Principle (ISP)

- ISP acknowledges that there are interfaces objects that require non-cohesive interfaces.

- Clients should know about the abstract base classes that have cohesive interfaces

"Clients should not be forced to depend upon interfaces that they do not use."

- By making use of ADAPTER pattern, either through delegation (object form) or multiple interfaces (class form), fat interfaces can be segregated into abstract classes that break the unwanted coupling between clients.

Q What would be your interface design?

A. One or two interfaces are enough to show behavior. Interfaces should be complete, no extra methods should be there.