VIRTUAL UNIVERSITY OF PAKISTAN

Software Architecture and Design

CS603

Complete Slides

GRADUP

Practice. Pass. Graduate.

thegradup.com

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 001 – 015

Software Design Concepts

© Virtual University of Pakistan

Software Design and Architecture

Topic#001
Design – What and Why

What is design?

Definition from Cambridge Dictionary

Verb

to make or draw plans for something, for example clothes or buildings

What is design?

Definition from Cambridge Dictionary

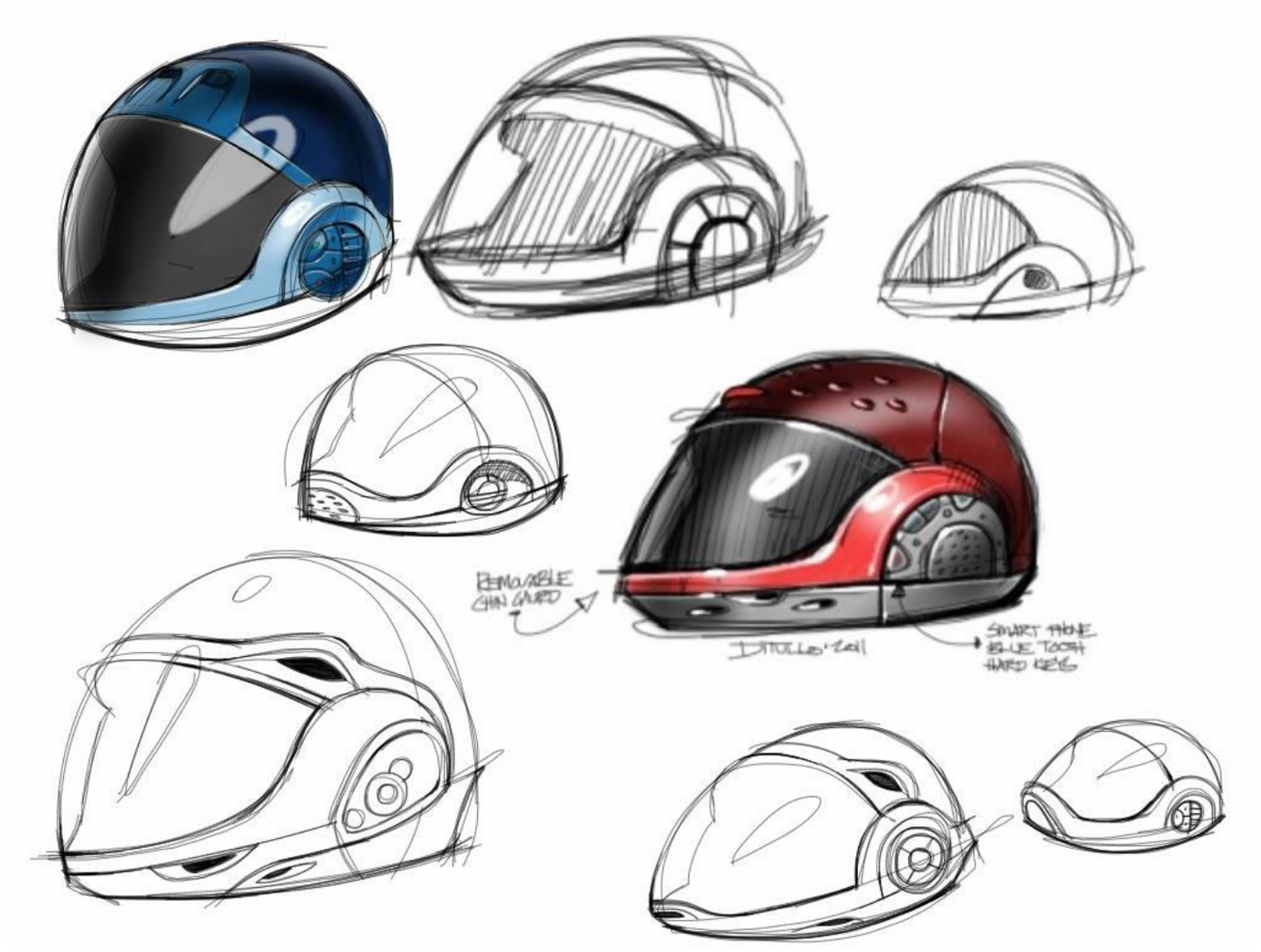
Noun

a drawing or set of drawings showing how a building or product is to be made and how it will work and look

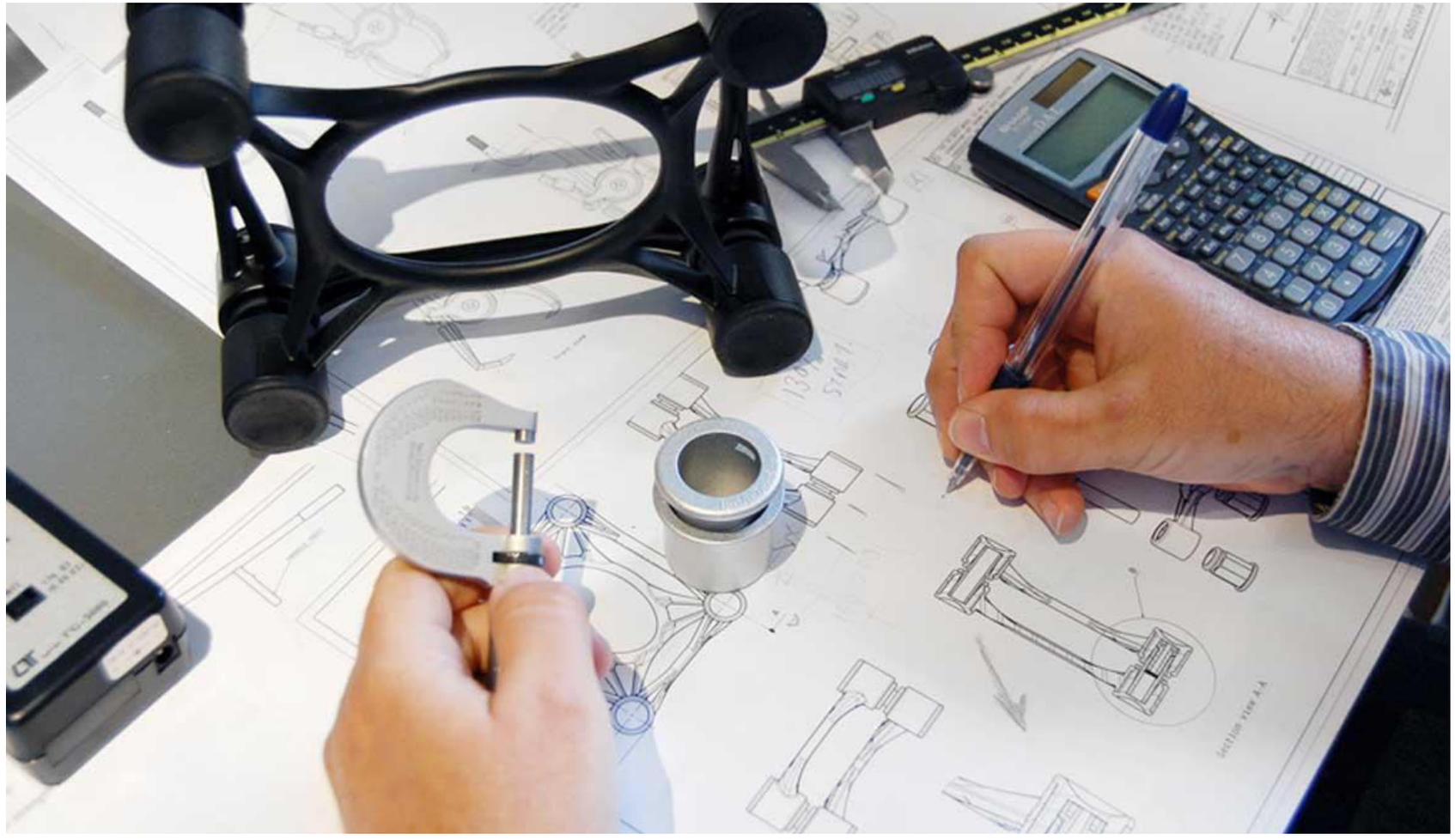




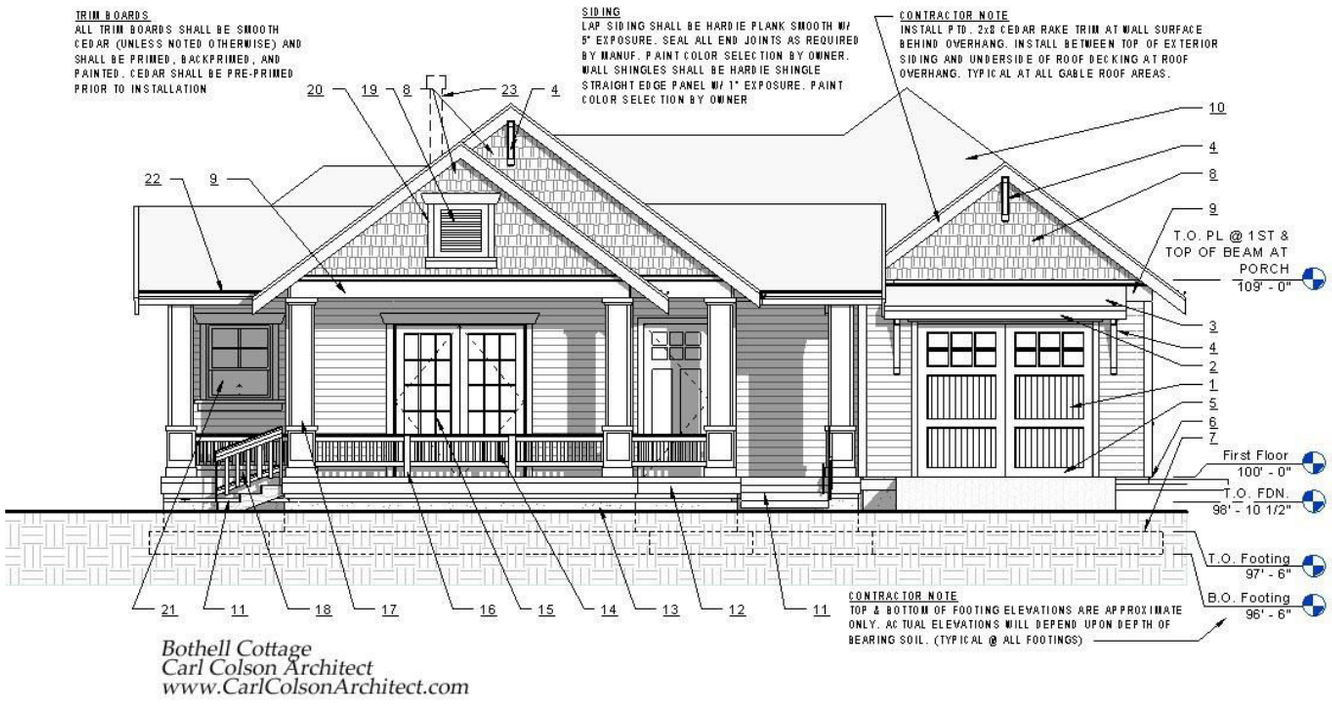
Dress Design



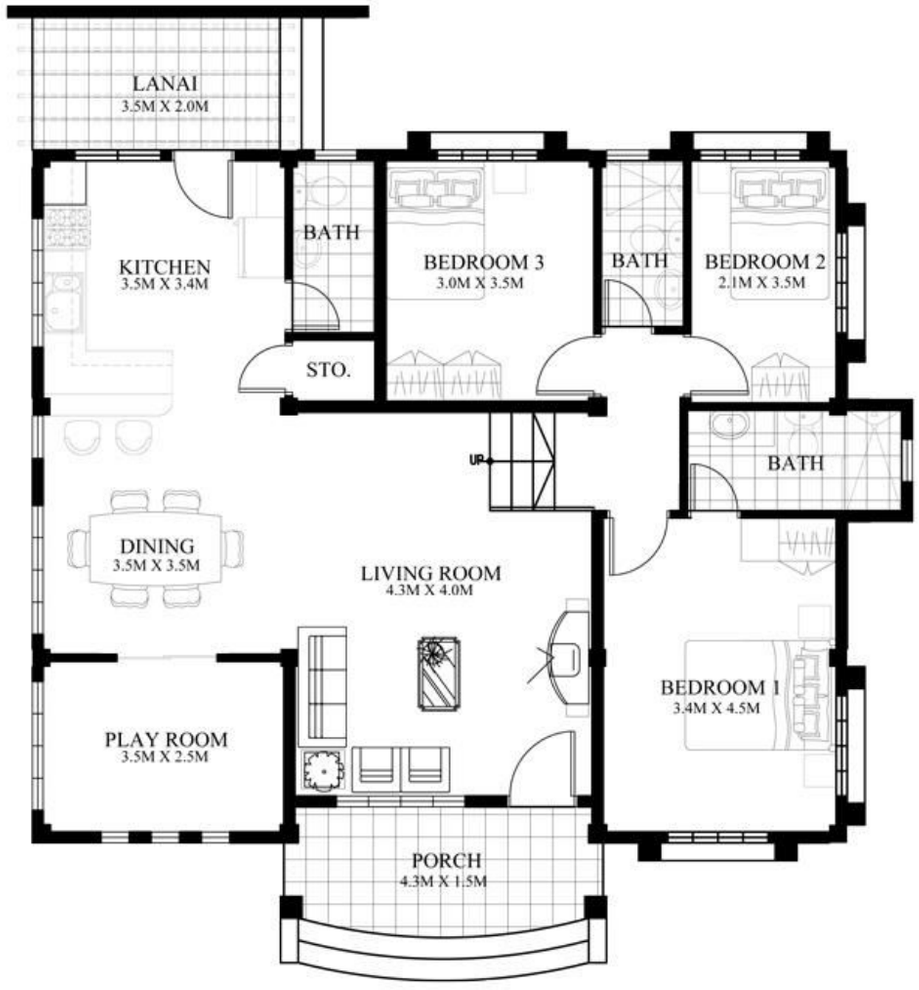
Product Design



Mechanical Design



Building Elevation Design



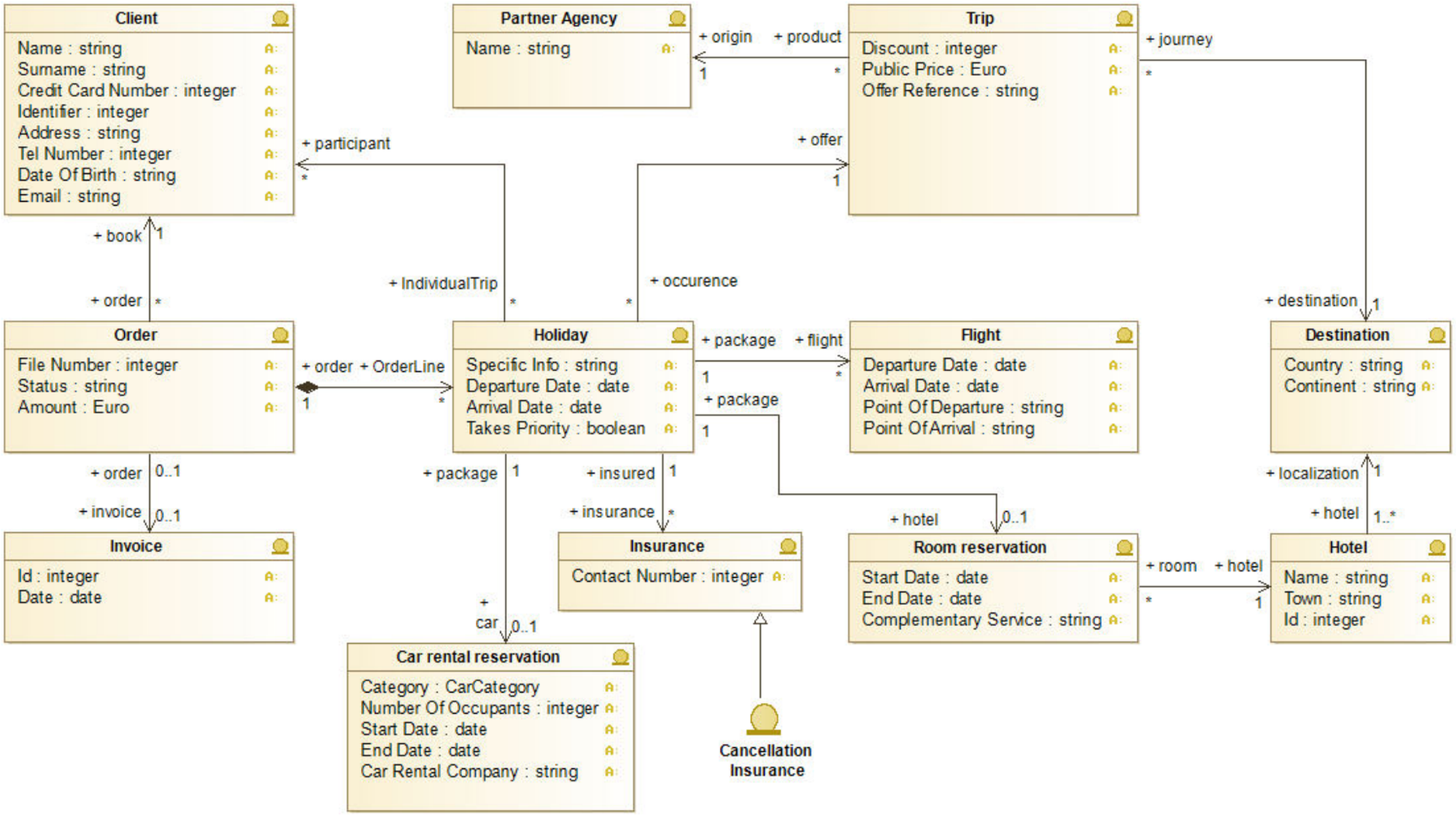
Building Floor Plan Design



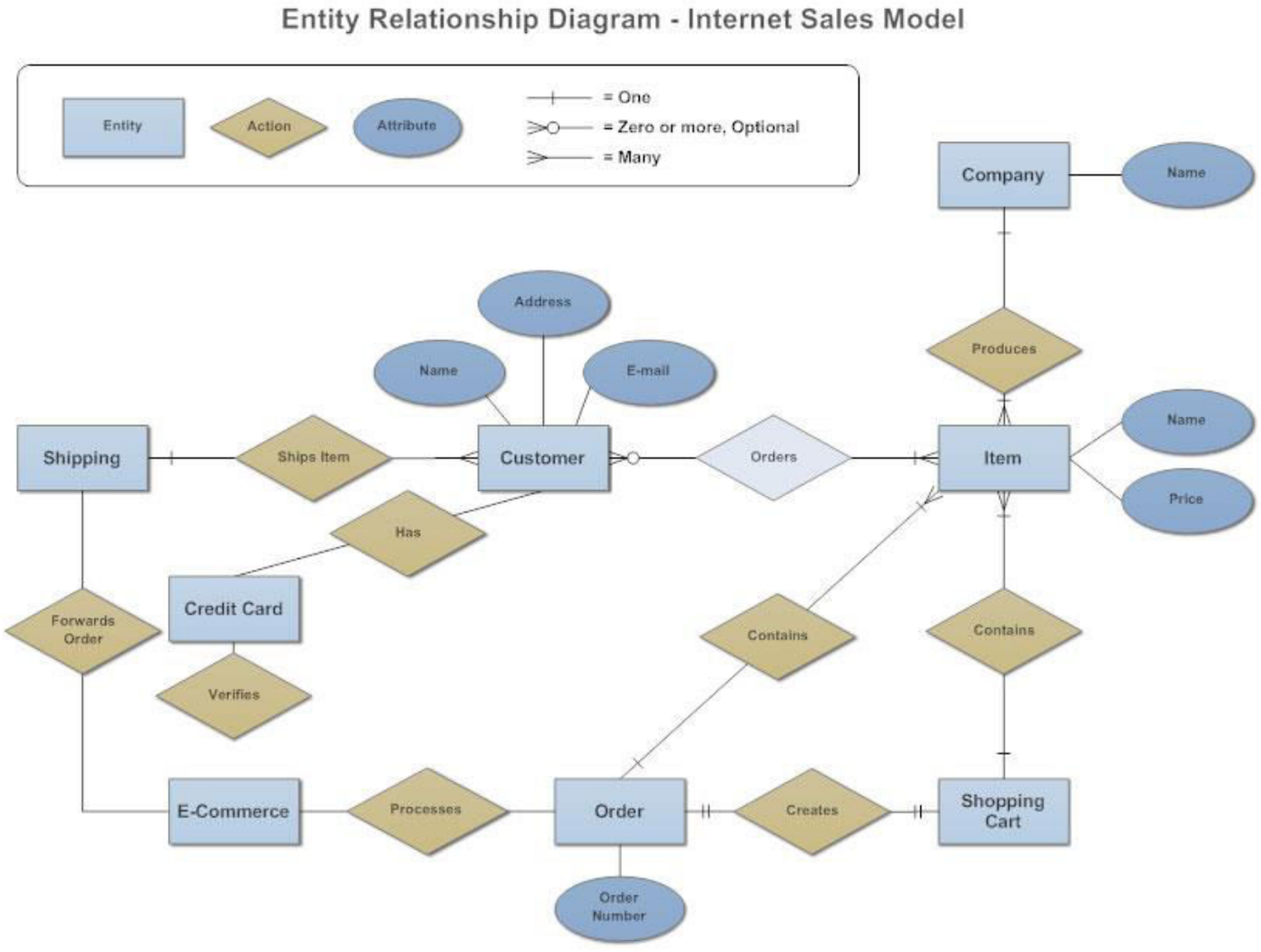
Landscape Design



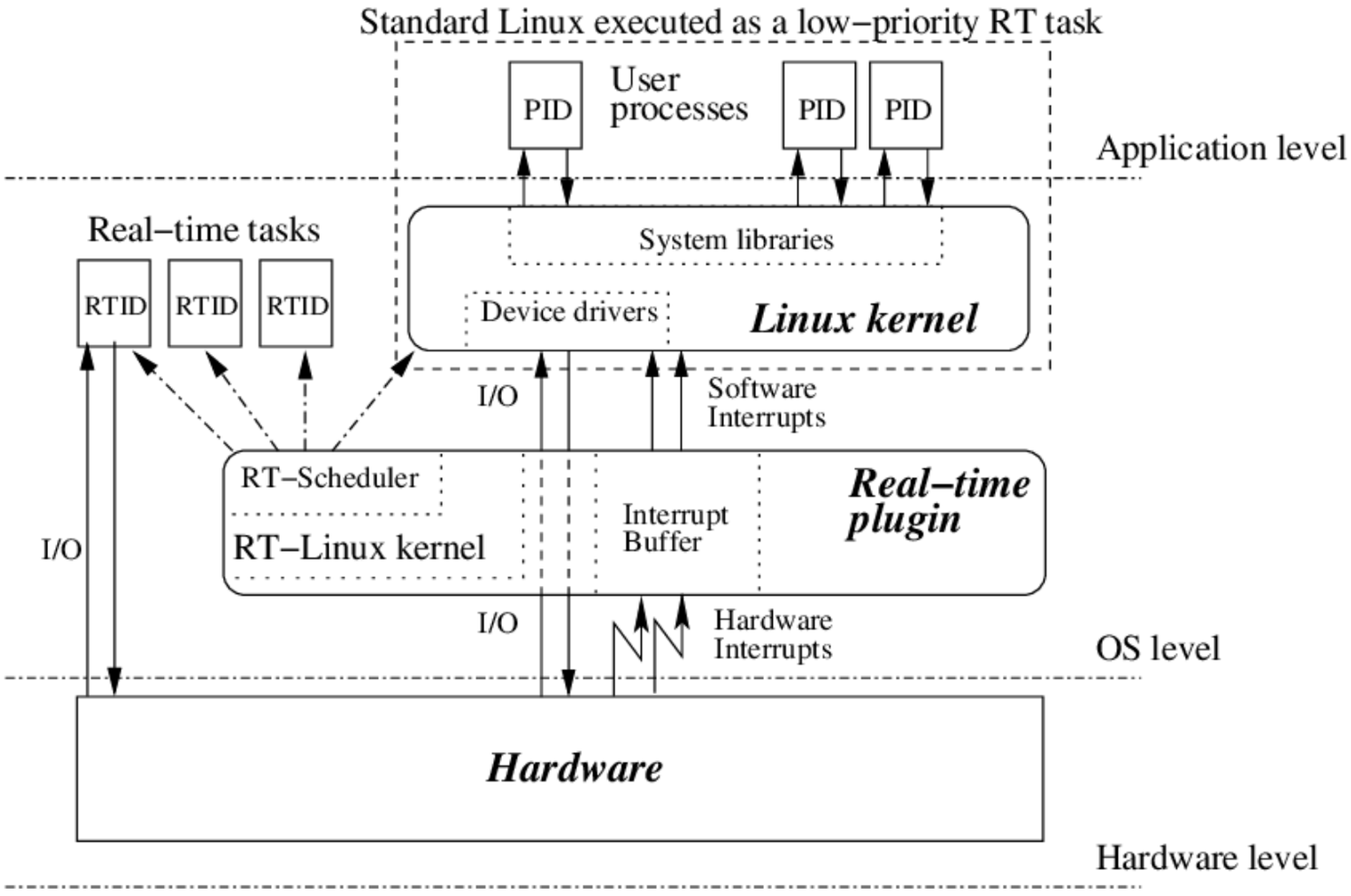
Interior Design



Software Design



Software Design



Software Design

Why should structures/systems need to be designed at all?



"You can use an eraser on the drafting table or a sledgehammer on the construction site.“

--Frank Lloyd Wright


© @candywarner1 /Instagram





Software Design and Architecture

Topic#001

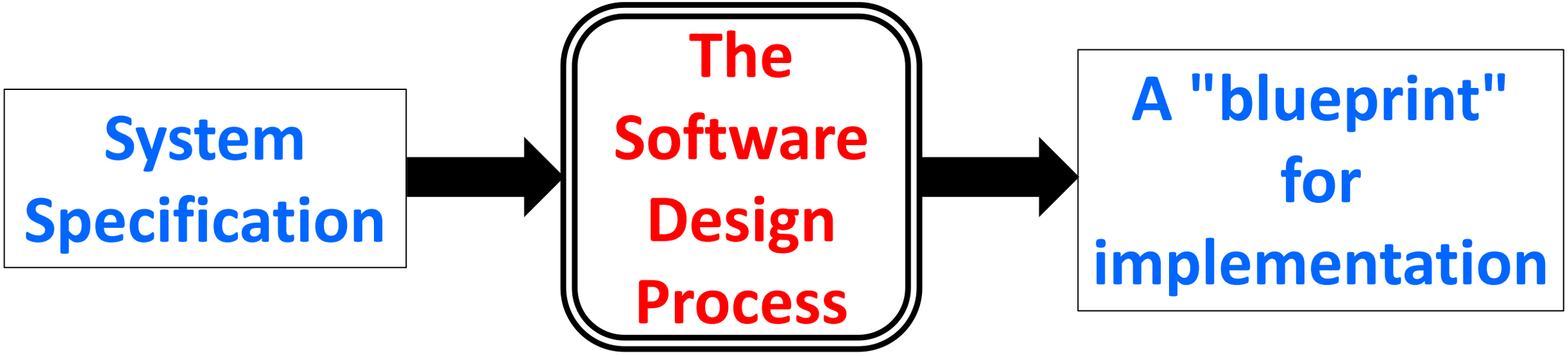
END!

Software Design and Architecture

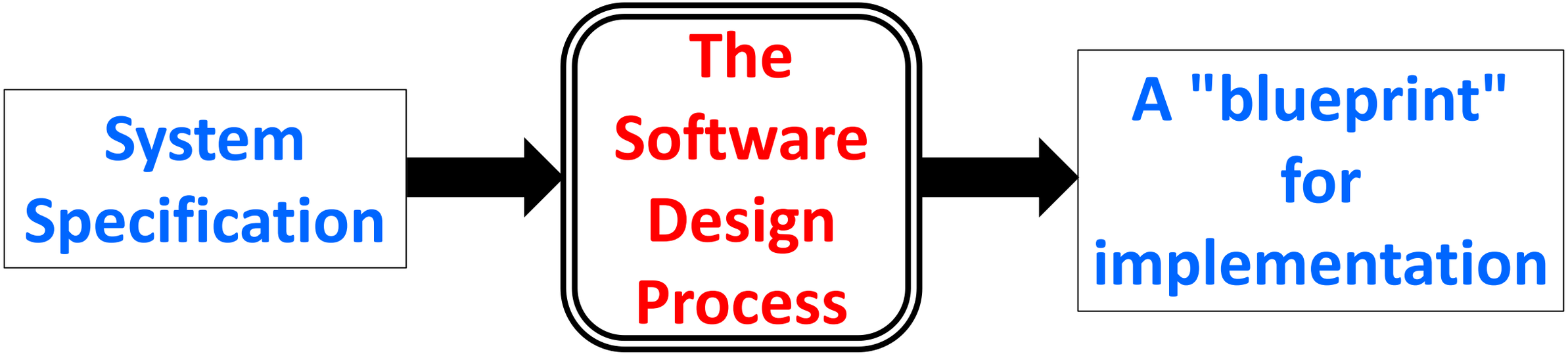
Topic#002
Design – Objectives

© Virtual University of Pakistan

Software Design Objectives



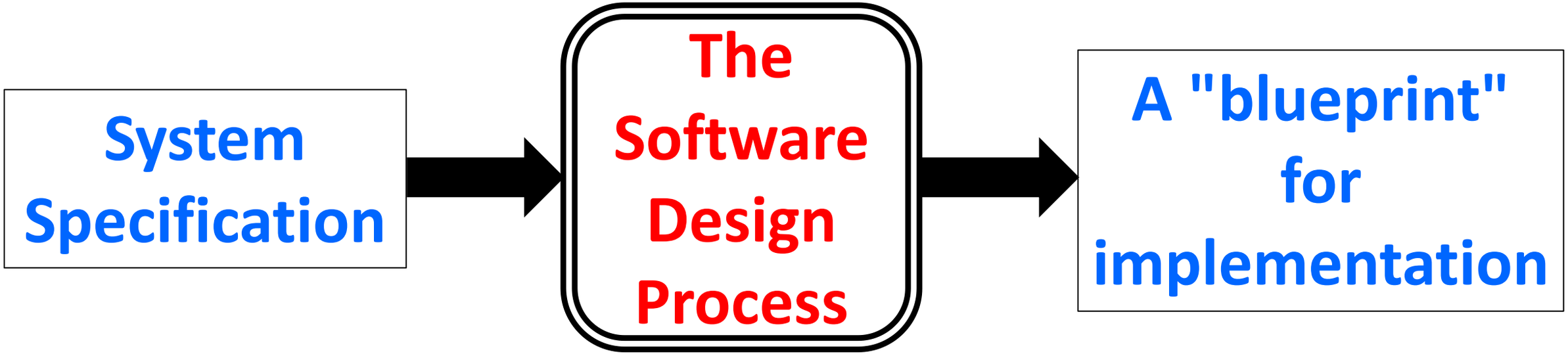
Software Design Objectives



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

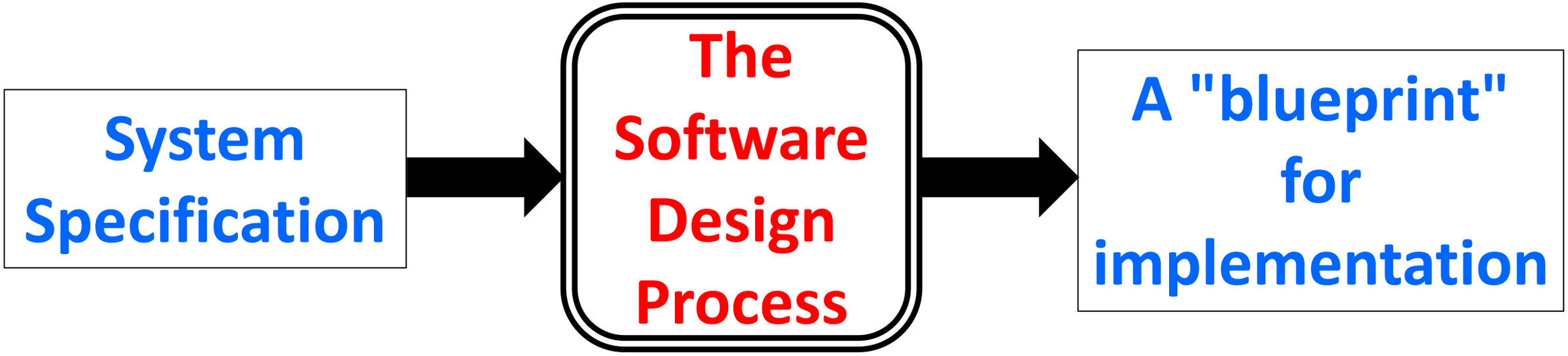
- **the problem domain**
- **implementation technologies**

Software Design Objectives



translate system specifications into plans for the technical implementation of the software

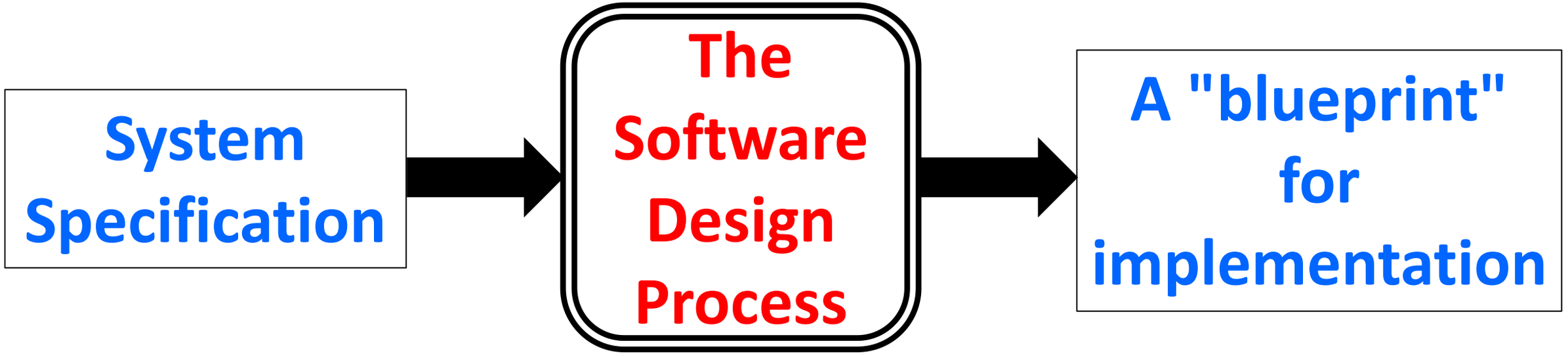
Software Design Objectives



Serves as a technical plan for implementation

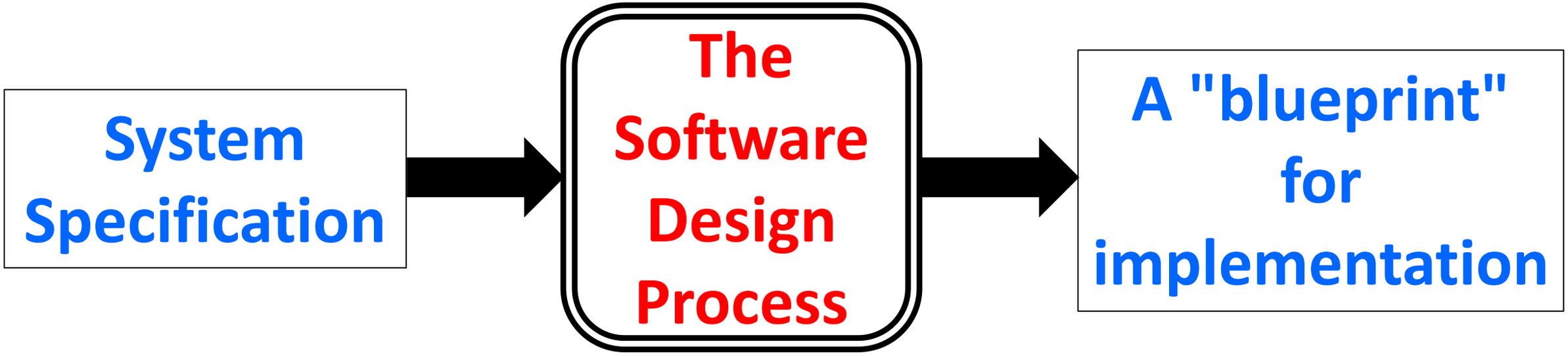
© Virtual University of Pakistan

Software Design Objectives



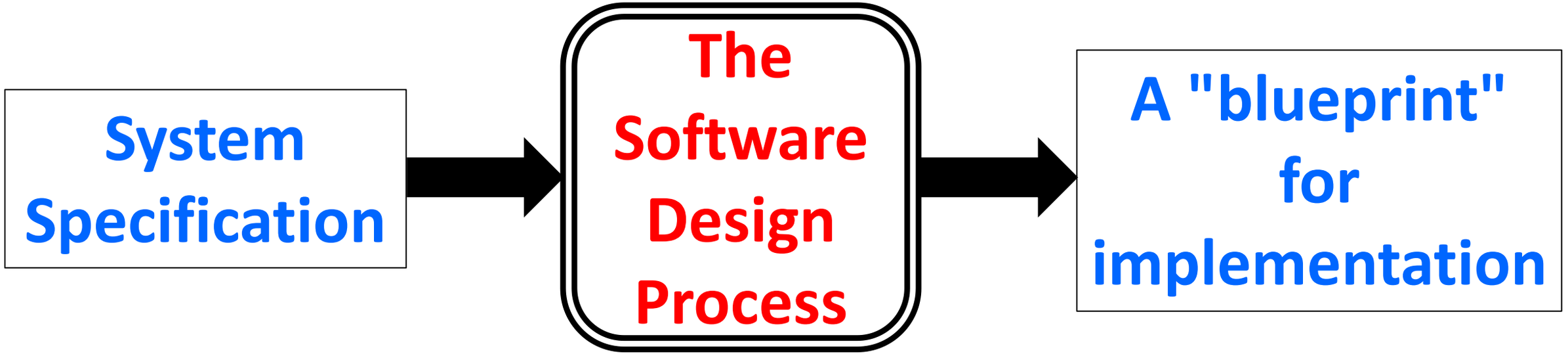
Specifies the overall structure and organization of the eventual code

Software Design Objectives



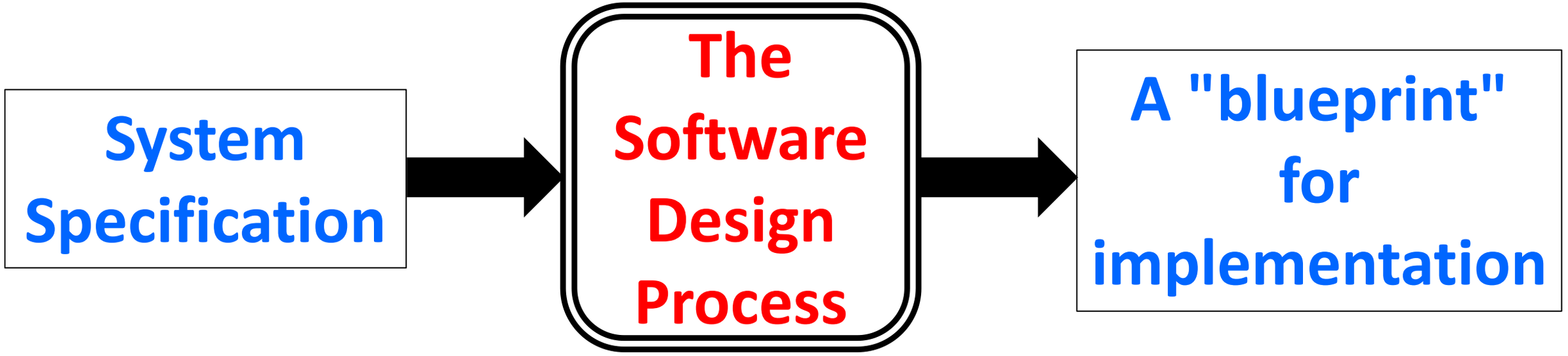
The system or program should meet customer's needs

Software Design Objectives



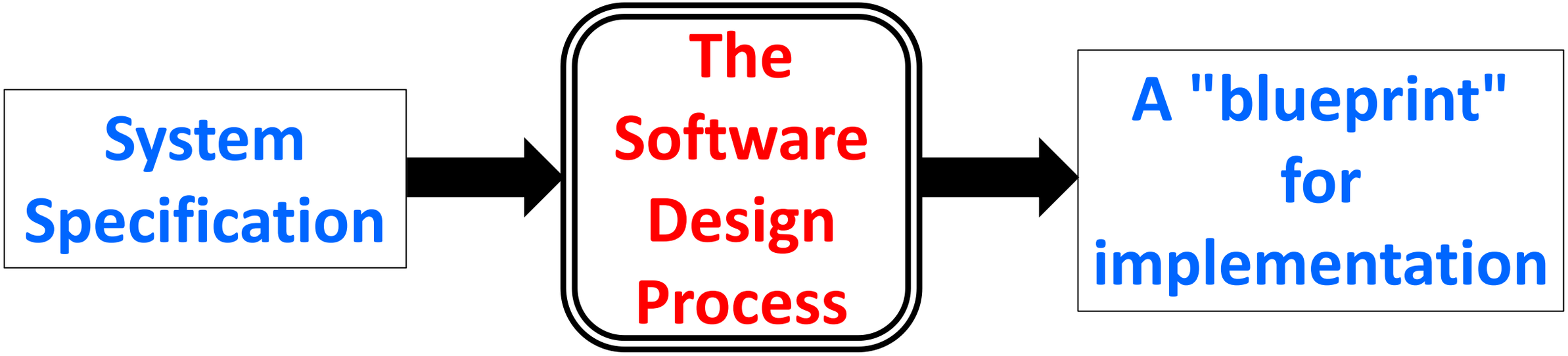
Should be easy to implement

Software Design Objectives



Should be “efficient”

Software Design Objectives



Should be “easily extendible” to meet new needs

Software Design and Architecture

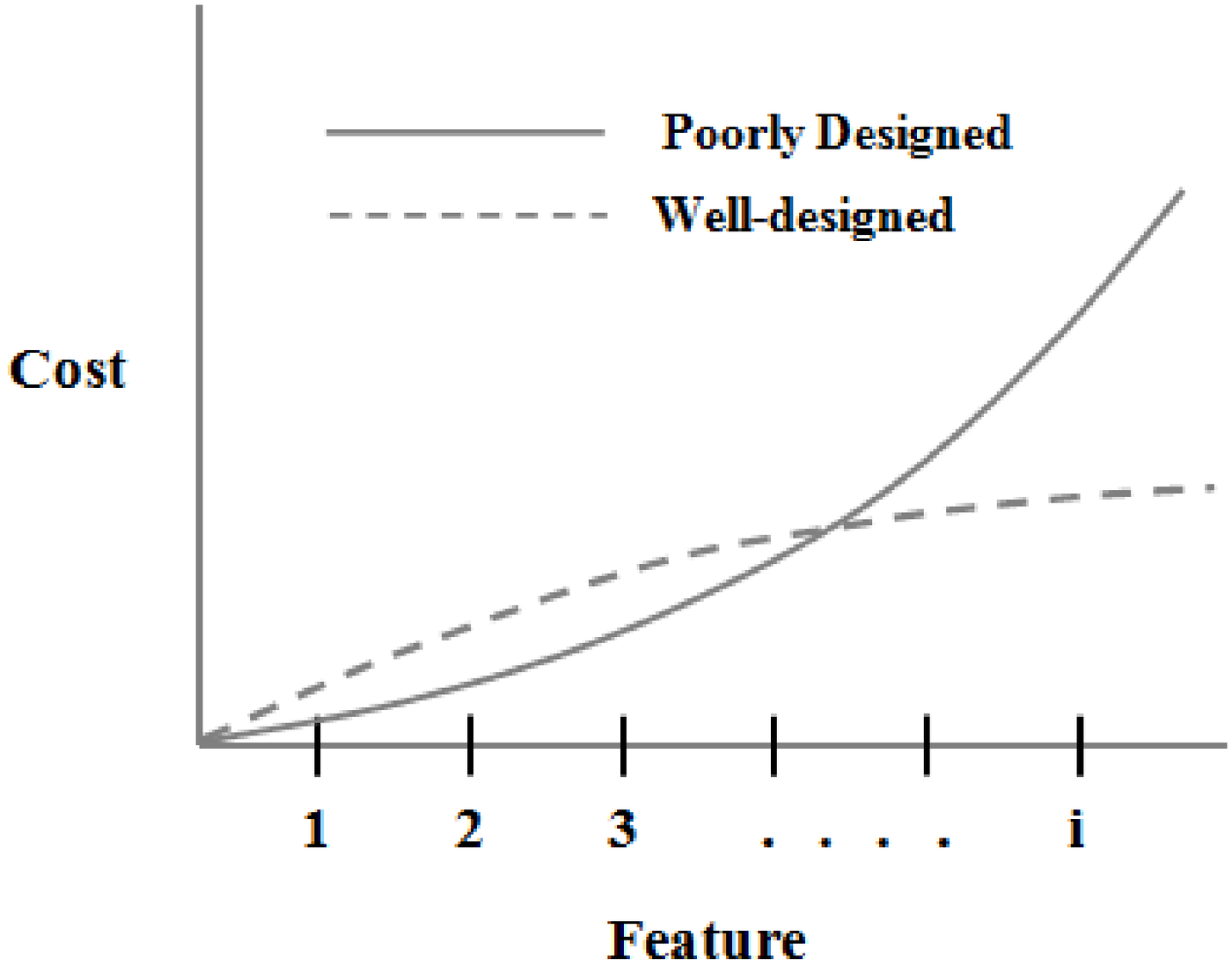
Topic#002

END!

Software Design and Architecture

Topic#003
Software Design – Complexity

Poorly designed programs are difficult to understand and modify



Cost of adding the i^{th} feature to a well-designed and poorly designed program

Software Design and Architecture

Topic#003

END!

Software Design and Architecture

Topic#004
Software Design
Complexity and Size

© Virtual University of Pakistan

The size does matter!

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

Software Design and Architecture

Topic#004

END!

Software Design and Architecture

Topic#005
Software Design
Types of Complexities

© Virtual University of Pakistan

Two Types of Complexity in Software

Essential complexities – complexities that are inherent in the problem.

Two Types of Complexity in Software

Accidental/incidental complexities –

complexities that are artifacts of the solution.

Two Types of Complexity in Software

The total amount of complexity in a software solution is:

Essential Complexities

+

Accidental complexities

Design
An Antidote to Complexity

primary tool for managing essential and accidental complexities in software

Software Design and Architecture

Topic#005

END!

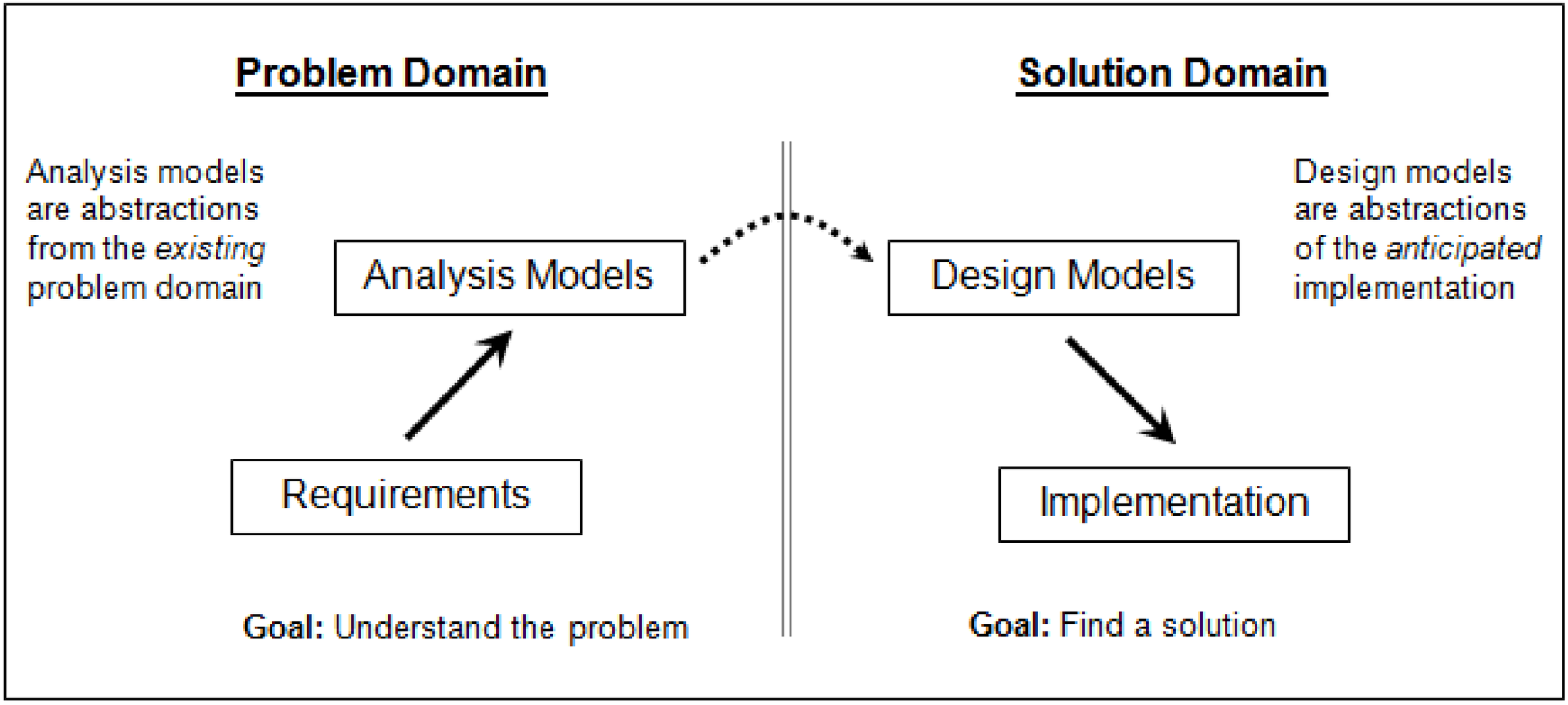
Software Design and Architecture

Topic#006

Why is Software Design Hard?

Why Design is Hard

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


Problem Domain
Solution Domain
Analysis models are abstractions from the existing problem domain
Analysis Models
Requirements
Design Models
Implementation
Design models are abstractions of the anticipated implementation
Goal: Understand the problem
Goal: Find a solution

Software Design and Architecture

Topic#006

END!

Software Design and Architecture

Topic#007
Software Design
A science or an art?

Difficulty with design



© Virtual University of Pakistan

Software Design and Architecture

Topic#007

END!

Software Design and Architecture

Topic#008
Software Design
A Wicked Problem

A wicked problem

A problem that can only be clearly defined by solving it

Design is a wicked problem!

Even those proficient at software design might not be able to fully explain how they arrived at their result

Software Design and Architecture

Topic#008

END!

Software Des Architecture





T 9

H mak sign process

ore s tic and

pr e?

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.

Software Design and Architecture

Toipc#009

END!

Software Design and Architecture

Topic#010

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

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

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

Modularity

subdivide the solution into smaller easier to manage components

(divide and conquer)

Hierarchical Organization

larger components may be composed of smaller components

Information Hiding

hide details and complexity behind simple interfaces

Abstraction

use abstractions to suppress details in places where they are unnecessary

Software Design and Architecture

Topic#010

END!

Software Design and Architecture

Topic#011
Characteristics of Software Design

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



© Virtual University of Pakistan

Design is non-deterministic

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

Heuristic

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

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

Software Design and Architecture

Topic#011

END!

Software Design and Architecture

Topic#012
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.

- It improves software quality.
- Good design exposes defects and makes it easier to test the software.
- 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.

Software Design and Architecture

Topic#012

END!

Software Design and Architecture

Topic#013
A Generic Design Process

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

1.

Understand the problem (software requirements)

2.

Construct a “black-box” model of solution (system specification).

System specifications are typically represented with use cases (especially when doing OOD).

3.

Look for existing solutions (e.g. architecture and design patterns) that cover some or all of the software design problems identified.

4.

Design not complete? Discover missing requirements

Consider building prototypes

5.
Document and review design

6.

Iterate over solution (Refactor) (Evolve the design until it meets functional requirements and maximizes non-functional requirements)

Software Design and Architecture

Topic#013

END!

Software Design and Architecture

Topic#014

Inputs to the design process

User requirements and system specification (including any constraints on design and implementation options)

Domain knowledge (For example, if it’s a healthcare application the designer will need some knowledge of healthcare terms and concepts.)

Implementation knowledge (capabilities and limitations of eventual execution environment)

Software Design and Architecture

Topic#014

END!

Software Design and Architecture

Topic#015

Desirable Internal Design Characteristics

© Virtual University of Pakistan

Poorly designed programs are difficult to understand and modify

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

Desirable Internal Design Characteristics

Minimal complexity

Keep it simple

Maybe you don’t need high levels of generality.

Extensibility

Design for today but with an eye toward the future.

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

Engineering is all about balancing conflicting objectives

Loose coupling

minimize dependencies between modules

Reusability

Reuse is a hallmark of a mature engineering discipline

Portability

works or can easily be made to work in other environments

Software Design and Architecture

Topic#015

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 016 – 020

Desirable Internal Design Characteristics

© Virtual University of Pakistan

Software Design and Architecture

Topic#016

What is good design?

What is good design?

- **Is it efficient code?**
- **compact implementation?**
- **Most maintainable?**

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**

High Cohesion and Low Coupling

Software Design and Architecture

Topic#016

END!

Software Design and Architecture

Topic#017
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)**

Why is high degree of coupling bad?

- **Highly coupled systems have strong interconnection**
- **High dependence of program units upon one another**
- **Difficult to test, reuse, and understand separately**

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

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

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**

Software Design and Architecture

Topic#017

END!

Software Design and Architecture

Topic#018
Coupling – Example
Design of a simple web app

```
<!doctype html>
<html>
<head>
     <script type="text/javascript" src="base.js">
     </script>
     <link rel="stylesheet" href="default.css">
</head>
<body>
     <button onclick="highlight2()">Highlight </button>
     <button onclick="normal2()">Normal </button>
     <h1 id="title" class="NormalClass">
          CSS <--> JavaScript Coupling
     </h1>
</body>
</html>
```

```
.NormalClass {
  color:inherit;
  font-style:normal;
}
```

default.css

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src="base.js">
    </script>
    <link rel="stylesheet" href="default.css">
</head>
<body>
    <button onclick="highlight2()">Highlight </button>
    <button onclick="normal2()">Normal </button>
    <h1 id="title" class="NormalClass">
        CSS <--> JavaScript Coupling
    </h1>
</body>
</html>
```

We want to change the style of the title in response to user action

```
function highlight() {
    document.getElementById("title").
                                    style.color="red";
    document.getElementById("title").
                                    style.fontStyle="italic";
}

function normal() {
    document.getElementById("title").
                                    style.color="inherit";
    document.getElementById("title").
                                    style.fontStyle="normal";
}
```

Option A

JavaScript code modifies the style attribute of HTML element.

```
function highlight() {
    document.getElementById("title").
                                    className = "HighlightClass";
}

function normal() {
    document.getElementById("title").
                                    className = "NormalClass";
}
```

Option B

JavaScript code modifies the class attribute of HTML element.

```
.NormalClass {
  color:inherit;
  font-style:normal;
}

.HighlightClass {
  color:red;
  font-style:italic;
}
```

default.css

Software Design and Architecture

Topic#018

END!

Software Design and Architecture

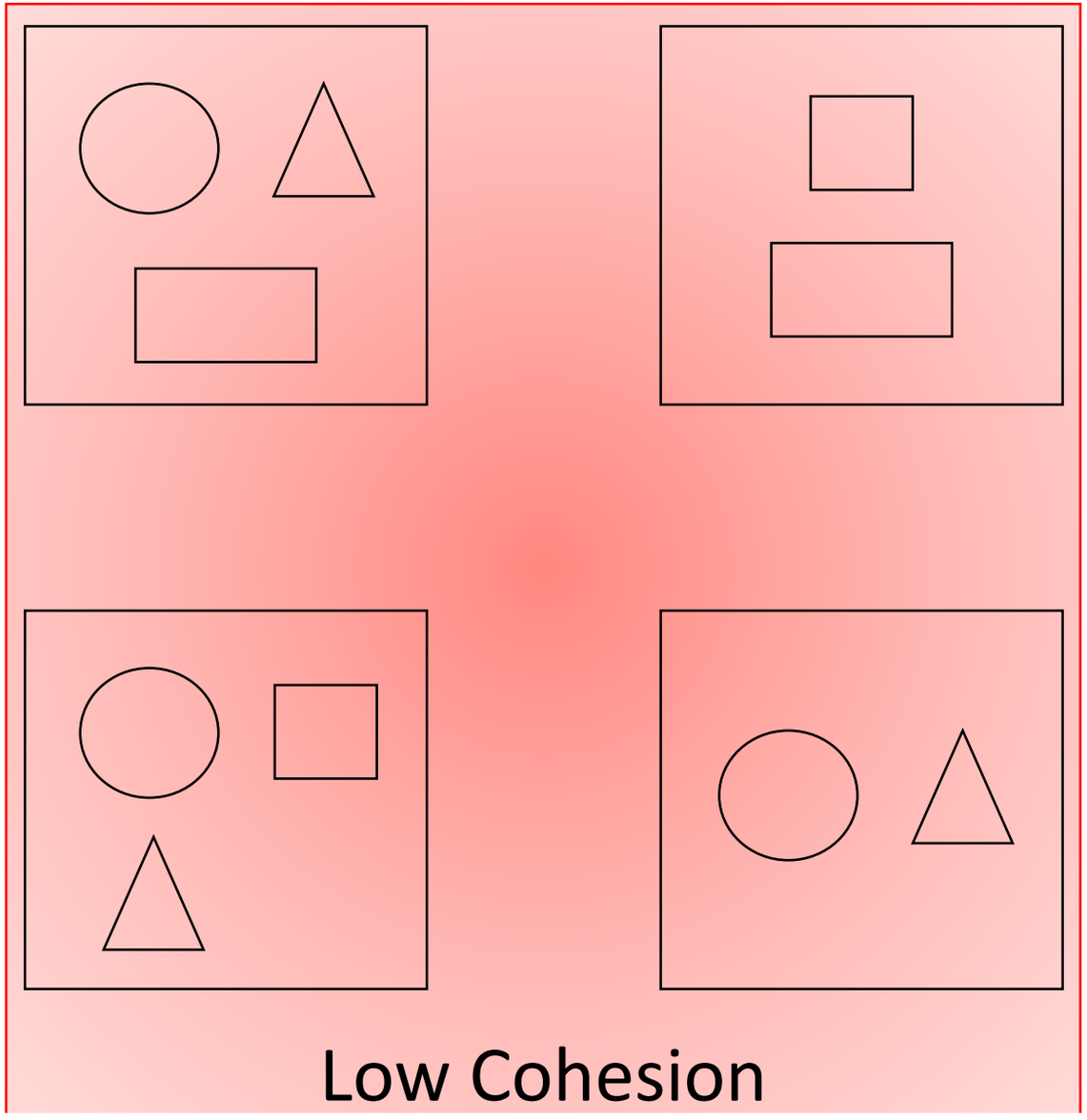
Topic#019
Cohesion

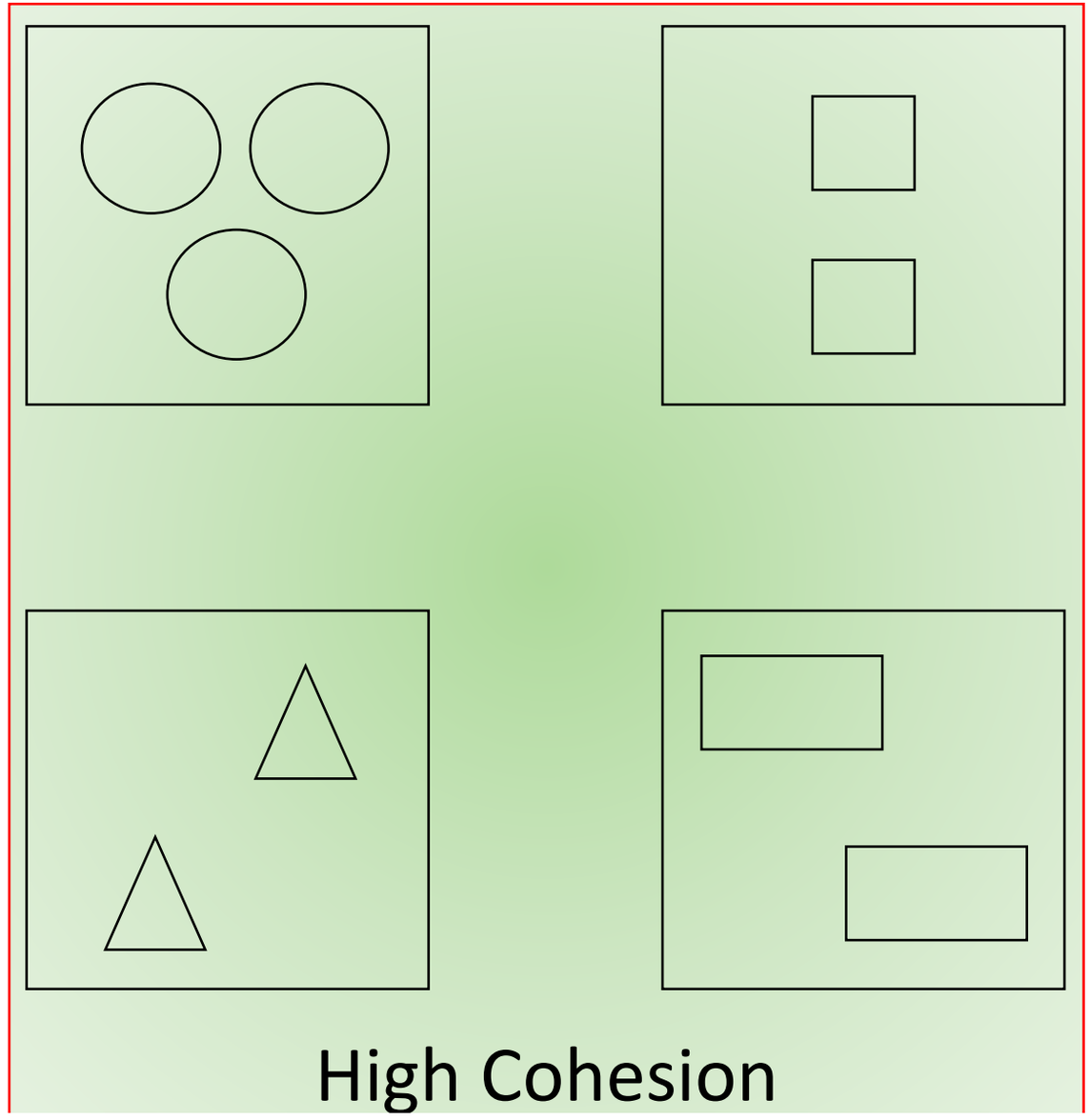
Cohesion

- **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

Cohesion: Intra-component relationship





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

© Virtual University of Pakistan

Software Design and Architecture

Topic#019

END!

Software Design and Architecture

Topic#020

Relationship between Coupling and Cohesion

High Cohesion and Low Coupling - Benefits

Highly Cohesive Modules

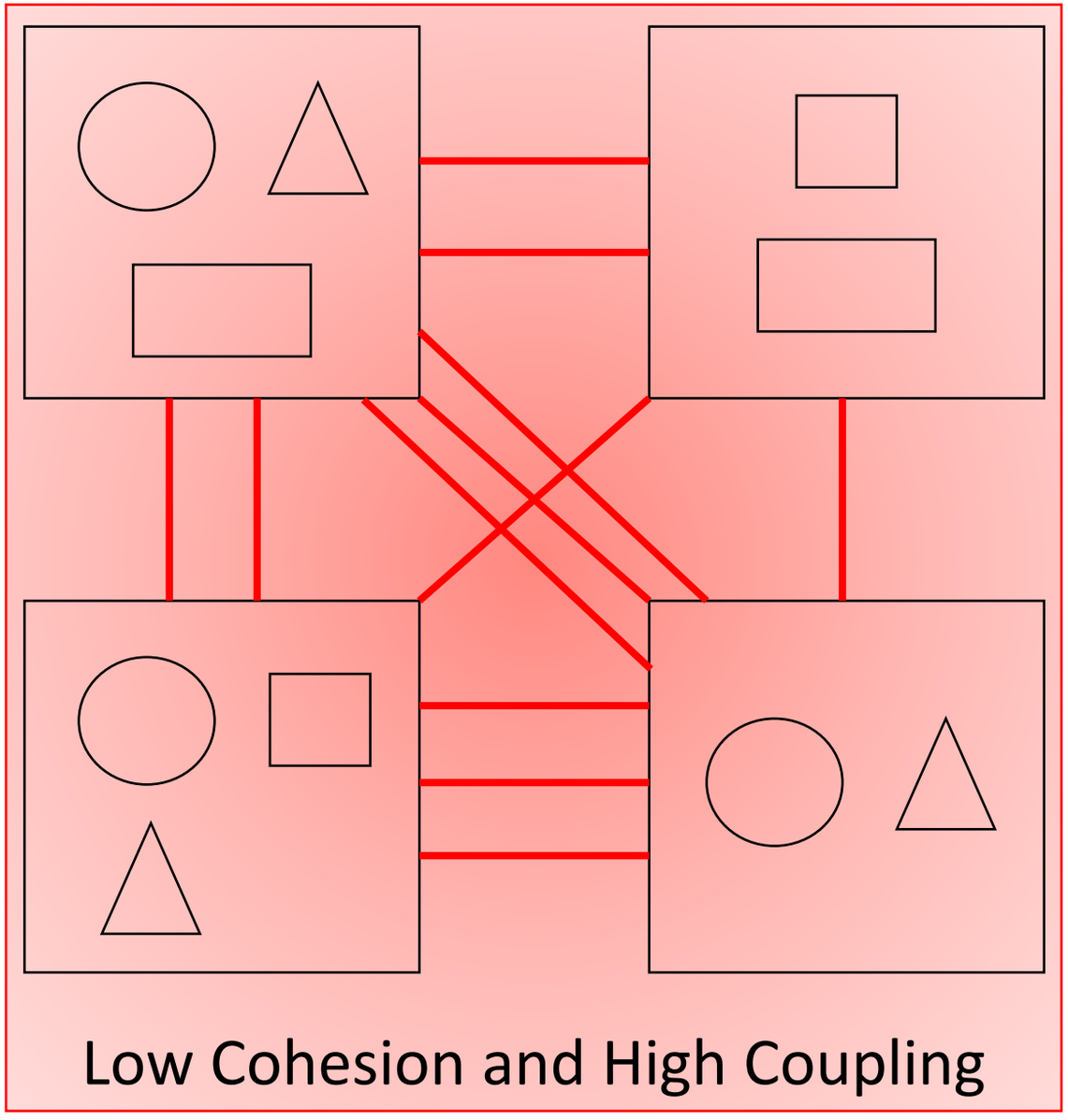
- easier to read and understand
- Easier to modify
- Increased potential for reuse
- Easier to develop and test

Lowly Coupled Modules

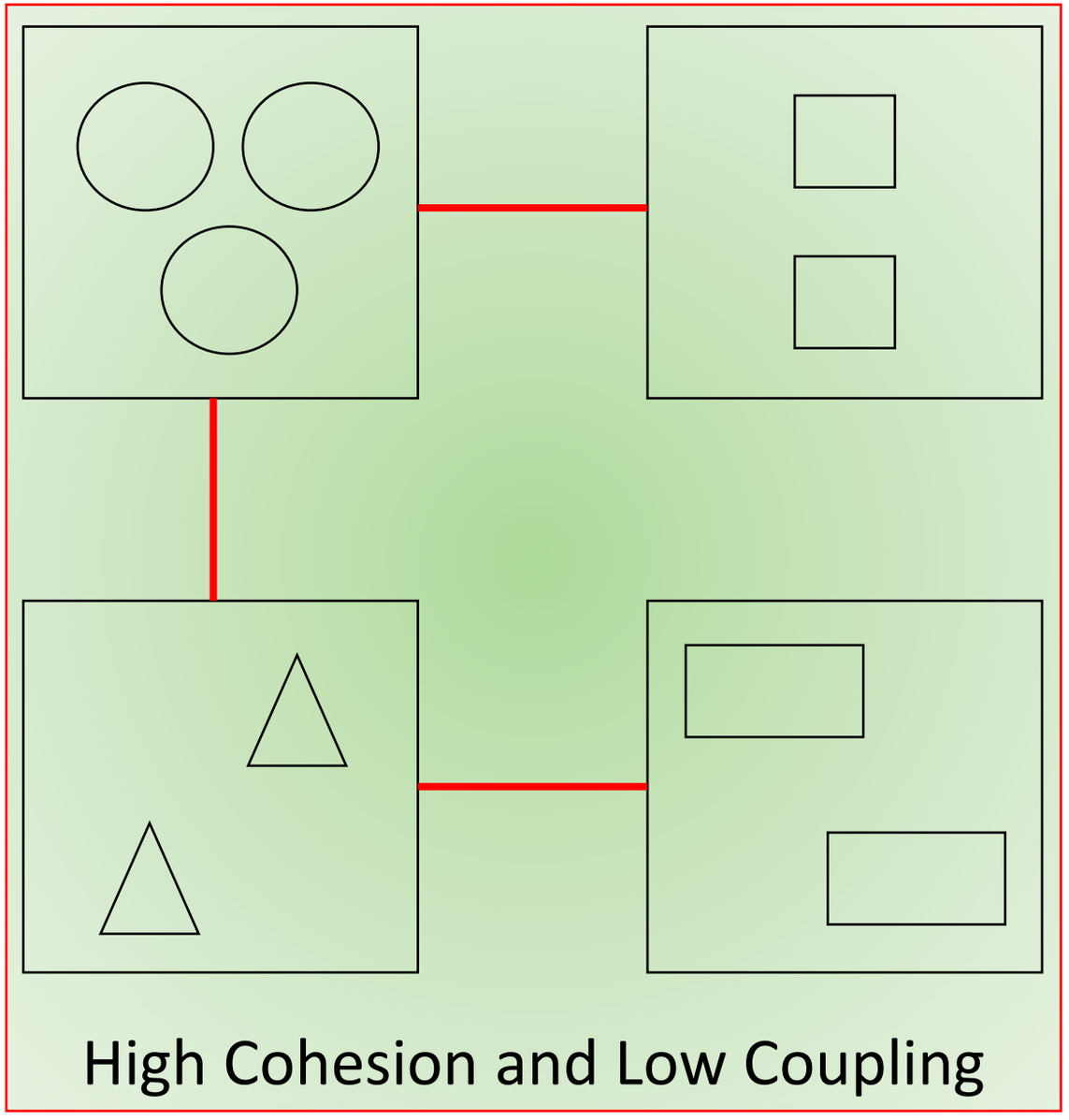
- easier to read and understand
- Easier to modify
- Increased potential for reuse
- Easier to develop and test

Surprised?

Are they somehow related?



Low Cohesion and High Coupling



High Cohesion and Low Coupling

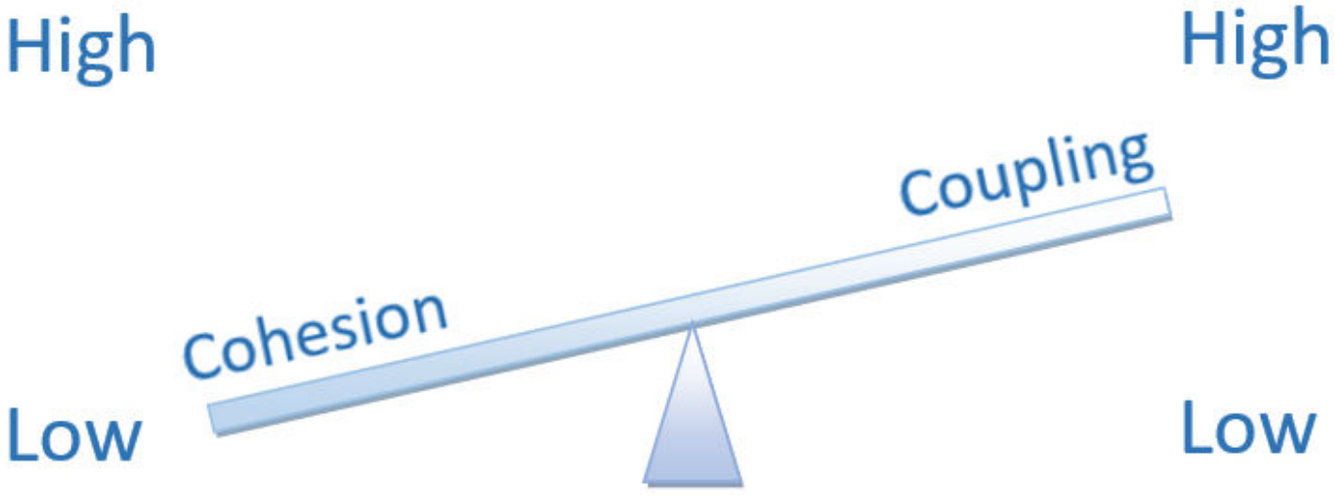
Cohesion and Coupling

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

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

Cohesion and Coupling

Tend to be Inversely Correlated



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

Software Design and Architecture

Topic#020

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 021 – 027

Object-Orientation – Introduction

© Virtual University of Pakistan

Software Design and Architecture

Topic#021
Software Design Strategies
Structured and Object-Oriented Design

© Virtual University of Pakistan

Software Design Strategies

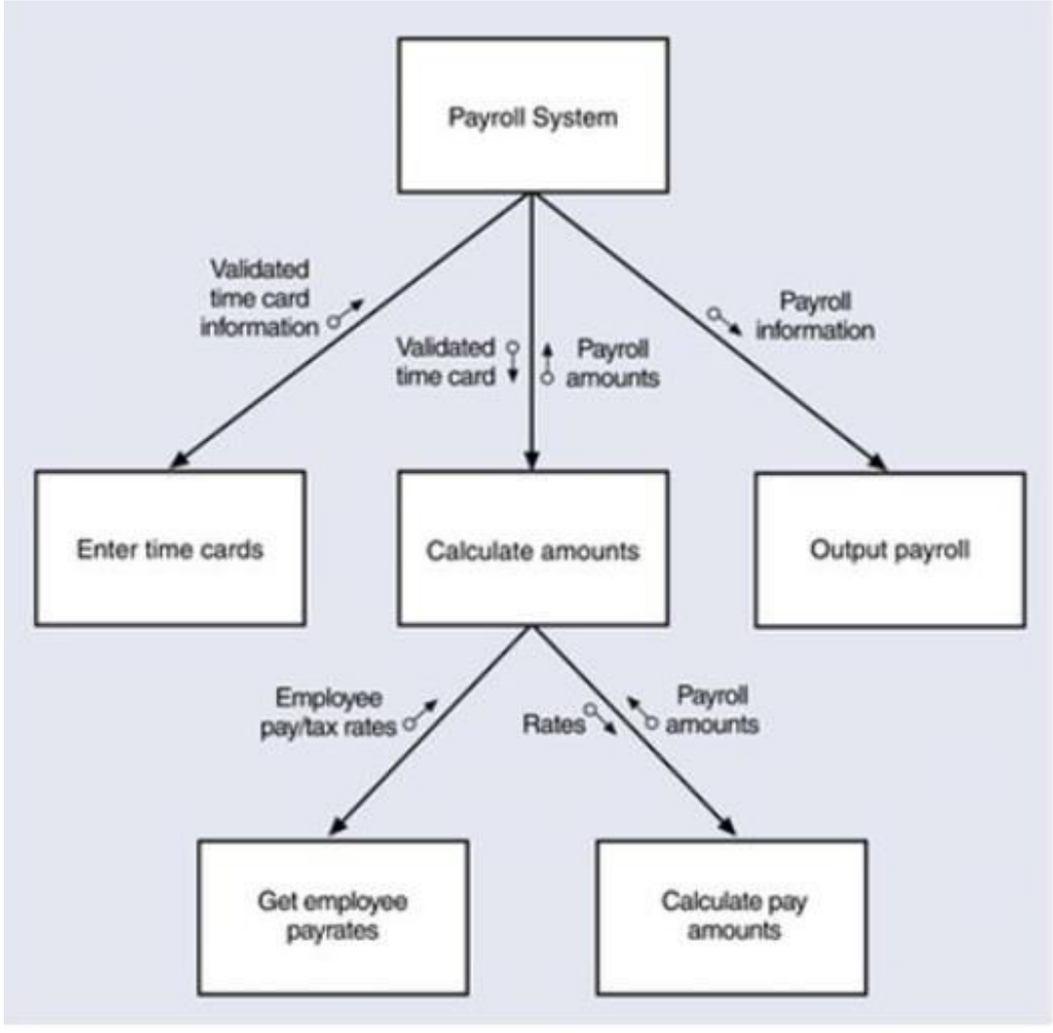
- **Structured Design**
- **Object-Oriented Design**

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**

Structured Design

- What the set of functions/programs should be?
- What each function should accomplish?
- How functions should be organized in a hierarchy?
- Start with a high level view and progressively refine this into more detailed design



Action-Oriented

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

Software Design and Architecture

Topic#021

END!

Software Design and Architecture

Topic#022

Object-Orientation Basic Concepts

The Object and The Class

The Object

- **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**

An object has state, behavior, and identity

© Virtual University of Pakistan

The State

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

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?

© Virtual University of Pakistan

Software Design and Architecture

Topic#022

END!

Software Design and Architecture

Topic#023

Object-Orientation Basic Concepts

The Class

The Class

- **Represents similar objects**
  - **Its instances**

Template of an object

Properties and Behavior

© Virtual University of Pakistan

The Class

- A class represents an abstraction
  - 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!

Software Design and Architecture

Topic#024

Object-Orientation Basic Concepts

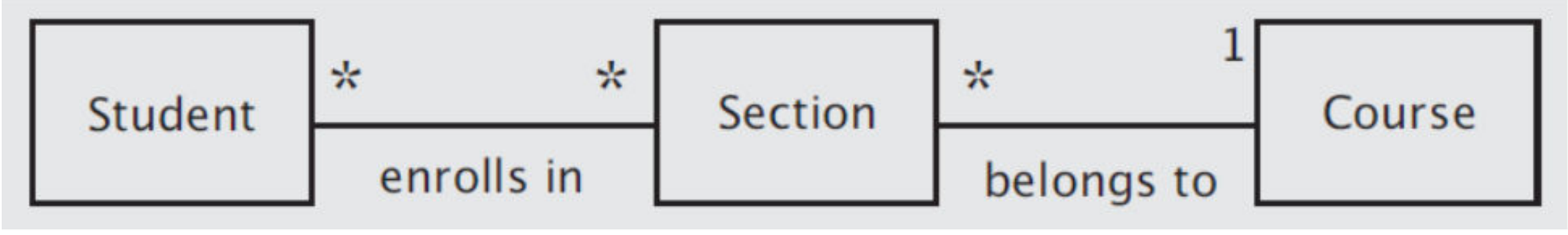
Relationship between Classes and Objects - Association

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

- **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.**



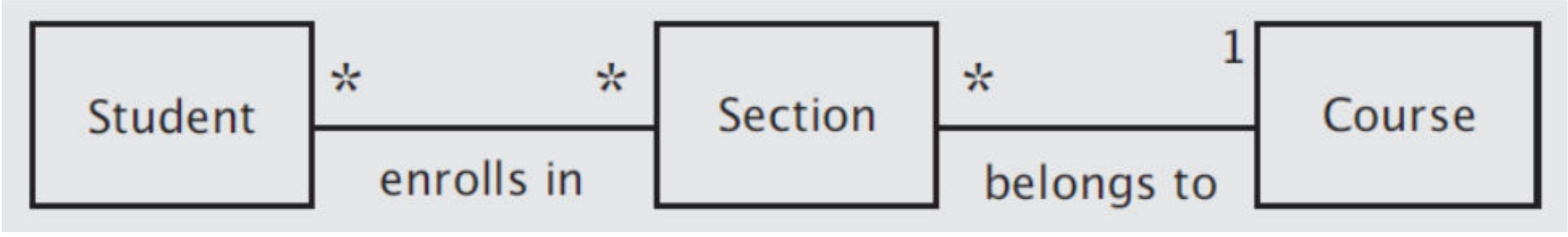
Association

- **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

Attributes of Associations

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



Software Design and Architecture

Topic#024

END!

Software Design and Architecture

Topic#025

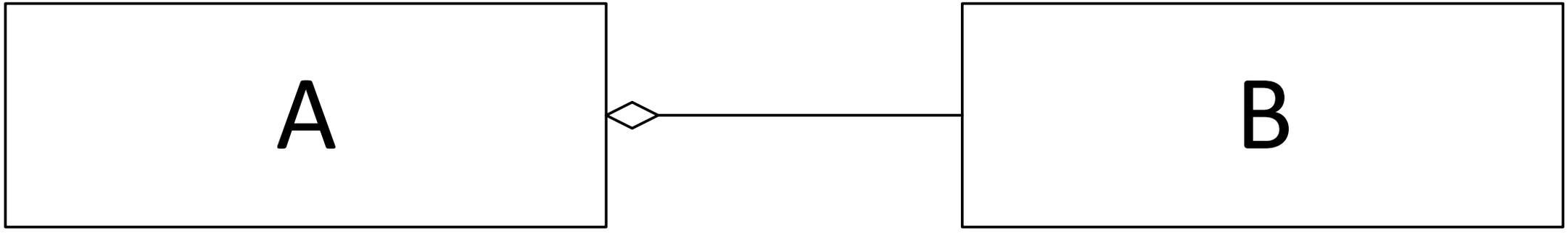
Object-Orientation Basic Concepts

Relationship between Classes and Objects

Aggregation and Composition

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**



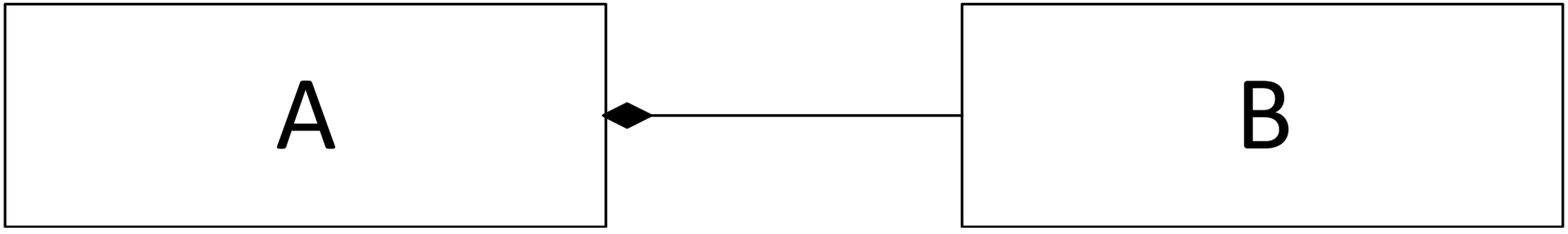
Aggregation

- **Most experts have downplayed the importance of general form of aggregation as not something that deserves to be embellished in any way**

Think of it as a modeling placebo!

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**



Software Design and Architecture

Topic#025

END!

Software Design and Architecture

Topic#026

Object-Orientation Basic Concepts

Relationship between Classes and Objects

Inheritance

© Virtual University of Pakistan

Classification and Hierarchy

- **Classification of objects into groups of related abstractions**

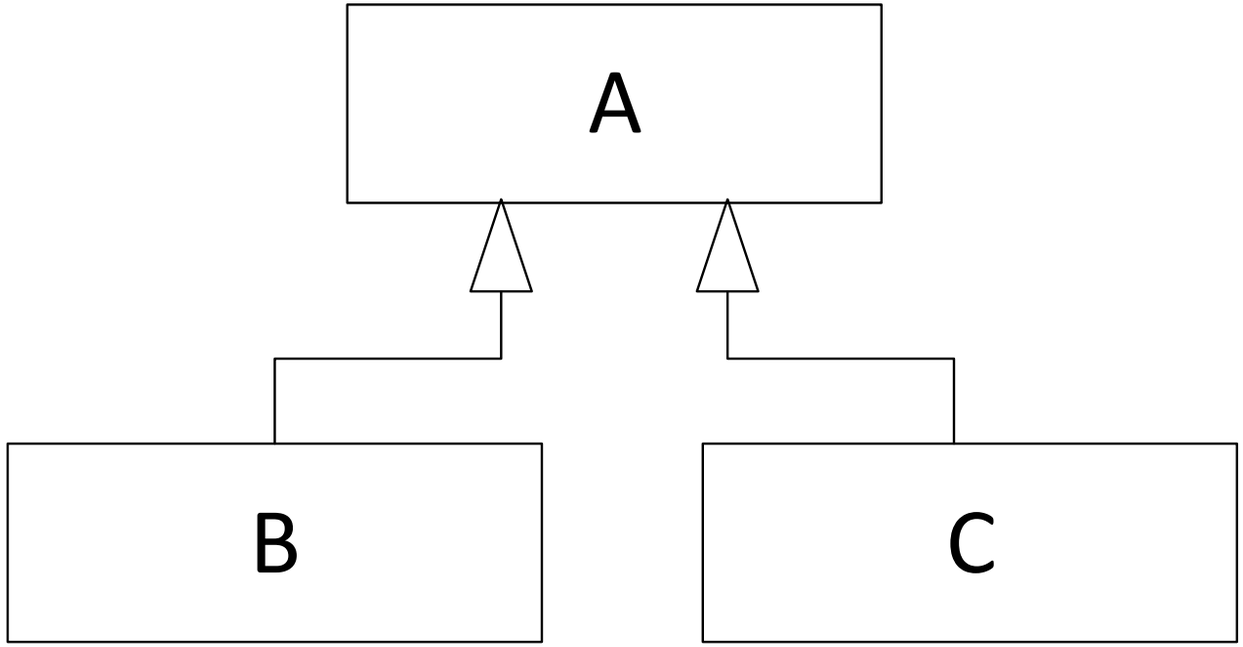
Inheritance

- **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**



© Virtual University of Pakistan

Inheritance

- **The implicit possession by all subclasses of features defined in its superclasses**
- **The implicit possession by all subclasses of features defined in its superclasses**

The Isa Rule

Software Design and Architecture

Topic#026

END!

Software Design and Architecture

Topic#027

Object-Orientation Basic Concepts

Abstract Classes and Interfaces

© Virtual University of Pakistan

Abstract Class

- **When we write a class, we code every field and method; in other words, the code is complete in a sense**
- **Sometimes, we might know the specifications for a class, but might not have the information needed to implement the class completely**
- **In such cases, we can implement a class partially using what are called abstract classes.**

Abstract Class

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

© Virtual University of Pakistan

Interface

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

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.

Software Design and Architecture

Topic#027

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 028 – 047

Implementing Relationships

© Virtual University of Pakistan

Software Design and Architecture

Topic#028
Relationships – Basic Concepts
Object Oriented Programming Defined

© Virtual University of Pakistan

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.

© Virtual University of Pakistan

thegradup.com

Relationships

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

Object and Class Relationships

End - Object Oriented Programming Defined

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#028

END!

Software Design and Architecture

Topic#029
Relationships – Basic Concepts
Association

Association

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

Association

- Associations are the “glue” that holds together a system.

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

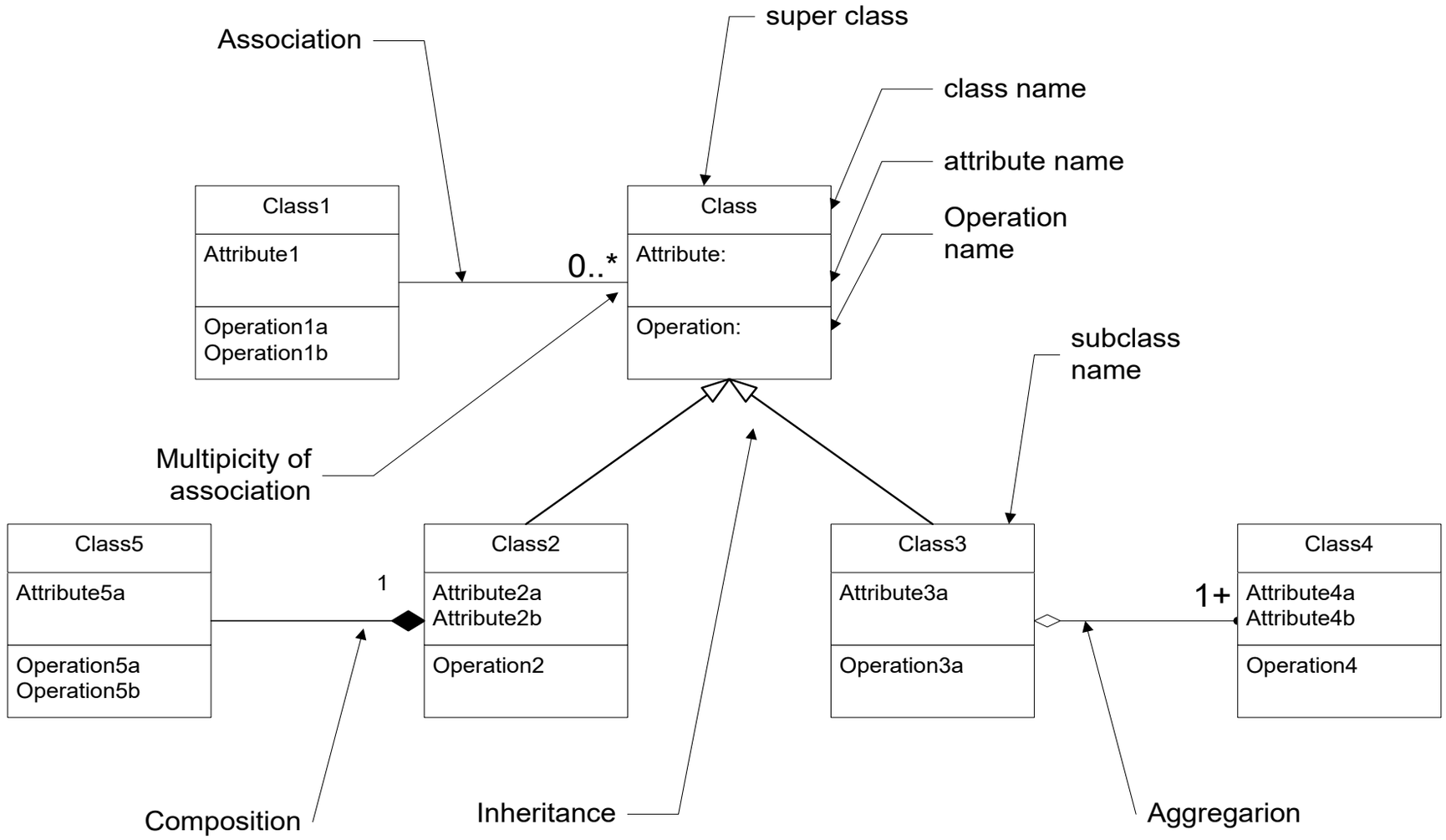
Types of Association

- Uses Relationship
  - Simple Association
- Whole-Part Relationship
  - Aggregation
  - Composition

© Virtual University of Pakistan

UML Object Model Notation

End - Association



Software Design and Architecture

Topic#029

END!

Software Design and Architecture

Topic#030
Relationships – Basic Concepts
Aggregation and Composition

Aggregation

A form of association that specifies a whole-part relationship between an aggregate (a whole) and a constituent part.

Aggregation

The distinction between aggregation and association is often a matter of taste rather than a difference in semantics.

Think of it as a modeling placebo

Aggregation

"Aggregation is strictly meaningless; as a result, I recommend that you ignore it in your own diagrams"

UML Distilled by Martin Fowler

Composition

A form of aggregation association with strong ownership and coincident lifetime of parts by the whole

Composition

A form of aggregation association with strong ownership and coincident lifetime of parts by the whole

End – Aggregation and Composition

Software Design and Architecture

Topic#030

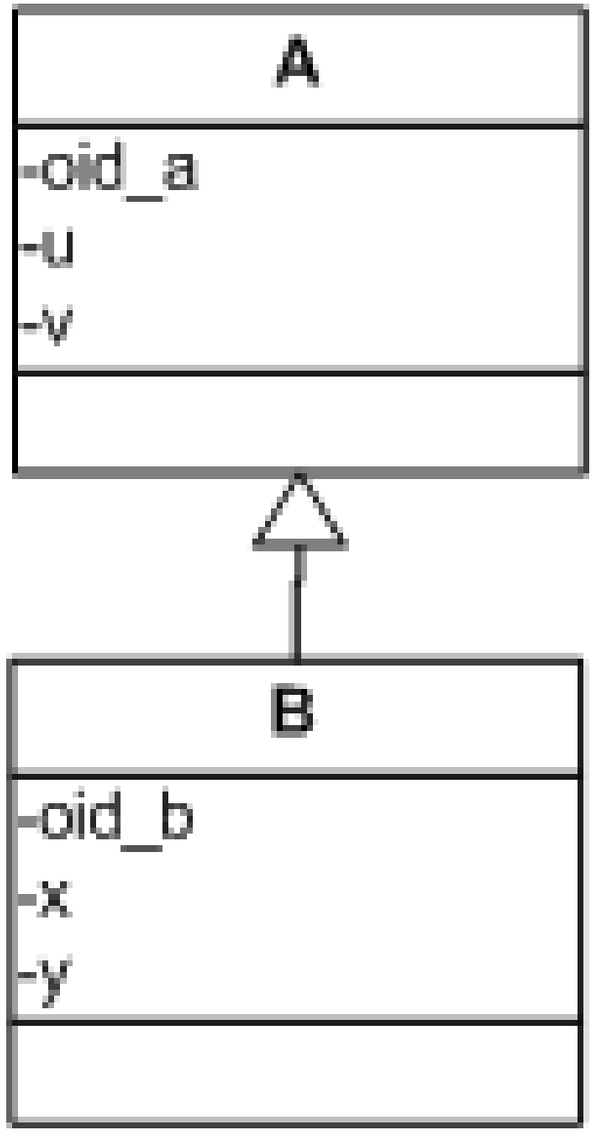
END!

Software Design and Architecture

Topic#031
Object Lifetime and Visibility
Inheritance

Inheritance

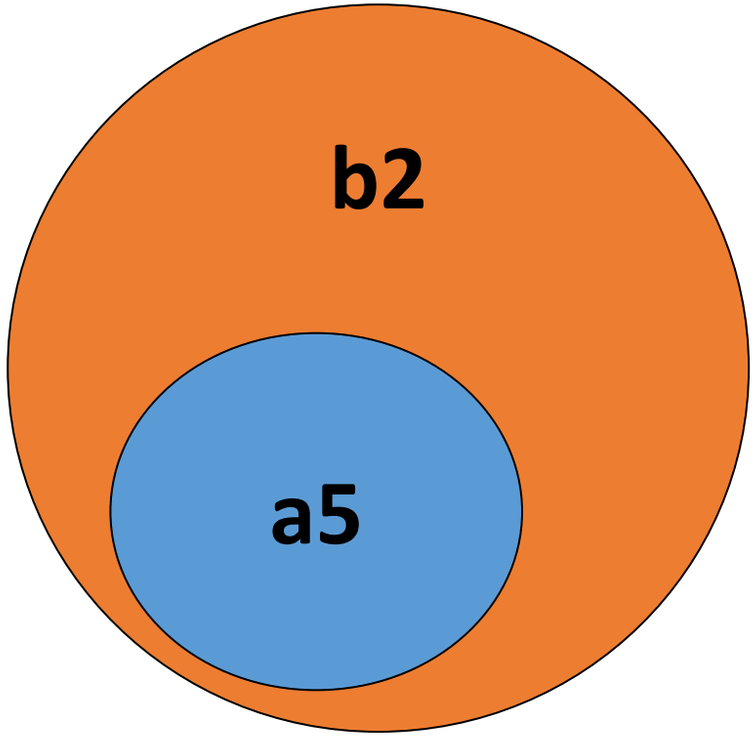
- B inherits from A
- A is the super or base class and B is the sub or derived class
- A does not know about B



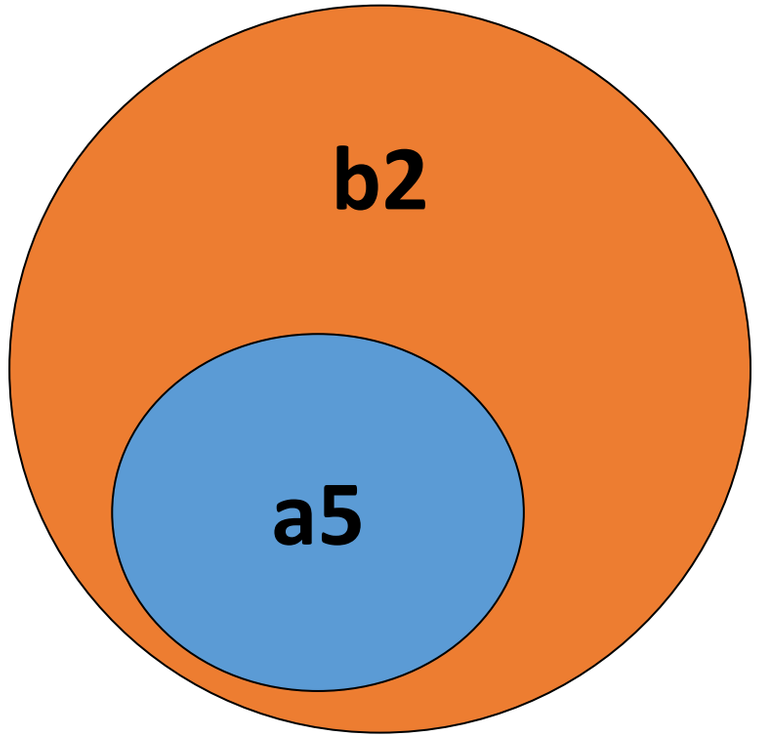
- When an object of Type B is created, a corresponding object of Type A is also created

- Conceptually speaking, the object of Type B holds the corresponding object of Type A as a private component.

For example, when an object, say **b2**, of Type B is created, another object, say **a5**, of Type A is also created.



- a5 is only accessible from b2.
  - **exclusive holding**
- The life of a5 is dependent upon b2.
  - **a5 is created with b2 and it dies with b2.**
- Cardinality of this relationship is **1:1**



End of topic

© Virtual University of Pakistan

Software Design and Architecture

Topic#031

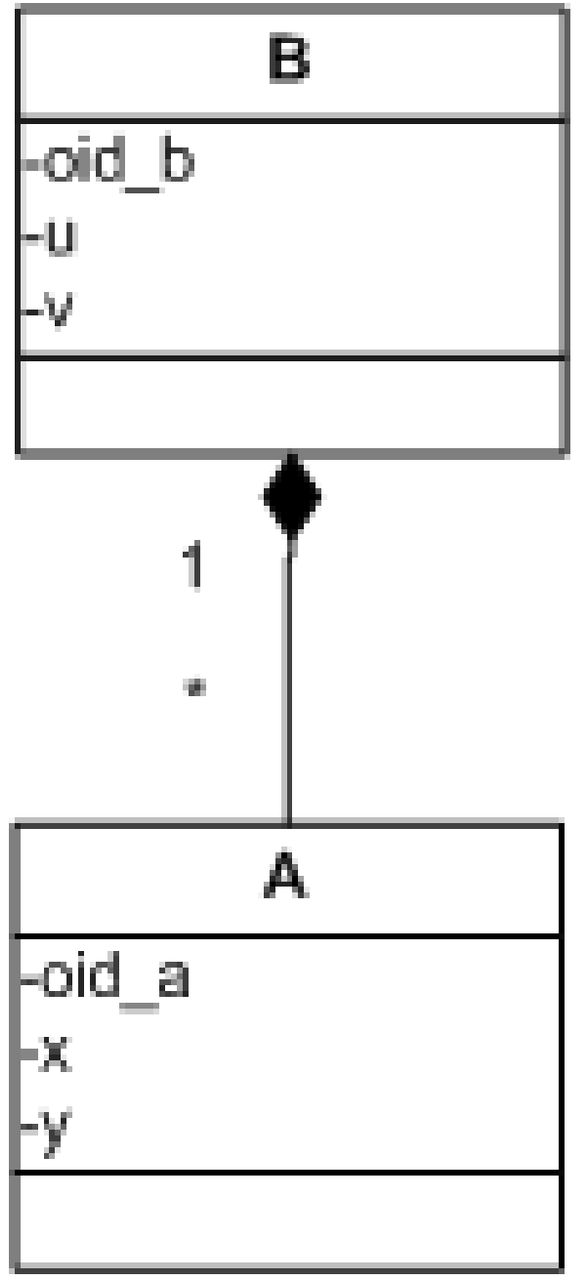
END!

Software Design and Architecture

Topic#032
Object Lifetime and Visibility
Composition

Composition

- Whole-part relationship
- B has many instances of A
- B is the container or whole, A is the part.
- A does not know about B



Composition

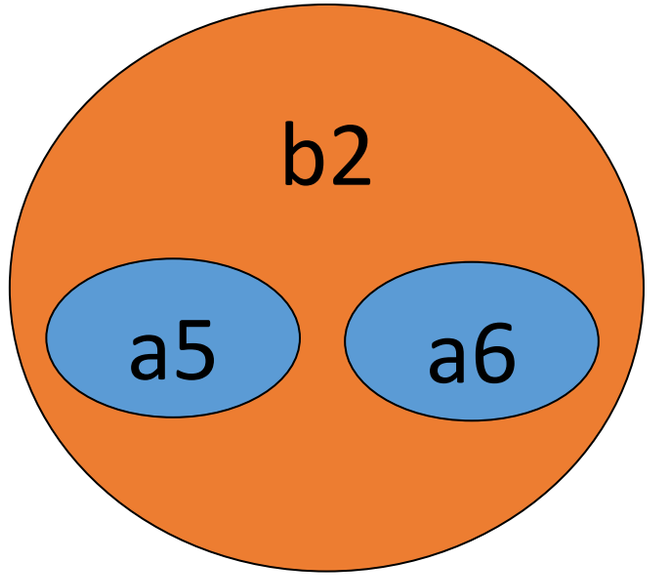
- A form of aggregation association with strong ownership and coincident lifetime of parts by the whole.
- A part may belong to only one composite.

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.

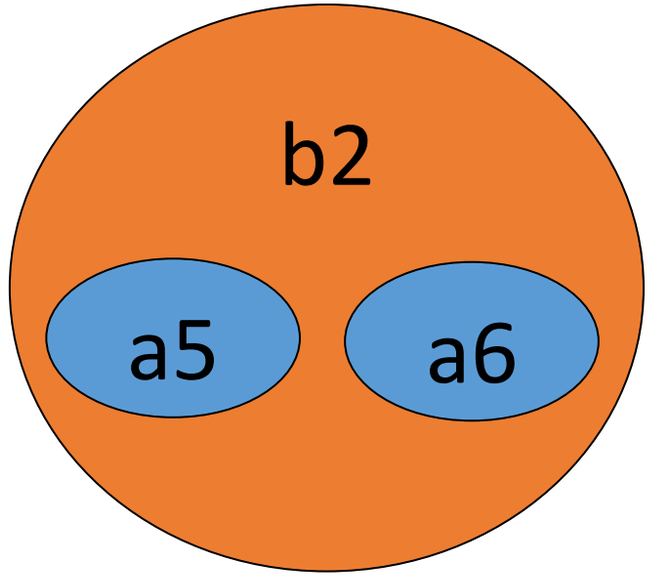
Composition – object creation and life-time

- When an object of Type B is created, corresponding instances of Type A are also created
- Conceptually speaking, the object of Type B holds the corresponding objects of Type A as private components.
  - For example, when an object, say b2, of Type B is created, some objects of Type A, say a5 and a6, are also created



Composition – object creation and life-time

- a5 and a6 are only accessible from b2.
  - exclusive holding
- The life of a5 and a6 is dependent upon b2.
  - a5 and a6 are created with b2 and destroyed with b2.
- Cardinality of this relationship could be 1:1 or 1:m



End of topic

Software Design and Architecture

Topic#032

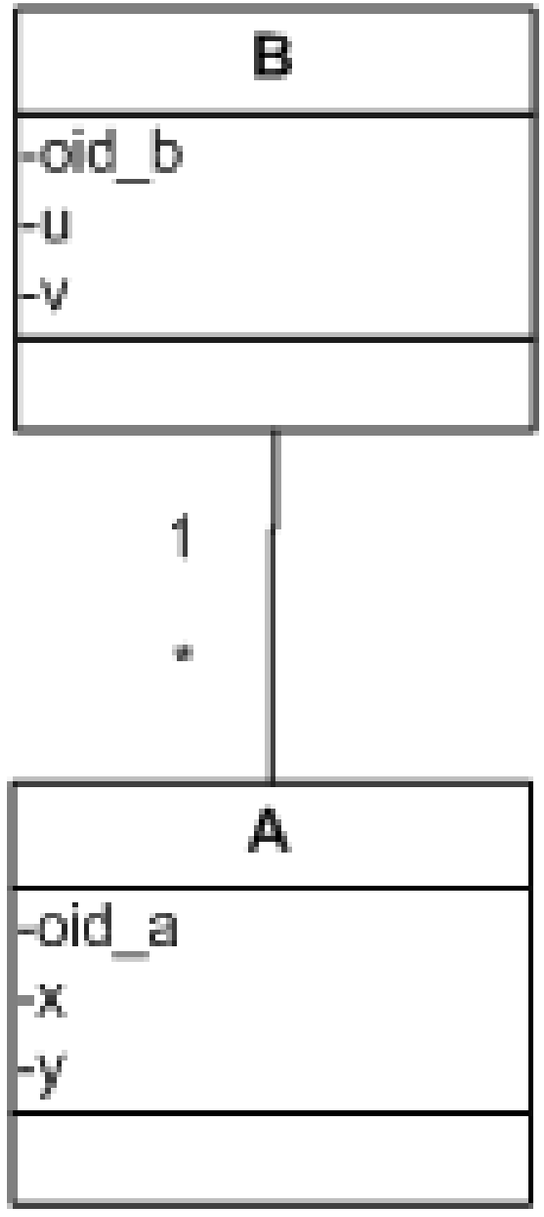
END!

Software Design and Architecture

Topic#033
Object Lifetime and Visibility
Association

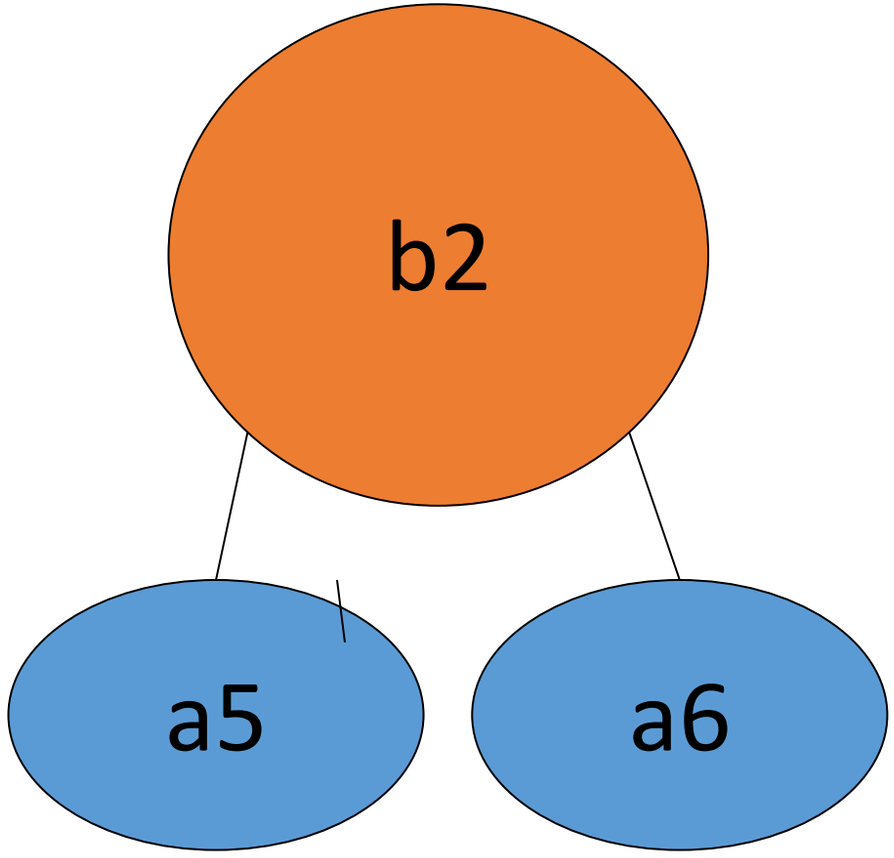
Association – some sort of link between objects

- One instance of B is associated with many instances of A



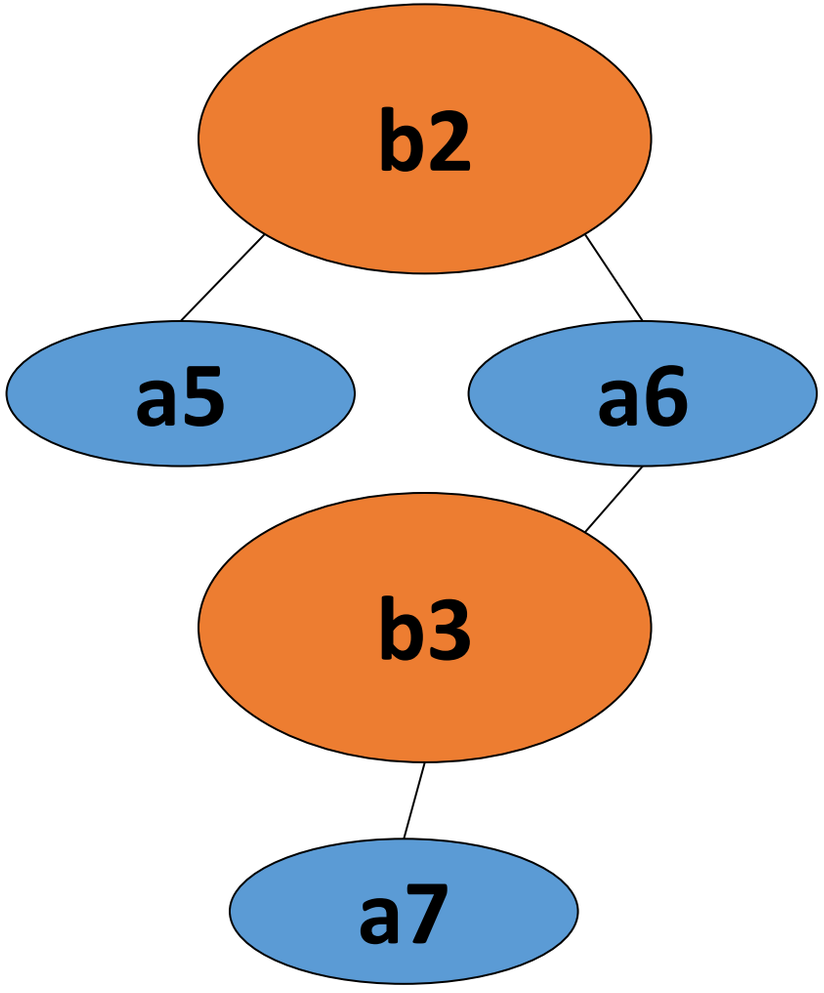
Association – object creation and life-time

- **Objects of the two types have independent life times.**
- **Conceptually speaking, objects of Type B are linked with objects of Type A.**



Association – object creation and life-time

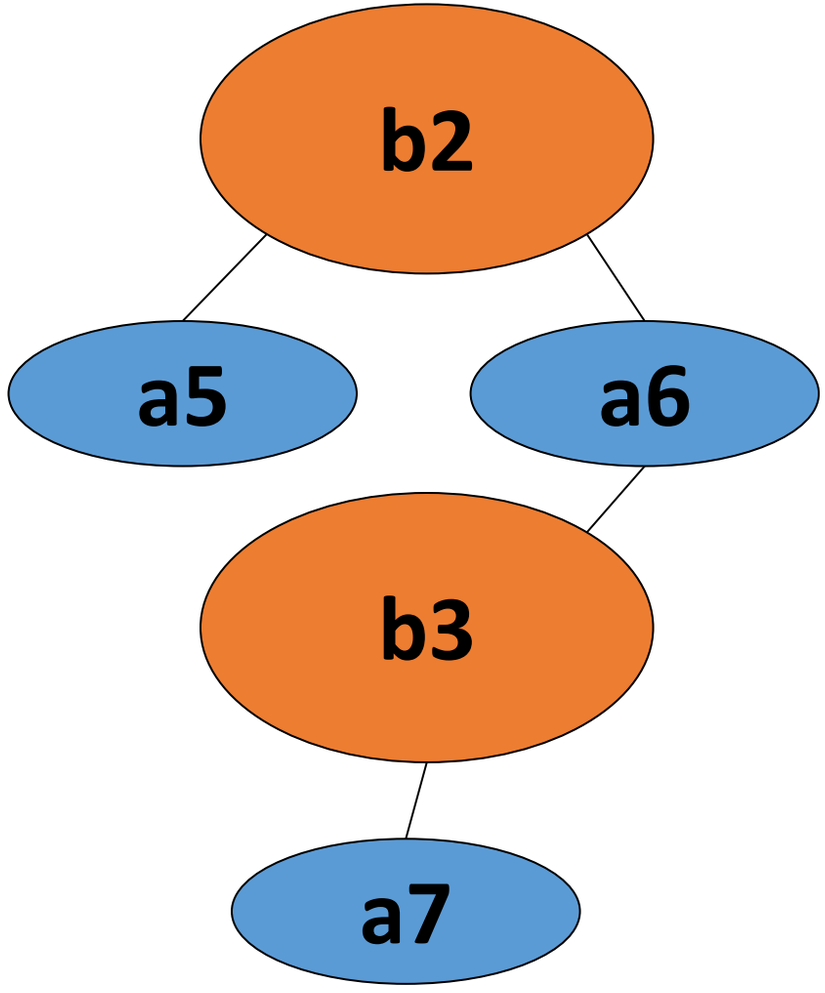
- **The relationship is not exclusive. That is, the same instance of A may also be linked and accessed directly by other objects.**



© Virtual University of Pakistan

Association – object creation and life-time

- **Cardinality of this relationship could be 1:1, 1:m, or m:m**



End of topic

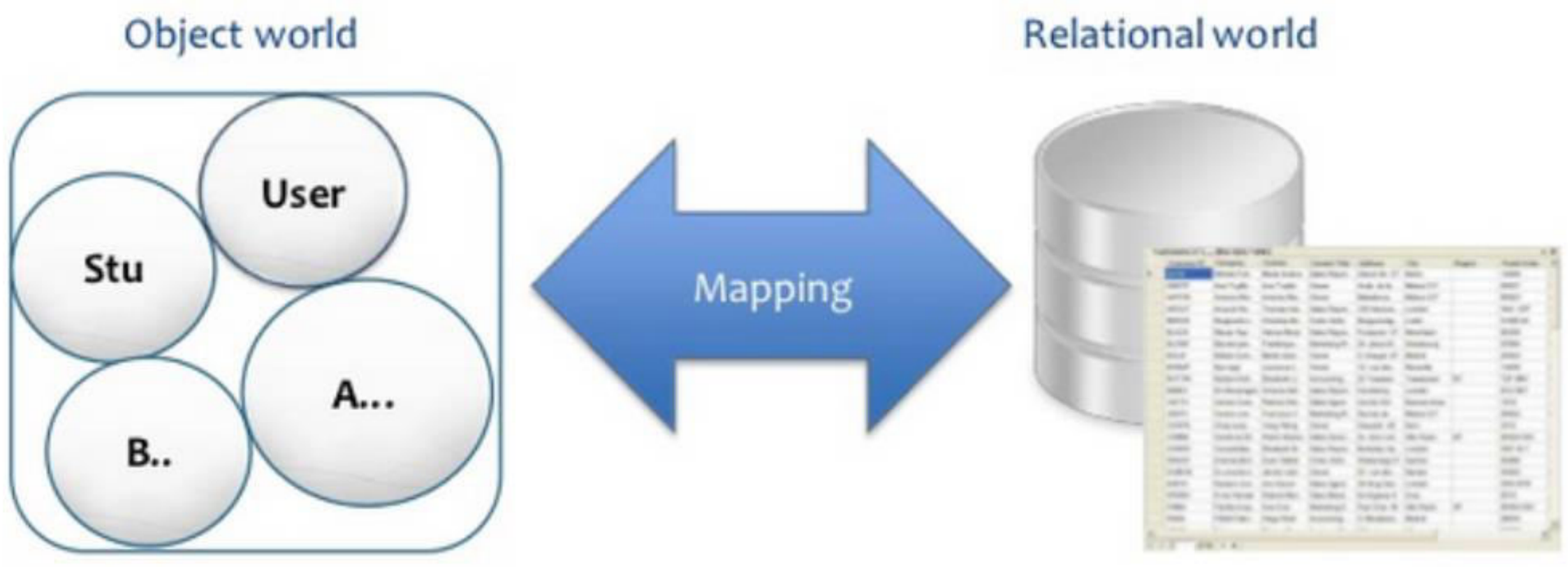
Software Design and Architecture

Topic#033

END!

Software Design and Architecture

Topic#034
Object to Relational Mapping Basics



- Relational Database
- Objects
- Mapping between Objects & Relational Database

O2R Mapping

- aka – **O2R**, **ORM**, **O/RM**, and **O/R**

- Mapping attributes
- Mapping Inheritance
- Mapping Association
- Mapping Composition

End of topic

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#034

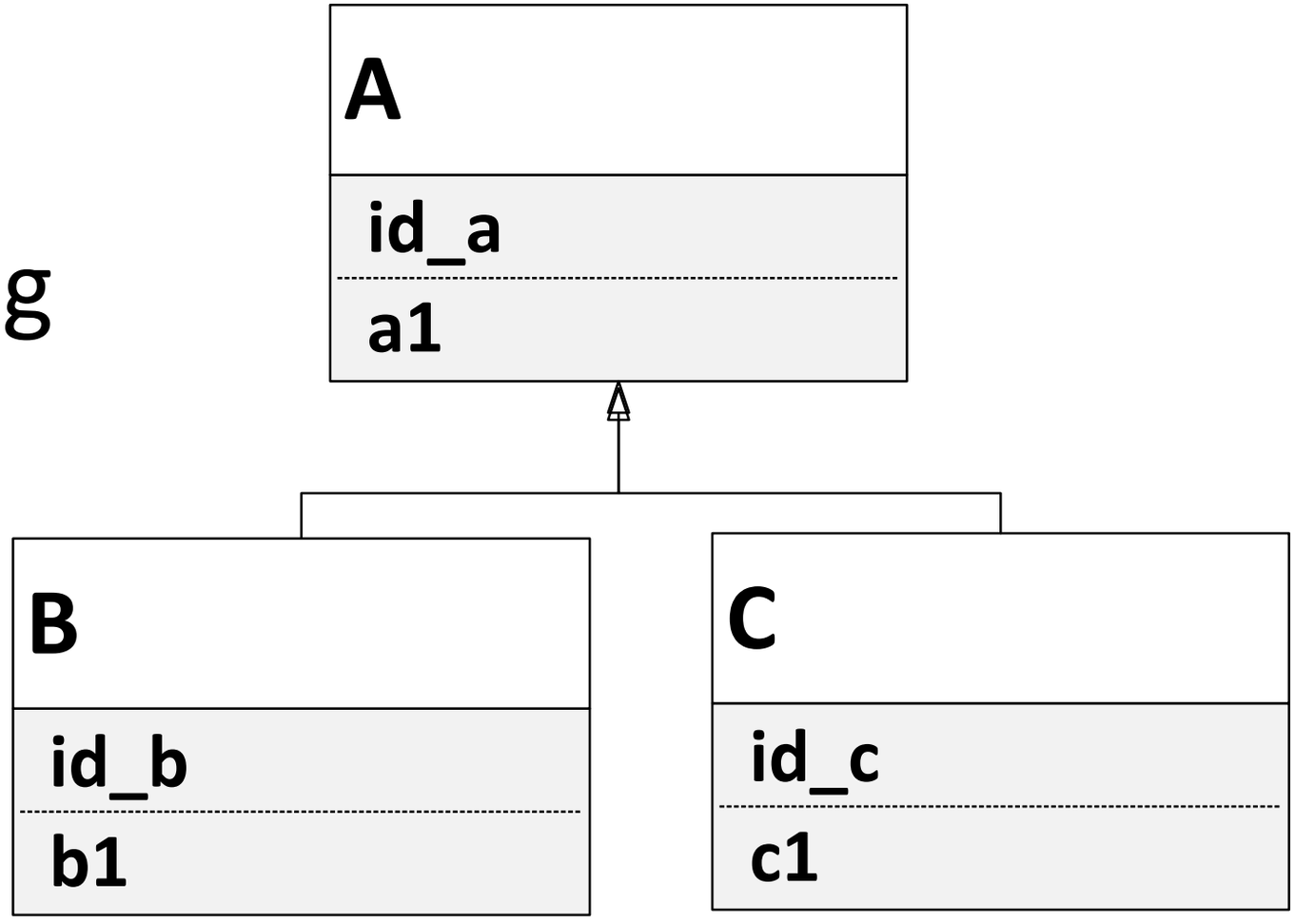
END!

Software Design and Architecture

Topic#035
Object to Relational Mapping
Mapping Inheritance

Mapping Inheritance

- Filtered Mapping
- Horizontal Mapping
- Vertical Mapping



© Virtual University of Pakistan

thegradup.com

Filtered Mapping

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

TableName
a_id
a1
b_id
b1
c_id
c1

© Virtual University of Pakistan

thegradup.com

Horizontal Mapping

- 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

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#035

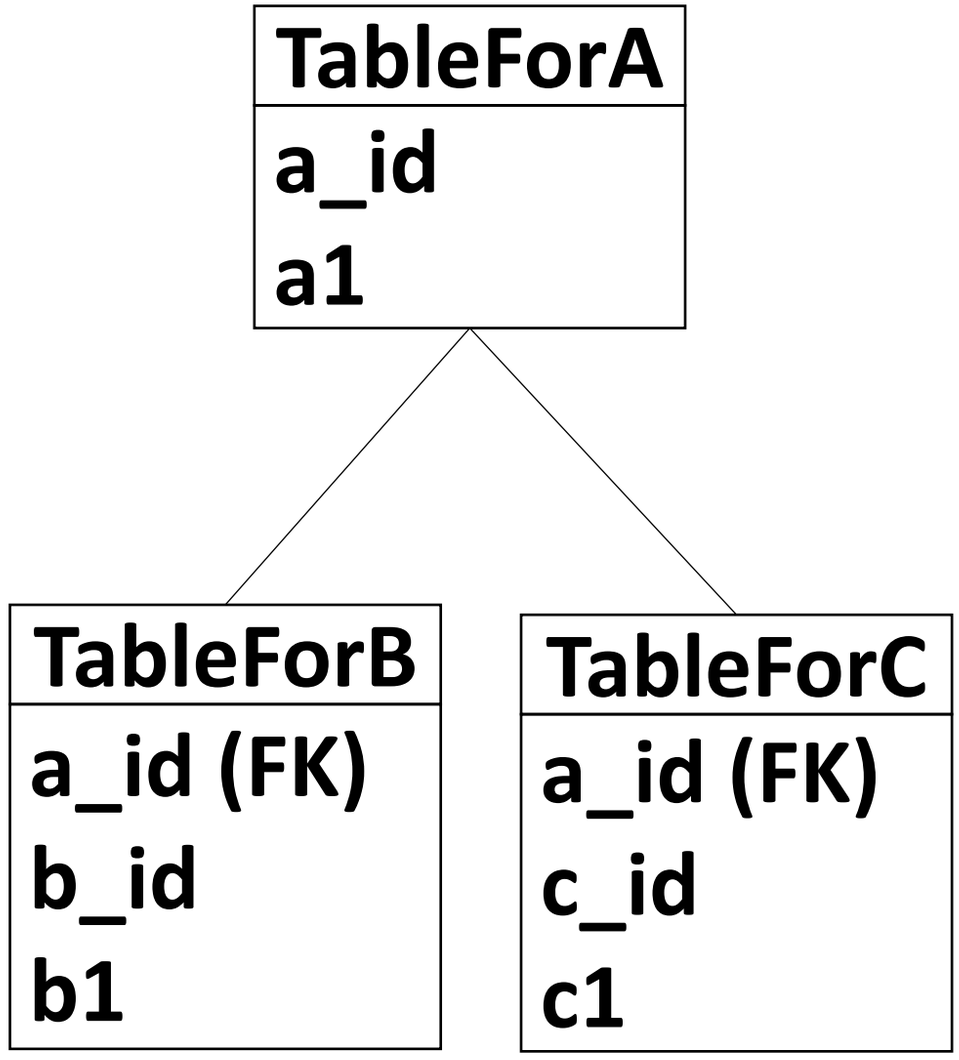
END!

Software Design and Architecture

Topic#036
Object to Relational Mapping
Mapping Inheritance – Vertical Mapping

Vertical Mapping

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

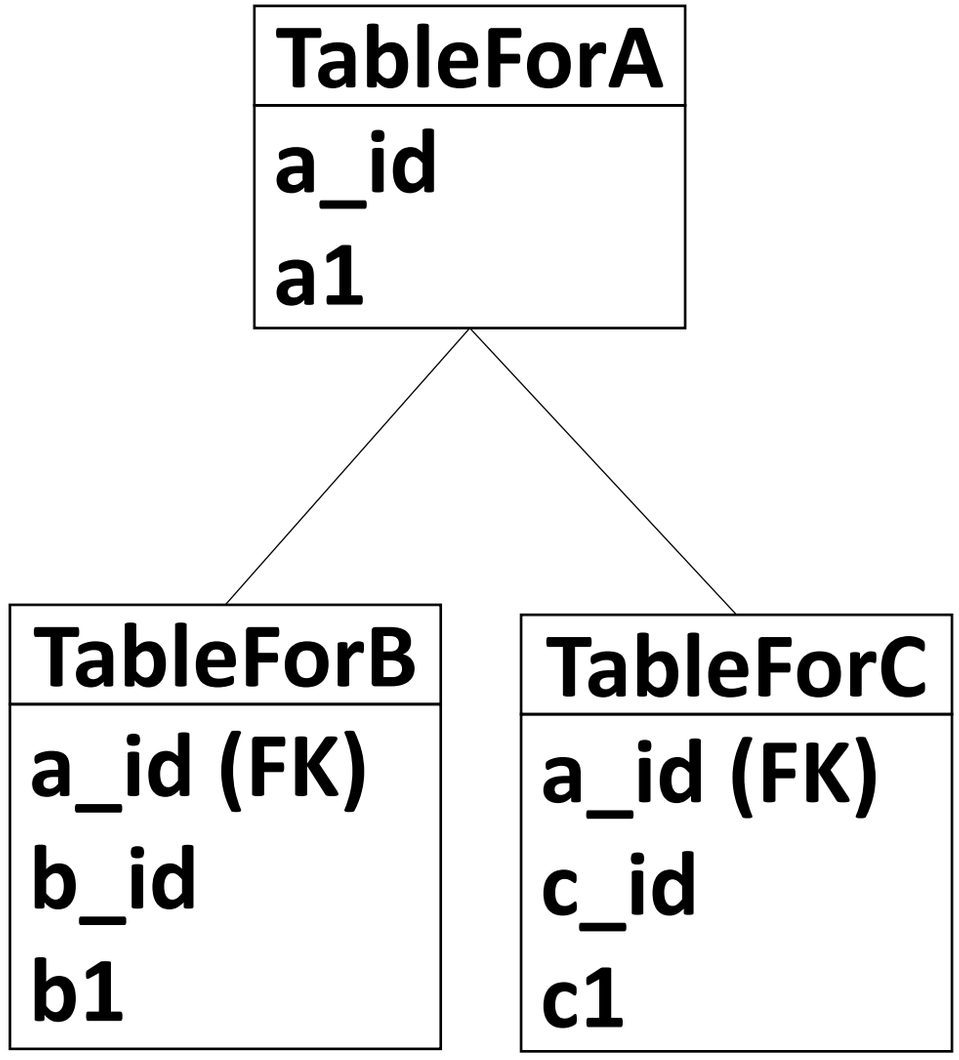


© Virtual University of Pakistan

thegradup.com

Vertical Mapping

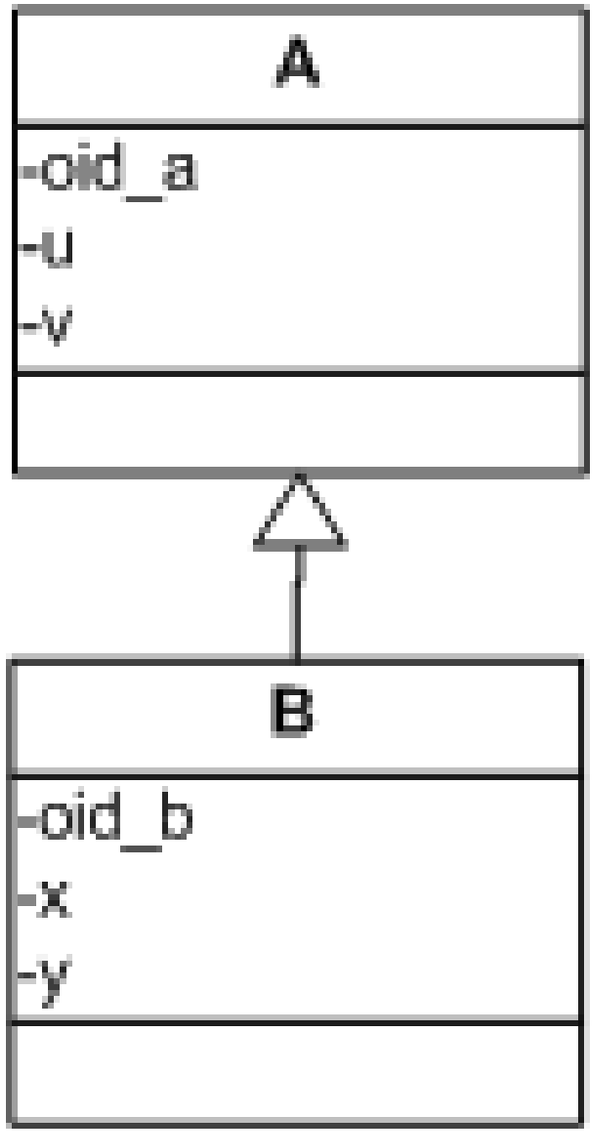
- To maintain the inheritance relationship between parent and child classes, OID of the parent class is inserted in the child classes as a foreign key.



© Virtual University of Pakistan

thegradup.com

Vertical Mapping Example



Vertical Mapping Example

oid_a	u	v
10	23	10
21	34	54
35	78	17

oid_b	oid_a(FK)	x	y
1	21	567	100
2	10	98	43
3	35	718	127

- For example, object of type B with oid_b = 3 contains object of type A with oid_a = 35

End of topic

© Virtual University of Pakistan

Software Design and Architecture

Topic#036

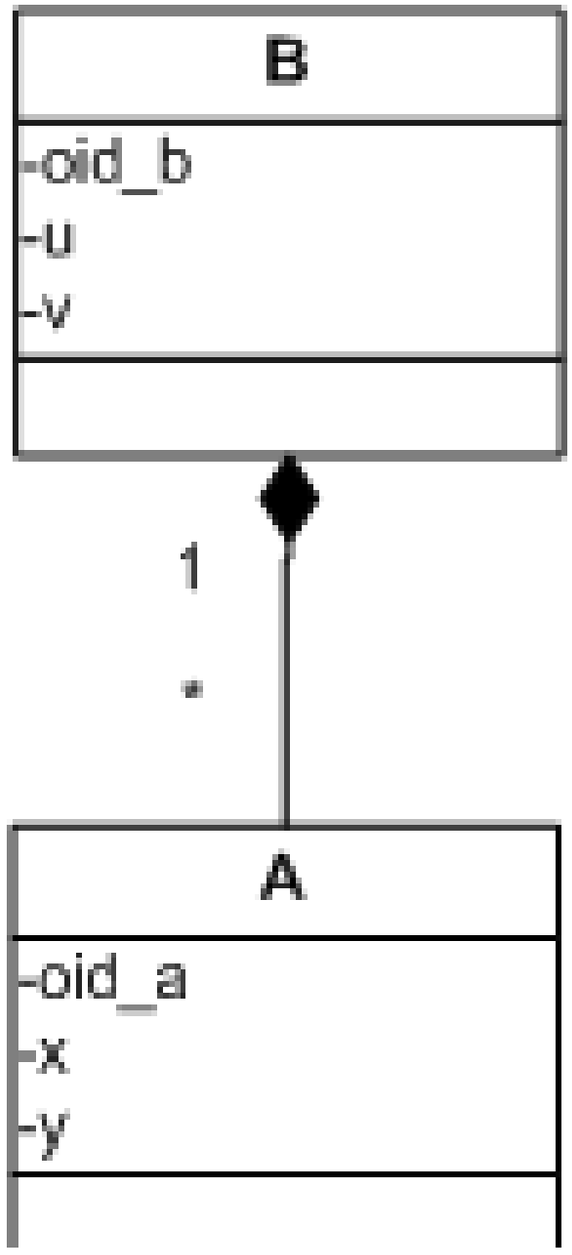
END!

Software Design and Architecture

Topic#037
Object to Relational Mapping
Mapping Composition

- each class of the whole-part hierarchy is mapped to a separate table.
- To maintain the whole-part relationshipbetween the whole (composite class) and the part (component class), primary key (OID) of the composite class is inserted in the part classes as a foreign key.
  - note the difference between composition and inheritance

Composition – Persistence - Example



Composition – Persistence - Example

oid_b	u	v
10	23	10
21	34	54
35	78	17

oid_a	oid_b(FK)	x	y
1	21	567	100
2	21	98	43
3	35	718	127

- For example, object of type B with oid_b = 21 contains objects of type A with oid_a = 1 and oid_a = 2

End of topic

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#037

END!

Software Design and Architecture

Topic#038
Object to Relational Mapping
Mapping Association

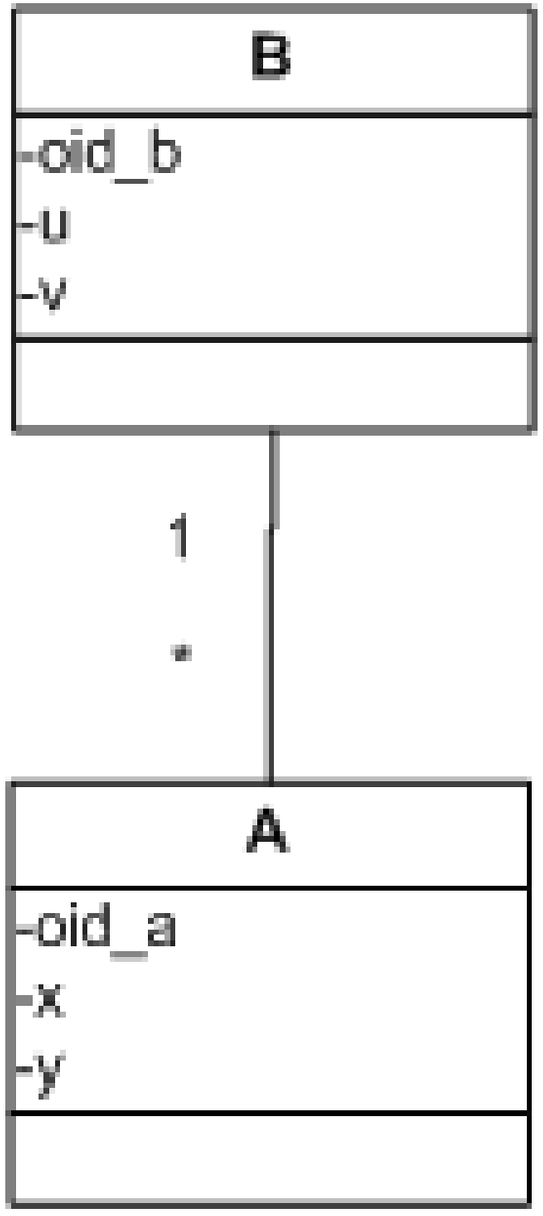
- Each class involved in the relationship is mapped to a separate table.
- To maintain the relationshipbetween the classes, primary key (OID) of the **user class** is inserted in the classes that have been used as a foreign key.

© Virtual University of Pakistan

- Note that, in the OO world, the client knows the identity of the server whereas in the relational world, the table for server holds the identity of the client.
- For many-t-many relationship a separate table is used to maintain the information about the relationship.

© Virtual University of Pakistan

Association – Persistence



Composition – Persistence - Example

oid_b	u	v
10	23	10
21	34	54
35	78	17

oid_a	oid_b(FK)	x	y
1	21	567	100
2	21	98	43
3	35	718	127

- For example, object of type B with oid_b = 21 uses objects of type A with oid_a = 1 and oid_a = 2

End of topic

© Virtual University of Pakistan

Software Design and Architecture

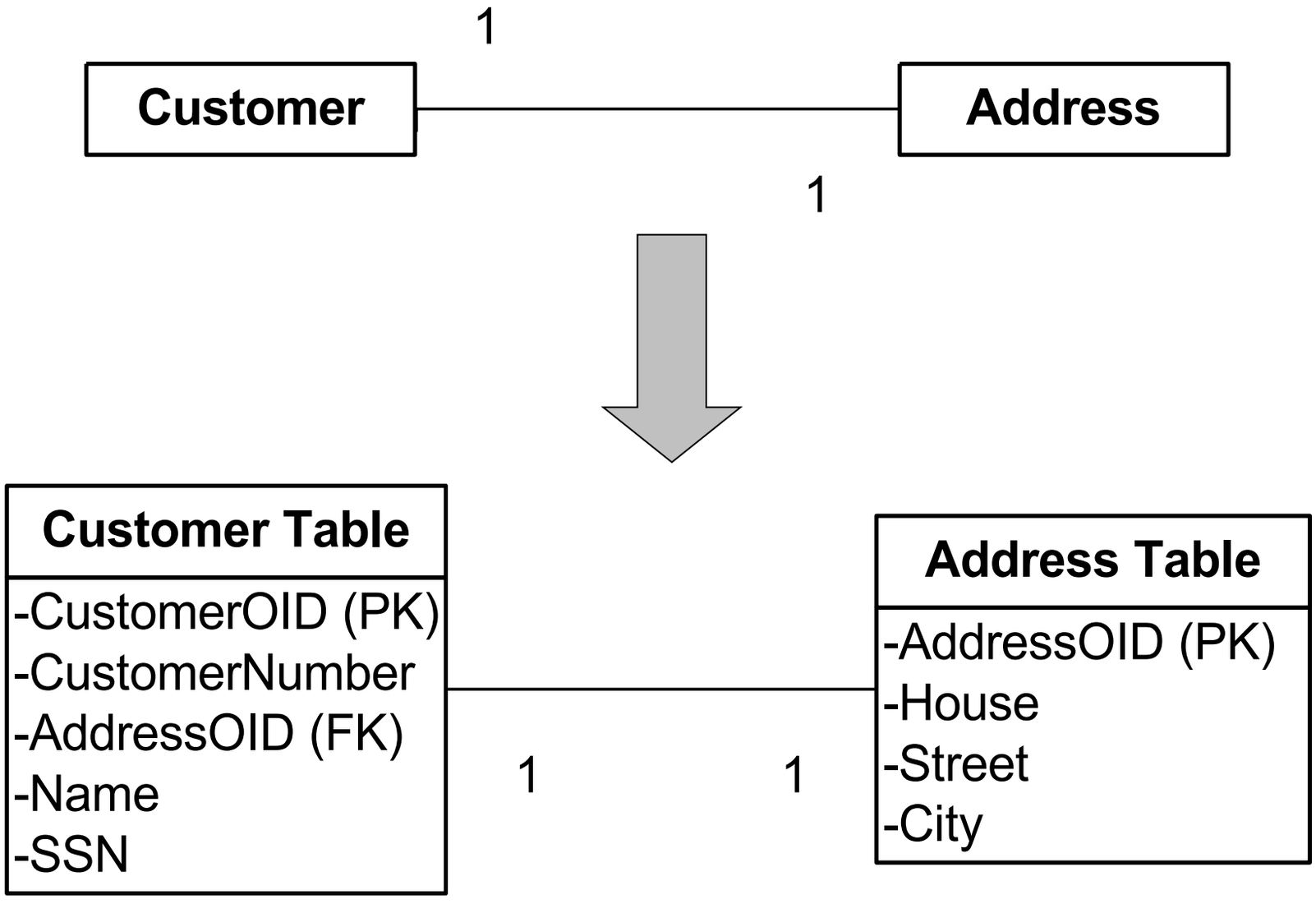
Topic#038

END!

Software Design and Architecture

Topic#039
Object to Relational Mapping
Mapping Cardinality of Association

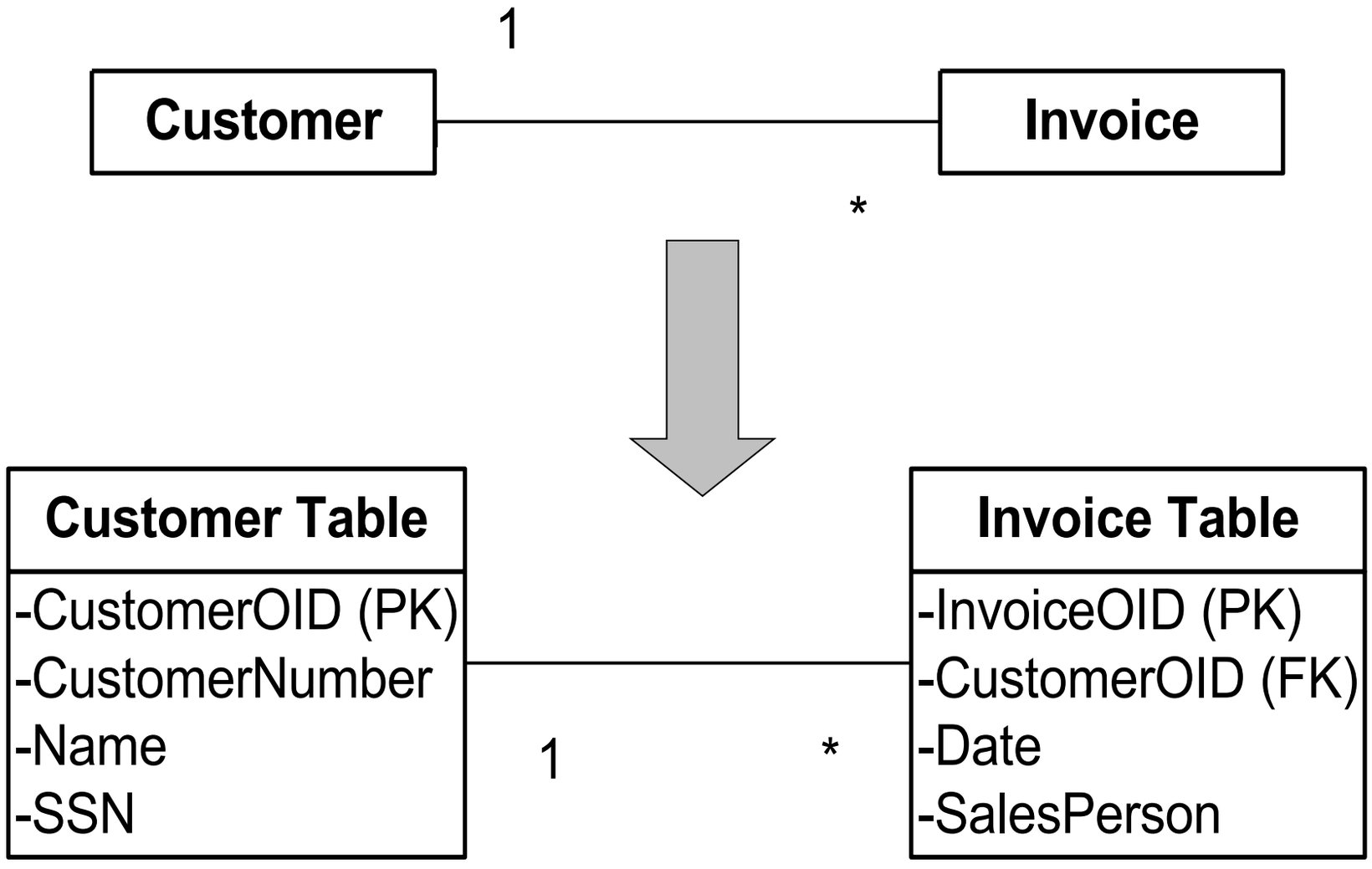
One-to-One Relationships



© Virtual University of Pakistan

thegradup.com

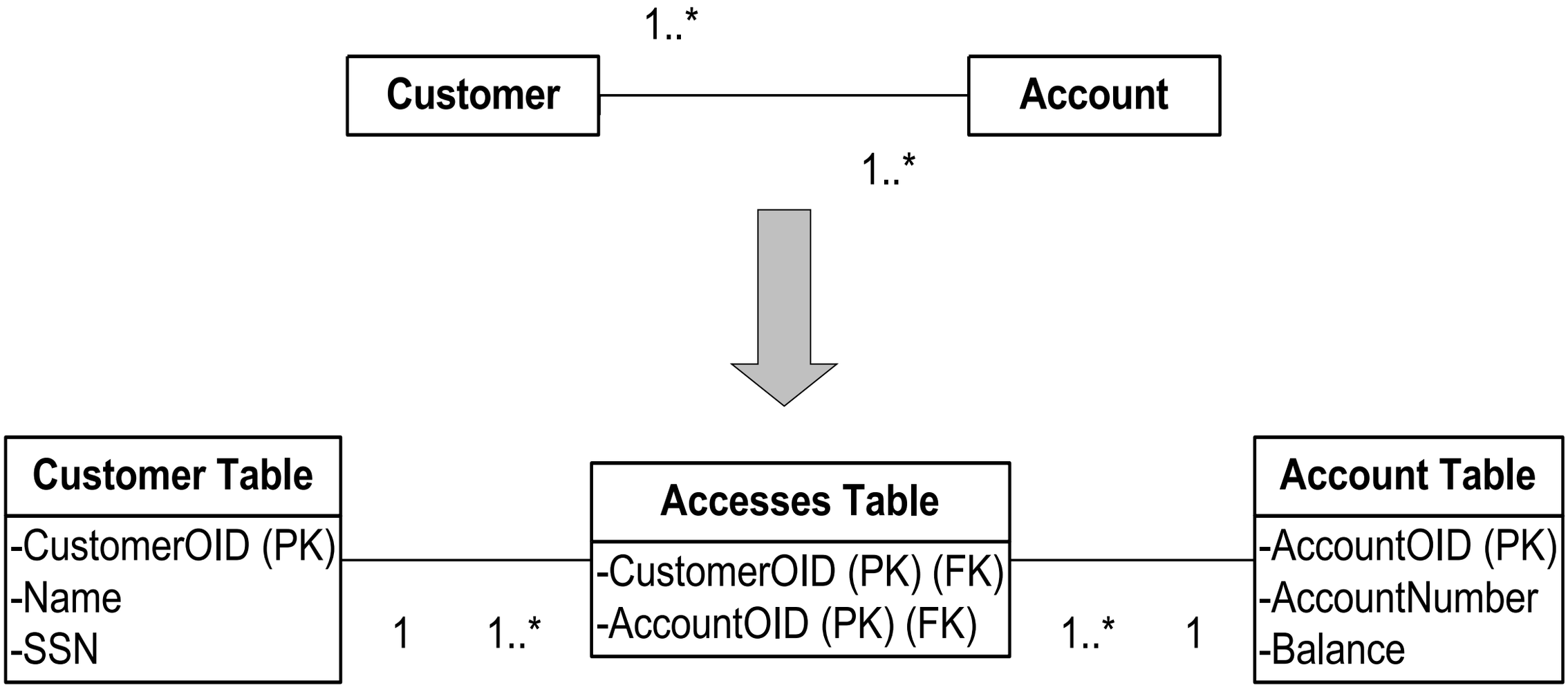
One-to-Many Relationships



© Virtual University of Pakistan

thegradup.com

Many-to-Many Relationships



End of topic

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#039

END!

Software Design and Architecture

Topic#040
Object to Relational Mapping
Summary

	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	1:1
Composition	whole-part relationship	Part is created and destroyed with the whole	Exclusive Part does not know about the whole	1:1, 1:m
Association	Using relationship link	independent	Non-exclusive Can be one-way or bi-directional	1:1, 1:m, m:m

End of topic

Software Design and Architecture

Topic#040

END!

Software Design and Architecture

Topic#041
OOP – Inheritance and Polymorphism

```
class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    virtual int area (void) =0;
    void printarea()
      { cout << this->area() << '\n'; }
};
```


© Virtual University of Pakistan

thegradup.com

```
class Rectangle: public Polygon {
  public:
    Rectangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height; }
};
```


© Virtual University of Pakistan

thegradup.com

```
class Triangle: public Polygon {
  public:
    Triangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height/2; }
};
```


© Virtual University of Pakistan

thegradup.com

```
int main () {
  Polygon * ppoly1 = new Rectangle (4,5);
  Polygon * ppoly2 = new Triangle (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```


© Virtual University of Pakistan

thegradup.com

Questions

- What do we gain by using the base class in the declaration

```
Polygon * ppoly1 = new Rectangle (4,5);
Polygon * ppoly2 = new Triangle (4,5);
```

- Why not

```
Rectangle * ppoly1 = new Rectangle (4,5);
Triangle  * ppoly2 = new Triangle (4,5);
```


© Virtual University of Pakistan

thegradup.com

```
int main () {
        Polygon * ppoly[10];
        ppoly[0] = new Rectangle (4,5);
        ppoly[1] = new Triangle (4,5);
  // and so on
        for (int i = 0; i < 10; i++)
                ppoly[i]->printarea();
  // rest of the code
  return 0;
}
```

End of topic

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#041

END!

Software Design and Architecture

Topic#042
OOP – Object Creation and Factory Method

```
int main () {
        Polygon * ppoly[10];
        ppoly[0] = new Rectangle (4,5);
        ppoly[1] = new Triangle (4,5);
  // and so on
        for (int i = 0; i < 10; i++)
        ppoly[i]->printarea();
  // rest of the code
  return 0;
}
```

Object creation is not polymorphic

© Virtual University of Pakistan

thegradup.com

Object creation and Factory Method

```
Polygon * createPolygon(char t, int w, int h) {
    if (t == 'R')
        return new Rectangle(w, h);
    else if (t == 'T')
        return new Triangle(w, h);
}
```


© Virtual University of Pakistan

thegradup.com

```
int main () {
  Polygon * ppoly[10];
  for (int i = 0; i < 10; i++) {
        // get type, width, and height of the object
        // to be created
        ppoly[i] = createPolygon(t, w, h);
        // pseudo polymorphic behavior
  }
  for (int i = 0; i < 10; i++)
        ppoly[i]->printarea();
  // rest of the code
  return 0;
}
```

End of topic

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#042

END!

Software Design and Architecture

Topic#043
OOP – The Magic Behind Polymorphism

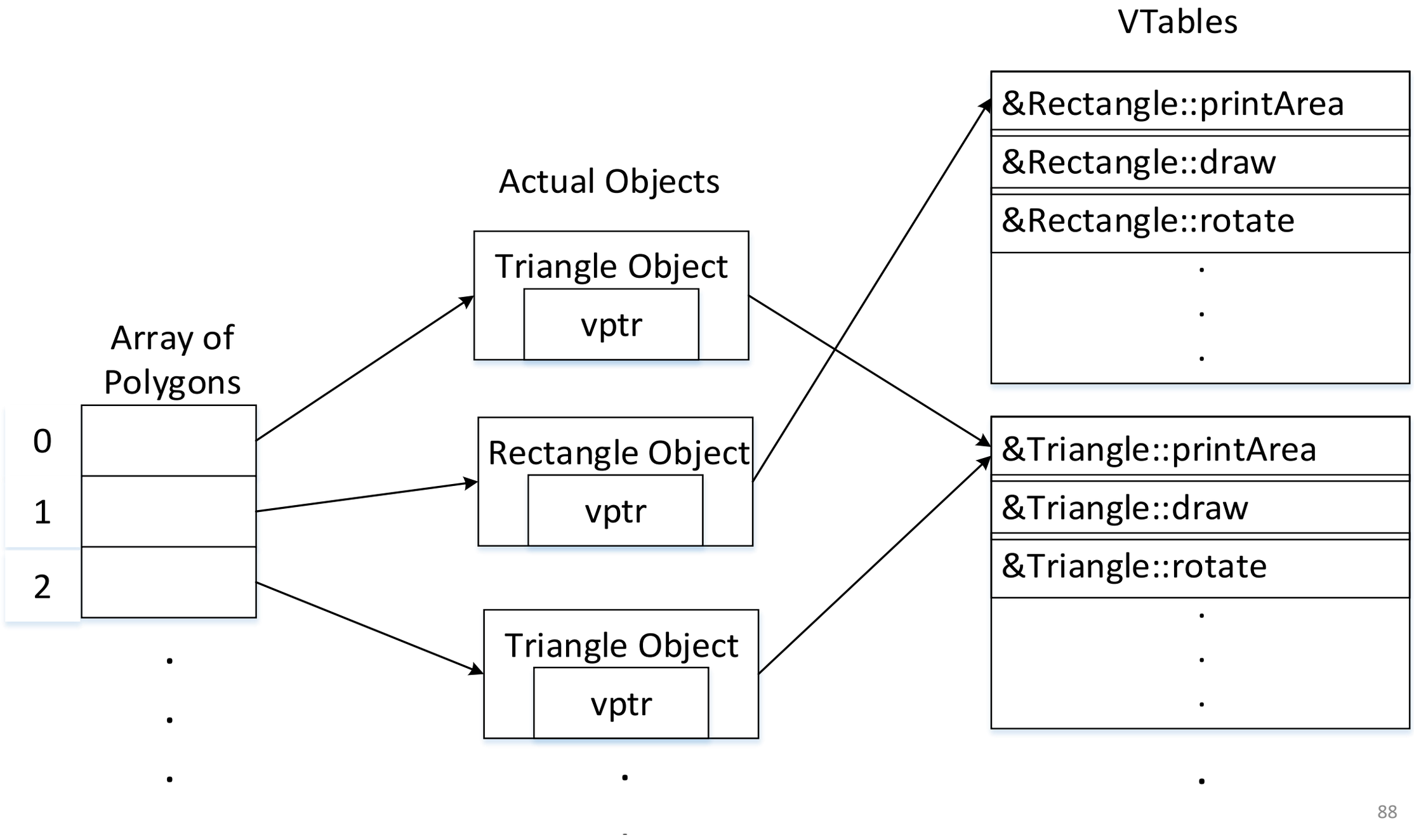
- Compiler maintains two things:

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

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

© Virtual University of Pakistan

thegradup.com



© Virtual University of Pakistan

thegradup.com

- 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.**

© Virtual University of Pakistan

thegradup.com

- 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

© Virtual University of Pakistan

thegradup.com

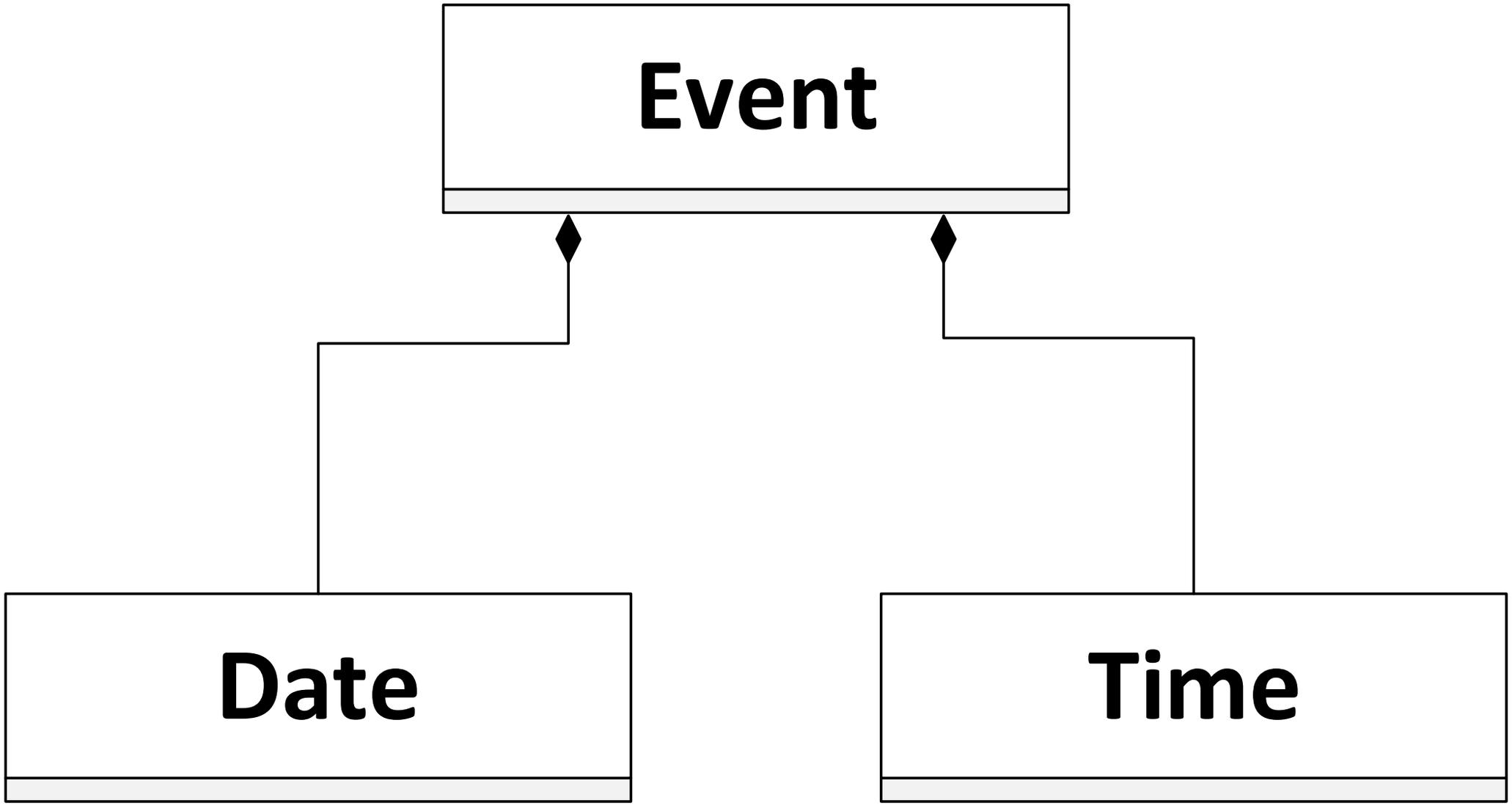
Software Design and Architecture

Topic#043

END!

Software Design and Architecture

Topic#044
OOP – Implementing Composition



© Virtual University of Pakistan

thegradup.com

```
class Time
{
public:
    Time();
    Time(int, int);
    // some setters, getters, and utility functions
    void printTime();
private:
    int hr;
    int min;
};
```


© Virtual University of Pakistan

thegradup.com

```
class Date
{
public:
    Date();
    Date(int, int, int);
    // some setters, getters, and utility functions
    void printDate();
private:
    int month;
    int day;
    int year;
};
```


© Virtual University of Pakistan

thegradup.com

```
class Event {
public:
   Event(int hours = 0, int minutes = 0,
              int m = 1, int d = 1, int y = 1900,
              string name = “Start of 20th Century");

   // setters, getters, and other utility functions
     void printEventData();
private:
     string eventName;
     Time eventTime;
     Date eventDay;
};
```


© Virtual University of Pakistan

thegradup.com

```
int main()
{
    //instantiate an object and set data for Eid prayer
    Event eidPrayer(6, 30, 12, 8, 2019, “Eid Prayer”);

    eidPrayer.printEventData();

            //print out the data for object
```

```
    //instantiate the second object

    Event independenceDay(8, 0, 14, 8, 2019,
                                        "National Anthem");

    independenceDay.printEventData();
            //print out the data for the second object

     return 0;
}
```

```
Event::Event(int hours, int minutes,
             int m, int d, int y, string name)
                 : eventTime(hours, minutes),
                   eventDay(m, d, y)
{
    eventName = name;
}
```

```
void Event::printEventData()
{
    cout << eventName << " occurs ";
    eventDay.printDate();
    cout << " at ";
    eventTime.printTime();
    cout << endl;
}
```

End of topic

Software Design and Architecture

Topic#044

END!

Software Design and Architecture

Topic#045
OOP – Inheritance vs Composition

```
Derived_class_name::
Derived_class_constructor_name(types and parameters):
base_class_construcor_name(parameters)
{
    set parameters which are not set by the base
    class constructor;
}
```

Inheritance

```
Aggregate_class_name::
Aggregate_class_construcor_name(types and params):
component_class_instance(parameters)
{
    set parameters which are not set by the component
    class constructor;
}
```

Composition

- 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

Topic#045

END!

Software Design and Architecture

Topic#046
OOP – Implementing Uni-directional Associations

Example – Lecturer-Course Relationship

- A lecturer may be assigned a course to teach. That lecturer may be removed from that course at anytime.

Unidirectional: lecturer is added to the course and not vice-versa

```
class course
{
private:
        lecturer * L    //    L is a pointer to an
                        //    object of type lecturer
public:
        course();
        void addlecturer(Lecturer *);
        void removelecturer();
};
```

Lecturer and course are created separately and then linked together.

```
void course::addlecturer(Lecturer *teacher)
{
        L = teacher;
}
void course::removelecturer()
{
        L = NULL;
}
```

End of topic

Software Design and Architecture

Topic#046

END!

Software Design and Architecture

Topic#047
OOP – Implementing Bi-directional Associations

- Example:
  - A lecturer can be added to the course and vice versa

- In a bi-directional associations both classes must include a reference(via a pointer) to the linked object.

- Problem: User has to update both ends of the relationship.
  - Can be achieved by using the operator ***this.***

```
void course::addlecturer(lecturer *teacher)
{
        L = teacher;
        L -> addcourse(this);
}

void lecturer::addcourse(course *subject)
{
        C = subject;
}
```

```
void main()
{
        course * SE101 = new course(…);
        lecturer *Fakhar = new lecturer(…);
        SE101->addlecturer(Fakhar);
        // the association is updated for both objects.
}
```

N:M associations are implemented using arrays of pointers from one class to another.

What would happen if it could be added from either side?

```
void main()
{
        course * SE101 = new course(…);
        lecturer *Fakhar = new lecturer(…);

        Fakhar->addcourse(SE101);
}
```

```
void course::addlecturer(lecturer *teacher)
{
        L = teacher;
        L -> addcourse(this);
}

void lecturer::addcourse(course *subject)
{
        C = subject;
        C -> addlecturer(this);
}
```

Simple – Isn’t it?

```
void course::addlecturer(lecturer *teacher)
{
        if (L == NULL) {
                L = teacher;
                L -> addcourse(this);
        }
}

void lecturer::addcourse(course *subject)
{
        if (C == NULL) {
                C = subject;
                C -> addlecturer(this);
        }
}
```

Control the infinite recursion

```
void main()
{
        course * SE101 = new course(…);
        lecturer *Fakhar = new lecturer(…);
        SE613->addlecturer(Fakhar);
        // or
        // Fakhar->addcourse(SE101);
        …

}
```

End of topic

Software Design and Architecture

Topic#047

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 048 – 061

Good Object-Oriented Design Principles

© Virtual University of Pakistan

Software Design and Architecture

Topic#028

END!

Software Design and Architecture

Topic#048
SOLID Design Principles - Introduction

© Virtual University of Pakistan

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

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

© Virtual University of Pakistan

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)



Robert C. Martin

SOLID Principle - History

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

© Virtual University of Pakistan

S.O.L.I.D.

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

© Virtual University of Pakistan

Software Design and Architecture

Topic#048

END!

Software Design and Architecture

Topic#049
SOLID Design Principles
Single Responsibility Principle (SRP)

© Virtual University of Pakistan

Single Responsibility Principle (SRP)

A class should have only one reason to change

- Cohesion
  - Defined as the functional relatedness of the elements of a module
  - Related to the forces that cause a module, or a class, to change

Responsibility - a reason for change

- What a class does
  - The more a class does, the more likely it will change
  - The more a class changes, the more likely we will introduce bugs

SRP – Coupling and Cohesion

- If a class has more than one responsibility
  - Low cohesion
  - High coupling - the responsibilities become coupled
    - Changes to one responsibility may impair or inhibit the class's ability to meet the others
    - Leads to fragile designs that break in unexpected ways when changed

SRP

- One of the simplest of the principles but one of the most difficult to get right.
- Finding and separating conjoined responsibilities is much of what software design is really about
- The rest of the principles come back to this issue in one way or another

Software Design and Architecture

Topic#049

END!

Software Design and Architecture

Topic#050
SOLID Design Principles
Single Responsibility Principle (SRP) - Example

© Virtual University of Pakistan

```
class Post
{
    void CreatePost(Database db, string postMessage)
    {
        try
        {
            db.Add(postMessage);
        }
        catch (Exception ex)
        {
            db.LogError("An error occured: ", ex.ToString());
            File.WriteAllText("\LocalErrors.txt", ex.ToString());
        }
    }
}
```

Create a new post

Log Error

Multiple Responsibilities

```
class ErrorLogger
{
    void log(string error)
    {
      db.LogError("An error occured: ", error);
      File.WriteAllText("\LocalErrors.txt", error);
    }
}
```

Class for handling error logging

```
class Post
{
    private ErrorLogger errorLogger = new ErrorLogger();

    void CreatePost(Database db, string postMessage)
    {
        try
        {
            db.Add(postMessage);
        }
        catch (Exception ex)
        {
            errorLogger.log(ex.ToString());
        }
    }
}
```

create a new post

Log Error delegated using abstraction

Single Responsibility!

Software Design and Architecture

Topic#050

END!

Software Design and Architecture

Topic#051
SOLID Design Principles
Open-Closed Principle (OCP)

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.

© Virtual University of Pakistan

- 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.

- Open-Close - Are these two attributes are at odds with each other?
  - The normal way to extend the behavior of a module is to make changes to that module.
  - A module that cannot be changed is normally thought to have a fixed behavior.
  - How can these two opposing attributes be resolved?

© Virtual University of Pakistan

Extension through inheritance, composition, and association

Software Design and Architecture

Topic#051

END!

Software Design and Architecture

Topic#052
SOLID Design Principles
Open-Closed Principle (OCP) - Example

```
class Post
{
    void CreatePost(Database db, string postMessage)
    {
        if (postMessage.StartsWith("#"))
        {
            db.AddAsTag(postMessage);
        }
        else
        {
            db.Add(postMessage);
        }
    }
}
```

Case for #

Otherwise

What Happens if we had to add a case for @

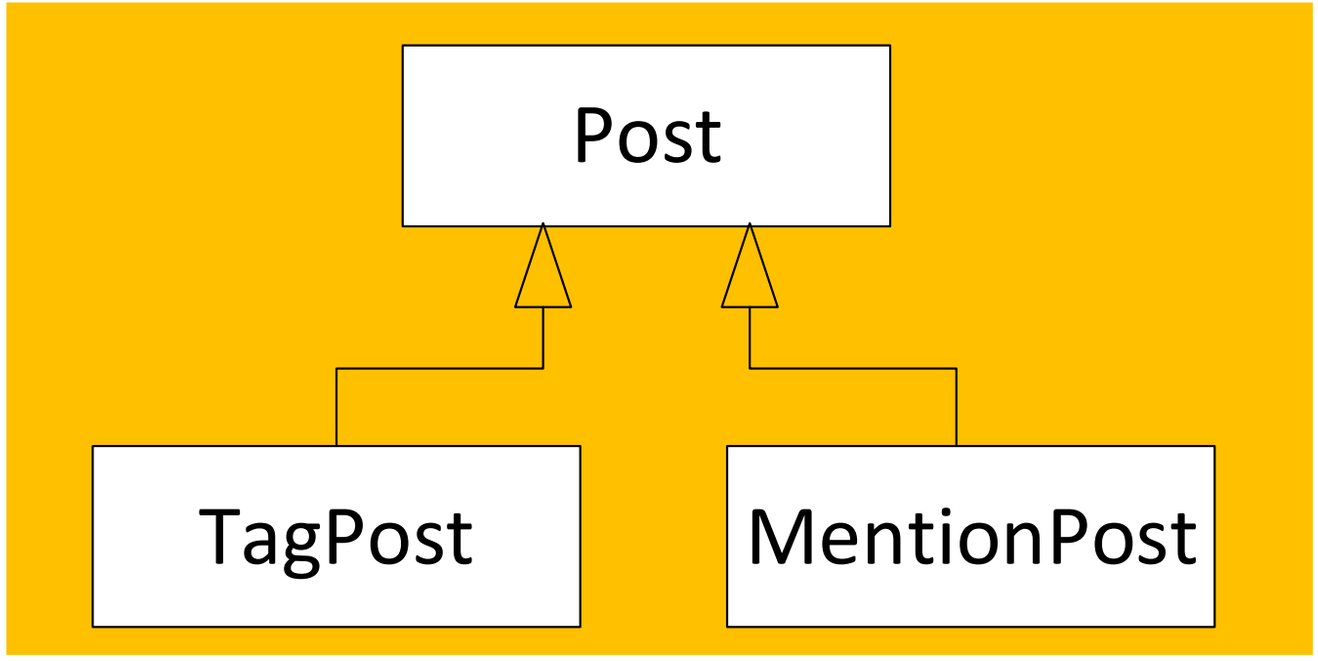
Violates OCP

```
class Post
{
   void CreatePost(Database db,
                   string postMessage)
   {
      db.Add(postMessage);
   }
}
```

```
class TagPost
{
   override void CreatePost(Database db,
                            string postMessage)
   {
      db.AddAsTag(postMessage);
   }
}
```

Now OCP Compliant

Can easily add a new post type



© Virtual University of Pakistan

Software Design and Architecture

Topic#052

END!

Software Design and Architecture

Topic#053
SOLID Design Principles
Liskov's Substitution Principle (LSP)

© Virtual University of Pakistan

Liskov Substitution Principle (LSP)

- *Subtypes must be substitutable for their base types.*

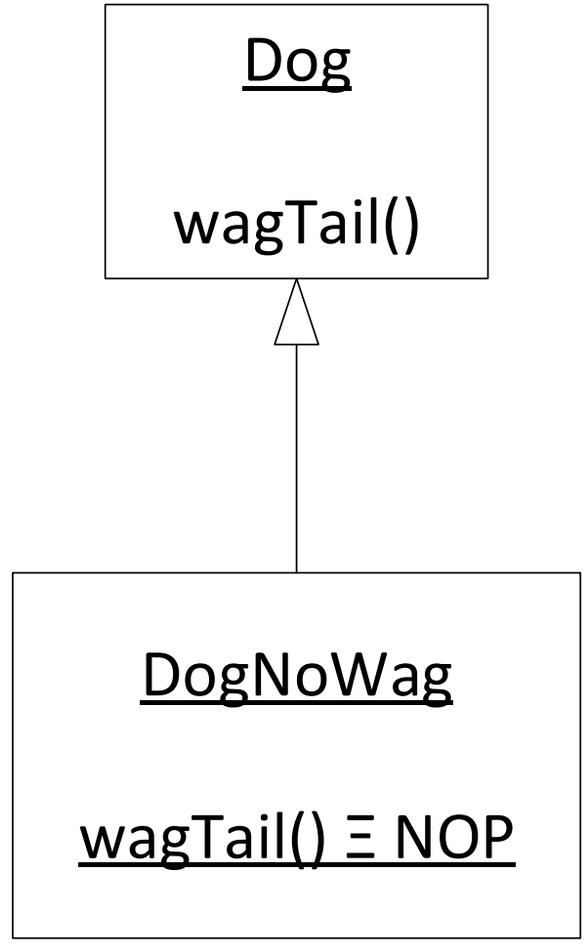
- *Deals with design of inheritance hierarchies*

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

All dogs know how to wag their tails.

DogNoWag is a special type of dog.

DogNoWag does not know how to wag its tail.



Ignoring the functionality defined in the base class is violation of LSP

© Virtual University of Pakistan

Software Design and Architecture

Topic#053

END!

Software Design and Architecture

Topic#054
SOLID Design Principles
Liskov's Substitution Principle (LSP) - Example

© Virtual University of Pakistan

```
class Rectangle
{
public:
   virtual void SetWidth(double w) {itsWidth=w;}
   virtual void SetHeight(double h) {itsHeight=w;}
   double GetHeight() const {return itsHeight;}
   double GetWidth() const {return itsWidth;}
private:
   double itsWidth;
   double itsHeight;
};
```

```
class Square : public Rectangle
{
public:
    virtual void SetWidth(double w);
    virtual void SetHeight(double h);
};

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

SetHeight will be the same as SetWidth

By setting the width of a Square object, its height will change correspondingly and vice versa.

The Square object will remain a mathematically proper square

Good? – Not really!

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

Works fine for a rectangle
but
assertion error if passed a square!

A violation of LSP!

Software Design and Architecture

Topic#054

END!

Software Design and Architecture

Topic#055

SOLID Design Principles

Liskov's Substitution Principle and Inheritance

The True Meanings of IsA Relationship

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.

© Virtual University of Pakistan

Validity is not Intrinsic

- Square and Rectangle classes viewed in isolation, appeared to be self consistent and valid.

- Viewed from a programmer standpoint who made reasonable assumptions about the base class, the model broke down.

Validity is not Intrinsic

- When considering whether a particular design is appropriate or not, one must not simply view the solution in isolation.
  - One must view it in terms of the reasonable assumptions that will be made by the users of that design.

© Virtual University of Pakistan

What Went Wrong? (W^3)

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

What Went Wrong? (W^3)

- No!

- A square might be a rectangle, but a Square object is definitely not a Rectangle object.

What Went Wrong? (W^3)

- Why?
  - Because the behavior of a Square object is not consistent with the behavior of a Rectangle object.
  - Behaviorally, a Square is not a Rectangle!

And it is behavior that software is really all about!

- The LSP makes clear that in OOD the ISA relationship pertains to behavior.
  - Not intrinsic private behavior, but extrinsic public behavior;
  - behavior that clients depend upon.

Software Design and Architecture

Topic#055

END!

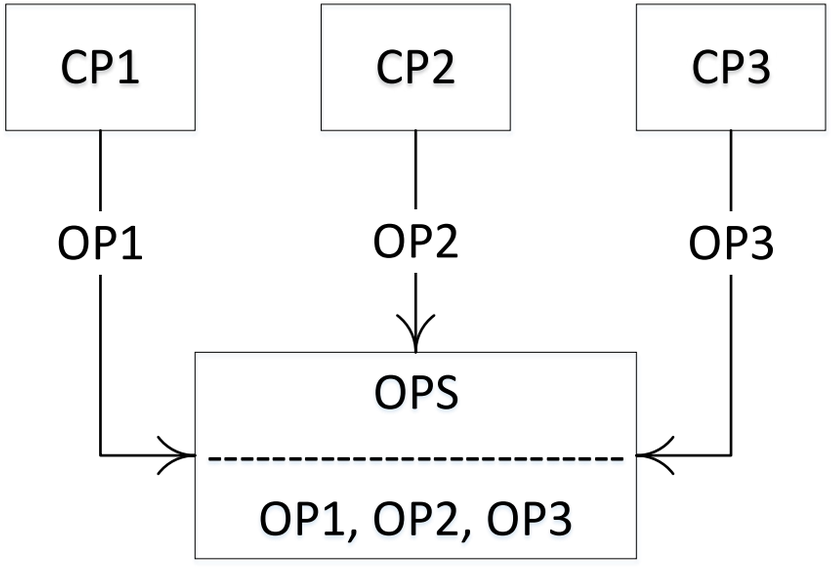
Software Design and Architecture

Topic#056
SOLID Design Principles
Interface Segregation Principle (ISP)

© Virtual University of Pakistan

Interface Segregation Principle (ISP)

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



if you have an abstract class or an interface, then the implementers should not be forced to implement parts that they don't care about.

Interface Segregation Principle (ISP)

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

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

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

Deals with non-cohesive interfaces!
Coupling

- This principle deals with the disadvantages of “fat” interfaces.
  - 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.

- 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.

- The ISP acknowledges that there are objects that require non-cohesive interfaces
- It suggests that clients should not know about them as a single class.
- Instead, clients should know about abstract base classes that have cohesive interfaces.

Software Design and Architecture

Topic#056

END!

Software Design and Architecture

Topic#057
SOLID Design Principles
Interface Segregation Principle (ISP) - Example

```
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

To avoid the problem, simply create a new interface

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

interface IPostRead
{
    void ReadPost();
}
```

If any class might need both the CreatePost() method and the ReadPost() method, it will implement both interfaces.

Complying to ISP!

Software Design and Architecture

Topic#057

END!

Software Design and Architecture

Topic#058
SOLID Design Principles
Dependency Inversion Principle (DIP)

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.

The Dependency-Inversion Principle (DIP)

Natural progression of the Liskov Substitution Principle, the Open Closed Principle and even the Single Responsibility Principle.

© Virtual University of Pakistan

The Dependency-Inversion Principle (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

Software Design and Architecture

Topic#058

END!

Software Design and Architecture

Topic#059
SOLID Design Principles
Dependency Inversion Principle (DIP) - Example

© Virtual University of Pakistan

```
class Post
{
    private ErrorLogger errorLogger = new ErrorLogger();

    void CreatePost(Database db, string postMessage)
    {
        try
        {
            db.Add(postMessage);
        }
        catch (Exception ex)
        {
            errorLogger.log(ex.ToString())
        }
    }
}
```

Dependency

We create the ErrorLogger instance from within the Post class.

This is a violation of the dependency inversion principle.

If we wanted to use a different kind of logger, we would have to modify the Post class.

```
class Post
{
    private Logger _logger;

    public Post(Logger injectedLogger)
    {
        _logger = injectedLogger;
    }

    void CreatePost(Database db, string postMessage)
    {
        // no change here
    }
}
```

Dependency Injection

Software Design and Architecture

Topic#059

END!

Software Design and Architecture

Topic#060
Good OO design principles
Law of Demeter (LoD)

© Virtual University of Pakistan

Law of Demeter – A Style Rule for building systems

- Demeter Origins
  - Proposed by The Demeter Research Group in 1987
    - "Law of Demeter" Paper
      - Lieberherr, Karl. J. and Holland, I., “Assuring good style for object-oriented programs”, IEEE Software, September 1989, pp 38-48
- Covered in many major books on OO design and programming.

Reduces coupling and improves maintainability

Law of Demeter – A Style Rule for building systems

- Its essence is the **"principle of least knowledge"** regarding the object instances used within a method.

Law of Demeter – A Style Rule for building systems

The Law of Demeter says that if I need to request a service of an objects sub-part, I should instead make the request of the object itself and let it propagate this request to all relevant sub-parts, thus the object is responsible for knowing its internal make-up instead of the method that uses it.

Law of Demeter in its original form

- For all classes C, and for all methods M attached to C, all objects to which M sends a message must be:
  - M's argument objects, including the self object or
  - The instance variable objects of C
- Object created by M, or by functions or methods which M calls, and objects in global variables are considered as arguments of M.

Law of Demeter Principle

- **Each unit should only use a limited set of other units: only units "closely" related to the current unit.**

- "Each unit should only talk to its friends."

- "Don't talk to strangers."

© Virtual University of Pakistan

A side-effect of this is that if you conform to LoD, while it may quite increase the maintainability and "adaptiveness" of your software system, you also end up having to write lots of little wrapper methods to propagate methods calls to its components

Software Design and Architecture

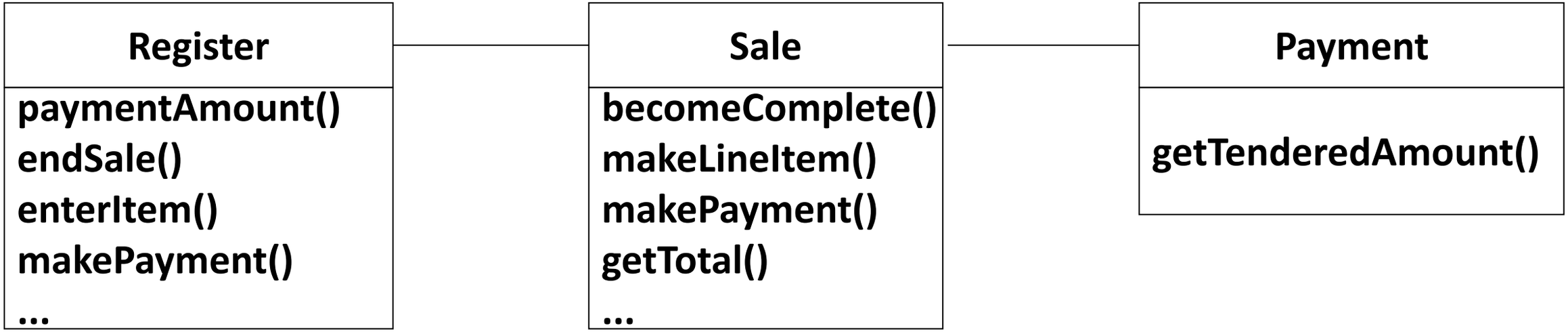
Topic#060

END!

Software Design and Architecture

Topic#061
Good OO design principles
Law of Demeter (LoD) - Example

Register needs to find out the amount of the payment



The class diagram shows that

- Register knows about Sale, and
- Sale knows about the corresponding Payments

```
public void paymentAmount()
{
    Payment payment = sale.getPayment();
    Money amount = payment.getTenderedAmount();
    // same as:
    // Money amount = sale.getPayment().getTenderedAmount();
    // chain calls
}
```



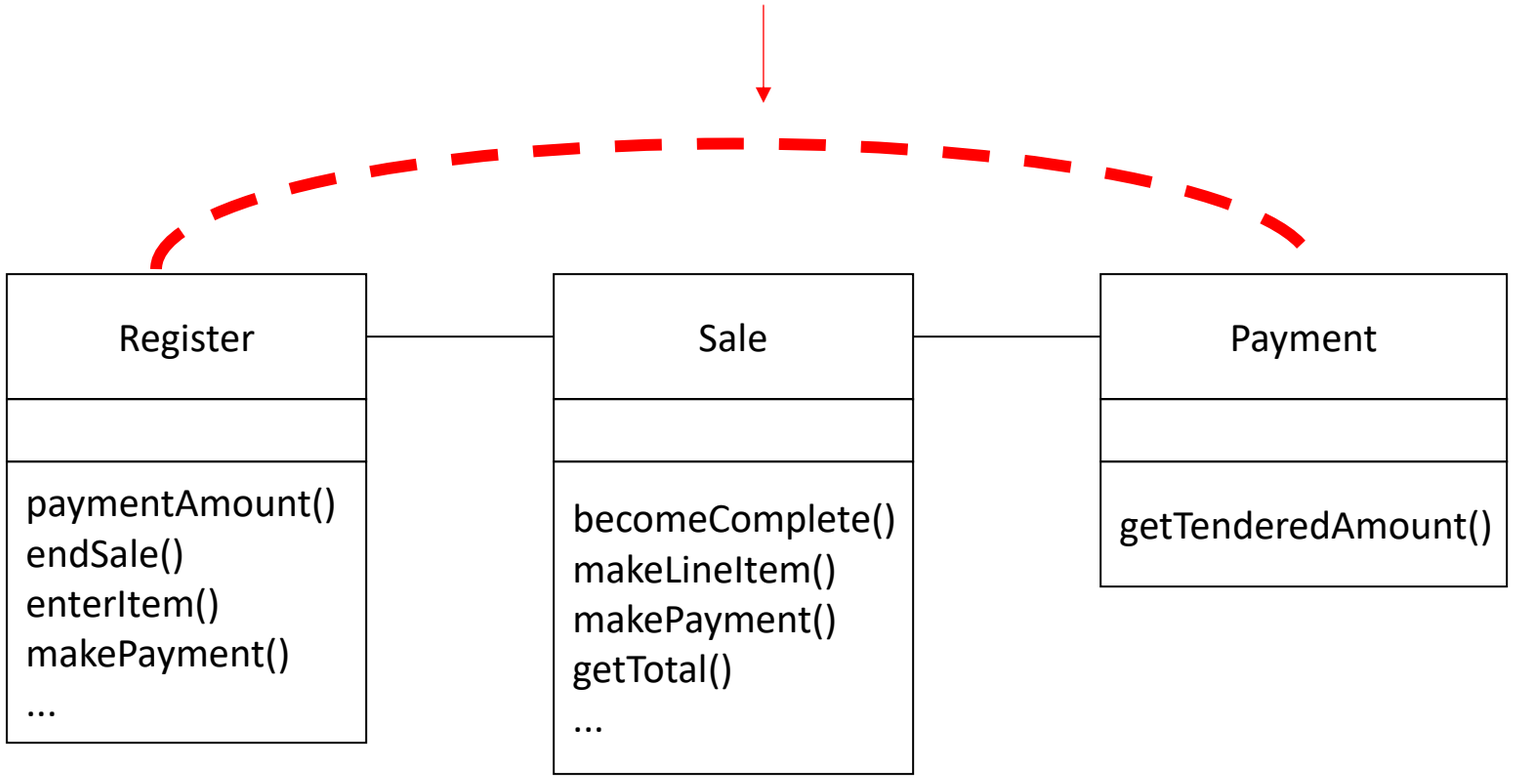


Model vs Actual



Payment is stranger to Register (not visible in the model)

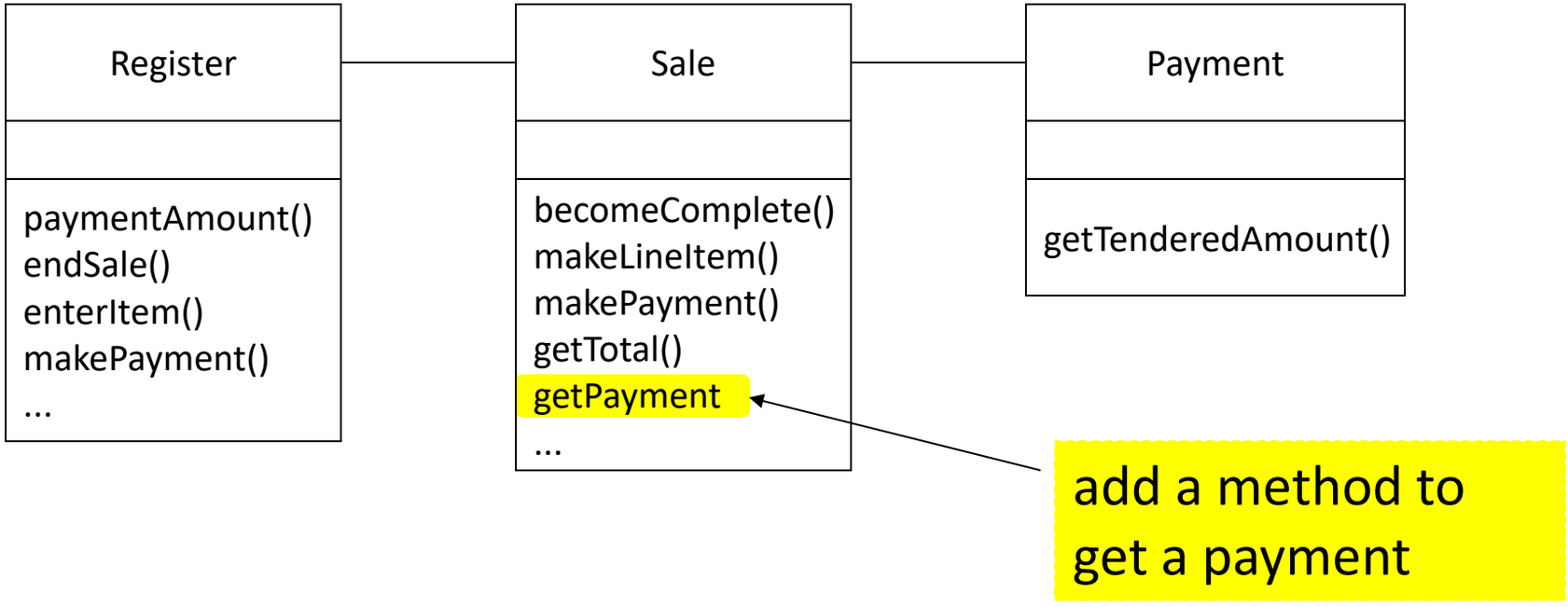
hidden dependency – not visible in the model but present in the code



Register has a "hidden" dependency on Payment

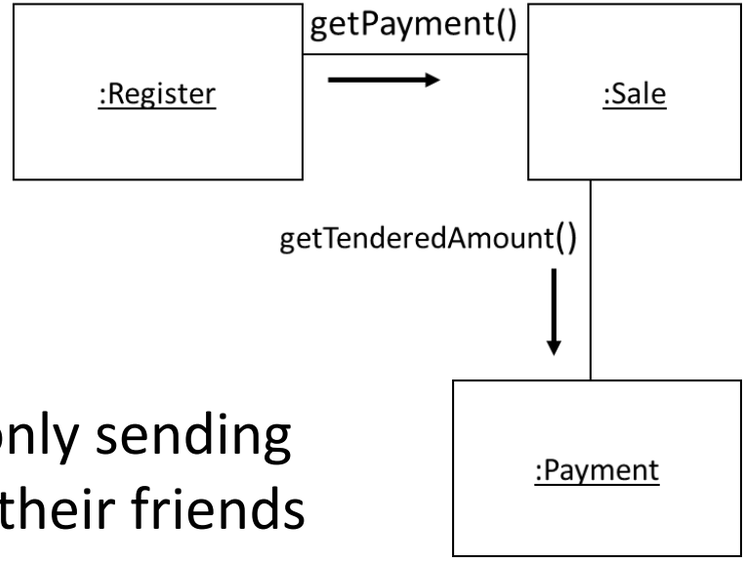
This increases the coupling in our system

LoD – Don’t Talk to Strangers



LoD – Don’t Talk to Strangers

If getPayment() in Sale would invoke getTenderedAmount() in Payment, and return the payment amount, then we can de-couple Register from Payment



Objects are only sending messages to their friends

Register will get the payment amount it is after, **but it won’t know how it was obtained** - see Parnas’ concept of *information hiding* on

make the solution more robust, less sensitive to changes, less coupling

Software Design and Architecture

Topic#061

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 062 – 068

Object-Oriented Design

Solutions to some general design problems

© Virtual University of Pakistan

Software Design and Architecture

Topic#062
Action-Oriented Design in the Disguise of object-Oriented Design

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

- **Poorly distributed system intelligence**
  - **god classes**
    - **Behavioral form** **- can’t answer what I know**
    - **Data form** **- can’t answer what I do**

© Virtual University of Pakistan

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**

Distribute system intelligence horizontally as uniformly as possible

© Virtual University of Pakistan

Software Design and Architecture

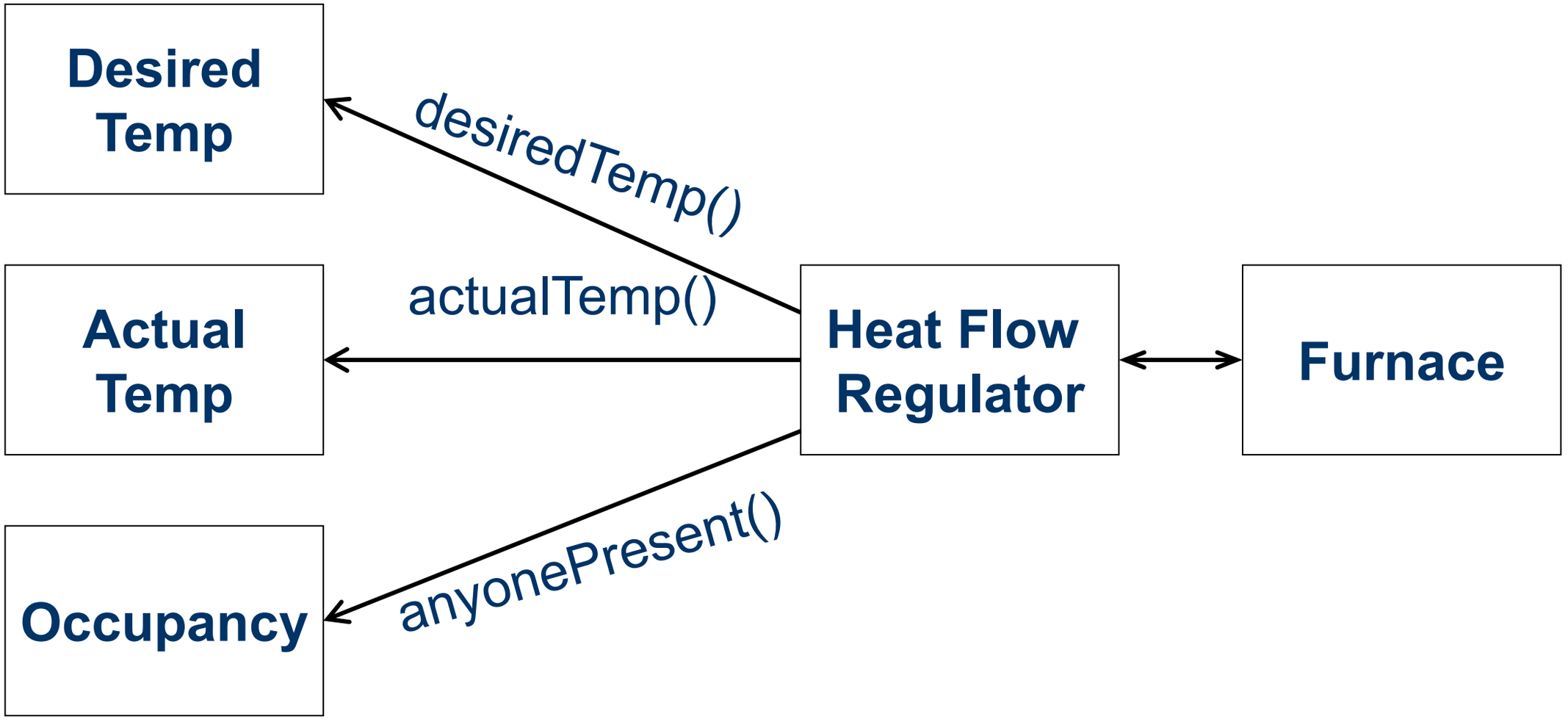
Topic#062

END!

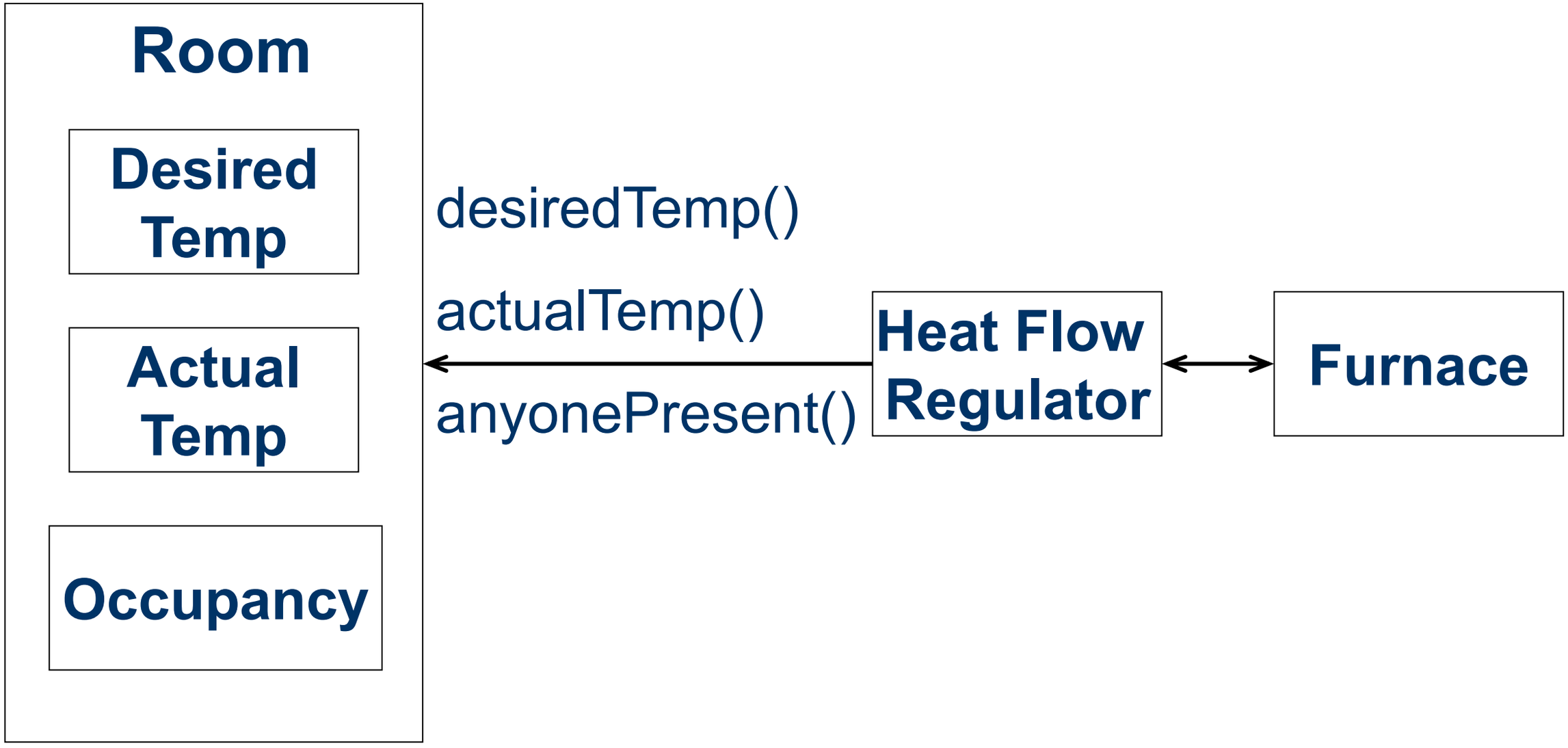
Software Design and Architecture

Topic#063
Handling Poorly Distributed System Intelligence

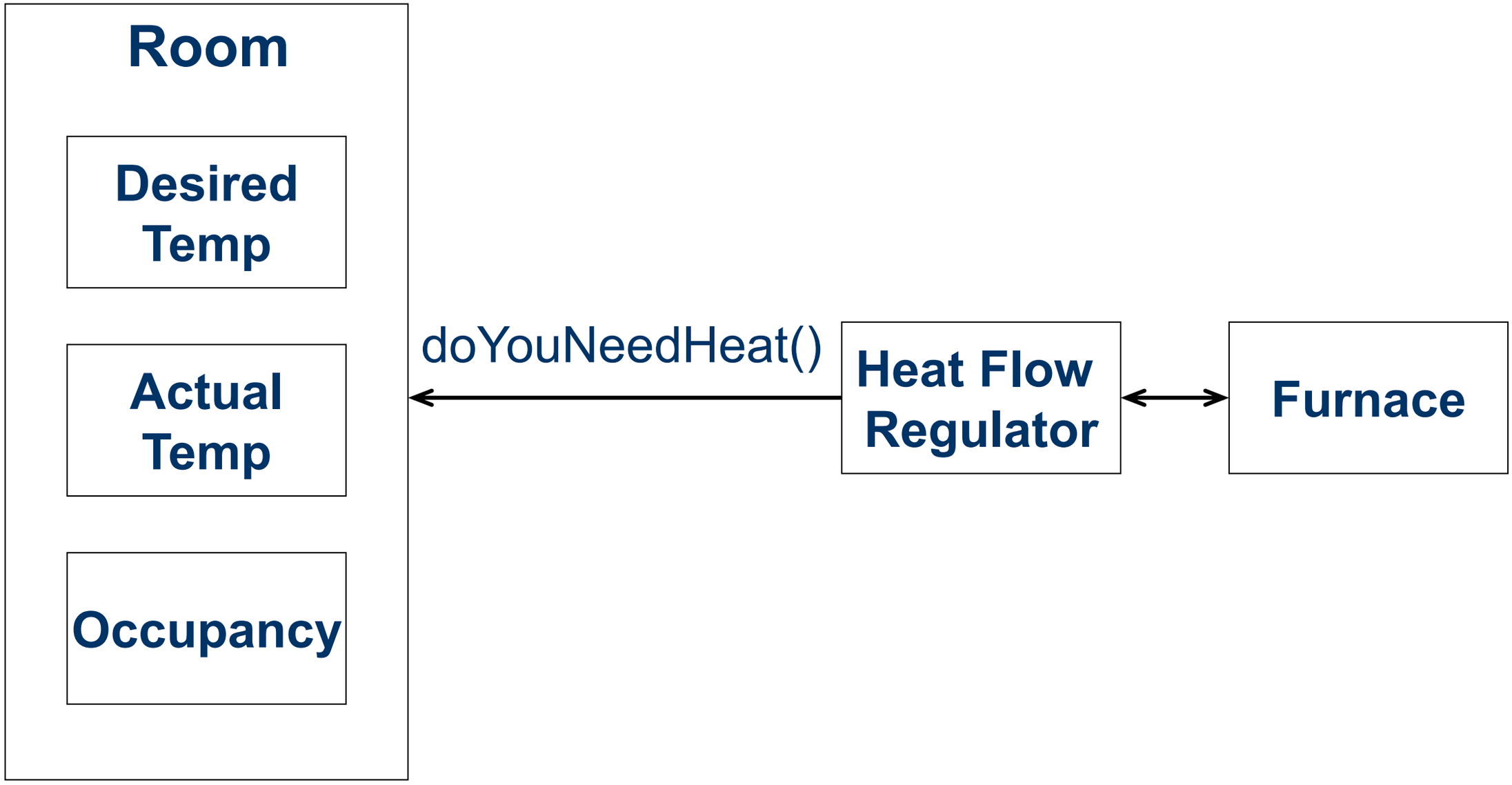
© Virtual University of Pakistan



© Virtual University of Pakistan



© Virtual University of Pakistan



© Virtual University of Pakistan

Software Design and Architecture

Topic#063

END!

Software Design and Architecture

Topic#064
Modelling Policy – Jacobson's Heuristic

© Virtual University of Pakistan

- Student – Course – Course Offering

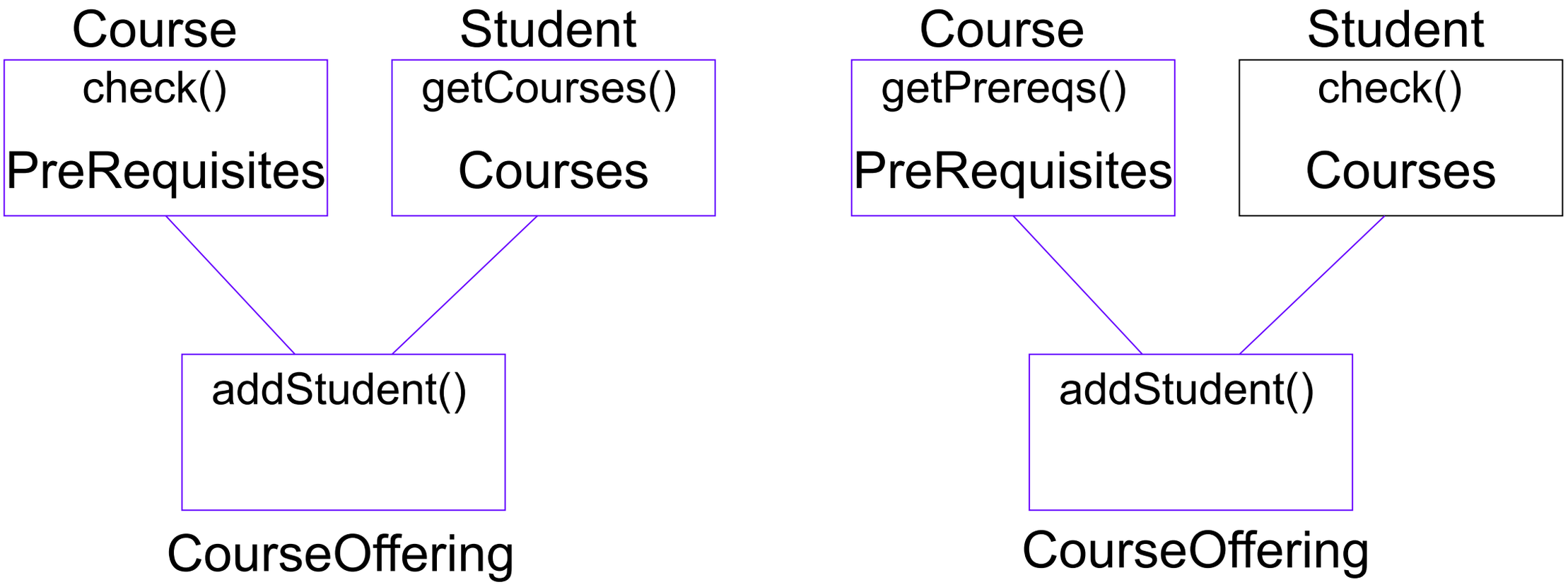
- Course offering would like to check that the student has taken the pre-reqs.

- How to perform this function?
  - Student knows which course he/she has taken
  - Course knows the pre-reqs

Policy Information

Requires data of two classes to make a decision.

© Virtual University of Pakistan



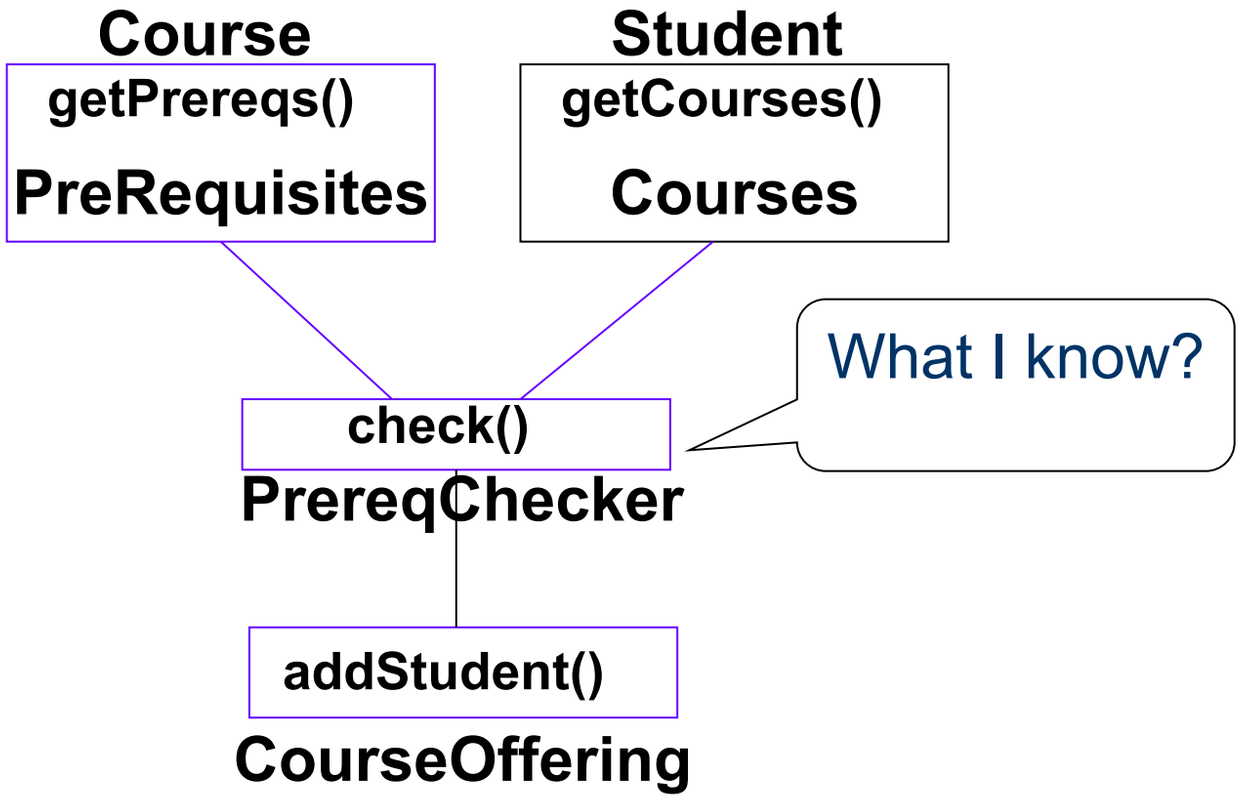
© Virtual University of Pakistan

Jocobson’s Heuristic

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.

© Virtual University of Pakistan

Controller Class



© Virtual University of Pakistan

Software Design and Architecture

Topic#064

END!

Software Design and Architecture

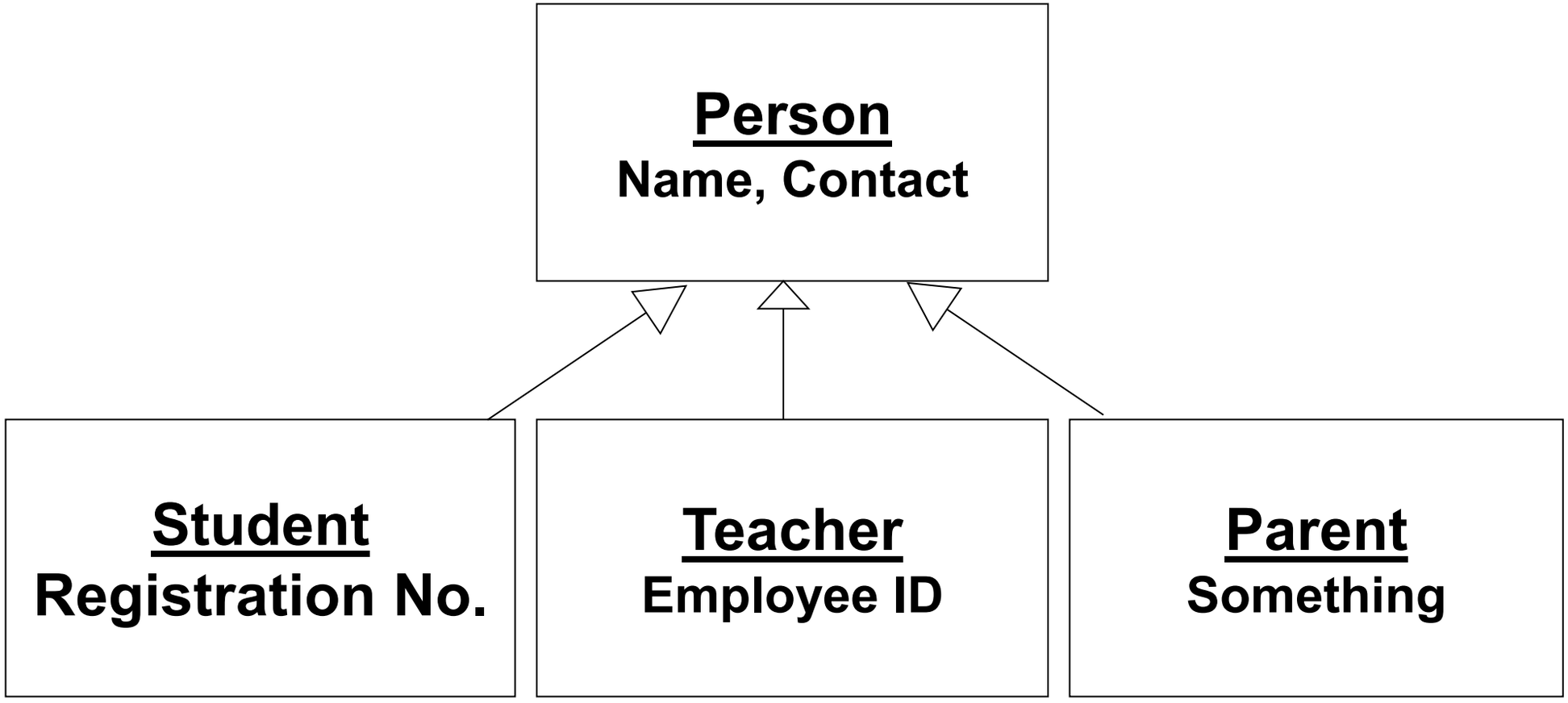
Topic#065
Handling roles - Player-Role Pattern

Roles

Student	Teacher	Parent
Name, Contact	Name, Contact	Name, Contact
Registration No.	Employee ID	Something

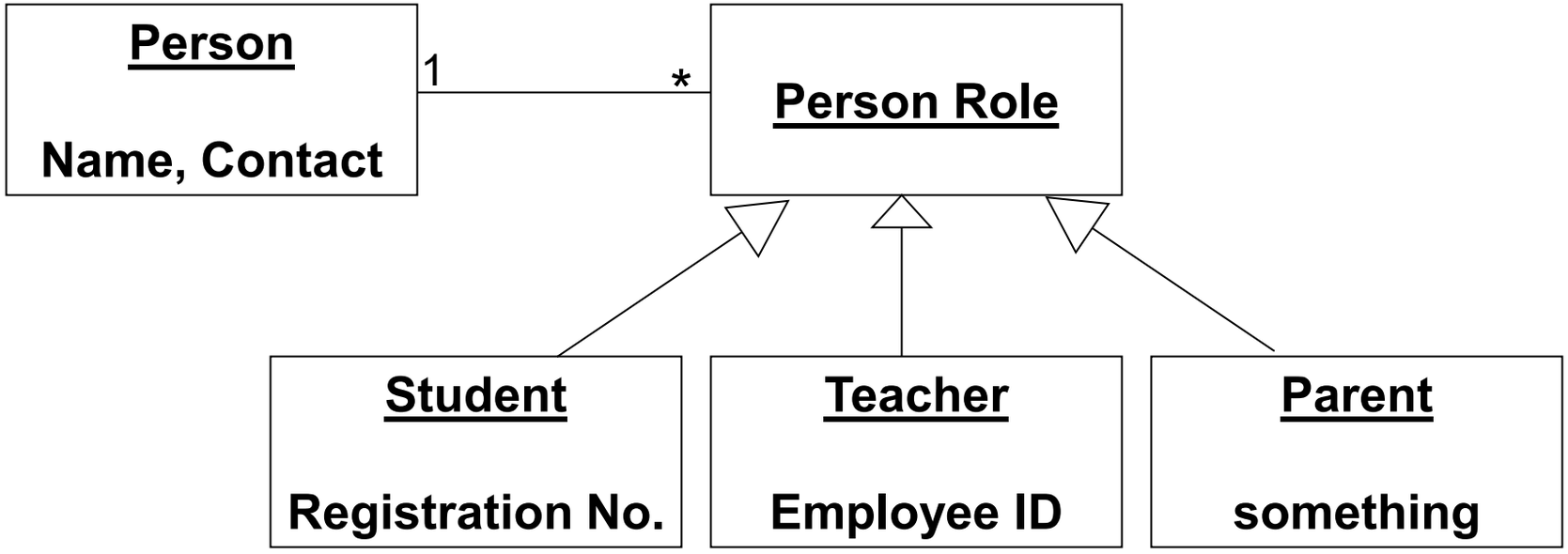
© Virtual University of Pakistan

Roles



If one person plays different roles
Data Coherence Problem

Role Object



© Virtual University of Pakistan

Software Design and Architecture

Topic#065

END!

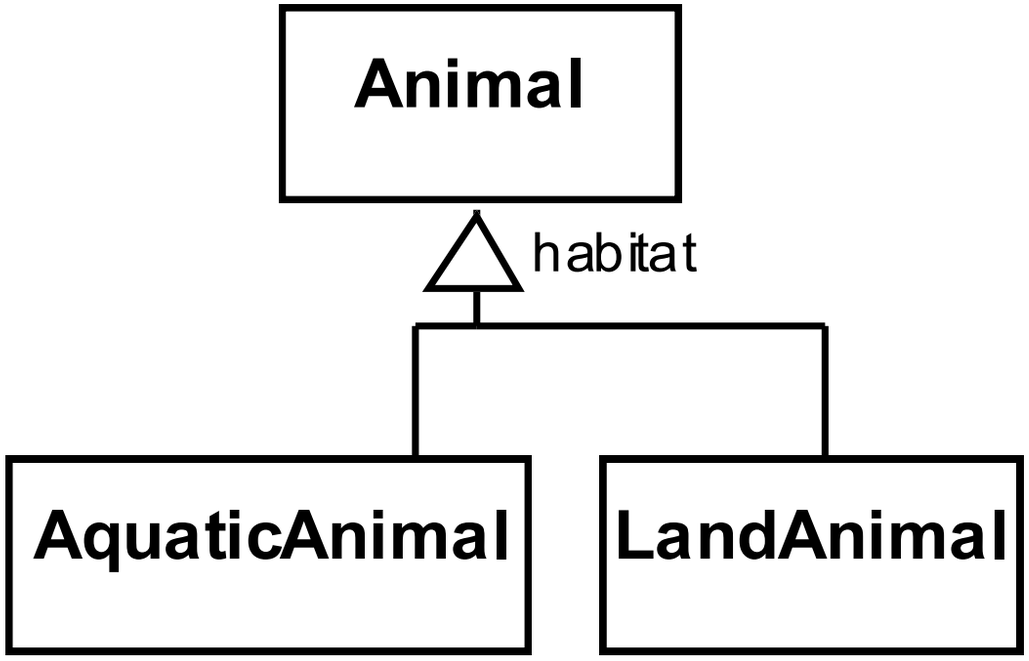
Software Design and Architecture

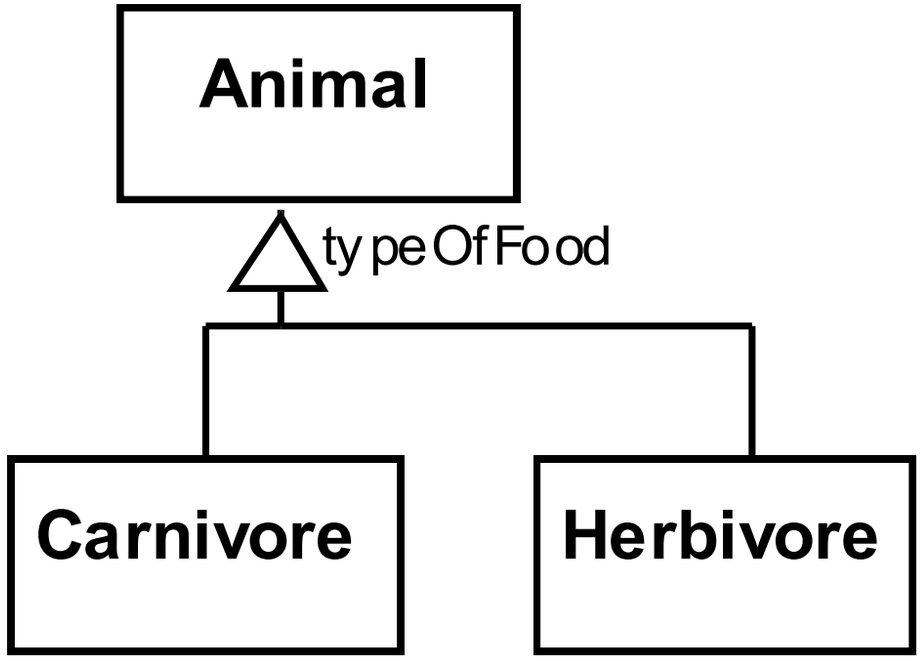
Topic#066
Handling multiple discriminators using Player-Role Pattern

© Virtual University of Pakistan

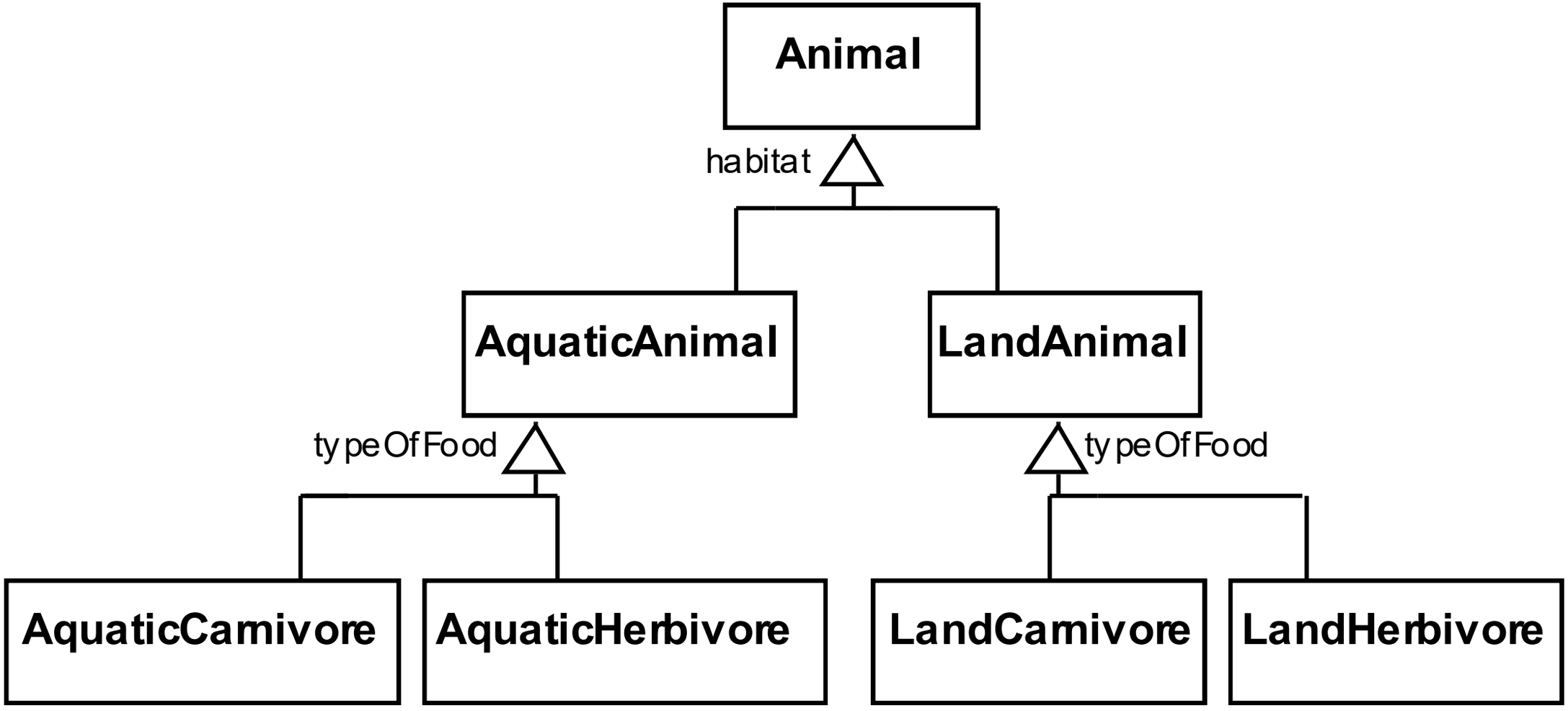
- A **discriminator** is a label that describes the criteria used in the specialization

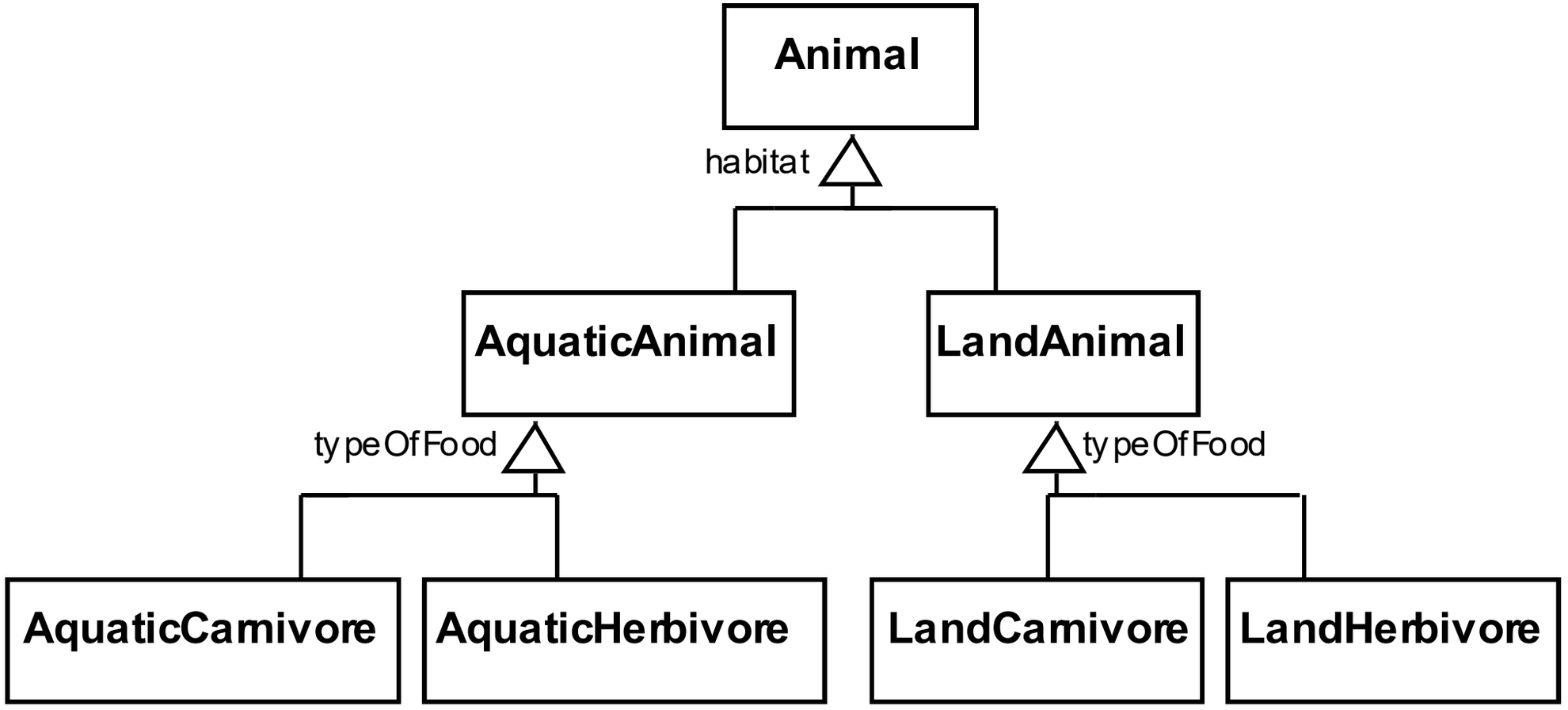
- Can be thought of an attribute that will have a different value in each subclass


Animal
habitat
AquaticAnimal
LandAnimal

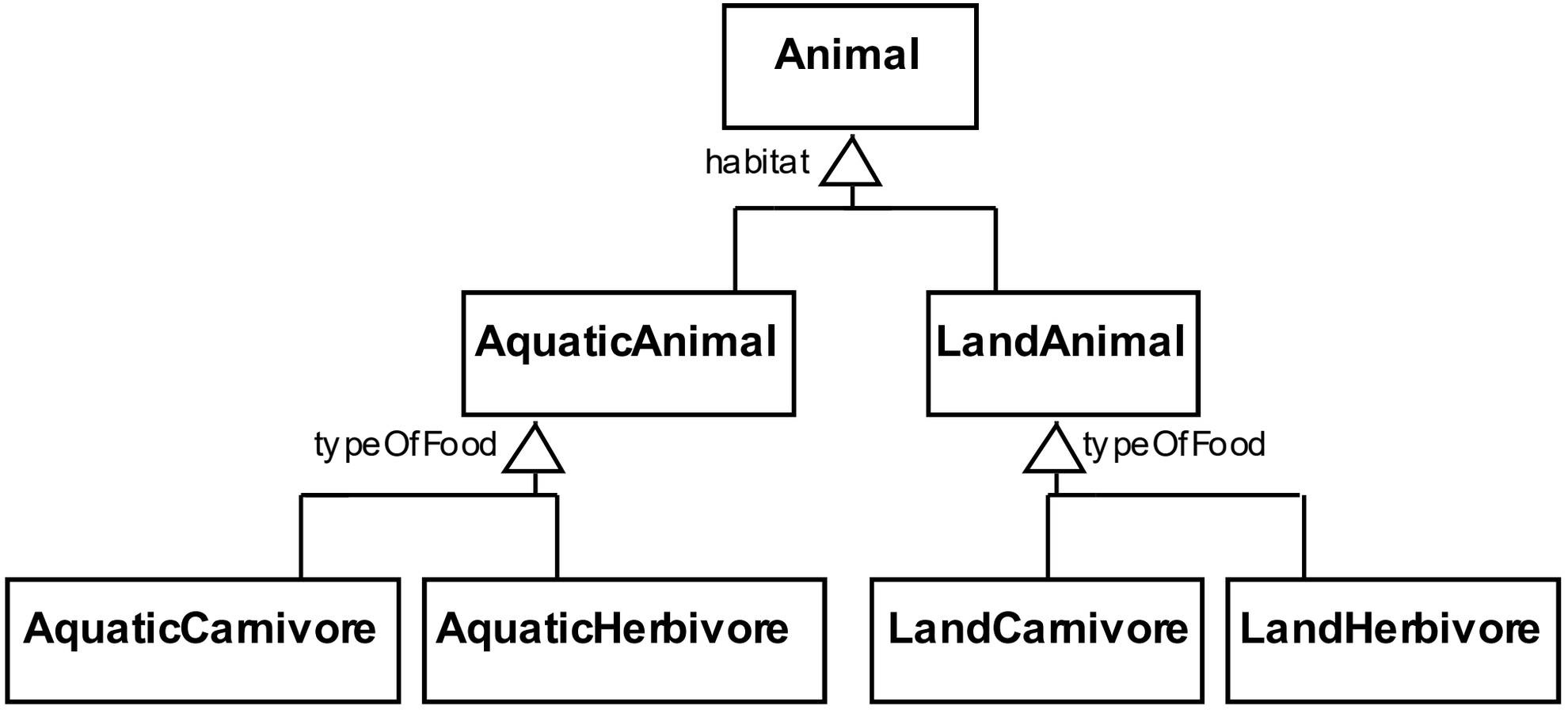

Animal
typeOfFood
Carnivore
Herbivore

Handling multiple discriminators by Creating higher-level generalization

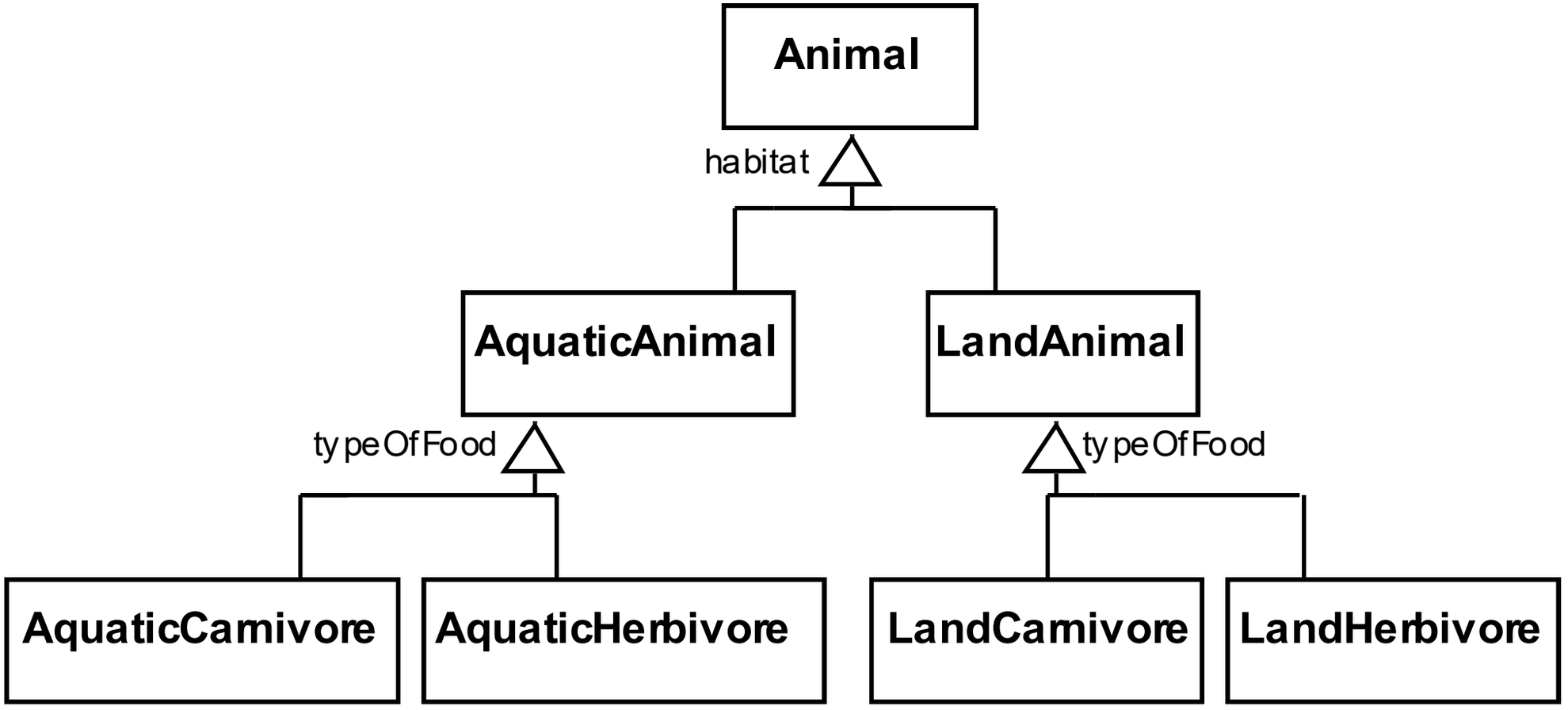




Properties associated with both discriminators have to be present



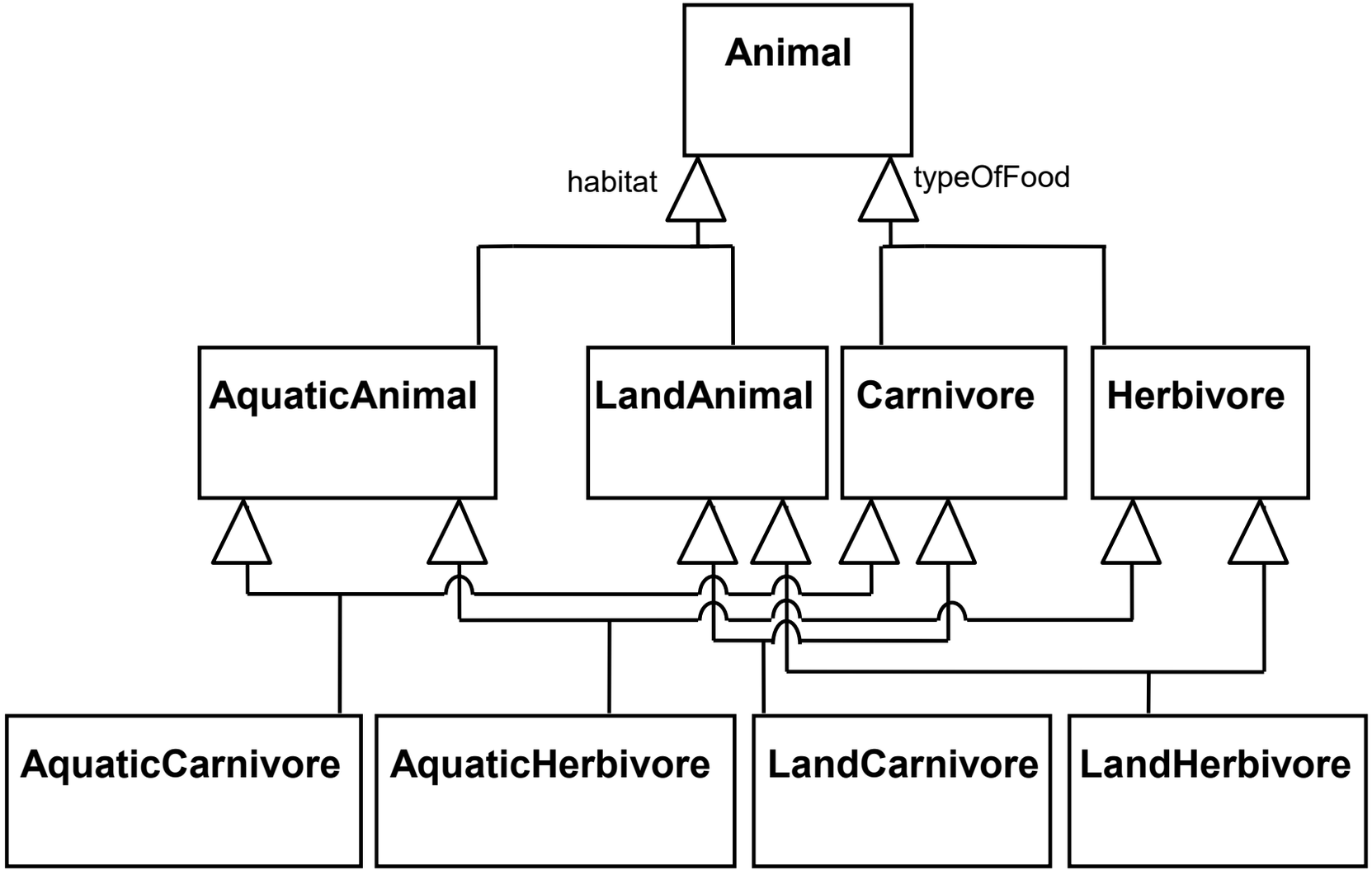
All the features associated with the second generalization set would have to be duplicated

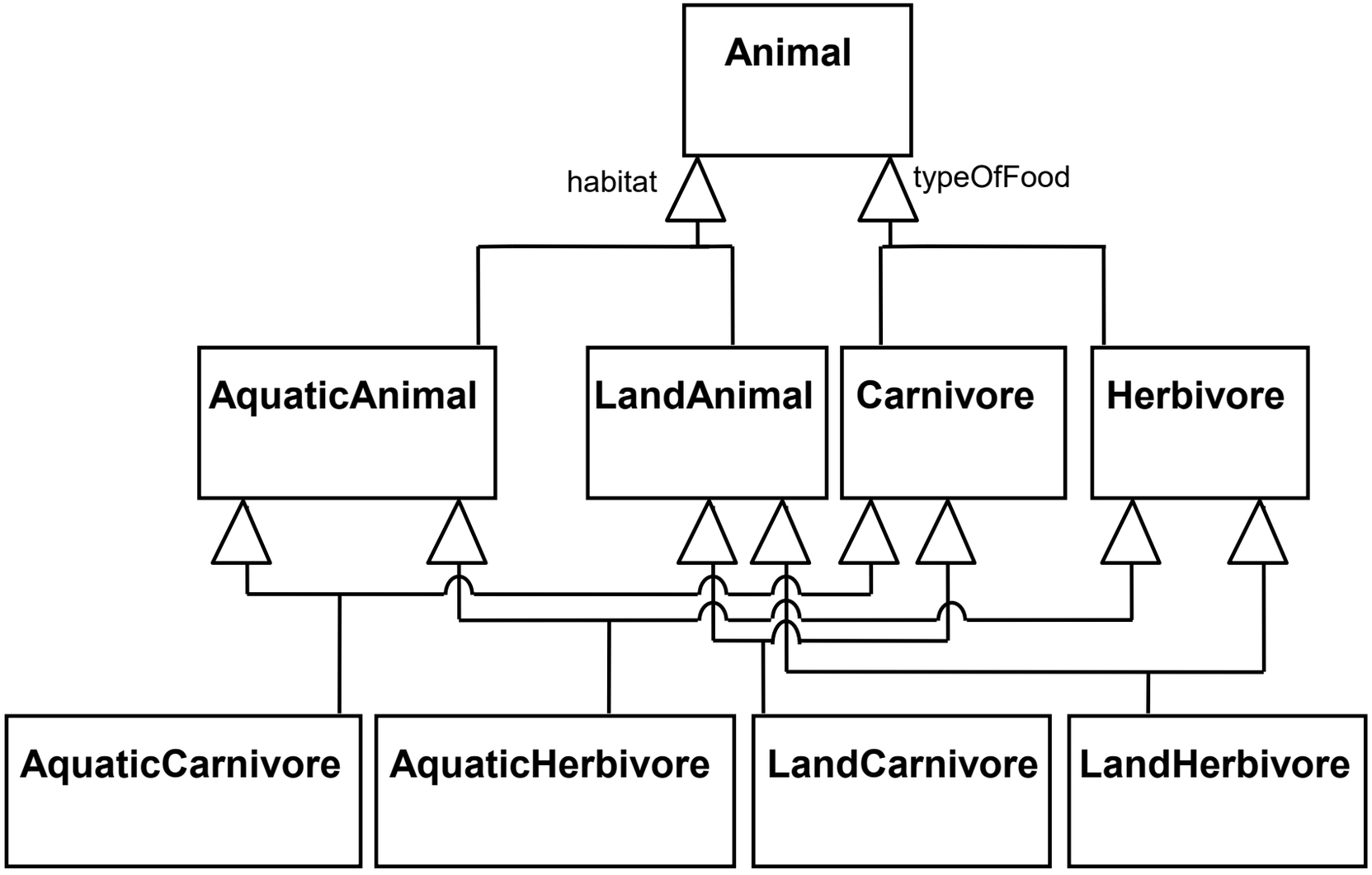


No. of classes can grow very large

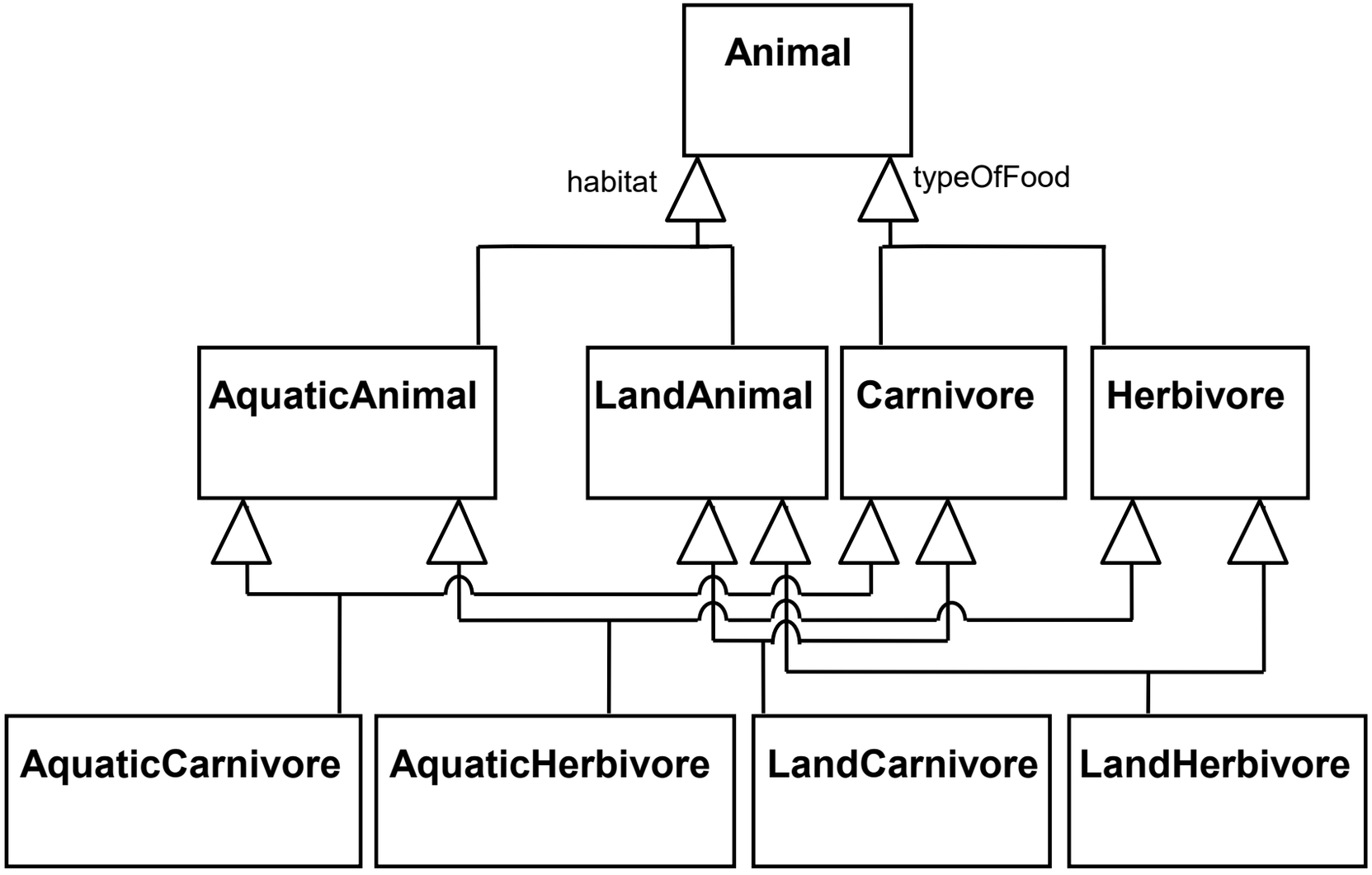
Try adding omnivores and/or amphibians!

Handling multiple discriminators by Using Multiple Inheritance





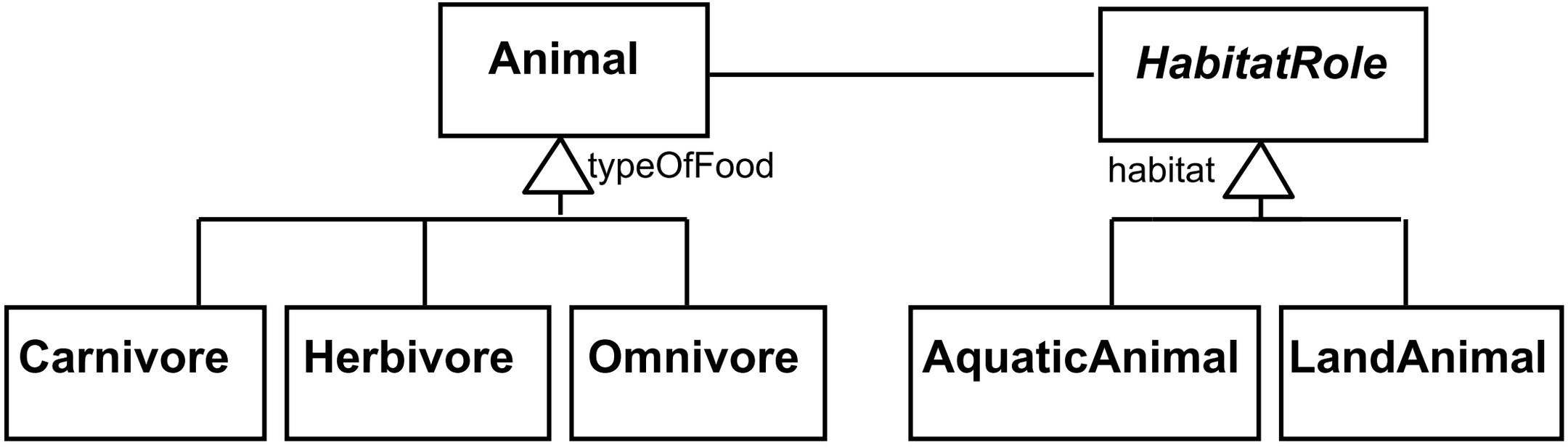
Avoids duplication of features



Adds too much complexity - Proliferation of classes

Not supported by many languages like Java and C#

Handling multiple discriminators by Using Player-Role Pattern



Can very easily add Amphibian

Software Design and Architecture

Topic#066

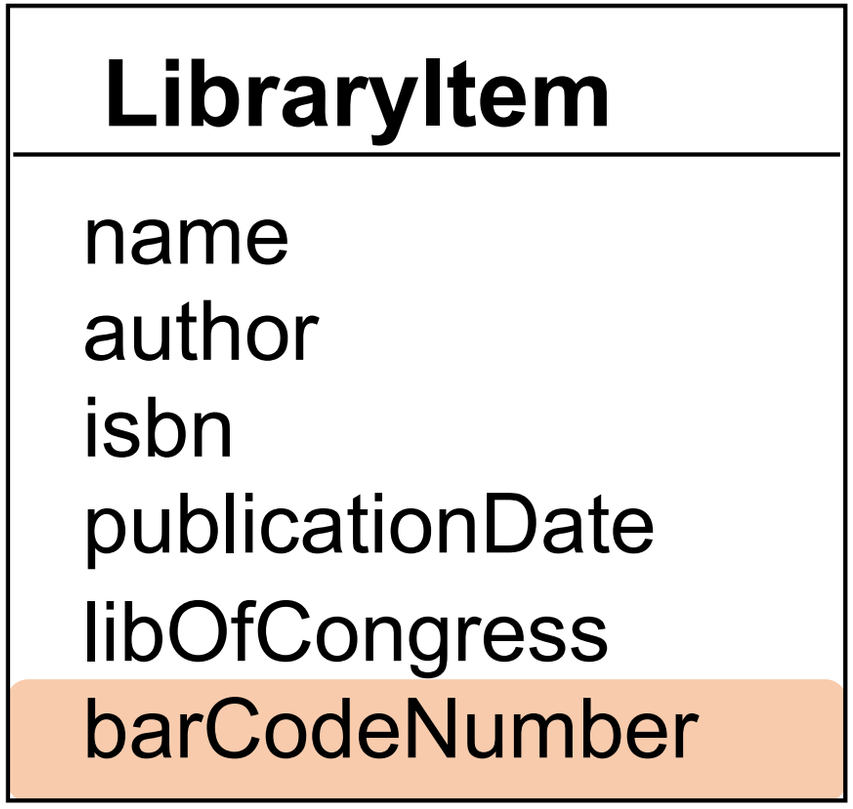
END!

Software Design and Architecture

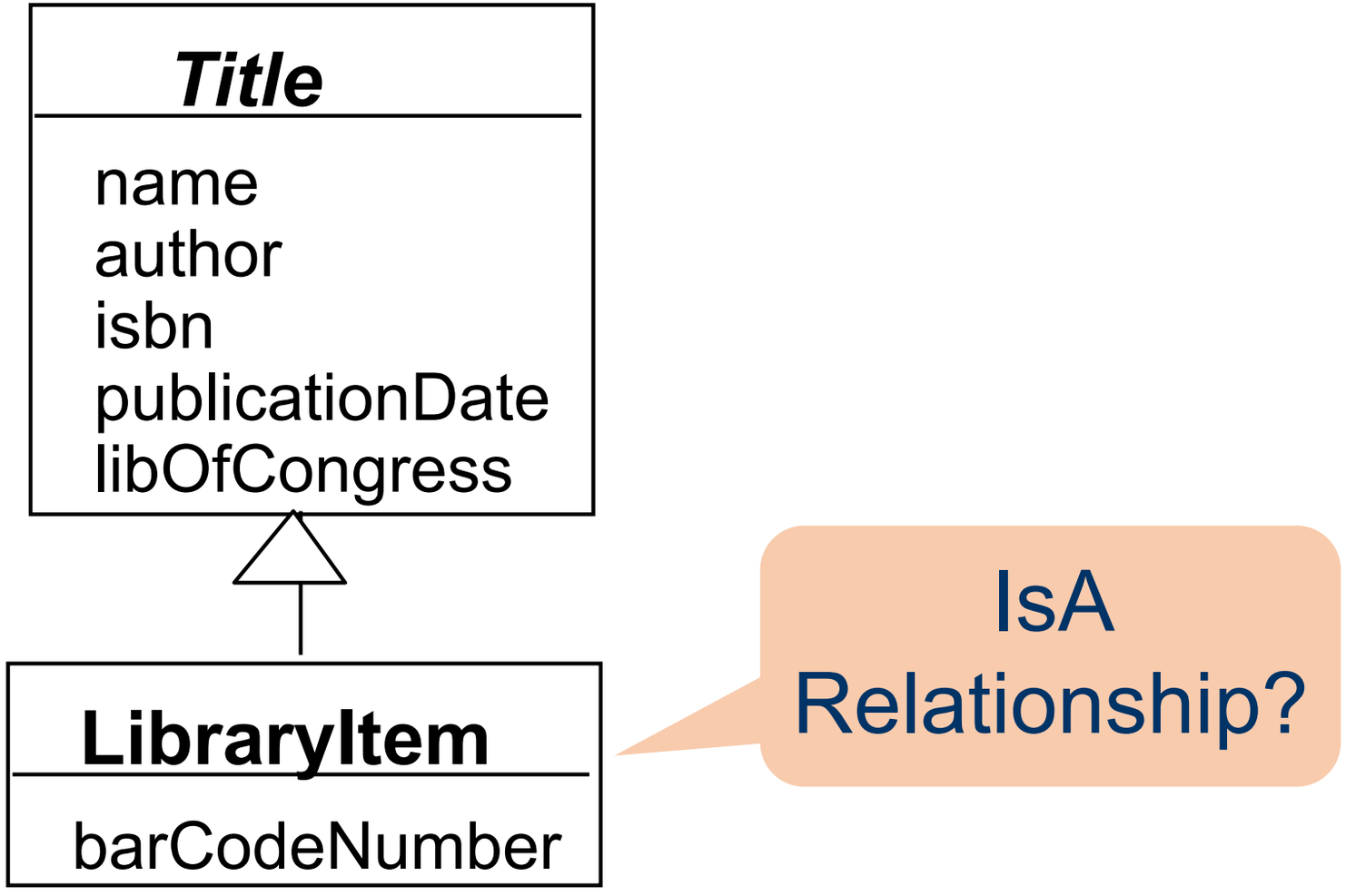
Topic#067
Abstraction-Occurrence Pattern

© Virtual University of Pakistan

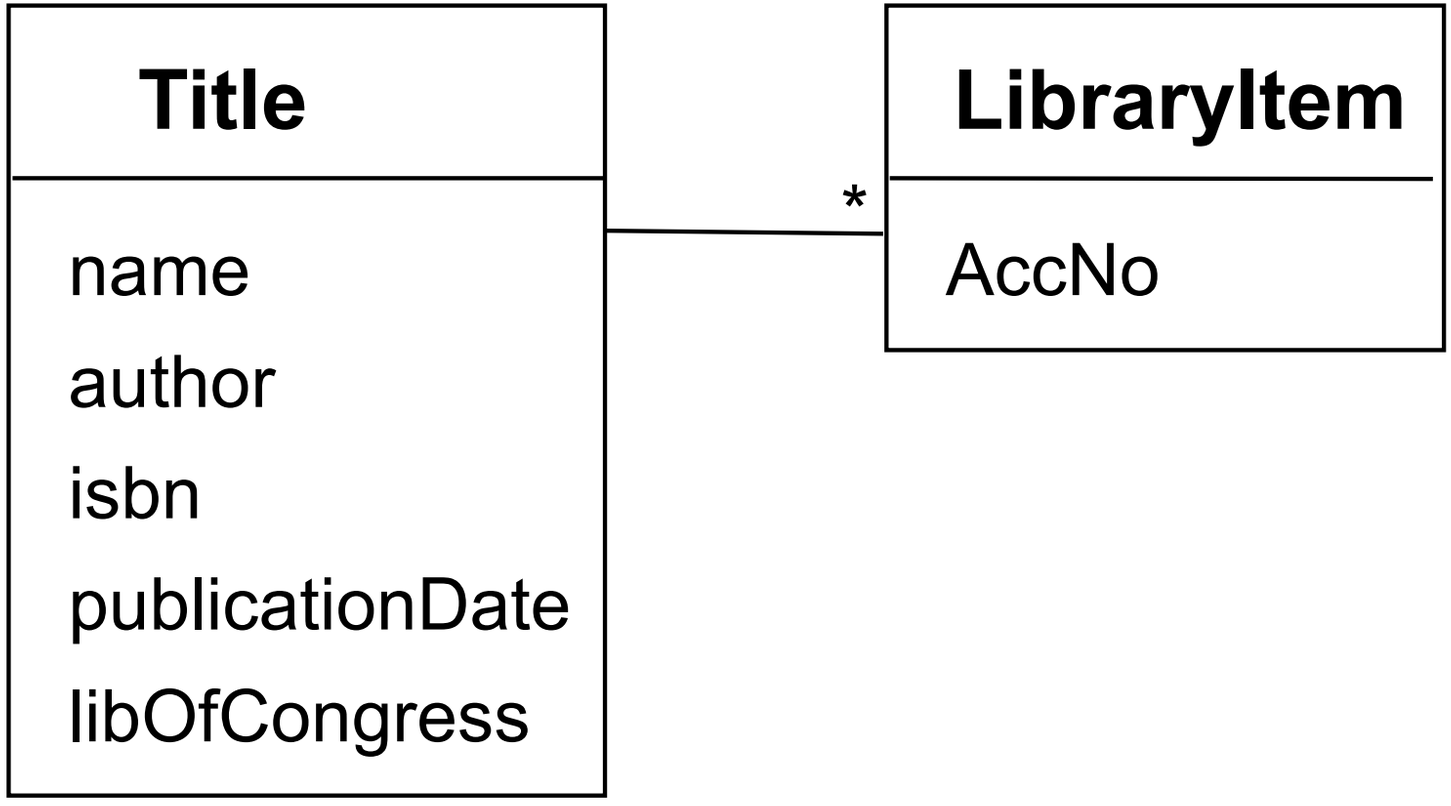
- Problem
  - A library has multiple copies of the same book.
  - How to model?



- Problem
  - A library has multiple copies of the same book.
  - How to model?



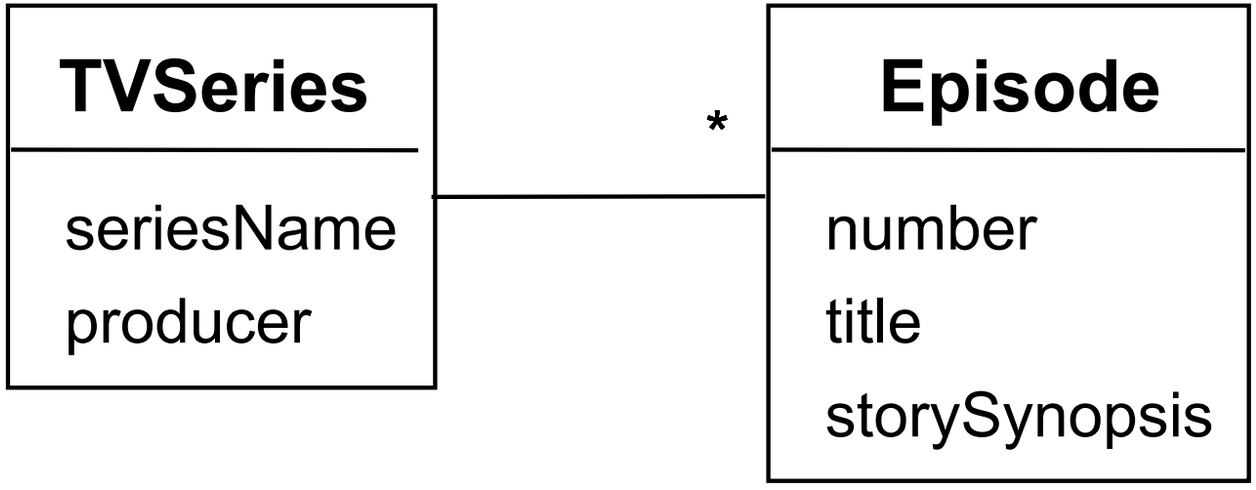
- Problem
  - A library has multiple copies of the same book.
  - How to model?



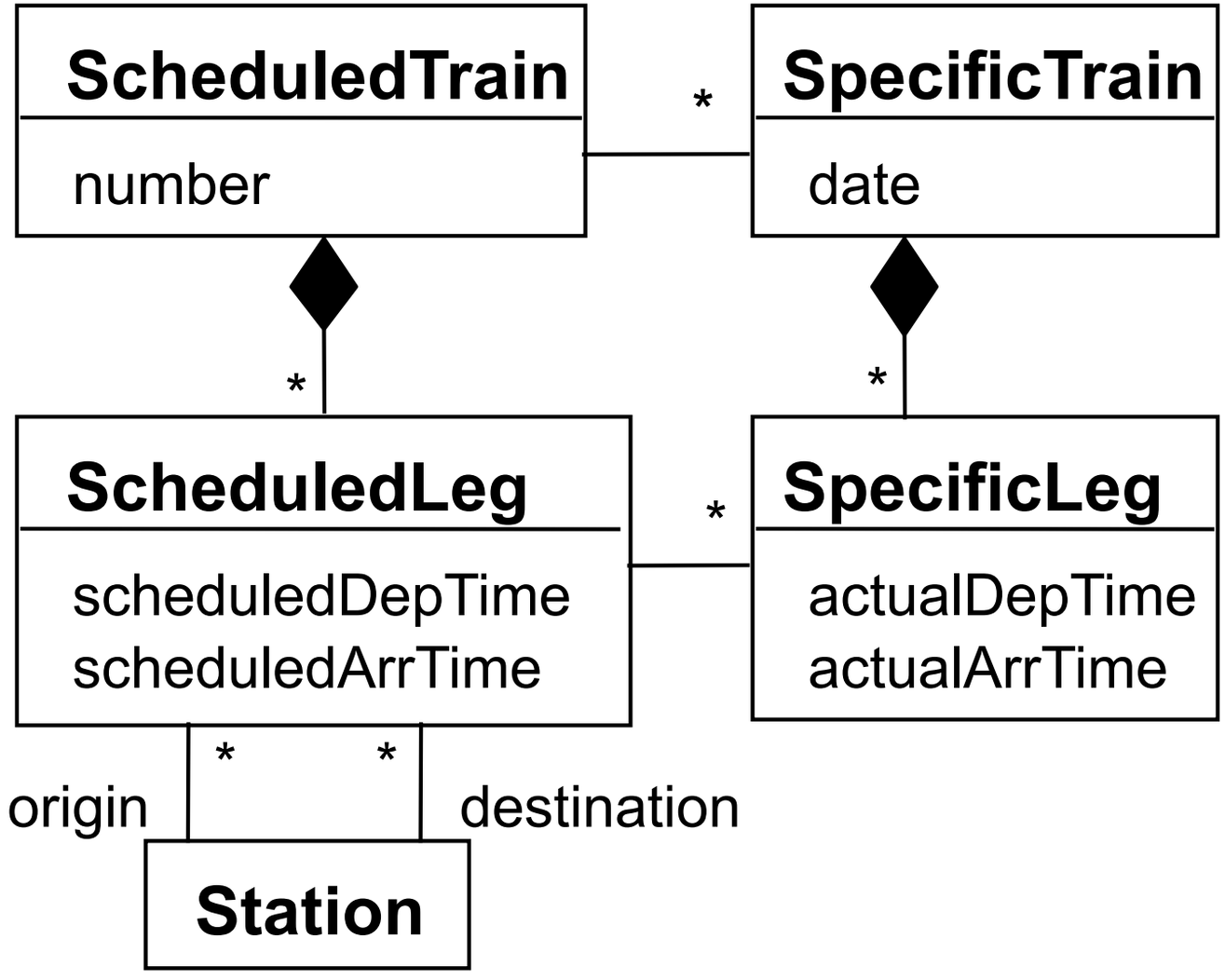
The Abstraction-Occurrence Pattern

- ***Context***:
  - Often in a domain model you find a set of related objects (*occurrences)*.
  - The members of such a set share common information
    - but also differ from each other in important ways.
- ***Problem***:
  - What is the best way to represent such sets of occurrences in a class diagram?
- ***Forces***:
  - You want to represent the members of each set of occurrences without duplicating the common information

Abstraction-Occurrence - Examples



Square variant



Software Design and Architecture

Topic#067

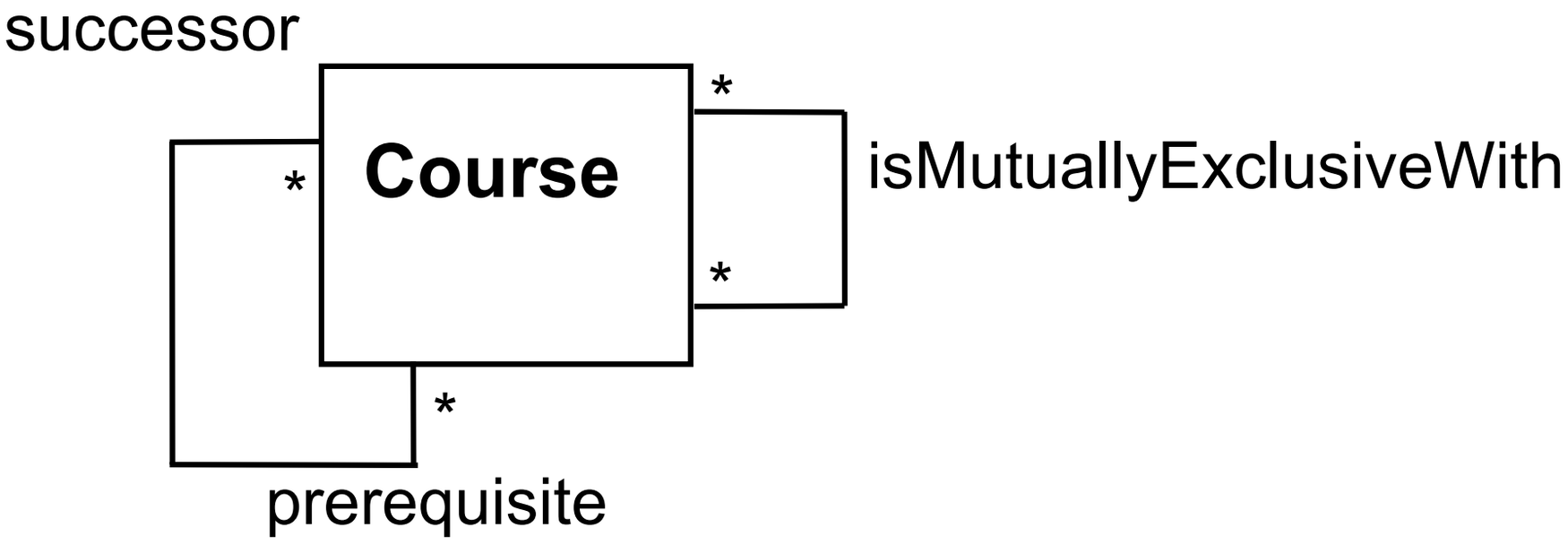
END!

Software Design and Architecture

Topic#068
Reflexive Associations

Reflexive associations

An association to connect a class to itself



Two main types: Symmetric and Asymmetric.

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

Asymmetric Reflexive Association

```
// Example - asymmetric association


class Person {
   Person *parents[2];
   …
};
```

Asymmetric Reflexive Association

```
// Example - asymmetric association with role name
// on the other end


class Person {
   Person *parents[2];
   vector <Person *> child;
   …
}
```

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

Symmetric Reflexive Association

```
// Example symmetric association.


class Course {
        Course  *mutuallyExclusiveWith[3];
};
```

Software Design and Architecture

Topic#068

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 069 – 075

Design Patterns

© Virtual University of Pakistan

Software Design and Architecture

Topic#069
Design Patterns – Introduction

- **Good OOD is more than just knowing and applying concepts like abstraction, inheritance, and polymorphism**

- **A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.**

© Virtual University of Pakistan

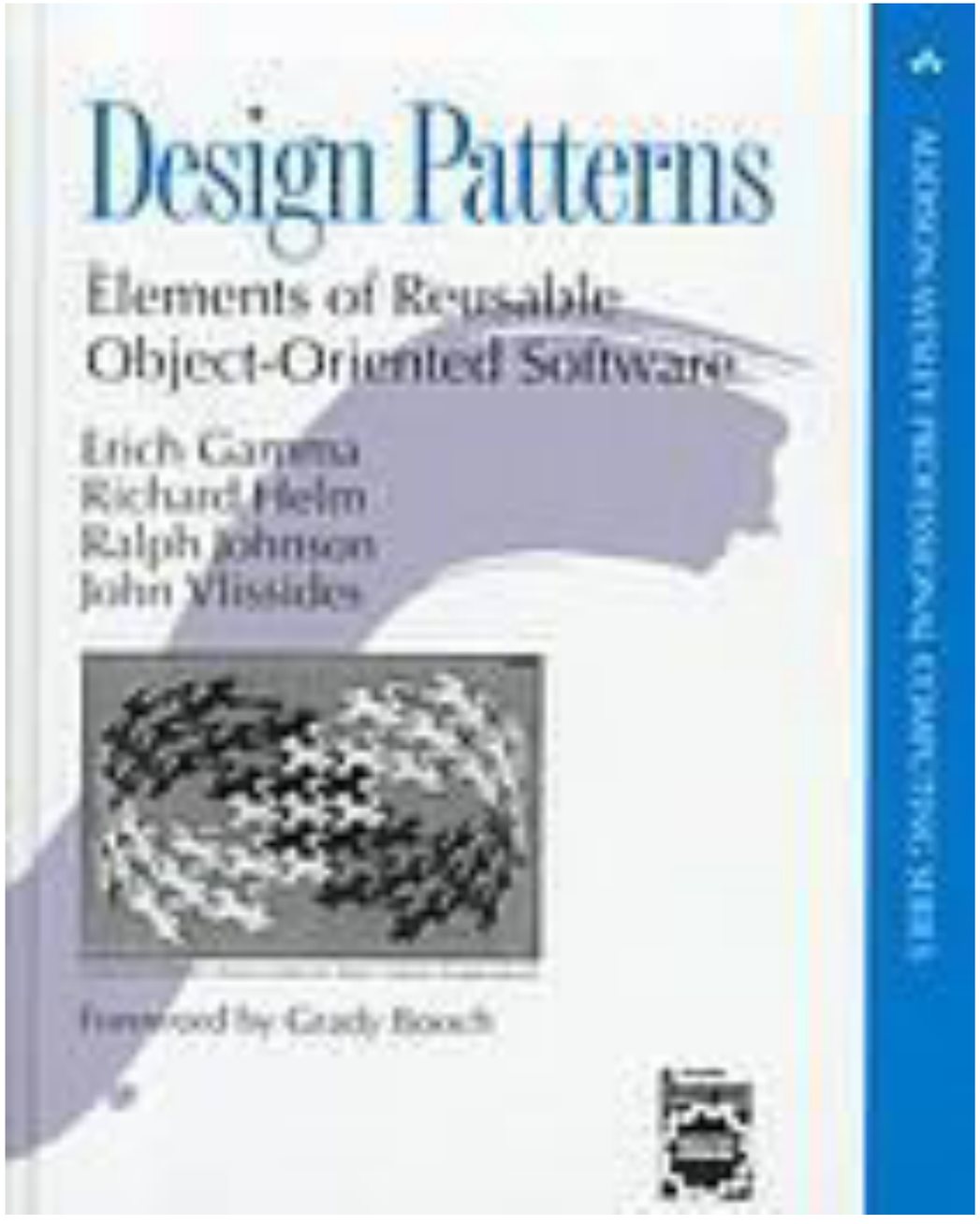
Design Patterns

"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

Software Design Patterns

- **Search for recurring successful designs – emergent designs from practice**
- **Described in Gama, Helm, Johnson, Vlissides 1995 (i.e., "gang of 4 book" aka GoF)**
- **Represent solutions to problems that arise when developing software within a particular context.**
- **Describes recurring design structures**
- **Describes the context of usage**


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

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**

© Virtual University of Pakistan

- **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.**

Software Design and Architecture

Topic#069

END!

Software Design and Architecture

Topic#070
Elements of design patterns

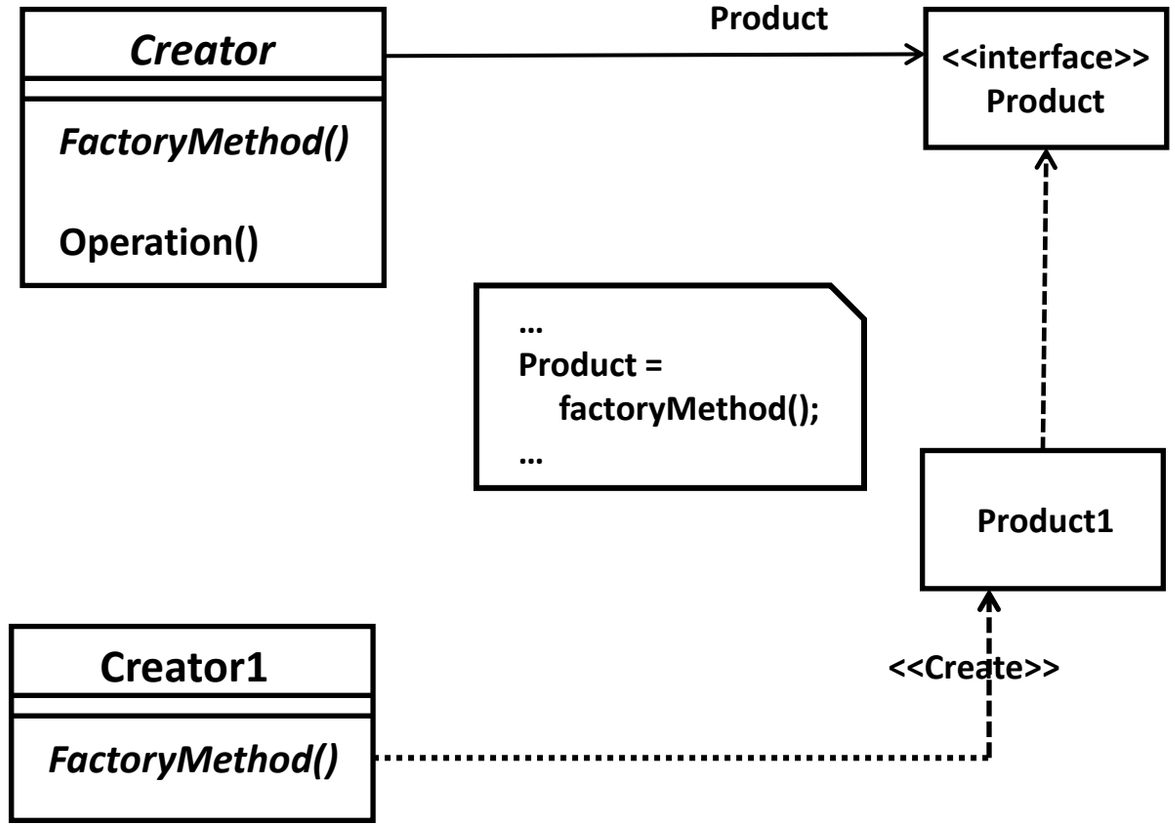
Elements of Design Patterns

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

Pattern Name

- **A handle used to describe:**
  - **a design problem**
  - **its solutions**
  - **its consequences**
- **Increases design vocabulary**
- **Makes it possible to design at a higher level of abstraction**
- **Enhances communication**
- ***"The Hardest part of programming is coming up with good variable [function, and type] names."***

Factory Method



Problem

- **Describes when to apply the pattern**
  - **Explains the problem and its context**
  - **May describe specific design problems and/or object structures**
  - **May contain a list of preconditions that must be met before it makes sense to apply the pattern**

Solution

- **Describes the elements that make up the**
  - **design**
  - **relationships**
  - **responsibilities**
  - **collaborations**
- **Abstract description of design problems and how the pattern solves it**
- **Does not describe specific concrete implementation**

Consequences

- **Results and trade-offs of applying the pattern**
- **Critical for:**
  - **evaluating design alternatives**
  - **understanding costs**
  - **understanding benefits of applying the pattern**
- **Includes the impacts of a pattern on a system's:**
  - **flexibility**
  - **extensibility**
  - **portability**

Design Pattern Descriptions

- **Name and Classification**: Essence of pattern
- **Intent**: What it does, its rationale, its context
- **AKA**: Other well-known names
- **Motivation**: Scenario illustrates a design problem
- **Applicability**: Situations where pattern can be applied
- **Structure**: Class and interaction diagrams
- **Participants**: Objects/classes and their responsibilities
- **Collaborations**: How participants collaborate
- **Consequences**: Trade-offs and results
- **Implementation**: Pitfalls, hints, techniques, etc.
- **Sample Code**
- **Known Uses**: Examples of pattern in real systems
- **Related Patterns**: Closely related patterns

Software Design and Architecture

Topic#070

END!

Software Design and Architecture

Topic#071
Categories of Design Patterns

Categories of Patterns

- **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**

Pattern Types

- **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**

		Purpose		
		Creational	Structural	Behavioural
Scope	Class	• Factory method	• Adapter (class)	• Interpreter • Template method
	Object	• Abstract factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

© Virtual University of Pakistan

Software Design and Architecture

Topic#071

END!

Software Design and Architecture

Topic#072
Benefits and drawbacks of design patterns

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**

© Virtual University of Pakistan

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.**

© Virtual University of Pakistan

Design Patterns are NOT

- **Designs that can be encoded in classes and reused as is (i.e., linked lists, hash tables)**
- **Complex domain-specific designs (for an entire application or subsystem)**

- **They are:**
  - **“Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.”**

Software Design and Architecture

Topic#072

END!

Software Design and Architecture

Topic#073
Singleton Design Pattern

The Singleton Pattern

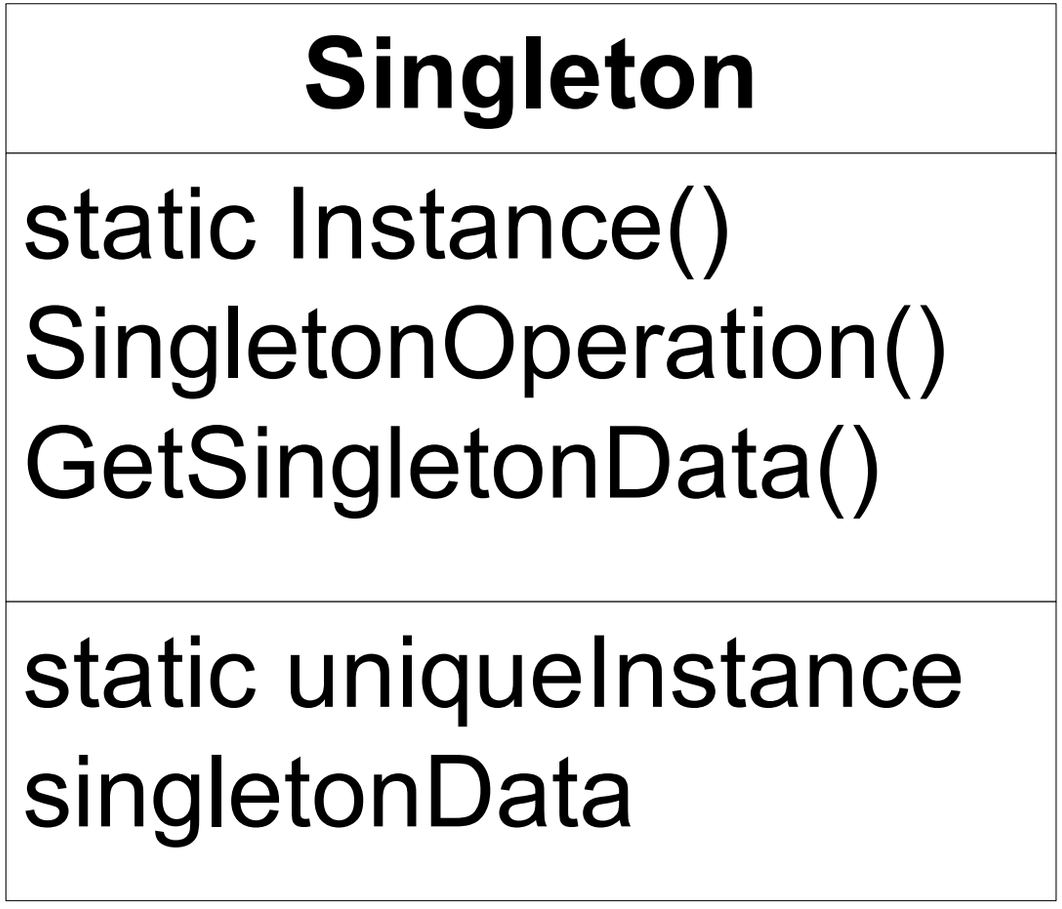
- ***Context***:
  - It is very common to find classes for which only one instance should exist (*singleton)*
- ***Problem***:
  - How do you ensure that it is never possible to create more than one instance of a singleton class?
- ***Forces***:
  - The use of a public constructor cannot guarantee that no more than one instance will be created.
  - The singleton instance must also be accessible to all classes that require it

Singleton - Uses

Use the Singleton when:

- **There must be exactly one (or fixed number) instance of the class and it must be accessible to clients from a well-known access point**

Singleton - Structure



Implementation

```
Class Singleton {
public:
  static singleton* Instance();
protected:
  Singleton();
private:
  static singleton* _instance;
};
```

```
Singleton* Singleton::
                        _instance= NULL;

Singleton* Singleton::Instance() {
    if (_instance == NULL) {
       _instance = new Singleton;
    }
    return _instance;
}
```

Software Design and Architecture

Topic#073

END!

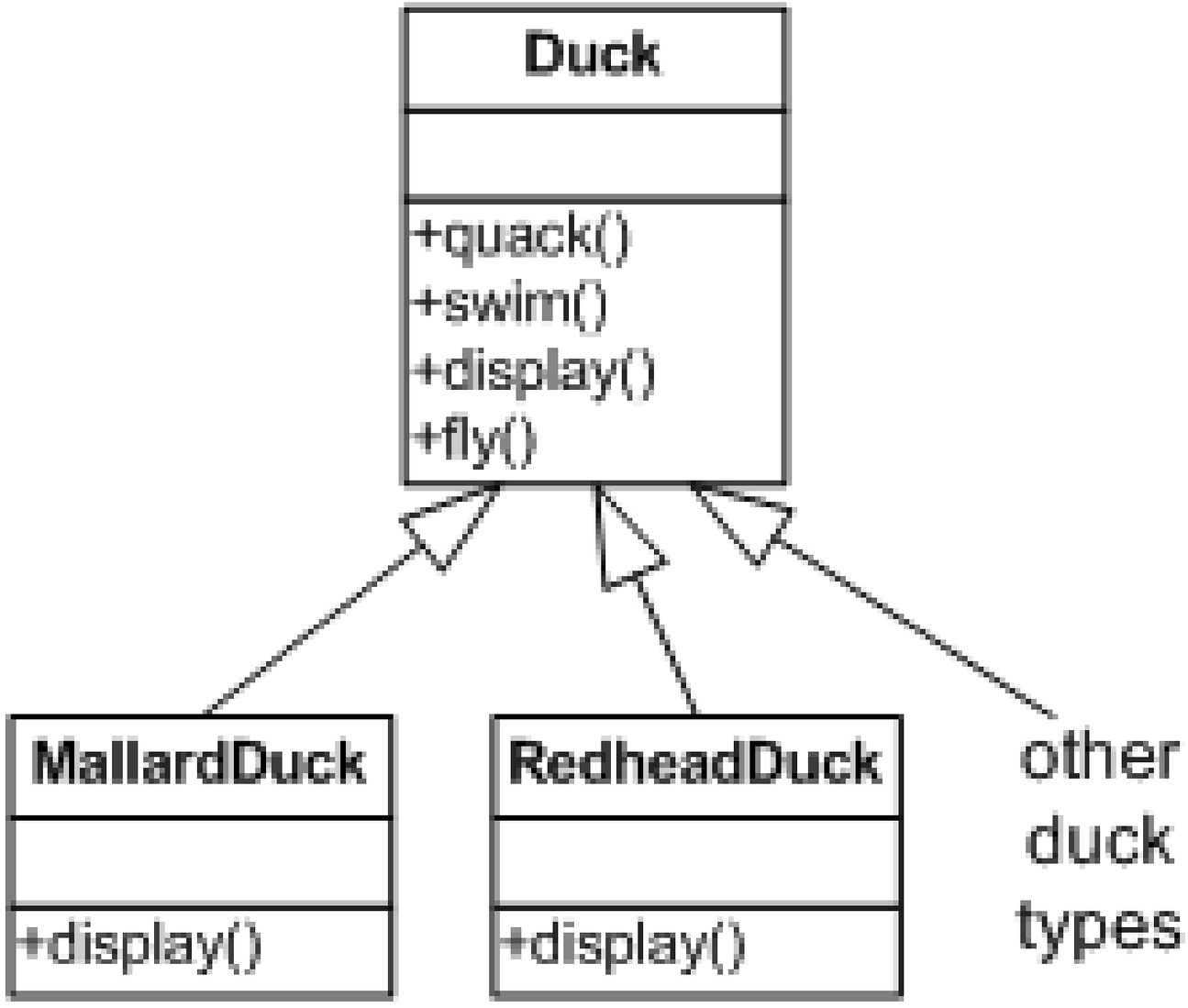
Software Design and Architecture

Topic#074
Strategy Design Pattern

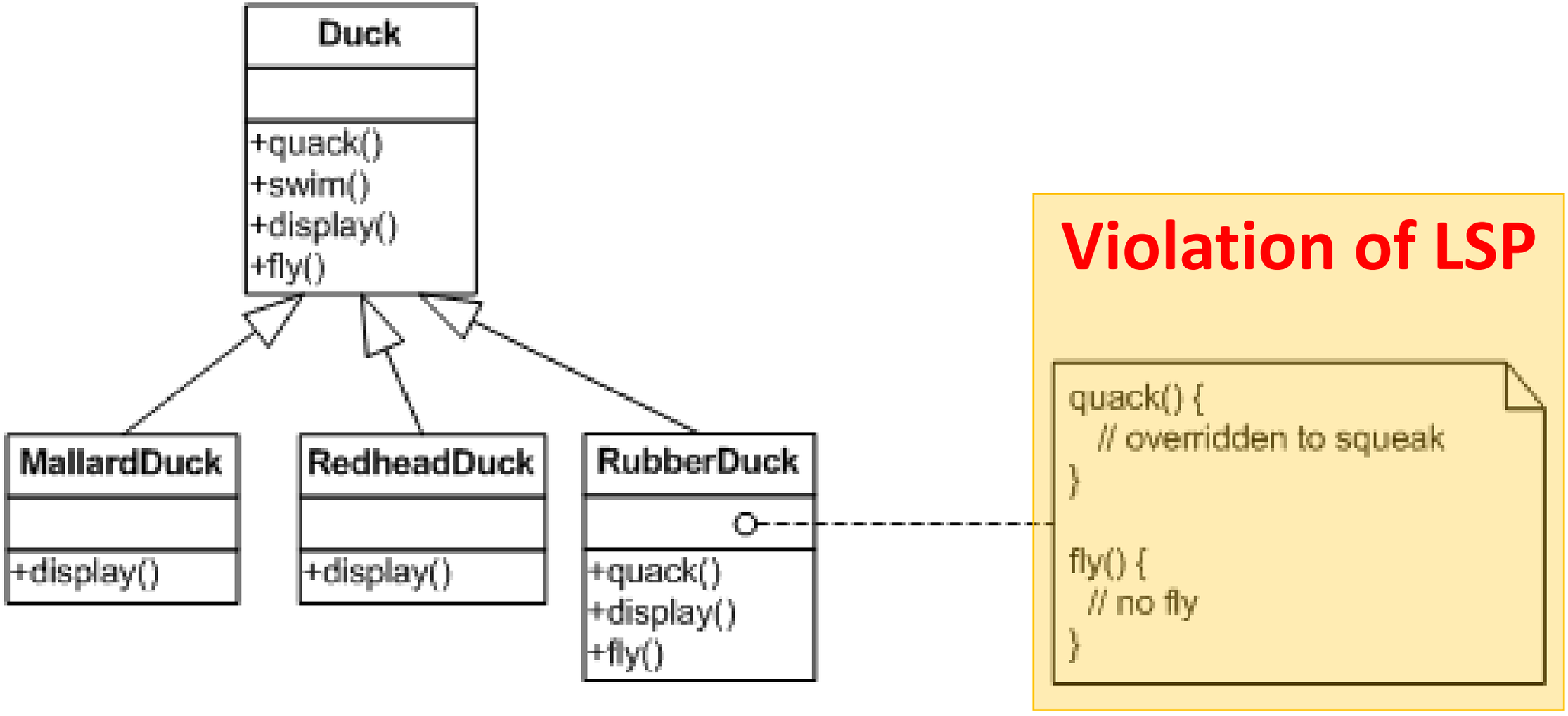
SimDuck

Add fly()

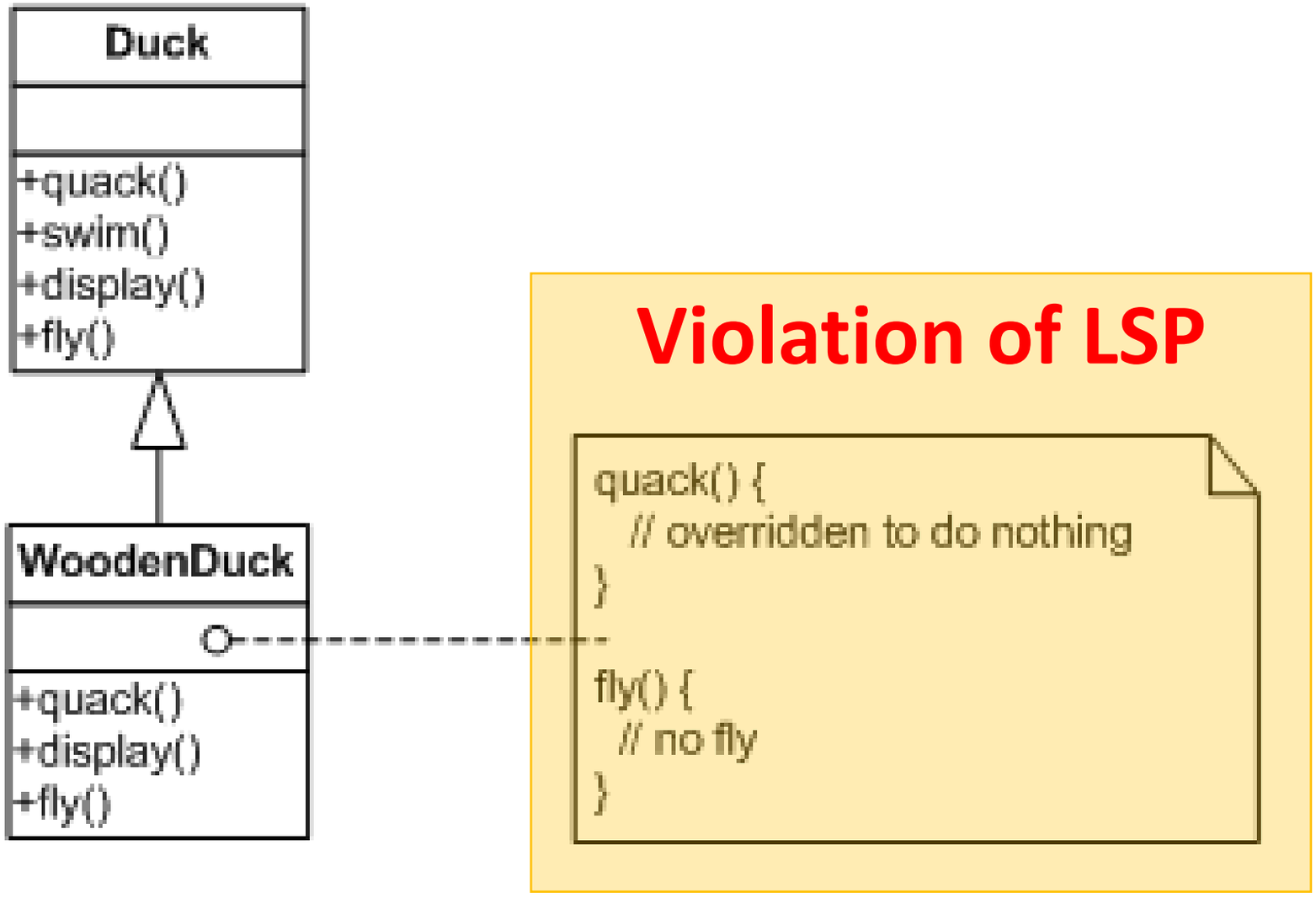
- **What about a rubber duck?**



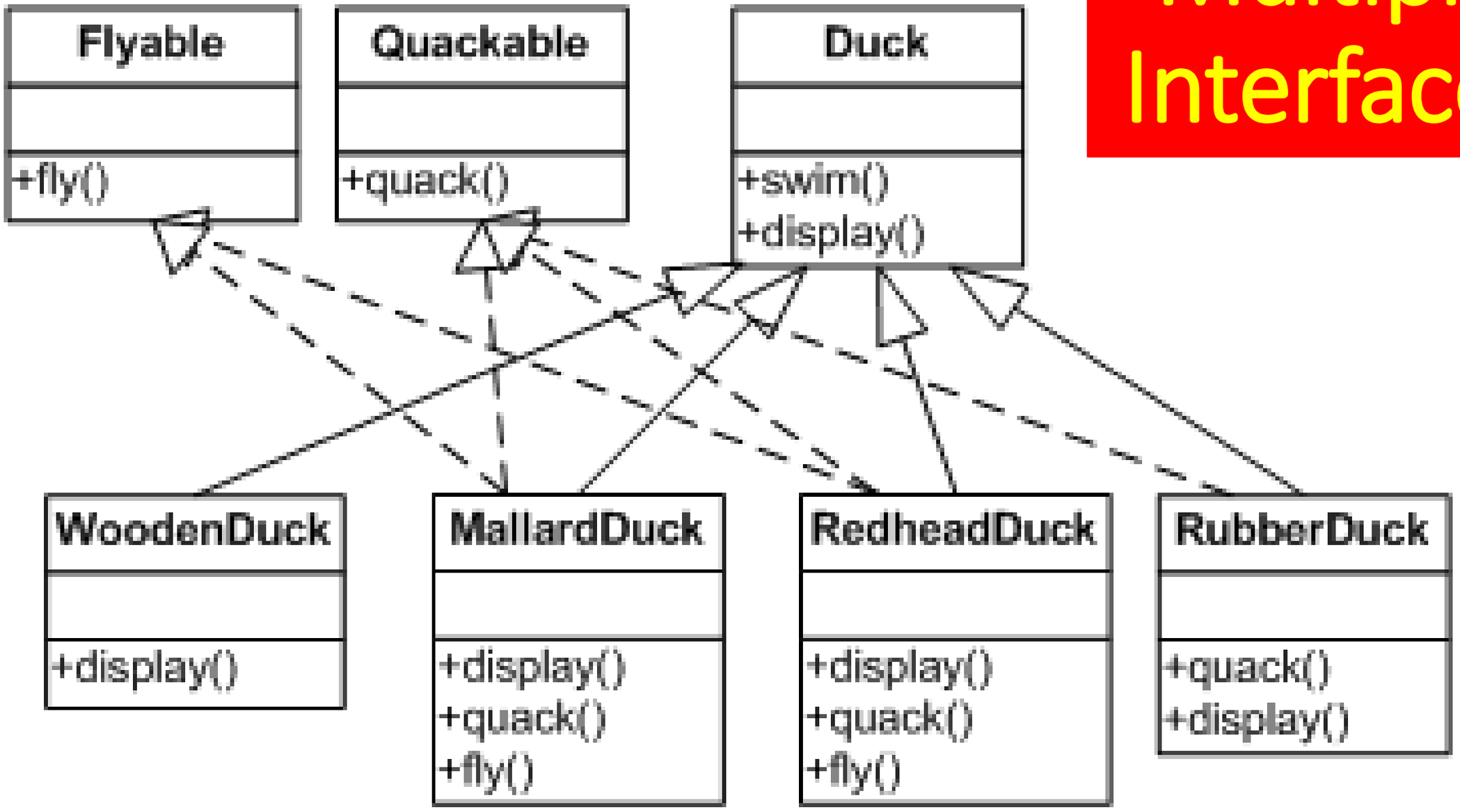
Rubber Duck


Duck
+quack()
+swim()
+display()
+fly()
MallardDuck
+display()
RedheadDuck
+display()
RubberDuck
+quack()
+display()
+fly()
Violation of LSP
quack() {
// overridden to squeak
}
fly() {
// no fly
}

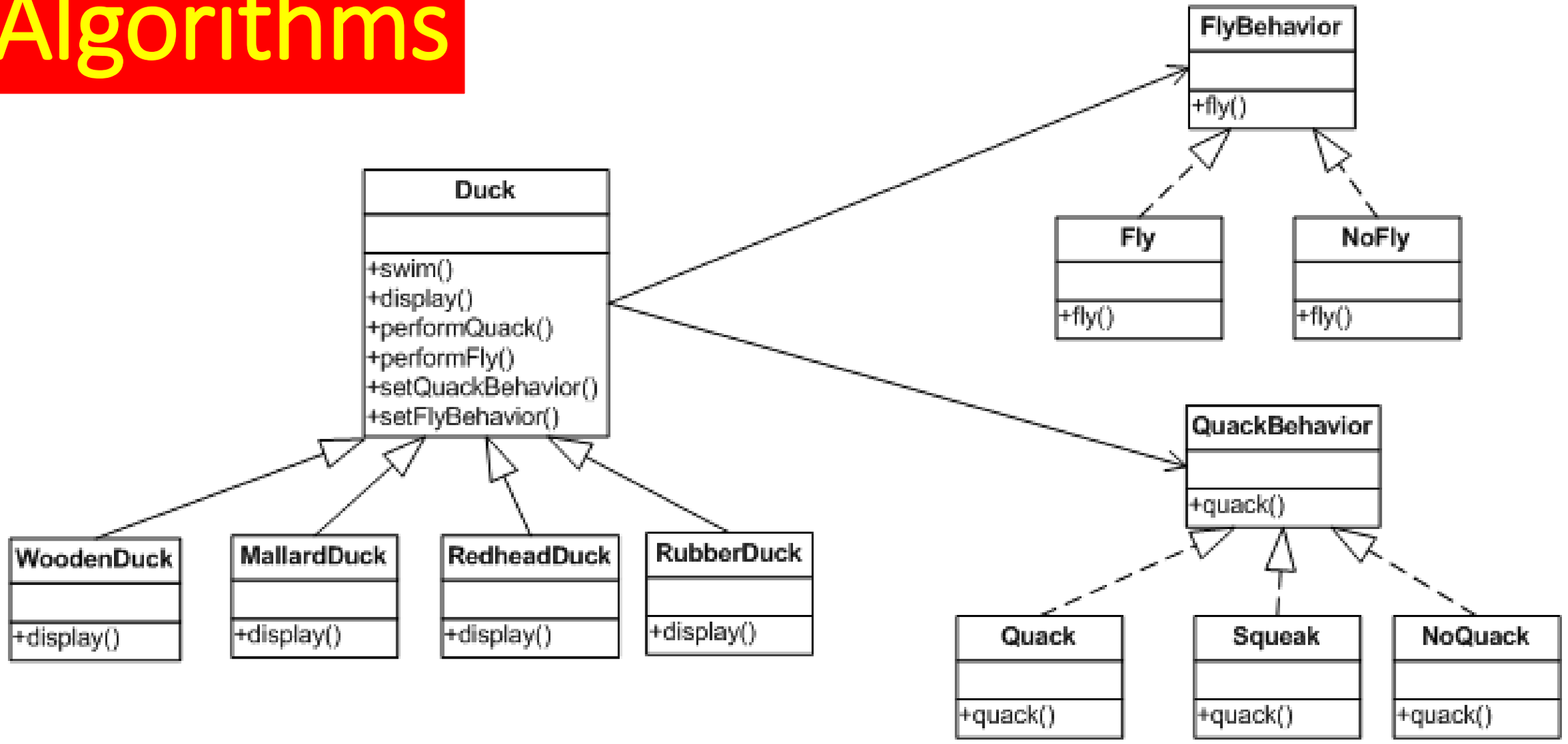
Another duck



Multiple Interfaces



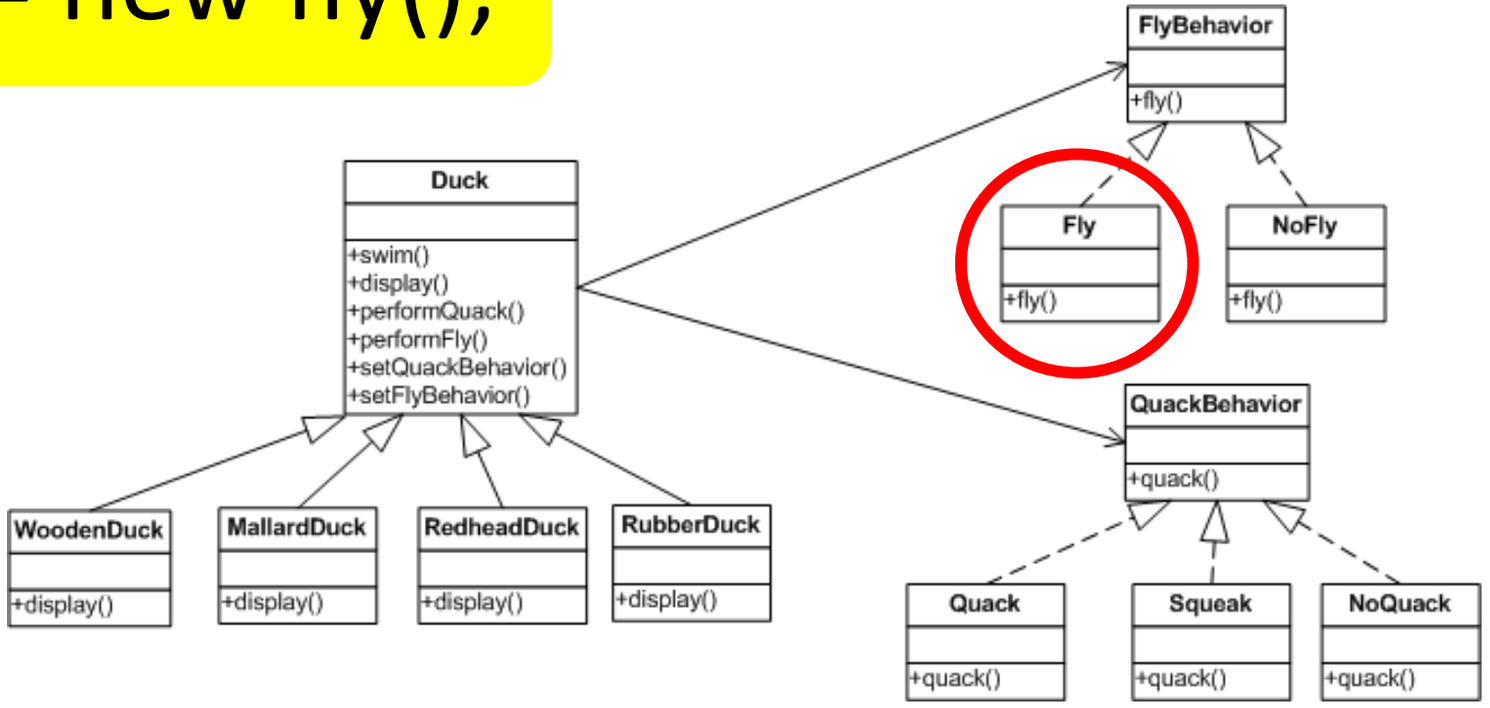
Associate Algorithms



© Virtual University of Pakistan

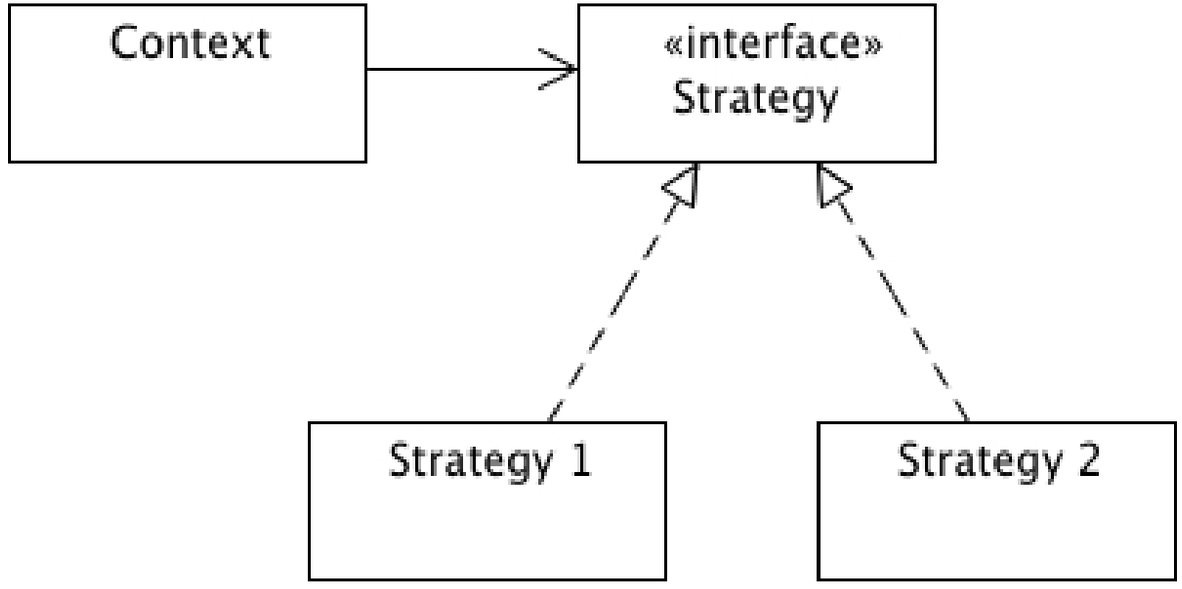
```
public abstract class Duck {
    FlyBehavior   flyBehavior;
    QuackBehavior quackBehavior;
    public Duck() { }
    public abstract void display();
    public void performFly() {
        flyBehavior.fly();
    }
    public void performQuack() {
        quackBehavior.quack();
    }
    // other
```

```
public class MallardDuck extends Duck {
  public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new fly();
  }
  // other
}
```



Strategy Pattern

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



© Virtual University of Pakistan

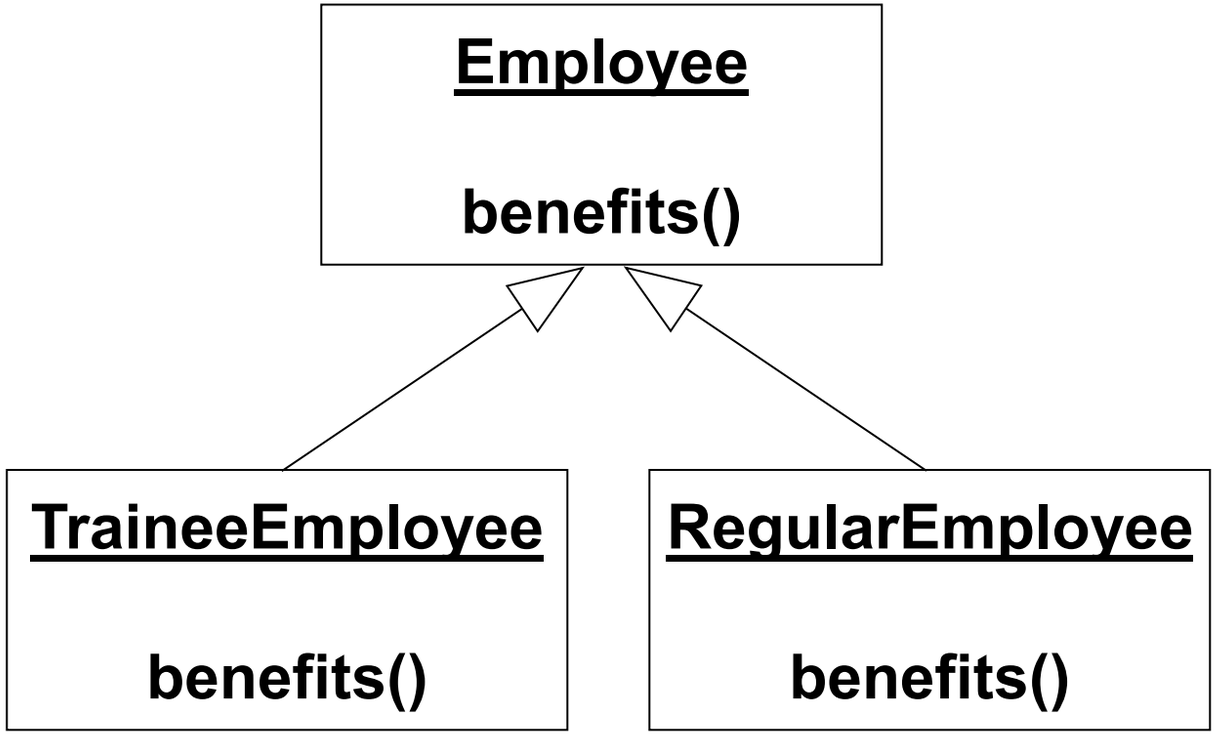
Software Design and Architecture

Topic#074

END!

Software Design and Architecture

Topic#075
Object-Morphing and Strategy Pattern



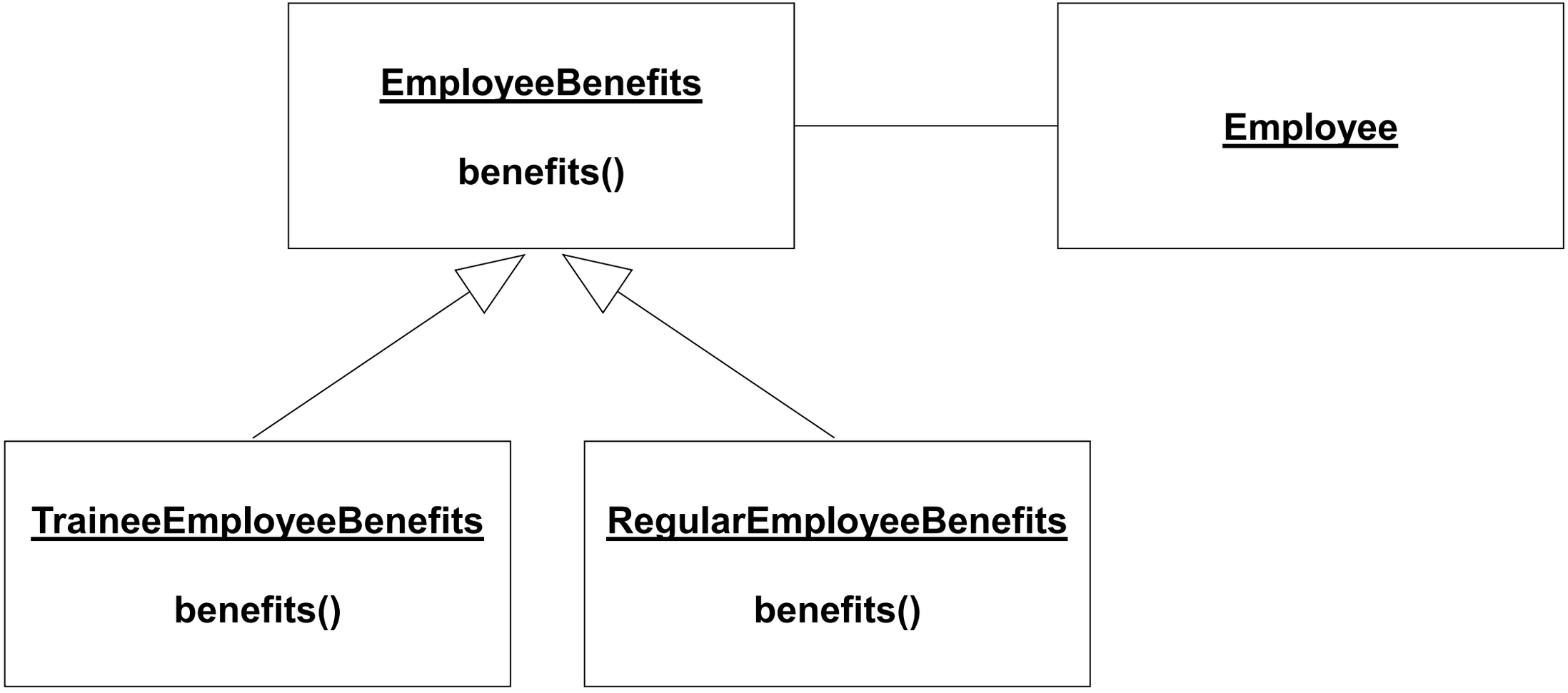
Everything seems satisfactory until we think about the life of the traineeEmployee object

Object Morphing

Avoiding having instances change class

- **Object Morphing**
  - **When an instance changes its class**
  - **An instance should never need to change class**

- **You have to destroy the old one and create a new one.**
  - **What happens to associated objects?**
    - **You have to make sure that all links that connected to the old object now connect to the new one.**



Object Morphing and Strategy Design Pattern

Software Design and Architecture

Topic#075

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 076 – 082

Design Patterns – Part II

© Virtual University of Pakistan

Software Design and Architecture

Topic#076
Façade Design Pattern

Problem

- There is a legacy application written in C
- There is a rich library of functionality that is very difficult to be re-written
- A new GUI-based interface needs to be added to the application

Problem

The application needs to be “object-orientized” to make it communicate/compatible with the rest of a newly built system.

How to go about it?

Facade Pattern

- **Structural Pattern**
- **Provides a unified interface to a set of interfaces in a subsystem.**
- **Defines a higher-level interface that makes the subsystem easier to use.**
- **Wraps a complicated subsystem with a simpler interface.**

Facade

- The principal front of a building, that faces on to a street or open space
- A deceptive outward appearance

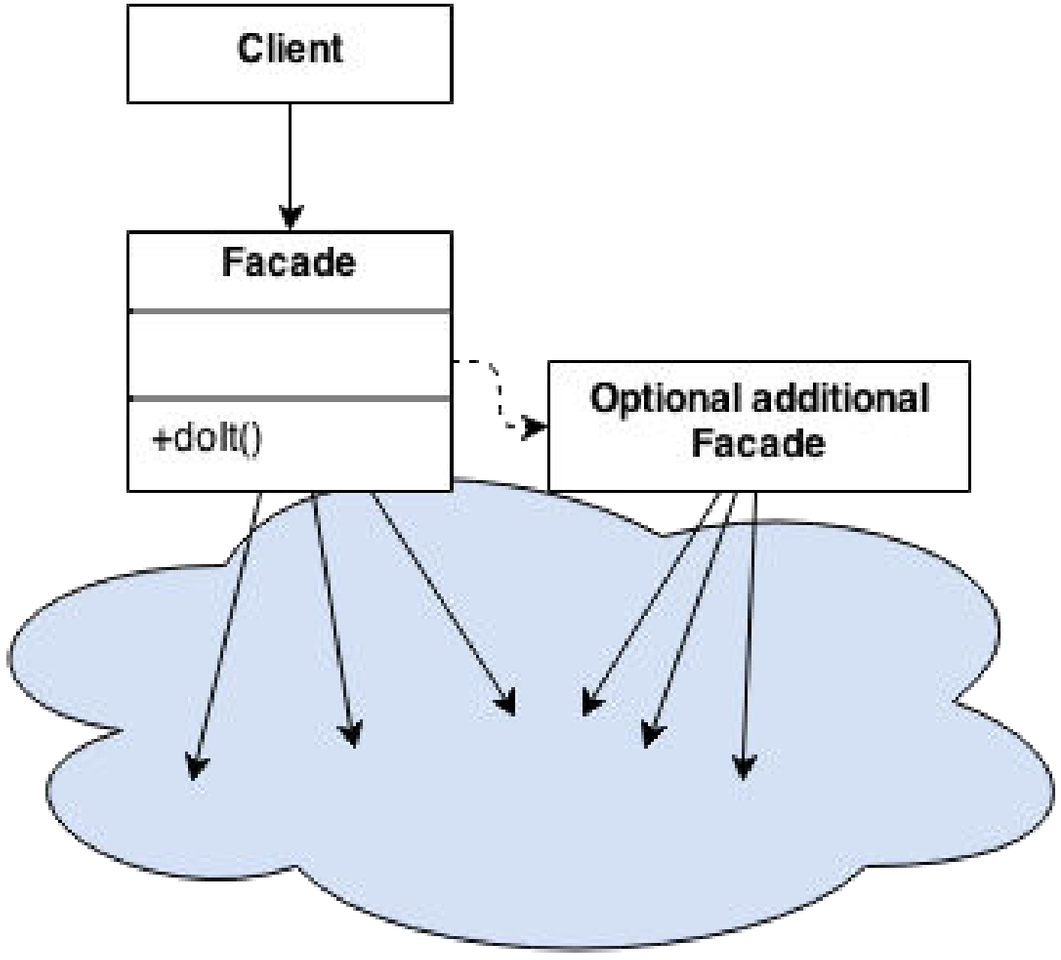
© Virtual University of Pakistan



General-building-façade: From Wikimedia Commons

Facade Pattern

- **Involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.**
- **Can be used to add an interface to existing system to hide its complexities.**



Facade objects are often Singletons because only one Facade object is required.

Software Design and Architecture

Topic#076

END!

Software Design and Architecture

Topic#077
Modeling Sate Machine

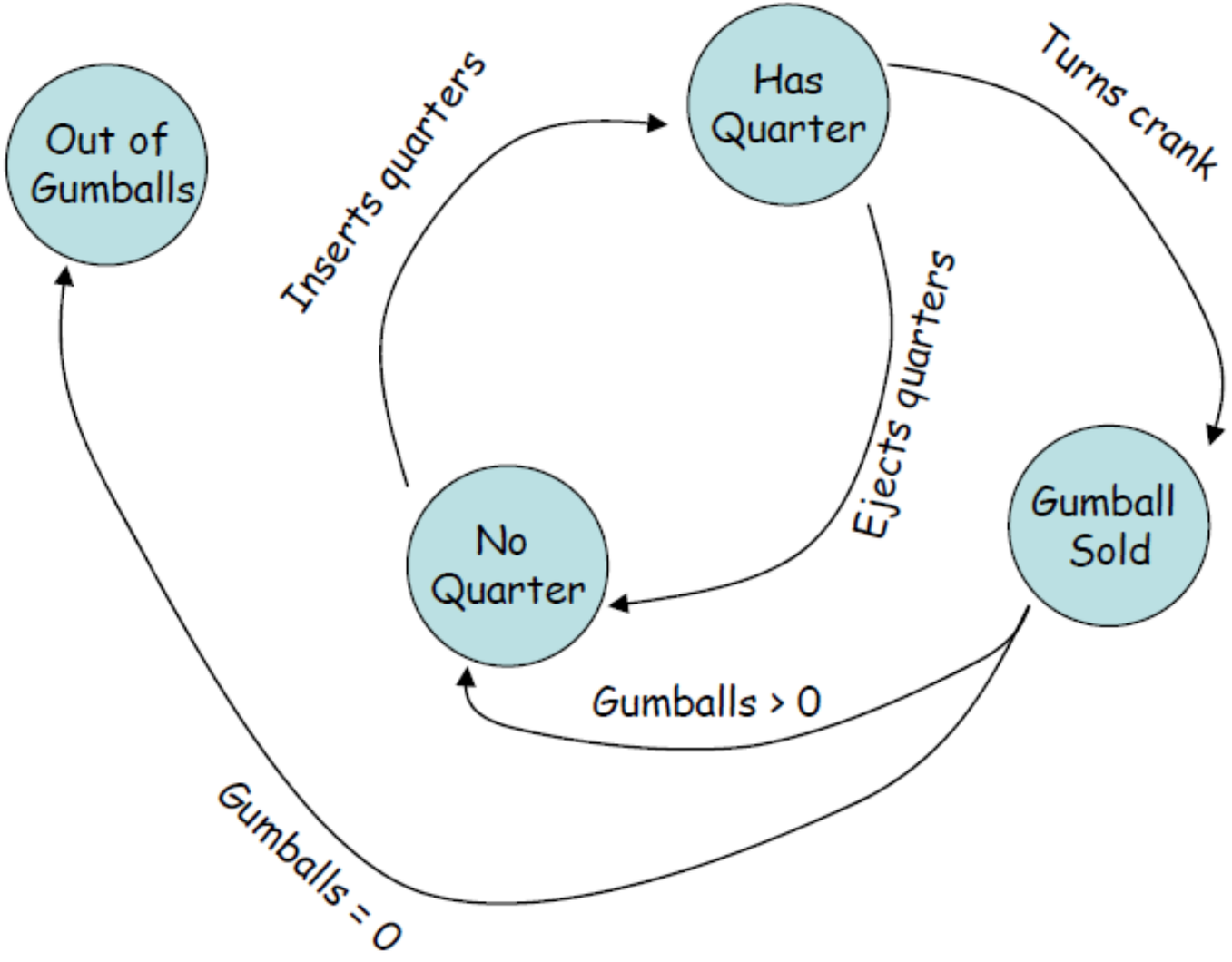
© Virtual University of Pakistan

Gumball Machine

By putting CPUs into their machines, they can increase sales, monitor inventory over the network and measure customer satisfaction more accurately.



Gumball Machine – Sate Diagram





Implementing State Machines
Step 1: Identify states

1. SOLD_OUT
2. NO_QUARTERS
3. HAS_QUARTERS
4. SOLD

© Virtual University of Pakistan

Implementing State Machines
Step 2: Create instance variables to hold state

```
final static int SOLD_OUT        = 0;
final static int NO_QUARTERS     = 1;
final static int HAS_QUARTERS    = 2;
final static int SOLD            = 3;
```

Implementing State Machines
Step 3: Identify initial state

```
int state = SOLD_OUT;
int count = 0;
```

Implementing State Machines
The Gumball Machine Class so far

```
public class GumballMachine {

    final static int SOLD_OUT          = 0;
    final static int NO_QUARTERS       = 1;
    final static int HAS_QUARTERS      = 2;
    final static int SOLD              = 3;

    int state = SOLD_OUT;
    int count = 0;
```

Implementing State Machines
Step 5: List all the actions that happen in the system

1. Inserts quarters
2. Turns crank
3. Dispense
4. Ejects Quarters

Implementing State Machines
Step 6: Create a method that acts as the state machine

- Gumball machine class
- A method for each action
- We use conditionals to determine what behavior is appropriate in each state

```
public void insertQuarter ( ) {
    if (state == HAS_QUARTER) {
        System.out.println(“ Can’t insert another quarter”);
    } else if (state == SOLD_OUT) {
        System.out.println(
                    “ Can’t insert a quarter, the machine is sold out”);
    } else if (state == SOLD) {
        System.out.println (
                    “ Please wait we are getting you a gumball”);
    } else if (state == NO_QUARTER) {
        state = HAS_QUARTER;
        System.out.println(“You inserted a quarter”);
    }
}
```

```
public void ejectQuarter ( ) { … }
public void turnCrank ( ) { … }
public void dispense ( ) { … }
public void somthingElse ( ) { … }
```

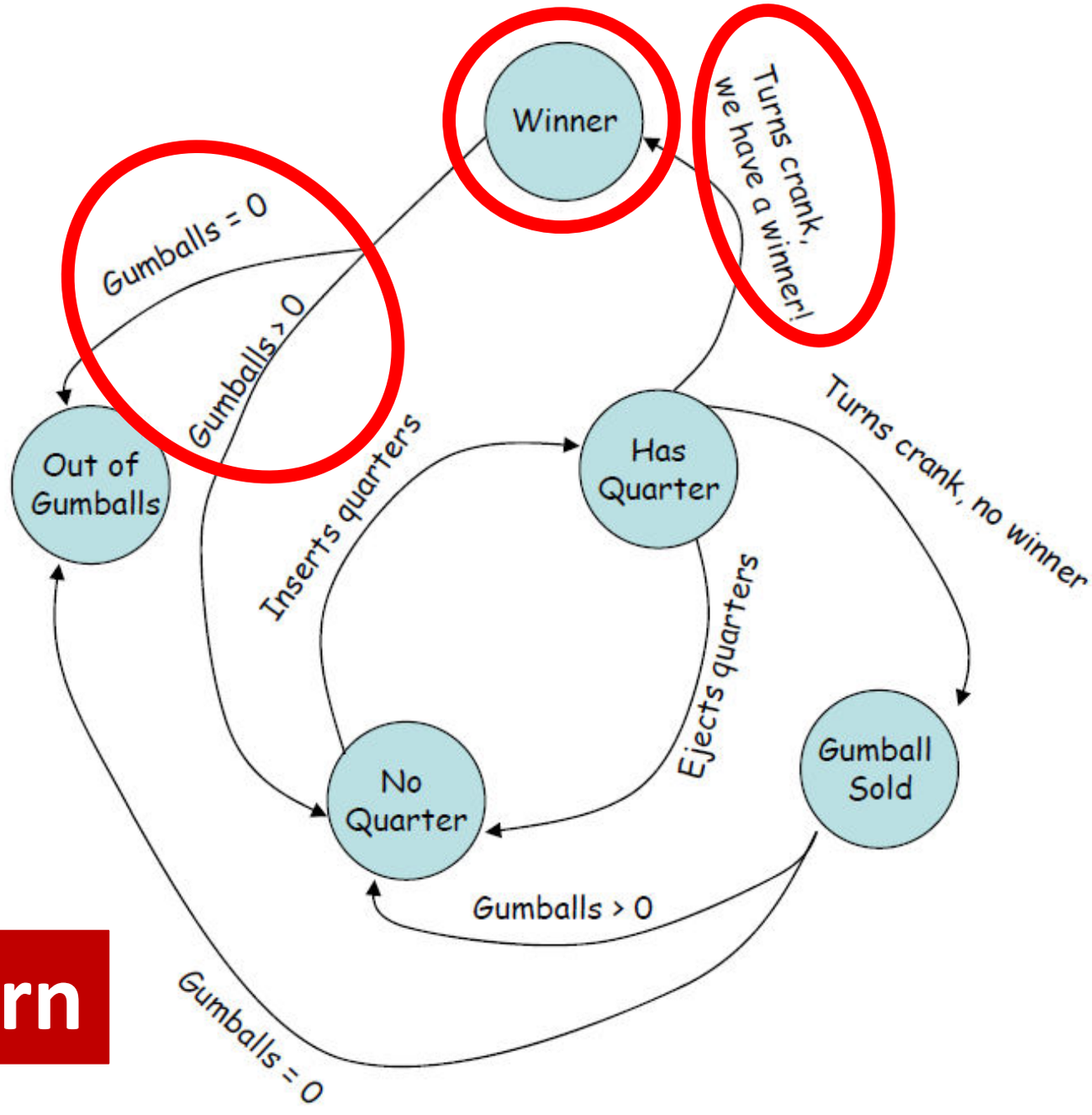
Everything seems satisfactory until we need some changes

New Requirements

- Turn gumball buying into a game!
- 10% of the time when the crank is turned, the customer gets two gumballs instead on one!

New State Diagram

- add a new state
- add three new methods
- add a new condition in each of the existing methods that correspond to action.



Solution: State Pattern

Software Design and Architecture

Topic#077

END!

Software Design and Architecture

Topic#078
State Design Pattern

© Virtual University of Pakistan

The State Pattern

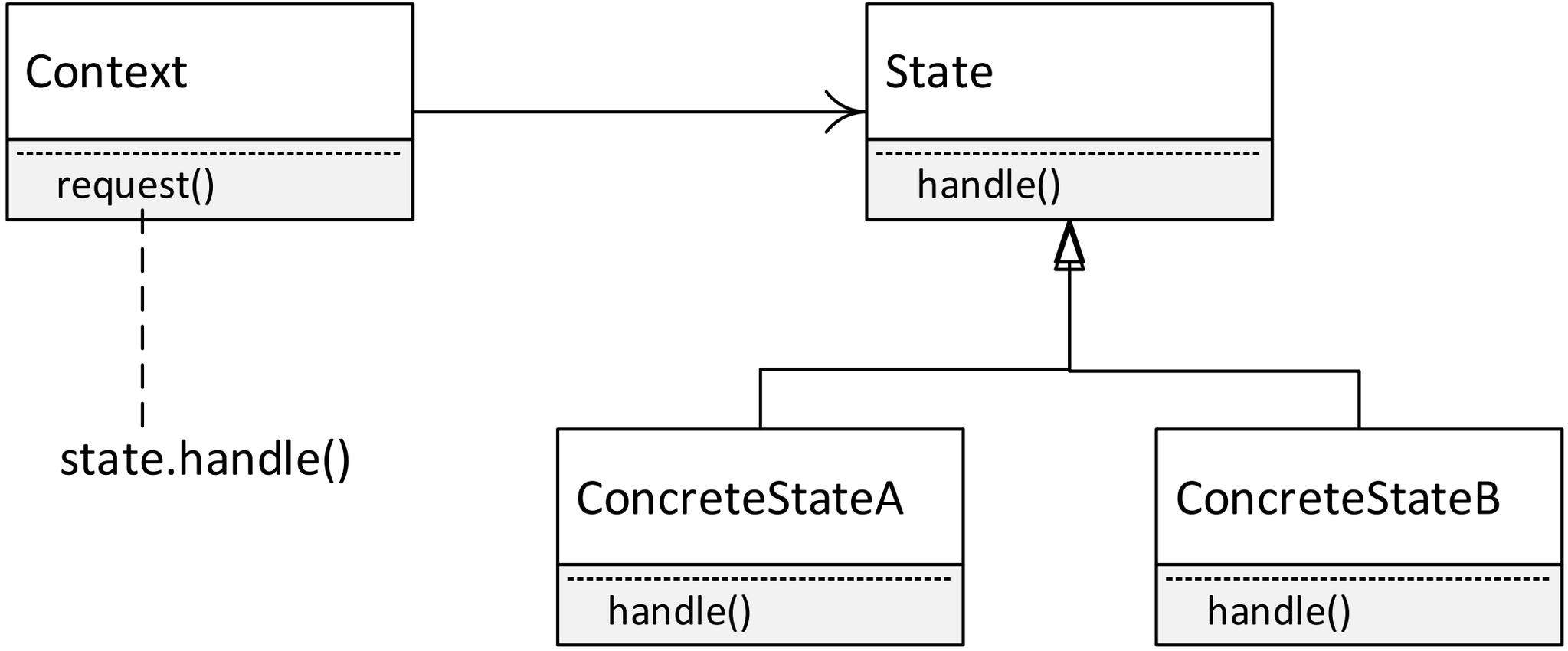
The **State Pattern** allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

State Design Pattern: Motivation

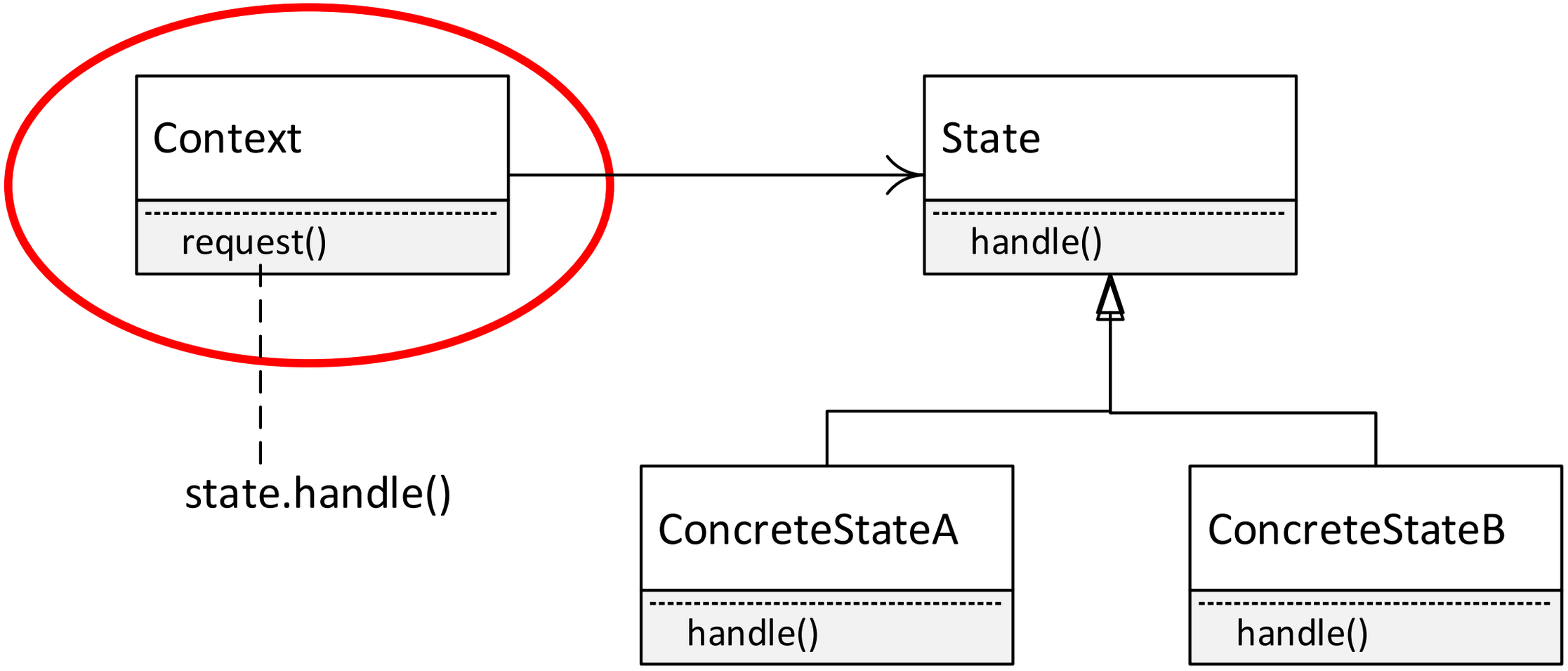
- The State pattern is useful when you want to have an object represent the state of an application, and you want to change the state by changing that object.
- The State pattern is intended to provide a mechanism to allow an object to alter its behavior in response to internal state changes.

State Design Pattern: Motivation

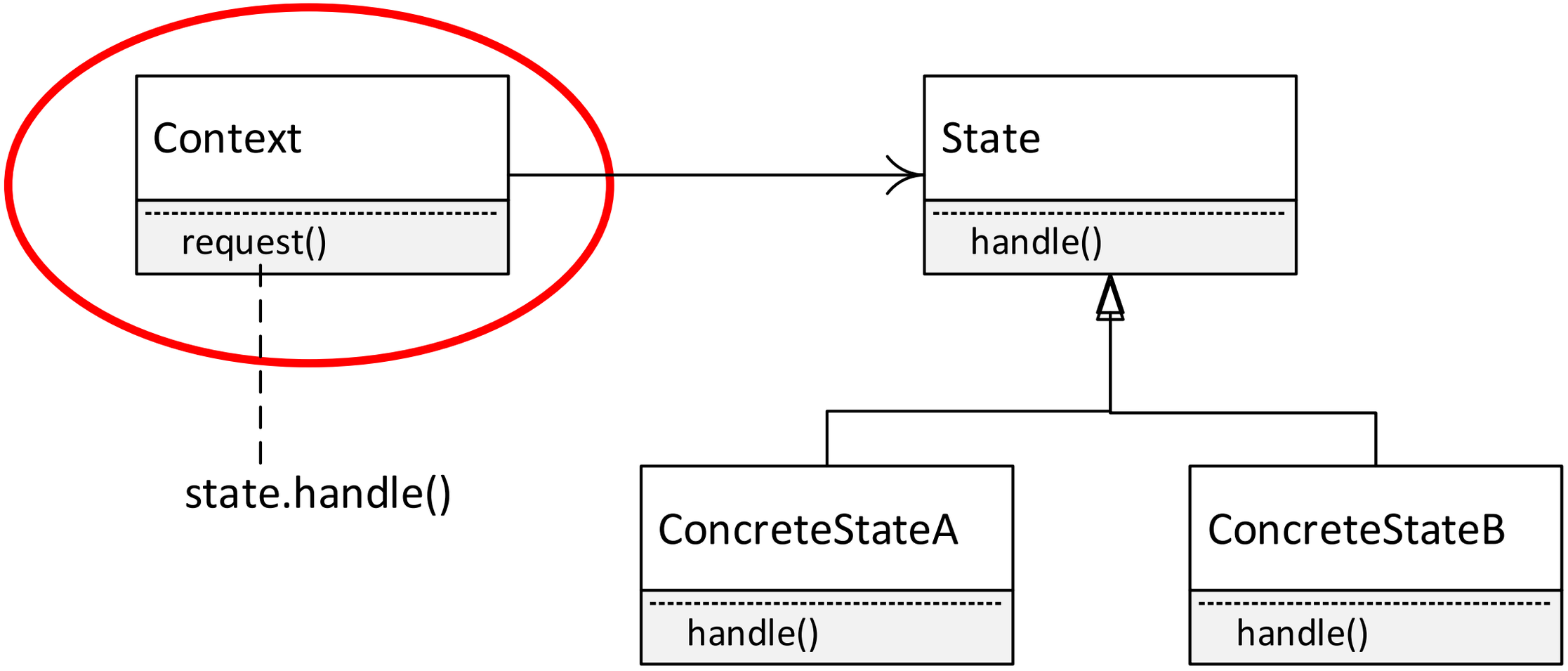
- To the client, it appears as though the object has changed its class.
- The benefit of the State pattern is that state-specific logic is localized in classes that represent that state.



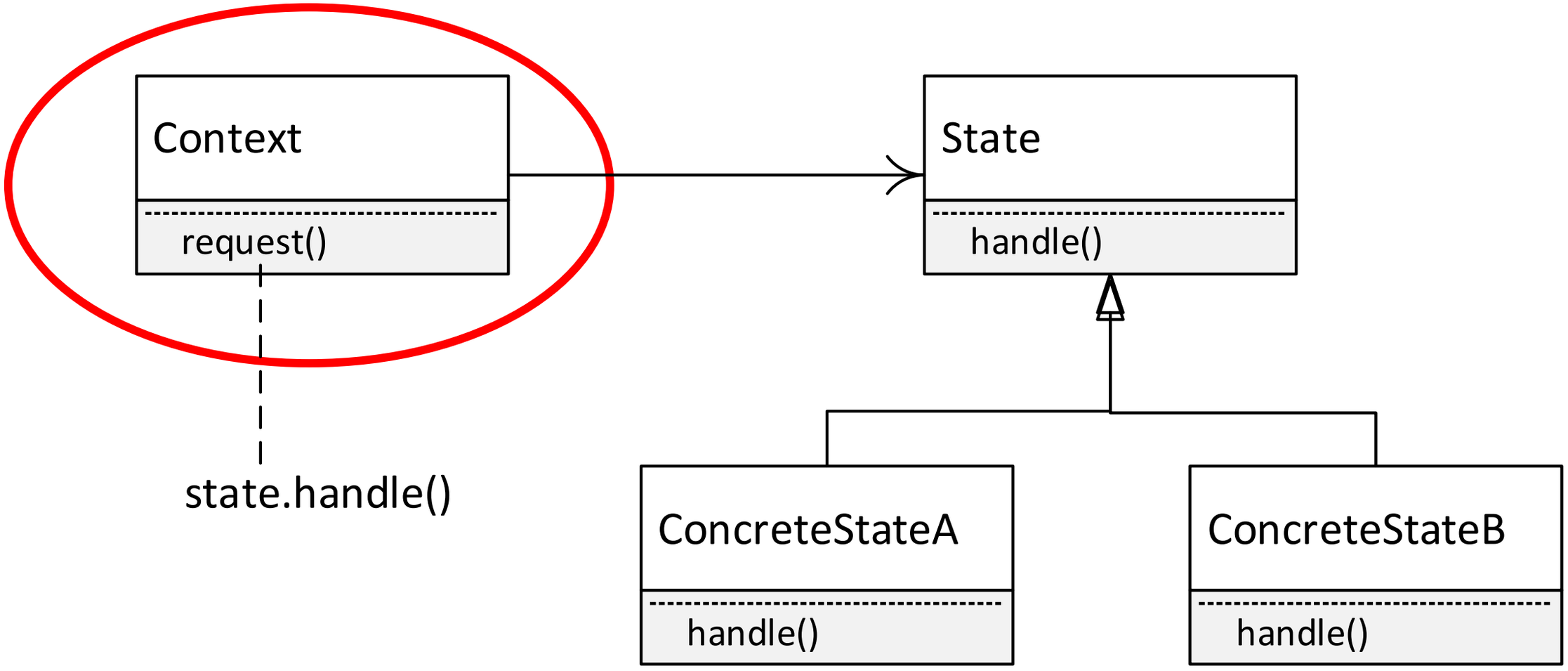
State Design Pattern Structure



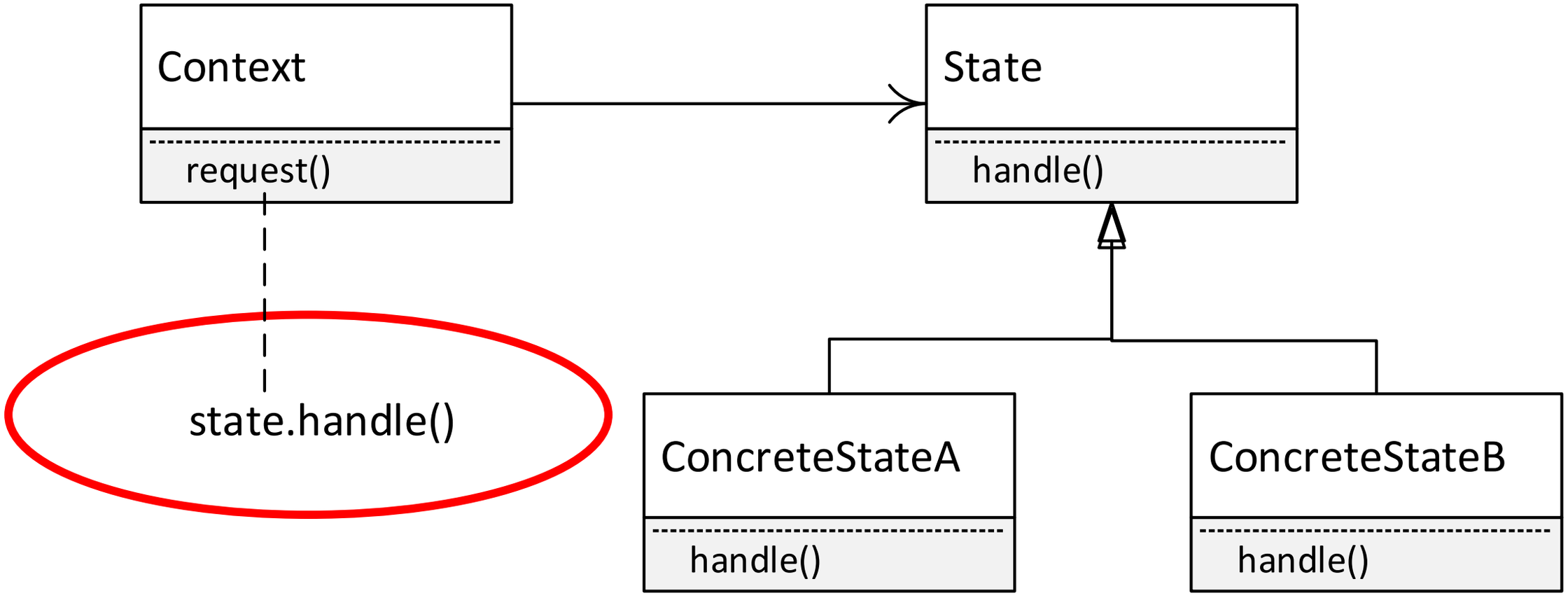
The context is the class that can have a number of internal states – for example Gumball Machine



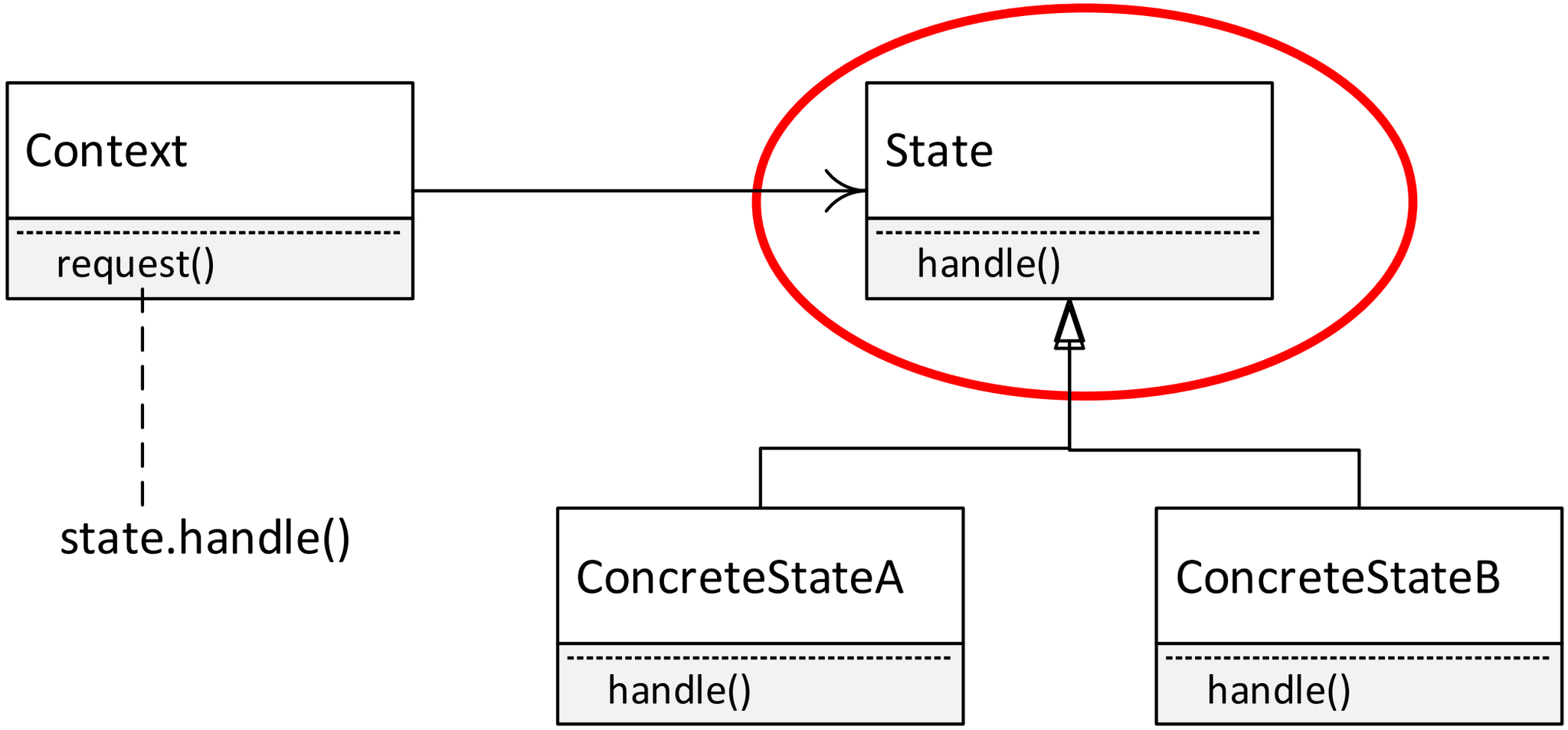
defines the interface of interest to clients



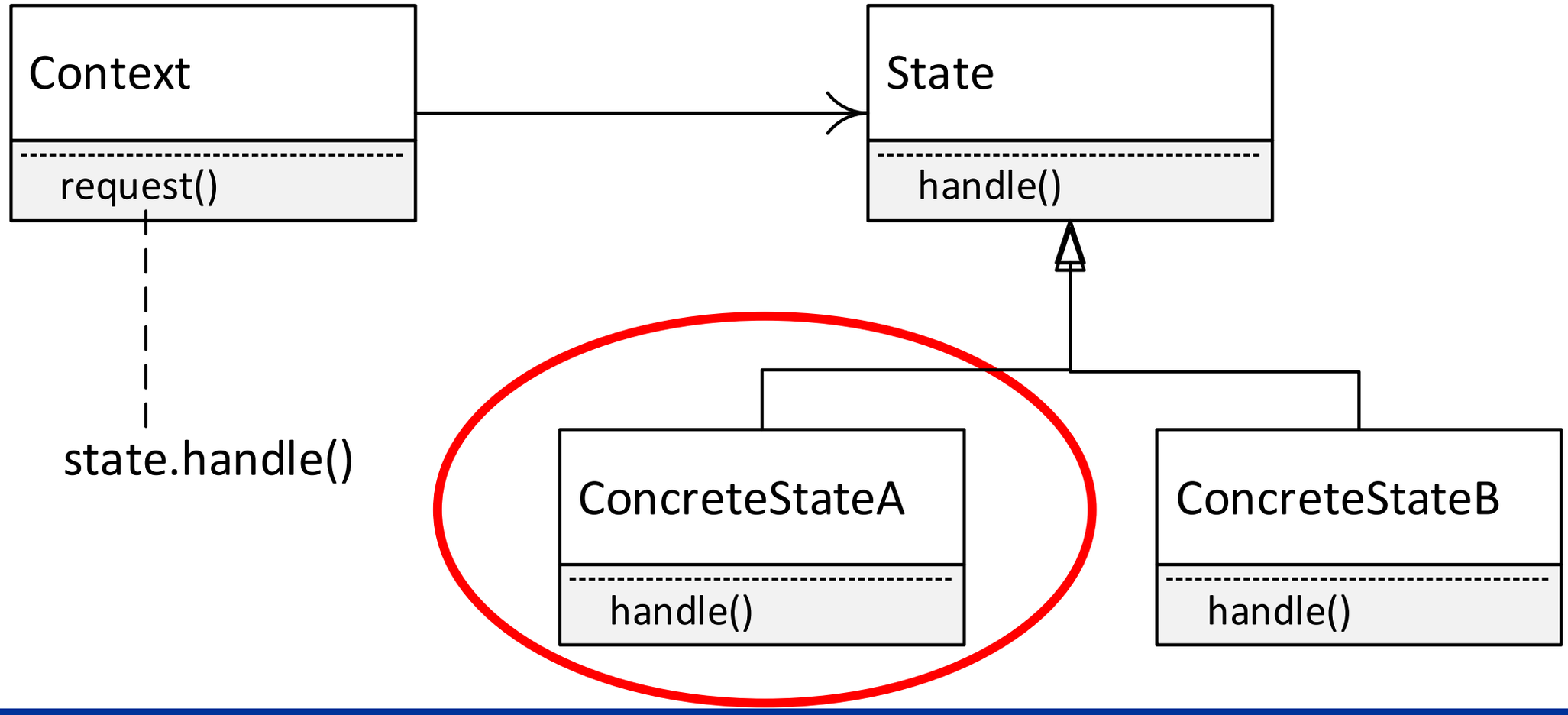
maintains an instance of a ConcreteState subclass that defines the current state



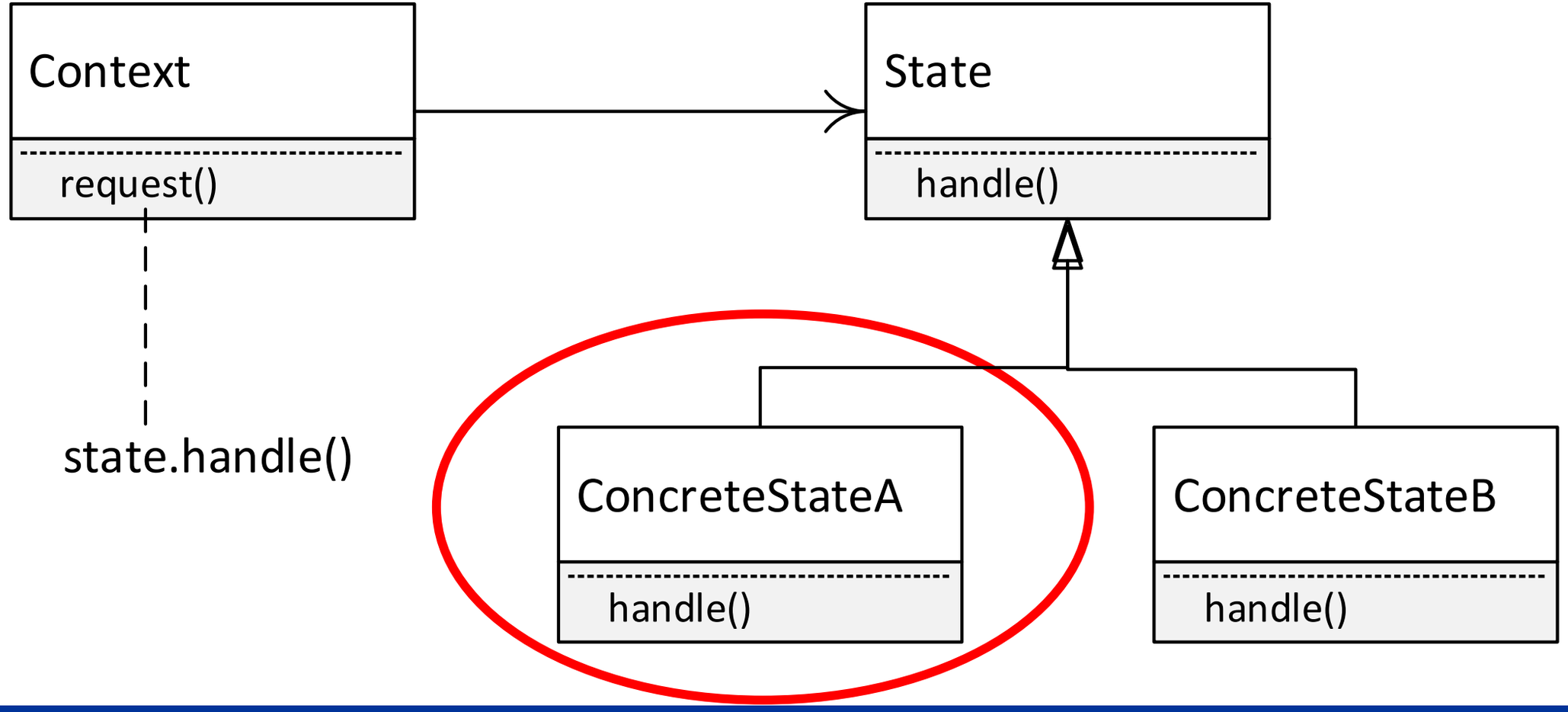
Whenever the request() is made on the Context, it is delegated to the State handle()



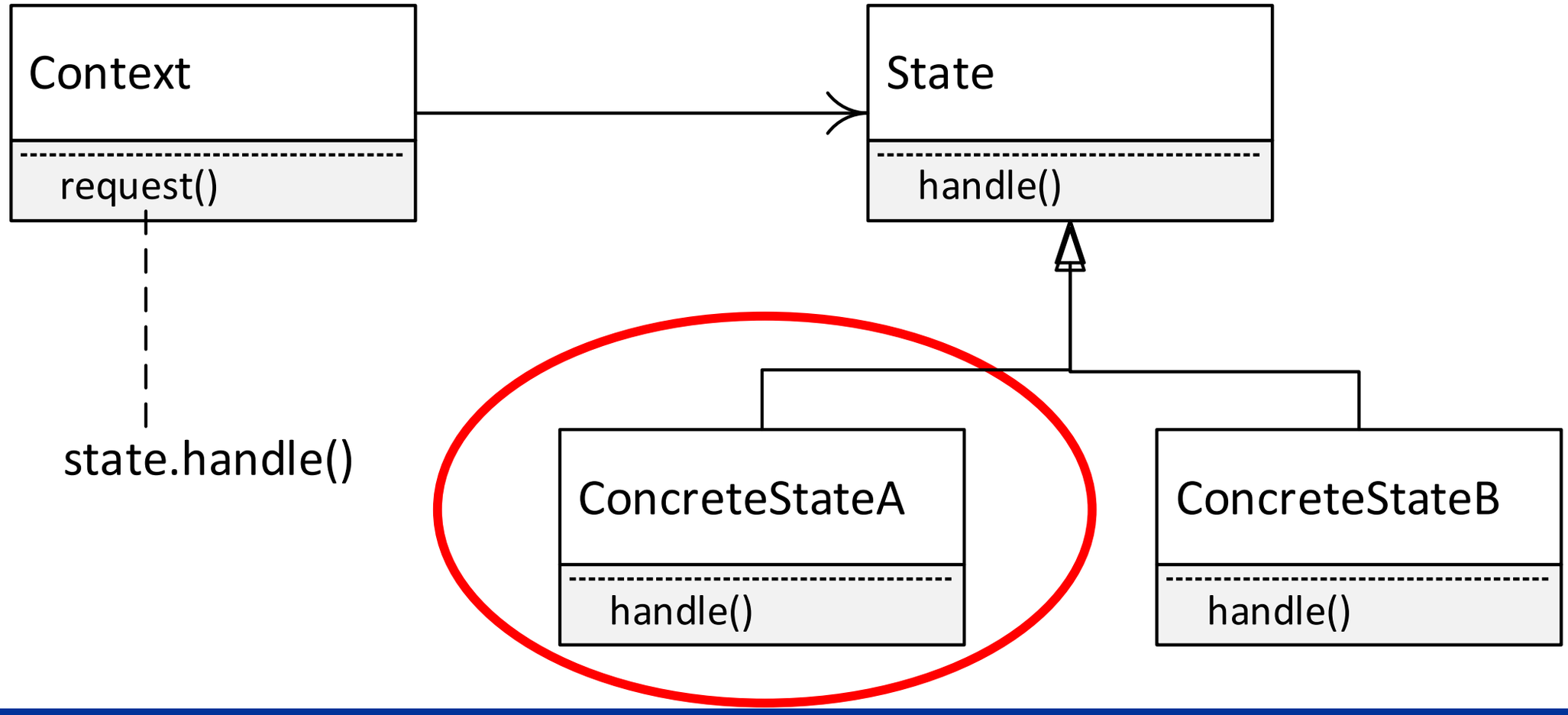
defines an interface for encapsulating the behavior associated with a particular state of the Context



Each state is represented by a concrete state class – There are many concrete state classes



Concrete states handle request from the Context



Each Concrete state provides its own implementation for a request – in this way the context changes state

Software Design and Architecture

Topic#078

END!

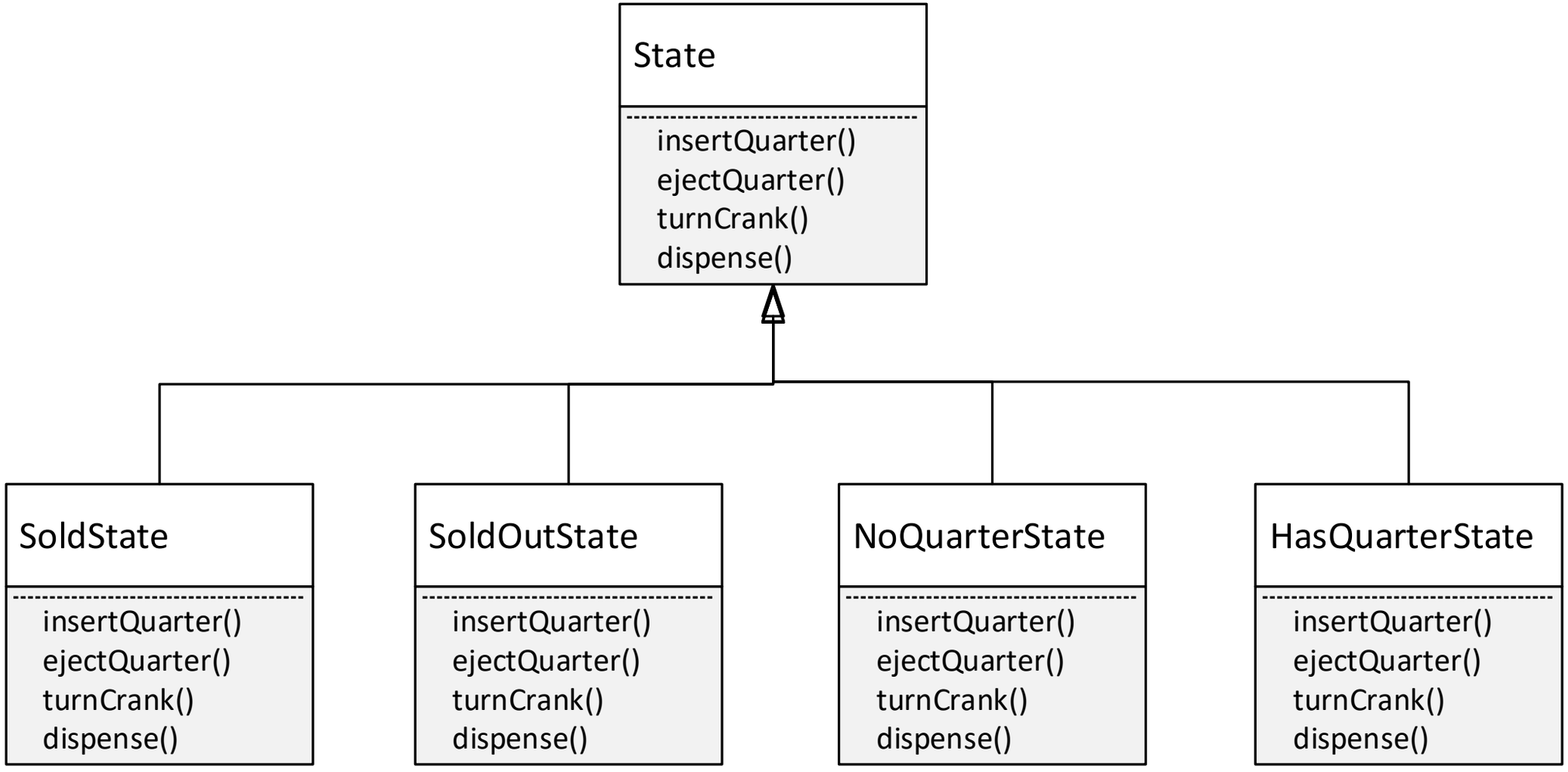
Software Design and Architecture

Topic#079
State Design Pattern - Example

Gumball Machine - The new design

1. Define a State interface that contains a method for every action in the Gumball Machine
2. Implement a State class for every state of the machine. These classes will be responsible for the behavior of the machine when it is in the corresponding state
   - We are going to get rid of all the conditional code and instead delegate to the state class to do all the work

© Virtual University of Pakistan



© Virtual University of Pakistan

```
public class NoQuarterState extends State {
        GumballMachine gbMachine;

        public NoQuarterState (GumballMachine gbm) {
                this.gbMachine = gbm;
        }

        public void insertQuarter() {
                system.out.println("You have inserted a quarter");
                gbMachine.setState(gbMachine.hasQuarterState.getState();
        }

        public void ejectQuarter() {
                system.out.println("You have not inserted a quarter");
        }
        …
}
```

constructor

Do the needful and then change state to next state

```
public class GumballMachine {
        State soldState;
        State soldOutState;
        State noQuarterState;
        State hasQuarterState;

        State state = soldOutState;
        int     count = 0;
        public void insertQuarter();
        …
}
```

All state objects are created and assigned in constructor

object representing the current state initialized to soldOutState

```
public class GumballMachine {
        ...
        public void insertQuarter() {
            state.insertQuarter();
        }
        ...
}
```

Just delegate.

That's it!!

- Localized the behavior of each state into its own class
- Removed all if statements

```
public class GumballMachine {
        ...
        public void insertQuarter() {
                state.insertQuarter();
        }
        ...
}
```

Just delegate.

That's it!!

- When an action is called, it is delegated to the current state
- It does the needful and sets the state to the next state

State Pattern - Summary

- Allows a context to change its behavior as the state of the context changes
- When an action is called, it is delegated to the current state
- It does the needful and sets the state to the next state
- By encapsulating each state in a class, we localize any changes that need to be made.

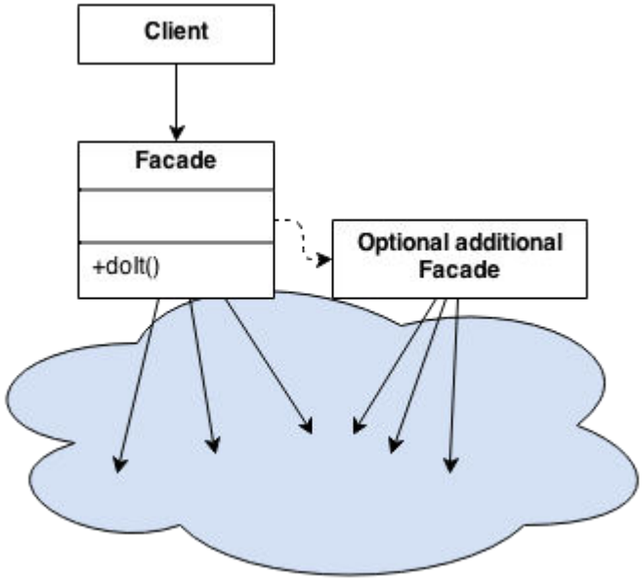
Software Design and Architecture

Topic#079

END!

Software Design and Architecture

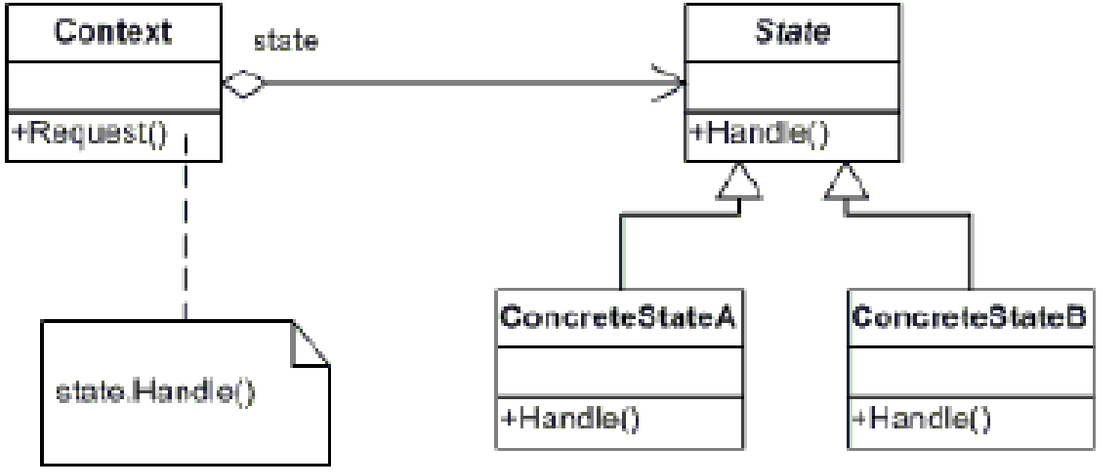
Topic#080
Comparison of Strategy and State Design Patterns

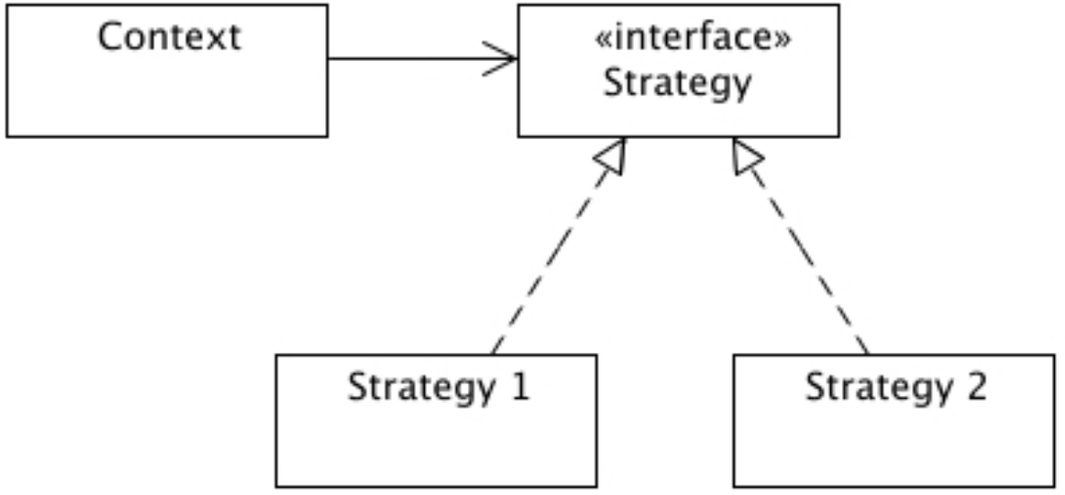


Everything seems satisfactory until we think about the life of the traineeEmployee object

Object Morphing

Strategy vs State





State vs Strategy

- They are both examples of association with delegation
- The difference between the State and Strategy patterns is one of intent

State	Strategy
• Set of behaviors encapsulated in state objects; • At any time the context is delegating to one of those states. • Over time, the current state changes across the set of state objects to reflect the internal state of the context, so the context's behavior changes over time. Client knows very little, if anything, about the state objects • Alternative to putting a lots of conditional in the context	• Client usually specifies the strategy object that the context is associated with. • While the pattern provides the flexibility to change the strategy object at run time, there is one strategy object that is most appropriate for a context object. • Flexible alternative to subclassing: if you use inheritance to define the behavior of a class, you are stuck with it even if you need to change it. • With strategy you can change the behavior by associating it with different objects

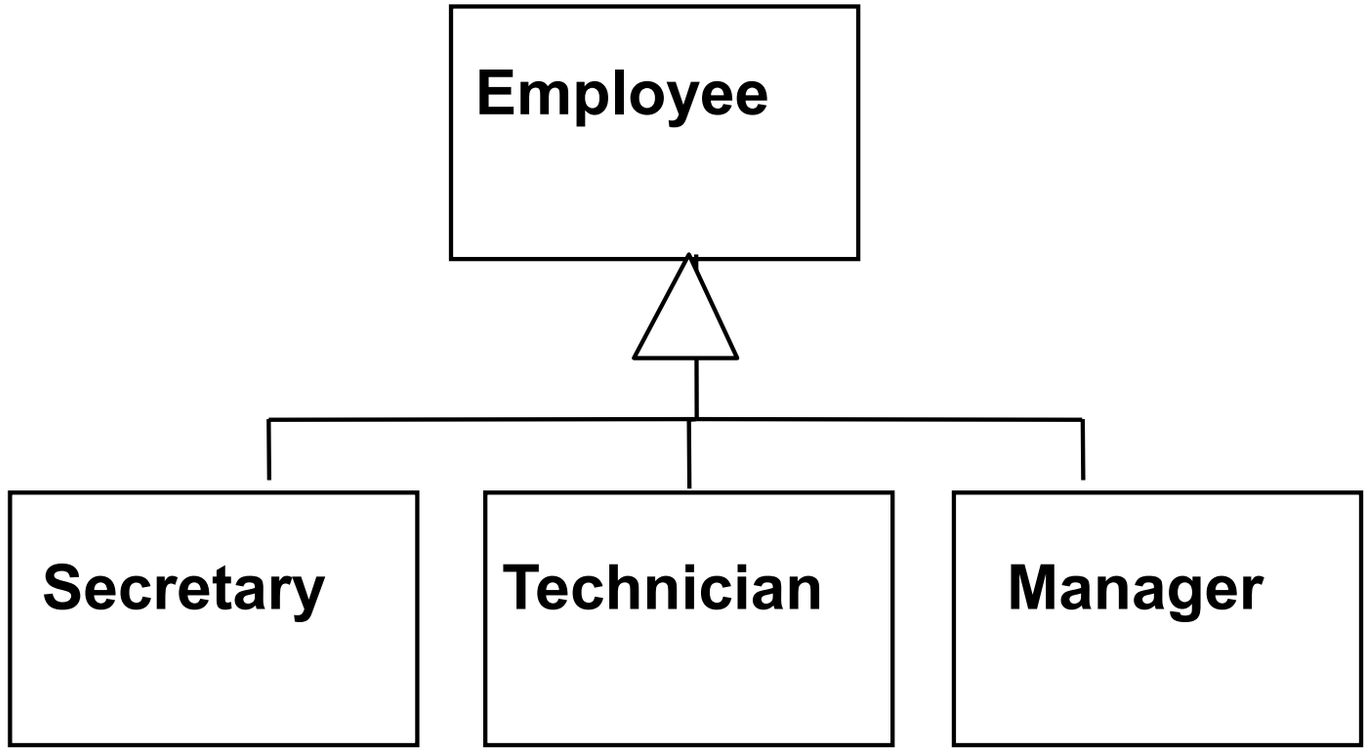
Software Design and Architecture

Topic#080

END!

Software Design and Architecture

Topic#081
Composite Design Pattern



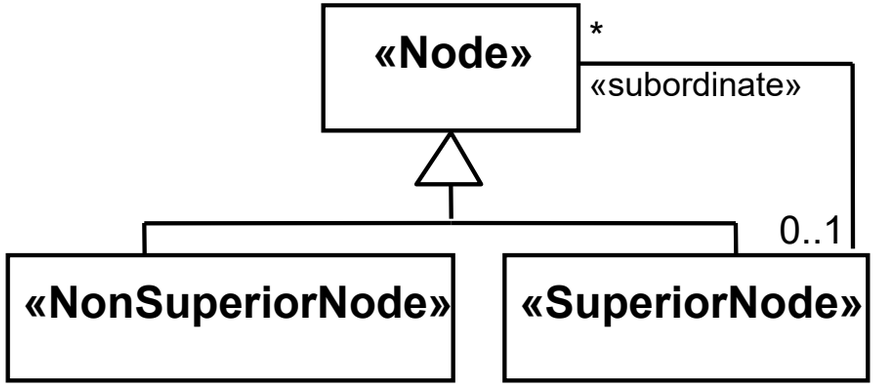
A company has different types of employees

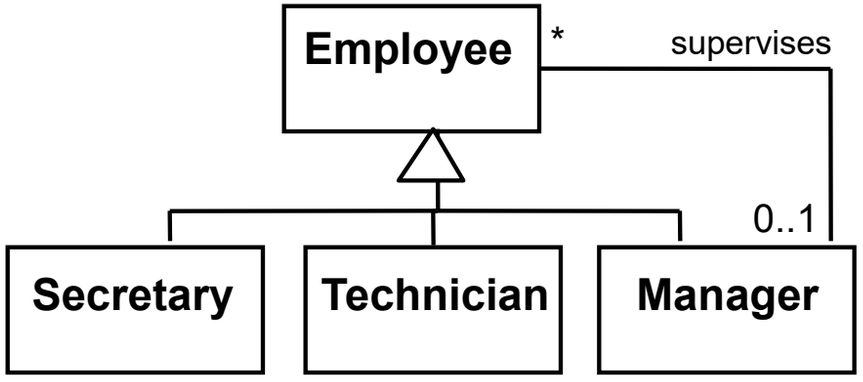
Problem: a manager manages other employees
How to model?

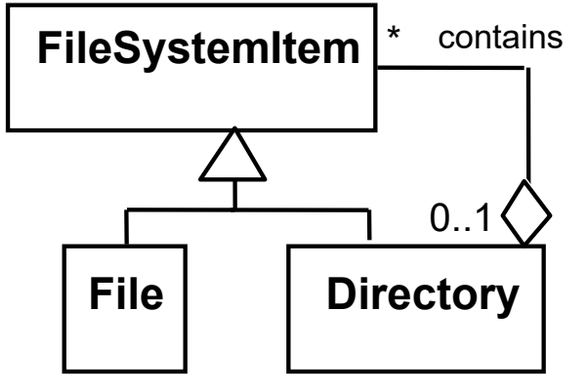
The General Hierarchy Pattern

- ***Context***:
  - Objects in a hierarchy can have one or more objects above them (superiors)
    - and one or more objects below them (subordinates).
  - Some objects cannot have any subordinates
- ***Problem***:
  - How can we represent a hierarchy of objects, in which some objects cannot have subordinates?
- ***Forces***:
  - You want a flexible way of representing the hierarchy
    - that prevents certain objects from having subordinates
  - All the objects have many common properties and operations

The General Hierarchy Pattern aka Composite Pattern

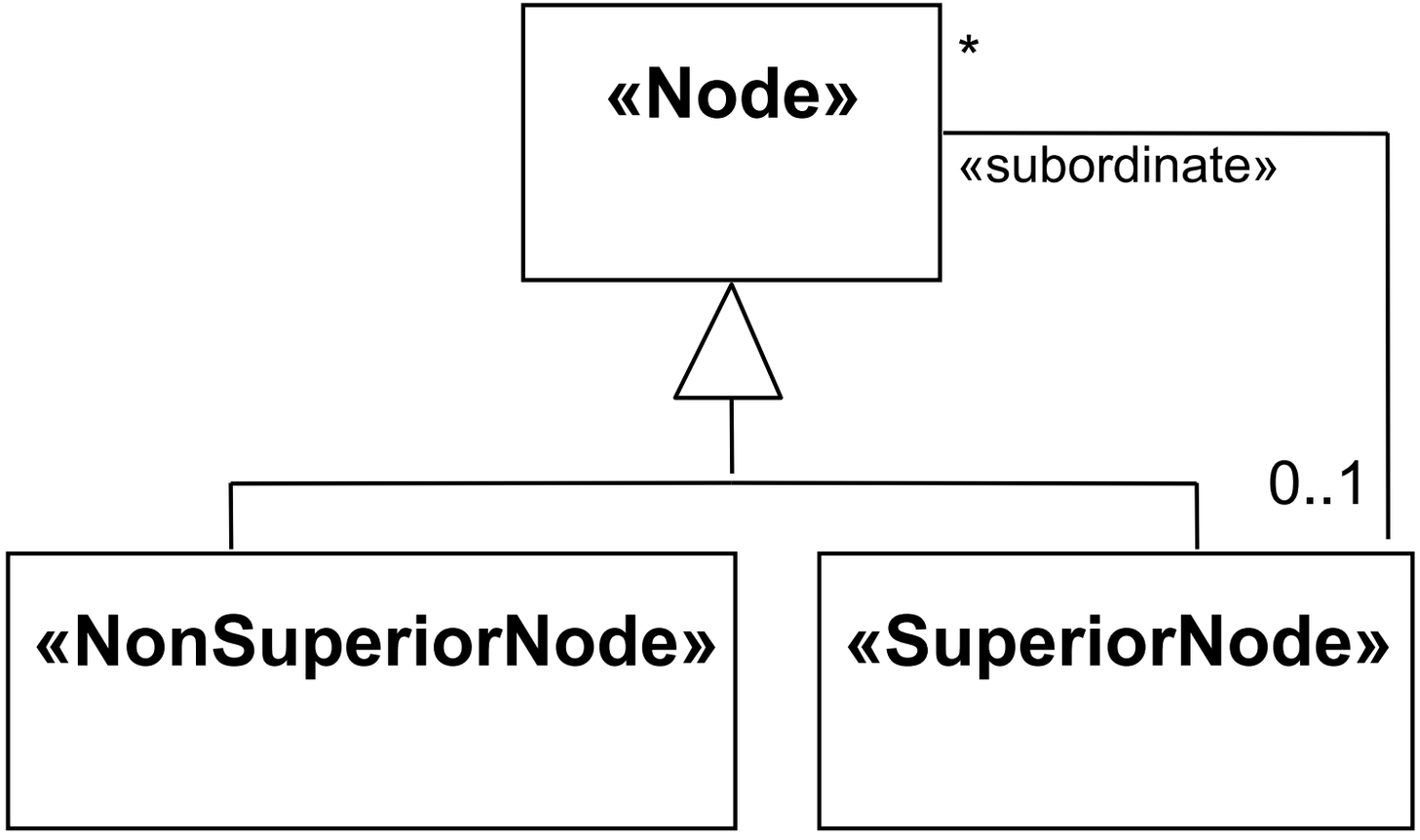






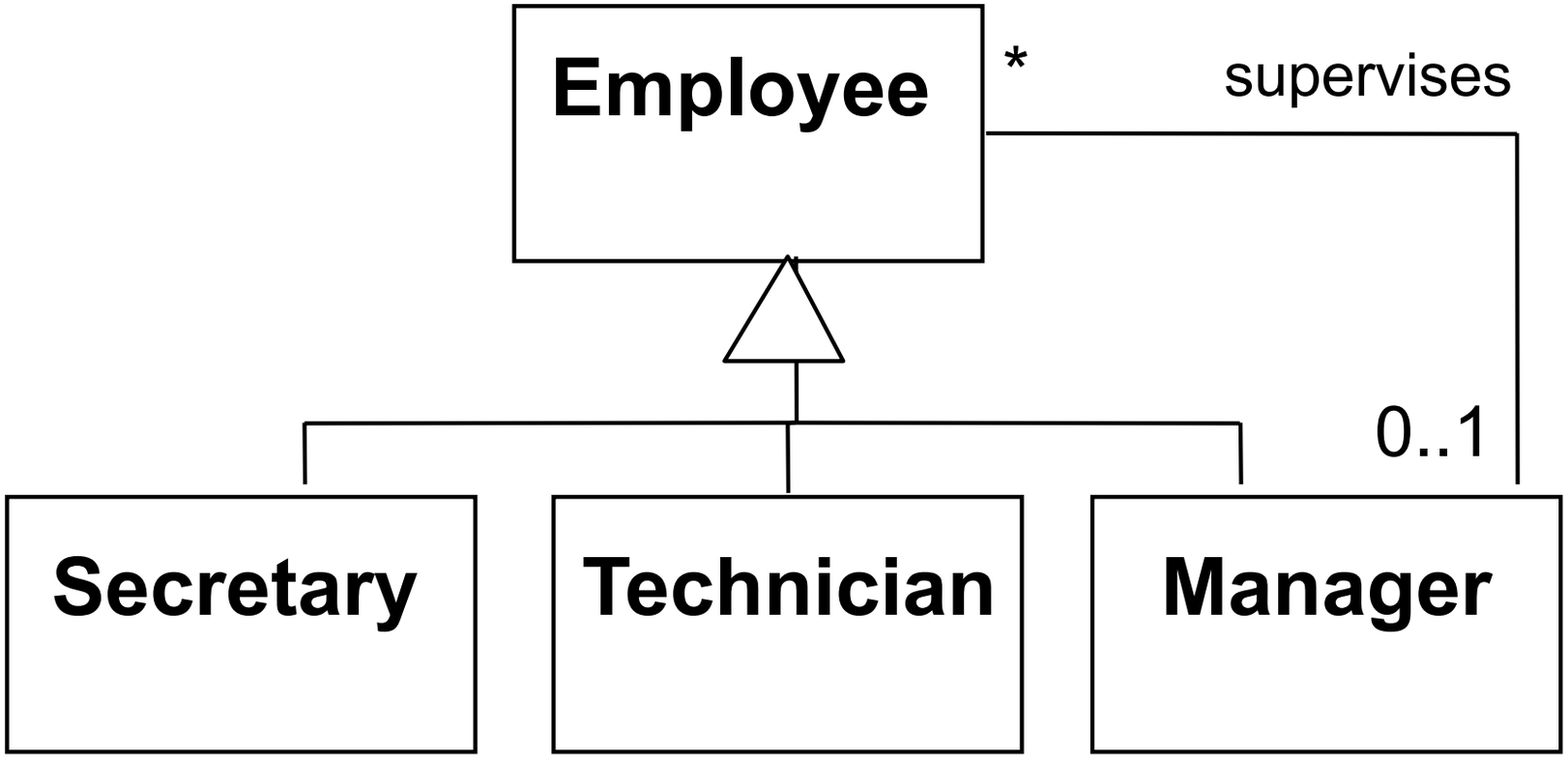
© Virtual University of Pakistan

The General Hierarchy Pattern aka Composite Pattern



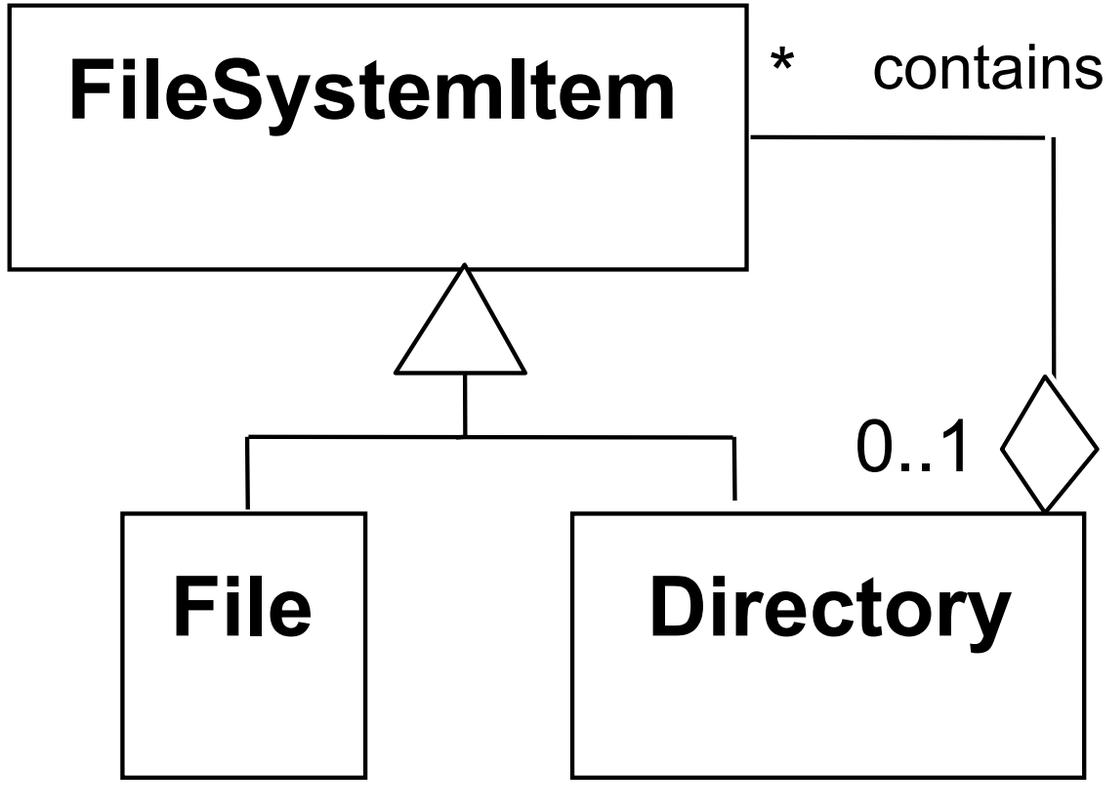
© Virtual University of Pakistan

The General Hierarchy Pattern



© Virtual University of Pakistan

The Composite Pattern



- The difference between a tree data structure and the composite pattern

- In the case of a tree data structure, all nodes have the same structure and behavior

Software Design and Architecture

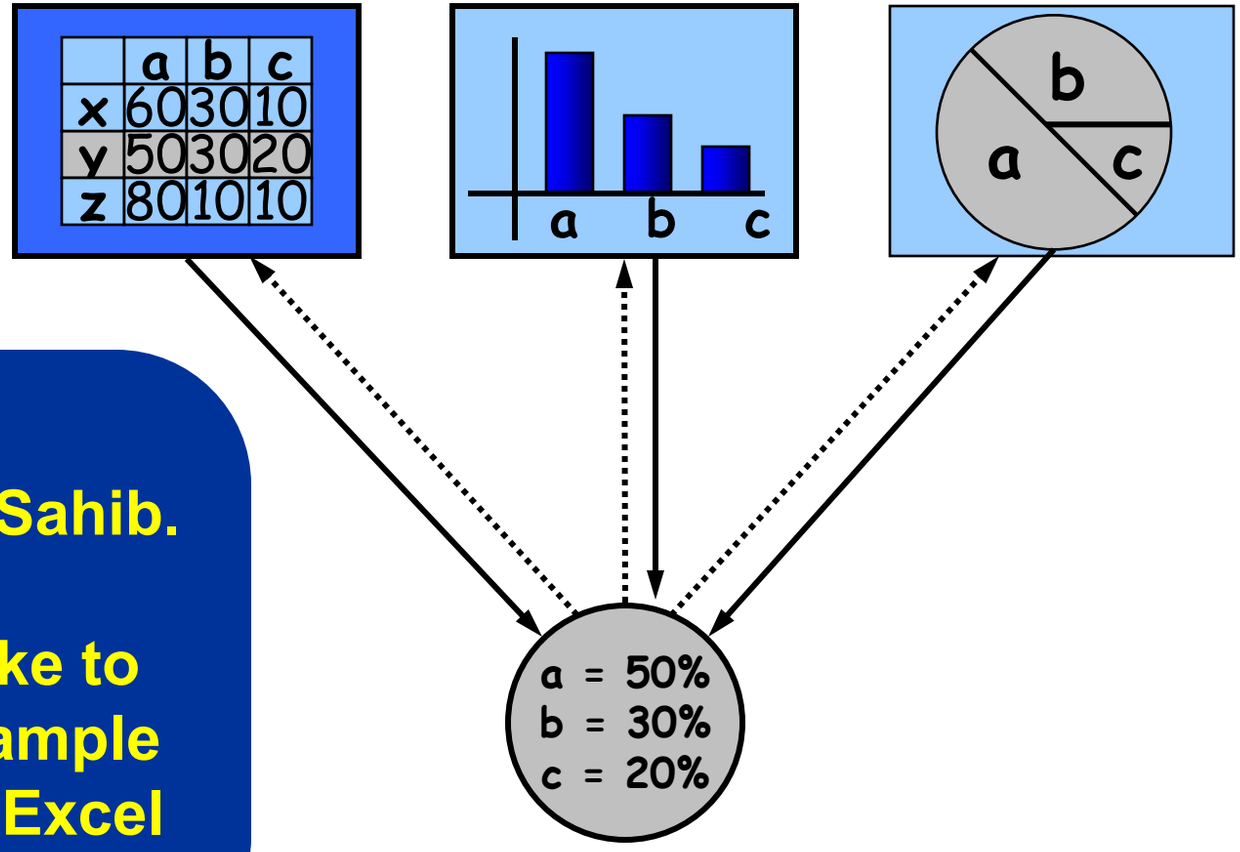
Topic#081

END!

Software Design and Architecture

Topic#082
Observer Design Pattern

Excel - Example



Note for Nasim Sahib.

Here I would like to insert a live example from Microsoft Excel

Observer Pattern

- Defines a “one-to-many” dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

- a.k.a Dependence mechanism / publish-subscribe / broadcast / change-update

© Virtual University of Pakistan

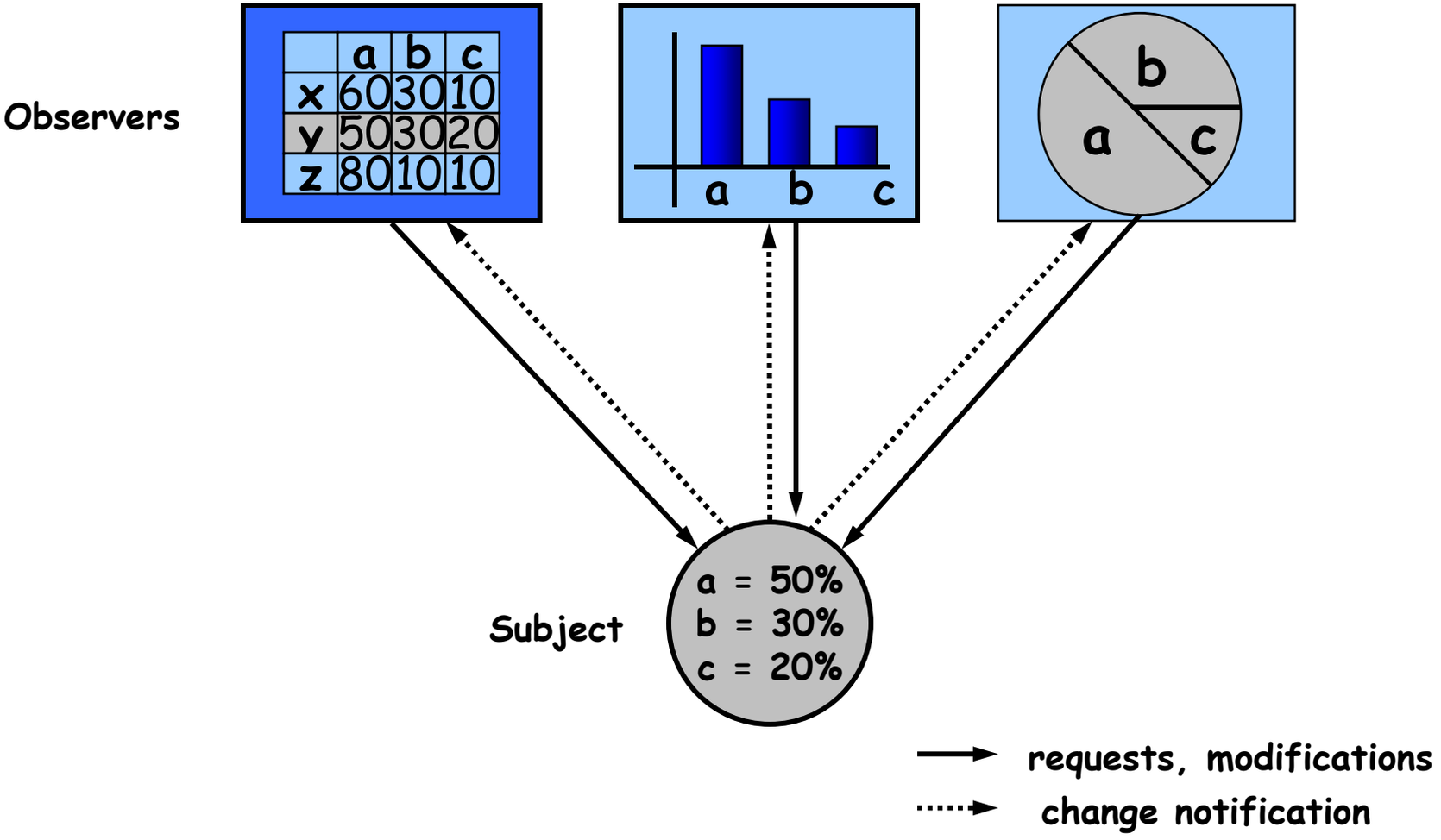
Subject & Observer

- Subject
  - the object which will frequently change its state and upon which other objects depend

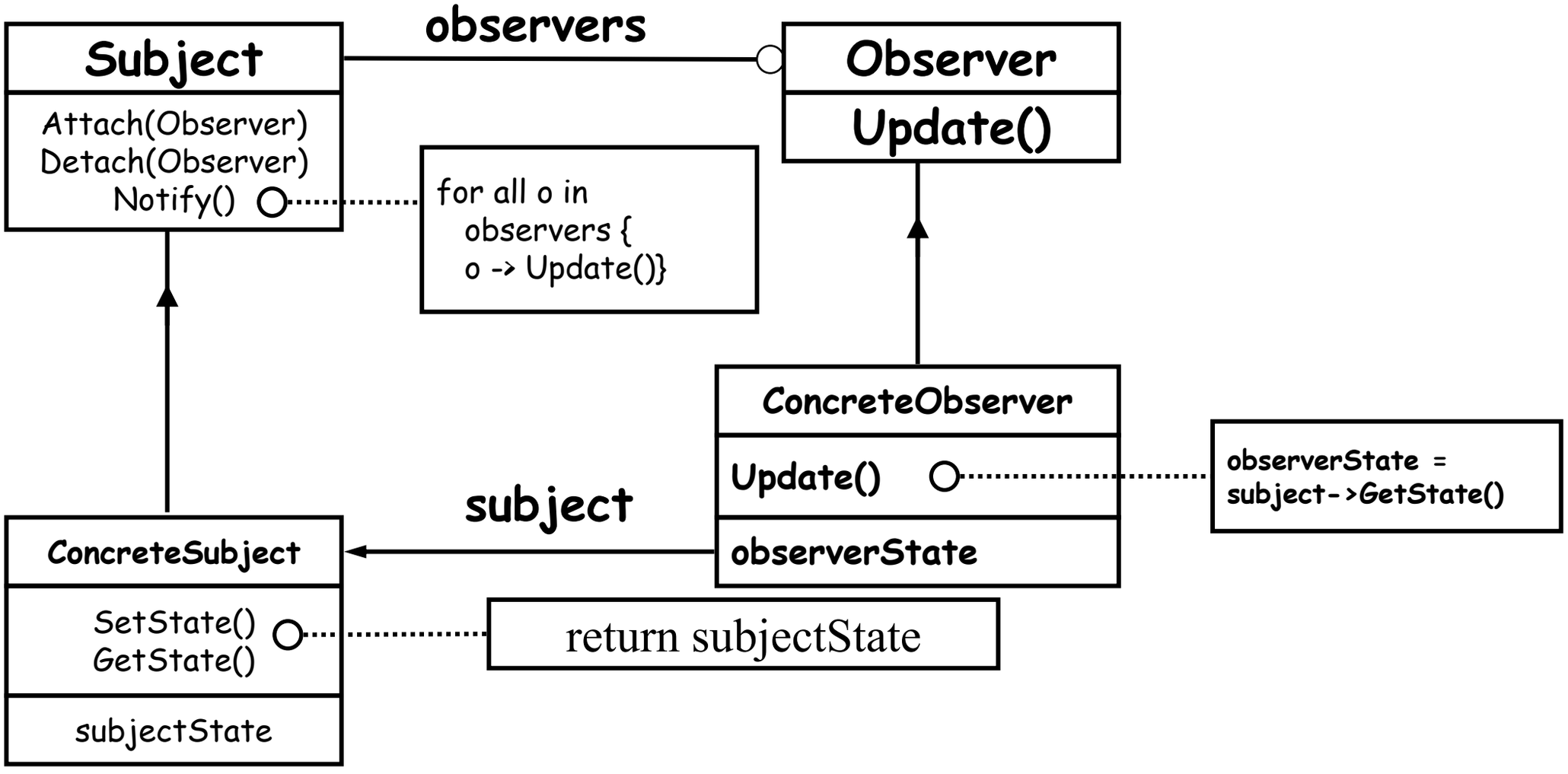
- Observer
  - the object which depends on a subject and updates according to its subject's state.

© Virtual University of Pakistan

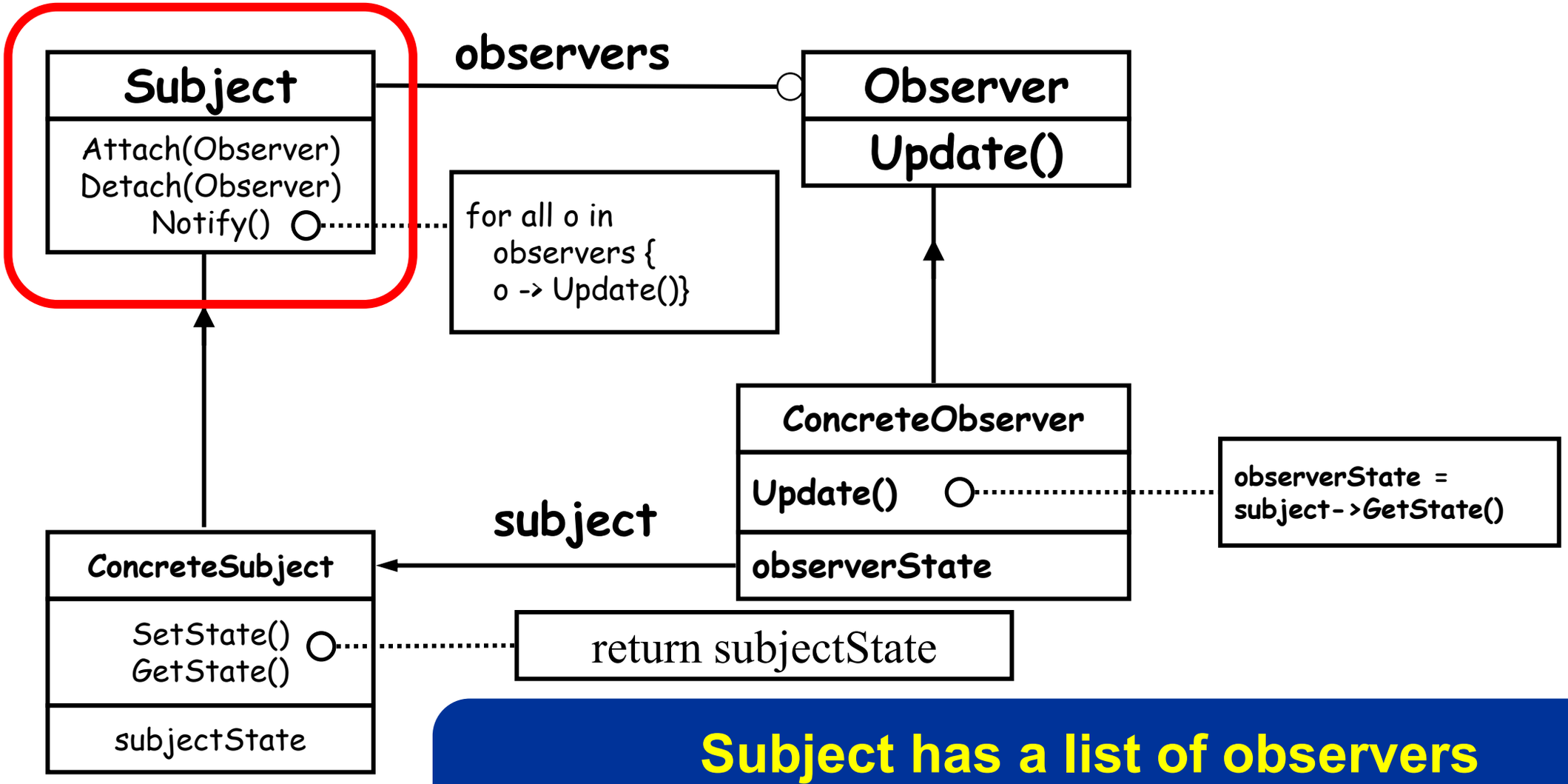
Observer Pattern - Example



Observer Pattern - UML

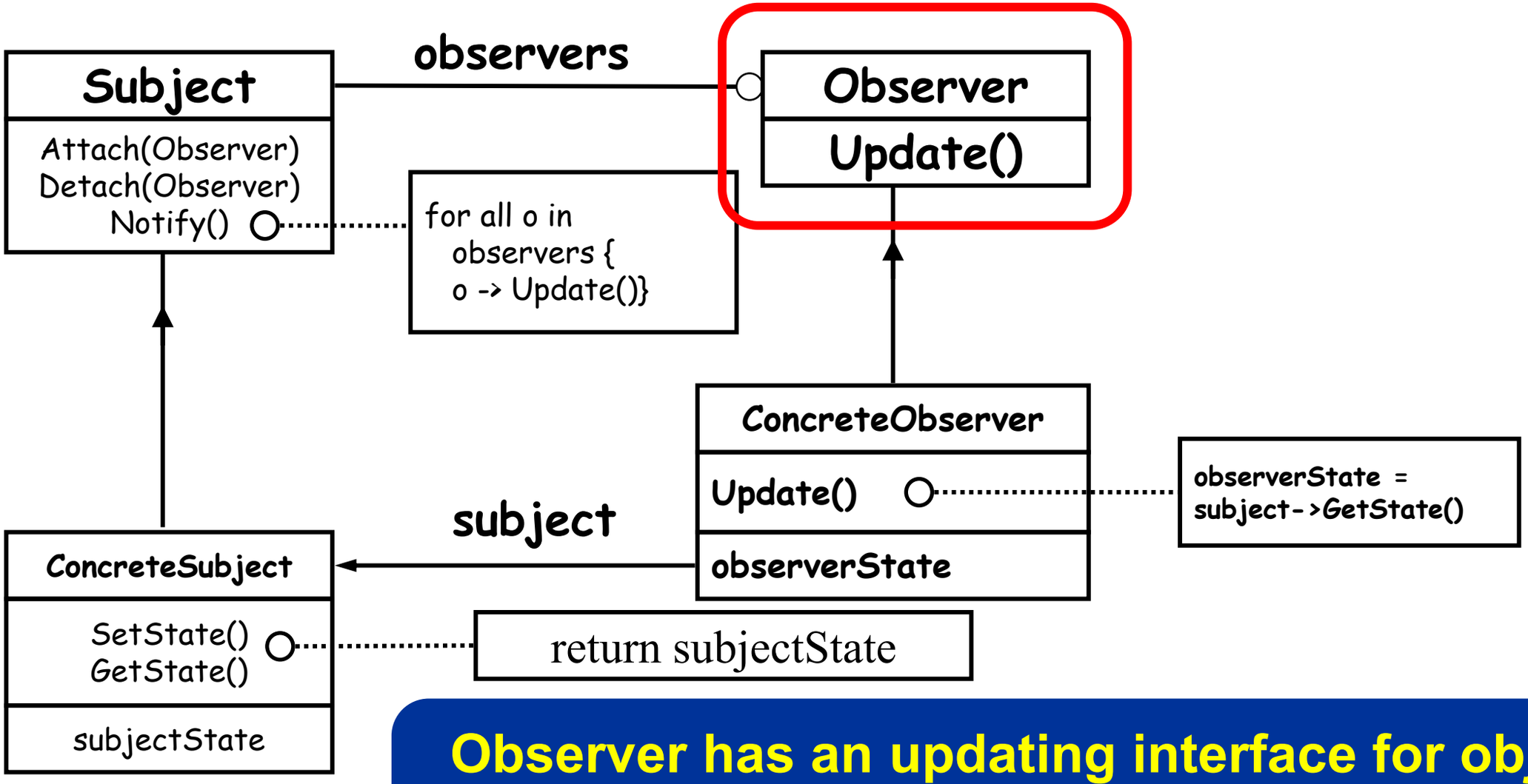


Observer Pattern - UML



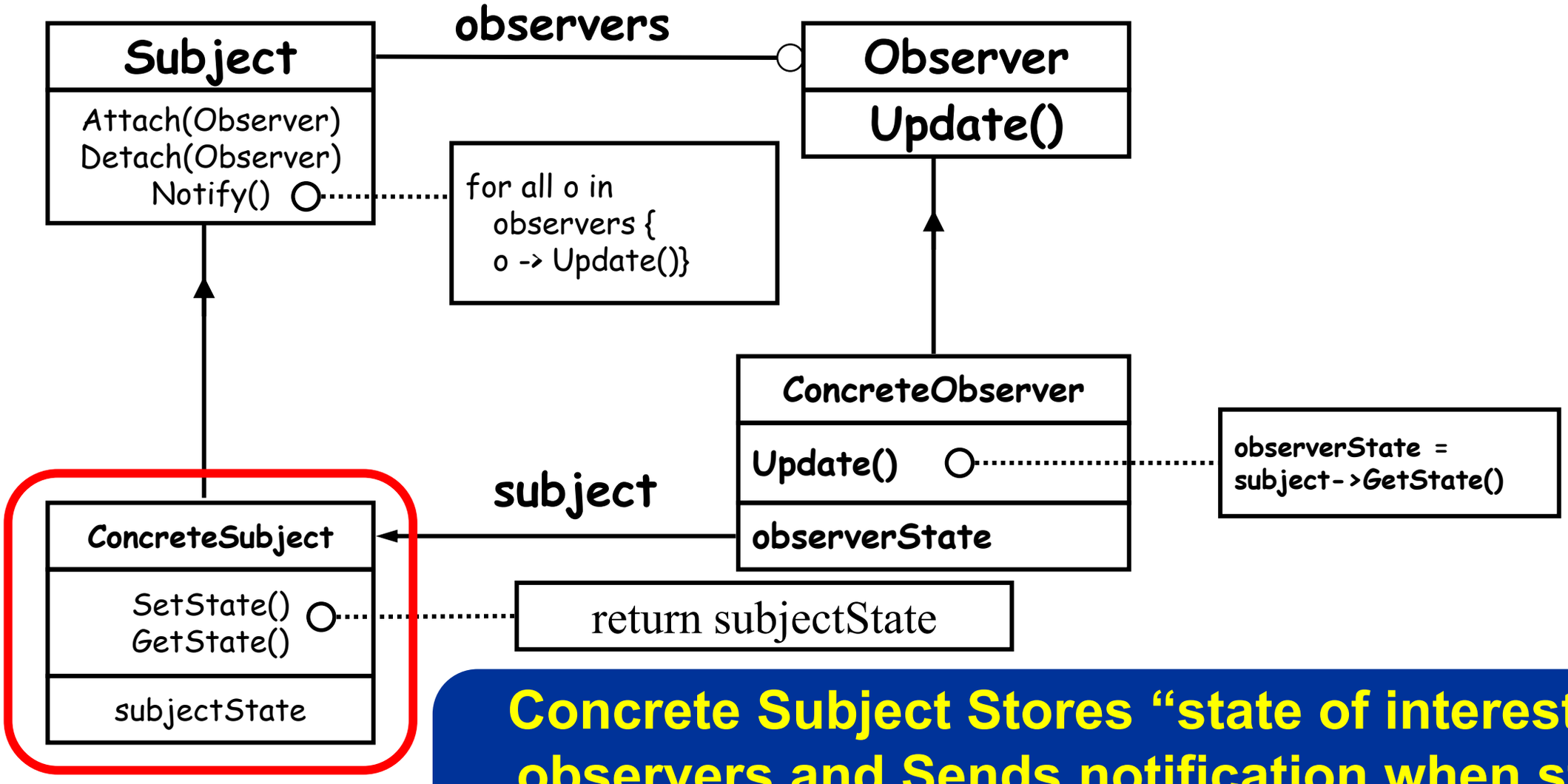
Subject has a list of observers
Interfaces for attaching/detaching an observer

Observer Pattern - UML



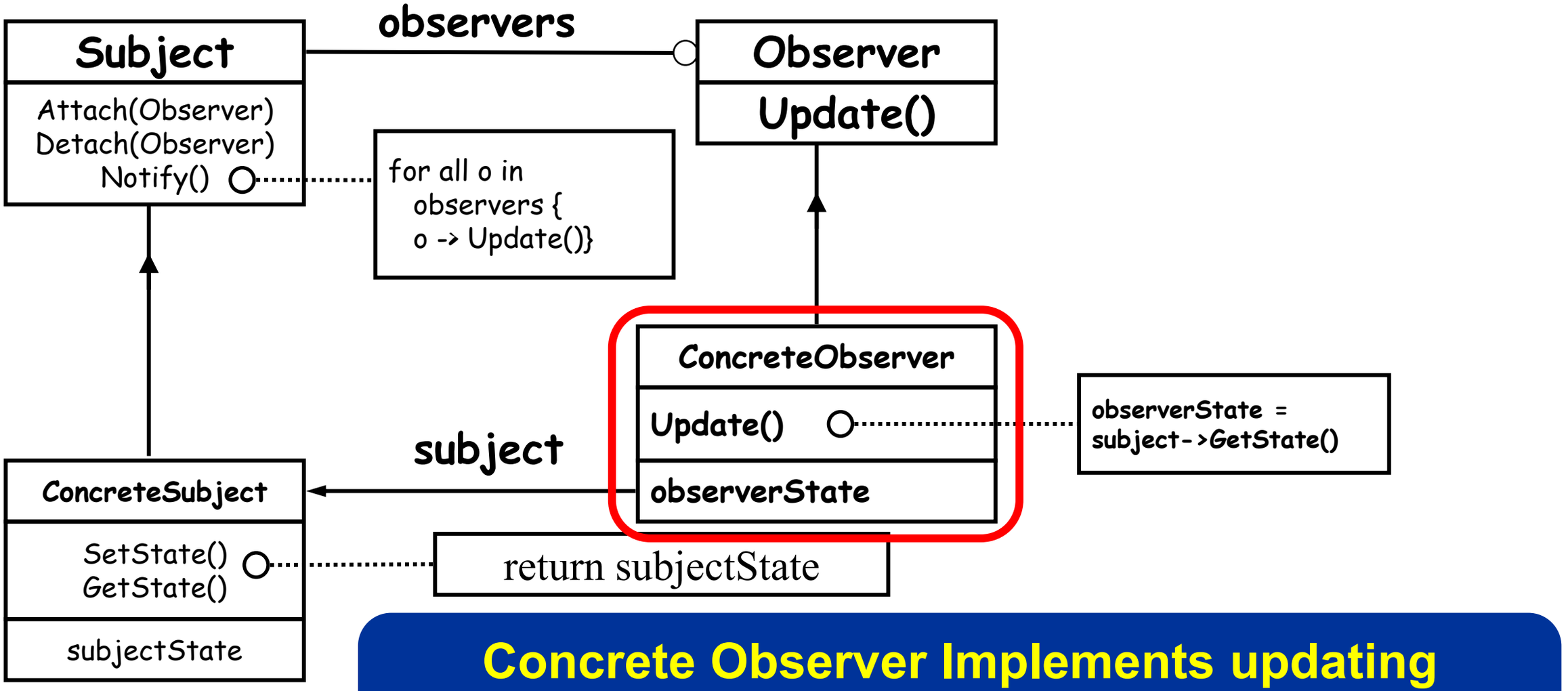
Observer has an updating interface for objects that gets notified of changes in a subject

Observer Pattern - UML



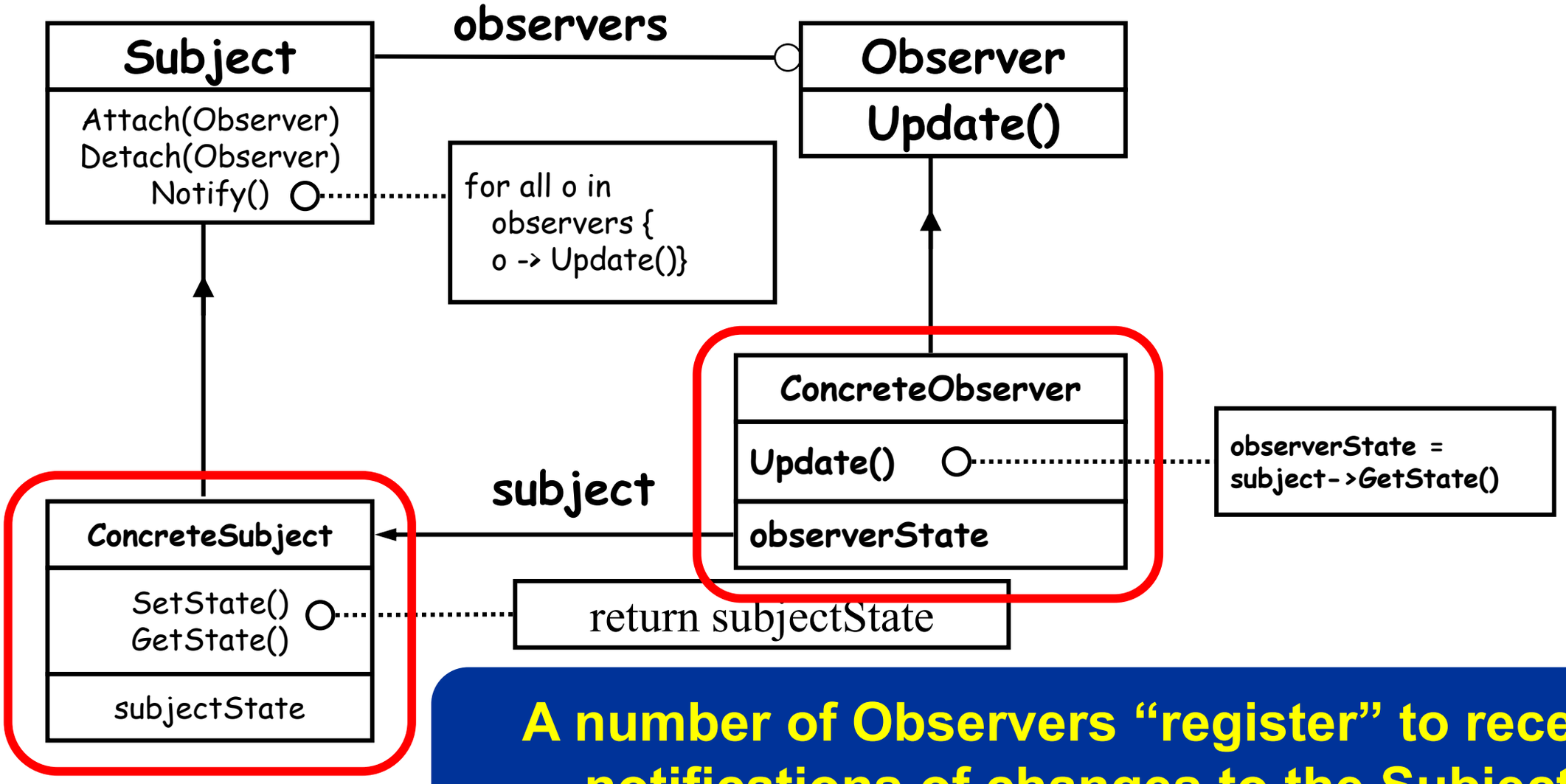
Concrete Subject Stores "state of interest" to observers and Sends notification when state changes

Observer Pattern - UML



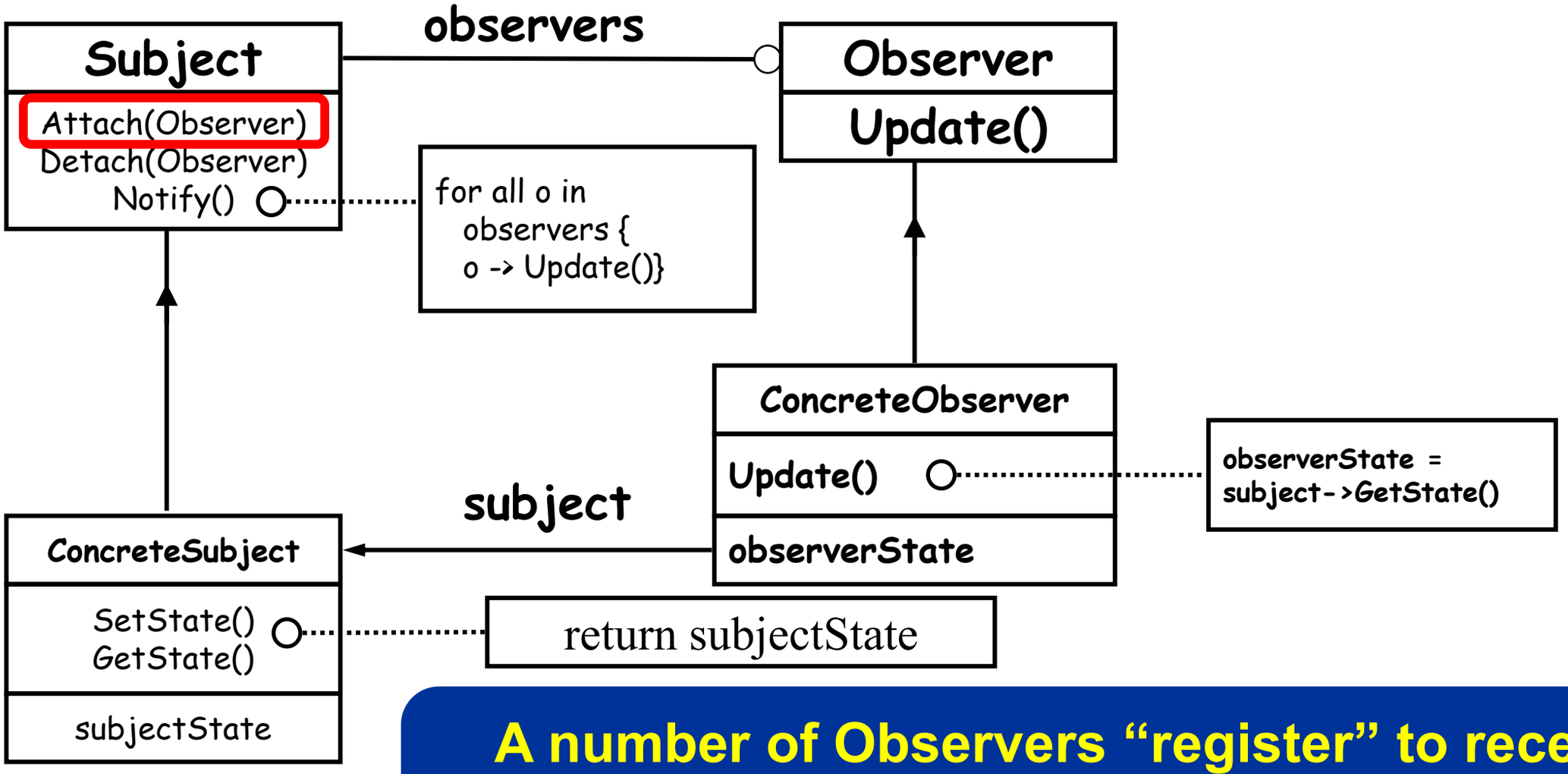
Concrete Observer Implements updating interface

Observer Pattern - UML



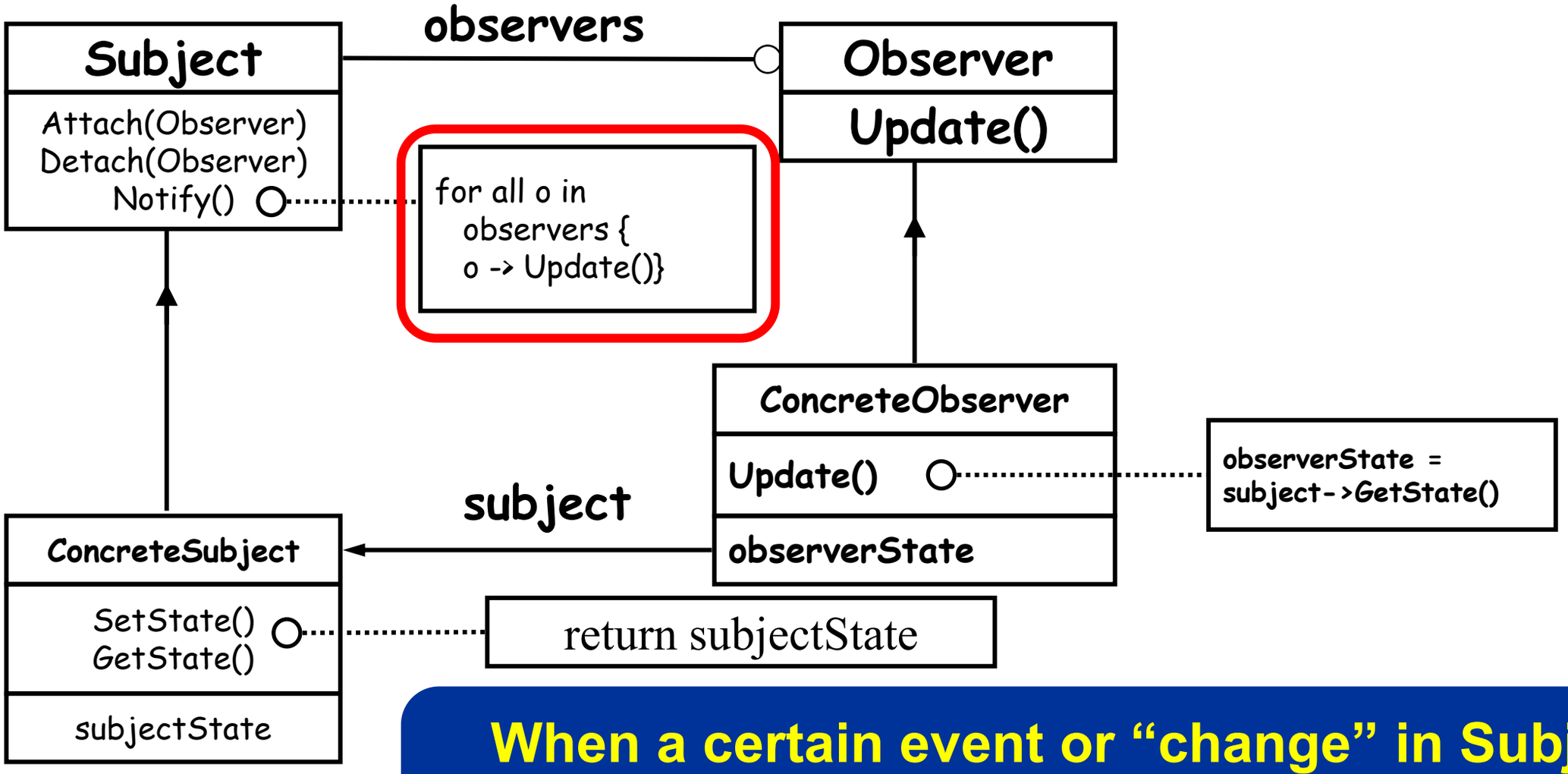
A number of Observers "register" to receive notifications of changes to the Subject.

Observer Pattern - UML



A number of Observers "register" to receive notifications of changes to the Subject.

Observer Pattern - UML



When a certain event or "change" in Subject occurs, all Observers are "notified'.

Observer Pattern - Consequences

- Loosely Coupled
  - Reuse subjects without reusing their observers, and vice versa
  - Add observers without modifying the subject or other observers
- Abstract coupling between subject and observer
  - Concrete class of none of the observers is known
- Support for broadcast communication
  - Subject doesn’t need to know its receivers

Software Design and Architecture

Topic#082

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 083 – 092

Refactoring – Part I

© Virtual University of Pakistan

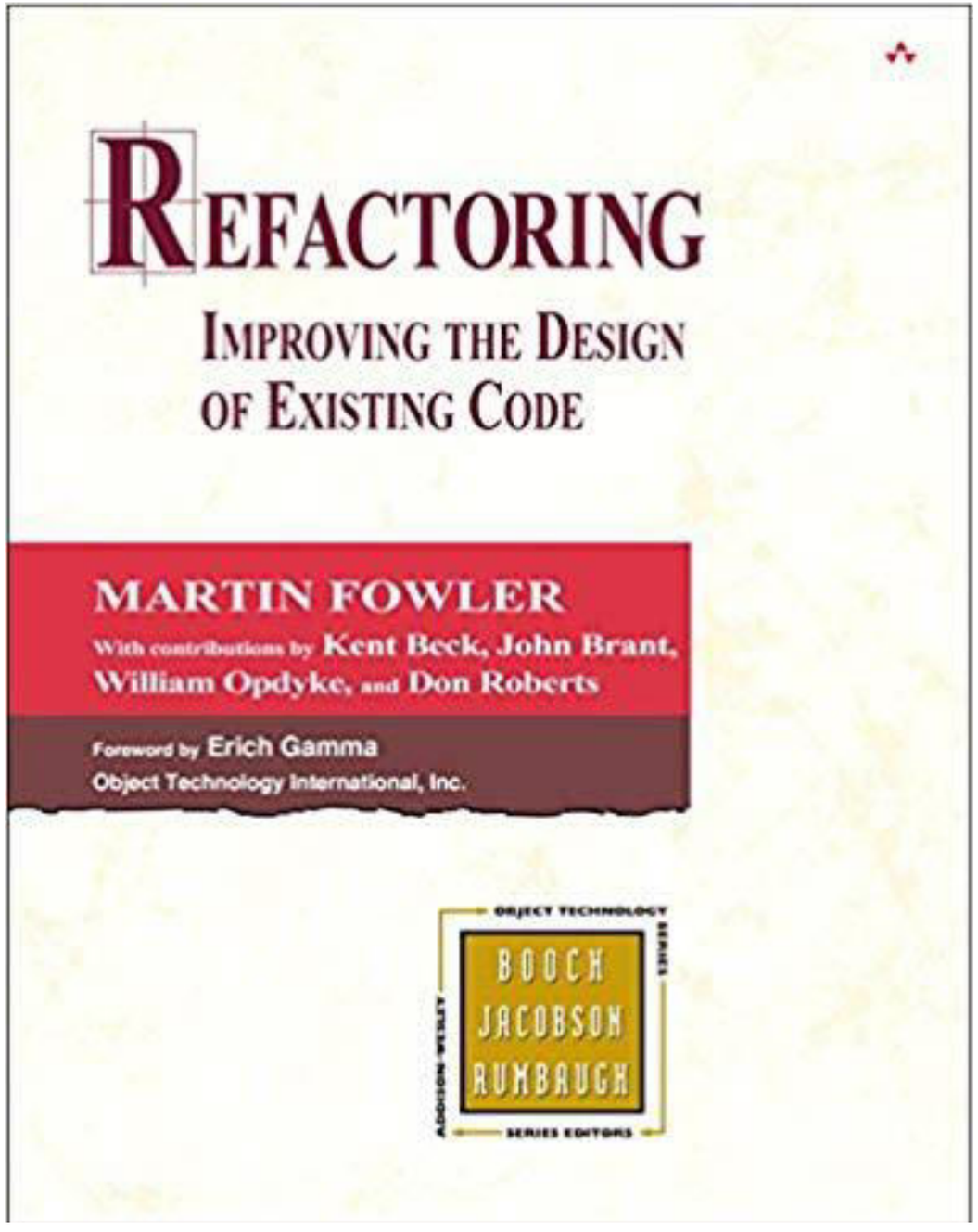
Software Design and Architecture

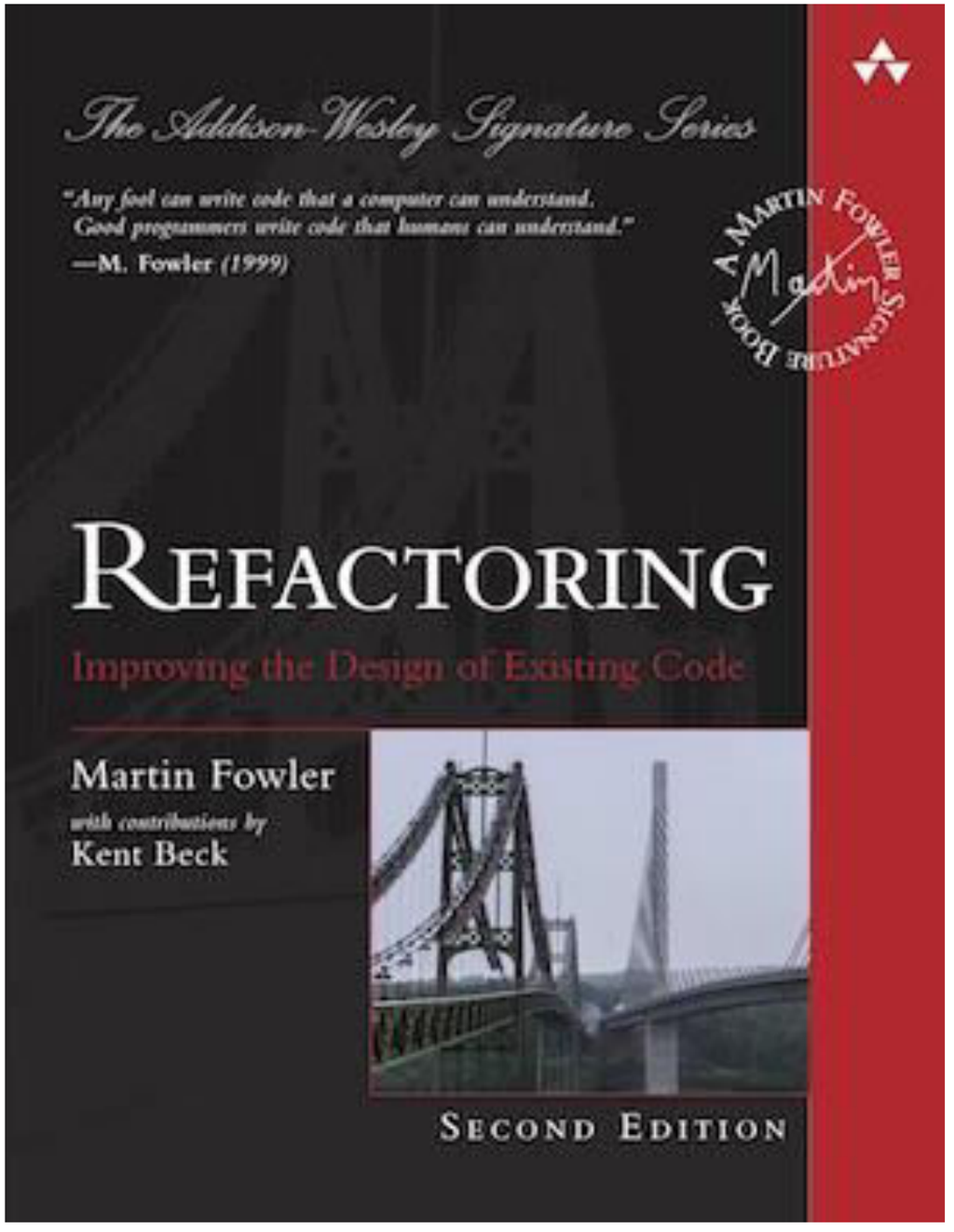
Topic#083
Refactoring Introduction

What is Refactoring?

- Refactoring is the process of changing a software system such that:
  - the external behavior of the system does not change
  - the internal structure of the system is improved
  - This is sometimes called “Improving the design after it has been written”

© Virtual University of Pakistan
thegradup.com


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International, Inc.
OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS


The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A MARTIN FOWLER SIGNATURE BOOK
Martin
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
SECOND EDITION

Why is Refactoring Useful?

- **The idea behind refactoring is to acknowledge that it will be difficult to get a design right the first time**
  - **as a program's requirements change, the design may need to change**
  - **refactoring provides techniques for evolving the design in small incremental steps**

© Virtual University of Pakistan

Refactoring Benefits

- **Refactoring improves the design of software without refactoring, a design will “decay” as people make changes to a software system**
- **Refactoring makes software easier to understand because structure is improved, duplicated code is eliminated, etc.**

© Virtual University of Pakistan
thegradup.com

Refactoring Benefits

- **Refactoring helps you find bugs**
- **Refactoring promotes a deep understanding of the code at hand, and this understanding aids the programmer in finding bugs and anticipating potential bugs**
- **Refactoring helps you program faster because a good design enables progress**

© Virtual University of Pakistan

Software Design and Architecture

Topic#083

END!

Software Design and Architecture

Topic#084
Refactoring – A simple example

A (Very) Simple Example

Consolidate Duplicate Conditional Fragments

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send ();
}
else {
    total = price * 0.98;
    send ();
}
```



```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;

send ();
```

© Virtual University of Pakistan

thegradup.com

A (Very) Simple Example

Better Still

```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;

send ();
```



```
if (isSpecialDeal())
    factor = 0.95;
else
    factor = 0.98;

total = price * factor;
send ();
```

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#084

END!

Software Design and Architecture

Topic#085
Refactoring vs Rewriting and optimization

© Virtual University of Pakistan

Refactoring versus Rewriting

- Code has to work mostly correctly before you refactor.
- If the current code just does not work, do not refactor.
  - Rewrite instead!

© Virtual University of Pakistan
thegradup.com

Optimization versus Refactoring

- The purpose of refactoring is
  - to make software easier to understand and modify
- Performance optimizations often involve making code harder to understand (but faster!)

© Virtual University of Pakistan

Optimization versus Refactoring

```
for (i=0; i < N; i++)
{
      // condition does not
      // change inside the
      // loop

      if (cond1)          {//1…}
      else if (cond2)     {//2…}
      else                {//3…}
      // xyz
}
```



```
if (cond1) {
      for (i=0; i < N; i++) {
            //1…
            // xyz
      }
else if (cond2) {
      for (i=0; i < N; i++) {
            //2…
            // xyz
       }
else if (cond2) {
      for (i=0; i < N; i++) {
            //2…
            // xyz
       }
}
```

© Virtual University of Pakistan

Software Design and Architecture

Topic#085

END!

Software Design and Architecture

Topic#086
Making Refactoring Safe

How do you make refactoring safe?

1. **use refactoring “patterns”**
   - **Fowler’s book assigns “names” to refactorings in the same way that the GoF’s book assigned names to patterns**

© Virtual University of Pakistan

How do you make refactoring safe?

2. **Test constantly!**
   - **This ties into the extreme programming paradigm, you write tests before you write code, after you refactor code, you run the tests and make sure they all still pass**
   - **if a test fails, the refactoring broke something, but you know about it right away and can fix the problem before you move on**

© Virtual University of Pakistan

Software Design and Architecture

Topic#086

END!

Software Design and Architecture

Topic#087
Code Smells – Bad smells in code

© Virtual University of Pakistan

Refactoring: Where to Start?

- How do you identify code that needs to be refactored?
  - “Bad Smells” in Code

© Virtual University of Pakistan

“if it stinks, change it”

Grandma Beck, discussing child-rearing philosophy

Nothing but good design principles or lack thereof!

22 Bad smells listed in Fowler’s book

© Virtual University of Pakistan

thegradup.com

Categories of bad smells

- Bloaters (5)
- Object-orientation abusers (4)
- Change preventers (3)
- Dispensables (5)
- Couplers (5)

Software Design and Architecture

Topic#087

END!

Software Design and Architecture

Topic#088
Code Smells – Bloaters

Bloaters (5)

- Bloaters are code, methods and classes that have increased to such enormous proportions that they are hard to work with.

- Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

© Virtual University of Pakistan

Bloaters (5)

- Long method
- Long class
- Primitive obsession
- Long parameter list
- Data Clumps

- Long Method
  - long methods are more difficult to understand
  - performance concerns with respect to lots of short methods are largely obsolete

© Virtual University of Pakistan

- Large Class
  - Large classes try to do too much, which reduces cohesion
    - violates SRP
- Long Parameter List
  - hard to understand, can become inconsistent
    - Symptom of behavioral form of god class

© Virtual University of Pakistan
thegradup.com

- Data Clumps
  - attributes that clump together but are not part of the same class
    - Lack of cohesion – SRP

- Primitive Obsession
  - characterized by a reluctance to use classes instead of primitive data types

© Virtual University of Pakistan

Software Design and Architecture

Topic#088

END!

Software Design and Architecture

Topic#089
Code Smells – Object-orientation abusers

Object-orientation Abusers (4)

- Incomplete or incorrect application of object-oriented programming principles

- Switch statement
- Temporary field
- Refused bequest
- Alternative classes with different interfaces

© Virtual University of Pakistan

- Switch Statements
  - Switch statements are often duplicated in code; they can typically be replaced by use of polymorphism (let OO do your selection for you!)
    - Typically a place where strategy or state pattern should have been used
- Temporary Field
  - An attribute of an object is only set in certain circumstances; but an object should need all of its attributes

- Refused Bequest
  - A subclass ignores most of the functionality provided by its superclass
    - violates LSP

- Alternative Classes with Different Interfaces
  - Symptom: Two or more methods do the same thing but have different signature for what they do

Software Design and Architecture

Topic#089

END!

Software Design and Architecture

Topic#090
Code Smells – Change Preventers

Change Preventers (3)

- These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too.
- Program development becomes much more complicated and expensive as a result.

- Divergent change
- Shotgun surgery
- Parallel inheritance hierarchies

- Divergent Change
  - Deals with cohesion; symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset
    - violates SRP

- Shotgun Surgery
  - a change requires lots of little changes in a lot of different classes

- Parallel Inheritance Hierarchies
  - Similar to Shotgun Surgery; each time I add a subclass to one hierarchy, I need to do it for all related hierarchies

Software Design and Architecture

Topic#090

END!

Software Design and Architecture

Topic#091
Code Smells – Dispensables

© Virtual University of Pakistan

Dispensables (5)

- A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

- Comments
- Duplicated code
- Lazy class
- Data class
- Speculative generality

- Comments (!)
  - Comments are sometimes used to hide bad code
  - “…comments often are used as a deodorant” (!)

- Duplicated Code
  - bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!

- Lazy Class
  - Each class costs money to maintain and understand.
  - A class that no longer “pays its way”
    - e.g. may be a class that was downsized by refactoring, or represented planned functionality that did not pan out

© Virtual University of Pakistan

- Data Class
  - These are classes that have fields, getting and setting methods for the fields, and nothing else; they are data holders, but objects should be about data AND behavior
    - Symptom of data form of god class

- Speculative Generality
  - “Oh I think we need the ability to do this kind of thing someday”

Software Design and Architecture

Topic#091

END!

Software Design and Architecture

Topic#092
Code Smells – Couplers

Couplers (5)

- All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

- Feature envy
- Inappropriate intimacy
- Message chain
- Middle man
- Incomplete library class

- Feature Envy
  - A method requires lots of information from some other class (move it closer!)
    - Symptom of behavioral form of god class

- Inappropriate Intimacy
  - Pairs of classes that know too much about each other's private details

- Message Chains
  - a client asks an object for another object and then asks that object for another object etc. Bad because client depends on the structure of the navigation
    - violates LoD

- Middle Man
  - If a class is delegating more than half of its responsibilities to another class, do you really need it?

- Incomplete Library Class
  - A framework class doesn’t do everything you need

© Virtual University of Pakistan

Software Design and Architecture

Topic#092

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 093 – 098

Refactoring – Part II

© Virtual University of Pakistan

Software Design and Architecture

Topic#093
Refactoring – The Catalog

Refactoring

- How to fix a code smell

"A change to the system that leaves its behavior unchanged, but enhances some nonfunctional quality – simplicity, flexibility, understandability, performance"

(Kent Beck, Extreme Programming Explained, page 179)

The Catalog

- **name.** The name is important to building a vocabulary of refactorings.
- **summary** of the situation in which you need the refactoring
  - summary of what the refactoring does
- **motivation**
- **mechanics**
- **examples**

© Virtual University of Pakistan

- Duplicated Code
  - Code repeated in multiple places
- Refactorings
  - Extract Method
  - Extract Class
  - Pull Up Method
  - Form Template Method

- Large Class
  - A class with too many instance variables or too much code
- Refactorings
  - Extract Class
  - Extract Subclass
  - Extract Interface
  - Replace Data Value with Object

The Catalog

- A few of the more common ones, include:
  - Extract Method
  - Replace Temp with Query
  - Move Method
  - Replace Conditional with Polymorphism
  - Introduce Null Object

© Virtual University of Pakistan

Software Design and Architecture

Topic#093

END!

Software Design and Architecture

Topic#094
Refactoring – Extract Method

Extract Method

- You have a code fragment that can be grouped together
  - Turn the fragment into a method whose name explains the purpose of the fragment

© Virtual University of Pakistan

Extract Method, continued

```
void printOwing(double amount) {
  printBanner()
  //print details
  System.out.println("name: " + _name);
  System.out.println("amount: " + amount);
}
```

© Virtual University of Pakistan
thegradup.com

Extract Method, continued

```
void printDetails(double amount) {
  System.out.println(“name: ” + _name);
  System.out.println(“amount: ” + amount);
}
```

© Virtual University of Pakistan
thegradup.com

Extract Method, continued

```
void printOwing(double amount) {
   printBanner()
   printDetails(amount)
}
```

© Virtual University of Pakistan

Software Design and Architecture

Topic#094

END!

Software Design and Architecture

Topic#095
Refactoring – Replace Temp with Query

© Virtual University of Pakistan

Replace Temp with Query

- You are using a temporary variable to hold the result of an expression
  - Extract the expression into a method
  - Replace all references to the temp with the expression
  - The new method can then be used in other methods

© Virtual University of Pakistan

Replace Temp with Query

```
double basePrice = _quantity * _itemPrice;

if (basePrice > 1000)
  return basePrice * 0.95;
else
  return basePrice * 0.98;
```

© Virtual University of Pakistan

```
double basePrice() {
        return _quantity * _itemPrice;
}
```

Replace Temp with Query

```
if (basePrice() > 1000)
   return basePrice() * 0.95;
else
   return basePrice() * 0.98;
...
```

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#095

END!

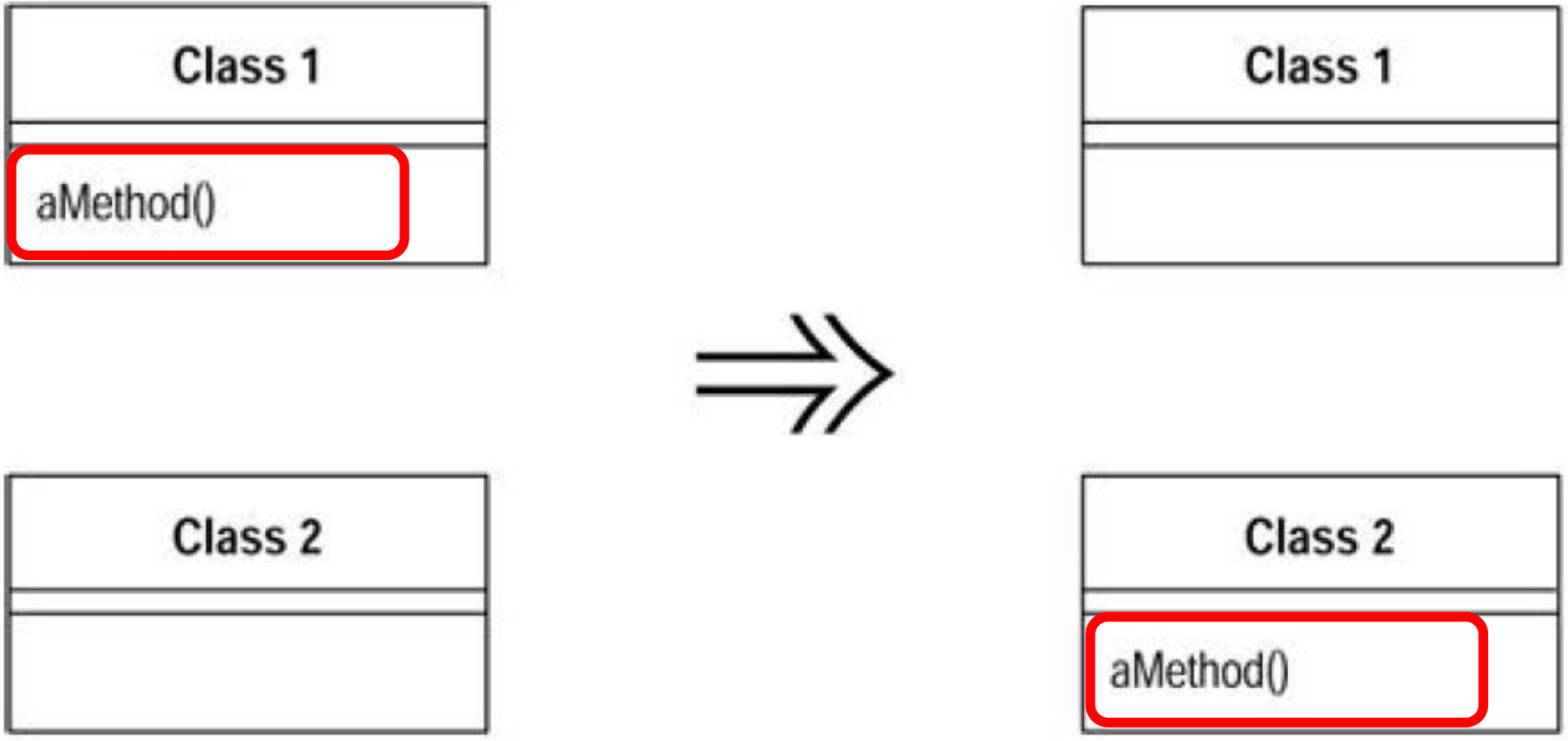
Software Design and Architecture

Topic#096
Refactoring – Move Method

Move Method

- A method is using more features (attributes and operations) of another class than the class on which it is defined
  - Create a new method with a similar body in the class it uses most.
  - Either turn the old method into a simple delegation, or remove it altogether

© Virtual University of Pakistan
thegradup.com


Class 1
aMethod()
Class 2
Class 1
Class 2
aMethod()

Software Design and Architecture

Topic#096

END!

Software Design and Architecture

Topic#097
Refactoring – Replace Conditional with Polymorphism

Replace Conditional with Polymorphism

- You have a conditional that chooses different behavior depending on the type of an object
- Move each “leg” of the conditional to an overriding method in a subclass.
- Make the original method abstract

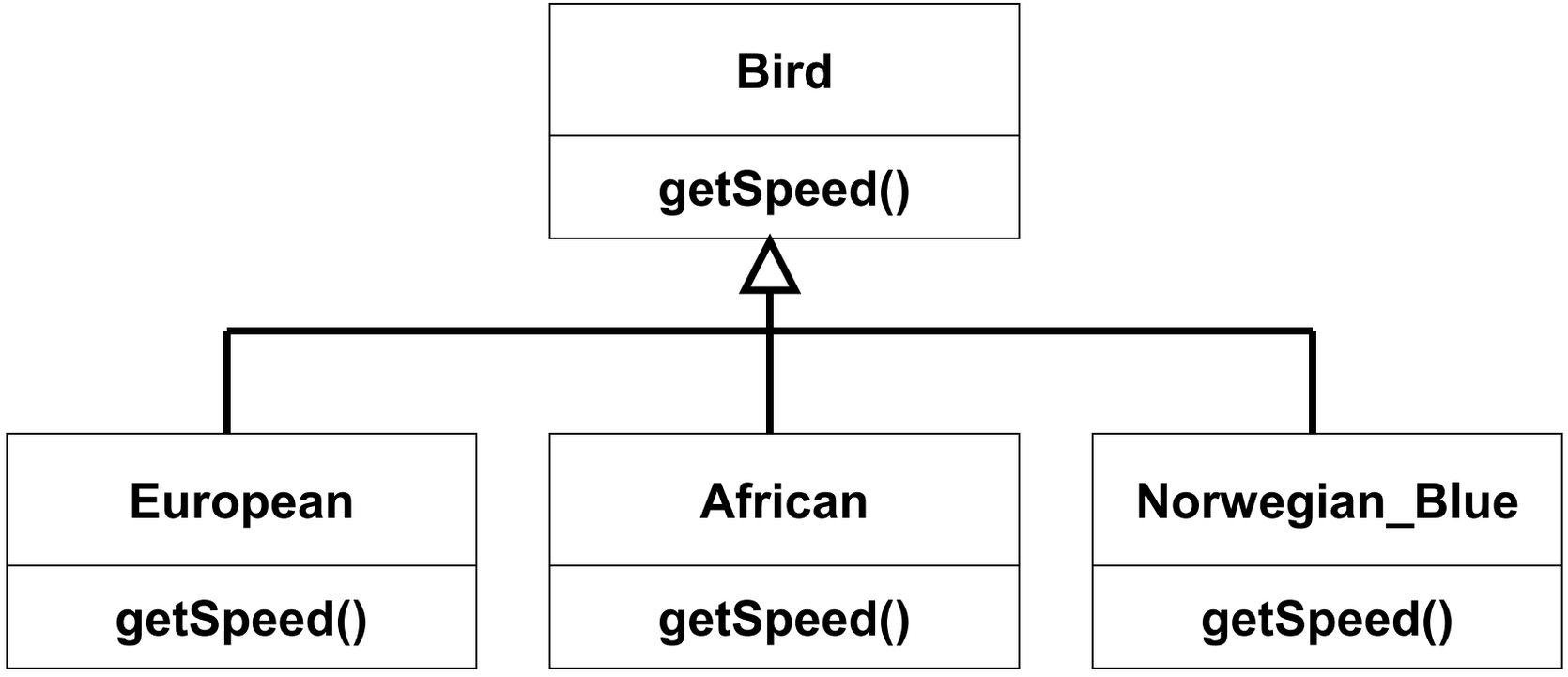
© Virtual University of Pakistan

Replace Conditional with Polymorphism

```
double getSpeed() {
  switch (_type) {
  case EUROPEAN:
        return getBaseSpeed();
  case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
  case NORWEGIAN_BLUE:
        return (_isNailed) ? 0 : getBaseSpeed(_voltage);
  }
  throw new RuntimeException("Unreachable")
}
```

© Virtual University of Pakistan

Replace Conditional with Polymorphism



© Virtual University of Pakistan

Software Design and Architecture

Topic#097

END!

Software Design and Architecture

Topic#098
Refactoring – Introduce Null Object

Introduce Null Object

- Repeated checks for a null value
- Replace the null value with a null object

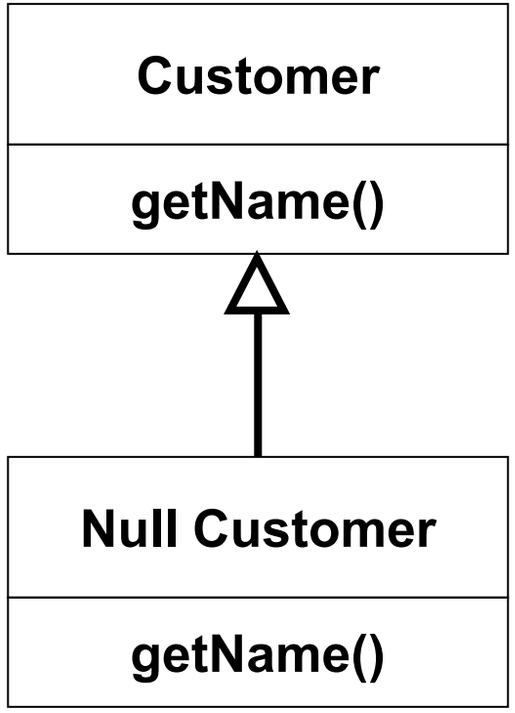
© Virtual University of Pakistan

Introduce Null Object

```
if (customer == null) {
        name = “occupant”
} else {
        name = customer.getName()
}

if (customer == null) {
…
```

Introduce Null Object



© Virtual University of Pakistan

Introduce Null Object

```
if (customer.isNull())
    name = “occupant”;
else
    name = customer.getName();
```

```
public class nullCustomer {
   public String getName() { return “occupant”;}
}
```

```
customer.getName();
The conditional goes away entirely!!
```

© Virtual University of Pakistan

Software Design and Architecture

Topic#098

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 099 – 109

Refactoring – Part III

© Virtual University of Pakistan

Software Design and Architecture

Topic#099
Refactoring – Example Part I
A Video Store

Refactoring Example

- A simple program for a video store
  - Movie
  - Rental
  - Customer
- Program is told which movies a customer rented and for how long and it then calculates:
  - the charges
  - Frequent renter points
- Customer object can print a statement (in ASCII)

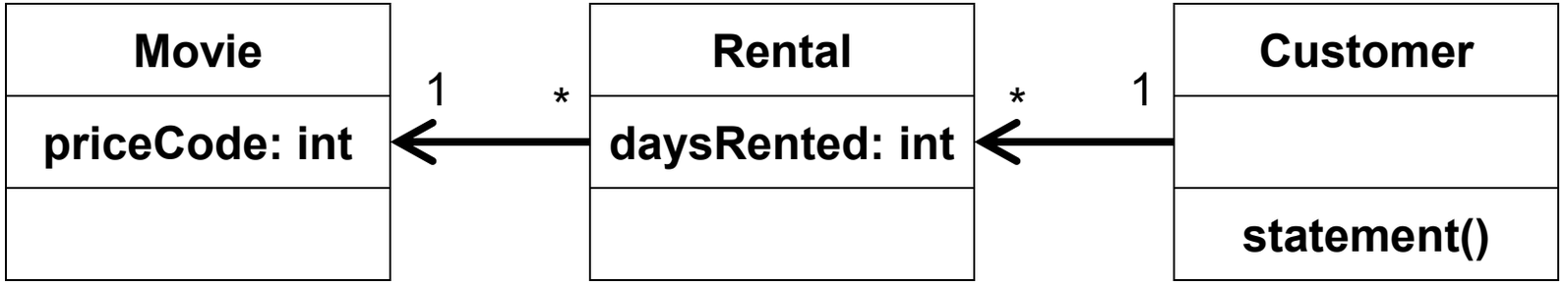
© Virtual University of Pakistan
thegradup.com

Refactoring Example

- We want to add a new method to generate an HTML statement

© Virtual University of Pakistan
thegradup.com

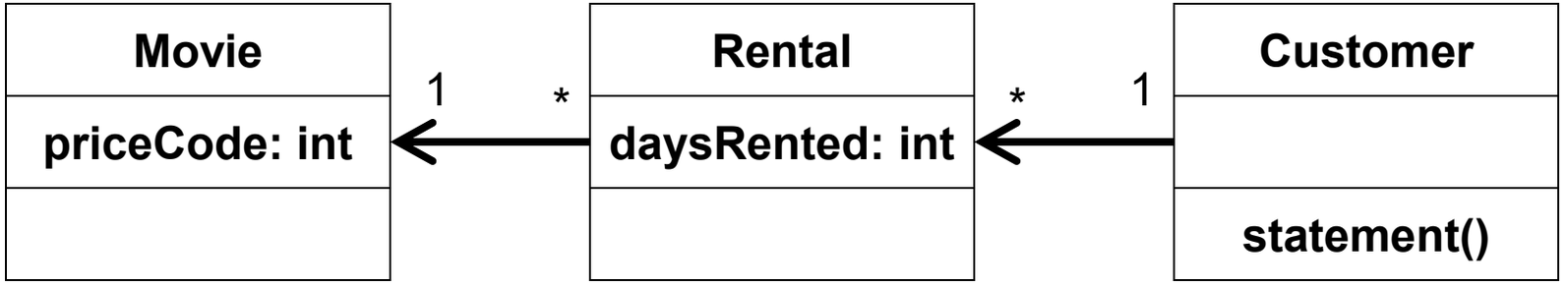
Initial Class Diagram



statement works by looping through all rentals

© Virtual University of Pakistan

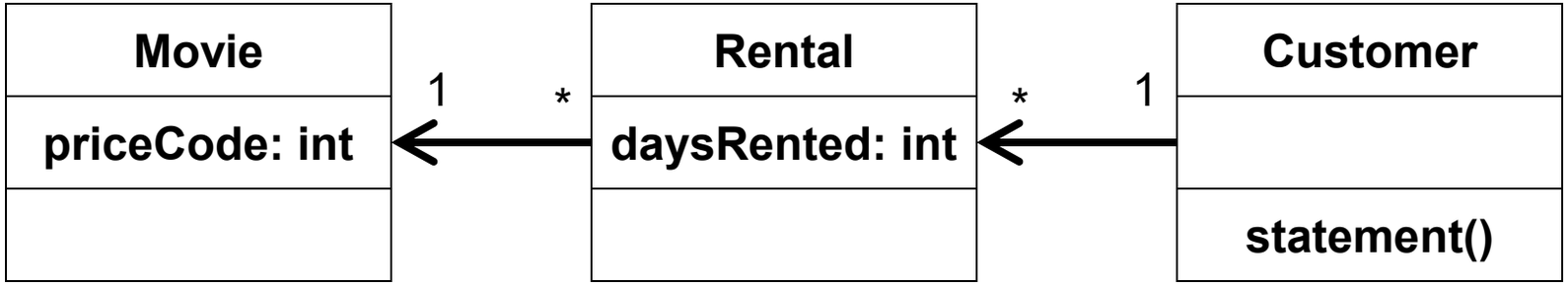
Initial Class Diagram



for each rental, it retrieves its movie and the number of days it was rented; it also retrieves the price code of the movie.

© Virtual University of Pakistan

Initial Class Diagram



it then calculates the price for each movie rental and the number of frequent renter points and returns the generated statement as a string;

© Virtual University of Pakistan

```
public class Movie {
    public static final int CHILDREN = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;
    private String _title;
    private int _priceCode;
    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode
    }
    public int getPriceCode() { return _priceCode; }
    public String getTitle() { return _title; }
    public void setPriceCode(int priceCode) {
        _priceCode = priceCode;
    }
}
```

© Virtual University of Pakistan

thegradup.com

```
class Rental {
    private Movie _movie;
    private int _daysRented;
    public Rental(Movie movie, int daysRented) {
        _movie = movie;
        _daysRented = daysRented;
    }
    public int getdaysRented() { return _daysRented; }
    public Movie getMovie() { return _movie; }
}
```

© Virtual University of Pakistan

thegradup.com

```
class Customer {
    private String _name;
    private Vector _rentals = new Vector();
    public Customer(String name) { _name = name; }
    public String getName() { return _name; }
    public void addRental (Rental arg) {
        _rentals.addElement(arg);
    }
    public String statement();
}
```

© Virtual University of Pakistan

thegradup.com

```
public String statement() {
   double totalAmount = 0;
   int frequentRenterPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = “Rental record for ”+getName() + “\n”;

   while (rentals.hasMoreElements()) {
         double thisAmount = 0;
         Rental each = (Rental) rentals.nextElement();
```

© Virtual University of Pakistan
thegradup.com

```
// determine amounts for each line

switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2)
          thisAmount += (each.getDaysRented() - 2) *1.5;
        break;
    case Movie.NEW_RELAESE:
        thisAmount += each.getDaysRented() * 3;
        break;
    case Movie.CHILDREN:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) *1.5;
        break;
}
```

© Virtual University of Pakistan

```
    // add frequent renter points

    frequentRenterPoints++;

    // add bonus for two day new release rental
    if ((each.getMovie().getPriceCode() ==
            Movie.NEW_RELEASE) &&
                        each.getDaysRented() > 1)
             frequentRenterPoints++;

    // show figures for this rental
    result += “\t” + each.getMovie().getTitle() + “\t” +
                    String.valueOf(thisAmount) + “\n”;

    totalAmount += thisAmount;

}   // end while
```

```
    // add footer lines

    result += “Amount owed is ” +
        String.valueOf(totalAmount) + “\n”;
    result += “You earned ”  +
        String.valueOf(frequentRenterPoints) + “\n”;
    return result;
}
```

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = “Rental record for ” + getName() + “\n”;

  while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        // determine amounts for each line

        switch (each.getMovie().getPriceCode()) {
           case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                  thisAmount += (each.getDaysRented() - 2) *1.5;
                break;
          case Movie.NEW_RELAESE:
                thisAmount += each.getDaysRented() * 3;
                break;
          case Movie.CHILDREN:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                  thisAmount += (each.getDaysRented() - 3) *1.5;
                break;
           }

           // add frequent renter points
           frequentRenterPoints++;
           // add bonus for two day new release rental
           if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) && each.getDaysRented() > 1) frequentRenterPoints++;

           // show figures for this rental
           result += “\t” + each.getMovie().getTitle() + “\t” +  String.valueOf(thisAmount) + “\n”;
           totalAmount += thisAmount;
  }

  // add footer lines
  result += “Amount owed is ” + String.valueOf(totalAmount) + “\n”;
  result += “You earned ”  + String.valueOf(frequentRenterPoints) + “\n”;
  return result;
}
```

Does it need refactoring?

- For such a simple system
  - probably not
- But it smells bad - long method
  - Besides, our purpose is to add a new method to generate an HTML statement and refactoring statement() may lead to code that can be used by the new function.

- matches one of Fowler's conditions for refactoring: **cleaning up the code to make it possible to add new function**

© Virtual University of Pakistan

Software Design and Architecture

Topic#099

END!

Software Design and Architecture

Topic#100
Refactoring – Example Part II
Extract Method

How to start?

- We want to decompose statement() into smaller pieces which are easier to manage and move around.
- We'll start with "Extract Method"
  - target the switch statement first

© Virtual University of Pakistan

```
switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2)
          thisAmount += (each.getDaysRented() - 2) *1.5;
        break;
    case Movie.NEW_RELAESE:
        thisAmount += each.getDaysRented() * 3;
        break;
    case Movie.CHILDREN:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) *1.5;
        break;
}
```

Extract method

© Virtual University of Pakistan

```
switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2)
          thisAmount += (each.getDaysRented() - 2) *1.5;
        break;
   case Movie.NEW_RELAESE:
        thisAmount += each.getDaysRented() * 3;
        break;
   case Movie.CHILDREN:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) *1.5;
        break;
}
```

Watch for local variables each and thisAmount

© Virtual University of Pakistan

thegradup.com

```
switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2)
          thisAmount += (each.getDaysRented() - 2) *1.5;
        break;
   case Movie.NEW_RELAESE:
        thisAmount += each.getDaysRented() * 3;
        break;
   case Movie.CHILDREN:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) *1.5;
        break;
}
```

non-modifiable variable can be passed as parameter to new method (if required)

© Virtual University of Pakistan

```
switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2)
          thisAmount += (each.getDaysRented() - 2) *1.5;
        break;
    case Movie.NEW_RELAESE:
        thisAmount += each.getDaysRented() * 3;
        break;
    case Movie.CHILDREN:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) *1.5;
        break;
}
```

modified variables require more care; since there is only one, we can make it the return value of the new method.

© Virtual University of Pakistan

thegradup.com

```
switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2)
          thisAmount += (each.getDaysRented() - 2) *1.5;
        break;
   case Movie.NEW_RELAESE:
        thisAmount += each.getDaysRented() * 3;
        break;
   case Movie.CHILDREN:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) *1.5;
        break;
}
```

each does not change but thisAmount does

© Virtual University of Pakistan

thegradup.com

```
switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2)
          thisAmount += (each.getDaysRented() - 2) *1.5;
        break;
   case Movie.NEW_RELAESE:
        thisAmount += each.getDaysRented() * 3;
        break;
   case Movie.CHILDREN:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) *1.5;
        break;
}
```

so each is a parameter and value is returned to thisAmount

© Virtual University of Pakistan

thegradup.com

```
private double amountFor(Rental each)
    int thisAmount = 0;
    switch (each.getMovie().getPriceCode()) {
       case Movie.REGULAR:
          thisAmount += 2;
          if (each.getDaysRented() > 2)
            thisAmount += (each.getDaysRented() - 2) *1.5;
          break;
      case Movie.NEW_RELAESE:
          thisAmount += each.getDaysRented() * 3;
          break;
      case Movie.CHILDREN:
          thisAmount += 1.5;
          if (each.getDaysRented() > 3)
            thisAmount += (each.getDaysRented() - 3) *1.5;
          break;
    }
    return thisAmount;
}
```

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#100

END!

Software Design and Architecture

Topic#101
Refactoring – Example Part III
Extract Method - Pitfalls

Be careful!

- Pitfalls
  - be careful about return types; in the original statement(), thisAmount is a double, but it would be easy to make a mistake of having the new method return an int. This will cause rounding-off error.
- Always remember to test after each change.

© Virtual University of Pakistan
thegradup.com

```
private double amountFor(Rental each)
      double thisAmount = 0;
      switch (each.getMovie().getPriceCode()) {
         case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
              thisAmount += (each.getDaysRented() - 2) *1.5;
            break;
        case Movie.NEW_RELAESE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDREN:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
              thisAmount += (each.getDaysRented() - 3) *1.5;
            break;
      }
      return thisAmount;
}
```

© Virtual University of Pakistan

Software Design and Architecture

Topic#101

END!

Software Design and Architecture

Topic#102
Refactoring – Example Part IV
Extract Method – Making it More Readable

© Virtual University of Pakistan

```
private double amountFor(Rental each)
      double thisAmount = 0;
      switch (each.getMovie().getPriceCode()) {
         case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
              thisAmount += (each.getDaysRented() - 2) *1.5;
            break;
        case Movie.NEW_RELAESE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDREN:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
              thisAmount += (each.getDaysRented() - 3) *1.5;
            break;
      }
      return thisAmount;
}
```

look at the variable names and use more suitable ones

© Virtual University of Pakistan

each and thisAmount

```
private double amountFor(Rental each)
      double thisAmount = 0;
      switch (each.getMovie().getPriceCode()) {
         case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
              thisAmount += (each.getDaysRented() - 2) *1.5;
            break;
        case Movie.NEW_RELAESE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDREN:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
              thisAmount += (each.getDaysRented() - 3) *1.5;
            break;
      }
      return thisAmount;
}
```

© Virtual University of Pakistan

each	→ aRental
thisAmount	→ result

```
private double amountFor(Rental aRental)
      double result = 0;
      switch (aRental.getMovie().getPriceCode()) {
         case Movie.REGULAR:
            result += 2;
            if (aRental.getDaysRented() > 2)
              result += (aRental.getDaysRented() - 2) *1.5;
            break;
        case Movie.NEW_RELAESE:
            result += aRental.getDaysRented() * 3;
            break;
        case Movie.CHILDREN:
            result += 1.5;
            if (aRental.getDaysRented() > 3)
              result += (aRental.getDaysRented() - 3) *1.5;
            break;
      }
      return result;
}
```

© Virtual University of Pakistan

thegradup.com

```
public String statement() {
   double totalAmount = 0;
   int frequentRenterPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = “Rental record for ” + getName() + “\n”;

   while (rentals.hasMoreElements()) {
         double thisAmount = 0;
         Rental each = (Rental) rentals.nextElement();

         thisAmount = amountFor(each);

         // the switch statement has been replaced by this
         // call to amountFor

         … the rest continues as before
```

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#102

END!

Software Design and Architecture

Topic#103
Refactoring – Example Part V
Move Method

```
private double amountFor(Rental aRental)
    double result = 0;
    switch (aRental.getMovie().getPriceCode()) {
       case Movie.REGULAR:
          result += 2;
          if (aRental.getDaysRented() > 2)
            result += (aRental.getDaysRented() - 2) *1.5;
          break;
      case Movie.NEW_RELAESE:
          result += aRental.getDaysRented() * 3;
          break;
      case Movie.CHILDREN:
          result += 1.5;
          if (aRental.getDaysRented() > 3)
            result += (aRental.getDaysRented() - 3) *1.5;
          break;
    }
    return result;
}
```

amountFor() uses information from the Rental class, but it does not use information from the Customer class where it is currently located.

© Virtual University of Pakistan

thegradup.com

Move method

- Methods should be located close to the data they operate.
  - we can get rid of the parameter this way.

© Virtual University of Pakistan
thegradup.com

Move method

- let’s also rename the method to **getCharge()** to clarify what it is doing.
- as a result, back in customer, we must delete the old method and change the call to **amountFor(each)** to **each.getCharge()**
- compile and test

© Virtual University of Pakistan

- initially replace the body with the call
- remove this method later on and call directly

```
private double amountFor(Rental aRental)
    return aRental.getCharge();
}
```

© Virtual University of Pakistan

```
while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        //thisAmount = amountFor(each);

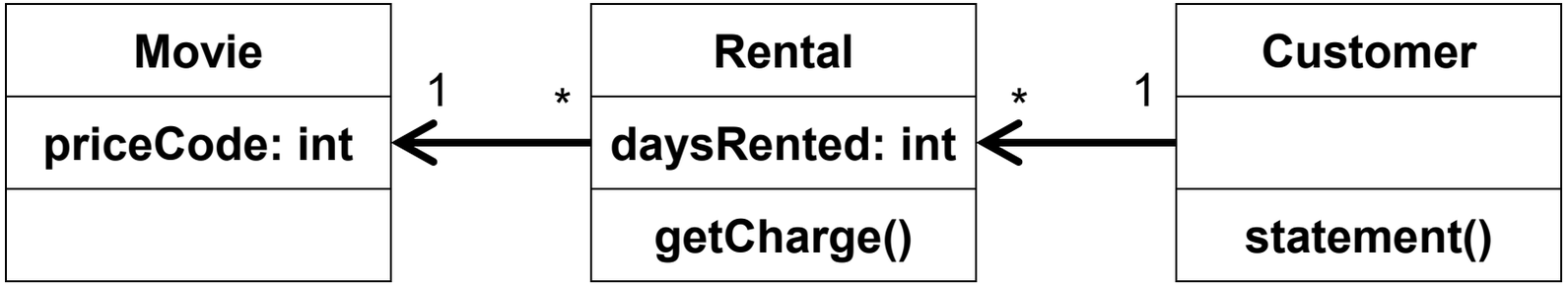
        thisAmount = each.getCharge();

        … the rest continues as before
```

- initially replace the body with the call
- remove this method later on and call directly

© Virtual University of Pakistan

New Class Diagram



- No major changes; however, Customer is now a smaller class and an operation has been moved to the class that has the data it needs to do the job

© Virtual University of Pakistan

Software Design and Architecture

Topic#103

END!

Software Design and Architecture

Topic#104
Refactoring – Example Part VI
Repeating the process for other areas

© Virtual University of Pakistan

frequent renter points

```
// add frequent renter points

frequentRenterPoints++;

// add bonus for two day new release rental
if ((each.getMovie().getPriceCode() ==
                    Movie.NEW_RELEASE) &&
                            each.getDaysRented() > 1)
        frequentRenterPoints++;

// show figures for this rental
result += “\t” + each.getMovie().getTitle() + “\t” +
                    String.valueOf(thisAmount) + “\n”;

totalAmount += thisAmount;


}   // end while
```

Let’s do the same thing with the logic to calculate frequent renter points

© Virtual University of Pakistan

frequent renter points

extract method

each can be parameter

frequentRenterPoints has a value before the method is invoked, but the new method does not read it;

we simply need to use appending assignment outside the method.

© Virtual University of Pakistan

frequent renter points

move method

Again, we are only using information from Rental, not Customer, so let’s move getFrequentRenterPoints to the Rental class.

Be sure to run your test cases after each step

© Virtual University of Pakistan

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = “Rental record for ” + getName() + “\n”;

  while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        // determine amounts for each line

        switch (each.getMovie().getPriceCode()) {
           case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                  thisAmount += (each.getDaysRented() - 2) *1.5;
                break;
          case Movie.NEW_RELAESE:
                thisAmount += each.getDaysRented() * 3;
                break;
          case Movie.CHILDREN:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                  thisAmount += (each.getDaysRented() - 3) *1.5;
                break;
           }

           // add frequent renter points
           frequentRenterPoints++;
           // add bonus for two day new release rental
           if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) && each.getDaysRented() > 1) frequentRenterPoints++;

           // show figures for this rental
           result += “\t” + each.getMovie().getTitle() + “\t” +  String.valueOf(thisAmount) + “\n”;
           totalAmount += thisAmount;
  }

  // add footer lines
  result += “Amount owed is ” + String.valueOf(totalAmount) + “\n”;
  result += “You earned ”  + String.valueOf(frequentRenterPoints) + “\n”;
  return result;
}
```

```
public String statement() {
   double totalAmount = 0;
   int frequentRenterPoints = 0;
   Enumeration rentals = _rentals.elements();
   String result = “Rental record for ” + getName() + “\n”;

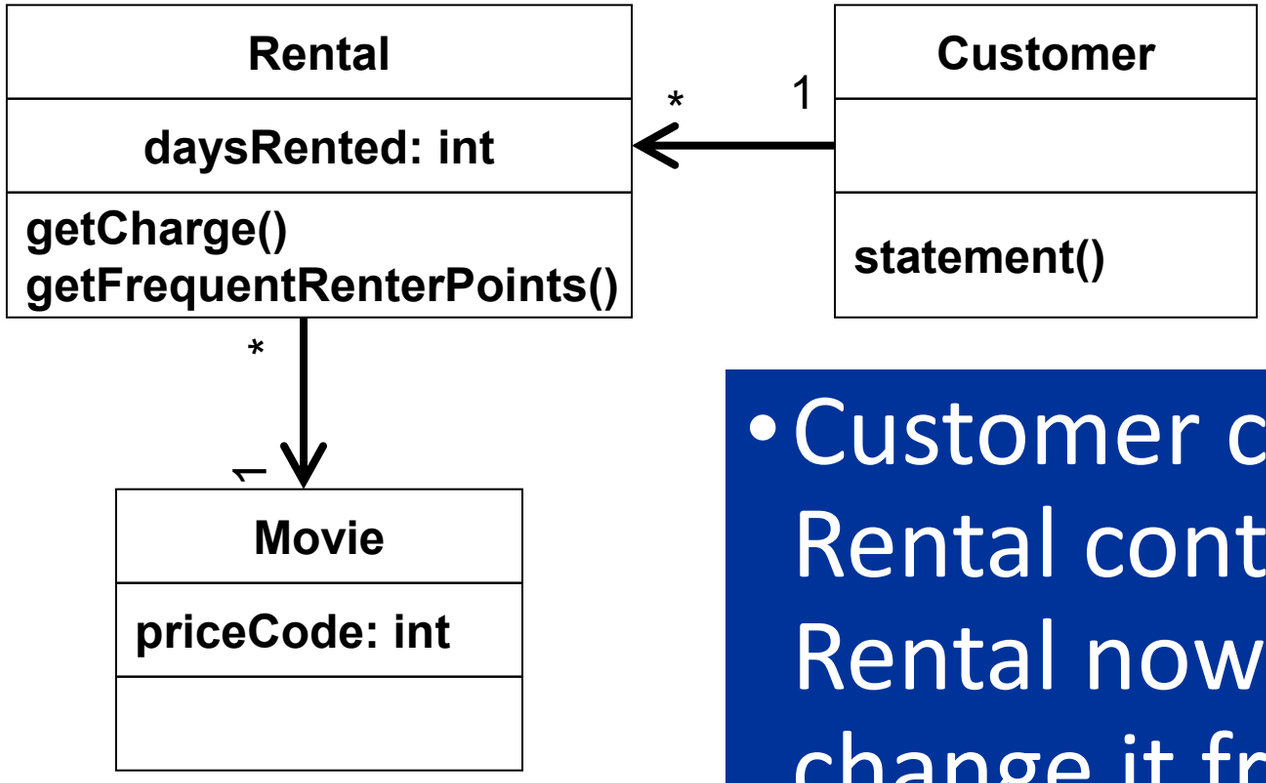
   while (rentals.hasMoreElements()) {
      double thisAmount = 0;
      Rental each = (Rental) rentals.nextElement();
      // determine amounts for each line

      thisAmount = each.getCharge();
      frequentRenterPoints += each.getFrequentRenterPoints();

      result += “\t” + each.getMovie().getTitle() + “\t” +
            String.valueOf(thisAmount) + “\n”;
      totalAmount += thisAmount;
    }

    // add footer lines
    result += “Amount owed is ” + String.valueOf(totalAmount) + “\n”;
    result += “You earned ”  + String.valueOf(frequentRenterPoints) + “\n”;
    return result;
}
```

New Class Diagram



- Customer continues to get smaller, Rental continues to get larger; but Rental now has operations that change it from being a “data holder” to a useful object.

© Virtual University of Pakistan

Software Design and Architecture

Topic#104

END!

Software Design and Architecture

Topic#105
Refactoring – Example Part VII
Adding new functionality

© Virtual University of Pakistan

Add htmlStatement()

- We are now ready to add the htmlStatement() function.

© Virtual University of Pakistan

New Requirements

- It is now anticipated that the store is going to have more than three initial types of movies
  - as a result of these new classifications, renter points and charges will vary with each new movie type.
  - as a result, we should probably move the getCharge() and getFrequentRenterPoints() methods to the Movie class.

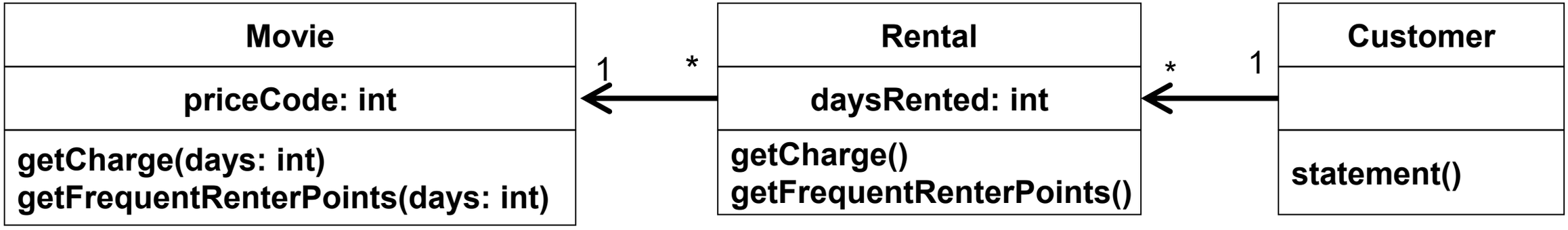
© Virtual University of Pakistan

Move Methods

- move getCharge to Movie
  - getCharge needs to know the number of days the movie was rented; since this is information that Rental has, it needs to be passed as a parameter.
- move getFrequentRenterPoints() to Movie

© Virtual University of Pakistan

New Class Diagram



© Virtual University of Pakistan

Software Design and Architecture

Topic#105

END!

Software Design and Architecture

Topic#106
Refactoring – Example Part VIII
Replace Temp with Query

Replace Temp with Query

- Removing temp variable is good thing because they often cause the need for parameters where none are required and can also cause problems in long methods

© Virtual University of Pakistan
thegradup.com

Remove Temp Variables

- statement() has two temp variables
  - totalAmount and frequentRentalPoints
- Both of these values are going to be needed by statement() and htmlStatement()

© Virtual University of Pakistan

Remove Temp Variables

- let’s replace them with query methods
  - **little more difficult because they were calculated within a loop; we have to move the loop to the query methods**

  replace totalAmount with getTotalCharge()

  replace frequentRentalPoints with getTotalFrequentPoints()

test after each point

© Virtual University of Pakistan

```
public String statement() {
  Enumeration rentals = _rentals.elements();
  String result = “Rental record for ” + getName() + “\n”;

  while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();

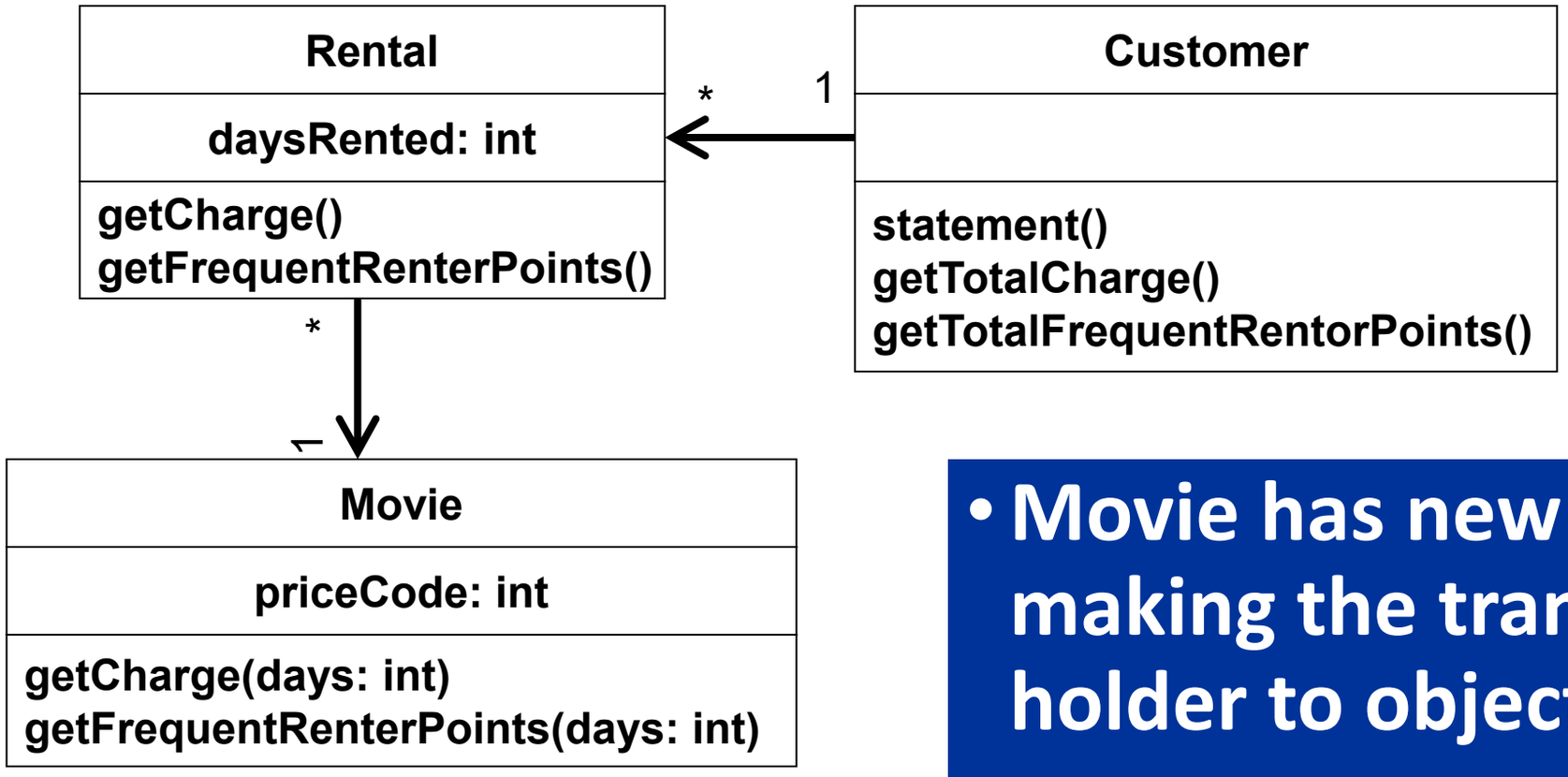
    // show figures for each rental
    result += “\t” + each.getMovie().getTitle() + “\t” +
              String.valueOf(each.getCharge()) + “\n”;
  }

  // add footer lines
  result += “Amount owed is ” +
              String.valueOf(getTotalCharge()) + “\n”;
  result += “You earned ”  +
              String.valueOf(getTotalFrequentRenterPoints()) + “\n”;
  return result;
}
```

© Virtual University of Pakistan

thegradup.com

New Class Diagram



- Movie has new methods; finally making the transition from data holder to object.
- These methods will allow us to handle new types of movies easily

© Virtual University of Pakistan

Software Design and Architecture

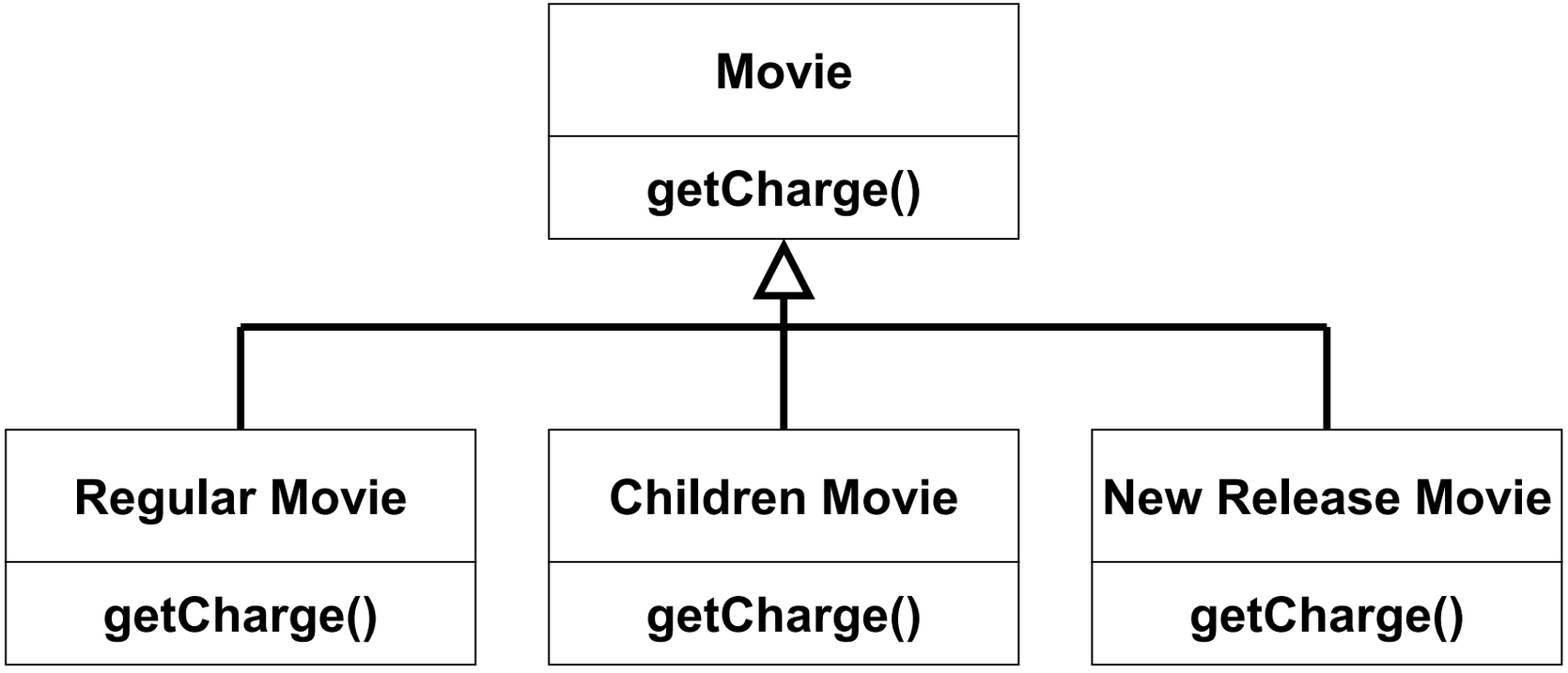
Topic#106

END!

Software Design and Architecture

Topic#107
Refactoring – Example Part IX
Handling Different Types of Objects

How to handle new movies



- **But movies can change type – a New Release Movie can later become regular movie**
- **A movie has a particular state: its charge (and its renter points) depend upon that state; so we can use the state pattern to handle new types of movies (for now, at least)**

© Virtual University of Pakistan

Replace Type Code with State/Strategy

- We need to get rid of our type code (e.g. Movie.Children) and replace it with a price object.
  - We first modify Movie to get rid of its **_priceCode** field and replace it with **_priceObject**
    - This involves changing the customer to make use of the setPriceCode() method; before it was setting priceCode directly.
    - We also have to change getPriceCode and setPriceCode to access the Price Object
  - We of course need to create Price and its subclasses

© Virtual University of Pakistan

Move Method

- Now we need to move the method getCharge to the newly created Price class
  - It is a very simple move, we just need to remember to change Movie to delegate its getCharge operation to Price

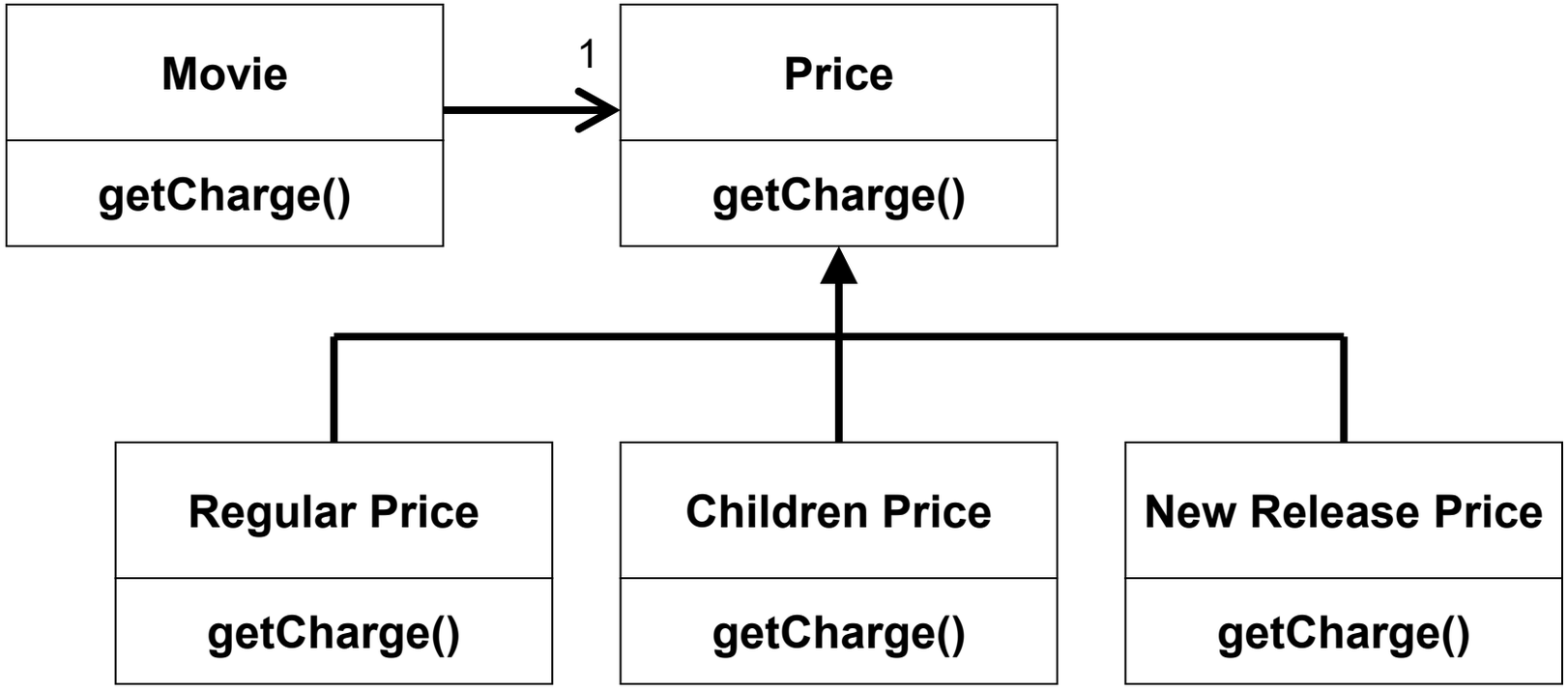
© Virtual University of Pakistan
thegradup.com

Replace Conditional with Polymorphism

- Now we move each branch of the switch statement into the appropriate subclass
- After you have done this, change Price's getCharge to an abstract method

© Virtual University of Pakistan

How to handle new movies



© Virtual University of Pakistan

Handle renter points

- Now we repeat for frequent renter points
  - We move the method over to price and use polymorphism to handle the logic
    - we leave a default implementation in Price and have newRelease override the implementation since it is the only class that returns that value.

© Virtual University of Pakistan

We’re done!

- We have added new functionality, changed “data holders” to “objects” and made it easy to add new types of movies with special charges and frequent rental points.

© Virtual University of Pakistan

Software Design and Architecture

Topic#107

END!

Software Design and Architecture

Topic#108
Refactoring – Managing Refactoring

© Virtual University of Pakistan

Why Developers are Reluctant to Refactor

- Lack of understanding
- Short-term focus
- Not paid for overhead tasks like refactoring
- Fear of breaking current program

Managing Refactoring!

- Manager's point-of-view
  - If my programmers spend time "cleaning up the code" then that's less time implementing required functionality (and my schedule is slipping as it is!)
- To address this concern
  - Refactoring needs to be systematic, incremental, and safe.

© Virtual University of Pakistan

When should you refactor?

- **The Rule of Three**
  - **Three strikes and you refactor**
  - **refers to duplication of code**
- **Refactor when you add functionality**
  - **do it before you add the new function to make it easier to add the function**
  - **or do it after to clean up the code after the function is added**
- **Refactor when you need to fix a bug**
- **Refactor as you do a code review**

© Virtual University of Pakistan

Software Design and Architecture

Topic#108

END!

Software Design and Architecture

Topic#109
Refactoring – Problems with Refactoring

© Virtual University of Pakistan

Problems with Refactoring

- Taken too far, refactoring can lead to incessant tinkering with the code, trying to make it perfect

© Virtual University of Pakistan

Problems with Refactoring

- Business applications are often tightly coupled to underlying databases
  - code is easy to change; databases are not

© Virtual University of Pakistan
thegradup.com

Problems with Refactoring

- Changing Interfaces
  - Some refactorings require that interfaces be changed
    - if you own all the calling code, no problem
    - if not, the interface is “published” and can’t change

© Virtual University of Pakistan
thegradup.com

Problems with Refactoring

- Design Changes that are difficult to refactor
  - Better to redo than to refactor
  - Software Engineers should have “courage”!

© Virtual University of Pakistan
thegradup.com

Software Design and Architecture

Topic#109

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 110 – 118

MVC Architecture

© Virtual University of Pakistan

Software Design and Architecture

Topic#110
MVC – Challenges of Interactive Application

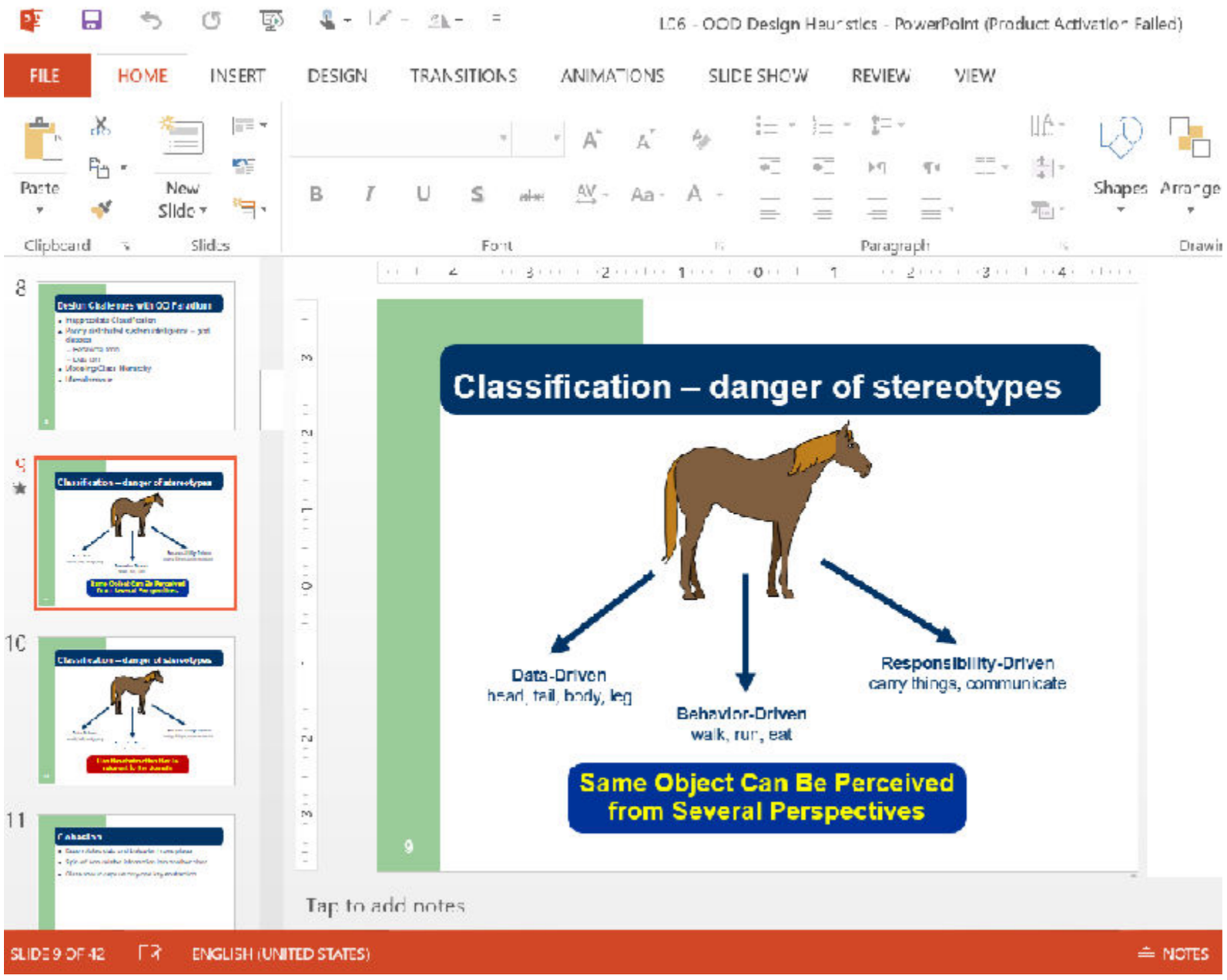
© Virtual University of Pakistan

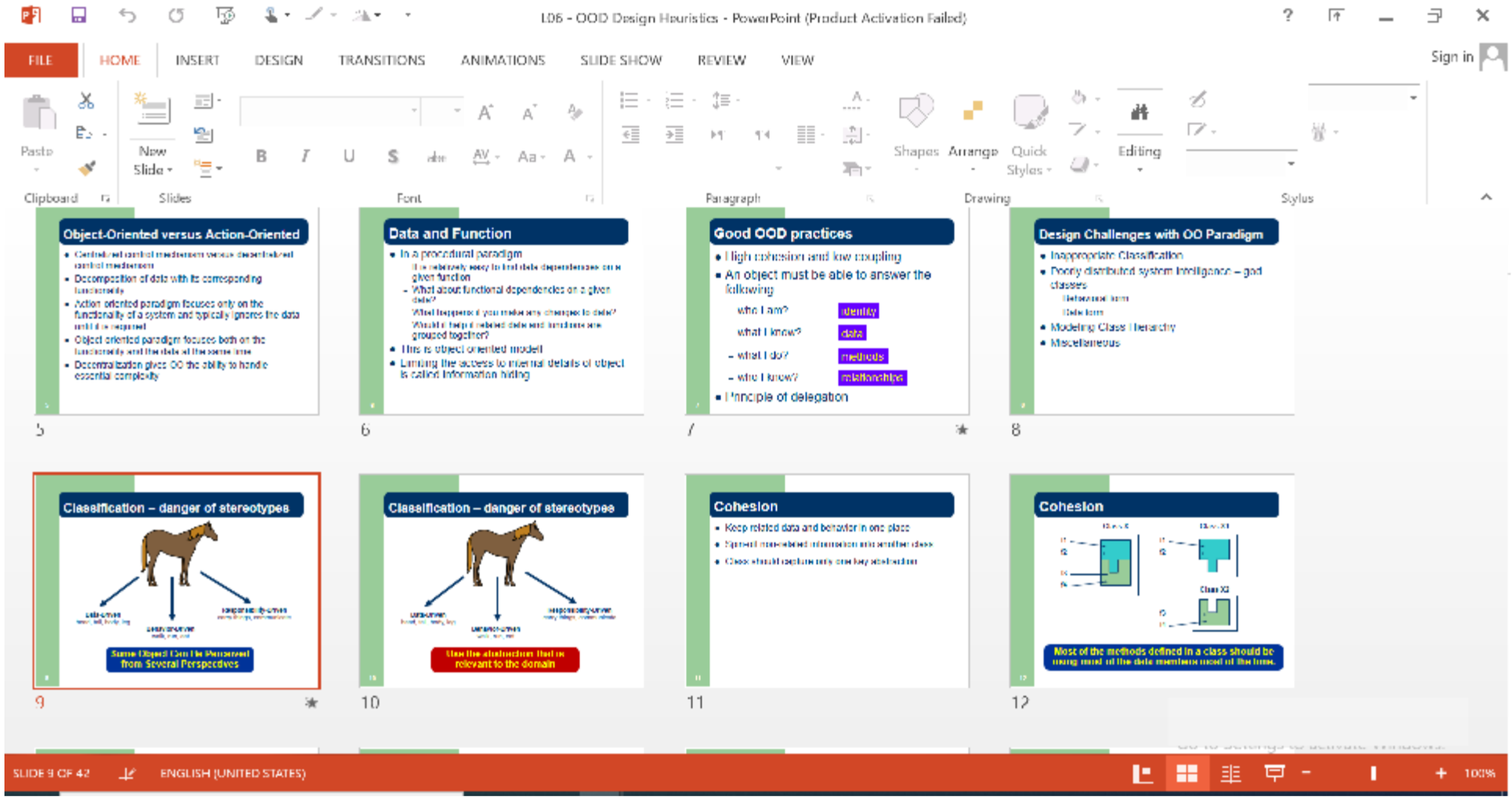
Questions

- How to reuse code among different interactive applications offering the same service?
- How to simultaneously create multiple user interfaces for same service?

Questions

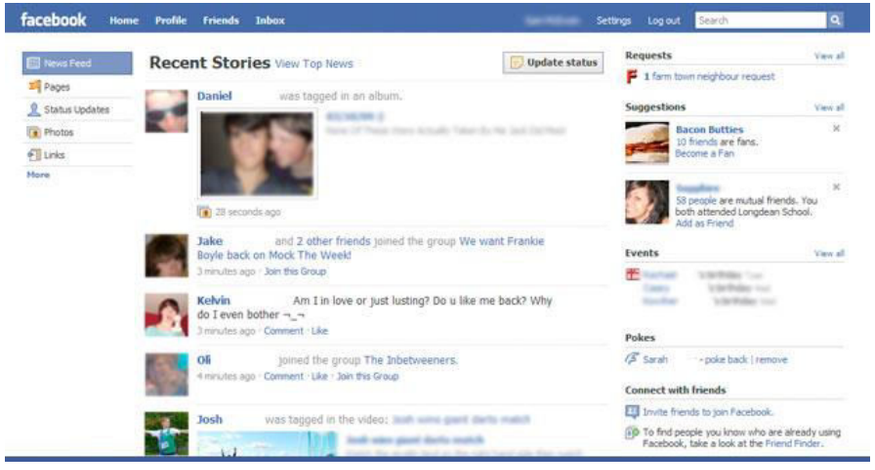
- Normal vs. Slide sorter
- Shortcuts vs. menus vs. buttons


LC6 - OOD Design Heuristics - PowerPoint (Product Activation Failed)
FILE
HOME
INSERT
DESIGN
TRANSITIONS
ANIMATIONS
SLIDE SHOW
REVIEW
VIEW
Paste
New Slide
Shapes
Arrange
Clipboard
Slides
Font
Paragraph
Classification – danger of stereotypes
Data-Driven
head, tail, body, leg
Behavior-Driven
walk, run, eat
Responsibility-Driven
carry things, communicate
Same Object Can Be Perceived from Several Perspectives
Tap to add notes
SLIDE 9 OF 42
ENGLISH (UNITED STATES)
NOTES


LD6 - OOD Design Heuristics - PowerPoint (Product Activation Failed)
FILE
HOME
INSERT
DESIGN
TRANSITIONS
ANIMATIONS
SLIDE SHOW
REVIEW
VIEW
Sign in
Object-Oriented versus Action-Oriented
Data and Function
Good OOD practices
Design Challenges with OO Paradigm
Classification – danger of stereotypes
Classification – danger of stereotypes
Cohesion
Cohesion
SLIDE 9 OF 42
ENGLISH (UNITED STATES)
100%

Questions

- How to simultaneously create multiple user interfaces for same service on different computers?
  - Facebook, email

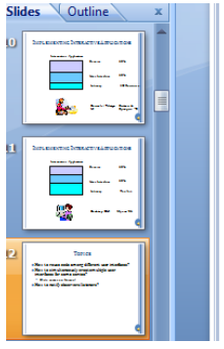




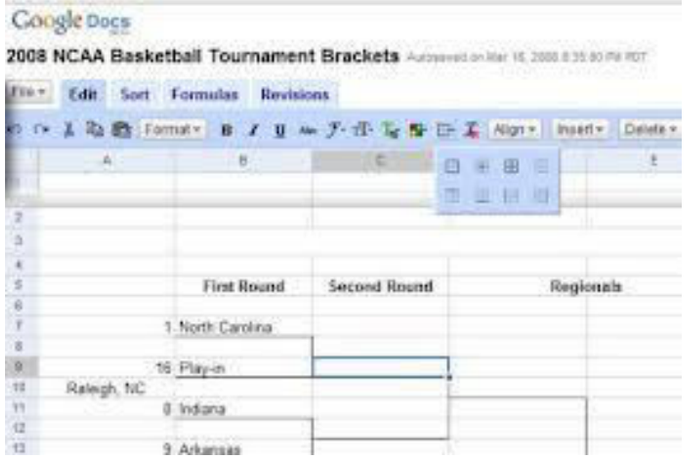
Questions

- How to simultaneously create distributed user interfaces
  - multiple complete user interfaces for different users on different computers
  - Single user-interface on large computer controlled by multiple mobile devices





- How to reus
- How to simu
  interfaces fo
  - Slide sorte
- How to noti





Software Design and Architecture

Topic#110

END!

Software Design and Architecture

Topic#111
MVC – Example Part I
Requirements

© Virtual University of Pakistan

Example: Counter

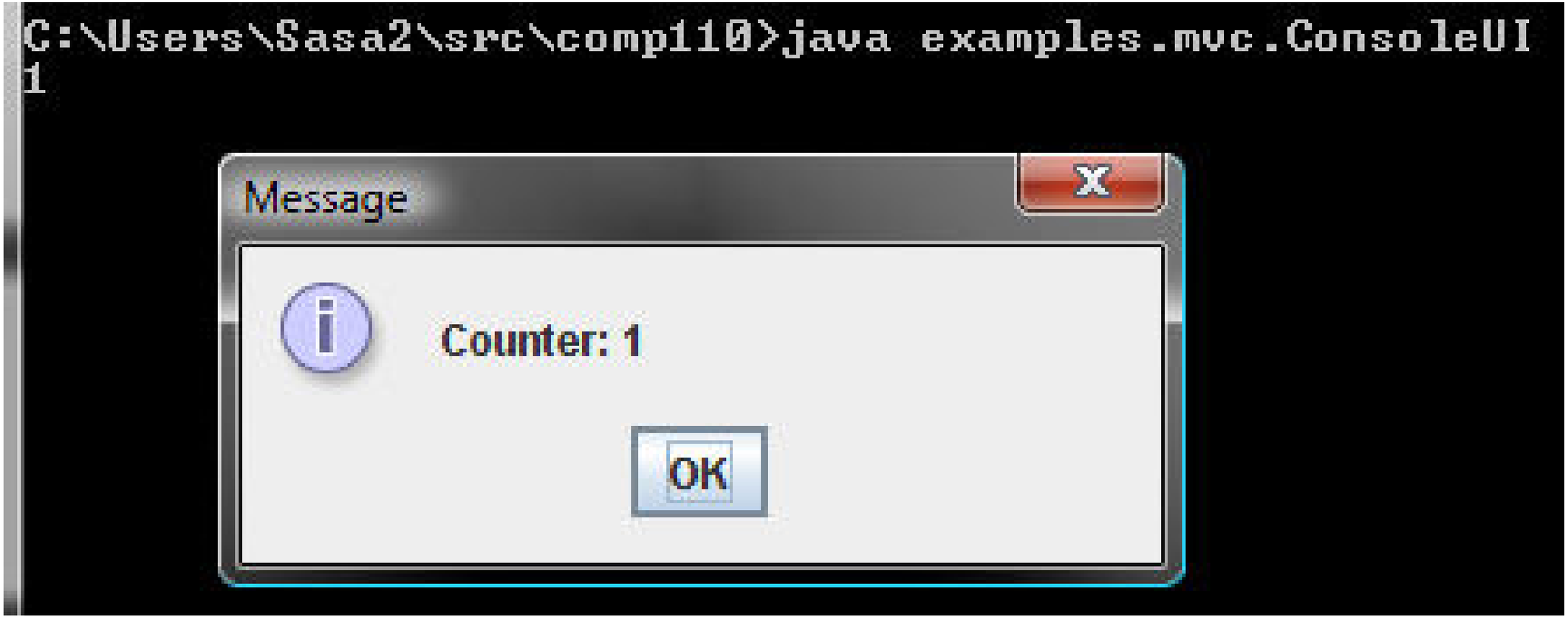
Can add arbitrary positive/negative value to an integer

Different user interfaces

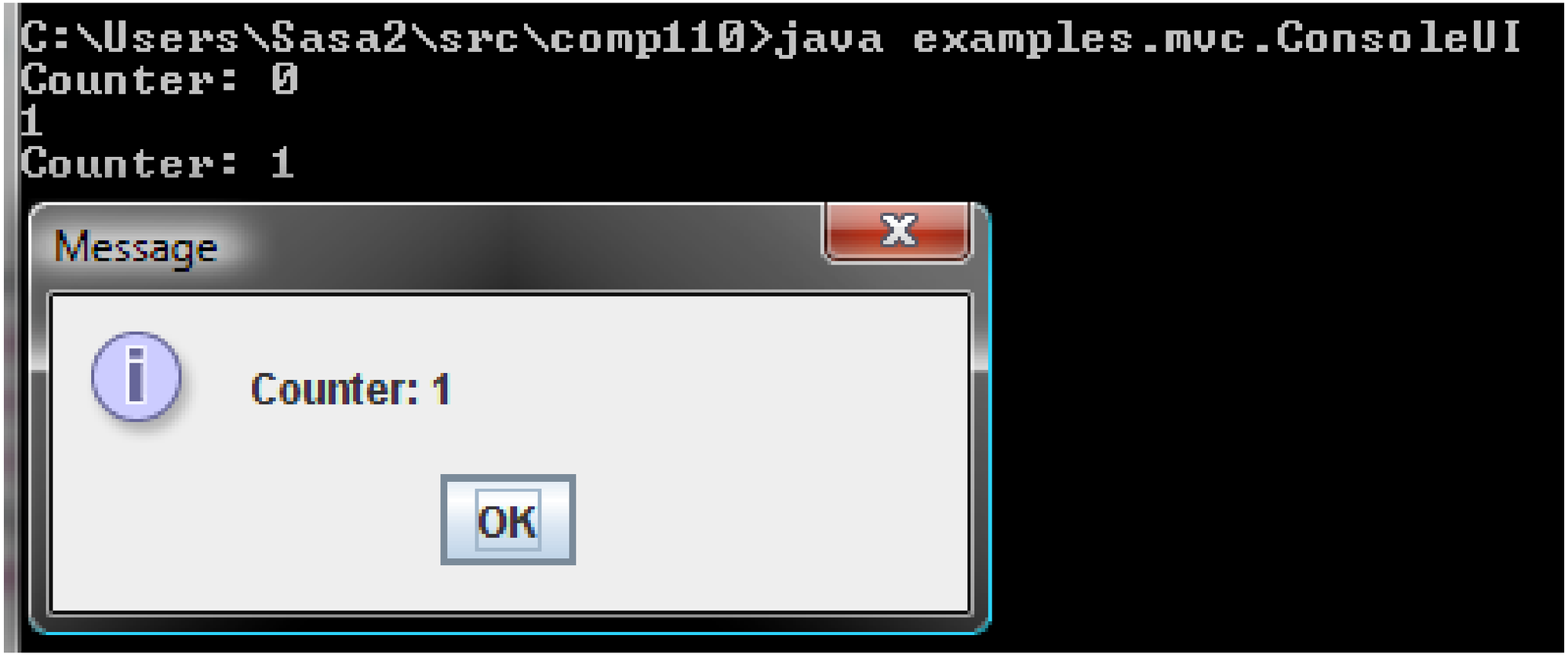
Console-based Input and Output

```
C:\Users\Sasa2\src\comp110>java examples.mvc.ConsoleUI
Counter: 0
1
Counter: 1
-1
Counter: 0
5
Counter: 5
```

Console-based Input and JOption Output



Console-based Input,Output and JOption Output



Software Design and Architecture

Topic#111

END!

Software Design and Architecture

Topic#112
MVC – Example Part II
Monolithic Console Implementation

© Virtual University of Pakistan

```
public class MonolithicConsoleUI {
public static void main(String[] args) {
        int counter = 0;
        while (true) {
            System.out.println("Counter: " + counter);
            int nextInput = Console.readInt();
            if (nextInput == 0) break;
            counter += nextInput;
        }
    }
}
```

Output

Input


C:\Users\Sasa2\src\comp110>java examples.mvc.ConsoleUI
Counter: 0
1
Counter: 1
-1
Counter: 0
5
Counter: 5

Software Design and Architecture

Topic#112

END!

Software Design and Architecture

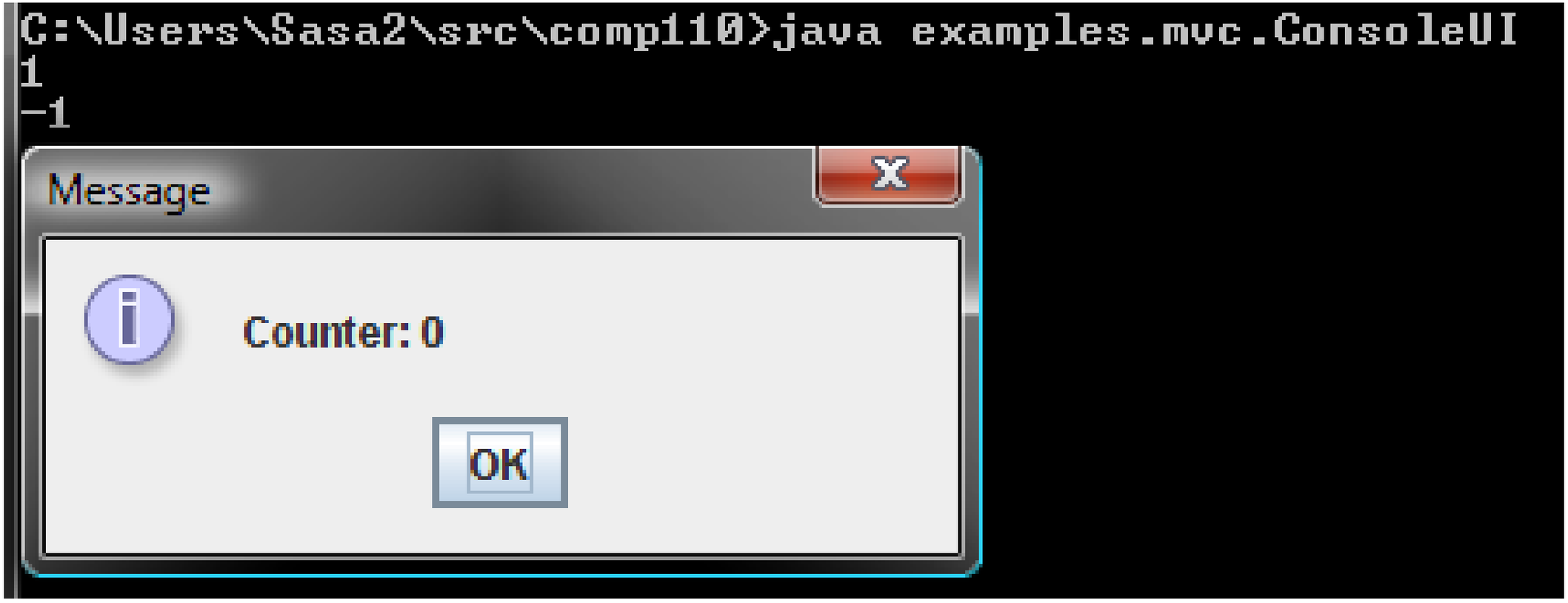
Topic#113
MVC – Example Part III
Monolithic Mixed UI Implementation

© Virtual University of Pakistan

```
import javax.swing.JOptionPane;
public class ConsoleUI {
        public static void main(String[] args) {
          int counter = 0;
          while (true) {
                JOptionPane.showMessageDialog(
                        null, “Counter: ” + counter);
                int nextInput = Console.readInt();
                if (nextInput == 0) break;
                counter += nextInput;
          }
    }
}
```

Output

Input


C:\Users\Sasa2\src\comp110>java examples.mvc.ConsoleUI
1
-1
Message
Counter: 0
OK

Software Design and Architecture

Topic#113

END!

Software Design and Architecture

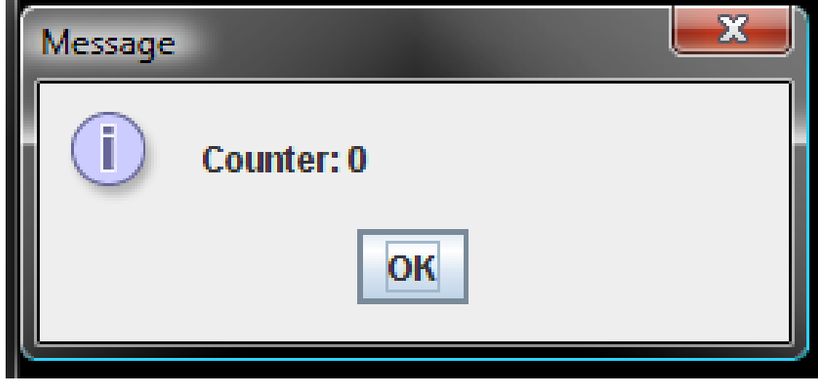
Topic#114
MVC – Example Part IV
Models and Interactors

© Virtual University of Pakistan

Multiple UIs?

```
C:\Users\Sasa2\src\comp110>java examples.mvc.ConsoleUI
Counter: 0
1
Counter: 1
-1
Counter: 0
5
Counter: 5
-
```

```
C:\Users\Sasa2\src\comp110>java examples.mvc.ConsoleUI
1
-1
```



Cannot use both UIs simultaneously to update same counter

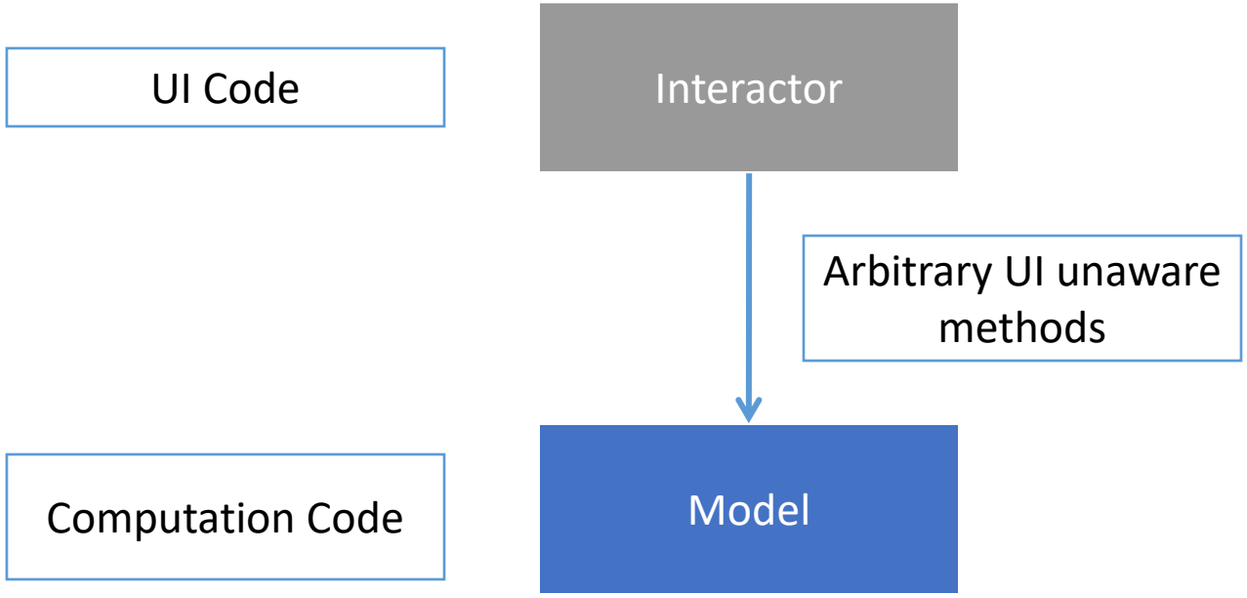
- A new application has to be developed

- Now we have three application
  - Console-based UI
  - Mixed UI
  - Multiple Mixed UI

Code Duplication

How can we remove the code duplication in these three classes?

Model/Interactor Pattern



Models and Interactors

- Model:
  - common functionality
  - counter semantics: storage and manipulation of an integer counter
- Interactor
  - different for each interface
  - counter user-Interface: user-interface to display and change the counter

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#114

END!

© Virtual University of Pakistan

Software Design and Architecture

Topic#115
MVC – Example Part V
Using Models and Interactors

© Virtual University of Pakistan

```
public class ConsoleUI {
public static void main(String[] args) {
        int counter = 0;
        while (true) {
                System.out.println("Counter: " + counter);
                int nextInput = Console.readInt();
                if (nextInput == 0) break;
                counter += nextInput;
        }
    }
}
```

Counter Model

```
public class ACounter implements Counter {
        int counter = 0;
        public void add (int amount) {
                counter += amount;
        }
        public int getValue() {
                return counter;
        }
}
```

No input/output

Any class that can be attached to a user-interface but is oblivious to the user-interface is referred to as a *model*.

Console Interactor

```
public class AConsoleUIInteractor implements CounterInteractor {
        public void edit (Counter counter) {
            while (true) {
                System.out.println("Counter: " + counter.getValue());
                int nextInput = Console.readInt();
                if (nextInput == 0) return;
                counter.add(nextInput);
            }
        }
}
```

This class does not know how the counter is implemented but is aware of the methods provided by it.

Console Interactor

```
public class AConsoleUIInteractor implements CounterInteractor {
        public void edit (Counter counter) {
            while (true) {
                System.out.println("Counter: " + counter.getValue());
                int nextInput = Console.readInt();
                if (nextInput == 0) return;
                counter.add(nextInput);
            }
        }
}
```

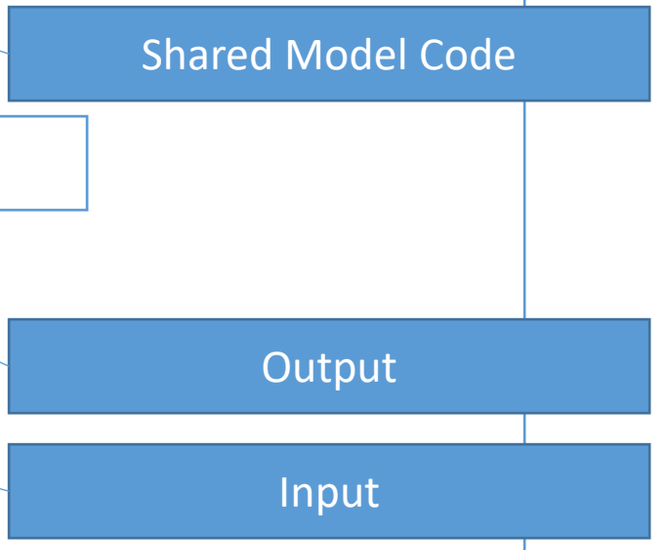
any class that implements the user-interface of a model without knowing its implementation is referred to as an *interactor*.

Putting it all together

```
public class InteractorBasedConsoleUI {
        public static void main(String args[]) {
            (new AConsoleUIInteractor()).edit(new ACounter());
        }
}
```

Mixed Interactor

```
public class AMixedUIInteractor implements CounterInteractor {
        public void edit (Counter counter) {
                while (true) {
                    JOptionPane.showMessageDialog(null,
                      "Counter: " + counter.getValue());
                    int nextInput = Console.readInt();
                    if (nextInput == 0) break;
                    counter.add(nextInput);
                }
    }
}
```



Multiple UI Interactor

```
public class AMultipleUI implements CounterInteractor {
        public void edit (Counter counter) {
                while (true) {
                    System.out.println("Counter: " + counter.getValue());
                    JOptionPane.showMessageDialog(null,
                      "Counter: " + counter.getValue());
                    int nextInput = Console.readInt();
                    if (nextInput == 0) break;
                    counter.add(nextInput);
                }
    }
}
```

Shared Model Code

Output

Input

I/O Code is Duplicated

UI Implementation is now monolithic

Software Design and Architecture

Topic#115

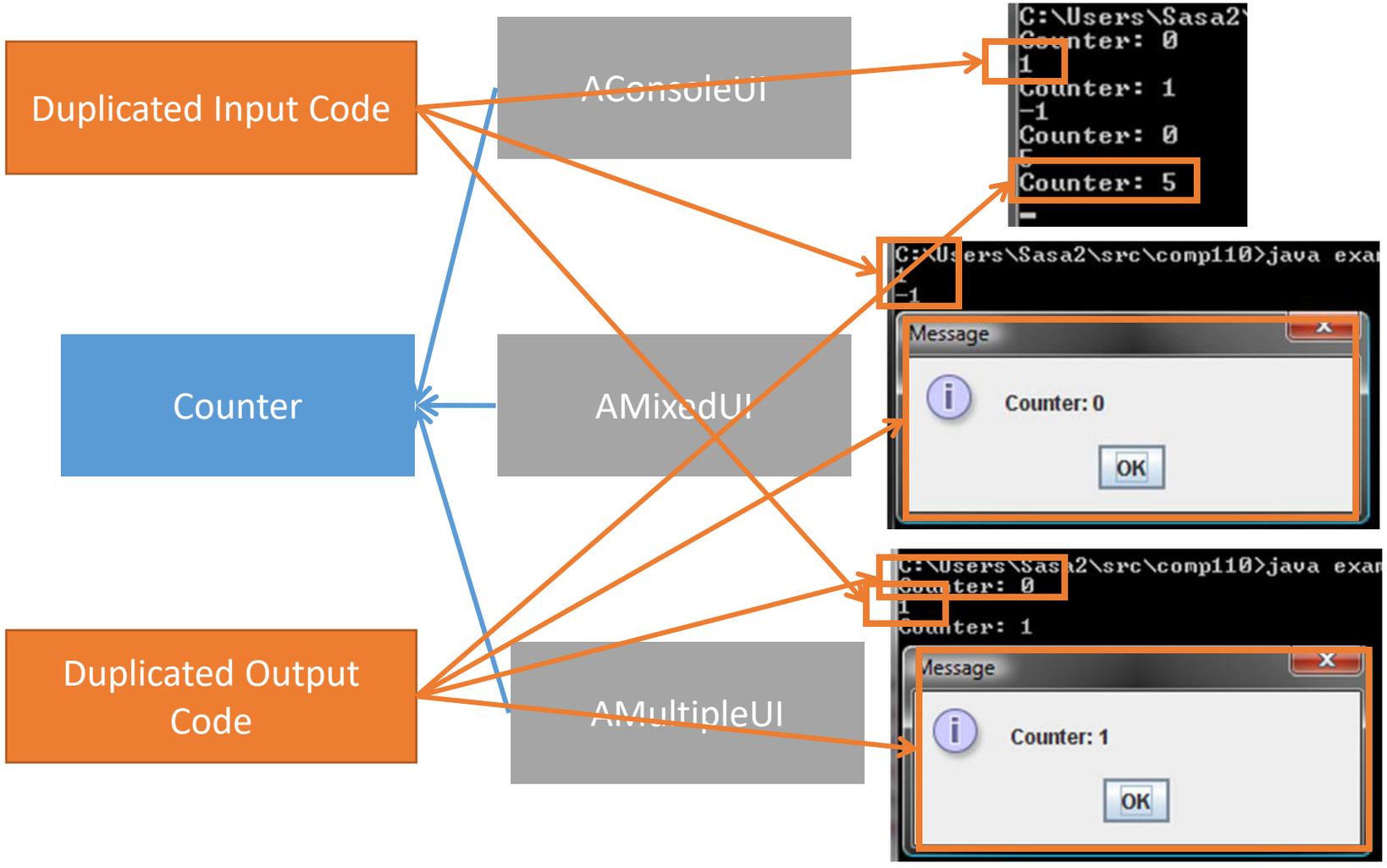
END!

Software Design and Architecture

Topic#116
MVC – Example Part VI
Drawbacks of Monolithic UI

© Virtual University of Pakistan

Drawbacks of Monolithic UI



Software Design and Architecture

Topic#116

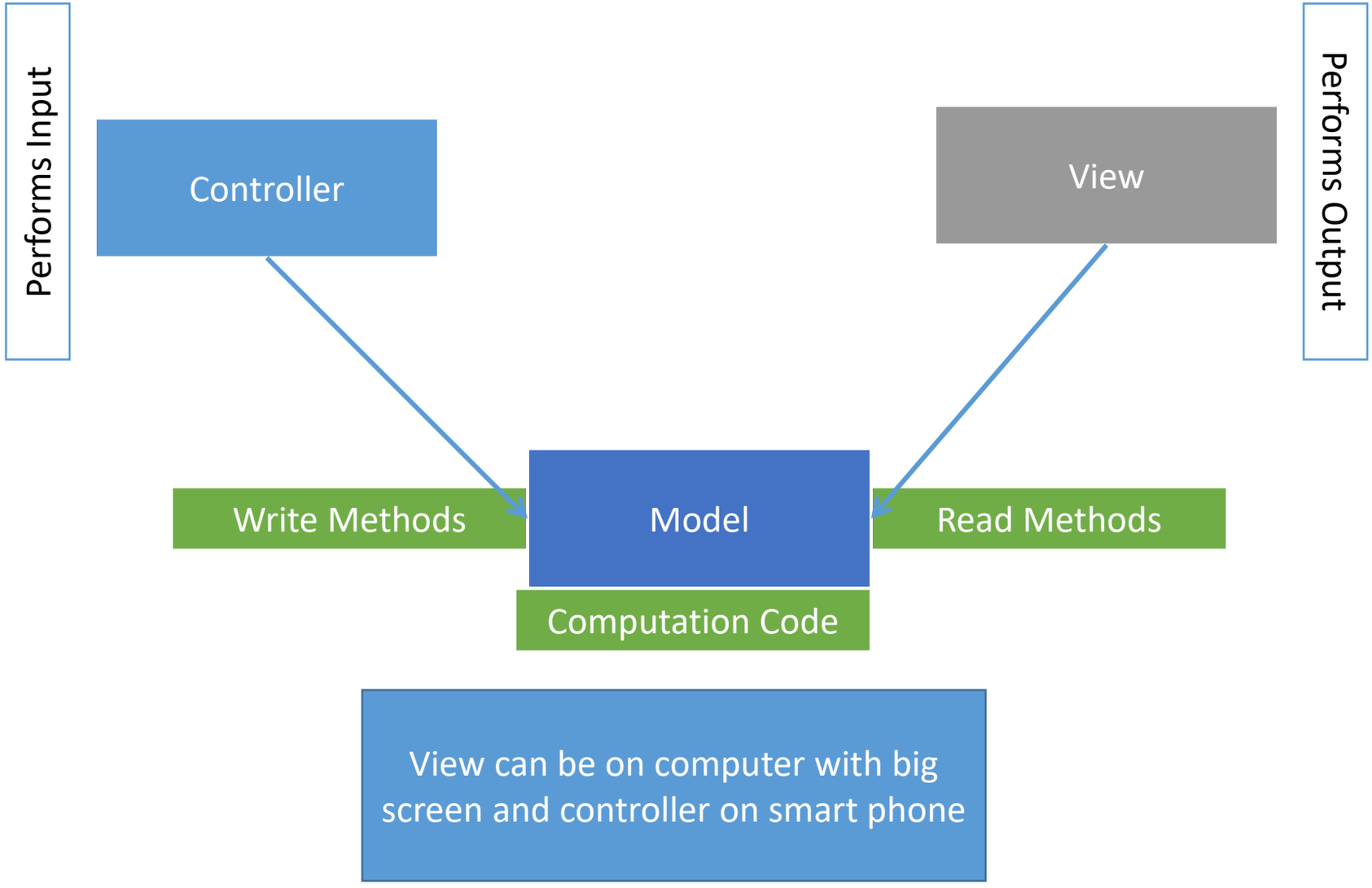
END!

Software Design and Architecture

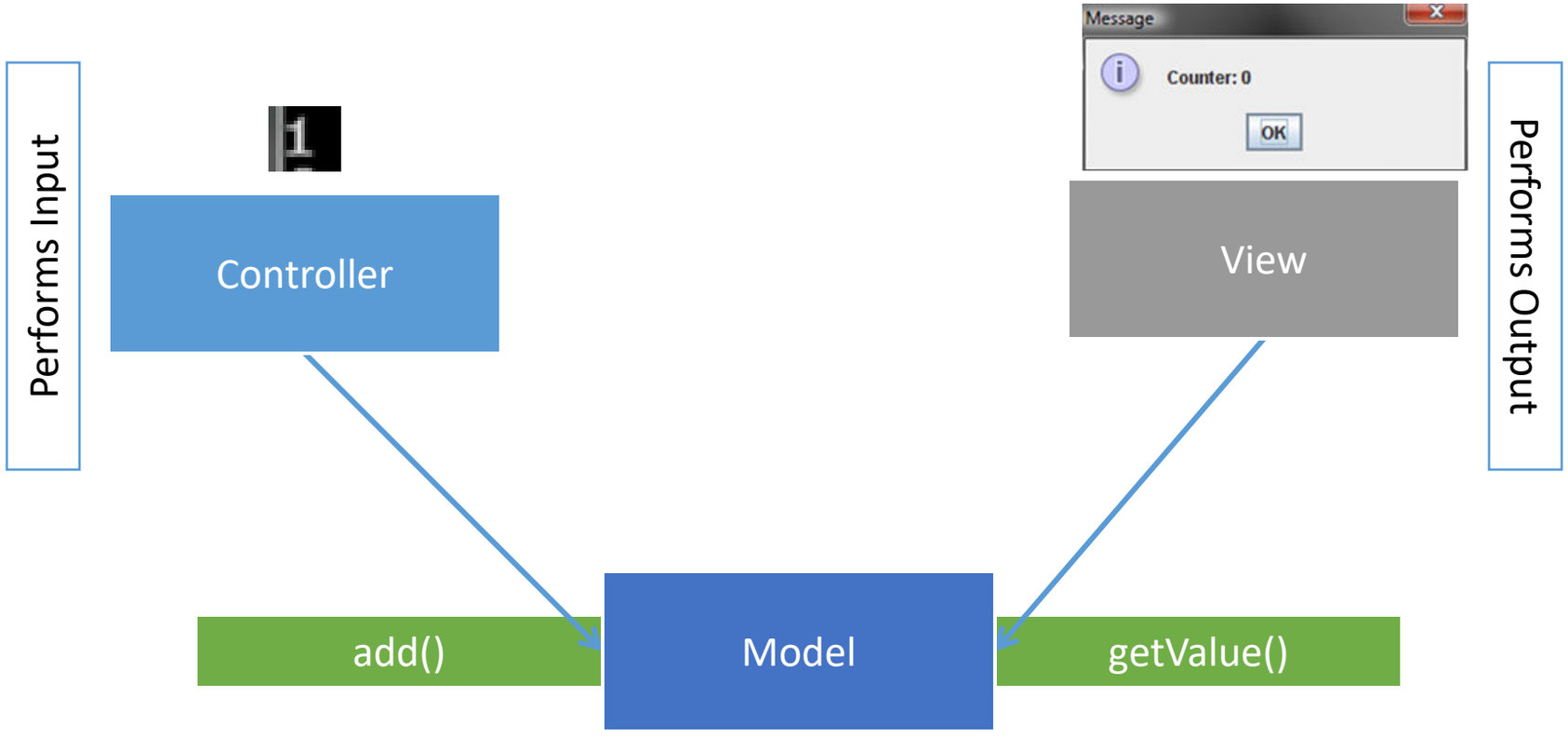
Topic#117
MVC – Example Part VII
MVC Architectural Pattern

© Virtual University of Pakistan

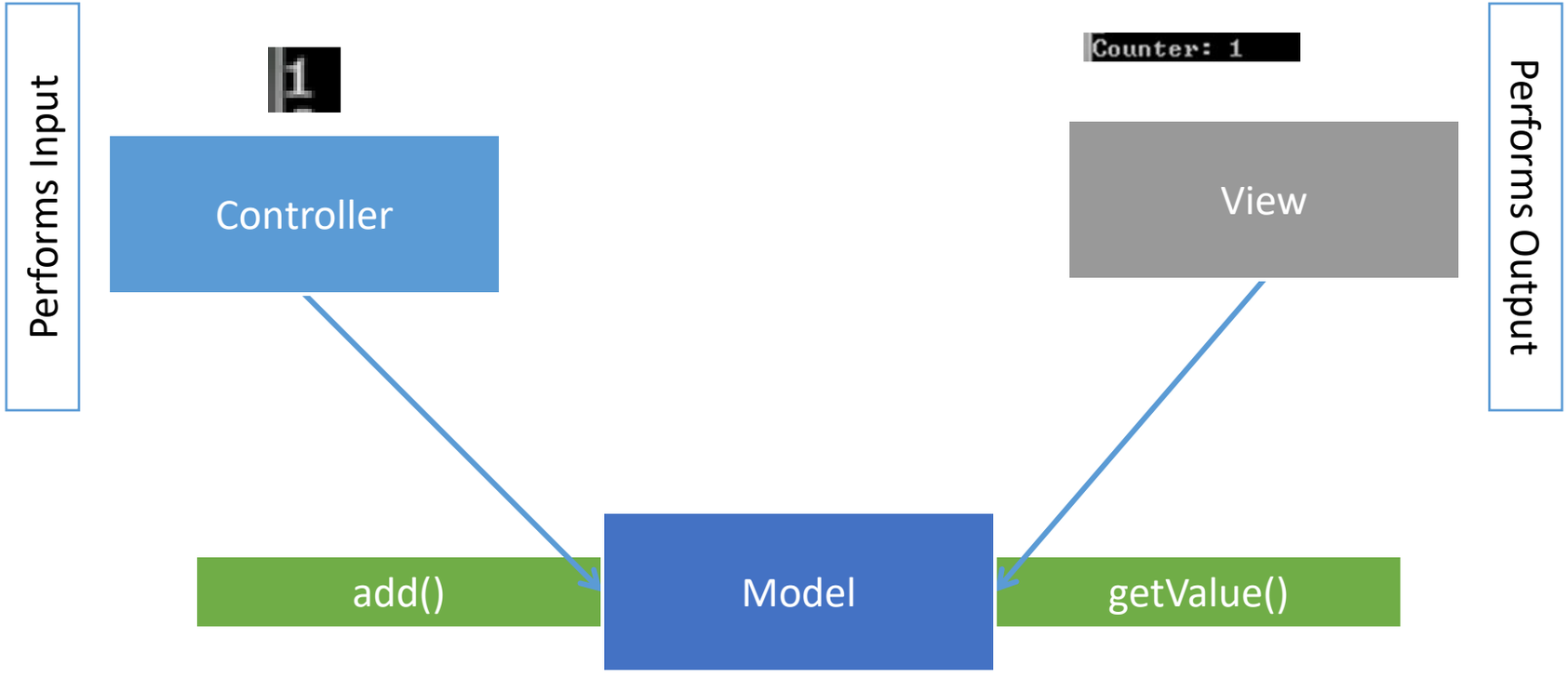
MVC Pattern


Performs Input
Controller
View
Performs Output
Write Methods
Model
Read Methods
Computation Code
View can be on computer with big screen and controller on smart phone

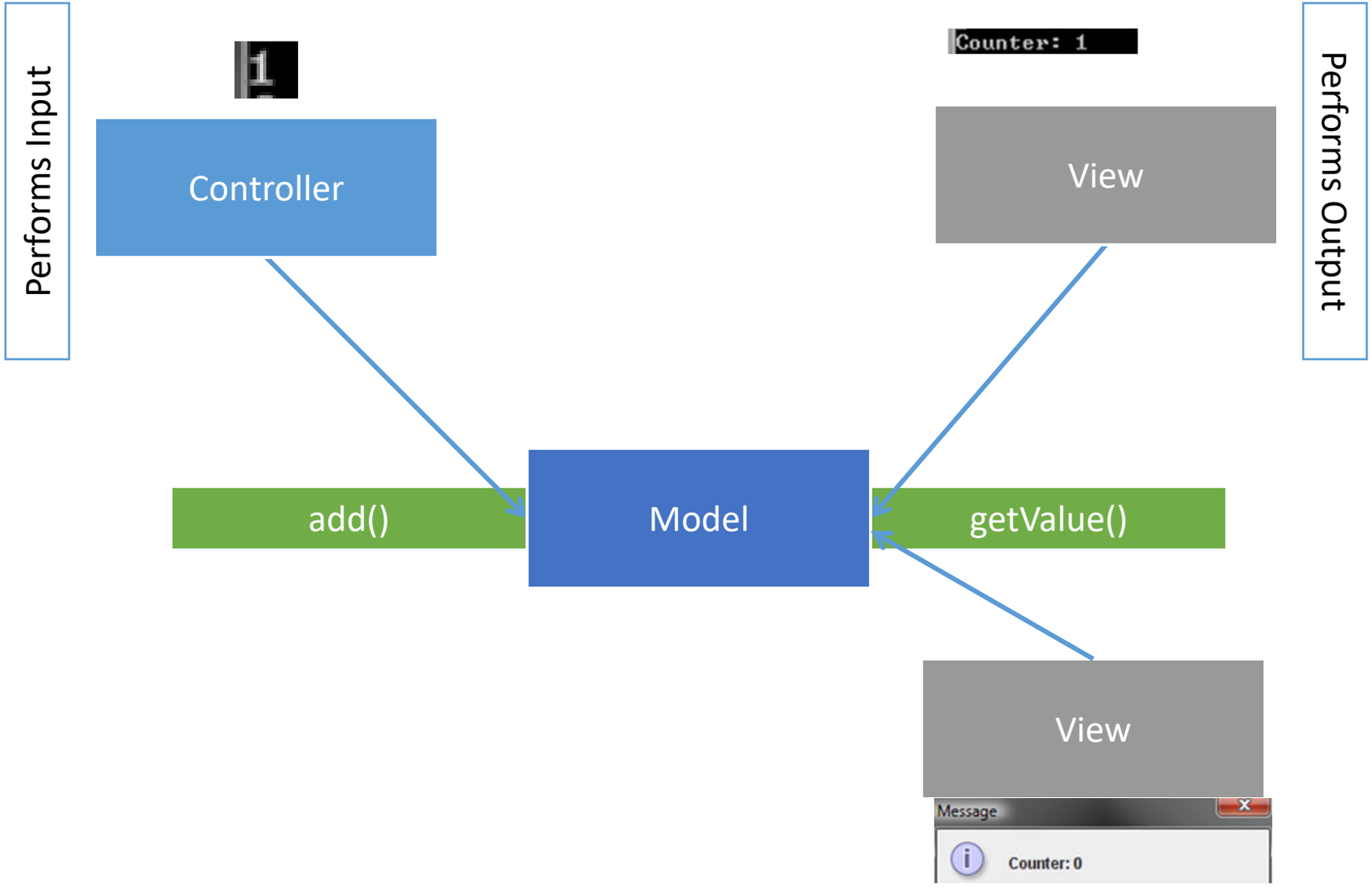
MVC Pattern in Counter



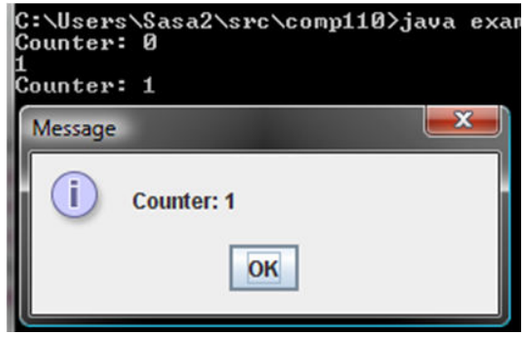
Changing to Console View



Multiple Views



Multiple Views and Controllers



Controller 1

Controller 2

Controller 3

Controller 4

Model

Menus, buttons, shortcuts can be handled by different controllers (in same or different program)

View 1

View 2

View 3

View 4

Software Design and Architecture

Topic#117

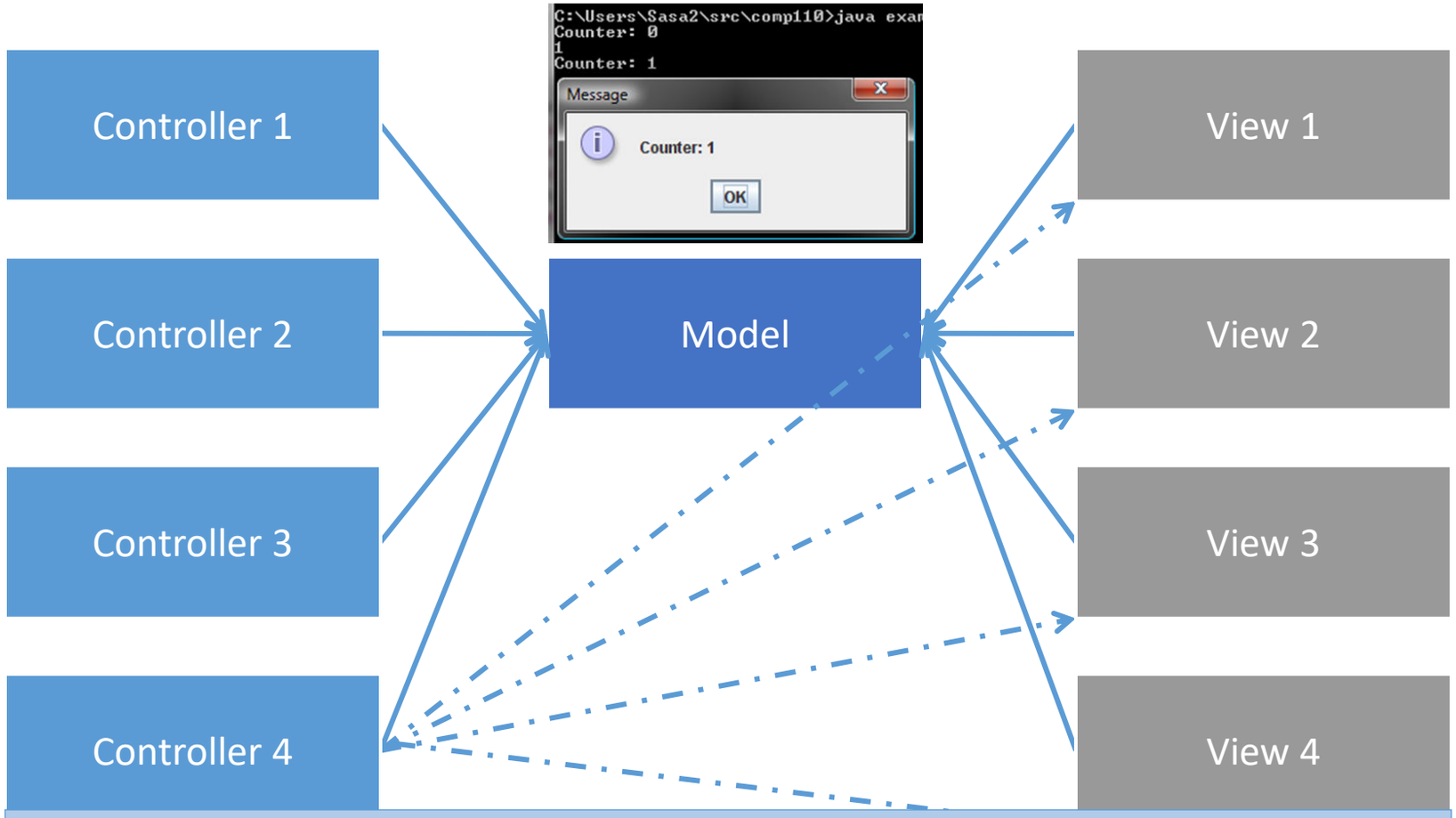
END!

Software Design and Architecture

Topic#118
MVC – Example Part VIII
MVC and Observer

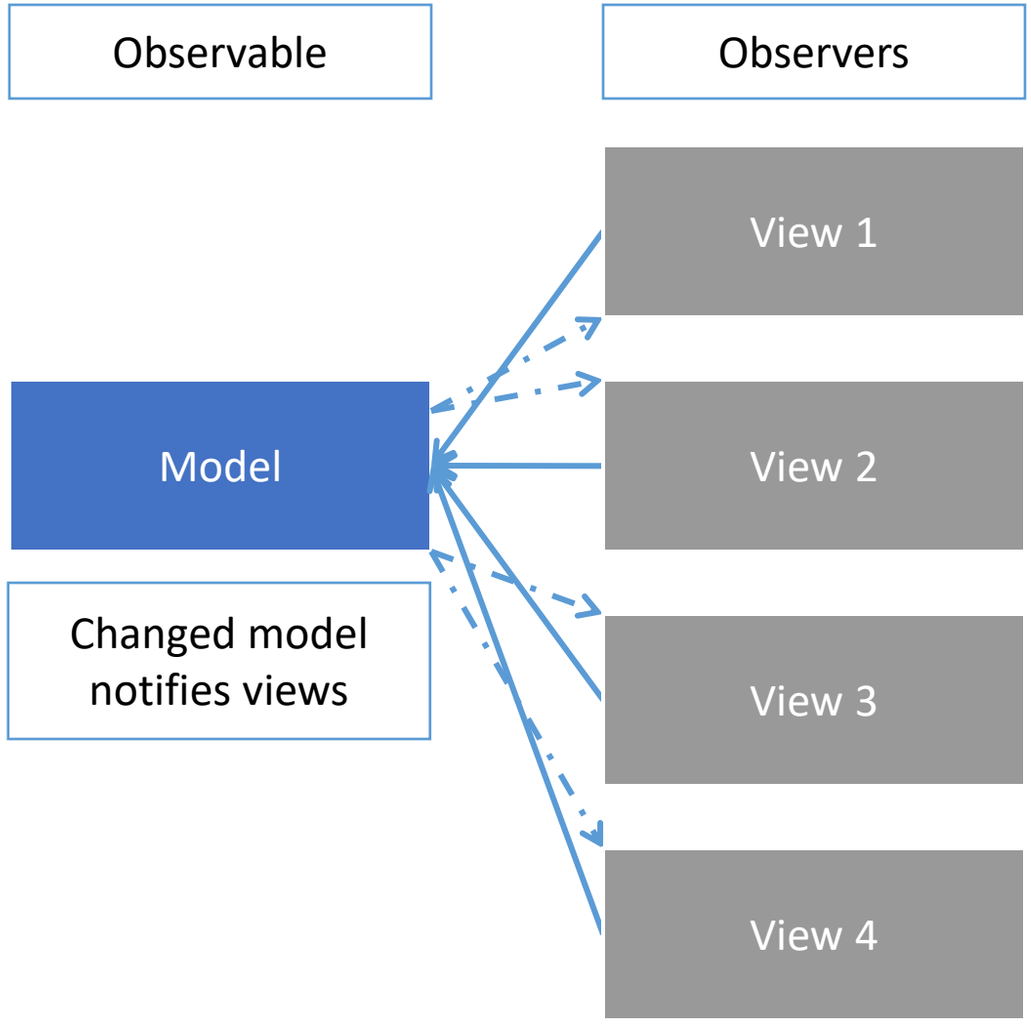
© Virtual University of Pakistan

Syncing Controllers & View



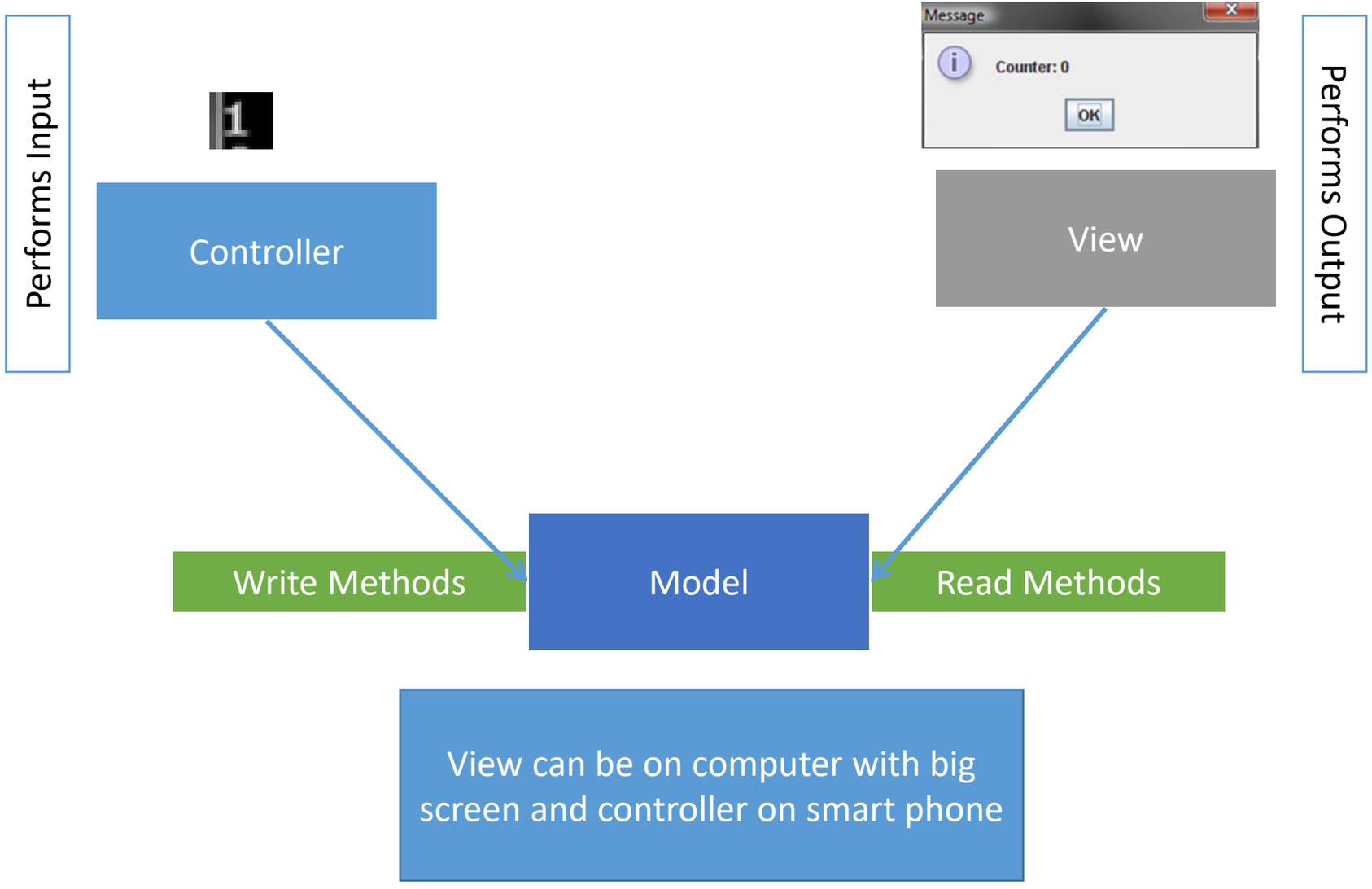
In Http-based “MVC” a single view and controller exists in the browser and the model in the server. A Model cannot initiate actions in the browser so the controller directly communicates with the view

Observer Pattern

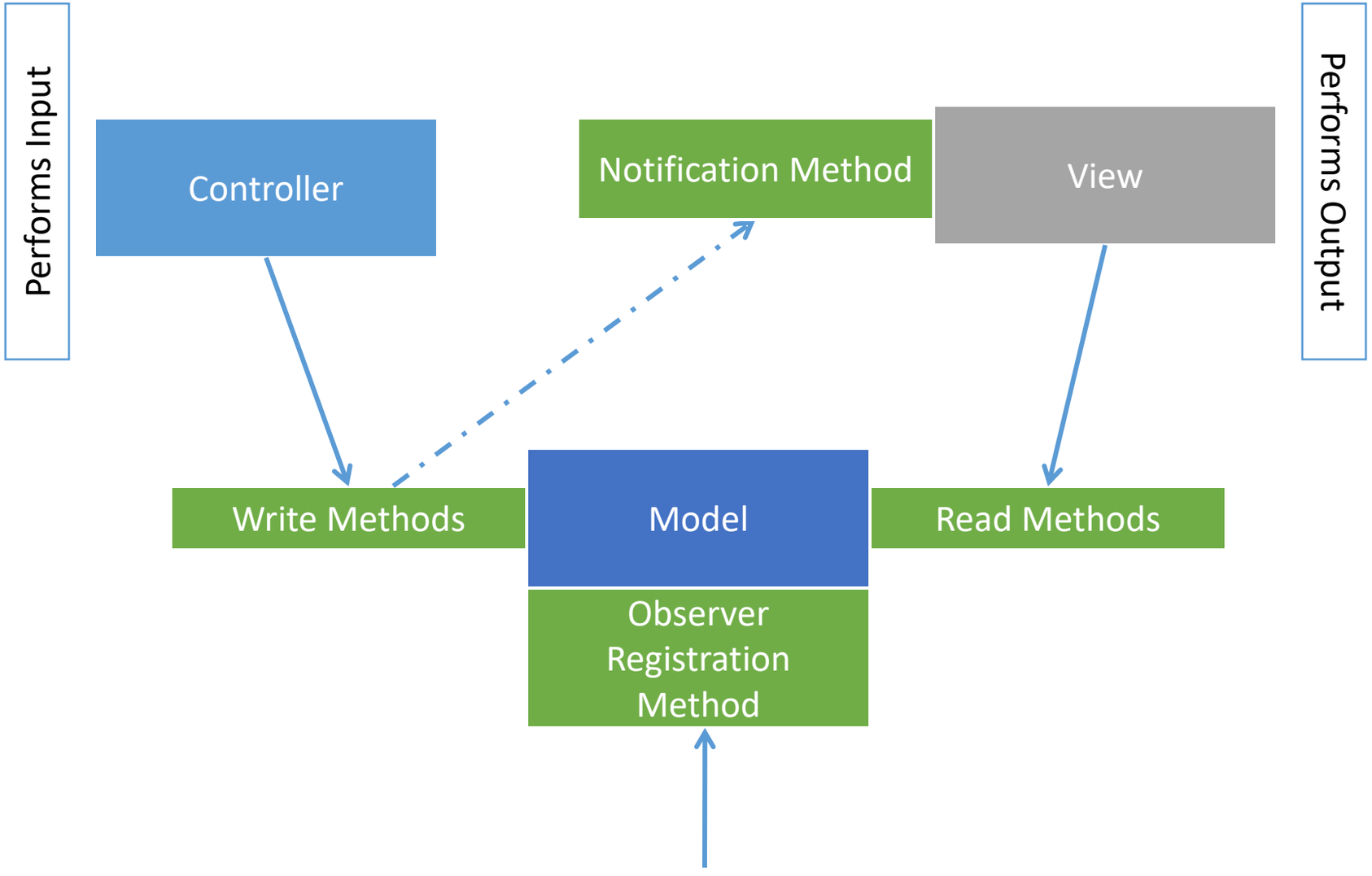


© Virtual University of Pakistan

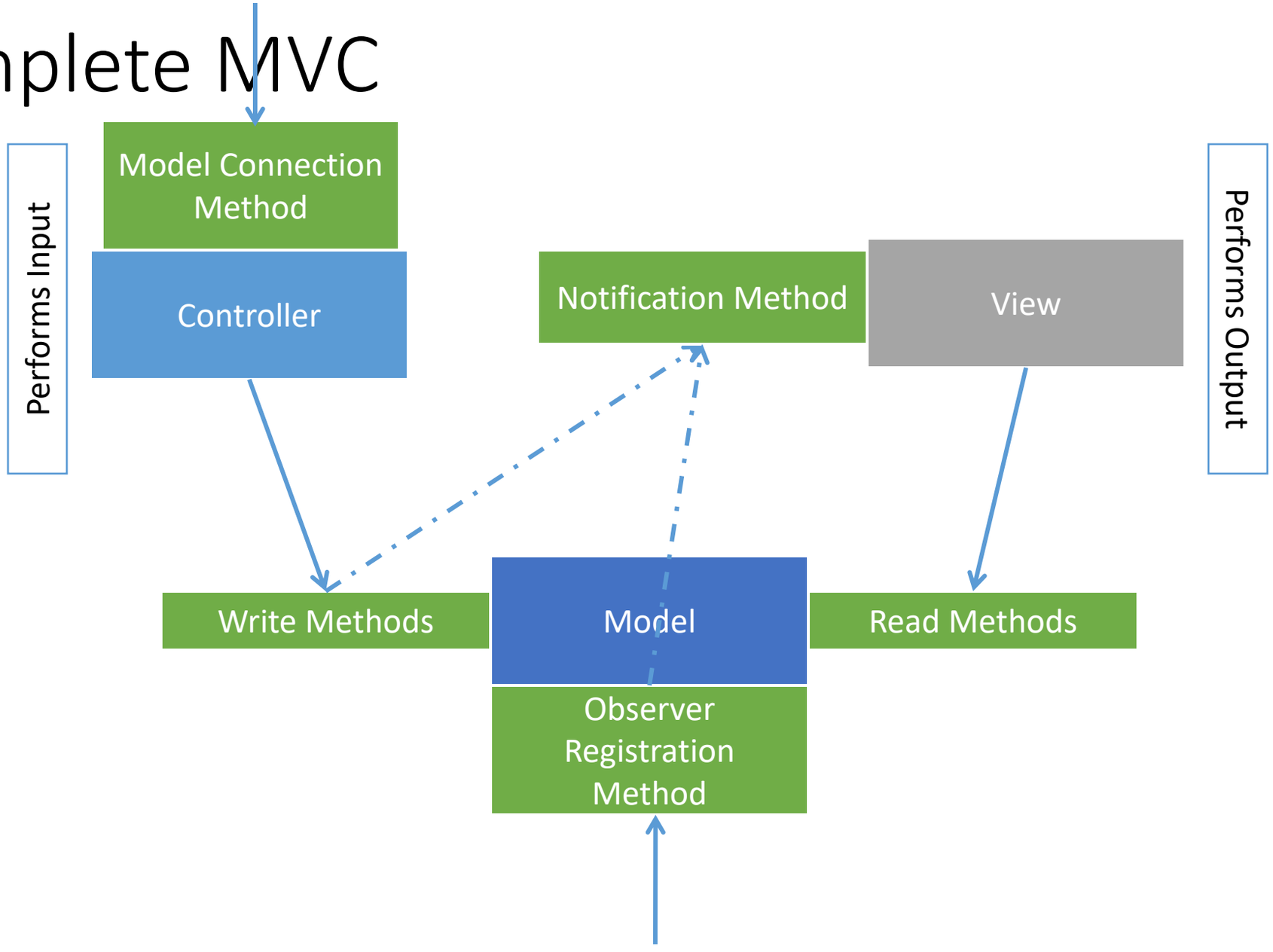
MVC Pattern (Review)



Notifications in MVC Pattern



Complete MVC



© Virtual University of Pakistan

Software Design and Architecture

Topic#118

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 119 – 127

Software Architecture - Overview

© Virtual University of Pakistan

Software Design and Architecture

Topic#119
What is software architecture?

What is software architecture

The word “architecture” means many different things to many different people

Software systems are constructed to satisfy organizations' business goals

The architecture is a bridge between those (often abstract) business goals and the final (concrete) resulting system.

Path from abstract goals to concrete systems

Software Architecture Defined

Some people define the architecture as the system's **"early" or "major"** design decisions

- Many architectural decisions are made early, not all are
  - especially in Agile or spiral-development projects.
- It’s also true that very many decisions are made early that are not architectural

Which one is a “major” decision - sometimes only time will tell

© Virtual University of Pakistan

Software Architecture Defined

- The software architecture of a system is the **set of structures** needed to reason about the system
- Comprises of
  - Software elements
  - Relations among them
  - Properties of both

Software Architecture as a set of Structures

- Structure - a set of elements held together by a relation
- Software systems are composed of many structures
- No single structure holds claim to being *the* architecture

© Virtual University of Pakistan

Software Design and Architecture

Topic#119

END!

Software Design and Architecture

Topic#120
Categories of structures in Architectural Design

© Virtual University of Pakistan

Categories of Structures in Architectural Design

- Three categories

Categories of Structures in Architectural Design

- Static Structures
  - Implementation units or modules
- Dynamic Structures
  - *component-and-connector* (C&C) structures – components are runtime entities
- Deployment Structures
  - aka allocation structures

© Virtual University of Pakistan

- Play an important role in the design, documentation, and analysis of architectures

Software Design and Architecture

Topic#120

END!

Software Design and Architecture

Topic#121
Static Software Structures - Modules

© Virtual University of Pakistan

Implementation units or modules

- Static structures
  - focus on the way the system's functionality is divided up and assigned to implementation teams.
- Assigned specific computational responsibilities
- Basis of work assignments for programming teams
  - (Team A works on the database, Team B works on the business rules, Team C works on the user interface, etc.)

© Virtual University of Pakistan
thegradup.com

Implementation units or modules

- In large projects, these elements (modules) are subdivided for assignment to subteams.
- For example, the database for a large enterprise resource planning (ERP) implementation might be so complex that its implementation is split into many parts.
- aka Module Decomposition Structure
  - captures that decomposition

Implementation units or modules

- Class Diagrams
  - Kind of module structure emerges as an output of object-oriented analysis and design
- Layered Structure
  - Modules aggregated (organized) into layers

© Virtual University of Pakistan

Software Design and Architecture

Topic#121

END!

Software Design and Architecture

Topic#122
Component-and-Connector and Allocation Structures

© Virtual University of Pakistan

Component-and-Connector (C&C) Structures

- Runtime structures
  - Component
- Focus on the way the elements interact with each other at runtime to carry out the system's functions.

© Virtual University of Pakistan

Component-and-Connector (C&C) Structures

- For example the system to be built as a set of services would include:
  - the service – made up of the programs in various implementation units
  - the infrastructure they interact with
  - the synchronization and interaction relations among them

Allocation Structures

- Mapping from software structures to the system's organizational, developmental, installation, and execution environments

Components are deployed onto hardware in order to execute

Software Design and Architecture

Topic#122

END!

Software Design and Architecture

Topic#123
Quality Attributes

Architectural Structures

- Software comprises of a large number of structures
  - not all of them are architectural.
  - For example - code

- A structure is architectural if it supports reasoning about the system and the system's properties
  - an attribute of the system that is important to some stakeholder

The set of architectural structures is not fixed or limited

What is architectural is what is useful in your context for your system

- These include
  - functionality achieved by the system
  - the availability of the system in the face of faults
  - the difficulty of making specific changes to the system
  - the responsiveness of the system to user requests
  - and many others.

Quality Attributes

Software Design and Architecture

Topic#123

END!

Software Design and Architecture

Topic#124
What should not be included in it?

Software Architecture

- an architecture is an *abstraction* of a system that selects certain details and suppresses others.
- The architecture specifically omits certain information about elements that is not useful for reasoning about the system
  - information that has no ramifications outside of a single element

© Virtual University of Pakistan

Software Architecture

- Architecture is concerned with the public interface
  - private details of elements—details having to do solely with internal implementation—are not architectural.

- The architectural abstraction lets us look at the system in terms of:
  - its elements
  - how they are arranged
  - how they interact
  - how they are composed
  - what are their properties that support our system reasoning
  - …

© Virtual University of Pakistan

- Abstraction is essential to taming the complexity of a system
  - we simply cannot, and do not want to, deal with all of the complexity all of the time

© Virtual University of Pakistan

Software Design and Architecture

Topic#124

END!

Software Design and Architecture

Topic#125

Difference between Architecture and Representation of the Architecture

Every Software System Has a Software Architecture

- Every system can be shown to comprise elements and relations among them to support some type of reasoning.
  - In the most trivial case, a system is itself a single element—an uninteresting and probably non-useful architecture, but an architecture nevertheless.

Even though every system has an architecture, it does not necessarily follow that the architecture is known to anyone

Because an architecture can exist independently of its description or specification, this raises the importance of *architecture documentation*

Software Design and Architecture

Topic#125

END!

Software Design and Architecture

Topic#126

Difference between Software, System, and Enterprise Architectures

© Virtual University of Pakistan

- System and enterprise architectures share a great deal with software architectures
- All can be designed, evaluated, and documented
  - all answer to requirements
  - all are intended to satisfy stakeholders
  - all consist of structures, which in turn consist of elements and relationships
- Each has its own specialized vocabulary and techniques

© Virtual University of Pakistan
thegradup.com

System and Enterprise Architectures

- Both of these disciplines have broader concerns than software
- Affect software architecture through the establishment of constraints within which a software system must live

System Architecture

- Concerned with a total system, including hardware, software, and humans
  - mapping of functionality onto hardware and software components
    - a mapping of the software architecture onto the hardware architecture, and a concern for the human interaction with these components

System Architecture

- A description of the software architecture, as it is mapped to hardware and networking components, allows reasoning about qualities such as performance and reliability
- A description of the system architecture will allow reasoning about additional qualities such as power consumption, weight, and physical footprint

Enterprise Architecture

- Concerned with how an enterprise's software systems support the business processes and goals of the enterprise

Enterprise Architecture

A description of the structure and behavior of an organization's processes, information flow, personnel, and organizational subunits, aligned with the organization's core goals and strategic direction

May or may not include information systems

© Virtual University of Pakistan

Software Design and Architecture

Topic#126

END!

Software Design and Architecture

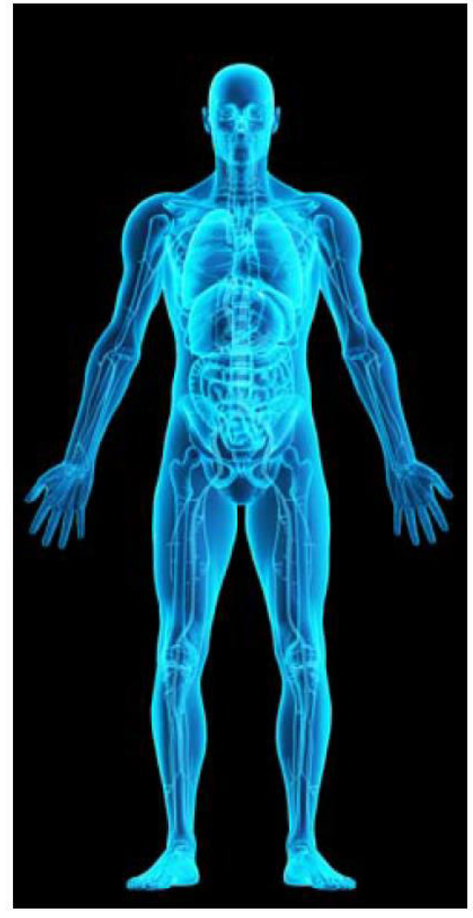
Topic#127
Architectural Views

Different views of the human body: the skeletal, the vascular, and the X-ray

Although these views are pictured differently and have very different properties, all are inherently related, interconnected







Different views of the human body: the skeletal, the vascular, and the X-ray

Together they describe the architecture of the human body







Software Architectural Views

- Modern systems are frequently too complex to grasp all at once
- Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures

Software Architectural Views

- To communicate meaningfully about an architecture, we must make clear which structure or structures we are discussing at the moment
  - which *view* we are taking of the architecture

Structures and Views

- A view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders
- consists of a representation of a set of elements and the relations among them

Structures and Views

- A view is a representation of a structure.
  - For example, a module *structure* is the set of the system's modules and their organization
  - A module *view* is the representation of that structure, documented according to a template in a chosen notation, and used by some system stakeholders

Software Design and Architecture

Topic#127

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 128 – 138

Software Architecture – Overview - Part II

© Virtual University of Pakistan

Software Design and Architecture

Topic#128
4+1 View Model of Software Architecture

© Virtual University of Pakistan

“4+1 view model”

- Philippe Kruchten
  - leader of RUP development team in Rational corp. (now owned by IBM)
  - Valuable experiences in industry (Telecom, Air traffic control system) which he used them for confirmation of his model

© Virtual University of Pakistan

thegradup.com

Problem

- Various stakeholders of software system:
  - end-user, developers, system engineers, project managers
- Arch. documents over-emphasize an aspect of development (i.e. team organization)
  - do not address the concerns of all stakeholders
- Software engineers struggled to represent more on one blueprint, and so arch.
  - documents contain complex diagrams

© Virtual University of Pakistan

thegradup.com

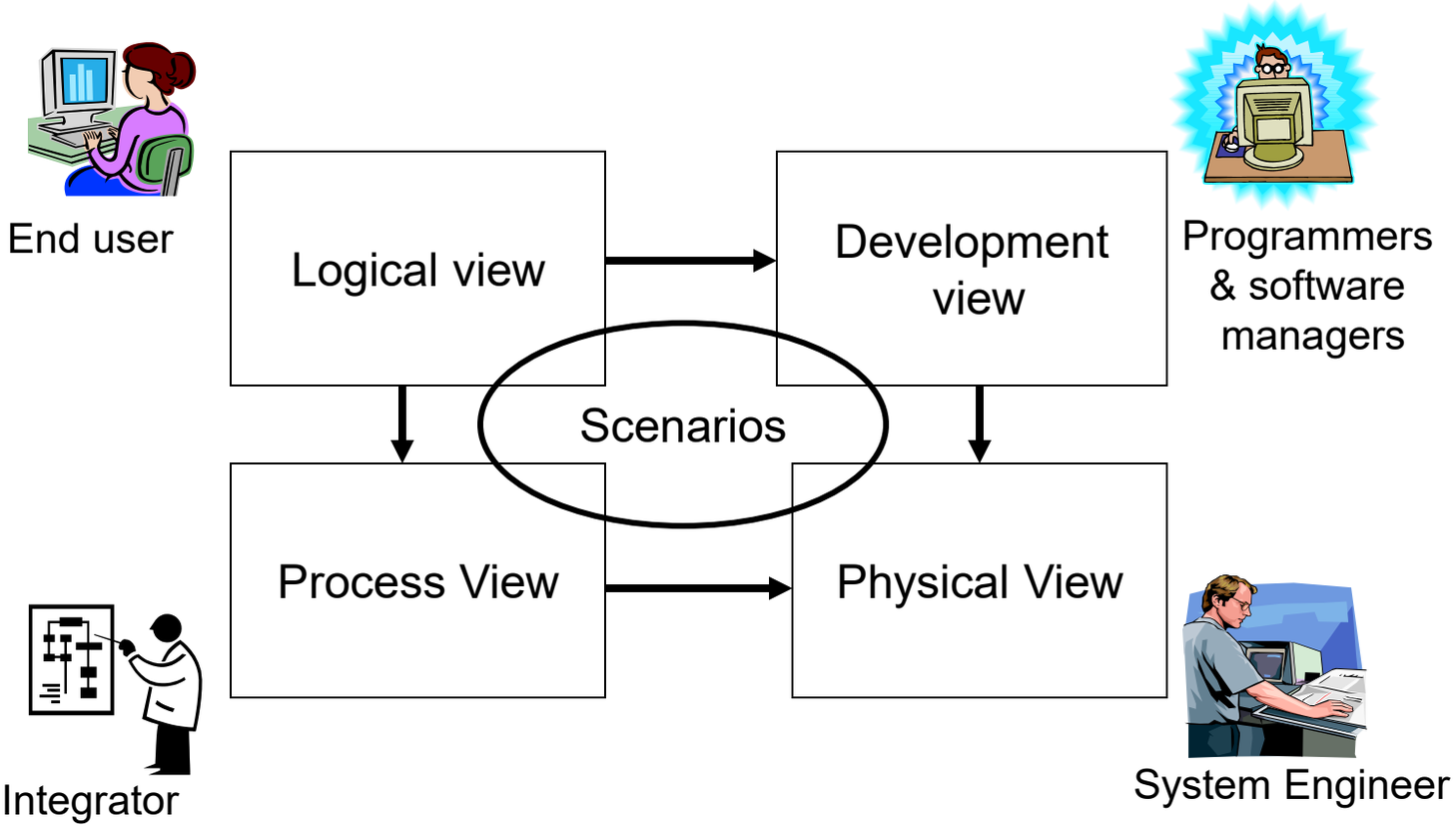
Solution

- Using several concurrent *views* or *perspectives*, with different notations each one addressing one specific set for concerns
- “4+1” view model presented to address large and challenging architectures

© Virtual University of Pakistan

thegradup.com

4+1 View Model of Architecture



© Virtual University of Pakistan

thegradup.com

Logical View

(Object-oriented Decomposition)

Viewer: End-user

considers:

- Functional requirements
  - What the system should provide in terms of services to its users.

© Virtual University of Pakistan

thegradup.com

Process View

(The process decomposition)

viewer: Integrators

considers:

- Non - functional requirements
  - (concurrency, performance, scalability)

© Virtual University of Pakistan

thegradup.com

Development View

(Subsystem decomposition)

Basis of a line of product

Viewer: Programmers and Software Managers

considers:

- software module organization
- (Hierarchy of layers, software management, reuse, constraints of tools)

© Virtual University of Pakistan

thegradup.com

Physical View

(Mapping the software to the Hardware)

Viewer: System Engineers

Considers:

- Non-functional req. regarding underlying hardware (Topology, Communication)

© Virtual University of Pakistan

thegradup.com

Scenarios

(Putting it all together)

Viewer: All users of other views and Evaluators.

Considers: System consistency, validity

© Virtual University of Pakistan

thegradup.com

Correspondence between views

- Views are interconnected.
- Start with Logical view (Req. Doc) and Move to Development or Process view and then finally go to Physical view.

© Virtual University of Pakistan

thegradup.com

Software Design and Architecture

Topic#128

END!

Software Design and Architecture

Topic#129
Module Structures

© Virtual University of Pakistan

The Three Categories of Structures in Architectural Design

- Static Structures
  - Module Structures
- Dynamic Structures
  - *component-and-connector* (C&C) structures – components are runtime entities
- Deployment Structures
  - aka allocation structures

Module Structures

- Embody decisions as to how the system is to be structured as a set of code or data units that have to be constructed or procured.
- In any module structure, the elements are modules of some kind (perhaps classes, or layers, or merely divisions of functionality, all of which are units of implementation).

© Virtual University of Pakistan

Module Structures

- Modules represent a static way of considering the system.
- Modules are assigned areas of functional responsibility
  - there is less emphasis in these structures on how the resulting software manifests itself at runtime.

© Virtual University of Pakistan

Module Structures

- Module structures allow us to answer questions such as these:
  - What is the primary functional responsibility assigned to each module?
  - What other software elements is a module allowed to use?
  - What other software does it actually use and depend on?
  - What modules are related to other modules by generalization or specialization
    - (i.e., inheritance) relationships?

Module Structures

- Module structures convey this information directly, but they can also be used by extension to ask questions about the impact on the system when the responsibilities assigned to each module change.
  - looking at its module views is an excellent way to reason about a system’s modifiability

Software Design and Architecture

Topic#129

END!

Software Design and Architecture

Topic#130
Component and Connector Structures

© Virtual University of Pakistan

Component-and-connector Structures

- Embody decisions as to how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors)
- The elements are runtime components
  - which are the principal units of computation
    - services, peers, clients, servers, filters, etc
  - connectors - communication vehicles among components
    - call-return, process synchronization operators, pipes, etc

© Virtual University of Pakistan

Component-and-connector Structures

- Component-and-connector views help us answer questions such as these:
  - What are the major executing components and how do they interact at runtime?
  - What are the major shared data stores?
  - Which parts of the system are replicated?
  - How does data progress through the system?
  - What parts of the system can run in parallel?
  - Can the system's structure change as it executes and, if so, how?

Component-and-connector Structures

- component-and-connector views are crucially important for asking questions about the system's runtime properties
  - performance, security, availability, and more

Software Design and Architecture

Topic#130

END!

Software Design and Architecture

Topic#131
Allocation Structures

Allocation Structures

- *E*mbody decisions as to how the system will relate to non software structures in its environment
  - CPUs, file systems, networks, development teams, etc.
- These structures show the relationship between the software elements and elements in one or more external environments in which the software is created and executed.

Allocation Structures

- Allocation views help us answer questions such as these:
  - What processor does each software element execute on?
  - In what directories or files is each element stored during development, testing, and system building?
  - What is the assignment of each software element to development teams?

© Virtual University of Pakistan

Useful for

- Performance, availability, security analysis
- Configuration control, integration, test activities
- Project management, best use of expertise and available resources, management of commonality

© Virtual University of Pakistan

Software Design and Architecture

Topic#131

END!

Software Design and Architecture

Topic#132
Structures and Quality Attributes

Putting it all together

- Static Structures
  - Module Structures
- Dynamic Structures
  - *component-and-connector* (C&C) structures – components are runtime entities
- Deployment Structures
  - aka allocation structures

© Virtual University of Pakistan

- Each structure provides a perspective for reasoning about some of the relevant quality attributes.

- For example:
  - The module “uses” structure, which embodies what modules use what other modules, is strongly tied to the ease with which a system can be extended or contracted.

© Virtual University of Pakistan

- For example:
  - The concurrency structure, which embodies parallelism within the system, is strongly tied to the ease with which a system can be made free of deadlock and performance bottlenecks.

- For example:
  - The deployment structure is strongly tied to the achievement of performance, availability, and security goals.

- Each structure provides the architect with a different insight into the design (that is, each structure can be analyzed for its ability to deliver a quality attribute).
- Each structure presents the architect with an engineering leverage point
  - By designing the structures appropriately, the desired quality attributes emerge.

Software Design and Architecture

Topic#132

END!

Software Design and Architecture

Topic#133
Relating Structures to Each Other

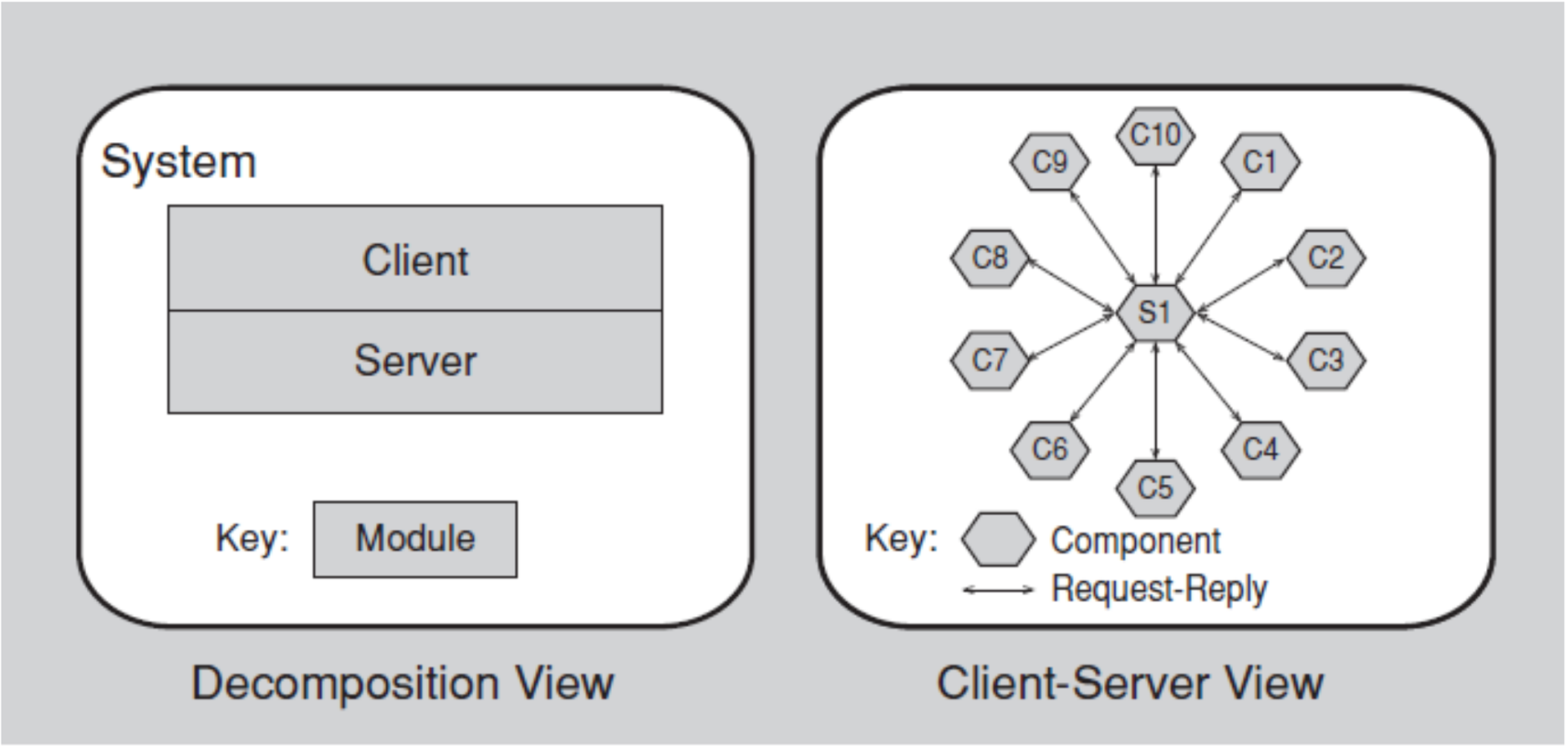
© Virtual University of Pakistan

Relating Structures to Each Other

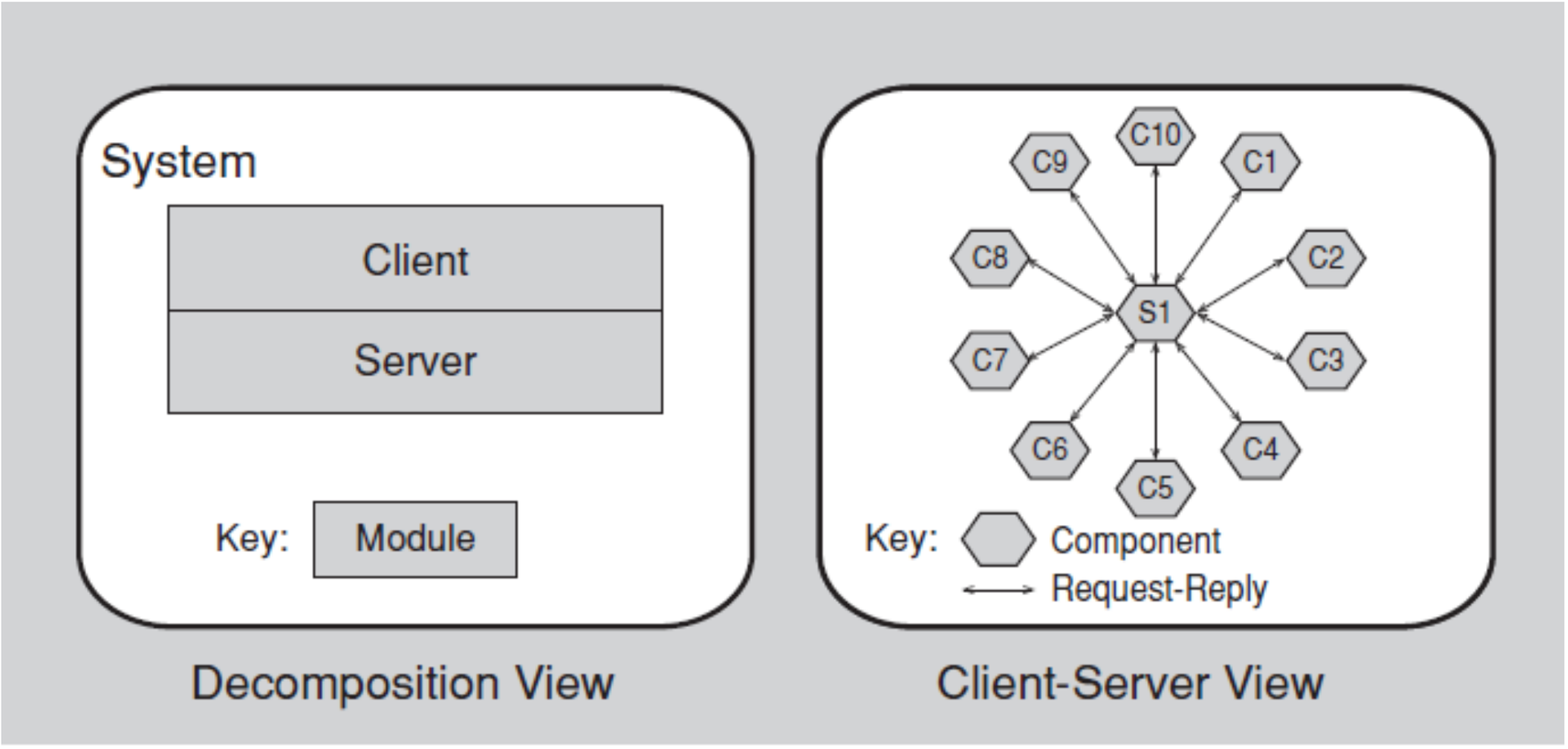
- Each of these structures provides a different perspective and design handle on a system
- each is valid and useful in its own right.

Relating Structures to Each Other

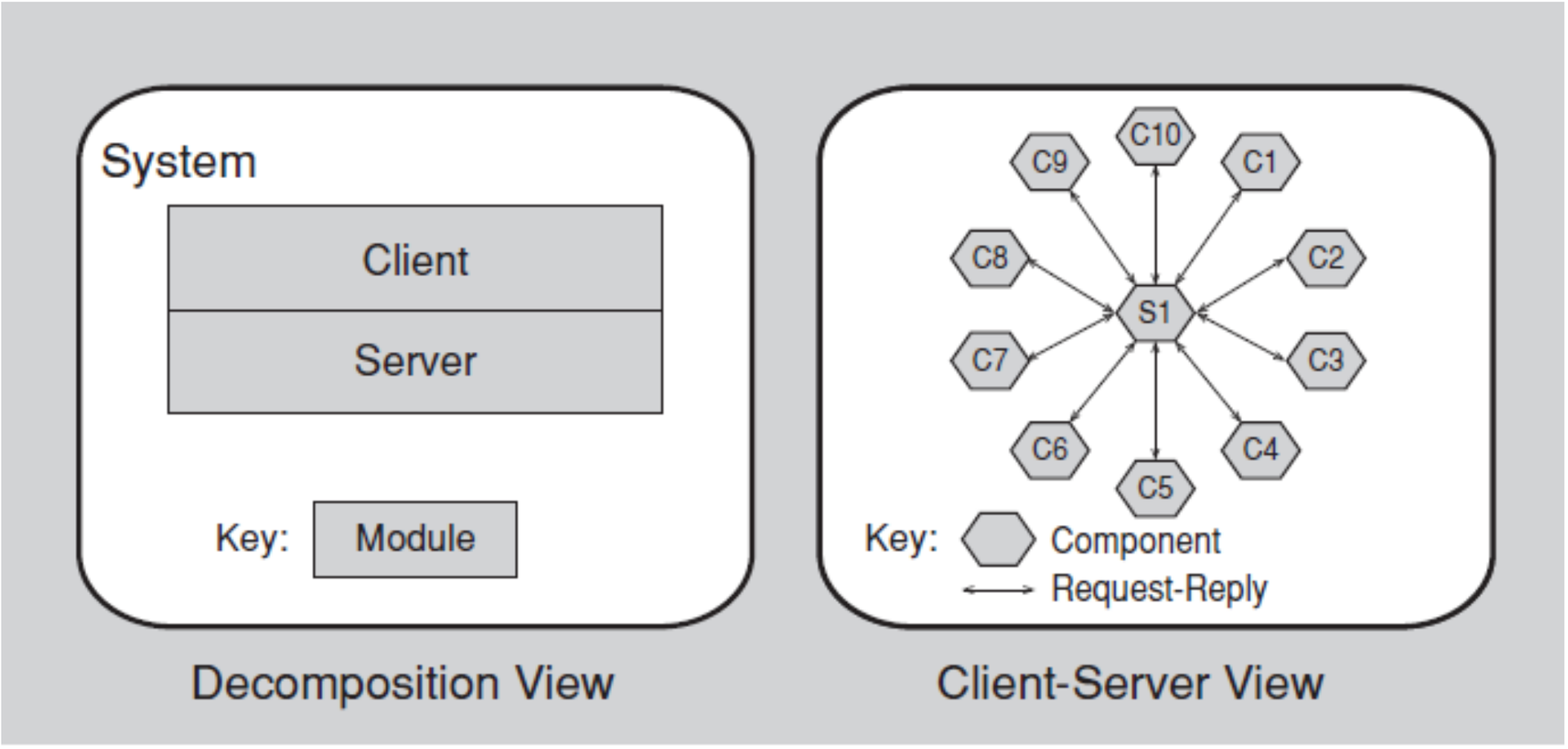
- Although the structures give different system perspectives, they are not independent.
  - Elements of one structure will be related to elements of other structures
  - For example, a module in a decomposition structure may be manifested as one, part of one, or several components in one of the component-and-connector structures,
  - In general, mappings between structures are many to many.



Example of how two structures might relate to each other



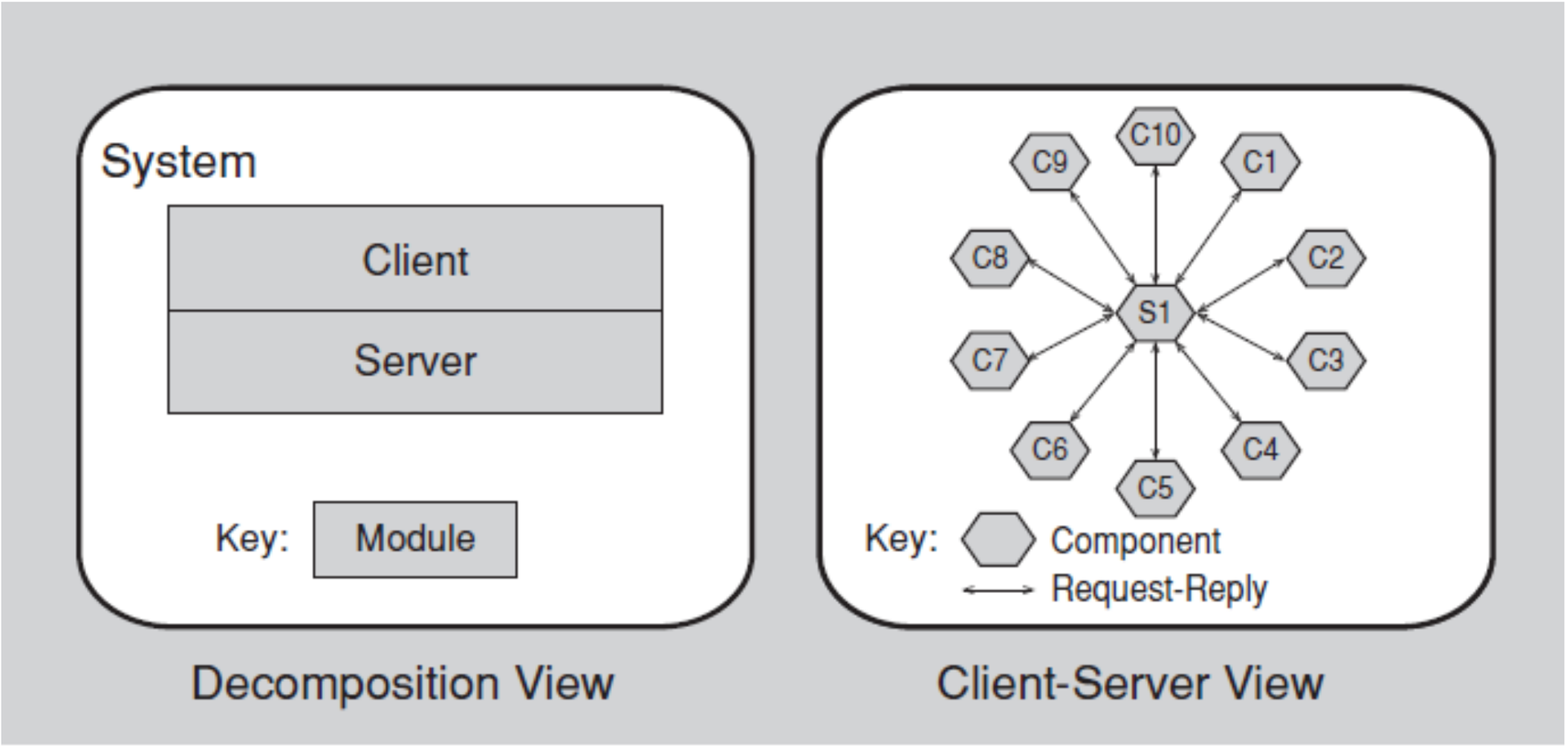
The figure on the left shows a module decomposition view of a tiny client-server system



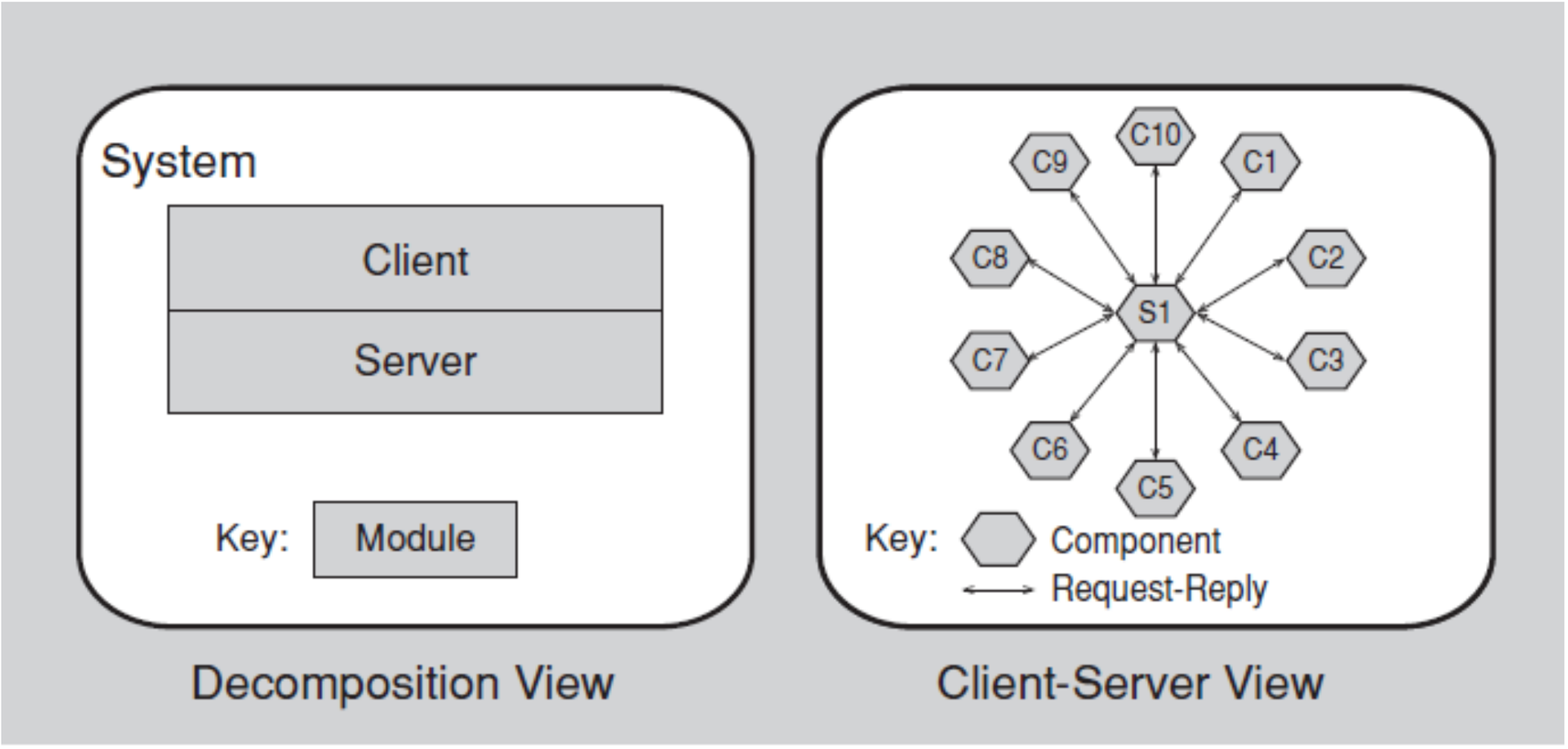
Decomposition View

Two modules must be implemented:

The client software and the server software

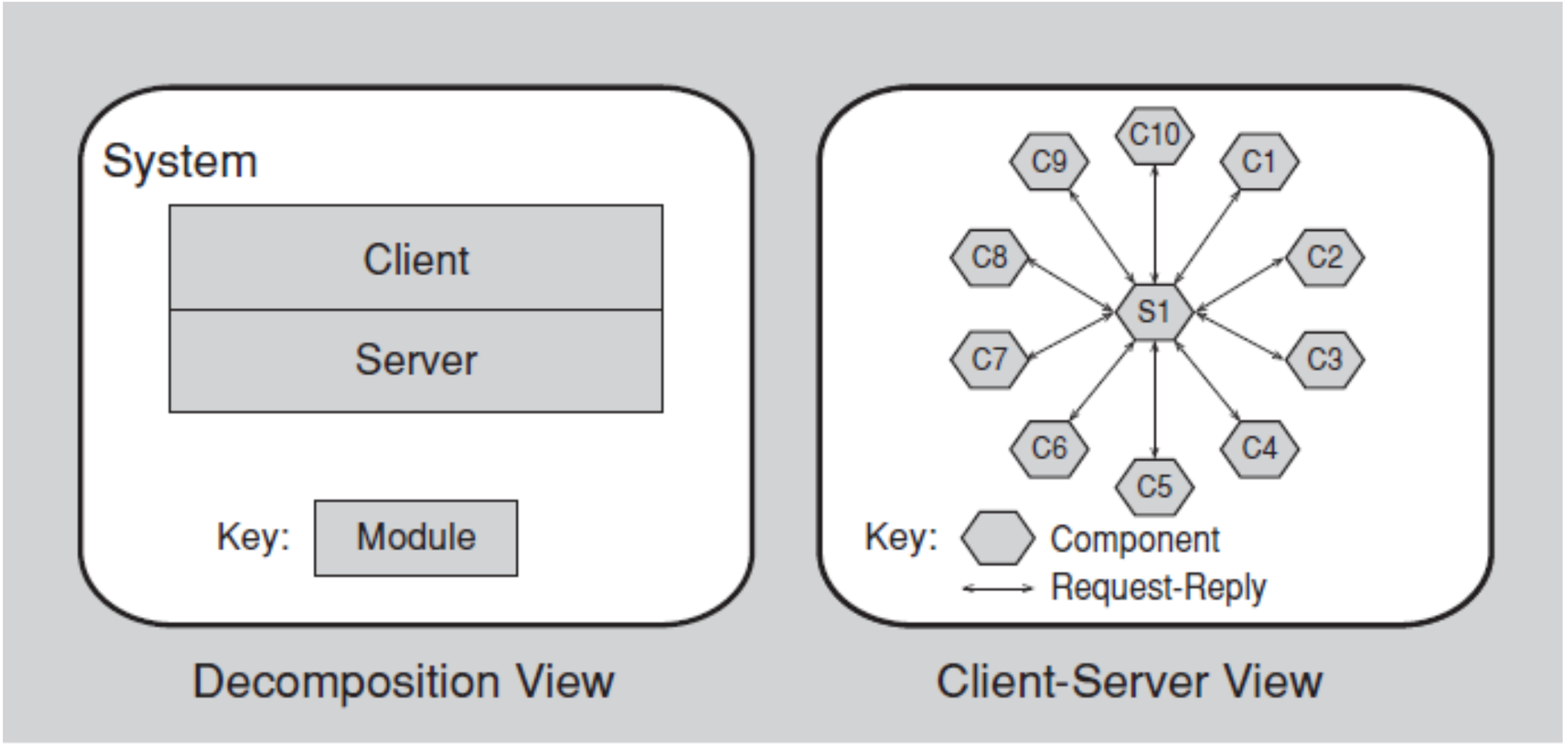


The figure on the right shows a component-and-connector view of the same system

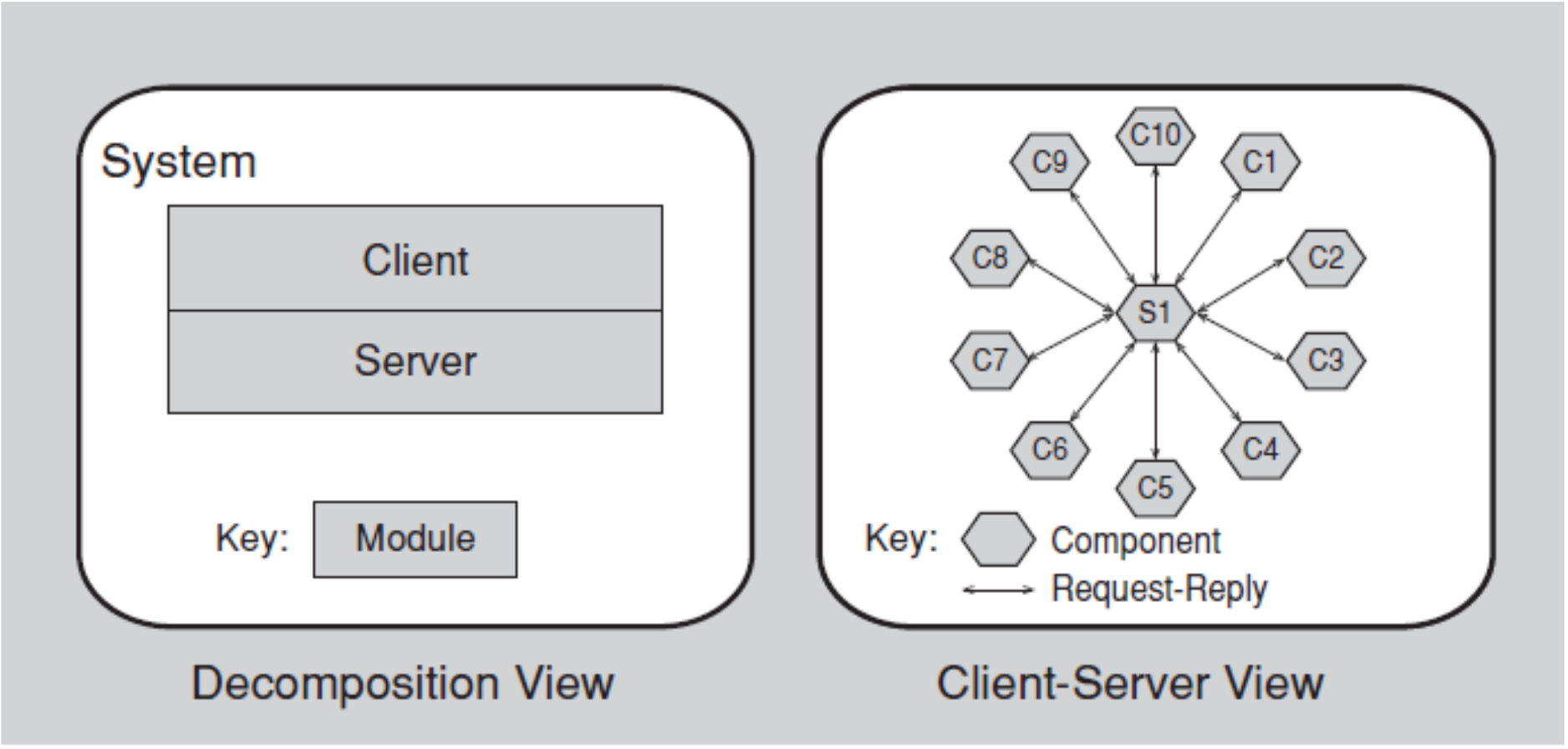


Component-and-connector view of the same system

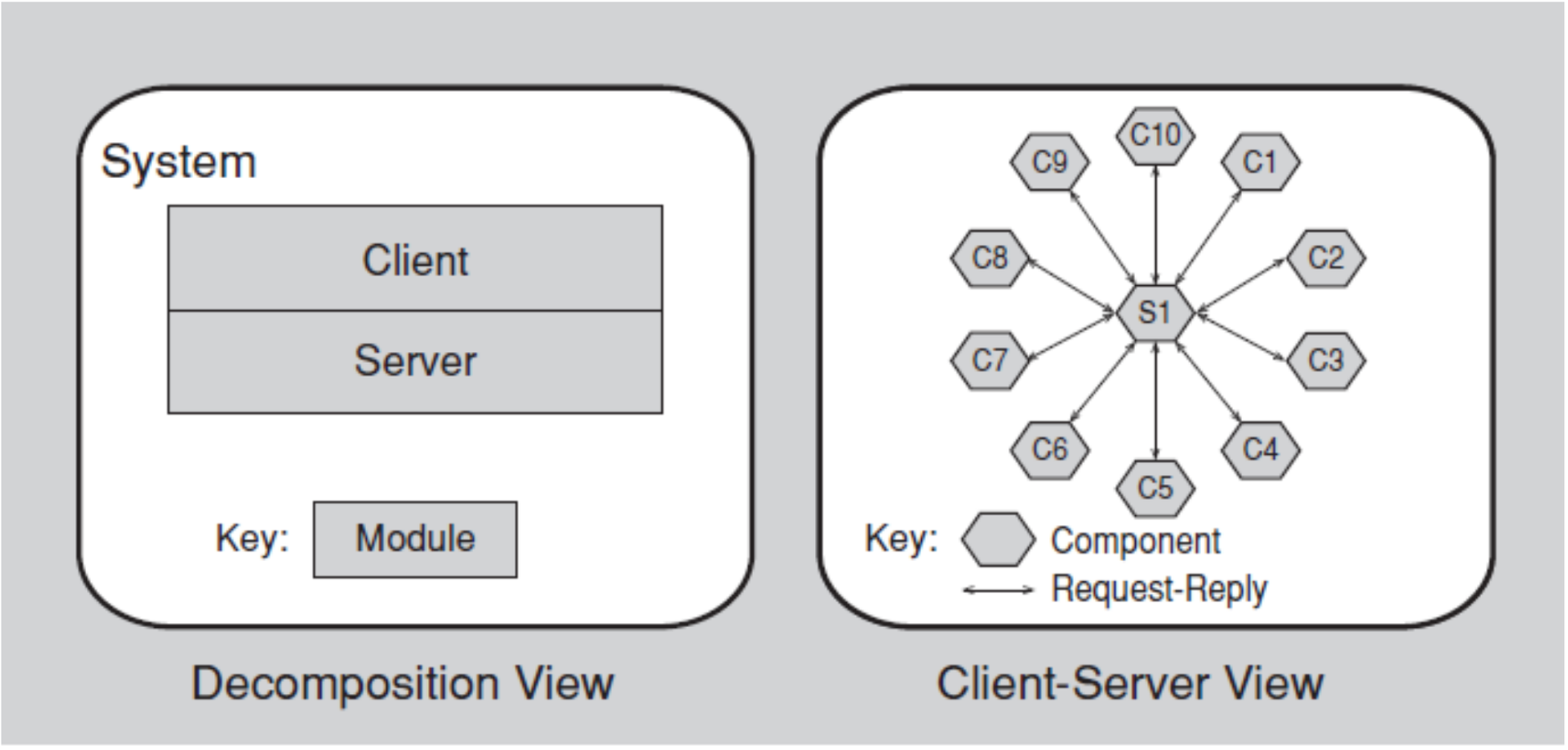
At runtime there are ten clients running and accessing the server



two modules and eleven components (and ten connectors)



Used for very different things



For example, the view on the right could be used for performance analysis, bottleneck prediction, and network traffic management, which would be extremely difficult or impossible to do with the view on the left.

Software Design and Architecture

Topic#133

END!

Software Design and Architecture

Topic#134
Choosing an Appropriate Structure

© Virtual University of Pakistan

Degree of Rigor

- Not all systems warrant consideration of many architectural structures
- The larger the system, the more dramatic the difference between these structures tends to be
- For small systems we can often get by with fewer

Degree of Rigor

- Instead of working with each of several component-and-connector structures, usually a single one will do
- If there is only one process, then the process structure collapses to a single node and need not be explicitly represented in the design
- If there is to be no distribution (that is, if the system is implemented on a single processor), then the deployment structure is trivial and need not be considered further

Degree of Rigor

- In general, design and document a structure only if doing so brings a positive return on the investment, usually in terms of decreased development or maintenance costs

© Virtual University of Pakistan

Which Structures to Choose?

- Many architectural structures to choose from
- Which ones shall an architect choose to work on?
- Which ones shall the architect choose to document?
- Surely not all of them.

Which Structures to Choose?

- How the various available structures provide insight and leverage into the system's most important quality attributes, and then choose the ones that will play the best role in delivering those attributes

Software Design and Architecture

Topic#134

END!

Software Design and Architecture

Topic#135
Architectural Patterns – Overview

© Virtual University of Pakistan

- In some cases, architectural elements are composed in ways that solve particular problems.
- The compositions have been found useful over time, and over many different domains, and so they have been documented and disseminated.
- These compositions of architectural elements, called *architectural patterns*, provide packaged strategies for solving some of the problems facing a system.

© Virtual University of Pakistan
thegradup.com

- An architectural pattern delineates the element types and their forms of interaction used in solving the problem.
- Patterns can be characterized according to the type of architectural elements they use

© Virtual University of Pakistan

Some Examples

- Module Patterns
  - Layered Pattern
- Component-and-connector type pattern
  - Shared Data (or repository) Pattern
  - Client-Server Pattern
- Allocation Patterns
  - Multi-tier pattern
  - Competence center and platform center

Layered Pattern

- *Module type pattern*
- When the *uses* relation among software elements is strictly unidirectional, a system of layers emerges.
- A layer is a coherent set of related functionality.
- In a *strictly* layered structure, a layer can only use the services of the layer immediately below it.
- Many variations of this pattern, lessening the structural restriction, occur in practice.
- Layers are often designed as abstractions (virtual machines) that hide implementation specifics below from the layers above, engendering portability

Shared Data (or repository) Pattern

- Component-and-connector type pattern
- comprises components and connectors that create, store, and access persistent data.
- The repository usually takes the form of a database.
- The connectors are protocols for managing the data, such as SQL.

© Virtual University of Pakistan

Client-Server Pattern

- Component-and-connector type pattern
- The components are the clients and the servers, and the connectors are protocols and messages they share among each other to carry out the system's work.

Multi-tier Pattern

- Allocation pattern
- Describes how to distribute and allocate the components of a system in distinct subsets of hardware and software, connected by some communication medium.
- This pattern specializes the generic deployment (software-to-hardware allocation) structure.

© Virtual University of Pakistan

Competence center and platform

- Allocation pattern
- specialize a software system’s work assignment structure.
- In *competence center*, work is allocated to sites depending on the technical or domain expertise located at a site.
  - For example, user-interface design is done at a site where usability engineering experts are located.
- In *platform*, one site is tasked with developing reusable core assets of a software product line, and other sites develop applications that use the core assets.

© Virtual University of Pakistan

Software Design and Architecture

Topic#135

END!

Software Design and Architecture

Topic#136
What makes a good architecture?

- No such thing as an inherently good or bad architecture
- Is it fit for some purpose?

- A three-tier layered service-oriented architecture may be the right thing for a large enterprise's web-based B2B system but completely wrong for an avionics application.
- An architecture carefully crafted to achieve high modifiability does not make sense for a throwaway prototype (and vice versa!).

- Architectures can in fact be *evaluated*—but only in the context of specific stated goals

- There are rules of thumb that should be followed when designing most architectures
- Failure to apply any of these does not automatically mean that the architecture will be fatally flawed, but it should at least serve as a warning sign that should be investigated.

© Virtual University of Pakistan

Recommendation

- process recommendations
- product (or structural) recommendations

Software Design and Architecture

Topic#136

END!

Software Design and Architecture

Topic#137
Process Recommendations

© Virtual University of Pakistan

Process Recommendations - 1

- The architecture should be the product of a single architect or a small group of architects with an identified technical leader.
  - This approach gives the architecture its conceptual integrity and technical consistency.
  - This recommendation holds for Agile and open source projects as well as “traditional” ones.
  - There should be a strong connection between the architect(s) and the development team, to avoid ivory tower designs that are impractical.

© Virtual University of Pakistan

Process Recommendations - 2

- The architect (or architecture team) should, on an ongoing basis, base the architecture on a prioritized list of well-specified quality attribute requirements.
- These will inform the tradeoffs that always occur.
  - Functionality matters less.

© Virtual University of Pakistan

Process Recommendations - 3

- The architecture should be documented using views.
- The views should address the concerns of the most important stakeholders in support of the project timeline.
- This might mean minimal documentation at first, elaborated later.
- Concerns usually are related to construction, analysis, and maintenance of the system, as well as education of new stakeholders about the system.

© Virtual University of Pakistan

Process Recommendations - 4

- The architecture should be evaluated for its ability to deliver the system's important quality attributes.
- This should occur early in the life cycle, when it returns the most benefit, and repeated as appropriate, to ensure that changes to the architecture (or the environment for which it is intended) have not rendered the design obsolete

Process Recommendations - 5

- The architecture should lend itself to incremental implementation, to avoid having to integrate everything at once (which almost never works) as well as to discover problems early.
- One way to do this is to create a “skeletal” system in which the communication paths are exercised but which at first has minimal functionality.
- This skeletal system can be used to “grow” the system incrementally, refactoring as necessary.

Software Design and Architecture

Topic#137

END!

Software Design and Architecture

Topic#138
Product Recommendations

© Virtual University of Pakistan

Structural Recommendations - 1

- The architecture should feature well-defined modules whose functional responsibilities are assigned on the principles of information hiding and separation of concerns.
  - The information-hiding modules should encapsulate things likely to change, thus insulating the software from the effects of those changes.
  - Each module should have a well-defined interface that encapsulates or “hides” the changeable aspects from other software that uses its facilities.
  - These interfaces should allow their respective development teams to work largely independently of each other.

© Virtual University of Pakistan

Structural Recommendations - 2

- Unless your requirements are unprecedented—possible, but unlikely—your quality attributes should be achieved using well-known architectural patterns and tactics specific to each attribute

Structural Recommendations - 3

- The architecture should never depend on a particular version of a commercial product or tool.
- If it must, it should be structured so that changing to a different version is straightforward and inexpensive

© Virtual University of Pakistan

Structural Recommendations - 4

- Modules that produce data should be separate from modules that consume data.
  - This tends to increase modifiability because changes are frequently confined to either the production or the consumption side of data.
- If new data is added, both sides will have to change, but the separation allows for a staged (incremental) upgrade.

© Virtual University of Pakistan

Structural Recommendations - 5

- Don’t expect a one-to-one correspondence between modules and components.
  - For example, in systems with concurrency, there may be multiple instances of a component running in parallel, where each component is built from the same module. For systems with multiple threads of concurrency, each thread may use services from several components, each of which was built from a different module.

Structural Recommendations - 6

- Every process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.

© Virtual University of Pakistan

Structural Recommendations - 7

- The architecture should feature a small number of ways for components to interact.
- That is, the system should do the same things in the same way throughout.
- This will aid in understandability, reduce development time, increase reliability, and enhance modifiability.

© Virtual University of Pakistan
thegradup.com

Structural Recommendations - 8

- The architecture should contain a specific (and small) set of resource contention areas, the resolution of which is clearly specified and maintained.
- For example, if network utilization is an area of concern, the architect should produce (and enforce) for each development team guidelines that will result in a minimum of network traffic.
- If performance is a concern, the architect should produce (and enforce) time budgets for the major threads.

© Virtual University of Pakistan

Software Design and Architecture

Topic#138

END!

© Virtual University of Pakistan

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 139 – 148

Architecture Meta-Frame – Part I

© Virtual University of Pakistan

Software Design and Architecture

Topic#139
Architectural Drivers

considerations that need to be made for the software system that are **architecturally significant**.

They **drive and guide** the design of the software architecture.

Architectural drivers describe what you are doing and why you are doing it.

Software architecture design satisfies architectural drivers.

- Architectural drivers are inputs into the design process, and include:
  - Design objectives
  - Primary functional requirements
  - Quality attribute scenarios
  - Constraints
  - Architectural concerns

Software Design and Architecture

Topic#139

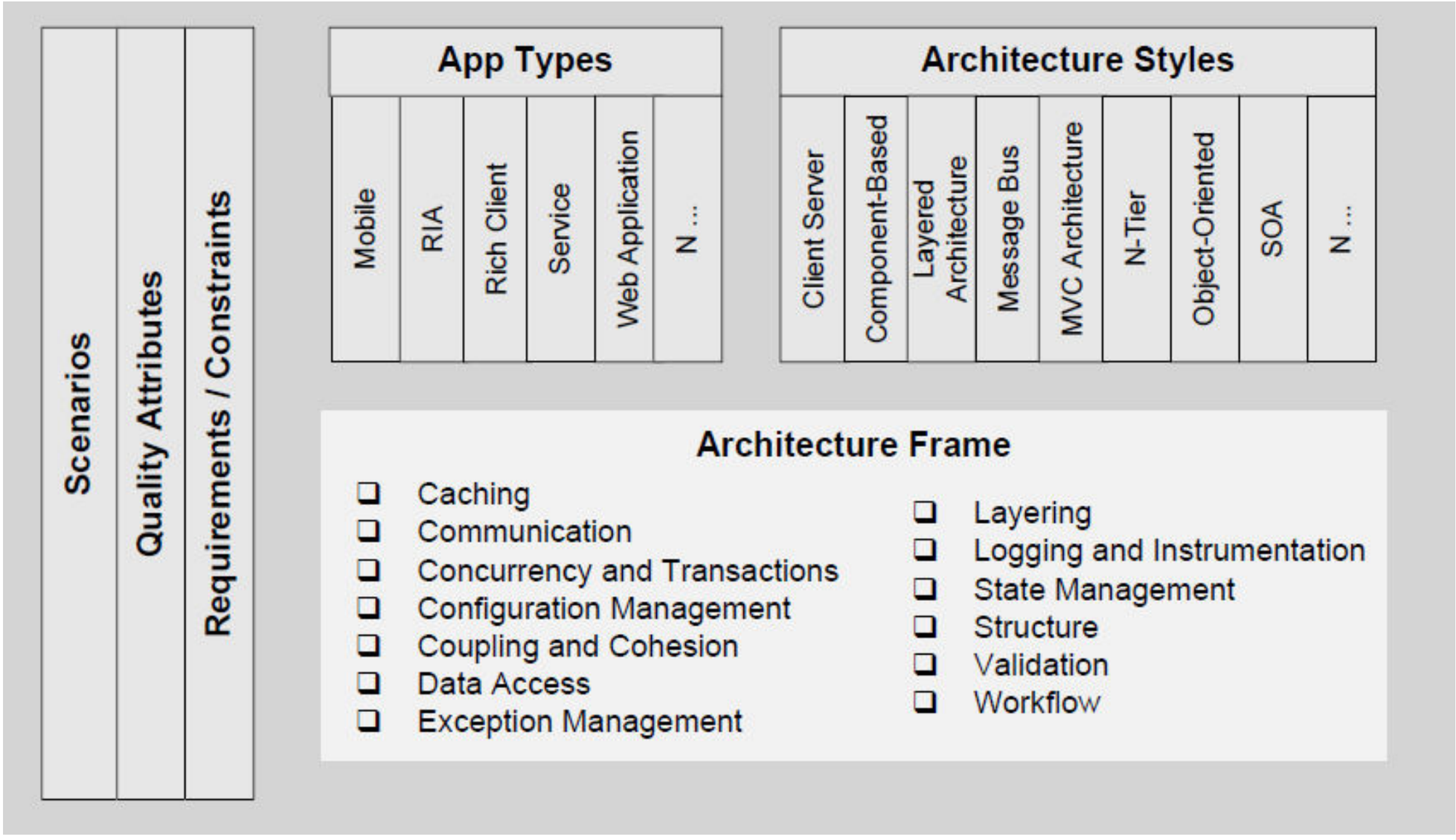
END!

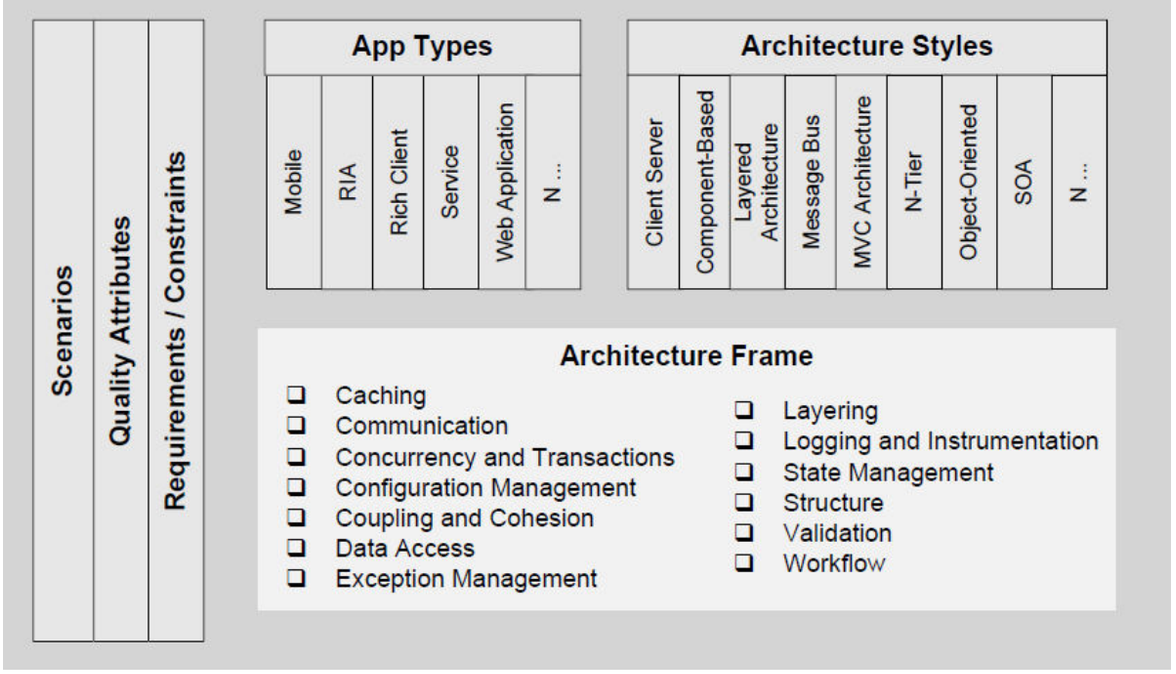
Software Design and Architecture

Topic#140
The Architecture Meta-Frame

considerations that need to be made for the software system that are **architecturally significant**.

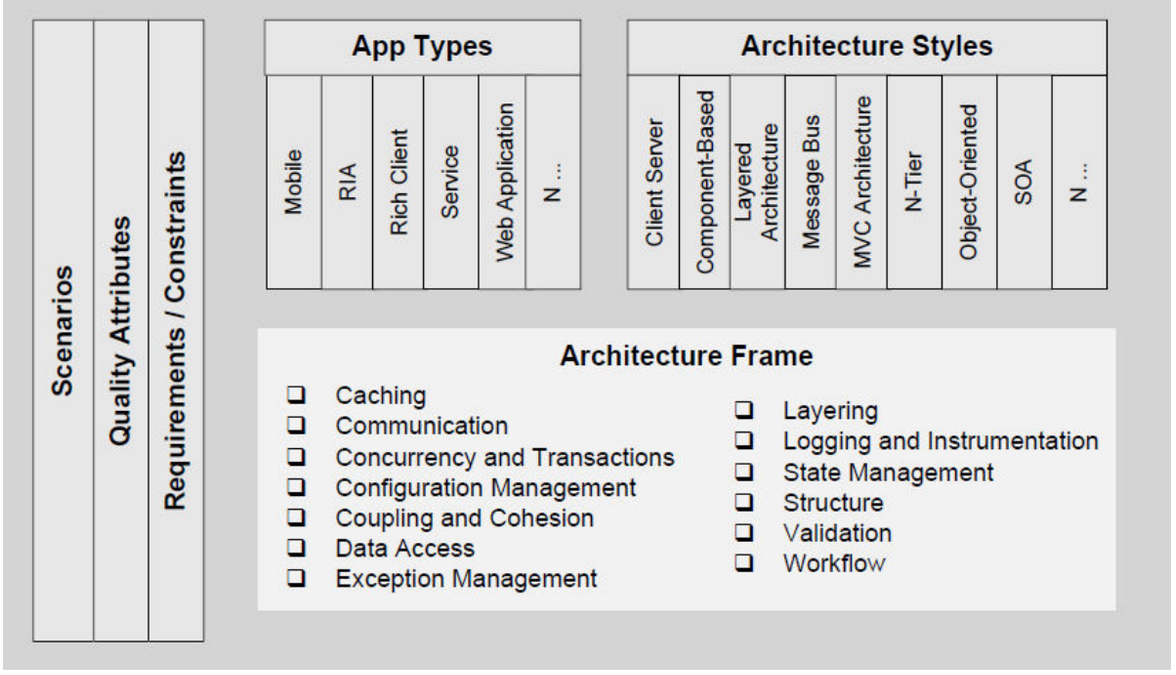
The Architecture Meta-Frame





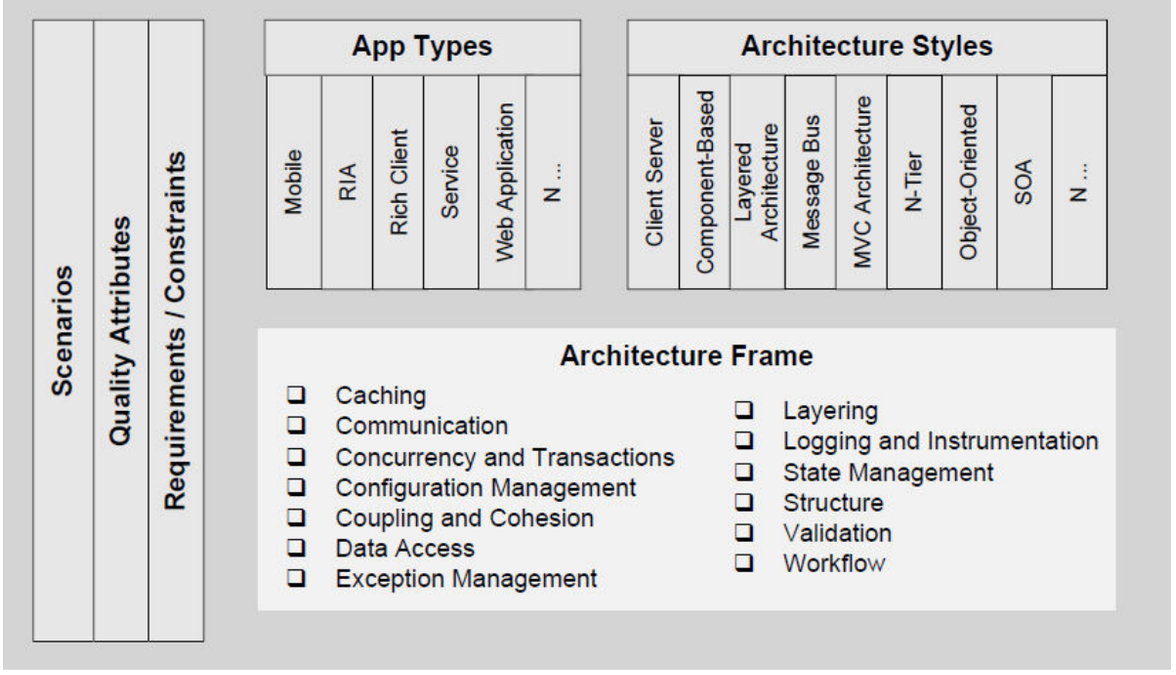
Scenarios.

Application scenarios tie architecture solutions to the real-world scenarios that impact your application design.



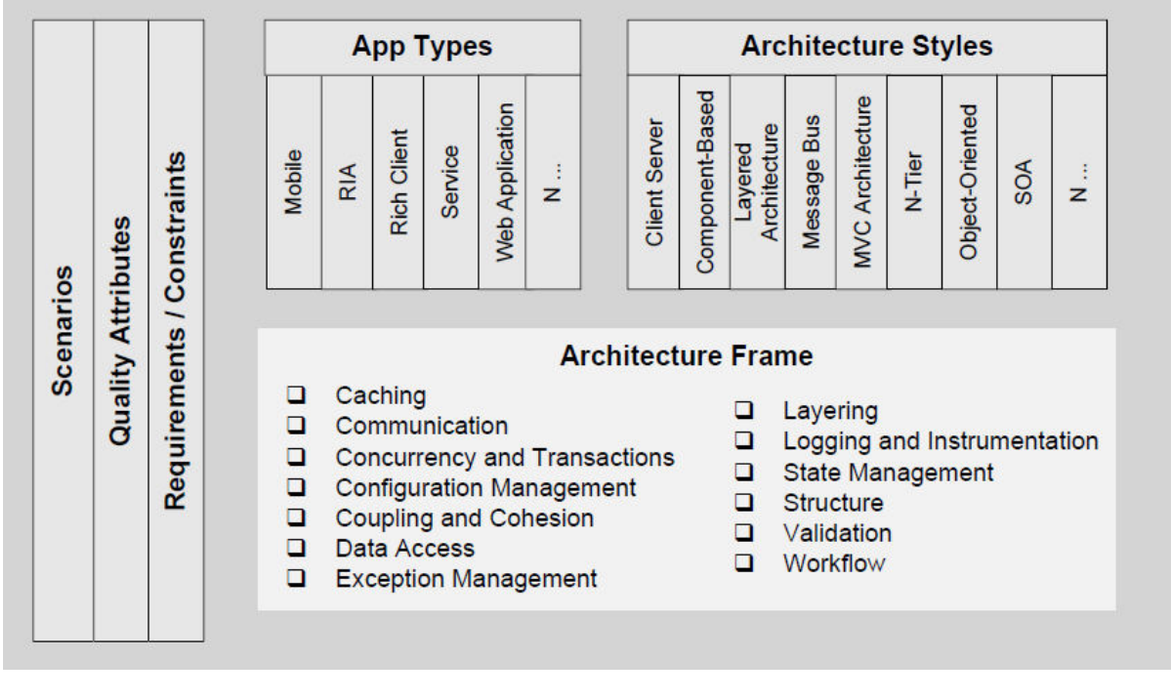
Scenarios - Examples

your application may map to an Internet Web application scenario, which has unique architecture solutions compared to a mobile client application.



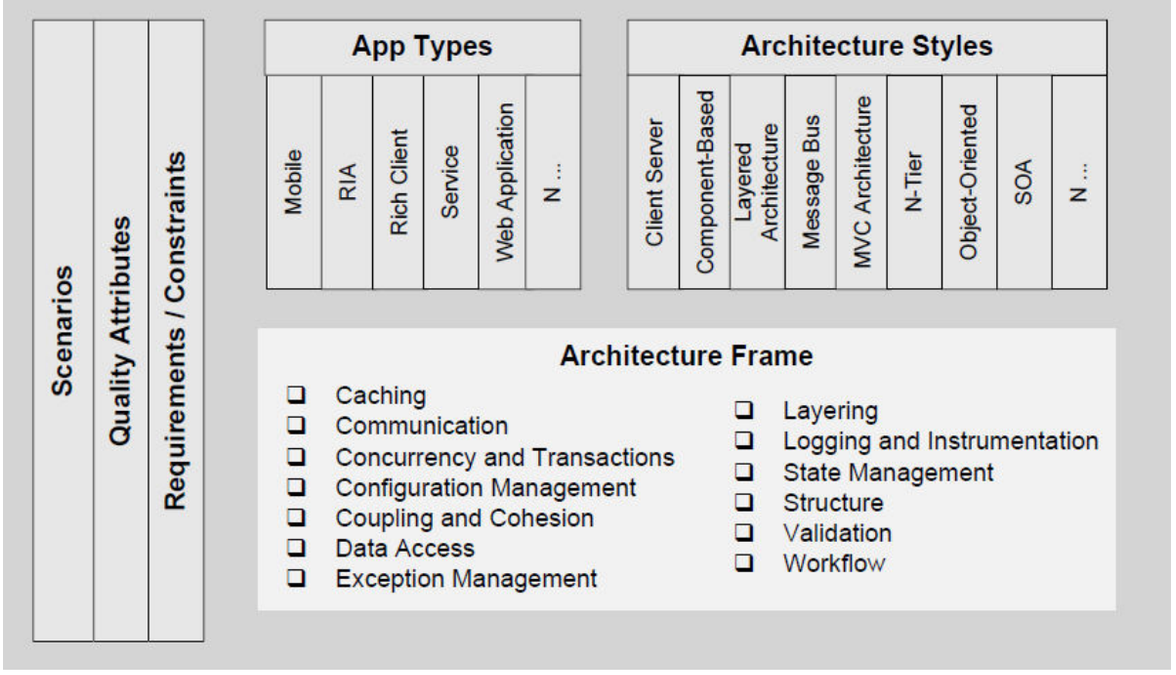
Quality attributes.

Quality attributes represent cross-cutting concerns that apply across application types, and should be considered regardless of architecture style.



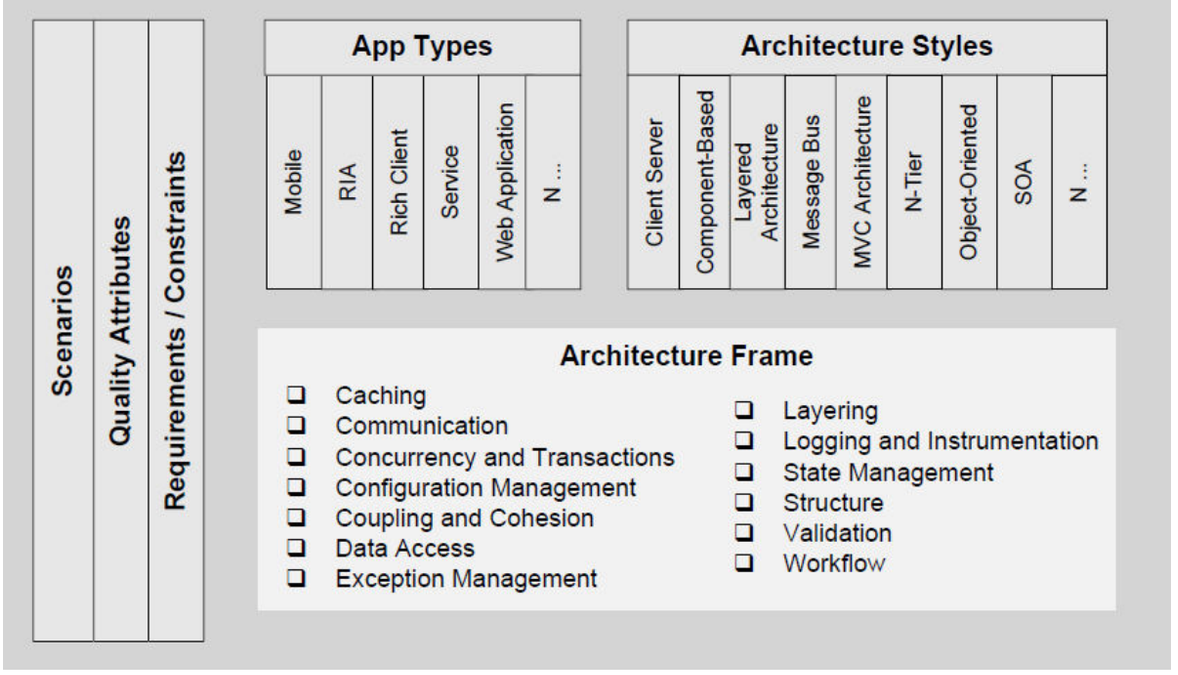
Quality attributes - Examples

Security, performance, maintainability, reusability, etc.



Requirements and constraints.

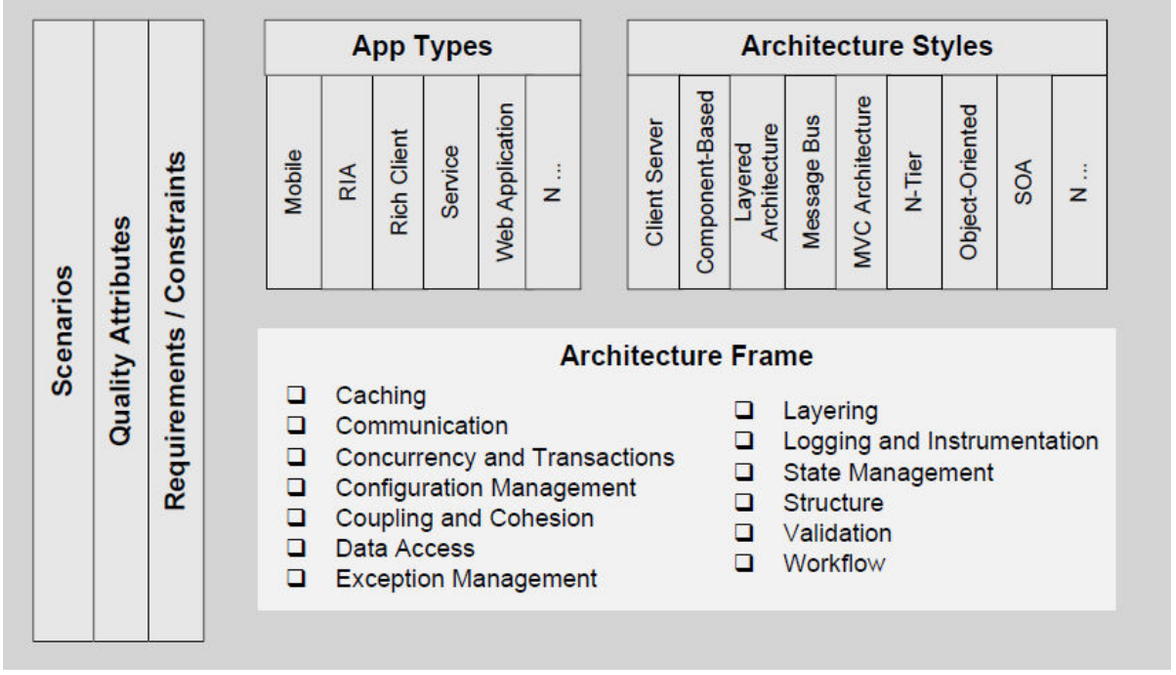
Requirements and constraints narrow the range of possible solutions for your application architecture problems



Application types.

Application types categorize the major application technology stacks.

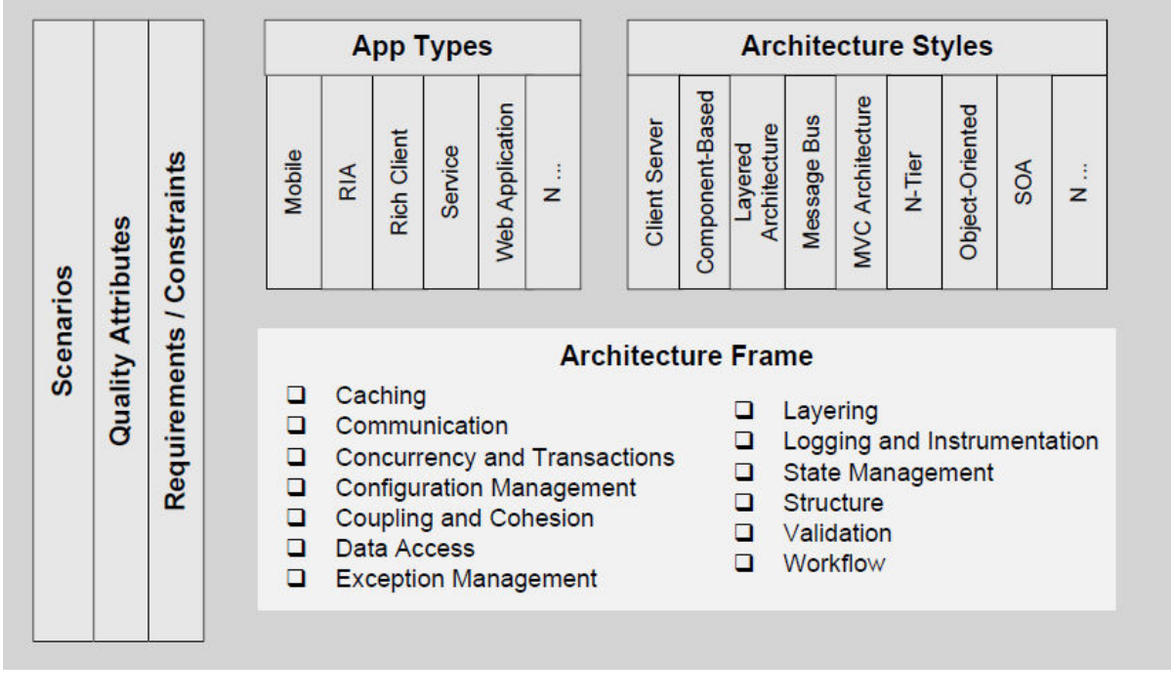
e.g. Mobile, Rich Internet Application, Web App, Service App, etc



Architecture styles.

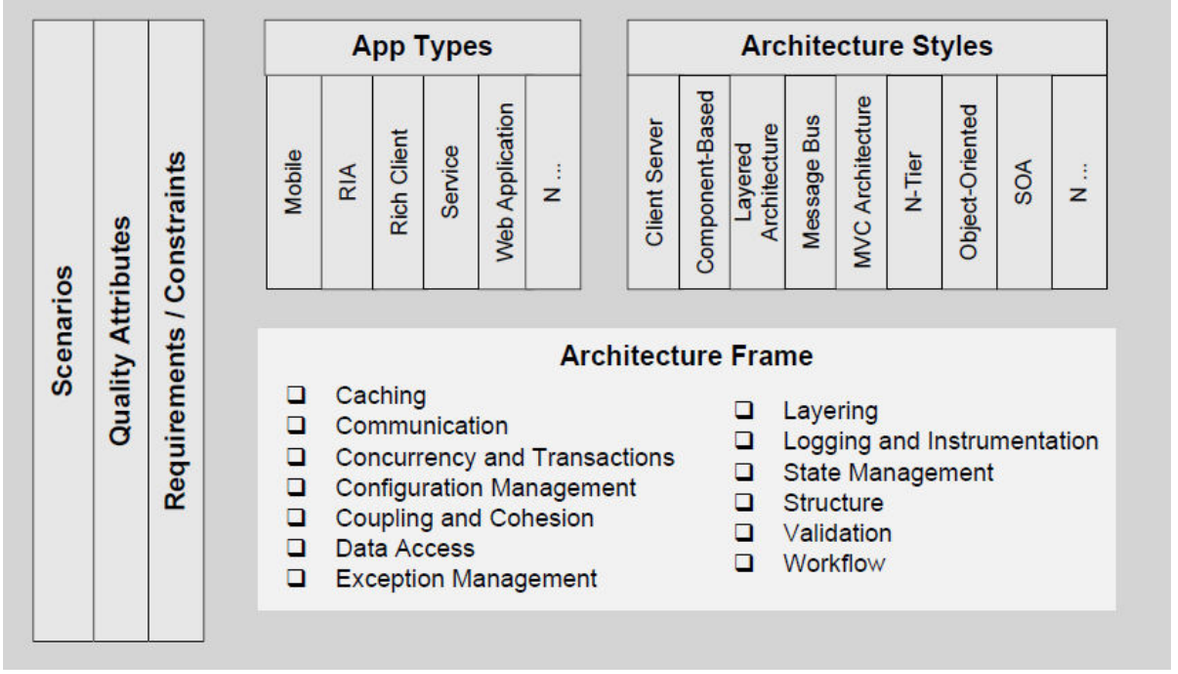
An architectural style is a collection of principles that shapes the design of your application.

Many of these styles overlap and can be used in combination.



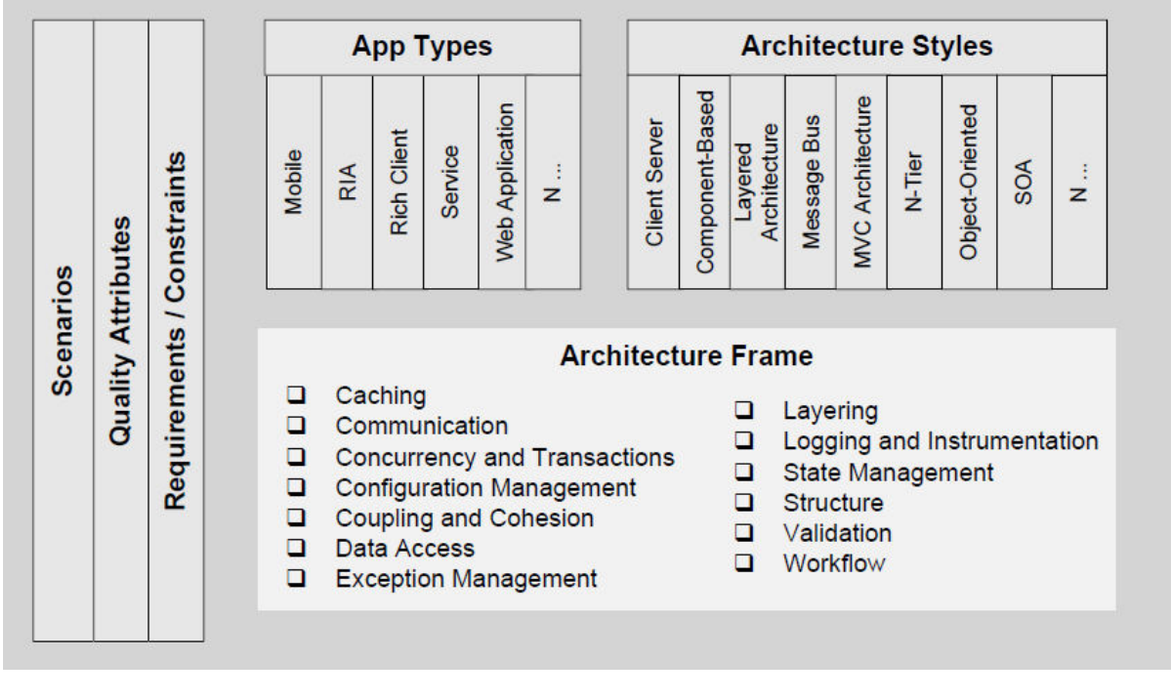
Architecture styles.

Architectural styles tend to be tied both to the application type and to the point in time in which the application was developed.



Architecture frame.

The architecture frame is a collection of hotspots that you can use to analyze your application architecture.



Architecture frame.

This helps you to turn core features such as caching, data access, validation, and workflow into actions.

Software Design and Architecture

Topic#140

END!

Software Design and Architecture

Topic#141
The Architecture Meta-Frame
Quality Attributes: Introduction

© Virtual University of Pakistan

- A quality attribute (QA) is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders.

- a quality attribute is a measure of the “goodness” of a product along some dimension of interest to a stakeholder

- the qualities that must be provided for in a system's architecture

- aka cross-cutting concerns, non-functional requirements, service-level agreements, etc.

© Virtual University of Pakistan

- Systems are frequently redesigned not because they are functionally deficient but because they are difficult to maintain, port, or scale; or they are too slow; or they have been compromised by hackers.

© Virtual University of Pakistan

Software Design and Architecture

Topic#141

END!

Software Design and Architecture

Topic#142
The Architecture Meta-Frame
Architecturally Significant Quality Attributes

- architecture provides the mapping of a system's functionality onto software structures that determines the architecture's support for qualities

- how various qualities are supported by architectural design decisions

- How to express the qualities we want our architecture to provide to the system or systems we are building from it
- How to achieve those qualities
- How to determine the design decisions we might make with respect to those qualities

© Virtual University of Pakistan

Understand how to **capture**, **refine** and **challenge** quality attributes

© Virtual University of Pakistan

- Performance
- Scalability
- Availability
- Security
- Disaster Recovery
- Accessibility
- Monitoring
- Management
- Audit
- Flexibility
- Extensibility
- Maintainability
- Interoperability
- Legal
- Regulatory
- Compliance
- i18n
- L10n

Create a **checklist** of quality attributes you regularly encounter

© Virtual University of Pakistan

Quality Attributes

- Quality attributes can be used to focus your thinking around the critical problems that your design should solve.
- Depending on your requirements, you might or might not need to consider every quality attribute

Quality Attributes

- For example, every application design must consider security and performance, but not every design needs to consider interoperability or scalability.

Quality Attributes

- Understand your requirements and deployment scenarios first so that you know which quality attributes are important for your design.

© Virtual University of Pakistan

Quality Attributes

- quality attributes may conflict; for example, security often requires a tradeoff against performance or usability.
- Analyze and understand the key tradeoffs when designing for security attributes so that side effects do not become obvious later.

© Virtual University of Pakistan

Software Design and Architecture

Topic#142

END!

Software Design and Architecture

Topic#143
The Architecture Meta-Frame
Guidelines for Quality Attributes

- When designing to accommodate quality attributes, consider the following guidelines:
  - Quality attributes are system properties that are separate from the functionality of the system.
  - From a technical perspective, implementing quality attributes can differentiate a good system from a bad one.

© Virtual University of Pakistan

- There are two types of quality attributes:
  - those that are measured at run time, and
  - those that can only be estimated through inspection.
- Analyze the tradeoffs between quality attributes.

- Questions you should ask when considering quality attributes include:
  - What are the key quality attributes required for your application? Identify them as part of the design process.
  - What are the key requirements for addressing these attributes? Are they actually quantifiable?
  - What are the acceptance criteria that will indicate that you have met the requirements?

Software Design and Architecture

Topic#143

END!

© Virtual University of Pakistan

Software Design and Architecture

Topic#144
The Architecture Meta-Frame
Quality Attributes – Description

© Virtual University of Pakistan

Category	Description
Availability	Availability defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period. Availability will be affected by system errors, infrastructure problems, malicious attacks, and system load.
Conceptual Integrity	Conceptual integrity defines the consistency and coherence of the overall design. This includes the way that components or modules are designed, as well as factors such as coding style and variable naming.

Category	Description
Flexibility	Flexibility is the ability of a system to adapt to varying environments and situations, and to cope with changes to business policies and rules. A flexible system is one that is easy to reconfigure or adapt in response to different user and system requirements.
Interoperability	Interoperability is the ability of diverse components of a system or different systems to operate successfully by exchanging information, often by using services. An interoperable system makes it easier to exchange and reuse information internally as well as externally.
Maintainability	Maintainability is the ability of a system to undergo changes to its components, services, features, and interfaces as may be required when adding or changing the functionality, fixing errors, and meeting new business requirements.

Manageability	Manageability defines how easy it is to manage the application, usually through sufficient and useful instrumentation exposed for use in monitoring systems and for debugging and performance tuning.
Performance	Performance is an indication of the responsiveness of a system to execute any action within a given interval of time. It can be measured in terms of latency or throughput. *Latency* is the time taken to respond to any event. *Throughput* is the number of events that take place within given amount of time.
Reliability	Reliability is the ability of a system to remain operational over time. Reliability is measured as the probability that a system will not fail to perform its intended functions over a specified interval of time.

Reusability	Reusability defines the capability for components and subsystems to be suitable for use in other applications and in other scenarios. Reusability minimizes the duplication of components and also the implementation time.
Scalability	Scalability is the ability of a system to function well when there are changes to the load or demand. Typically, the system will be able to be extended by scaling up the performance of the server, or by scaling out to multiple servers as demand and load increase.
Security	Security defines the ways that a system is protected from disclosure or loss of information, and the possibility of a successful malicious attack. A secure system aims to protect assets and prevent unauthorized modification of information.

Supportability	Supportability defines how easy it is for operators, developers, and users to understand and use the application, and how easy it is to resolve errors when the system fails to work correctly.
Testability	Testability is a measure of how easy it is to create test criteria for the system and its components, and to execute these tests in order to determine if the criteria are met. Good testability makes it more likely that faults in a system can be isolated in a timely and effective manner.
Usability	Usability defines how well the application meets the requirements of the user and consumer by being intuitive, easy to localize and globalize, able to provide good access for disabled users, and able to provide a good overall user experience.

Software Design and Architecture

Topic#144

END!

Software Design and Architecture

Topic#145
The Architecture Meta-Frame
Requirements and Constraints

Requirements drive architecture

(use cases, user stories, features, etc)

Software lives in the real world,
and the real world has

constraints

- Constraints
  - a design decision with zero degrees of freedom
  - external factors (such as not being able to train the staff in a new language, or having a business agreement with a software supplier, or pushing business goals of service interoperability) have led those in power to dictate these design outcomes.

Typical constraints include time and budget, technology, people and skills, politics, etc

© Virtual University of Pakistan

Constraints can **sometimes** be prioritised

Software Design and Architecture

Topic#145

END!

Software Design and Architecture

Topic#146
The Architecture Meta-Frame
Application Types

© Virtual University of Pakistan

Application Types

- Your choice of application type will be related both to the technology constraints and the type of user experience you plan to deliver.
- Choosing the right application type is the key part of the process of designing and architecting an application.

Application Types

- Your choice of an appropriate application type is governed by your specific requirements and infrastructure limitations.
- Use scenarios to help you choose an application type.

© Virtual University of Pakistan

Application Types

- For example, if you want to support rich media and graphics delivered over the Internet, a rich Internet application (RIA) is probably the best choice.
- However, if you want to support data entry with forms in an occasionally connected scenario, a rich client is probably the best choice.

© Virtual University of Pakistan
thegradup.com

Application Type

- Mobile applications designed for mobile devices.
- Rich client applications designed to run primarily on a client PC.
- Rich Internet applications designed to be deployed from the Internet, which support rich user interface (UI) and media scenarios.

© Virtual University of Pakistan

Application Type

- Service applications designed to support communication between loosely coupled components.
- Web applications designed to run primarily on the server in fully connected scenarios.

© Virtual University of Pakistan

Software Design and Architecture

Topic#146

END!

© Virtual University of Pakistan

Software Design and Architecture

Topic#147
The Architecture Meta-Frame
Application Types and Deployment Strategy

© Virtual University of Pakistan

Deployment Strategy

- When you design your application architecture, you must take into account corporate policies and procedures, together with the infrastructure on which you plan to deploy your application.

© Virtual University of Pakistan

Deployment Strategy

- Whether or not the target environment is inflexible, your application design must accommodate any restrictions that exist in that environment.
- Your application design must also take into account quality attributes such as security, performance, and maintainability.

Deployment Strategy

- Your application design must also take into account quality attributes such as security, performance, and maintainability.
- Sometimes you must make design tradeoffs due to protocol restrictions and network topologies.

Deployment Strategy

- Identify the requirements and constraints that exist between the application architecture and infrastructure architecture early in the design process.

Deployment Strategy

- This helps you to choose an appropriate deployment topology, and to resolve conflicts between the application and infrastructure architecture early in the process.

Software Design and Architecture

Topic#147

END!

Software Design and Architecture

Topic#148
The Architecture Meta-Frame
Application Types – Description

© Virtual University of Pakistan

Application type	Description
Mobile Application	• Can be developed as a Web application or a rich client application. • Can support occasionally connected scenarios. • Runs on devices with limited hardware resources.
Rich Client Application	• Usually developed as a stand-alone application. • Can support disconnected or occasionally connected scenarios. • Uses the processing and storage resources of the local machine.
Rich Internet Application	• Can support multiple platforms and browsers. • Can be deployed over the Internet. • Designed for rich media and graphical content. • Runs in the browser sandbox for maximum security. • Can use the processing and storage resources of the local machine.

Service Application	• Designed to support loose coupling between distributed components. • Service operations are called using XML-based messages. • Can be accessed from the local machine or remotely, depending on the transport protocol.
Web Application	• Can support multiple platforms and browsers. • Supports only connected scenarios. • Uses the processing and storage resources of the server.

Software Design and Architecture

Topic#148

END!

© Virtual University of Pakistan

Software Design and Architecture

Topic#149
The Architecture Meta-Frame
Architecture Styles

© Virtual University of Pakistan

Architectural Style

- aka architectural pattern
- The choice of architectural styles represents a set of principles that a design will follow
  - an organizing set of ideas that can be used to keep the design cohesive and focused on the key objectives and scenarios.

© Virtual University of Pakistan

Architectural Style

- Each style defines a set of rules that specify:
  - the kinds of components you can use to assemble a system,
  - the kinds of relationships used in their assembly,
  - constraints on the way they are assembled, and
  - assumptions about the meaning of how you put them together.

Architectural Style

- Examples of architectural styles are:
  - client/server, component-based, layered architecture, message-bus, Separated Presentation, 3-tier/N-tier, object-oriented, and service-oriented architecture (SOA).

© Virtual University of Pakistan

Architectural Style

- factors influencing the choice of architectural styles include:
  - the capacity of your organization for design and implementation
  - the capabilities and experience of developers
  - the infrastructure constraints and deployment scenarios available

Architectural Styles

- choice of architectural styles depends upon
  - application type
  - the requirements and constraints
  - the scenarios you want to support, and
  - the styles with which one is most familiar and comfortable.

© Virtual University of Pakistan

Software Design and Architecture

Topic#149

END!

© Virtual University of Pakistan

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 150 – 164

Architecture Meta-Frame – Part II

© Virtual University of Pakistan

Software Design and Architecture

Topic#150
The Architecture Meta-Frame
Architecture Frame – Part I – Cross-Cutting Concerns

© Virtual University of Pakistan

Architecture frame

- The architecture frame is a collection of hotspots that you can use to analyze your application architecture.

- This helps you to turn core features such as caching, data access, validation, and workflow into actions.

© Virtual University of Pakistan

Architecture Frame

- Each hotspot represents key engineering decisions
- Each represents an opportunity to improve your design and build a technically more effective architecture.

© Virtual University of Pakistan

Key Engineering Decisions

- Architecture frame helps in organizing and thinking about key engineering decisions

Architecture Frame

- These categories help one to focus on the most important areas, and obtain the most meaningful and actionable guidance.

© Virtual University of Pakistan

Cross-Cutting Concerns

- Cross-cutting concerns represent key areas of your design that are not related to a specific layer in your application.
- For example, you might want to cache data in the presentation layer, the business layer, and the data access layer.

© Virtual University of Pakistan

Software Design and Architecture

Topic#150

END!

Software Design and Architecture

Topic#151
The Architecture Meta-Frame
Architecture Frame – Part II – Key Cross-Cutting Concerns

© Virtual University of Pakistan

Key Cross-Cutting Concerns - **Authentication and authorization**

- allow one to identify the users of an application with confidence, and to determine the resources and operations to which they should have access.

Key Cross-Cutting Concerns

- **Authentication**.
  - Determine how to authenticate your users and pass authenticated identities across the layers.

Key Cross-Cutting Concerns

- **Authorization**
  - Ensure proper authorization with appropriate granularity within each layer, and across trust boundaries.

© Virtual University of Pakistan

Key Cross-Cutting Concerns - **Caching**

- Caching improves performance, reduces server round trips, and can be used to maintain the state of your application.
- Identify what should be cached, and where to cache, to improve application's performance and responsiveness.

Key Cross-Cutting Concerns - Communication

- Communication strategies determine how will communication be achieved between layers and tiers, including protocol, security, and communication-style decisions.

- Choose appropriate protocols, reduce calls across the network, and protect sensitive data passing over the network.

Key Cross-Cutting Concerns – Exception Management

- Exception-management strategies describe techniques for handling errors, logging errors for auditing purposes, and notifying users of error conditions.
- **Exception management**. Catch exceptions at the boundaries. Do not reveal sensitive information to end users.

Key Cross-Cutting Concerns - Instrumentation and logging

- Logging and instrumentation represent the strategies for logging key business events, security actions, and provision of an audit trail in the case of an attack or failure.
- Instrument all of the business and system-critical events, and log sufficient details to recreate events in your system. Do not log sensitive information.

© Virtual University of Pakistan

Software Design and Architecture

Topic#151

END!

Software Design and Architecture

Topic#152
The Architecture Meta-Frame

Key Engineering Decisions – Part I
Authentication and Authorization

© Virtual University of Pakistan

- allow one to identify the users of an application with confidence, and to determine the resources and operations to which they should have access

© Virtual University of Pakistan

Key Engineering Decisions - *Authentication and Authorization*

- How to store user identities
- How to authenticate callers
- How to authorize callers
- How to flow identity across layers and tiers

© Virtual University of Pakistan

Software Design and Architecture

Topic#152

END!

Software Design and Architecture

Topic#153
The Architecture Meta-Frame

Key Engineering Decisions – Part II
Caching and State

© Virtual University of Pakistan

- Caching improves performance, reduces server round trips, and can be used to maintain the state of your application.

- How to choose effective caching strategies
- How to improve performance by using caching
- How to improve availability by using caching
- How to keep cached data up to date
- How to determine the data to cache

© Virtual University of Pakistan

- How to determine where to cache the data
- How to determine an expiration policy and scavenging mechanism
- How to load the cache data
- How to synchronize caches across a Web or application farm

© Virtual University of Pakistan

Software Design and Architecture

Topic#153

END!

Software Design and Architecture

Topic#154
The Architecture Meta-Frame

Key Engineering Decisions – Part III
Communication

© Virtual University of Pakistan

- Communication strategies determine how will communication be achieved between layers and tiers, including protocol, security, and communication-style decisions

- How to communicate between layers and tiers
- How to perform asynchronous communication
- How to communicate sensitive data

© Virtual University of Pakistan

Software Design and Architecture

Topic#154

END!

Software Design and Architecture

Topic#155
The Architecture Meta-Frame

Key Engineering Decisions – Part IV
Concurrency and Transaction

© Virtual University of Pakistan

- Concurrency is concerned with the way that your application handles conflicts caused by multiple users creating, reading, updating, and deleting data at the same time.
- Transactions are used for important multi-step operations in order to treat them as though they were atomic, and to recover in the case of a failure or error.

- How to handle concurrency between threads
- How to handle distributed transactions
- How to handle long-running transactions
- How to determine appropriate transaction isolation levels

Software Design and Architecture

Topic#155

END!

Software Design and Architecture

Topic#156
The Architecture Meta-Frame

Key Engineering Decisions – Part V
Data Access

© Virtual University of Pakistan

- Data access strategies describe techniques for abstracting and accessing data in your data store.
- This includes
  - data entity design,
  - error management, and
  - managing database connections.

© Virtual University of Pakistan

- How to manage database connections
- How to handle exceptions
- How to improve performance
- How to improve manageability
- How to handle binary large objects (BLOBs)
- How to page records
- How to perform transactions

© Virtual University of Pakistan

Software Design and Architecture

Topic#156

END!

Software Design and Architecture

Topic#157
The Architecture Meta-Frame

Key Engineering Decisions – Part VI
User Experience

© Virtual University of Pakistan

- User experience is the interaction between your users and your application.
- A good user experience can improve the efficiency and effectiveness of the application, while a poor user experience may deter users from using an otherwise well designed application.

© Virtual University of Pakistan

- How to improve task efficiency and effectiveness
- How to improve responsiveness
- How to improve user empowerment
- How to improve the look and feel

© Virtual University of Pakistan

Software Design and Architecture

Topic#157

END!

Software Design and Architecture

Topic#158
Agility and Architecture Design

© Virtual University of Pakistan

- Modern thinking on architecture assumes that your design will evolve over time and that you cannot know everything you need to know up front in order to fully architect your system.

- Your design will generally need to evolve during the implementation stages of the application as you learn more, and as you test the design against real-world requirements.

- Create your architecture with this evolution in mind so that it will be agile in terms of adapting to requirements that are not fully known at the start of the design process.

- Consider the following questions as you create an architectural design with agility in mind:
  - What are the foundational parts of the architecture that represent the greatest risk if you get them wrong?
  - What are the parts of the architecture that are most likely to change, or whose design you can delay until later with little impact?

- What are your key assumptions, and how will you test them?
- What conditions may require you to refactor the design?

- There will be aspects of your design that you must fix early in the process, which may represent significant cost if redesign is required. Identify these areas quickly and invest the time necessary to get them right.

© Virtual University of Pakistan

Software Design and Architecture

Topic#158

END!

Software Design and Architecture

Topic#159
Key Architecture Principles

© Virtual University of Pakistan

Key Architecture Principles

- **Build to change over build to last**.
  - Wherever possible, design your application so that it can change over time to address new requirements and challenges.
- **Model to analyze and reduce risk**.
  - Use threat models to understand risks and vulnerabilities.

© Virtual University of Pakistan

Key Architecture Principles

- **Models and views are a communication and collaboration tool**.
  - Efficient communication of design principles and design changes is critical to good architecture.
  - Use models and other visualizations to communicate your design efficiently and to enable rapid communication of changes to the design.

© Virtual University of Pakistan

Key Architecture Principles

- **Identify key engineering decisions**.
  - Use the architecture frame in this guide to understand the key engineering decisions and the areas where mistakes are most often made.
  - Invest in getting these key decisions right the first time so that the design is more flexible and less likely to be broken by changes.

© Virtual University of Pakistan

Software Design and Architecture

Topic#159

END!

Software Design and Architecture

Topic#160
Incremental and Iterative Approach to Architectural Design

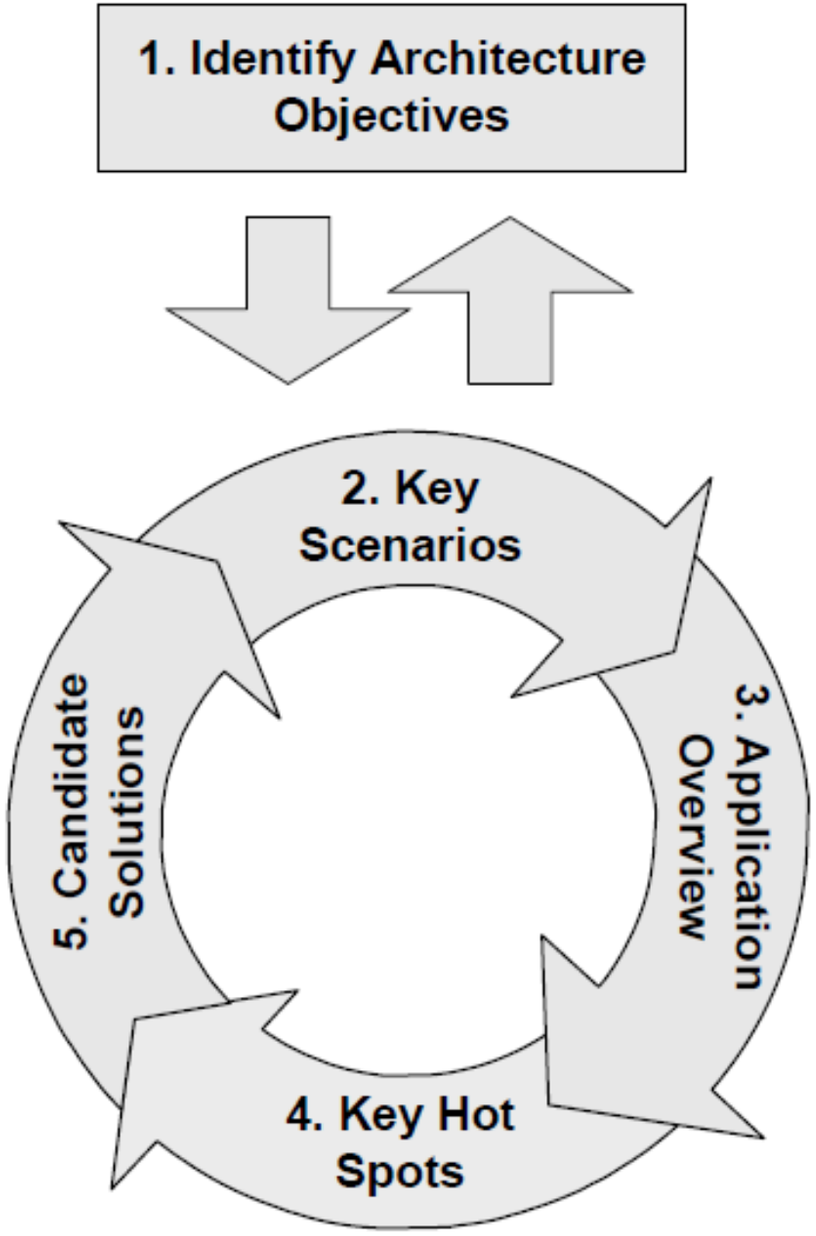
© Virtual University of Pakistan

Incremental and Iterative Approach to Architecture

- Consider using an incremental and iterative approach to refining your architecture.
- Do not try to get it all right the first time—design just as much as you can in order to start testing the design against requirements and assumptions.

- Iteratively add details to the design over multiple passes to make sure that you get the big decisions right first, and then focus on the details.
  - A common pitfall is to dive into the details too quickly and get the big decisions wrong by making incorrect assumptions, or by failing to evaluate your architecture effectively.

© Virtual University of Pakistan


1. Identify Architecture Objectives
2. Key Scenarios
3. Application Overview
4. Key Hot Spots
5. Candidate Solutions

Software Design and Architecture

Topic#160

END!

Software Design and Architecture

Topic#161
Baseline and Candidate Architectures

© Virtual University of Pakistan

Baseline Architecture

- A *baseline architecture* describes the existing system
  - it is how your system looks today.
- If this is a new architecture, your initial baseline is the first high-level architectural design from which candidate architectures will be built.

Candidate Architecture

- A candidate architecture includes the application type, the deployment architecture, the architectural style, technology choices, quality attributes, and cross-cutting concerns.

- Use baseline architectures to get the big picture right, and use candidate architectures to iteratively test and improve your architecture.

- When testing your architecture, consider the following questions:
  - What assumptions have I made in this architecture?
  - What explicit or implied requirements is this architecture meeting?
  - What are the key risks with this approach?
  - What countermeasures are in place to mitigate key risks?
  - In what ways is this architecture an improvement over the baseline or the last candidate architecture?

© Virtual University of Pakistan

Software Design and Architecture

Topic#161

END!

Software Design and Architecture

Topic#162
Architectural Spikes

Architectural Spikes

- An *architectural spike* is an end-to-end test of a small segment of the application.
- The purpose of an architectural spike is to reduce risk and to test potential paths.

Architectural Spikes

- As you evolve your architecture, you may use spikes to explore different scenarios without impacting the existing design.
- An architectural spike will result in a candidate architecture that can be tested against a baseline.

© Virtual University of Pakistan

Architectural Spikes

- If the candidate architecture is an improvement, it can become the new baseline from which new candidate architectures can be created and tested.

Architectural Spikes

- This iterative and incremental approach allows you to get the big risks out of the way first, iteratively render your architecture, and use architectural tests to prove that each new baseline is an improvement over the last.

© Virtual University of Pakistan

- Consider the following questions to help you test a new candidate architecture that results from an architectural spike:
  - Does this architecture introduce new risks?
  - Does this architecture mitigate additional known risks?

- Does this architecture meet additional requirements?
- Does this architecture enable architecturally significant use cases?
- Does this architecture address quality attribute concerns?
- Does this architecture address additional cross-cutting concerns?

© Virtual University of Pakistan

Software Design and Architecture

Topic#162

END!

Software Design and Architecture

Topic#163
Architecturally Significant Use Cases

© Virtual University of Pakistan

Architecturally Significant Use Cases

- Architecturally significant use cases are those that meet the following criteria:
  - They are important for the success and acceptance of the deployed application.
  - They exercise enough of the design to be useful in evaluating the architecture.

© Virtual University of Pakistan

Architecturally Significant Use Cases

- After you have determined architecturally significant use cases for your application, you can use them as a way to evaluate the success or failure of candidate architectures.

Architecturally Significant Use Cases

- If the candidate architecture addresses more use cases, or addresses existing use cases more effectively, it will help you to determine that this candidate architecture is an improvement over the baseline architecture.

© Virtual University of Pakistan

Software Design and Architecture

Topic#163

END!

© Virtual University of Pakistan

Software Design and Architecture

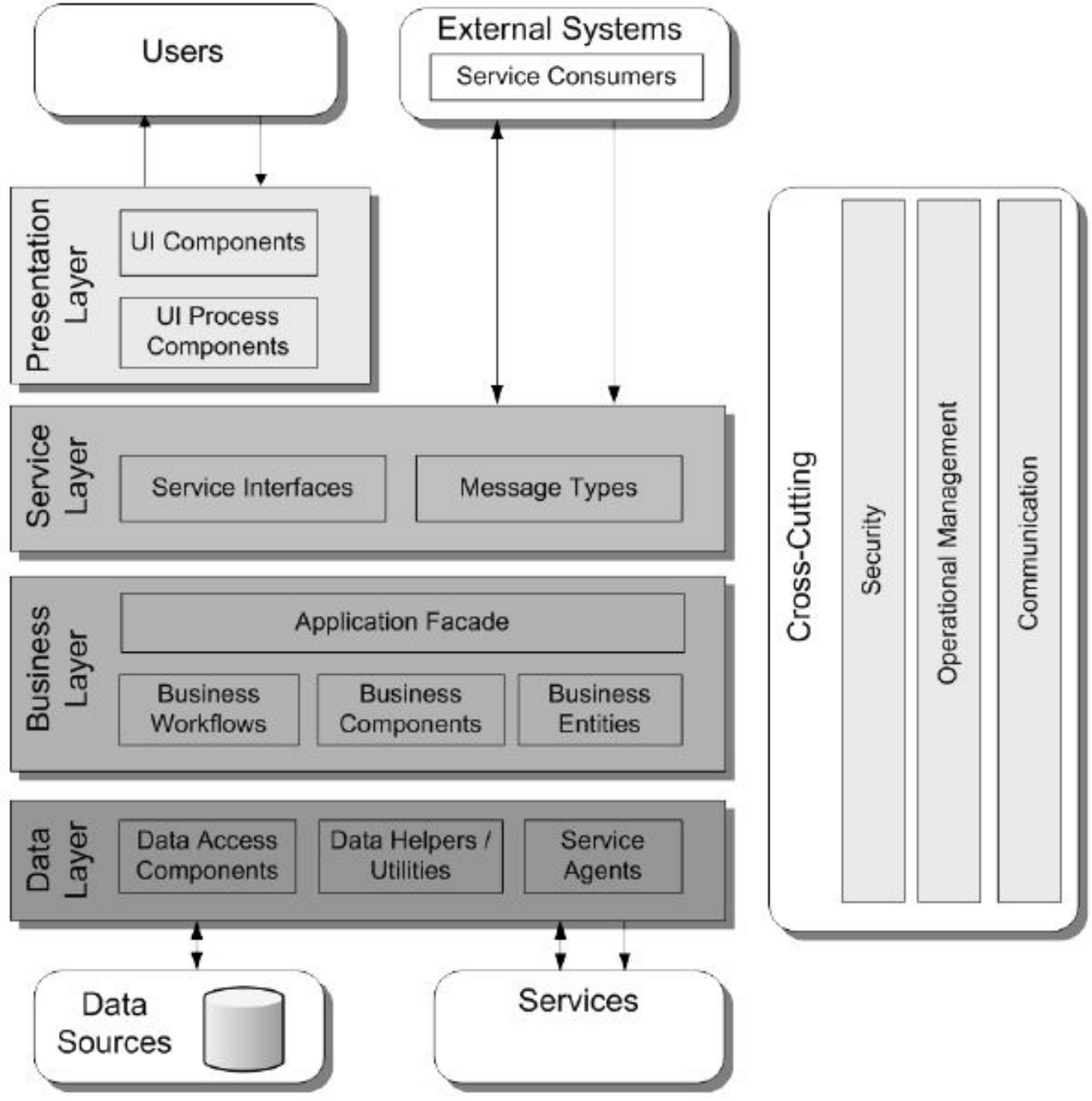
Topic#164
Reference Application Architecture

- Represents a canonical view of a typical application architecture, using a layered style to separate functional areas into separate layers.
- The reference application architecture demonstrates how a typical application might interact with its users, external systems, data sources, and services.

- The reference application architecture also shows how cross-cutting concerns such as security and communication impact all of the layers in your design, and must be designed with the entire application in mind.

© Virtual University of Pakistan

Microsoft's reference application architecture



Software Design and Architecture

Topic#164

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 165 – 184

Software Architecture Patterns

© Virtual University of Pakistan

Software Design and Architecture

Topic#165
Software Architecture Patterns - Introduction

© Virtual University of Pakistan

Software Architecture Patterns
aka architecture styles

- A **software architecture pattern** is a solution to a recurring problem that is well understood, in a particular context.
- Each pattern consists of a context, a problem, and a solution.

Software Architecture Patterns

- The problem may be to overcome some challenge, take advantage of some opportunity, or to satisfy one or more quality attributes.
- Patterns codify knowledge and experience into a solution that we can reuse.

© Virtual University of Pakistan

Software Architecture Patterns

- Using patterns simplifies design and allows us to gain the benefits of using a solution that is proven to solve a particular design problem.

© Virtual University of Pakistan

Software Architecture Patterns

- When working with others who are familiar with patterns, referencing one of them provides a shorthand with which to reference a solution, without having to explain all its details.
- As a result, they are useful during discussions to communicate ideas.

Software Architecture Patterns

- Each pattern has its own characteristics, strengths, and weaknesses.

Software Design and Architecture

Topic#165

END!

Software Design and Architecture

Topic#166
Difference between Software Architecture Patterns and Design Patterns

© Virtual University of Pakistan

- Software architecture patterns are similar to design patterns, except that they are broader in scope and are applied at the architecture level.

- Architecture patterns tend to be more coarse-grained and focus on architectural problems, while design patterns are more fine-grained and solve problems that occur during implementation.

© Virtual University of Pakistan

- A software architecture pattern provides a high-level structure and behavior for software systems.
- It is a grouping of design decisions that have been repeated and used successfully for a given context.

© Virtual University of Pakistan

- They address and satisfy architectural drivers and as a result, the ones that we decide to use can really shape the characteristics and behavior of the architecture.

© Virtual University of Pakistan

- Software architecture patterns provide the structure and main components of the software system being built.

- They introduce design constraints, which reduce complexity and help to prevent incorrect decisions.

© Virtual University of Pakistan

- When a software architecture pattern is followed consistently during design, we can anticipate the properties that the software system will exhibit.
- This allows us to consider whether a design will satisfy the requirements and quality attributes of the system.

- Software architecture patterns can be applied to the entire software system or to one of the subsystems.
- Consequently, more than one software architecture pattern can be used in a single software system. These patterns can be combined to solve problems.

© Virtual University of Pakistan

Software Design and Architecture

Topic#166

END!

Software Design and Architecture

Topic#167
Types of Architecture Patterns

The Three Categories of Structures in Architectural Design

- Static Structures
  - Module Structures
- Dynamic Structures
  - *component-and-connector* (C&C) structures – components are runtime entities
- Deployment Structures
  - aka allocation structures

The Three Categories of Structures in Architectural Design

- Static Structures
  - Module Structures
    - assigned areas of functional responsibility
    - how the system is to be structured as a set of code or data units that have to be constructed or procured

The Three Categories of Structures in Architectural Design

- *component-and-connector* (C&C) structures – components are runtime entities
  - how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors)
  - components - the principal units of computation
  - connectors - communication vehicles among components

The Three Categories of Structures in Architectural Design

- Deployment Structures
  - how the system will relate to non-software structures in its environment
  - show the relationship between the software elements and elements in one or more external environments in which the software is created and executed

- Patterns can be categorized by the dominant type of elements that they show
  - module patterns show modules,
  - component-and-connector (C&C) patterns show components and connectors
  - allocation patterns show a combination of software elements (modules, components, connectors) and non-software elements.

- Most published patterns are C&C patterns, but there are module patterns and allocation patterns as well.

Software Design and Architecture

Topic#167

END!

Software Design and Architecture

Topic#168
Module Patterns
Layered Pattern – I: Introduction

© Virtual University of Pakistan

- **Context:**
  - All complex systems experience the need to develop and evolve portions of the system independently.
  - For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.

© Virtual University of Pakistan
thegradup.com

- **Problem:**
  - The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.

© Virtual University of Pakistan

- **Solution:**
  - the layered pattern divides the software into units called layers.
  - Each layer is a grouping of modules that offers a cohesive set of services.

- one of the most common techniques.
- In a **layered architecture**, the software application is divided into various horizontal layers, with each layer located on top of a lower layer.

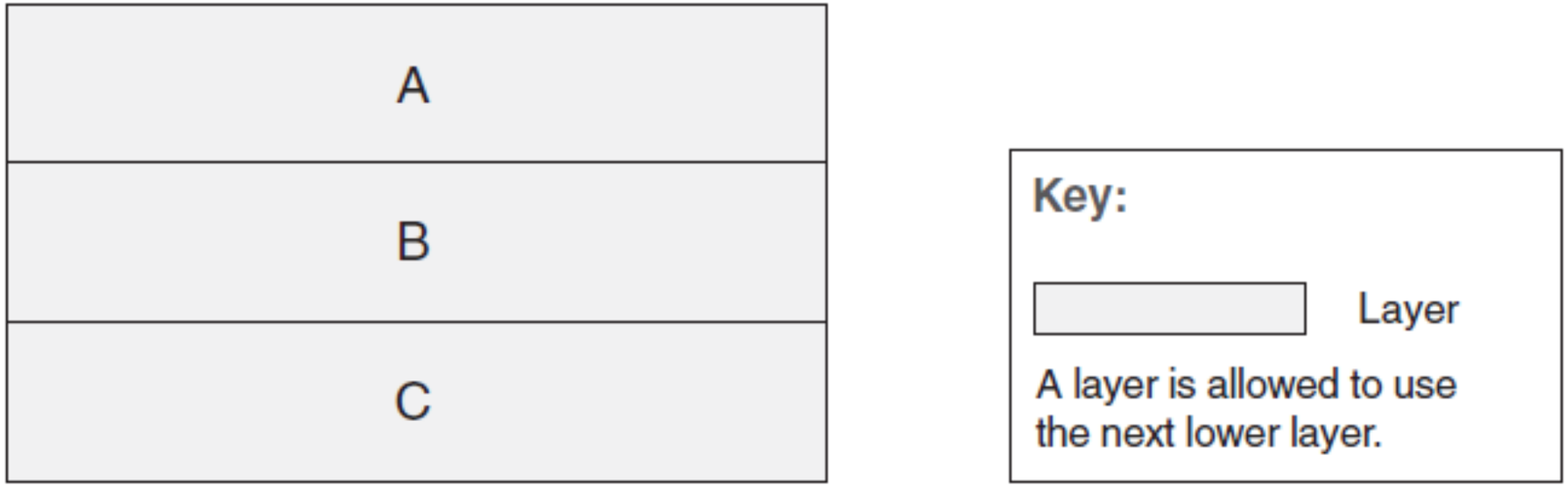


FIGURE 13.1 Stack-of-boxes notation for layered designs

© Virtual University of Pakistan

- Each layer is dependent on one or more layers below it (depending on whether the layers are open or closed), but is independent of the layers above it.

Software Design and Architecture

Topic#168

END!

Software Design and Architecture

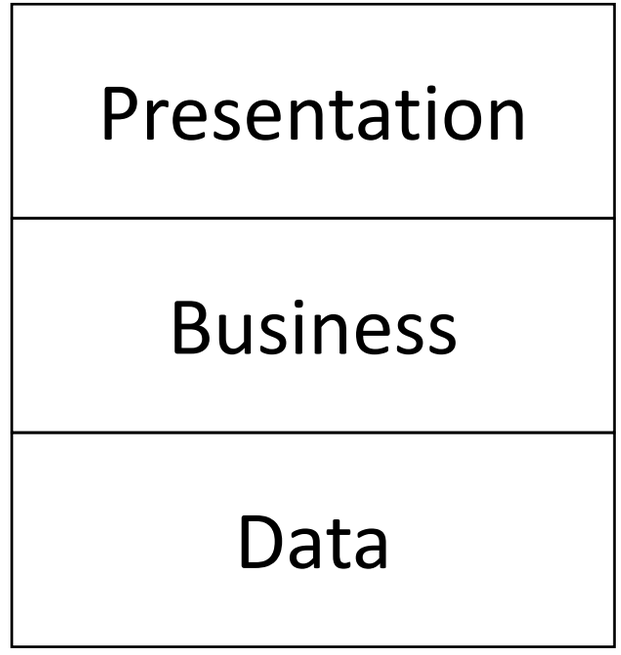
Topic#169
Module Patterns
Layered Pattern – II: Open vs Closed Layers

© Virtual University of Pakistan

- **Open versus closed layers**
  - With a closed layer, requests that are flowing down the stack from the layer above must go through it and cannot bypass it.

© Virtual University of Pakistan

- **Open versus closed layers**
- Example:
  - three-layer architecture with presentation, business, and data layers
  - if the business layer is closed, the presentation layer must send all requests to the business layer and cannot bypass it to send a request directly to the data layer.



© Virtual University of Pakistan

- Closed layers provide layers of isolation, which makes code easier to change, write, and understand.
- This makes the layers independent of each other, such that changes made to one layer of the application will not affect components in the other layers.

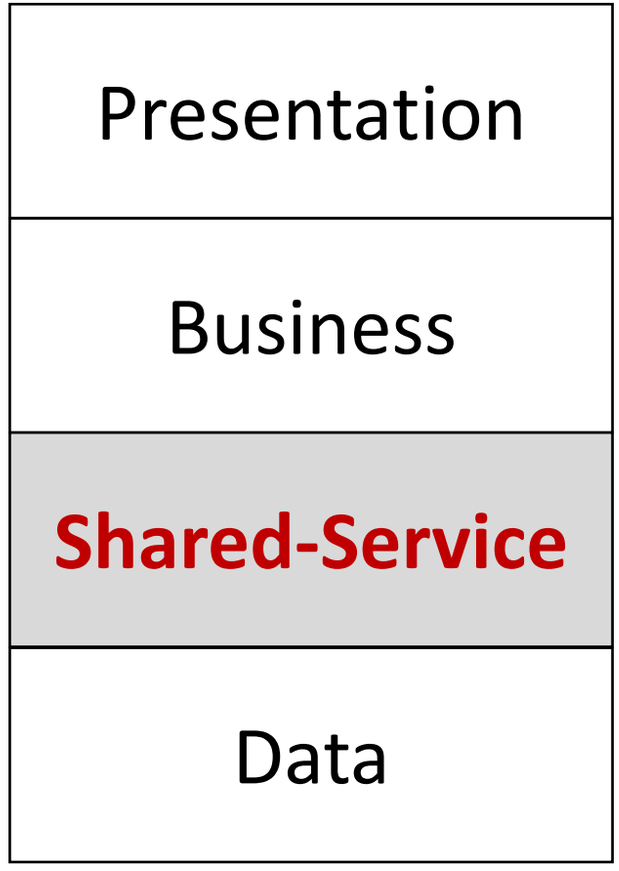
© Virtual University of Pakistan

- If the layers are open, this increases complexity.
- Maintainability is lowered because multiple layers can now call into another layer, increasing the number of dependencies and making changes more difficult.

© Virtual University of Pakistan

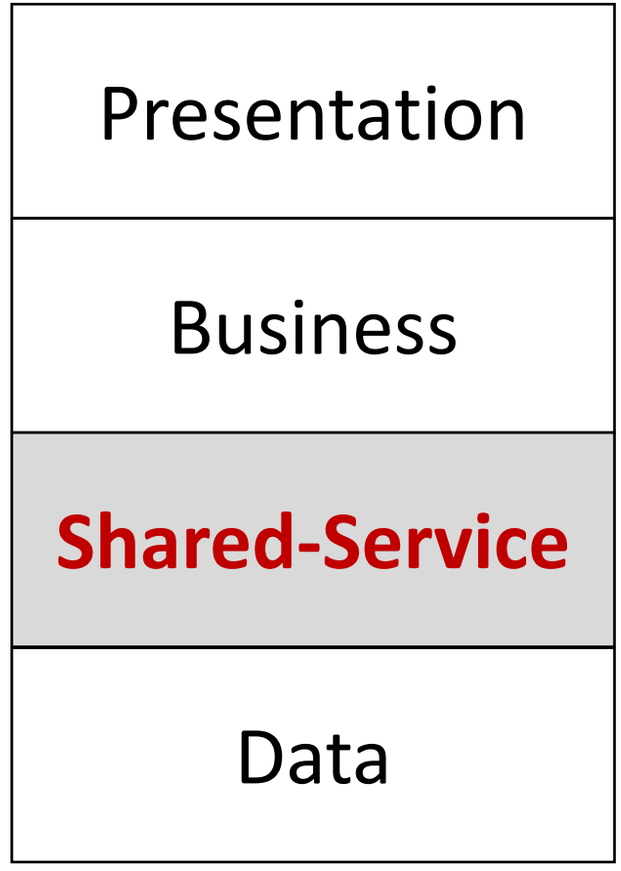
- unnecessary traffic can result when each layer must be passed even if one or more of them is just passing requests on to the next layer.

- added a shared services layer between the business and data layers.
  - may contain reusable components needed by multiple components in the business layer.
  - placed it below the business layer so that only the business layer has access to it.



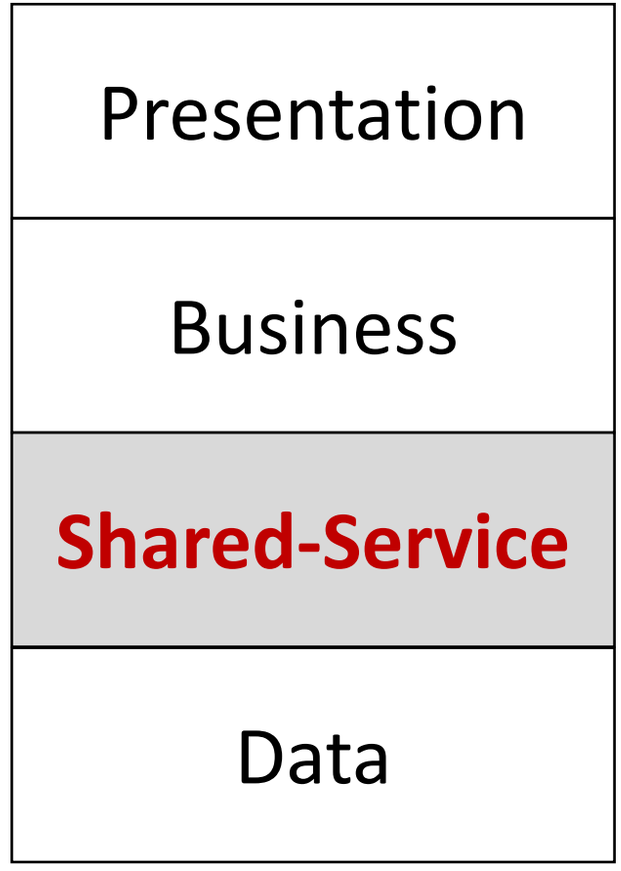
© Virtual University of Pakistan

- all requests from the business layer to the data layer must go through the shared services layer even though nothing is needed from that layer.

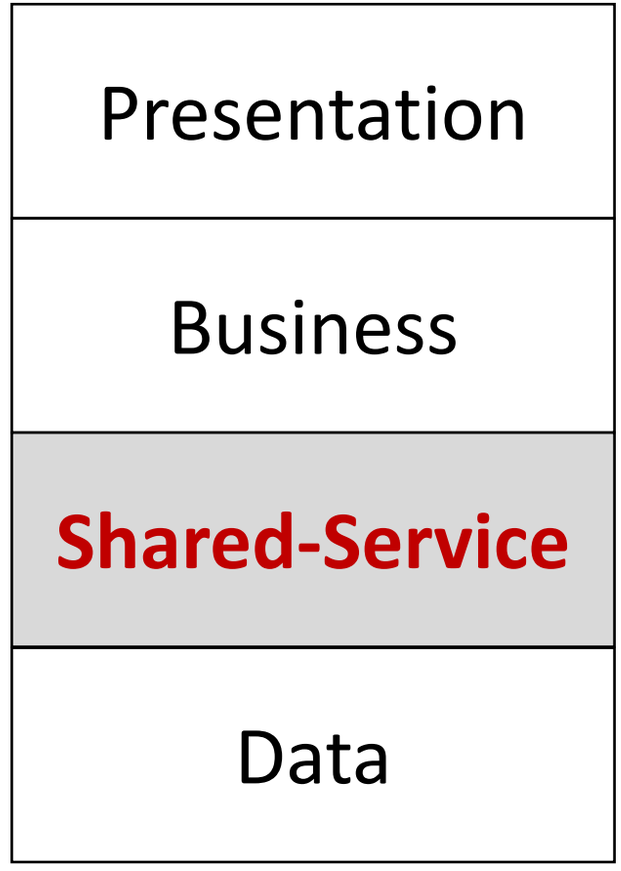


© Virtual University of Pakistan

- If the shared services layer was made open (aka layer bridging), requests to the data layer can be made directly from the business layer.



- there are advantages to closed layers and achieving layers of isolation.
- However, it might be appropriate to open a layer.
- It is not necessary to make all of the layers open or closed.
- You may selectively choose which layers, if any, are open.



© Virtual University of Pakistan

Software Design and Architecture

Topic#169

END!

Software Design and Architecture

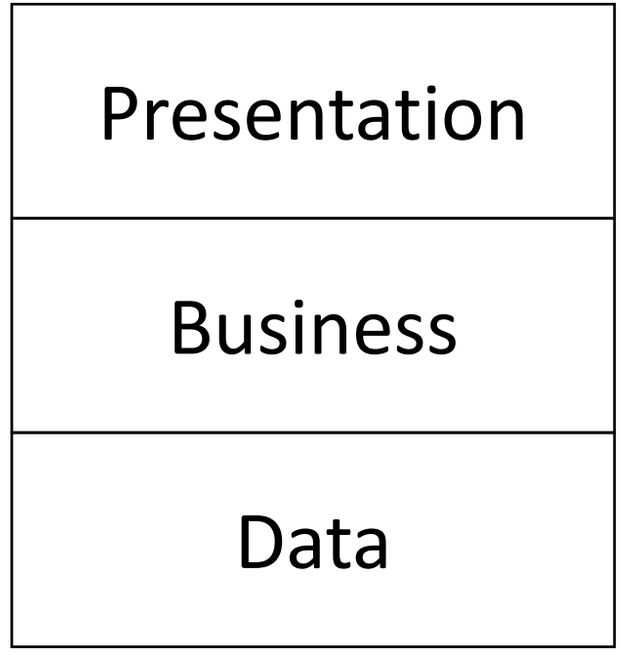
Topic#170
Module Patterns
Layered Pattern – III: Tiers vs Layers

© Virtual University of Pakistan

- Layers – Logical separation
- Tiers – Physical separation

© Virtual University of Pakistan

- When partitioning application logic, layers are a way to organize functionality and components.
  - For example, in a three-layered architecture, the logic may be separated into presentation, business, and data layers.



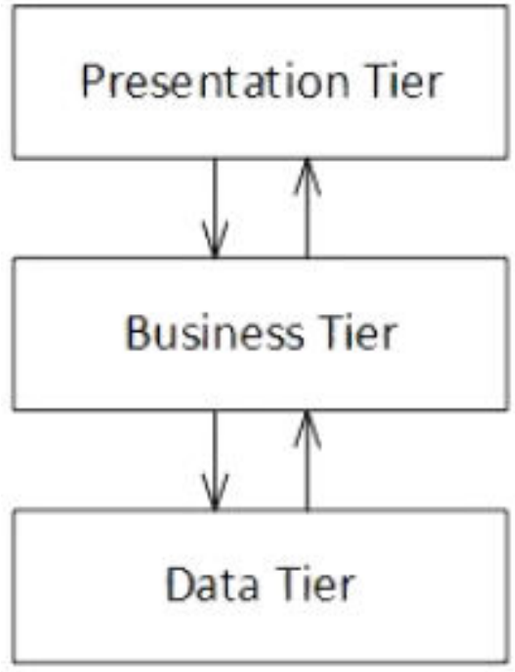
- When a software architecture is organized into more than one layer, it is known as a **multi-layer architecture**.
- Different layers do not necessarily have to be located on different physical machines.
- It is possible to have multiple layers on the same machine.

© Virtual University of Pakistan
thegradup.com

- Tiers concern themselves with the physical location of the functionality and components.

© Virtual University of Pakistan

- A three-tiered architecture with presentation, business, and data tiers implies that those three tiers have been physically deployed to three separate machines and are each running on those separate machines.



© Virtual University of Pakistan

- When a software architecture is partitioned into multiple tiers, it is known as a **multi-tier architecture**.

© Virtual University of Pakistan

Software Design and Architecture

Topic#170

END!

Software Design and Architecture

Topic#171
Module Patterns
Layered Pattern – IV: Advantages of layered architectures

© Virtual University of Pakistan

- This pattern reduces complexity by achieving a Separation of Concerns (SoC).
  - Each layer is independent and you can understand it on its own without the other layers.
  - Complexity can be abstracted away in a layered application, allowing us to deal with more complex problems.

© Virtual University of Pakistan

- Dependencies between layers can be minimized in a layered architecture, which further reduces complexity.
  - For example, the presentation layer does not need to depend directly on the data layer and the business layer does not depend on the presentation layer.
- Minimizing dependencies also allows you to substitute implementations for a particular layer without affecting the other layers.

- they can make development easier.
  - The pattern is pervasive and well known to many developers, which makes using it easy for the development team.
  - Due to the way that the architecture separates the application logic, it matches up well with how many organizations hire their resources and allocate tasks during a project.
  - Each layer requires a particular skill set and suitable resources can be assigned to work on each layer.

© Virtual University of Pakistan

- For example, UI developers for the presentation layer, and backend developers for the business and data layers.

- This architecture pattern increases the testability quality attribute of software applications.
  - Partitioning the application into layers and using interfaces for the interaction between layers allows us to isolate a layer for testing and either mock or stub the other layers.

© Virtual University of Pakistan

- For example, you can perform unit testing on classes in your business layer without the presentation and data layers.
- The business layer is not dependent on the presentation layer and the data layer can be mocked or stubbed.

- Applications using a layered architecture may have higher levels of reusability if more than one application can reuse the same layer.
  - For example, if multiple applications target the same business and/or data layers, those layers are reusable.

- When an application using a layered architecture is deployed to different tiers, there are additional benefits:

- increased scalability as more hardware can be added to each tier
- greater levels of availability when multiple machines are used per layer.
  - Uptime is increased because, if a hardware failure takes place in a layer, other machines can take over.
- Having separate tiers enhances security as firewalls can be placed in between the various layers.
- If a layer can be reused for multiple applications, it means that the physical tier can be reused as well.

Software Design and Architecture

Topic#171

END!

Software Design and Architecture

Topic#172
Module Patterns
Layered Pattern – V: Disadvantages of layered architectures

© Virtual University of Pakistan

- Although the layers can be designed to be independent, a requirement change may require changes in multiple layers.
- This type of coupling lowers the overall agility of the software application.
  - For example, adding a new field will require changes to multiple layers: the presentation layer so that it can be displayed, the business layer so that it can be validated/saved/processed, and the data layer because it will need to be added to the database.

- This can complicate deployment because, even for a change such as this, an application may require multiple parts (or even the entire application) to be deployed.
- more code will be necessary for layered applications.
  - This is to provide the interfaces and other logic that are necessary for the communication between the multiple layers.

- Development teams have to be diligent about placing code in the correct layer so as not to leak logic to a layer that belongs in another layer.
  - Examples of this include placing business logic in the presentation layer or putting data-access logic in the business layer.

© Virtual University of Pakistan

- there can be inefficiencies in having a request go through multiple layers.

© Virtual University of Pakistan

- moving from one layer to another sometimes requires data representations to be transformed.

© Virtual University of Pakistan

- One way to mitigate this disadvantage is to allow some layers to be open but this should only be done if it is appropriate to open a layer.

© Virtual University of Pakistan

- disadvantages to layered architectures when they are deployed to multiple tiers:
  - when layers are deployed to separate physical tiers, there is an additional performance cost.
    - With modern hardware, this cost may be small but it still won't be faster than an application that runs on a single machine.

- There is a greater monetary cost associated with having a multi-tier architecture.
  - The more machines are used for the application, the greater the overall cost.
  - Unless the hosting of the software application is handled by a cloud provider or has otherwise been outsourced, an internal team will be needed to manage the physical hardware of a multi-tier application.

© Virtual University of Pakistan

Software Design and Architecture

Topic#172

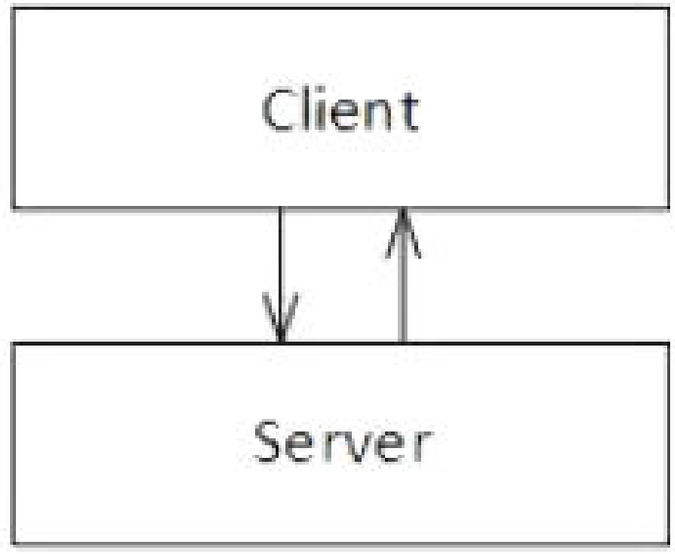
END!

Software Design and Architecture

Topic#173
Module Patterns
Layered Pattern – VI: Client-server architecture (two-tier architecture)

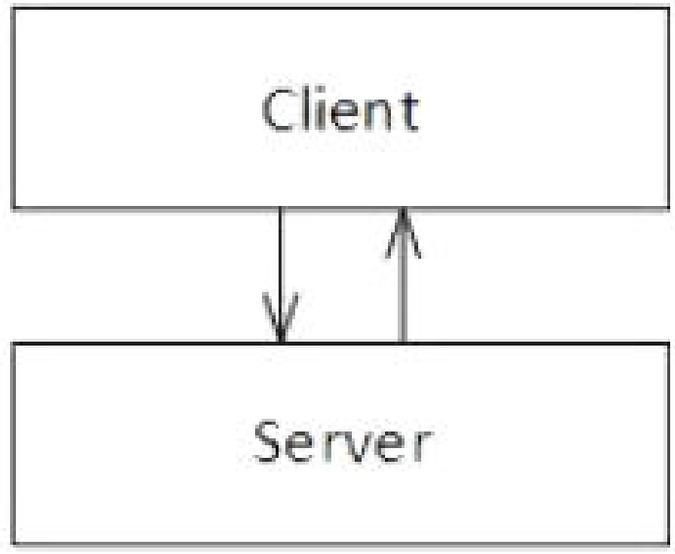
© Virtual University of Pakistan

- **client-server architecture**, also known as a two-tier architecture
- distributed application
- clients and servers communicate with each other directly.
- A client requests some resource or calls some service provided by a server and the server responds to the requests of clients.
- There can be multiple clients connected to a single server



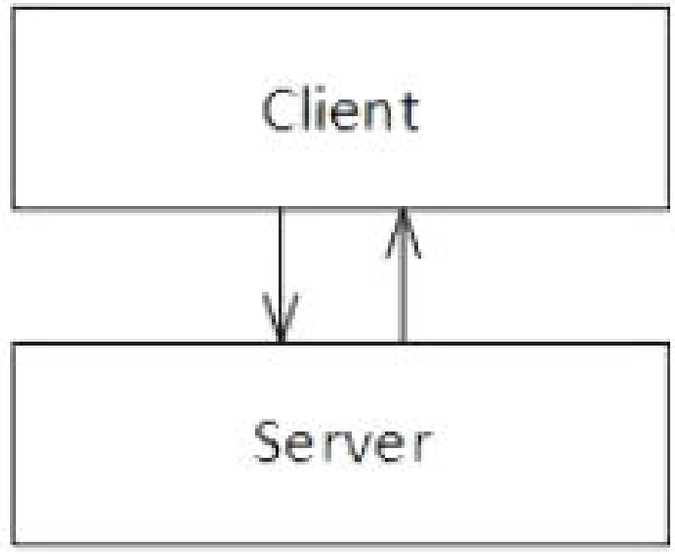
© Virtual University of Pakistan

- The **Client** part of the application contains the user interface code and the **Server** contains the database
- The majority of application logic in a client-server architecture is located on the server, but some of it could also be located in the client.
- The application logic located on the server might exist in software components, in the database, or both.



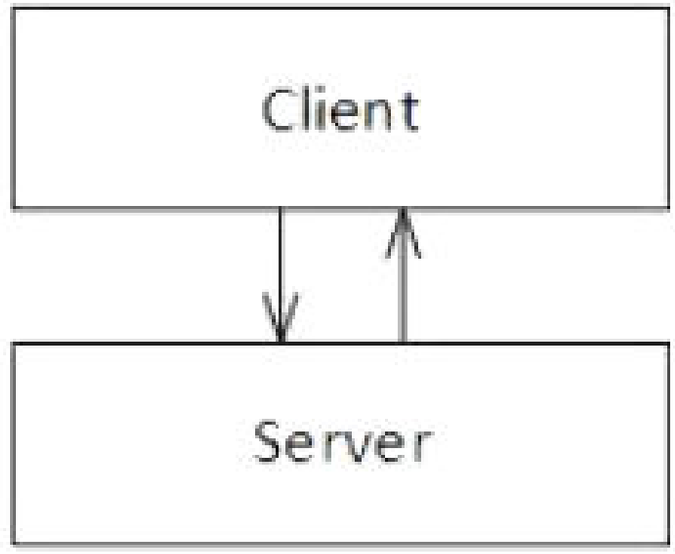
© Virtual University of Pakistan

- When the client contains a significant portion of the logic and is handling a large share of the workload, it is known as a thick, or fat, client.
- When the server is doing that instead, the client is known as a thin client.



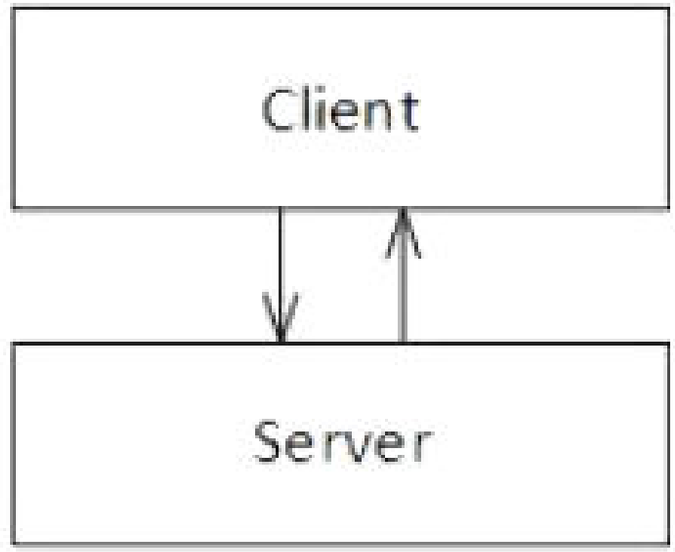
© Virtual University of Pakistan

- In some client-server applications, the business logic is spread out between the client and the server.
- If consistency isn't applied, it can make it difficult to always know where a particular piece of logic is located.
- If a team isn't diligent, business logic might be duplicated on the client and the server.

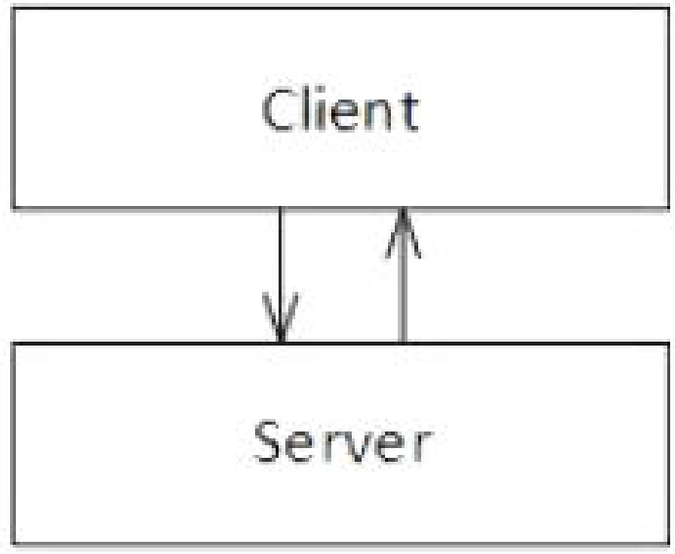


© Virtual University of Pakistan

- There may be instances in which the same piece of logic is needed on both the client and the server.

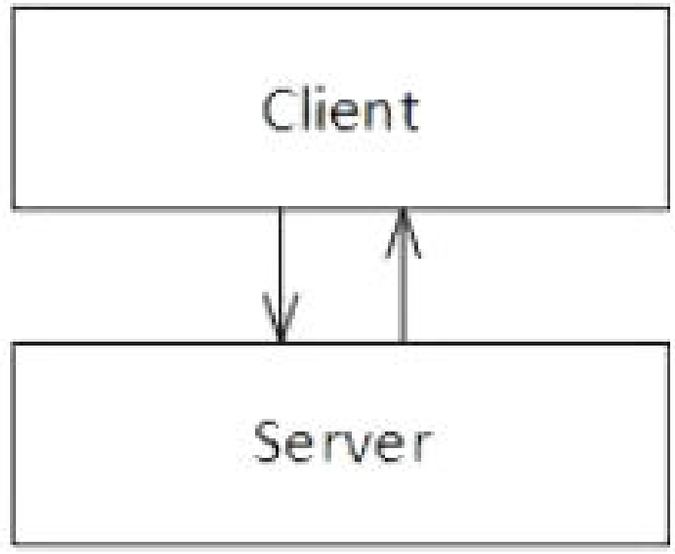


- For example, there may be business logic needed by the user interface to validate a piece of data prior to submitting the data to the server.
- The server may need this same business logic because it also needs to perform this validation.



© Virtual University of Pakistan

- While centralizing this logic may require additional communication between the client and the server, the alternative (duplication) lowers maintainability.
  - If the business logic were to change, it would have to be modified in multiple places.



© Virtual University of Pakistan

Software Design and Architecture

Topic#173

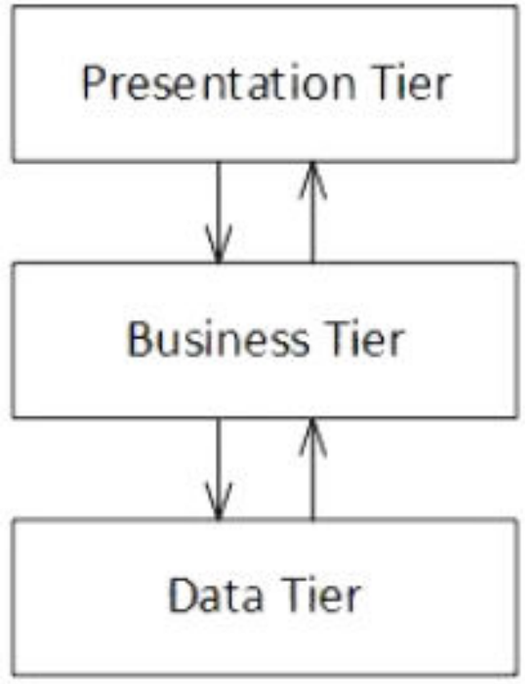
END!

Software Design and Architecture

Topic#174
Module Patterns
Layered Pattern – VII: n-tier architecture

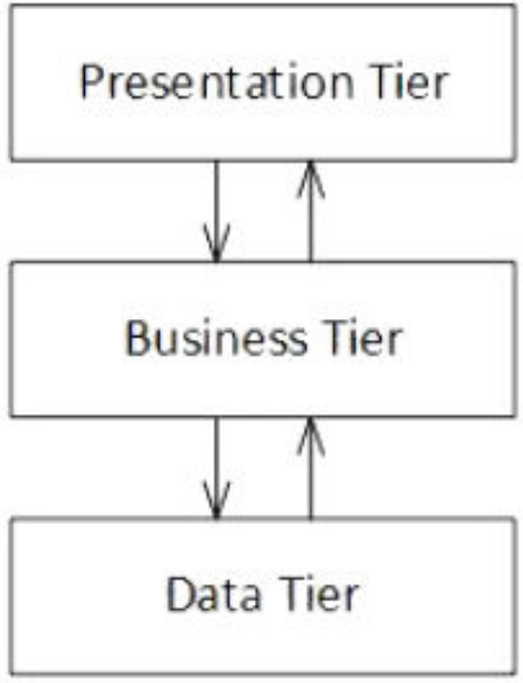
© Virtual University of Pakistan

- also known as a **multitier architecture**
- there are multiple tiers in the architecture.
- One of the most widely-used variations of this type of layered architecture is the three-tier architecture.
- The three-tier architecture separates logic into presentation, business, and data layers:



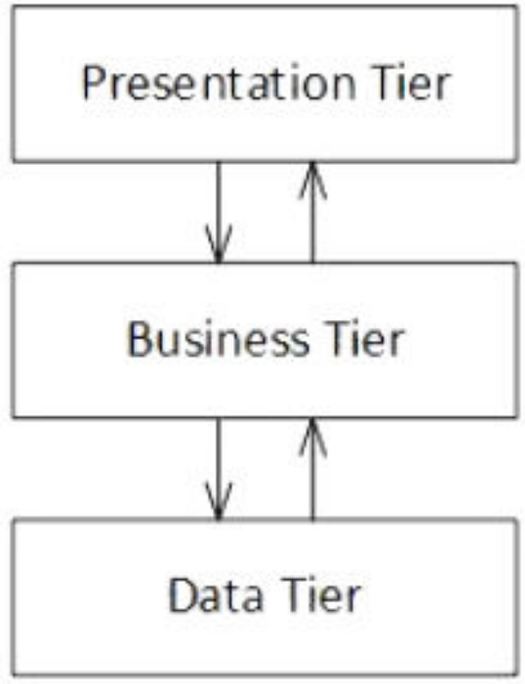
© Virtual University of Pakistan

- **Presentation tier**
- The **presentation tier** provides functionality for the application's UI.
- It should provide an appealing visual design as it is the part of the application that users interact with and see.
- Data is presented to the user and input is received from users in this tier.



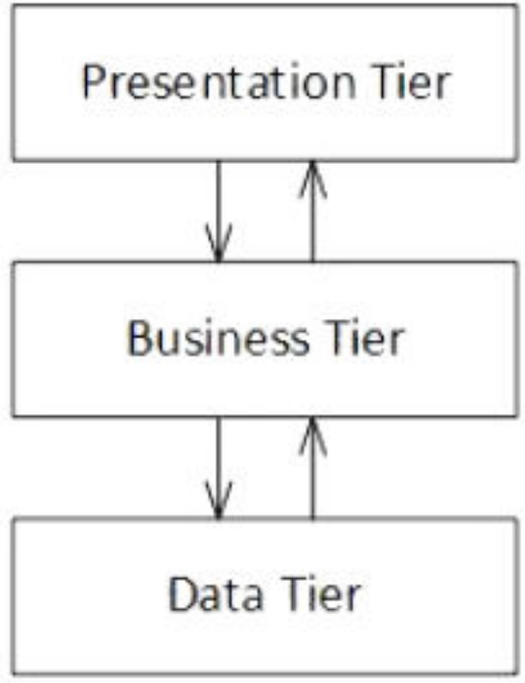
© Virtual University of Pakistan

- **Presentation tier**
- Aspects of the usability quality attribute should be the concern of the presentation tier.
- Software architects should strive to design *thin clients* that minimize the amount of logic that exists in the presentation tier.

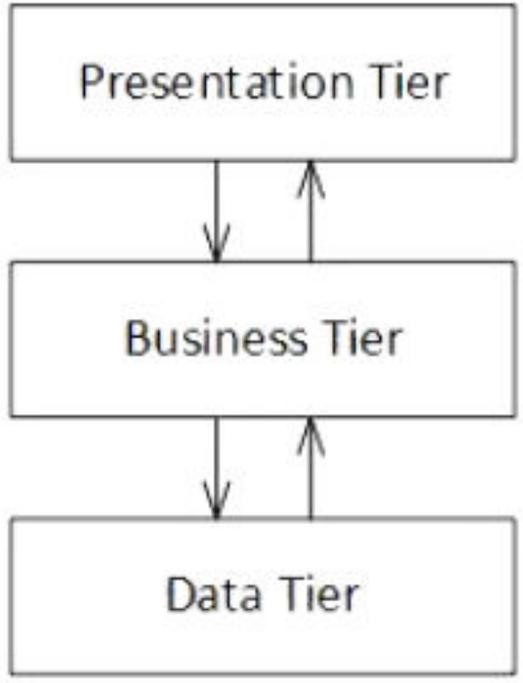


© Virtual University of Pakistan

- **Presentation tier**
- The logic in the presentation tier should focus on user interface concerns.
- A presentation tier devoid of business logic will be easier to test.

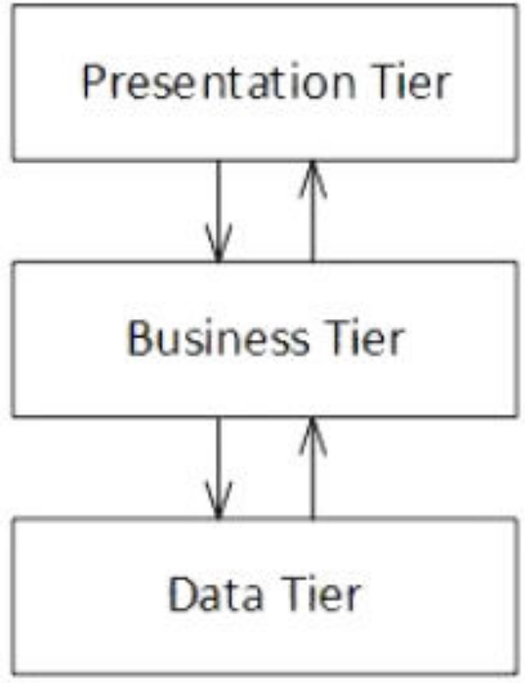


- **Business tier**
- The **business tier**, which is sometimes referred to as the application tier, provides the implementation for the business logic of the application, including such things as business rules, validations, and calculation logic.
- Business entities for the application's domain are placed in this tier.



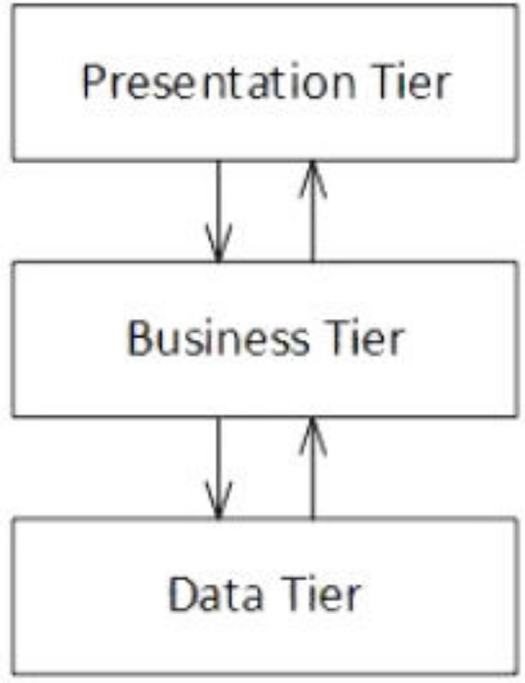
© Virtual University of Pakistan

- **Business tier**
- The business tier coordinates the application and executes logic.
- It can perform detailed processes and makes logical decisions.
- The business tier is the center of the application and serves as an intermediary between the presentation and data tiers.



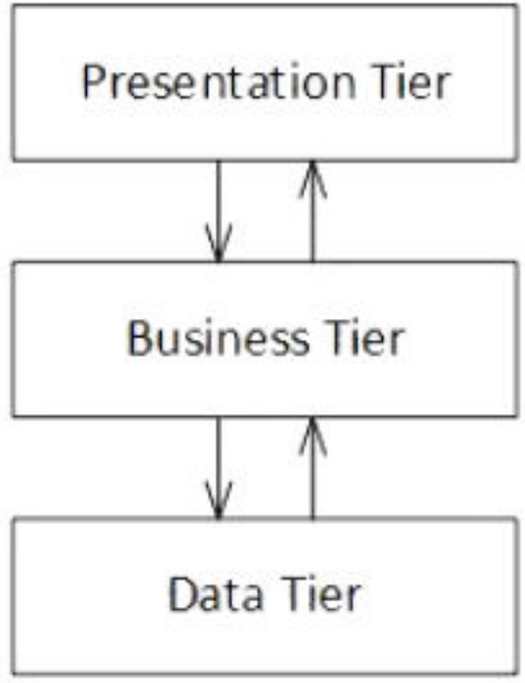
© Virtual University of Pakistan

- **Business tier**
- It provides the presentation tier with services, commands, and data that it can use, and it interacts with the data tier to retrieve and manipulate data.



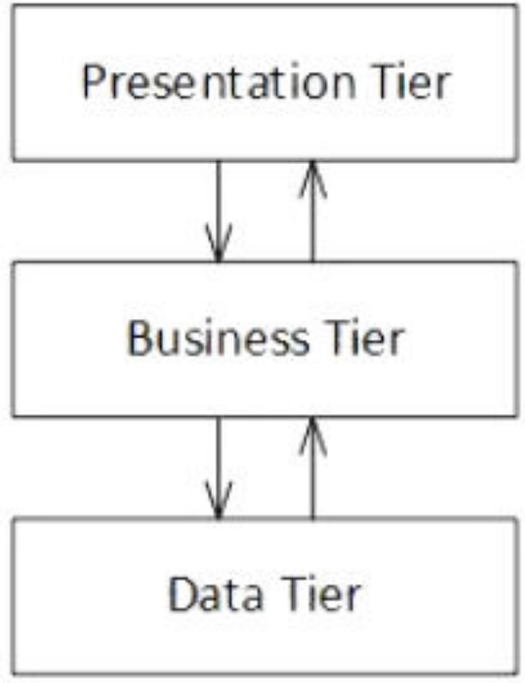
© Virtual University of Pakistan

- **Data tier**
- The **data tier** provides functionality to access and manage data.
- The data tier contains a data store for persistent storage, such as an RDBMS.
- It provides services and data for the business tier.



© Virtual University of Pakistan

- The rise of the web coincided with a shift from two-tier (client-server) architectures to three-tier architectures.
- With web applications and the use of web browsers, rich client applications containing business logic were not ideal.



© Virtual University of Pakistan

Software Design and Architecture

Topic#174

END!

Software Design and Architecture

Topic#175
Module Patterns
Variations in Layered Pattern

© Virtual University of Pakistan

Layered Architecture with “Sidecar”

- modules in A, B, or C can use modules in D
- “Sidecars” often contain common utilities such as error handlers, communication protocols, or database access mechanisms.

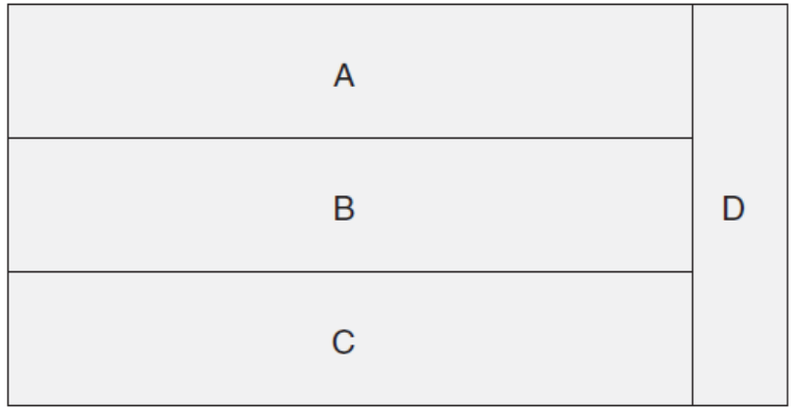


FIGURE 13.4 Layers with a “sidecar”

© Virtual University of Pakistan

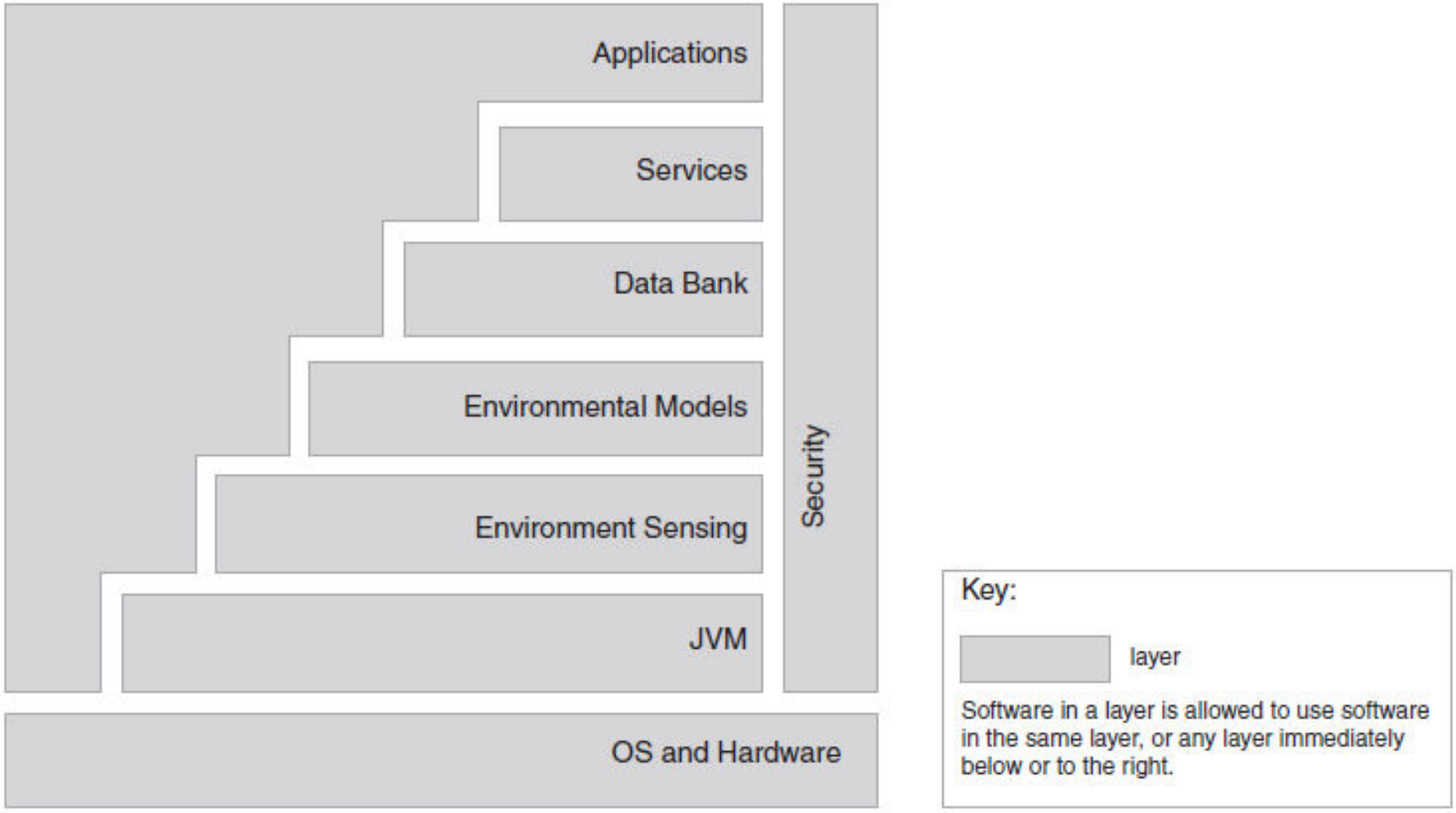


FIGURE 13.2 A simple layer diagram, with a simple key answering the uses question

© Virtual University of Pakistan

Segments within layers

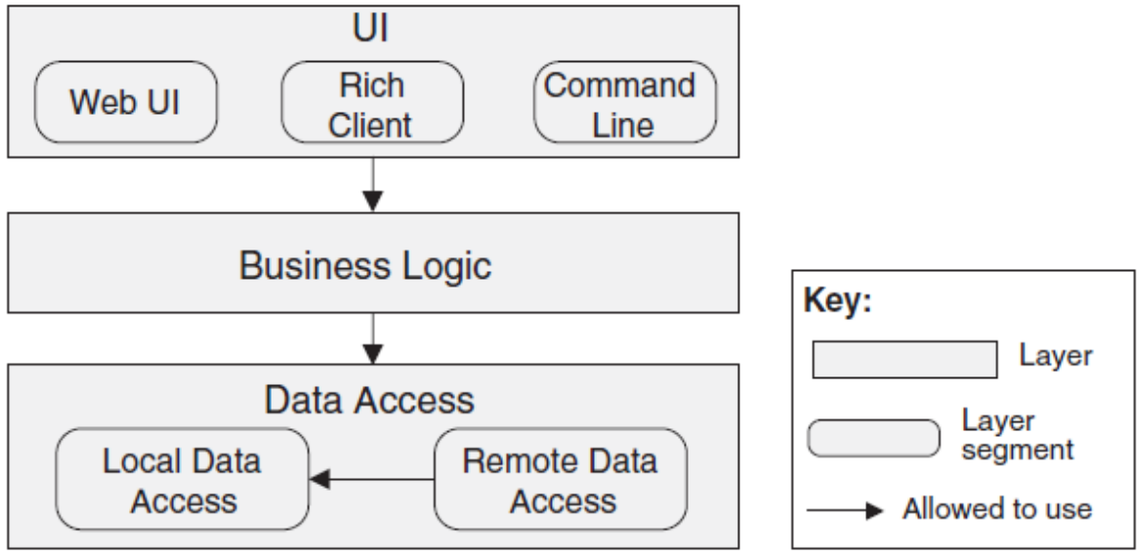


FIGURE 13.5 Layered design with segmented layers

- Sometimes layers are divided into segments denoting a finer-grained decomposition of the modules.

Segments within layers

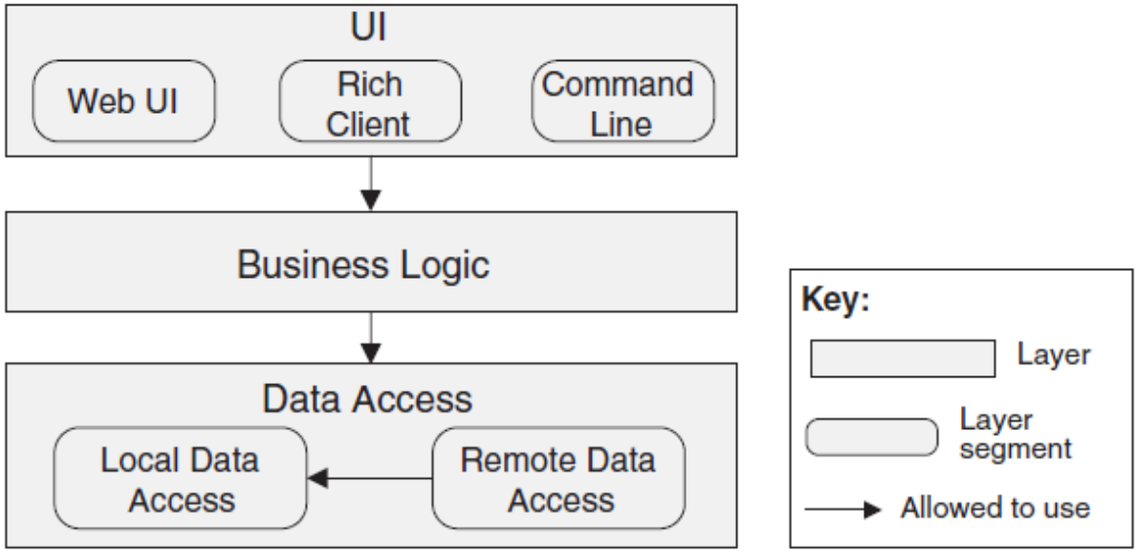


FIGURE 13.5 Layered design with segmented layers

- Segments of the top layer are not allowed to use each other, but segments of the bottom layer are.

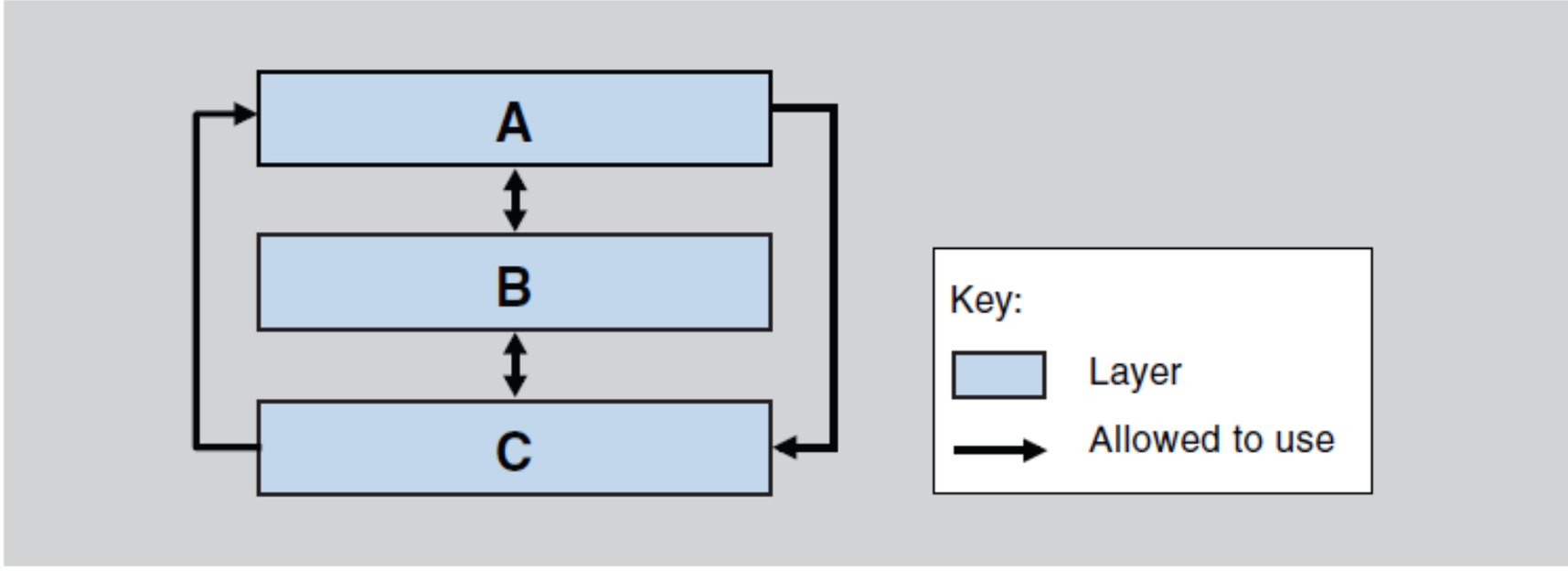


FIGURE 13.3 A wolf in layer's clothing

Software Design and Architecture

Topic#175

END!

Software Design and Architecture

Topic#176
Component-and-Connector Patterns
Broker Pattern – I: Introduction

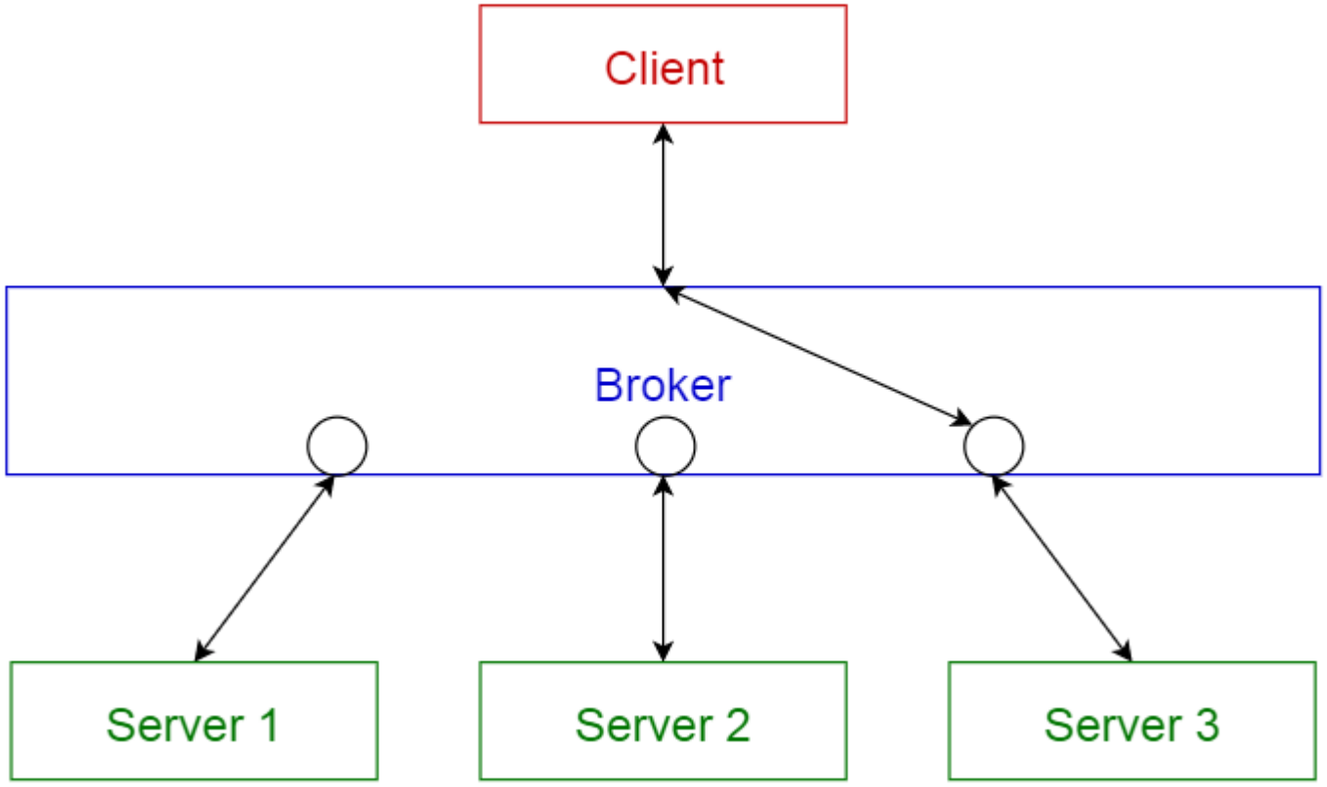
© Virtual University of Pakistan

- **Context:** Many systems are constructed from a collection of services distributed across multiple servers.
- Implementing these systems is complex because you need to worry about
  - how the systems will interoperate
  - how they will connect to each other
  - how they will exchange information
  - the availability of the component services.

- **Problem:**
- How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?

© Virtual University of Pakistan

- **Solution:**
- The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker.
- When a client needs a service, it queries a broker via a service interface.
- The broker then forwards the client's service request to a server, which processes the request.
- The service result is communicated from the server back to the broker, which then returns the result (and any exceptions) back to the requesting client.



© Virtual University of Pakistan

- The first widely used implementation of the broker pattern was in the Common Object Request Broker Architecture (CORBA).
- Other common uses of this pattern are found in Enterprise Java Beans (EJB) and Microsoft's .NET platform
- Essentially any modern platform for distributed service providers and consumers implements some form of a broker.
- The service-oriented architecture (SOA) approach depends crucially on brokers, most commonly in the form of an enterprise service bus.

© Virtual University of Pakistan

Software Design and Architecture

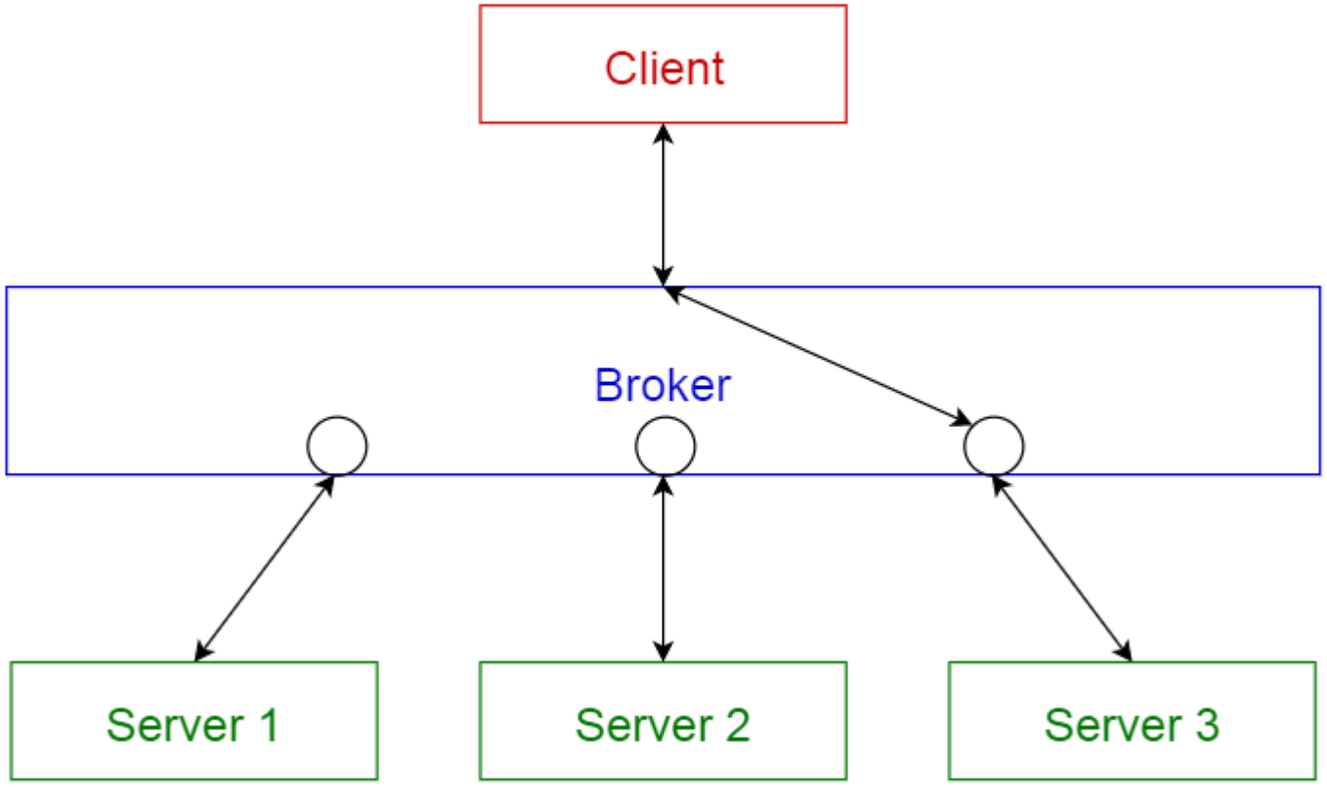
Topic#176

END!

Software Design and Architecture

Topic#177
Component-and-Connector Patterns
Broker Pattern – II: Advantages and Disadvantages

© Virtual University of Pakistan



© Virtual University of Pakistan

- the client remains completely ignorant of the identity, location, and characteristics of the server.
- Because of this separation, if a server becomes unavailable, a replacement can be dynamically chosen by the broker.
- If a server is replaced with a different (compatible) service, again, the broker is the only component that needs to know of this change, and so the client is unaffected.

- Proxies are commonly introduced as intermediaries in addition to the broker to help with details of the interaction with the broker, such as marshaling and unmarshaling messages.

© Virtual University of Pakistan

- The down sides of brokers are that they add complexity (brokers and possibly proxies must be designed and implemented, along with messaging protocols) and add a level of indirection between a client and a server, which will add latency to their communication.
- Debugging brokers can be difficult because they are involved in highly dynamic environments where the conditions leading to a failure may be difficult to replicate.

- The broker would be an obvious point of attack, from a security perspective, and so it needs to be hardened appropriately.
- Also a broker, if it is not designed carefully, can be a single point of failure for a large and complex system.
- And brokers can potentially be bottlenecks for communication.

© Virtual University of Pakistan
thegradup.com

Software Design and Architecture

Topic#177

END!

Software Design and Architecture

Topic#178
Component-and-Connector Patterns
MVC Pattern

Model-View-Controller (MVC)

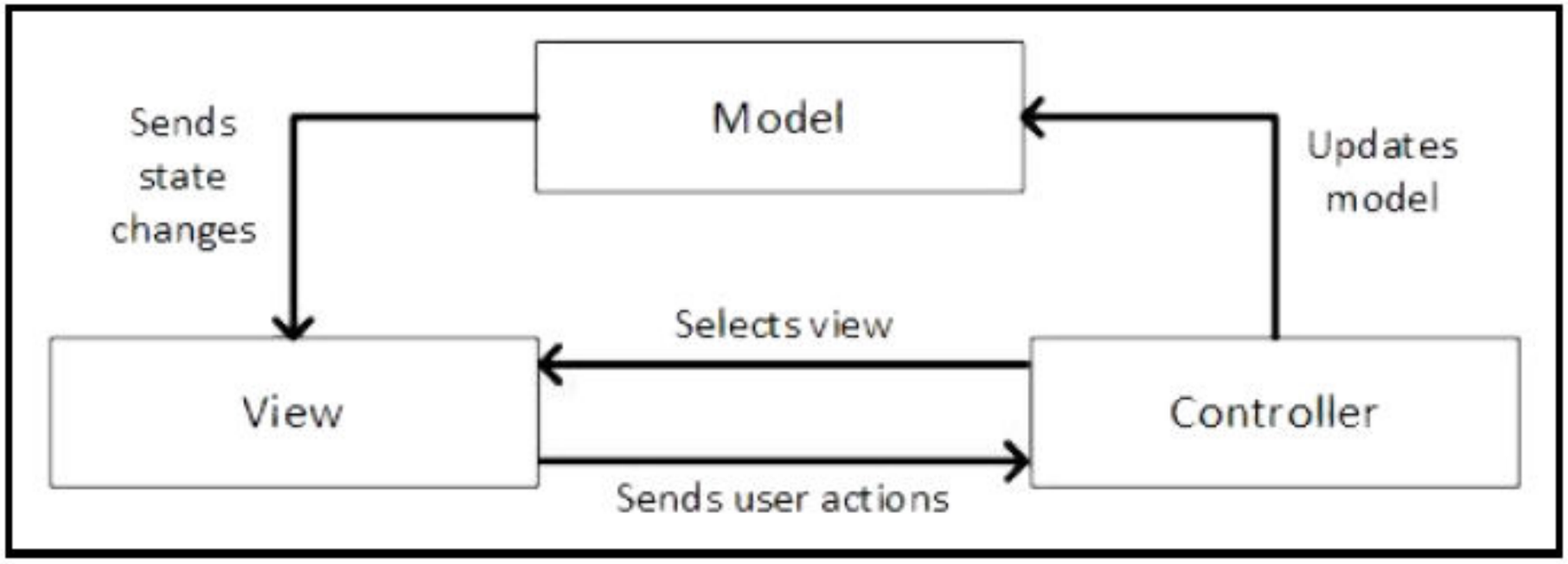
- widely used for the UI of an application.
- It is particularly well suited to web applications, although it can also be used for other types of applications, such as desktop applications.
- The pattern provides a structure for building user interfaces and provides a separation of the different responsibilities involved.

© Virtual University of Pakistan

Model-View-Controller (MVC)

- A number of popular web and application development frameworks make use of this pattern.
  - A few examples include Ruby on Rails, ASP.NET MVC, and Spring MVC.

© Virtual University of Pakistan


Model
Sends state changes
Updates model
Selects view
View
Controller
Sends user actions

- MVC is not appropriate for every situation.
  - The design and implementation of three distinct kinds of components, along with their various forms of interaction, may be costly, and this cost may not make sense for relatively simple user interfaces.

- the match between the abstractions of MVC and commercial user interface toolkits is not perfect.
  - The view and the controller split apart input and output, but these functions are often combined into individual widgets. This may result in a conceptual mismatch between the architecture and the user interface toolkit.

Software Design and Architecture

Topic#178

END!

Software Design and Architecture

Topic#179
Component-and-Connector Patterns
Pipe-and-Filter Pattern

© Virtual University of Pakistan

- **Context:** Many systems are required to transform streams of discrete data items, from input to output.
- Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

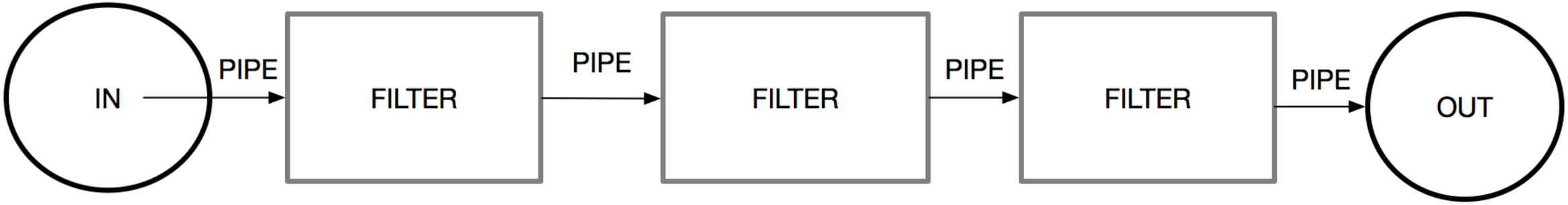
© Virtual University of Pakistan

- **Problem:** Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms.
- In this way they can be flexibly combined with each other.
- The components, being generic and loosely coupled, are easily reused.
- The components, being independent, can execute in parallel.

© Virtual University of Pakistan

- **Solution:** The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data.
- Data arrives at a filter's input port(s), is transformed, and then is passed via its output port(s) through a pipe to the next filter.
- A single filter can consume data from, or produce data to, one or more ports.

© Virtual University of Pakistan



© Virtual University of Pakistan

Software Design and Architecture

Topic#179

END!

Software Design and Architecture

Topic#180
Component-and-Connector Patterns
Pipe-and-Filter Pattern – Strengths and Weaknesses

© Virtual University of Pakistan

- typically not a good choice for an interactive system, as it disallows cycles (which are important for user feedback).
- having large numbers of independent filters can add substantial amounts of computational overhead, because each filter runs as its own thread or process.

© Virtual University of Pakistan
thegradup.com

- may not be appropriate for long-running computations, without the addition of some form of checkpoint/restore functionality, as the failure of any filter (or pipe) can cause the entire pipeline to fail.

© Virtual University of Pakistan

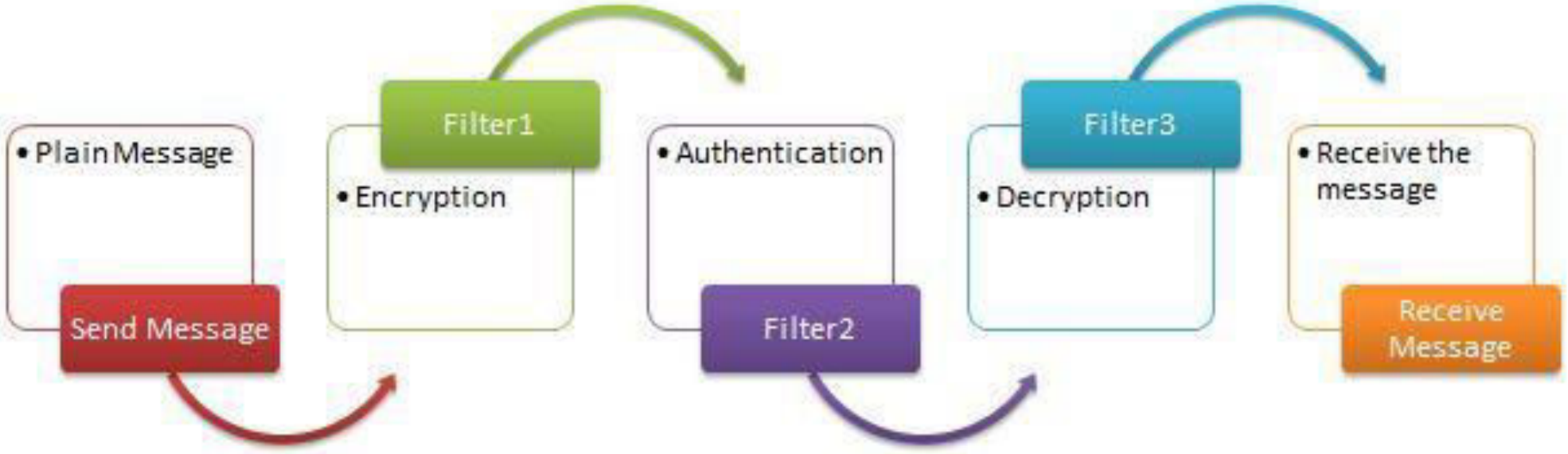
- Pipes buffer data during communication. Because of this property, filters can execute asynchronously and concurrently.

© Virtual University of Pakistan

- a filter typically does not know the identity of its upstream or downstream filters. For this reason, pipeline pipe-and-filter systems have the property that the overall computation can be treated as the functional composition of the computations of the filters, making it easier for the architect to reason about end-to-end behavior.

- Data transformation systems are typically structured as pipes and filters, with each filter responsible for one part of the overall transformation of the input data.
- The independent processing at each step supports reuse, parallelization, and simplified reasoning about overall behavior.

© Virtual University of Pakistan


• Plain Message
Send Message
Filter1
• Encryption
• Authentication
Filter2
Filter3
• Decryption
• Receive the message
Receive Message

Examples of Pipe-and-Filter Architecture

- systems built using UNIX pipes
- the request processing architecture of the Apache web server
- Yahoo! Pipes for processing RSS feeds
- many workflow engines
- many scientific computation systems that have to process and analyze large streams of captured data.

Software Design and Architecture

Topic#180

END!

Software Design and Architecture

Topic#181
Component-and-Connector Patterns
Client-Server Pattern

© Virtual University of Pakistan

- **Context:** There are shared resources and services that large numbers of distributed clients wish to access, and for which we wish to control access or quality of service.

- **Problem:**
  - By managing a set of shared resources and services, we can promote modifiability and reuse, by factoring out common services and having to modify these in a single location, or a small number of locations.
  - We want to improve scalability and availability by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers.

- **Solution:**
  - Clients interact by requesting services of servers, which provide a set of services. Some components may act as both clients and servers.
  - There may be one central server or multiple distributed ones.

- Components
  - *clients* and *servers*
- Connectors
  - a data connector driven by a request/reply protocol used for invoking services.

© Virtual University of Pakistan

Disadvantages

- The server can be a performance bottleneck and it can be a single point of failure.
- Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.

© Virtual University of Pakistan

Software Design and Architecture

Topic#181

END!

Software Design and Architecture

Topic#182
Component-and-Connector Patterns
Shared-Data Pattern

- **Context:**
  - Various computational components need to share and manipulate large amounts of data.
  - This data does not belong solely to any one of those components.
- **Problem:**
  - How can systems store and manipulate persistent data that is accessed by multiple independent components?

© Virtual University of Pakistan

- **Solution:**
  - Interaction is dominated by the exchange of persistent data between multiple *data accessors* and at least one *shared-data store*.
  - Exchange may be initiated by the accessors or the data store.
  - The connector type is *data reading and writing*.

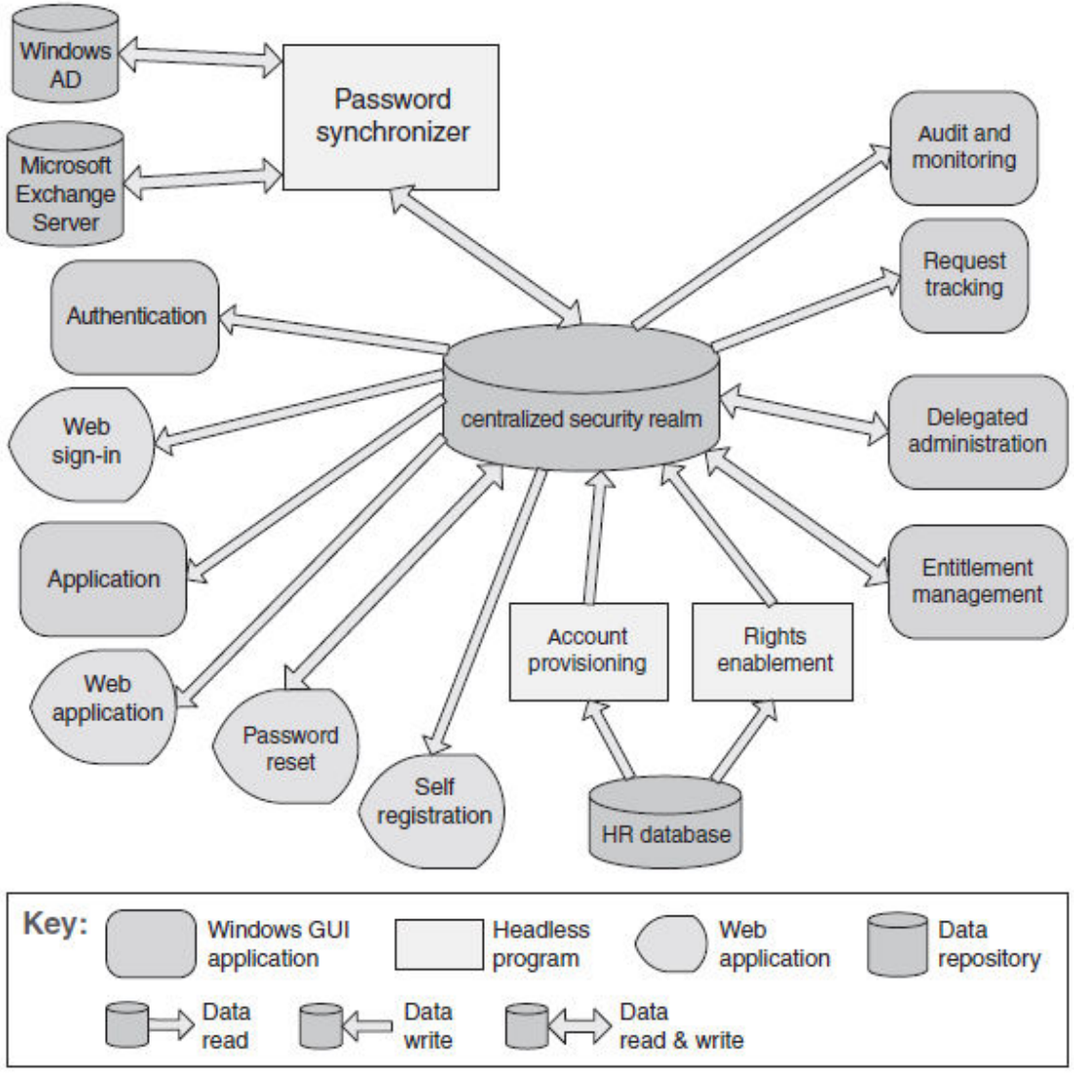


FIGURE 13.13 The shared-data diagram of an enterprise access management system

Software Design and Architecture

Topic#182

END!

Software Design and Architecture

Topic#183
Component-and-Connector Patterns
Shared-Data Pattern – Strengths and Weaknesses

© Virtual University of Pakistan

Strengths

- The shared-data pattern is useful whenever various data items are persistent and have multiple accessors.
- Use of this pattern has the effect of decoupling the producer of the data from the consumers of the data; hence, this pattern supports modifiability, as the producers do not have direct knowledge of the consumers.

Strengths

- Consolidating the data in one or more locations and accessing it in a common fashion facilitates performance tuning.

© Virtual University of Pakistan

- Analyses associated with this pattern usually center on qualities such as
  - data consistency,
  - performance,
  - security,
  - privacy,
  - availability,
  - scalability, and
  - compatibility with, for example, existing repositories and their data.

© Virtual University of Pakistan

Weaknesses

- the shared-data store may be a performance bottleneck.
  - For this reason, performance optimization has been a common theme in database research.
- The shared-data store is also potentially a single point of failure.

Weaknesses

- the producers and consumers of the shared data may be tightly coupled, through their knowledge of the structure of the shared data.

© Virtual University of Pakistan

Software Design and Architecture

Topic#183

END!

Software Design and Architecture

Topic#184
Allocation Patterns - Multi-tier Pattern

© Virtual University of Pakistan

Multi-tier Pattern

- The multi-tier pattern is a C&C pattern or an allocation pattern, depending on the criteria used to define the tiers.

© Virtual University of Pakistan

Multi-tier Pattern

- Tiers can be created to group components of similar functionality, in which case it is a C&C pattern.

Multi-tier Pattern

- However, in many, if not most, cases tiers are defined with an eye toward the computing environment on which the software will run:
  - A client tier in an enterprise system will not be running on the computer that hosts the database.

Multi-tier Pattern

- That makes it an allocation pattern
  - mapping software elements—perhaps produced by applying C&C patterns—to computing elements.

© Virtual University of Pakistan

Software Design and Architecture

Topic#184

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 185 – 186

Monolithic Architecture

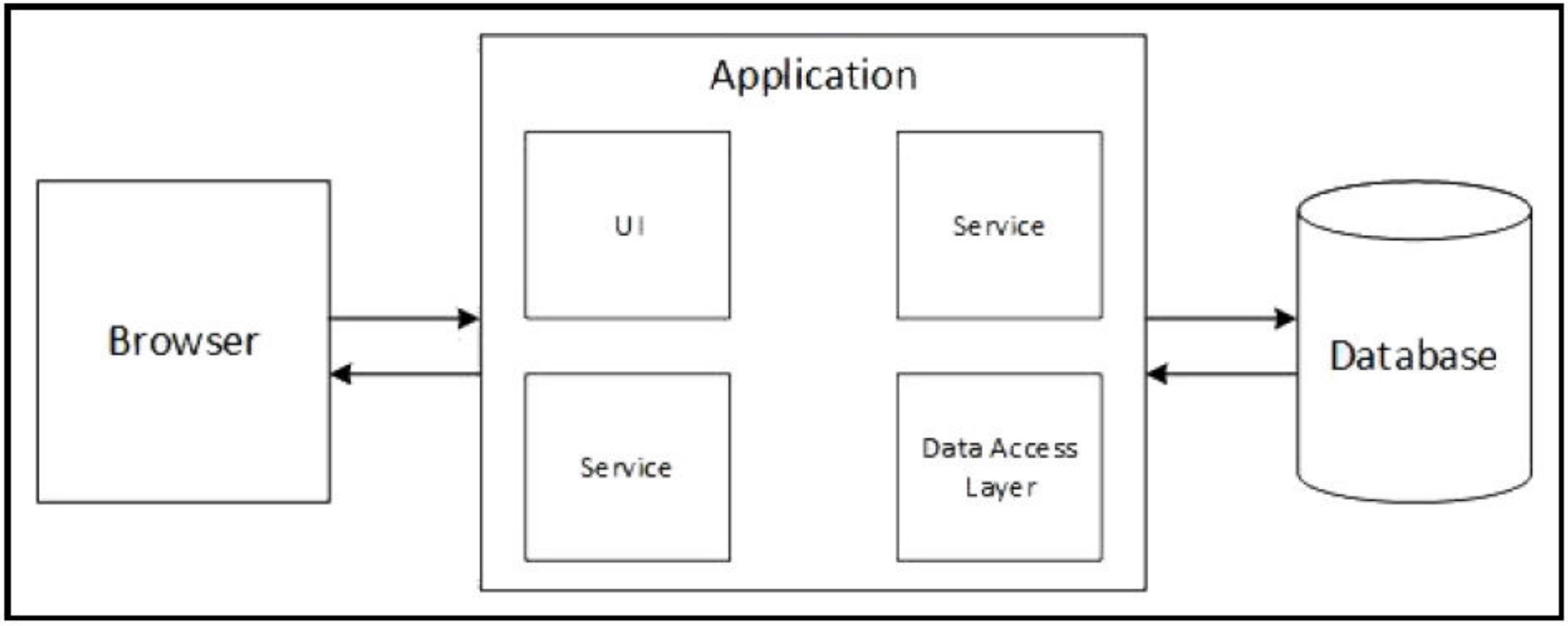
© Virtual University of Pakistan

Software Design and Architecture

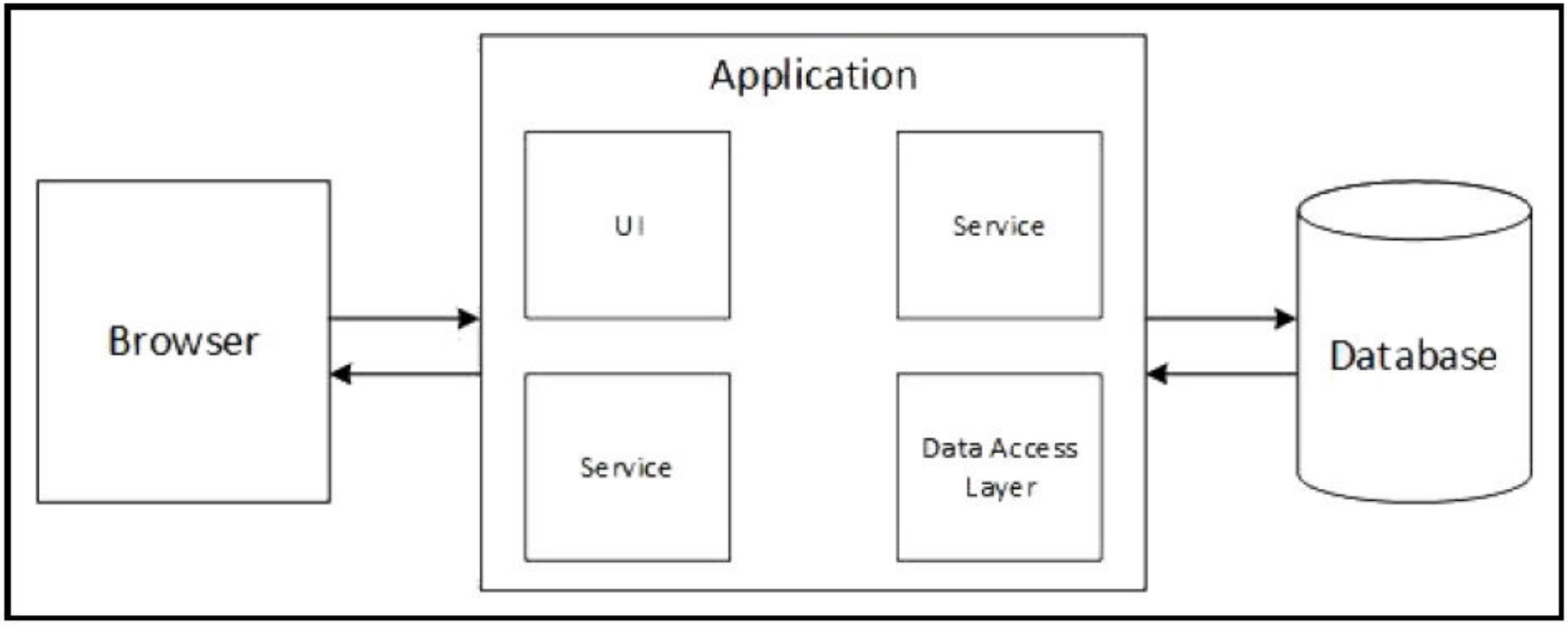
Topic#185
Monolithic Architecture

© Virtual University of Pakistan

- A **monolithic architecture** is one in which a software application is designed to work as a single, self-contained unit.
- Applications that have this type of architecture are common.
- The components within a monolithic architecture are interconnected and interdependent, resulting in tightly coupled code.



- The different concerns of an application, such as user interface, business logic, authorization, logging, and database access, are not kept separate in a monolithic architecture.



- These different pieces of functionality are intertwined in a monolithic application.

Software Design and Architecture

Topic#185

END!

Software Design and Architecture

Topic#186
Monolithic Architecture – Strengths and Weaknesses

© Virtual University of Pakistan

Benefits

- Applications with a monolithic architecture typically have better performance.
- Small applications that have this type of architecture are easier to deploy because of the simplicity of the high-level architecture.

© Virtual University of Pakistan

- In spite of the tightly coupled logic, monolithic applications can be easier to test and debug because they are simpler, with fewer separate components to consider.

- Monolithic applications are typically easy to scale because all it takes is to run multiple instances of the same application.
  - However, different application components have different scaling needs and we cannot scale the components independently with a monolithic architecture.
  - We are limited to adding more instances of the entire application in order to scale.

© Virtual University of Pakistan

Drawbacks

- Monolithic applications greatly inhibit the agility of the organization as it becomes difficult to make changes to the software.
- continuous deployment is difficult to achieve.
  - Even if a change is made to only one component of a monolithic application, the entire software system will need to be deployed.

- Organizations are required to devote more resources, such as time and testers, to deploy a new version of a monolithic application.

- Low maintainability
  - Tightly coupled components make it more difficult to make changes because a change in one part of the application is more likely to affect other parts of the application.
  - Difficult to understand

© Virtual University of Pakistan

- It also takes longer for such applications to start up, lowering the productivity of the team during development.

- Monolithic applications require a commitment to a particular programming language and technology stack.
  - If a migration to a different technology is needed, it requires the organization to commit to rewriting the entire application.

© Virtual University of Pakistan

Software Design and Architecture

Topic#186

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 187 – 212

Web Application Archetype

© Virtual University of Pakistan

Software Design and Architecture

Topic#187
Web Application Architecture Overview

- Why Web Application
  - Almost 23 million software developers (2018)
  - Over 15 million of these actively develop software for the web or related technologies

Web Application Overview

- The core of a Web application is its server-side logic.
- The Web application layer itself can be comprised of many distinct layers.
- The typical example is a three-layered architecture comprised of presentation, business, and data layers.

© Virtual University of Pakistan

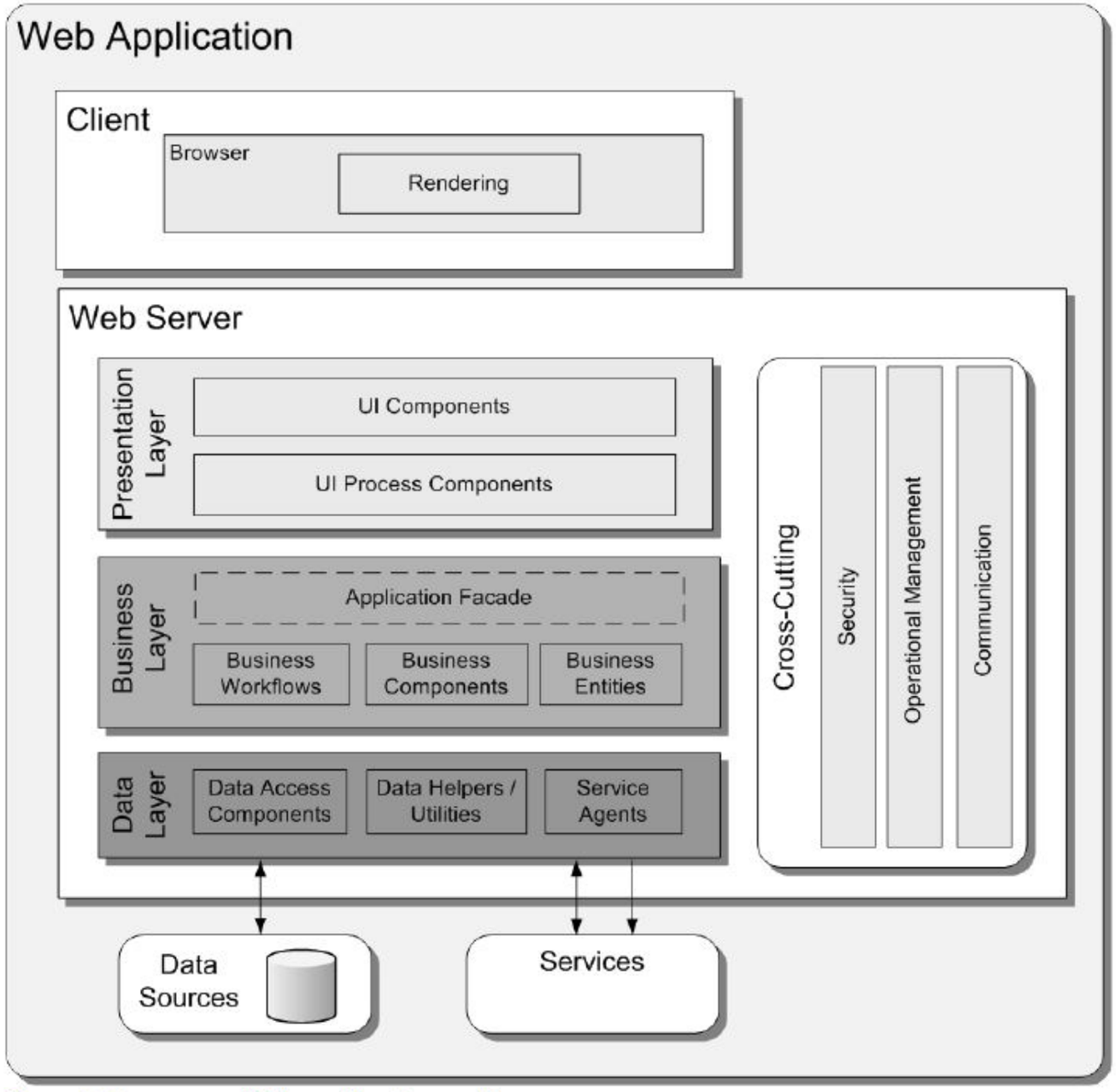


Figure 1 A common Web application architecture

Figure illustrates a common Web application architecture with common components grouped by different areas of concern

Design Considerations

- **When designing a Web application, the goals of a software architect are to minimize the complexity by separating tasks into different areas of concern while designing a secure, high performance application.**

Software Design and Architecture

Topic#187

END!

Software Design and Architecture

Topic#188
Web Application Design Considerations
Part I - Maintainability

© Virtual University of Pakistan

- **Partition your application logically**
  - Use layering to partition your application logically into presentation, business, and data access layers.
  - helps to create maintainable code
  - allows to monitor and optimize the performance of each layer separately.
  - offers more choices for scaling your application.

- **Use abstraction to implement loose coupling between layers.**
  - This can be accomplished by defining interface components, such as a façade (or interface types/abstract base classes) with well-known inputs and outputs that translates requests into a format understood by components within the layer.

© Virtual University of Pakistan
thegradup.com

- **Understand how components will communicate with each other**
  - This requires an understanding of the deployment scenarios your application must support.
  - You must determine if communication across physical boundaries or process boundaries should be supported, or if all components will run within the same process.

© Virtual University of Pakistan

Software Design and Architecture

Topic#188

END!

Software Design and Architecture

Topic#189
Web Application Design Considerations
Part II - Performance

© Virtual University of Pakistan

- **Reduce round trips.**
  - When designing a Web application, consider using techniques such as caching and output buffering to reduce round trips between the browser and the Web server, and between the Web server and downstream servers.

- **Consider using caching**
  - A well-designed caching strategy is probably the single most important performance-related design consideration.

- **Avoid blocking during long-running tasks**
  - If you have long-running or blocking operations, consider using an asynchronous approach to allow the Web server to process other incoming requests.

© Virtual University of Pakistan

Software Design and Architecture

Topic#189

END!

Software Design and Architecture

Topic#190
Web Application Design Considerations
Part III - Security

© Virtual University of Pakistan

- **Consider using logging and instrumentation**
  - You should audit and log activities across the layers and tiers of your application.
  - These logs can be used to detect suspicious activity, which frequently provides early indications of an attack on the system.

- **Consider authenticating users across trust boundaries**
  - You should design your application to authenticate users whenever they cross a trust boundary; for example, when accessing a remote business layer from your presentation layer.

© Virtual University of Pakistan

- **Do not pass sensitive data in plain text across the network.**
  - Whenever you need to pass sensitive data such as a password or authentication cookie across the network, consider encrypting and signing the data or using Secure Sockets Layer (SSL) encryption.

© Virtual University of Pakistan
thegradup.com

- **Design your Web application to run using a least-privileged account**
  - If an attacker manages to take control of a process, the process identity should have restricted access to the file system and other system resources in order to limit the possible damage.

© Virtual University of Pakistan

Software Design and Architecture

Topic#190

END!

Software Design and Architecture

Topic#191
Web Application Frame - Introduction

© Virtual University of Pakistan

Web Application Frame

- There are several common issues that you must consider as you develop your design.
- These issues can be categorized into specific areas of the design.
  - the common issues for each category where mistakes are most often made.

© Virtual University of Pakistan

Software Design and Architecture

Topic#191

END!

Software Design and Architecture

Topic#192
Web Application Frame - Authentication

- Designing an effective authentication strategy is important for the security and reliability of your application.
  - Improper or weak authorization can leave your application vulnerable to spoofing attacks, dictionary attacks, session hijacking, and other types of attack.

© Virtual University of Pakistan

Common Mistakes

- Lack of authentication across trust boundaries
- Storing passwords in a database as plain text
- Designing custom authentication mechanism instead of using built-in capabilities

© Virtual University of Pakistan

Guidelines

- Identify trust boundaries within Web application layers.
- This will help you to determine where to authenticate.
- Use a platform-supported authentication mechanism when possible.
- Enforce strong account management practices such as account lockouts and expirations.
- Enforce strong password policies. This includes specifying password length and complexity, and password expiration policies.

© Virtual University of Pakistan

Software Design and Architecture

Topic#192

END!

Software Design and Architecture

Topic#193
Web Application Frame - Authorization

© Virtual University of Pakistan

- Authorization determines the tasks that an authenticated identity can perform, and identifies the resources that can be accessed. Designing an effective authorization strategy is important for the security and reliability of your application.
- Improper or weak authorization leads to information disclosure, data tampering, and elevation of privileges.
- Defense in depth is the key security principle to apply to your application's authorization strategy.

Common Mistakes

- Lack of authorization across trust boundaries
- Incorrect role granularity
- Using impersonation and delegation when not required

© Virtual University of Pakistan

Guidelines

- Identify trust boundaries within the Web application layers and authorize users across trust boundaries.
- Consider the granularity of your authorization settings.
  - Building your authorization with too much granularity will increase your management overhead; however, using less granularity will reduce flexibility.

© Virtual University of Pakistan

Guidelines

- Access downstream resources using a trusted identity based on the trusted subsystem model.
  - but consider the effect on performance and scalability.

© Virtual University of Pakistan

Software Design and Architecture

Topic#193

END!

Software Design and Architecture

Topic#194
Web Application Frame - Caching

© Virtual University of Pakistan

- Caching improves the performance and responsiveness of your application.
- However, incorrect caching choices and poor caching design can degrade performance, security and responsiveness.
- You should use caching to optimize reference data lookups, avoid network round trips, and avoid unnecessary and duplicate processing.
- To implement caching, you must first decide when to load data into the cache.
- Try to load cache data asynchronously or by using a batch process to avoid client delays.

Common Mistakes

- Caching volatile data
- Not considering caching page output
- Caching sensitive data
- Failing to cache data in a ready-to-use format

Guidelines

- Avoid caching volatile data.
- Use output caching to cache pages that are relatively static.
- Consider using partial page caching through user controls for static data in your pages.
- Pool shared resources that are expensive, such as network connections, instead of caching them.
- Cache data in a ready-to-use format.

© Virtual University of Pakistan

Software Design and Architecture

Topic#194

END!

Software Design and Architecture

Topic#195
Web Application Frame – Exception Management

© Virtual University of Pakistan

- Designing an effective exception management strategy is important for the security and reliability of your application.
- Correct exception handling in your Web pages prevents sensitive exception details from being revealed to the user, improves application robustness, and helps to avoid leaving your application in an inconsistent state in the event of an error.

Guidelines

- Do not use exceptions to control the logical flow of your application.
- Do not catch exceptions unless you must handle them, you need to strip sensitive information, or you need to add additional information to the exception.
- Design a global error handler to catch unhandled exceptions.

Guidelines

- Display user-friendly messages to end users whenever an error or exception occurs.
- Log exception related information in an error file
- Do not reveal sensitive information, such as passwords, through exception details.

© Virtual University of Pakistan

Software Design and Architecture

Topic#195

END!

Software Design and Architecture

Topic#196
Web Application Frame – Logging and Instrumentation

© Virtual University of Pakistan

- Designing an effective logging and instrumentation strategy is important for the security and reliability of your application.
- You should audit and log activity across the tiers of your application.
- These logs can be used to detect suspicious activity, which frequently provides early indications of an attack on the system, and help to address the repudiation threat where users deny their actions.
- Log files may be required in legal proceedings to prove the wrongdoing of individuals.
- Generally, auditing is considered most authoritative if the audits are generated at the precise time of resource access and by the same routines that access the resource.

Guidelines

- Consider auditing for user management events.
- Consider auditing for unusual activities.
- Consider auditing for business-critical operations.
- Create secure log file management policies, such as restricting the access to log files, allowing only write access to users, etc.
- Do not store sensitive information in the log or audit files.

© Virtual University of Pakistan

Software Design and Architecture

Topic#196

END!

Software Design and Architecture

Topic#197
Web Application Frame – Navigation

© Virtual University of Pakistan

- Design your navigation strategy in a way that separates it from the processing logic.
- Your strategy should allow users to navigate easily through your screens or pages.
- Designing a consistent navigation structure for your application will help to minimize user confusion as well as reduce the apparent complexity of the application.

Guidelines

- Use well-known design patterns, such as Model-View-Controller (MVC), to decouple UI processing from output rendering.
- Consider encapsulating navigation in a master page so that it is consistent across pages.

Guidelines

- Design a site map to help users find pages on the site, and to allow search engines to crawl the site if desired.
- Consider using wizards to implement navigation between forms in a predictable way.

© Virtual University of Pakistan

Guidelines

- Consider using visual elements such as embedded links, navigation menus, and breadcrumb navigation in the UI to help users understand where they are, what is available on the site, and how to navigate the site quickly.

© Virtual University of Pakistan

Software Design and Architecture

Topic#197

END!

Software Design and Architecture

Topic#198
Web Application Frame – Page Layout

- Design your application so that the page layout can be separated from the specific UI components and UI processing.
- When choosing a layout strategy, consider whether designers or developers will be building the layout.
- If designers will be building the layout, choose a layout approach that does not require coding or the use of development-focused tools.

Guidelines

- Use Cascading Style Sheets (CSS) for layout whenever possible.
- Use table-based layout when you need to support a grid layout, but remember that table-based layout can be slow to render, does not have full cross-browser support, and there may be issues with complex layout.

© Virtual University of Pakistan

Guidelines

- Use a common layout for pages where possible to maximize accessibility and ease of use.
- Avoid designing and developing large pages that accomplish multiple tasks, particularly where only a few tasks are usually executed with each request.

© Virtual University of Pakistan

Software Design and Architecture

Topic#198

END!

Software Design and Architecture

Topic#199
Web Application Frame – Page Rendering

© Virtual University of Pakistan

- When designing for page rendering, you must ensure that you render the pages efficiently and maximize interface usability.

Guidelines

- Consider data-binding options.
  - For example, you can bind custom objects or datasets to controls.
- Consider using Asynchronous JavaScript and XML (AJAX) for an improved user experience and better responsiveness.
- Consider using data-paging techniques for large amounts of data to minimize scalability issues.

Guidelines

- Consider designing to support localization in UI components.
- Abstract the user process components from data rendering and acquisition functions.

© Virtual University of Pakistan

Software Design and Architecture

Topic#199

END!

Software Design and Architecture

Topic#200
Web Application Frame – Presentation Entity

© Virtual University of Pakistan

- Presentation entities store the data that you will use to manage the views in your presentation layer.
- Presentation entities are not always necessary.
- Consider using presentation entities only if the datasets are sufficiently large or complex that they must be stored separately from the UI controls.
- Design or choose appropriate presentation entities that you can easily bind to UI controls.

Guidelines

- Determine if you need presentation entities.
  - Typically, you might need presentation entities if the data or data format to be displayed is specific to the presentation layer.
- Consider the serialization requirements for your presentation entities, if they are to be passed across the network or stored on the disk.

Guidelines

- Consider implementing data type validation in the property setters of your presentation entities.
- Consider using presentation entities to store state related to the UI.
  - If you want to use this state to help your application recover from a crash, make sure after recovery that the user interface is in a consistent state.

Software Design and Architecture

Topic#200

END!

Software Design and Architecture

Topic#201
Web Application Frame – Request Processing

© Virtual University of Pakistan

- When designing a request-processing strategy, you should ensure separation of concerns by implementing the request-processing logic separately from the UI.

Guidelines

- Consider centralizing the common pre-processing and post-processing steps of Web page requests to promote logic reuse across pages.
- Consider dividing UI processing into three distinct roles—model, view, and controller/presenter—by using the Model-View-Controller (MVC) or Model-View-Presenter (MVP) pattern.

© Virtual University of Pakistan

Guidelines

- If you are designing views for handling large amounts of data, consider giving access to the model from the view by using the Supervising Controller pattern, which is a form of the MVP pattern.

Software Design and Architecture

Topic#201

END!

Software Design and Architecture

Topic#202
Web Application Frame – Session Management

© Virtual University of Pakistan

- When designing a Web application, an efficient and secure session-management strategy is important for performance, security and reliability.
- You must consider session-management factors such as what to store, where to store it, and how long information will be kept.

© Virtual University of Pakistan
thegradup.com

Guidelines

- If you have a single Web server, require optimum session state performance, and have a relatively limited number of concurrent sessions, use the in-process state store.
- If you have a single Web server, your sessions are expensive to rebuild, and you require durability in the event of a restart, use the session state service running on the local Web server.

Guidelines

- If you are storing state on a separate server, protect your session state communication channel.
- For security reasons, use session expiry (timeout)
- Prefer basic types for session data to reduce serialization costs.

© Virtual University of Pakistan

Software Design and Architecture

Topic#202

END!

Software Design and Architecture

Topic#203
Web Application Frame – Validation

- Designing an effective validation solution is important for the security and reliability of your application.
- Improper or weak authorization can leave your application vulnerable to cross-site scripting attacks, SQL injection attacks, buffer overflows, and other types of input attack.

Guidelines

- Identify trust boundaries within Web application layers, and validate all data crossing these boundaries.
- Assume that all client-controlled data is malicious and needs to be validated.
- Design your validation strategy to constrain, reject, and sanitize malicious input.

© Virtual University of Pakistan

Guidelines

- Design to validate input for length, range, format, and type.
- Use client-side validation for user experience, and server-side validation for security.

© Virtual University of Pakistan

Software Design and Architecture

Topic#203

END!

Software Design and Architecture

Topic#204
Web Application Frame
Presentation Layer Considerations

© Virtual University of Pakistan

- The presentation layer of your Web application displays the UI and facilitates user interaction.
- The design should focus on separation of concerns, where the user interaction logic is decoupled from the UI components.

© Virtual University of Pakistan

Guidelines

- Consider separating the UI components from the UI process components.
- Use client-side validation to improve user experience and responsiveness, and server-side validation for security.
  - Do not rely on just client-side validation.

© Virtual University of Pakistan

Guidelines

- Use page output caching or fragment caching to cache static pages or parts of pages.

Guidelines

- Use Web server controls if you need to compile these controls into an assembly for reuse across applications, or if you need to add additional features to existing server controls.

Guidelines

- Use Web user controls if you need to reuse UI fragments on several pages, or if you want to cache a specific parts of the page.

Software Design and Architecture

Topic#204

END!

Software Design and Architecture

Topic#205
Web Application Frame
Business Layer Considerations

- When designing the business layer for your Web application, consider how to implement the business logic and long-running workflows.
- Design business entities that represent the real world data, and use these to pass data between components.

© Virtual University of Pakistan

Guidelines

- Design a separate business layer that implements the business logic and workflows.
  - This improves the maintainability and testability of your application.
- Consider centralizing and reusing common business logic functions.

© Virtual University of Pakistan

Guidelines

- Design your business layer to be stateless.
  - This helps to reduce resource contention and increase performance.
- Use a message-based interface for the business layer.
  - This works well with a stateless Web application business layer.

Guidelines

- Design transactions for business-critical operations.

Software Design and Architecture

Topic#205

END!

Software Design and Architecture

Topic#206
Web Application Frame
Data Layer Considerations

© Virtual University of Pakistan

- Design a data layer for your Web application to abstract the logic necessary to access the database.
- Using a separate data layer makes the application easier to configure and maintain.
- The data layer may also need to access external services using service agents.

Guidelines

- Design a separate data layer to hide the details of the database from other layers of the application.
- Design entity objects to interact with other layers, and to pass the data between them.
- Design to take advantage of connection pooling to minimize the number of open connections.

© Virtual University of Pakistan

Software Design and Architecture

Topic#206

END!

Software Design and Architecture

Topic#207
Web Application Frame
Service Layer Considerations

© Virtual University of Pakistan

- Consider designing a separate service layer if you plan to deploy your business layer on a remote tier, or if you plan to expose your business logic using a Web service.

© Virtual University of Pakistan
thegradup.com

Guidelines

- If your business layer is on a remote tier, design coarse-grained service methods to minimize the number of client-server interactions, and to provide loose coupling.
- Design the services without assuming a specific client type.
- Design the services to be idempotent, assuming that the same message request may arrive multiple times.

Software Design and Architecture

Topic#207

END!

Software Design and Architecture

Topic#208
Web Application Frame
Testing and Testability Considerations

© Virtual University of Pakistan

- *Testability* is a measure of how well your system or components allow you to create test criteria and execute tests to determine if the criteria are met.
- You should consider testability when designing your architecture because it makes it easier to diagnose problems earlier and reduce maintenance cost.
- To improve the testability of your application, you can use logging events, provide monitoring resources, and implement test interfaces.

Guidelines

- Clearly define the inputs and outputs of the application or components during the design phase.
- Consider using the Passive View pattern (a variation of the MVP pattern) in the presentation layer, which removes the dependency between the view and the model.

Guidelines

- Design a separate business layer to implement the business logic and workflows, which improves the testability of your application.
- Design an effective logging strategy, which allows you to detect bugs that might otherwise be difficult to discover.

Guidelines

- Logging will help you to focus on faulty code when bugs are found.
  - Log files should contain information that can be used to replicate the issues.
- Design loosely coupled components that can be tested individually.

Software Design and Architecture

Topic#208

END!

Software Design and Architecture

Topic#209
Web Application Frame
Performance Considerations

© Virtual University of Pakistan

- You should identify your performance objectives early in the design phase of a Web application by gathering the non-functional requirements.
- Response time, throughput, CPU, memory, and disk I/O are a few of the key factors you should consider when designing your application.

© Virtual University of Pakistan

Guidelines

- Ensure that the performance requirements are specific, realistic, and flexible.
- Implement caching techniques to improve the performance and scalability of the application.
- Perform batch operations to minimize round trips across boundaries.

© Virtual University of Pakistan

Guidelines

- Reduce the volume of HTML transferred between server and client.
  - For instance, you can disable view state when you do not need it; limit the use of graphics, and considering using compressed graphics where appropriate.
- Avoid unnecessary round trips over the network.

Software Design and Architecture

Topic#209

END!

Software Design and Architecture

Topic#210
Web Application Frame
Security Considerations

© Virtual University of Pakistan

- Security is an important consideration for protecting the integrity and privacy of the data and the resources of your Web application.
- You should design a security strategy for your Web application that uses tested and proven security solutions, and implement authentication, authorization, and data validation to protect your application from a range of threats.

Guidelines

- Consider the use of authentication at every trust boundary.
- Consider implementing a strong authorization mechanism to restrict resource access and protect business logic.

Guidelines

- Consider the use of input validation and data validation at every trust boundary to mitigate security threats such as cross-site scripting and code-injection.
- Do not rely on client-side validation only. Use server-side validation as well.

© Virtual University of Pakistan

Guidelines

- Consider encrypting and digitally signing any sensitive data that is sent across the network.

Software Design and Architecture

Topic#210

END!

© Virtual University of Pakistan

Software Design and Architecture

Topic#211
Web Application Frame
Deployment Considerations

© Virtual University of Pakistan

- When deploying a Web application, you should take into account how layer and component location will affect the performance, scalability, and security of the application.
- You might also need to consider design trade-offs.
- Use either a distributed or a non-distributed deployment approach, depending on the business requirements and infrastructure constraints.

Guidelines

- Consider using non-distributed deployment to maximize performance.
- Consider using distributed deployment to achieve better scalability and to allow each layer to be secured separately.

© Virtual University of Pakistan

Software Design and Architecture

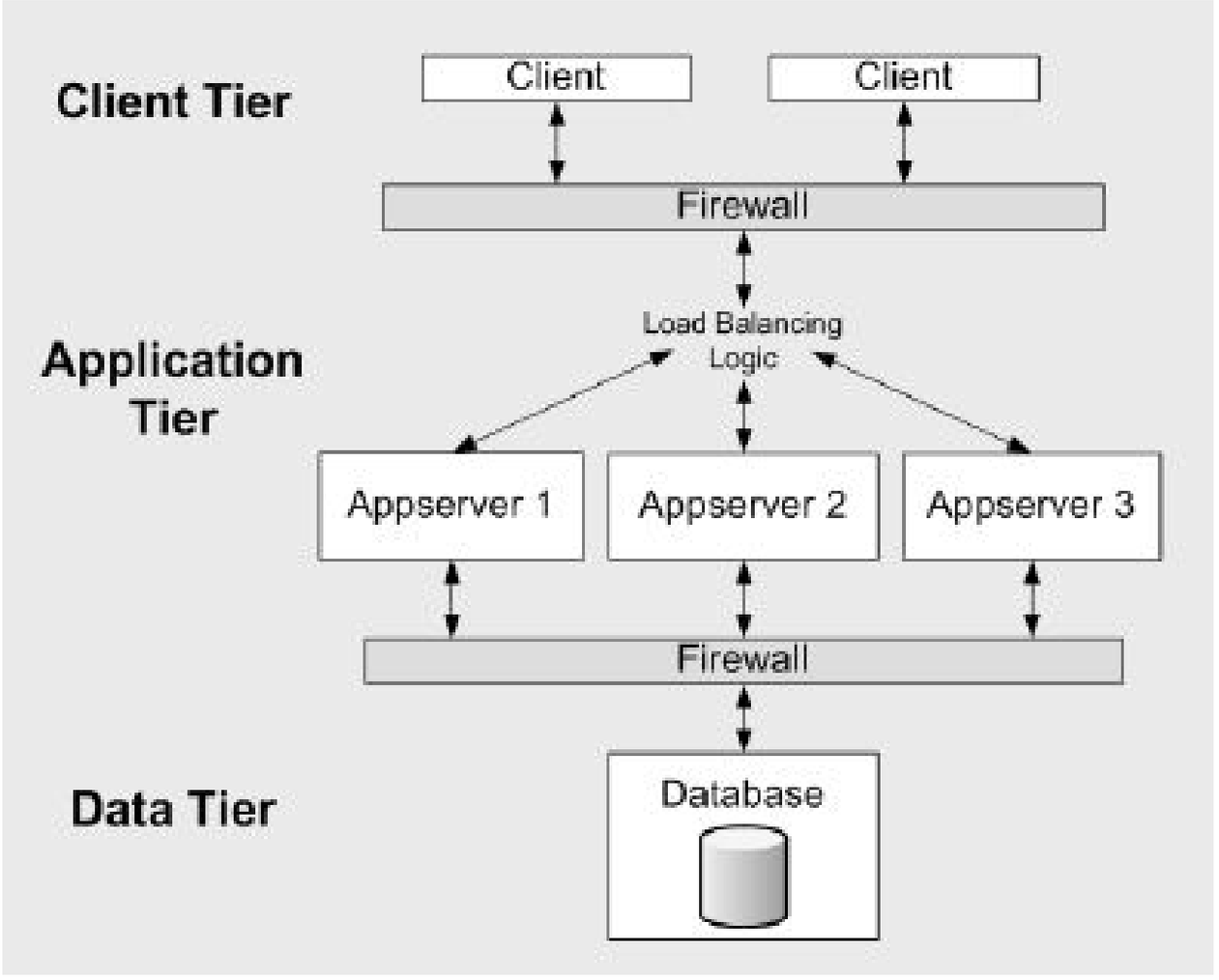
Topic#211

END!

Software Design and Architecture

Topic#212
Web Application Frame
Load Balancing

- When you deploy your Web application on multiple servers, you can use load balancing to distribute requests so that they are handled by different Web servers.
- This helps to maximize response times, resource utilization, and throughput.



Guidelines

- Avoid server affinity when designing scalable Web applications.
  - Server affinity occurs when all requests from a particular client must be handled by the same server.
  - It usually occurs when you use locally updatable caches, or in-process or local session state stores.

Guidelines

- Consider designing stateless components for your Web application

Software Design and Architecture

Topic#212

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 213-217

Architecture in the cloud

© Virtual University of Pakistan

Software Design and Architecture

Topic#213
Cloud Computing – Introduction

© Virtual University of Pakistan

Cloud Computing

- Cloud computing is the on-demand availability of computer system resources, especially data storage and computing power, without direct active management by the user.

© Virtual University of Pakistan

Cloud Computing

- The term is generally used to describe data centers available to many users over the Internet.
- Cloud computing relies on sharing of resources to achieve coherence and economies of scale.

Cloud Computing

- It provides an elastic set of resources through the use of virtual machines, virtual networks, and virtual file systems.
- The cloud has become a viable alternative for the hosting of data centers primarily for economic reasons.

Software Design and Architecture

Topic#214
Cloud Computing – Basic Definitions

© Virtual University of Pakistan

Basic Cloud Definitions

- The essential characteristics of cloud computing
  - based on definitions provided by the U.S. National Institute of Standards and Technology, or NIST

© Virtual University of Pakistan

Basic Cloud Definitions

- *On-demand self-service.*
  - A resource consumer can unilaterally provision computing services, such as server time and network storage, as needed automatically without requiring human interaction with each service's provider.

Basic Cloud Definitions

- *On-demand self-service.*
  - This is sometimes called empowerment of end users of computing resources.
  - Examples of resources include storage, processing, memory, network bandwidth, and virtual machines.

© Virtual University of Pakistan

- *Ubiquitous network access.*
  - Cloud services and resources are available over the network and accessed through standard networking mechanisms that promote use by a heterogeneous collection of clients.
  - This capability is independent of location and device;
    - all you need is a client and the Internet.

© Virtual University of Pakistan

- *Resource pooling*
  - The cloud provider’s computing resources are pooled.
  - In this way they can efficiently serve multiple consumers.
  - The provider can dynamically assign physical and virtual resources to consumers, according to their instantaneous demands.

- *Location independence.*
  - The location independence provided by ubiquitous network access is generally a good thing.

© Virtual University of Pakistan

- *Rapid elasticity.*
  - Due to resource pooling, it is easy for capabilities to be rapidly and elastically provisioned, in some cases automatically, to quickly scale out or in.
  - To the consumer, the capabilities available for provisioning often appear to be virtually unlimited.

© Virtual University of Pakistan

- *Multi-tenancy.*
  - Multi-tenancy is the use of a single application that is responsible for supporting distinct classes or users.
  - Each class or user has its own set of data and access rights, and different users or classes of users are kept distinct by the application.

© Virtual University of Pakistan

Software Design and Architecture

Topic#214

END!

Software Design and Architecture

Topic#215
Cloud Computing – Service Models and Deployment Options

© Virtual University of Pakistan

Software as a Service (SaaS)

- The consumer in this case is an end user.
- The consumer uses applications that happen to be running on a cloud.
- The applications can be as varied as email, calendars, video streaming, and real-time collaboration.

© Virtual University of Pakistan

Platform as a Service (PaaS)

- The consumer in this case is a developer or system administrator.
- The platform provides a variety of services that the consumer may choose to use.
- These services can include various database options, load-balancing options, availability options, and development environments.

© Virtual University of Pakistan

Infrastructure as a Service (IaaS)

- The consumer in this case is a developer or system administrator.
- The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications.

© Virtual University of Pakistan

Deployment Models

- The various deployment models for the cloud are differentiated by who owns and operates the cloud.
- two basic models are private cloud and public cloud:

Deployment Models

- *Private cloud.*
  - The cloud infrastructure is owned solely by a single organization and operated solely for applications owned by that organization.
  - The primary purpose of the organization is not the selling of cloud services.

© Virtual University of Pakistan

Deployment Models

- *Public cloud.*
  - The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services.

© Virtual University of Pakistan

Software Design and Architecture

Topic#215

END!

Software Design and Architecture

Topic#216
Cloud Computing – Multitenancy

© Virtual University of Pakistan

- Multi-tenancy applications are architected explicitly to have a single application that supports distinct sets of users.
- The economic benefit of multi-tenancy is based on the reduction in costs for application update and management.

© Virtual University of Pakistan

- Consider what is involved in updating an application for which each user has an individual copy on their own desktop.
  - New versions must be tested by the IT department and then pushed to the individual desktops.
- Different users may be updated at different times because of disconnected operation, user resistance to updates, or scheduling difficulties.

- Incidents result because the new version may have some incompatibilities with older versions, the new version may have a different user interface, or users with old versions are unable to share information with users of the new version.

- With a multi-tenant application, all of these problems are pushed from IT to the vendor, and some of them even disappear.
- Any update is available at the same instant to all of the users, so there are no problems with sharing.

© Virtual University of Pakistan

- Any user interface changes are referred to the vendor’s hotline rather than the IT hotline, and the vendor is responsible for avoiding incompatibilities for older versions.

© Virtual University of Pakistan

- The problems of upgrading do not disappear, but they are amortized over all of the users of the application rather than being absorbed by the IT department of every organization that uses the application.
- This amortization over more users results in a net reduction in the costs associated with installing an upgraded version of an application.

Software Design and Architecture

Topic#216

END!

Software Design and Architecture

Topic#217
Cloud Computing – Architecting in Cloud Environment

© Virtual University of Pakistan

- In some ways, the cloud is a platform, and architecting a system to execute in the cloud, especially using IaaS, is no different than architecting for any other distributed platform.

© Virtual University of Pakistan

- That is, the architect needs to pay attention to usability, modifiability, interoperability, and testability, just as he or she would for any other platform.
- The quality attributes that have some significant differences are security, performance, and availability.

Security

- Applications in the cloud are accessed over the Internet using standard Internet protocols.
- The security and privacy issues deriving from the use of the Internet are substantial but no different from the security issues faced by applications not hosted in the cloud.

© Virtual University of Pakistan

Security

- The one significant security element introduced by the cloud is multi-tenancy.
  - Multi-tenancy means that your application is utilizing a virtual machine on a physical computer that is hosting multiple virtual machines.
  - If one of the other tenants on your machine is malicious, what damage can they do to you?

Performance

- One virtue of the cloud is that it provides an elastic host.
- Elasticity means that additional resources can be acquired as needed.

Performance

- Regardless of whether the provider automatically allocates additional resources, the application should be self-aware of both its current resource usage and its projected resource usage.

Availability

- The cloud is assumed to be always available. But everything can fail

Availability

- The service-level agreement of some cloud service provider provides a 99.95 percent guarantee of service.
- There are two ways of looking at that number:
  - That is a high number. You as an architect do not need to worry about failure.
  - That number indicates that the service may be unavailable for .05 percent of the time. You as an architect need to plan for that .05 percent.

Availability

- Options:
  - *Stateless services - mirroring*
  - *Data stored across zones - partition*
  - *Graceful degradation*

© Virtual University of Pakistan

Software Design and Architecture

Topic#217

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 218-221

Service-Oriented Architecture

© Virtual University of Pakistan

Software Design and Architecture

Topic#218
Service-Oriented Architecture – Introduction

© Virtual University of Pakistan

- **Service-oriented architecture** (**SOA**) is an architectural pattern for developing software systems by creating loosely coupled, interoperable services that work together to automate business processes.

© Virtual University of Pakistan

- A service is a part of a software application that performs a specific task, providing functionality to other parts of the same software application or to other software applications.
- Some examples of service consumers include web applications, mobile applications, desktop applications, and other services.

- SOA achieves a *SoC*, which is a design principle that separates a software system into parts, with each part addressing a distinct concern.
- A key aspect of SOA is that it decomposes application logic into smaller units that can be reused and distributed.

© Virtual University of Pakistan

- By decomposing a large problem into smaller, more manageable concerns satisfied by services, complexity is reduced and the quality of the software is improved.

© Virtual University of Pakistan

- Each service in a SOA encapsulates a certain piece of logic.
- This logic may be responsible for a very specific task, a business process, or a sub-process.
- Services can vary in size and one service can be composed of multiple other services to accomplish its task.

Software Design and Architecture

Topic#218

END!

Software Design and Architecture

Topic#219
What Makes Service-Oriented Architecture Different from other Distributed Solutions?

© Virtual University of Pakistan

- SOA shares similarities with earlier distributed solutions
- Distributing application logic and separating it into smaller, more manageable units is not what makes SOA different from previous approaches to distributed computing.

© Virtual University of Pakistan

- Naturally, you might think the biggest difference is the use of web services, but keep in mind that SOA does not require web services, although they happen to be a perfect technology to implement with SOA.

- What really sets SOA apart from a traditional distributed architecture is not the use of web services, but how its core components are designed.

- SOA achieves a *SoC*, which is a design principle that separates a software system into parts, with each part addressing a distinct concern.
- Services can vary in size and one service can be composed of multiple other services to accomplish its task.

Software Design and Architecture

Topic#219

END!

Software Design and Architecture

Topic#220
Benefits of Service-Oriented Architecture

© Virtual University of Pakistan

- Increases alignment between business and technology
  - Business logic, in the form of business entities and business processes, exists in the form of physical services with an SOA.
- Promotes federation within an organization
  - Organizations have the flexibility to choose whether they want to replace certain systems, allowing them to use a phased approach to migration.

- Allows for vendor diversity
- Increases intrinsic interoperability
  - It allows for the sharing of data and the reuse of logic. Different services can be assembled together to help automate a variety of business processes.
- Works well with agile development methodologies
  - each task can be made to be manageable in size and more easily understood.

Software Design and Architecture

Topic#220

END!

Software Design and Architecture

Topic#221
Challenges with Service-Oriented Architecture

- SOA solutions may allow organizations to do more, including automating more of their business processes.
- This can cause enterprise architectures to grow larger in scope and functionality as compared to legacy systems.
- Taking on a larger scope of functionality will add complexity to a software system.

- In an SOA, new layers may be added to software architectures, providing more areas where failure can occur and making it more difficult to pinpoint those failures.

© Virtual University of Pakistan

- In addition, as increasing numbers of services are created, deploying new services and new versions of existing services must be managed carefully so that troubleshooting can be effective when an error occurs with a specific transaction.

© Virtual University of Pakistan

Software Design and Architecture

Topic#221

END!

Software Design and Architecture

Fakhar Lodhi

Topics Covered: 222-226

Microservice Architecture

© Virtual University of Pakistan

Software Design and Architecture

Topic#222
Microservice Architecture - Introduction

© Virtual University of Pakistan

- Microservice architecture emerged as the result of the shortcomings and drawbacks of the traditional **service-oriented architecture** (**SOA**) and monolithic architecture.

- The **microservice architecture** (**MSA**) pattern builds software applications using small, autonomous, independently versioned, self-contained services.
- These services use well-defined interfaces and communicate with each other over standard, lightweight protocols.

© Virtual University of Pakistan

- Interaction with a microservice takes place through a well-defined interface.

- A microservice should be a *black box* to the consumers of the service, hiding its implementation and complexity.

© Virtual University of Pakistan

- Each microservice focuses on doing one thing well and they can work together with other microservices in order to accomplish tasks that are more complex.

© Virtual University of Pakistan

- A microservice architecture is particularly well-suited for large and/or complex software systems.
- In contrast with the monolithic architecture, applications built using a microservice architecture handle complexity by splitting the application into smaller services that are easier to manage.

© Virtual University of Pakistan

Software Design and Architecture

Topic#222

END!

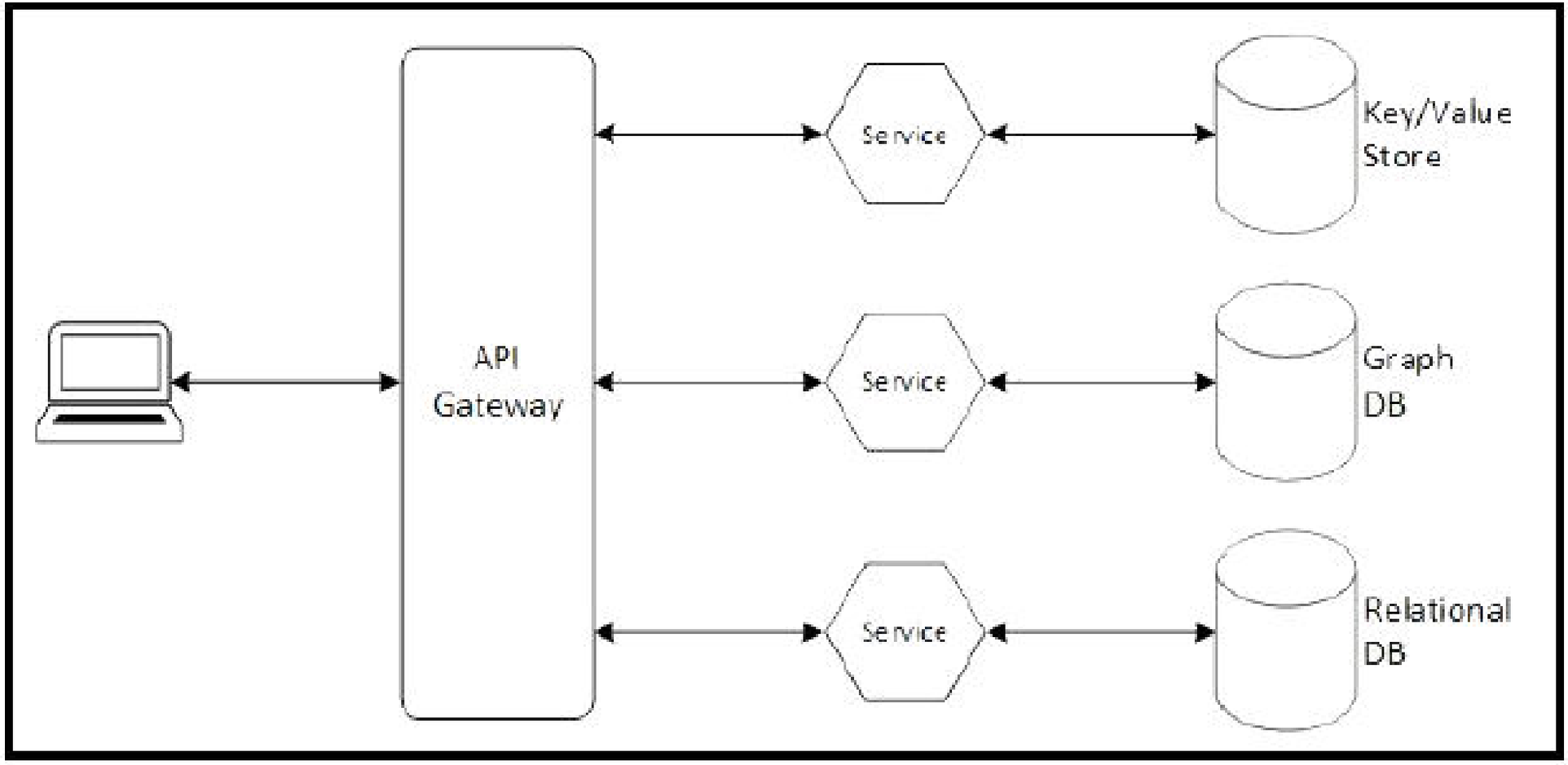
Software Design and Architecture

Topic#223
Characteristics of microservice architecture

- Small, focused services
- Well-defined service interfaces
- Autonomous and independently deployable services
- Independent data storage
- Communicating with lightweight protocols
- Better fault isolation

© Virtual University of Pakistan

Microservice Architecture Example



- Incoming requests are commonly handled by an API gateway, which serves as the entry point to the system.
- It is an HTTP server that takes requests from clients and routes them to the appropriate microservice through its routing configuration.

Software Design and Architecture

Topic#223

END!

Software Design and Architecture

Topic#224
Designing Polyglot Services

© Virtual University of Pakistan

- Monolithic applications focus on using a particular programming language and technology stack.
- Because that type of application is written as a single unit, it is more difficult to take advantage of different types of technology.

© Virtual University of Pakistan

- Complex applications need to solve a variety of problems.
- Being able to select different technologies for different problems can be useful rather than trying to solve all of the problems with a single technology.

© Virtual University of Pakistan

- Polyglot
  - a person who knows and is able to use several languages

- One of the many advantages of using a microservice architecture is that it affords you the option of using multiple programming languages, runtimes, frameworks, and data storage technologies.

© Virtual University of Pakistan

- Having polyglot microservices is certainly not required when using a microservice architecture.
- In many cases, an organization will focus on a limited number of technologies, and the skillsets of the development team will reflect that.

- However, software architects should be aware of the option and recognize opportunities where it can be used effectively.

- Two of the concepts related to polyglot microservices are:
- polyglot programming
  - With **polyglot programming**, a single application uses multiple programming languages in its implementation.
- polyglot persistence
  - multiple persistence options are used within a single application

Software Design and Architecture

Topic#224

END!

Software Design and Architecture

Topic#225
Stateless versus Stateful Microservices

- Stateful applications store state from client requests on the server itself and use that state to process further requests.
  - When a user sends a login request, it enables login to be true and user authenticated now, and on the second request, the user sees the dashboard.

- Stateful applications don’t need to make a call to DB second time as session info stored on the server itself, hence it is faster.

Problems with Stateful Apps

- Load balancing and scalability

© Virtual University of Pakistan

- Stateless applications don’t store any state on the server.
- Any data that flows via a stateless service is typically transitory and the state is stored only in a separate back-end service like a database.
  - Any associated storage is typically transitory.
  - If the container restarts for instance, anything stored is lost.

- Each microservice can either be *stateless* or *stateful*.

Stateless	Stateful
Requests are self-contained	Shorter messages
Better fault tolerance	Better performance
No space reserved for file descriptor tables	File locking possible
No limit on open files	Read ahead possible
No problem if client crashes	

Software Design and Architecture

Topic#225

END!

Software Design and Architecture

Topic#226
Microservice Architecture Challenges

- a microservice architecture introduces complexity simply not found in a monolithic application.

- When multiple services are working together in a distributed system and something goes wrong, there is added complexity in figuring out what and where something failed.

- Decomposing a complex system into the right set of microservices can be difficult.
- It requires a knowledge of the domain and can be somewhat of an art.

- You don't want a system with services that are too fine-grained, resulting in a large number of services.
  - As the number of services increase, the management of those services becomes increasingly complex.
- Balancing between fine-grained and coarse-grained services

- Controlling coupling between services that are deployed together
  - If you are not careful, you will end up with a microservice architecture that is a monolith in disguise.

- The use of multiple databases is another challenge when using a microservice architecture.
- It is common for a business transaction to update multiple entities, which will require the use of multiple microservices.
- With each one having its own database, this means that updates must take place in multiple databases.

Software Design and Architecture

Topic#226

END!

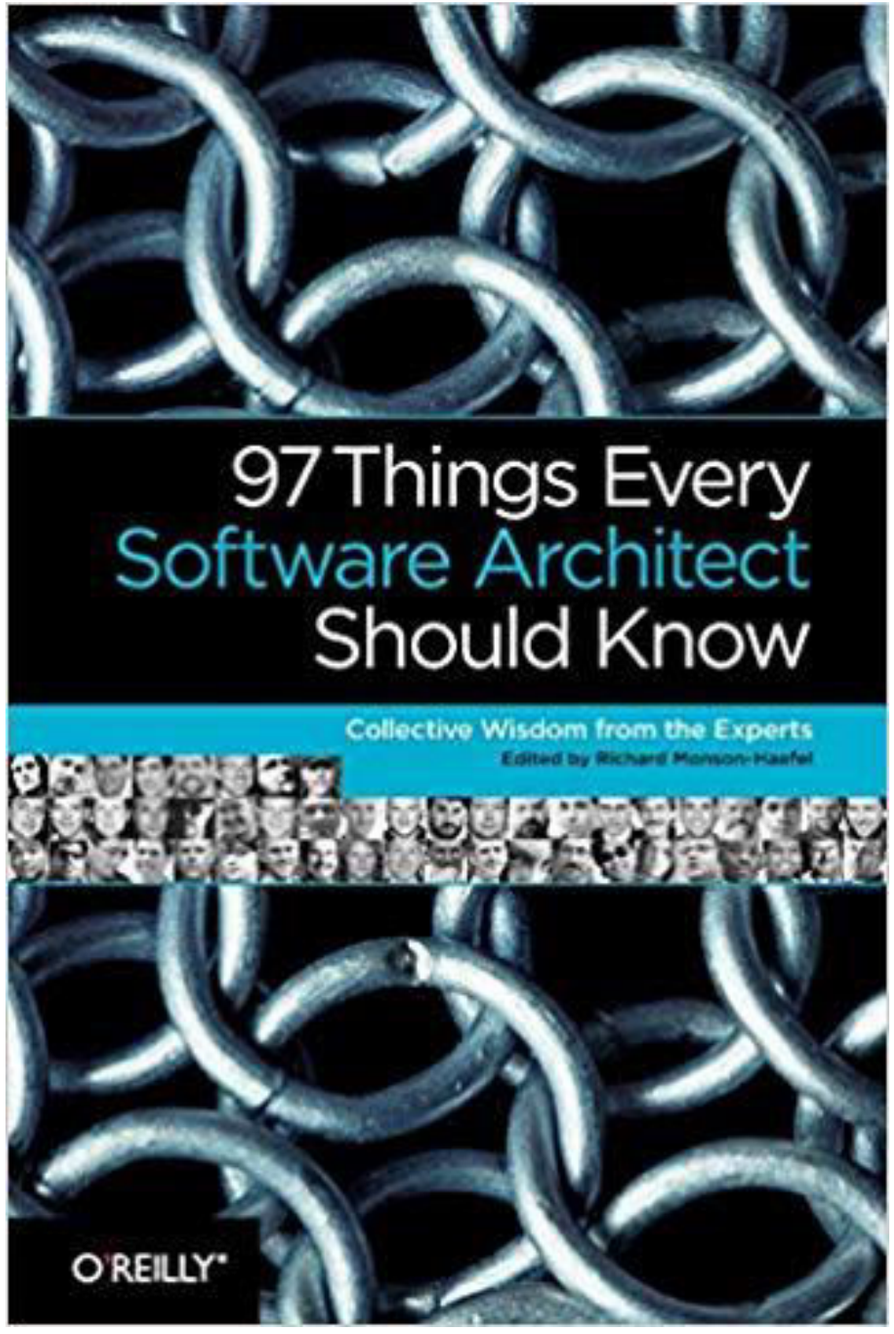
Software Design and Architecture

Fakhar Lodhi

Topics Covered: 227

Summing it up - What every software architect should know

© Virtual University of Pakistan


97 Things Every
Software Architect
Should Know
Collective Wisdom from the Experts
Edited by Richard Monson-Haefel
O'REILLY

- Business Drives
- Understand the Business Domain

- Architecting Is About Balancing
- Architectural Tradeoffs

- Simplify Essential Complexity; Diminish Accidental Complexity
- Simplicity Before Generality, Use Before Reuse

- Architects’ Focus Is on the Boundaries and Interfaces

- Communication Is King
- Record Your Rationale
- Share Your Knowledge and Experiences
- Quantify.

- Application Architecture Determines Application Performance
- It’s Never Too Early to Think About Performance

- Everything Will Ultimately Fail
- Time Changes Everything

- Architects Must Be Hands On
- Try Before Choosing

- There Is No One-Size-Fits-All Solution

- Software Architecture Has Ethical Consequences

Software Design and Architecture

Topic#227

END!