Software Design Objectives

MCQS



Digital World VU
By Sabahat Saad

Software Design Objectives



MCQS

Serves as a technical plan for implementation

Software Design Objectives



Digital World VU
By Sabahat Saad

During the design phase, software engineers apply their knowledge of:

1. **the problem domain**
2. **implementation technologies**

New MCQs.

Q: Objectives of Software design

Software Design Objectives



Digital World VU
By Sabahat Saad

1. **The system or program should meet customer's needs**

Software Design Objectives



2 **Should be easy to implement**

Software Design Objectives



(3) Should be "efficient"

Software Design Objectives



4

Should be "easily extendible" to meet new needs

MCQs.

Poorly designed programs are difficult to understand and modify

The size does matter!

The larger the program, the more pronounced are the consequences of poor design

Q: How many types of complexity?

Two Types of Complexity in Software

(1)

✓ **Essential complexities** – complexities that are inherent in the problem.

Short + Long + MCQs

Design ← MCQ's

✓ is

An Antidote to Complexity

primary tool for managing essential and accidental complexities in software

Two Types of Complexity in Software

② **Accidental/incidental complexities** –

complexities that are artifacts of the solution.

نمونے

MCQs
+ Subjective

Two Types of Complexity in Software

The total amount of complexity in a software solution is:

MCQs

✓ **Essential Complexities**

+

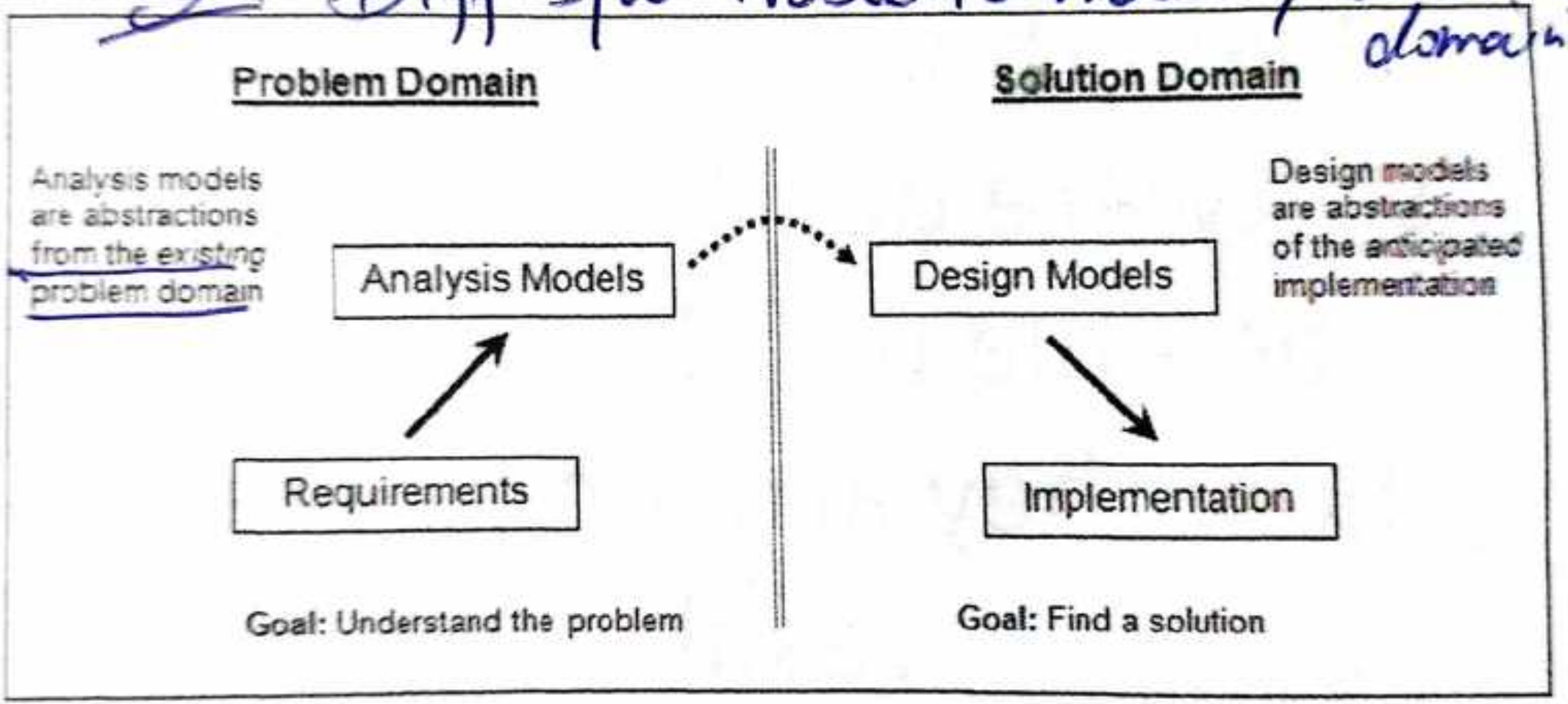
Accidental complexities

Why Design is Hard

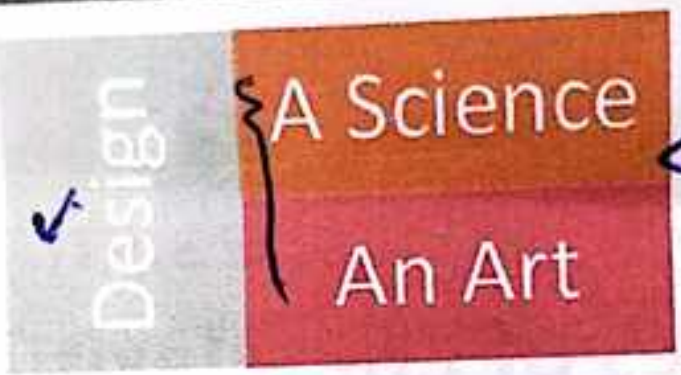
Subj + MCQs

Design is difficult because design is an abstraction of the solution which has yet to be created

Q: Diff b/w Problem domain & Solution domain?



Difficulty with design



MCQs

48

Q: Define Wicked Problem?

A wicked problem

A problem that can only be clearly defined by solving it

MCQS

Design is a wicked problem!

51

Compositi

•A form
wit

Q: How to make the design process more systematic/predictable

Imp Slide.

The design process can be made more systematic and predictable through the application of methods, techniques and patterns, all applied according to principles and heuristics.

55

Q: Characteristics of good design

(i) Good design doesn't reduce the total amount of essential complexity in a solution but it will reduce the amount of complexity that a programmer has to deal with at any one time

58

(2)

END!

56

Role of ... Organization, Information Hiding, and Abstraction in dealing with software complexity

57

(2) A good design will manage essential complexities inherent in the problem without adding to accidental complexities consequential to the solution.

59

MCQs **Modularity**

subdivide the solution into smaller easier to manage components

✓(divide and conquer)

60

MCQs **Information Hiding**

hide details and complexity behind simple interfaces

62

Abstraction MCQs

use abstractions to suppress details in places where they are unnecessary

Software Design and Architecture

Topic#011
Characteristics of Software Design

A deterministic process is one that produces the same output given the same inputs

MCQs

Design is non-deterministic

No two designers or design processes are likely to produce the same output.

Design is **Heuristic**

because design is non-deterministic, design techniques tend to rely on heuristics and rules-of-thumb rather than repeatable processes.

Subjective

Emergent

the final design evolves from experience and feedback

Design is an iterative and incremental process where a complex system arises out of relatively simple interactions

Q: Benefits of good design.

Good design reduces software complexity - making the software easier to understand and modify

Rapid development

Easier Maintenance

Good design makes it easier to reuse code.

4 • It improves software quality.

5 • Good design exposes defects and makes it easier to test the software.

MCQs

6 • Complexity is the root cause of other problems such as security. A program that is difficult to understand is more likely to be vulnerable to exploits than one that is simpler.

New MCQs

The final design evolves from experience and feedback

New MCQs

Design is an iterative and incremental process where a complex system arises out of relatively simple interactions

Poorly designed programs are difficult to understand and modify.

Ease of maintenance – Your code will be read more often then it is written.

Subjective

⇒ Desirable Internal Design Characteristics ?

① Minimal complexity

Keep it simple

Maybe you don't need high levels of generality.

② Extensibility

Design for today but with an eye toward the future.

Note, this characteristic can be in conflict with "minimize complexity".

③ Loose coupling

minimize dependencies between modules

④ Reusability

Reuse is a hallmark of a mature engineering discipline



Portability

works or can easily be made to work in other environments

Maintainable Design

- Cost of system changes is minimal
- Readily adaptable to modify existing functionality and enhance functionality
- Design is understandable
- Changes should be local in effect

MCQS IMP

High Cohesion and Low Coupling

Coupling

- Coupling is the measure of dependency between modules/components.
  - The degree of interdependence between two modules/components
- A dependency exists between two modules/components if a change in one could require a change in the other.

Inter-component relationship

Measuring Coupling

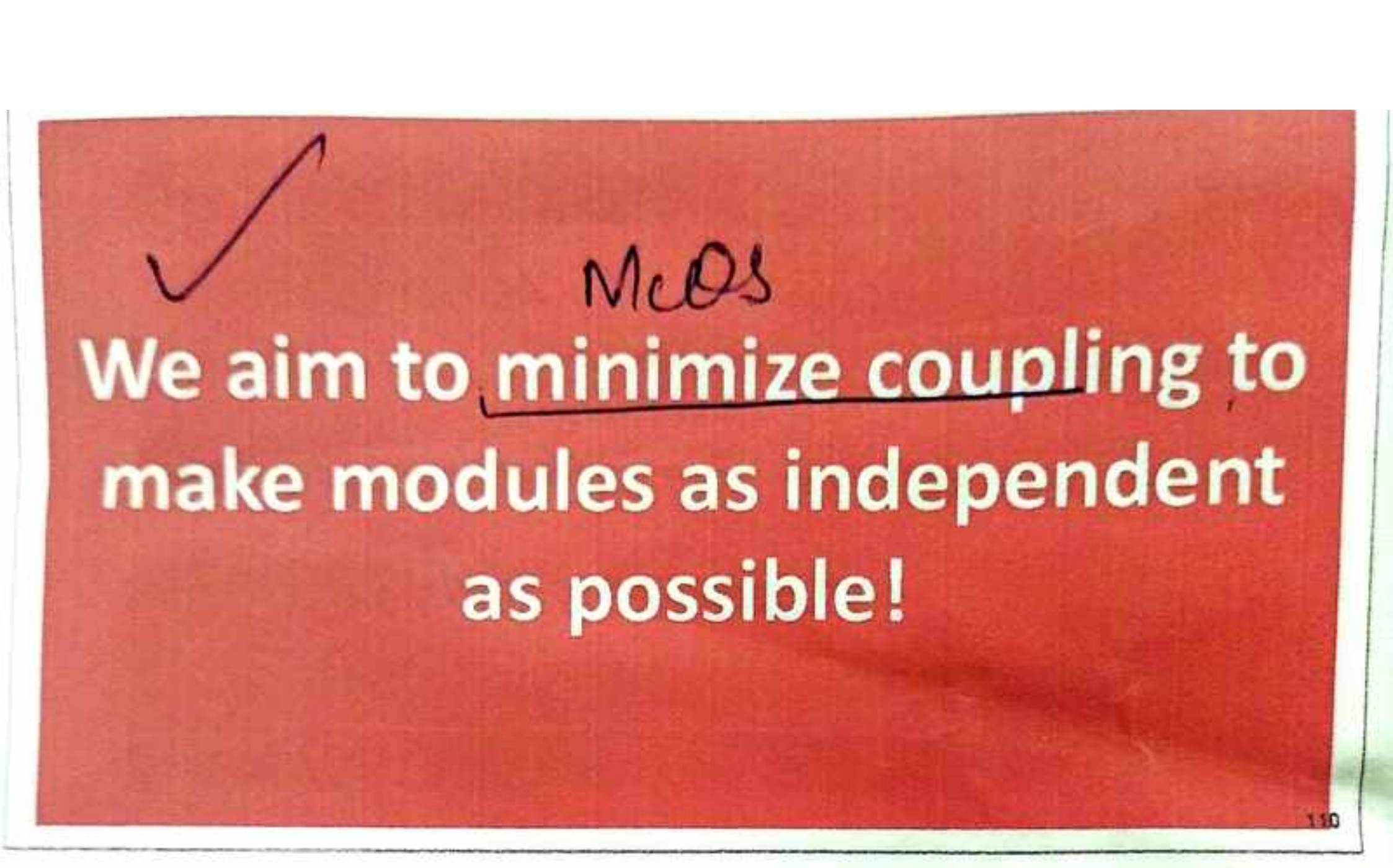
- The degree of coupling between modules is determined by:
  - The number of interfaces between modules (quantity)
  - Complexity of each interface (determined by the type of communication) (quality)

Subjective

Why is high degree of coupling bad?

1. Highly coupled systems have strong interconnection
2. High dependence of program units upon one another
3. Difficult to test, reuse, and understand separately

Loosely coupled modules are less dependent upon each other and hence are easier to change


MCOS
We aim to minimize coupling to make modules as independent as possible!
110

Subjective

Coupling – How to reduce it?

- Low coupling can be achieve by:
  - eliminating unnecessary relationships
  - reducing the number of necessary relationships
  - easing the 'tightness' of necessary relationships

Cohesion

Subjective

- Cohesion refers to the degree to which the elements inside a **module belong together**
- A measure of the strength of relationship between:
  - the methods and data of a class and some unifying purpose or concept served by that class
  - the class's methods and data themselves

A module has high cohesion if all of its elements are working towards the same goal

High vs Low Cohesion

Highly Cohesive Modules

- Easier to understand
- More Robust
- More Reliable
- More Reusable

Lowly Cohesive Modules

- Difficult to understand
- Difficult to maintain
- Difficult to test
- Difficult to reuse

Objective (Imp)

High Cohesion and Low Coupling - Benefits

Highly Cohesive Modules

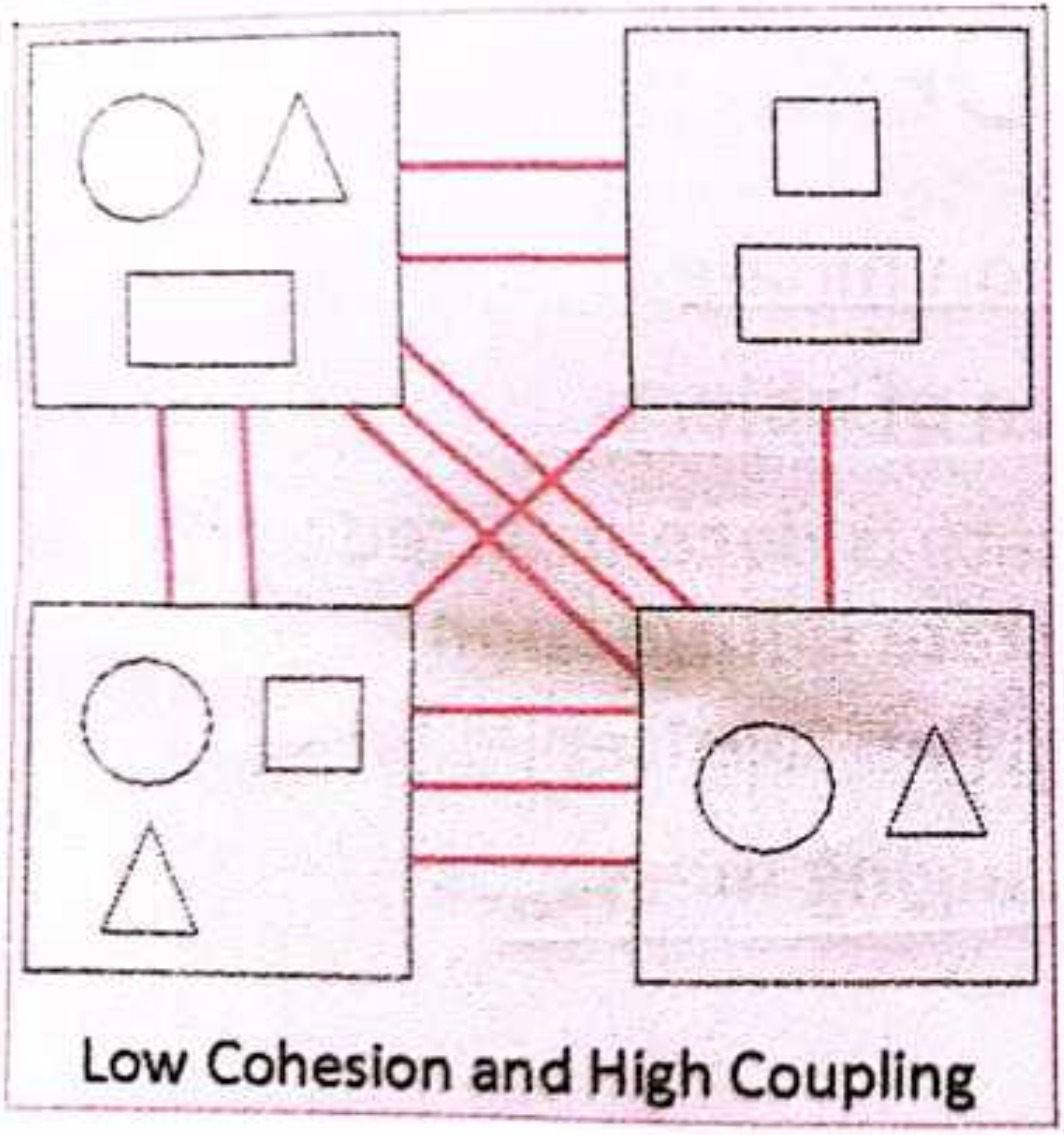
- easier to read and understand
- Easier to modify
- Increased potential for reuse
- Easier to develop and test

Lowly Coupled Modules

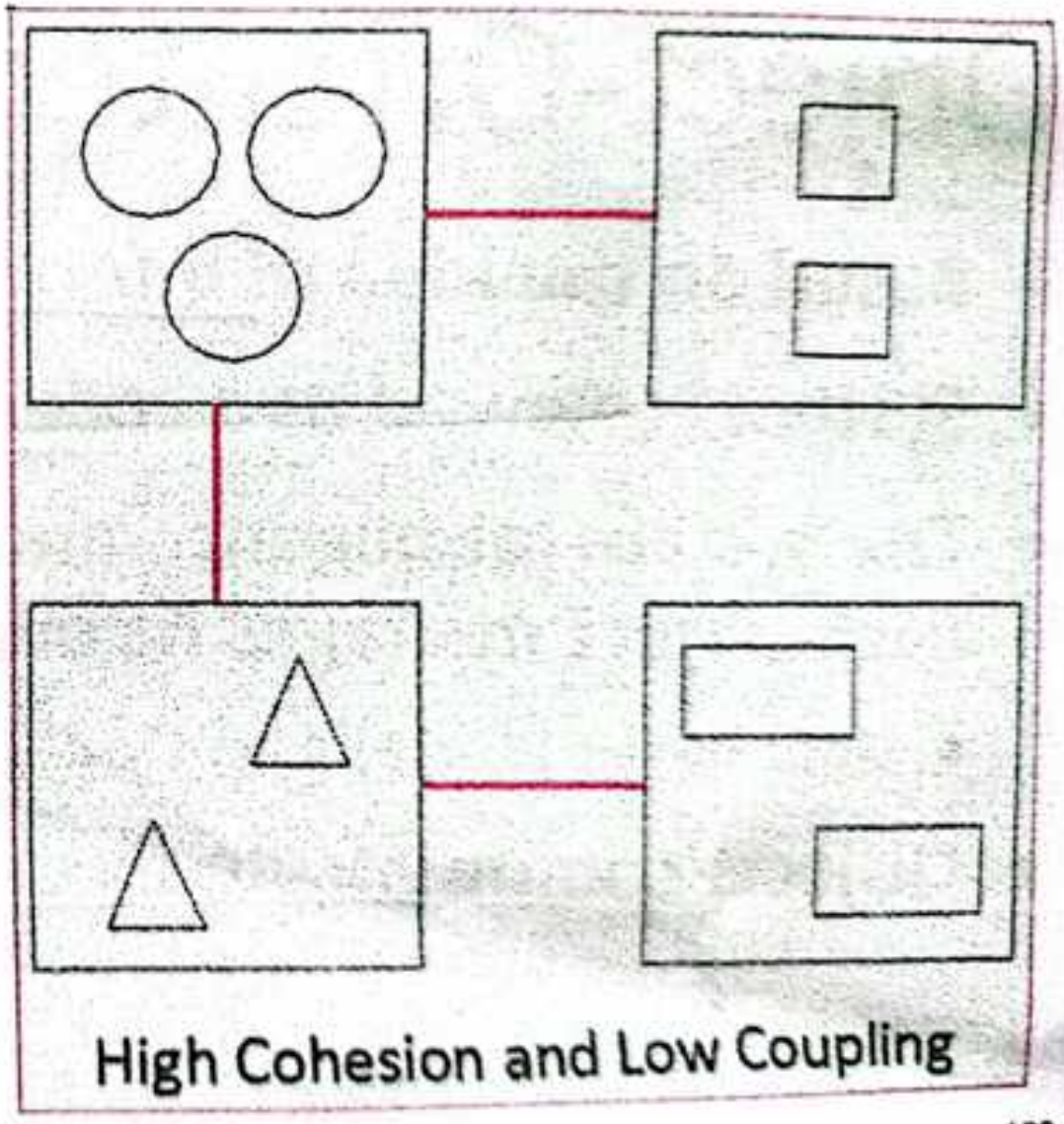
- easier to read and understand
- Easier to modify
- Increased potential for reuse
- Easier to develop and test

Surprised?

Are they somehow related?



Low Cohesion and High Coupling



High Cohesion and Low Coupling

MCQs

Cohesion and Coupling

- **Coupling:** Inter-component relationships
- **Cohesion:** Intra-component relationships

MCQs

The best designs have high cohesion within a module and low coupling between modules

Cohesion and Coupling

Tend to be Inversely Correlated



MCQS

When Coupling is high, cohesion tends to be low and vise versa

END.

Software Design Strategies

1. Structured Design
2. Object-Oriented Design

Q: How many software design strategies.

Structured Design

- System is designed from a functional view point
- Conceived as a set of functions that are used to solve the problem at hand
- The system state is centralized and is shared among the functions operating on that state

• Action Oriented ..

MCQs

Object-Oriented Design

- Based on the idea of information hiding
- System is viewed as a collection of objects
- The system intelligence and state is decentralized and each object manages its own state information
  - Attributes (state) and Operation (intelligence)
- Objects communicate by exchanging messages

Maintainable Design

What?

- Cost of system changes is minimal
- Design is understandable
- Changes should be local in effect

How?

- **Low coupling and high cohesion**
- **Self contained components**
- **Close relationship of data and function**

Object-Oriented Design

The Object

MCQs + Short

- An object is a tangible entity that exhibits some well defined behavior.
- An object represents an individual, identifiable item, unit, or entity, either real or abstract, with a well defined role in the problem domain

Separate → MSQ. MCQs MCQ

An object has state, behavior, and identity

The State

- **The state of an object is encapsulated within the object.**

Mc

Current values of the properties of the object

The Behavior

- How an object acts and reacts in terms of its state changes and message passing
  - How it acts and reacts

Part of System Intelligence Delegated to the Object

The Identity

- Identity is that property of an object which distinguishes it from all other objects

Who will provide a specific service?

The Class

Q: diff b/w class & object?

- Represents similar objects
  - Its instances

Short + MCQs.

Template of an object
Properties and Behavior

Software Design and Architecture

END!

The Class

The Class

- A class represents an abstraction MCQs.
  - Specifies an interface (the outside view - the public part)
    - What can I do for you?
      - declaration of all the operations applicable to instances of this class
  - Defines an implementation (the inside view - the private part)
    - How is it actually done?
      - The structure – how the information is maintained
      - The implementation of the operations declared in the interface
        - How a task is performed – the algorithm

Software Design and Architecture

Topic#023

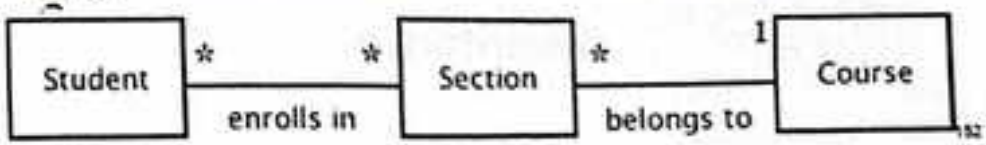
END!

Relationship between Classes

- Classes and Objects – two building blocks of the system
- Behavior of the system is defined by the manner in which the objects interact with one another
- In order to construct a software system, we need *mechanisms* that create connections between these building blocks
  - Association
  - Inheritance

Association Imp. MCQs.

- Indicates that the objects of the two classes are related in some non-hierarchical way
- Implies that an object of one class is making use of an object of another (or same) class
- Indicated simply by a solid line connecting the two class icons.



Objects - Association

Association

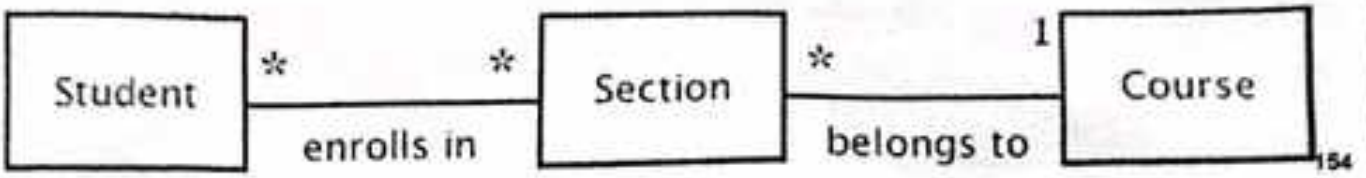
- Does Not imply that there is always a link between all objects of one class and all objects of the other
- Does imply that there is a persistent, identifiable connection between two classes.

Conceptual (between classes) as well as physical (between objects) relationship

MCQS

Attributes of Associations

- Descriptive name
- arity of the connection
- Direction
- Roles



154

Aggregation

- A form of association that specifies a whole-part relationship between an aggregate (a whole) and a constituent part
- Indicated by a "hollow" diamond

MCQs . .



157

Software Design and Architecture

Topic#025

Composite Aggregation

- Also known as Composition
  - has been recognized as being significant
- Implies that each instance of the part belongs to only one instance of the whole, and that the part cannot exist except as part of the whole.
- Composition is indicated with a filled-in diamond



Inheritance

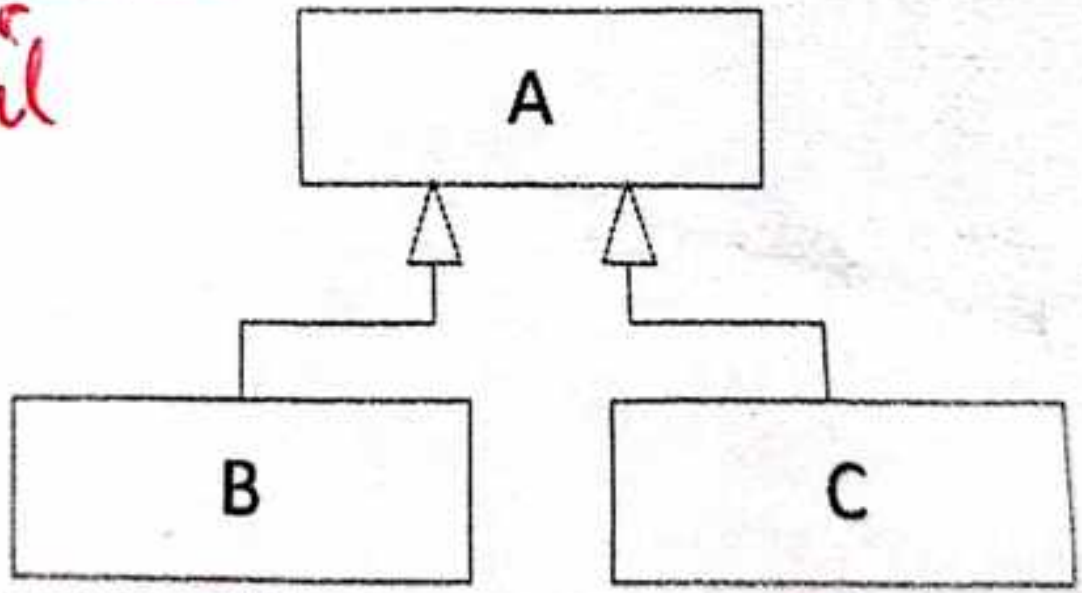
- Represents a hierarchy of abstraction in which one class inherits from one or more super-classes.
- One class shares the structure or behavior defined in one or more classes.
- A subclass augments or redefines the existing structure and behavior of its super-class.

Generalization/Specialization hierarchy

Inheritance Hierarchy

- Show the relationships among superclasses and subclasses
- A triangle shows a generalization

Inherit



Abstract Class

- It provides a basic implementation that other classes can inherit from
- An instance of the abstract class cannot be created

MCQS.

Interface

- A mechanism to achieve abstraction
- A collection of abstract methods MCQs.
  - No state or implementation

Subjective

Interface vs Abstract Class

- A class that implements an interface must provide an implementation of all the methods of that interface
- Interfaces can have no state or implementation

- Abstract classes can be inherited without implementing the abstract methods (though such a derived class is abstract itself)
- Abstract classes may contain state (data members) and/or implementation (methods)

As general OO terms, the differences are not necessarily well-defined.

Implementing Relationships

172

MCQS.

(OOP as defined by Grady Booch)

A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

175

Relationships

- Hierarchy of classes MCQS.
  - Inheritance - relationship among classes
- Cooperative collections of objects
  - Association - relationship among instances of classes

Object and Class Relationships

End - Object Oriented Programming Defined

Software Design and Architecture

Topic#029
Relationships – Basic Concepts
Association

178

Association

MCQs + Subjective

An association is a relationship among two or more specified classifiers that describes connections among their instances.

Association

MeO.

- Associations are the "glue" that holds together a system.

- Without associations, there is only a set of unconnected classes.

Composition ? Sully + Object

Meet

- A form of aggregation association with strong ownership and coincident lifetime of parts by the whole.
- A part may belong to only one composite.

Composition

- Parts with non-fixed multiplicity may be created after the composite itself.
- Once created, they live and die with it (that is, they share lifetimes).
- Such parts can also be explicitly removed before the death of the composite.

O2R Mapping

MCQs.

- aka – **O2R**, **ORM**, **O/RM**, and **O/R**
  (object to Relational mapping)
- Mapping attributes
- Mapping Inheritance
- Mapping Association
- Mapping Composition

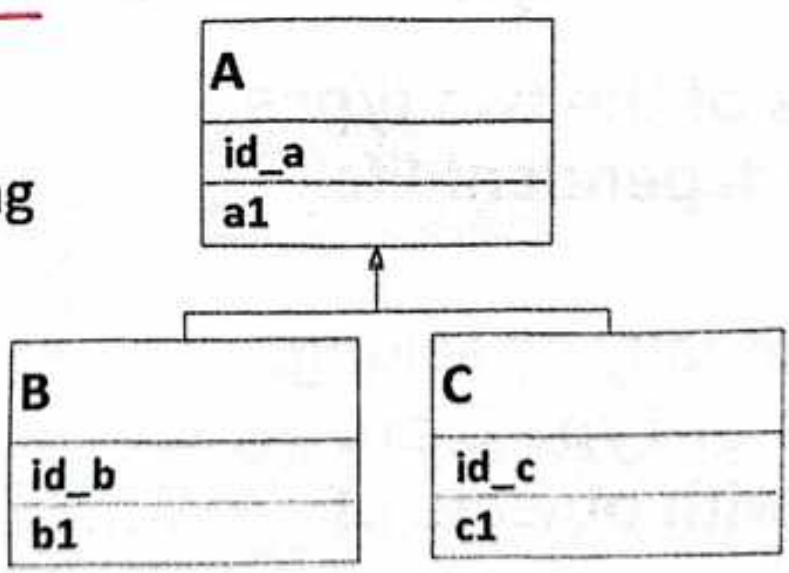
End of topic

Subjective + MCQ.

Mapping Inheritance

1/2/3

- Filtered Mapping
- Horizontal Mapping
- Vertical Mapping



Filtered Mapping

- All the classes of an inheritance hierarchy are mapped to one table.

TableName
a_id
a1
b_id
b1
c_id
c1

Horizontal Mapping

(2)

- Each concrete class is mapped to a separate table and each such table includes the attributes and the inherited attributes of the class that it is representing.

TableForB
a_id
a1
b_id
b1

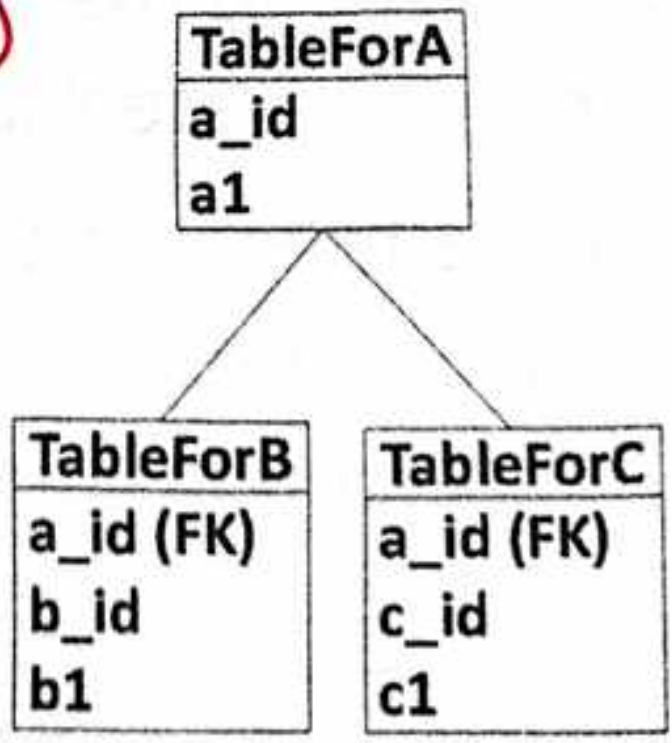
TableForC
a_id
a1
c_id
c1

End of topic

Vertical Mapping

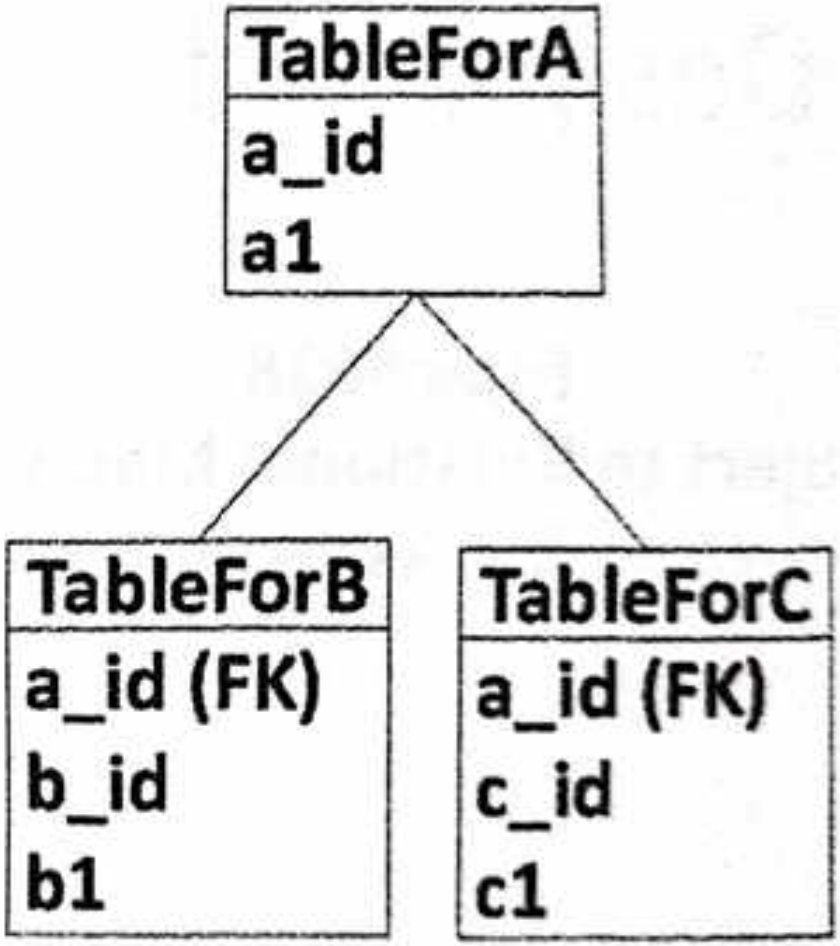
(3)

- Each class of the inheritance hierarchy, whether abstract or concrete, is mapped to a separate table.



Vertical Mapping

- To maintain the inheritance relationship between parent and child classes, OID of the parent class is inserted in the child classes as a foreign key.



Software Design and Architecture

Topic#040
Object to Relational Mapping
Summary

Lab # 04

End of topic

V. IMP

	What	Life-time	Holding / visibility	cardinality
Inheritance	Is-a relationship parent-child relationship	Object of base class is created and destroyed with derived class object	Exclusive Parent does not know about the child	MCQ 1:1
Composition	whole-part relationship	Part is created and destroyed with the whole	Exclusive Part does not know about the whole	MCQ 1:1, 1:m
Association	Using relationship link	independent	Non-exclusive Can be one-way or bi-directional	MCQ 1:1, 1:m, m:m

Subjective + MCQs.

- Compiler maintains two things:

vtable: A table of function pointers.
It is maintained per class.

vptr: A pointer to vtable.
It is maintained per object.

- Compiler adds additional code at two places to maintain and use *vptr*.
  - **Code in every constructor. This code sets vptr of the object being created. This code sets *vptr* to point to *vtable* of the class.**
  - **Code with polymorphic function call.**

Subjective

- Wherever a polymorphic call is made, compiler inserts code to first look for *vptr*.
- Once *vptr* is fetched, *vtable* of derived class can be accessed. Using *vtable*, address of derived class.

End of topic

Subj + Objective

- The main advantage inheritance has over composition is that it allows objects of the derived class to be dynamically bound in the same hierarchy as its base classes.

- It is also necessary where virtual methods must be overridden by the derived class
  - a composite aggregate cannot do this.

End of topic

Software Design and Architecture

L#5 ✓

Topic#048

SOLID Design Principles - Introduction

SMP

Maintainable Design

- Cost of system changes is minimal
- Readily adaptable to modify existing functionality and enhance functionality
- Design is understandable
- Changes should be local in effect
- Design should be modular

Goal: low coupling and high cohesion

Solid Stands for.

S.O.L.I.D. Design Principles

- Intended to make software designs more
  - Understandable
  - Flexible
  - Maintainable

SOLID Principle - History

- Originally gathered by Robert C. Martin (aka Uncle Bob)

MCOS



Robert C. Martin

SOLID Principle - History

- Arranged by Michael Feathers into an easily remembered, albeit ironic, mnemonic: SOLID.

S.O.L.I.D. ✓

M.I.M.P

MCQs + Subjective

Principle	Objective
Single Responsibility Principle (SRP)	Increased Cohesion
Open-Closed Principle (OCP)	Flexibility, reusability, and maintainability
Liskov Substitution Principle (LSP)	Proper Inheritance
Interface Segregation Principle (ISP)	Deals with non-cohesive interfaces for reducing coupling
Dependency Inversion Principle (DIP)	Creation of reusable frameworks Reduce inter-module coupling

Dependency Inversion Principle (DIP)	Reduce inter-module coupling

Single Responsibility Principle (SRP)



A class should have only one reason to change

}
}

Single Responsibility!

The Open-Closed Principle (OCP)

- Coined by Bertrand Meyer:

- *A software artifact should be open for extension but closed for modification.*

The behavior of a software artifact ought to be extendible, without having to modify that artifact.

The Open-Closed Principle (OCP)

- Not simply a principle that guides in the design of classes and modules.
- Fundamental to good architectural design

If simple extensions to the requirements force massive changes to the software, then the architects of that software system have engaged in a spectacular failure.

- Modules that conform to the open-closed principle have two primary attributes.
- They are "Open For Extension".
  - We can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.
- They are "Closed for Modification".
  - The source code of such a module is inviolate.
  - No one is allowed to make source code changes to it.

Liskov Substitution Principle (LSP)

- *Subtypes must be substitutable for their base types*
- *Deals with design of inheritance hierarchies*

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

McQs.

Validity is not Intrinsic

- A model, viewed in isolation, cannot be meaningfully validated.

- The validity of a model can only be expressed in terms of its clients.

What Went Wrong? (W^3) McQs.

- Isn't a Square a Rectangle?
- Doesn't the ISA relationship hold?

base classes that have cohesive interfaces.

```
interface IPost
{
    void CreatePost();
}
```

Initially an IPost interface with the signature of a CreatePost() method.

```
interface IPost
{
    void CreatePost();
    void ReadPost();
}
```

Later on, a new method ReadPost() added to it

Violation of ISP!
Old clients do not need the new interface

Interface Segregation Principle (ISP)

n programming, the interface segregation principle states that
no client should be forced to depend on methods it does not us

Put more simply: Do not add additional functionality to an existing interface by adding new methods.

nstead, create a new interface and let your class implement multiple interfaces if needed.

Deals with non-cohesive interfaces!

Coupling

352

ISP

- This principle deals with the disadvantages of "fat" interfaces. MCQs
  - Classes that have "fat" interfaces are classes whose interfaces are not cohesive.

- The interfaces of the class can be broken up into groups of member functions.
  - Each group serves a different set of clients.
  - Thus some clients use one group of member functions, and other clients use the other groups.

355

- Fat interfaces lead to inadvertent couplings between clients that ought otherwise to be isolated.

- Fat interfaces can be segregated into abstract base classes that break the unwanted coupling between clients.

Complying to ISP!

The Dependency-Inversion Principle (DIP)

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

In programming, the dependency inversion principle is a way to decouple software modules.

Dependency Injection

The Dependency-Inversion Principle (DIP)

DIP

- The problem of dependency is most often solved by using **dependency injection**.
- Typically, dependency injection is used simply by 'injecting' any dependencies of a class through the class' constructor as an input parameter.
  - Using association

Good OOD practices

- High cohesion and low coupling
- An object must be able to answer the following:
  - who I am? — identity
  - what I know? — data
  - what I do? — methods
  - who I know? — relationships
- Principle of delegation

Design Challenges with OO Paradigm

These are two design challanges

- **Poorly distributed system intelligence**
  - **god classes** → literature
  - (1) **Behavioral form** - **can't answer what I know**
  - (2) **Data form** - **can't answer what I do**

opposite to 1st
- have data but
- no intelligence

get information from everywhere
& then processing

god class problem – behavioral form

- **Action-oriented in the disguise of object-oriented**
  - **Central control mechanism**
  - **Super object that does most of the work leaving minor details to a collection of trivial classes**

1st identify & then the distribute the system intelligence throughout the system.

Distribute system intelligence horizontally as uniformly as possible

Subjective

Jocobson's Heuristic Imp.

Policy information should not be placed inside of classes involved in the policy decision because it renders them un-reusable by binding them to the domain that sets the policy.

409

MCQs

- A **discriminator** is a label that describes the criteria used in the specialization

- Can be thought of an attribute that will have a different value in each subclass

412

difference

Asymmetric Reflexive Association

- The ends of the association are semantically different from each other, even though the associated class is the same.

- Examples:
  - parent-child, supervisor-subordinate, and predecessor-successor, course-prerequisite

Symmetric Reflexive Association

- There is no logical difference in the semantics of each association end
- Example:
  - mutually exclusive courses
    - students who have taken one course cannot take another in the set
  - If course A is mutually exclusive with B, then course B is mutually exclusive with A

Imp

Design Patterns

"Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice"

- Christopher Alexander, A Pattern Language, 1977
- Context: City Planning and Building architectures

What Makes it a Pattern?

- **A Pattern must:**
  - **Solve a problem and be useful**
  - **Have a context and can describe where the solution can be used**
  - **Recur in relevant situations**
  - **Provide sufficient understanding to tailor the solution**
  - **Have a name and be referenced consistently**

- Describes the context of usage

- Patterns capture the static and dynamic *structure* and *collaboration* among key participants in software designs

- Especially good for describing how and why to resolve *non-functional issues*

- Patterns facilitate reuse of successful software architectures and designs.

Elements of Design Patterns

- Design patterns have four essential elements:
  - Pattern name
  - Problem
  - Solution
  - Consequences

MCQs + Subje

- **Related Patterns**: Closely related patterns

455

Categories of Patterns

MCQs + Subject

- **Creational patterns (5)**
  - Deal with initializing and configuring classes and objects
- **Structural patterns (8)**
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects
- **Behavioural patterns (11)**
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

458

456

Pattern Types

MCQs + Subjective

- Class Patterns (4)
  - Deal with relationships between classes and their subclasses
  - Established through inheritance, so they are static-fixed at compile-time
- Object Patterns (20)
  - Deal with object relationships
  - Established through association
  - Can be changed at run-time and are more dynamic

459

			• Proxy	• State • Strategy • Visitor

Subjective

Benefits of Design Patterns ?

- Design patterns enable large-scale reuse of software architectures and also help document systems
- Patterns explicitly capture expert knowledge and design trade-offs and make it more widely available
- Pattern names form a common vocabulary
  - Patterns help improve developer communication
- Patterns help ease the transition to OO technology

Drawbacks to Design Patterns ?

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple
- Patterns are validated by experience and discussion rather than by automated testing
- Integrating patterns into a software development process is a human-intensive activity.