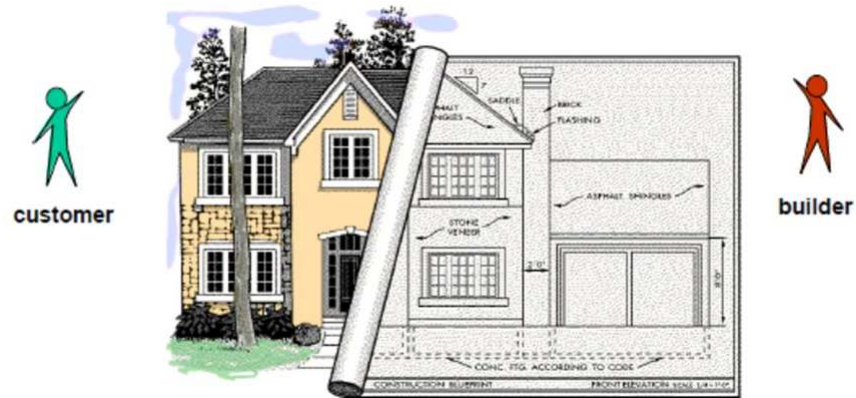


Department of Software Engineering Bahria University, Karachi

## Software Design & Architecture



### Agenda for this week

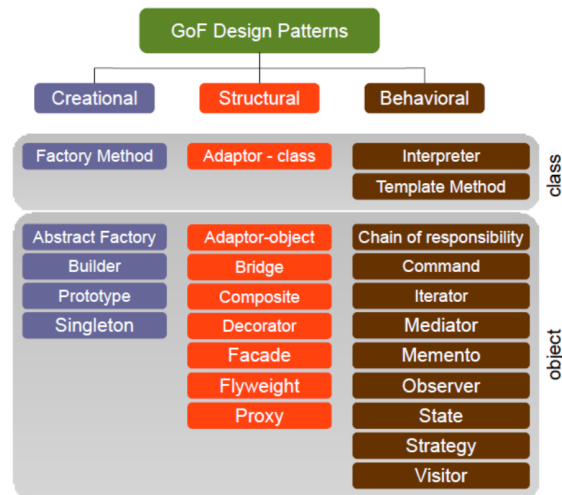
- As per lesson plan:

Week	Week Days	Lecture Number	Tentative Course Plan
7	16 <sup>th</sup> Mar	13 - 14	<ul style="list-style-type: none"> <li>STRUCTURAL DESIGN PATTERNS</li> </ul>



## Structural Design Patterns

- They are categorized in three groups: Creational, Structural, and Behavioral as listed below:



3



## Structural Design Patterns

- These design patterns are all about Class and Object *composition*.
- Structural class-creation patterns use *inheritance* to compose interfaces.
- Structural object-patterns define ways to compose objects to obtain new functionality.
- In everyday life, examples of a structural pattern include a reservation sign, placed on a restaurant table, that acts as a placeholder until the guests arrive; or an adapter that enables a U.S.-compliant electricity plug to make use of a European compliant power point.*

4



## Structural Design Patterns

- *What?*
- In software design, structural patterns give the software designer a tool that can be used to enhance classes: behavior or functionality may be varied by manipulating the structure of participating classes.
- *Where?*
- Structural patterns are used where a design needs to vary or enhance the behavior of classes. For example, arranging classes in a strategic structure (e.g., Remote-Proxy pattern) enables communication across a domain.


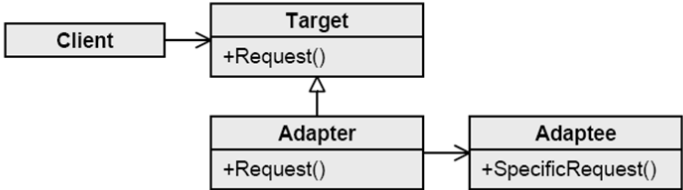
5


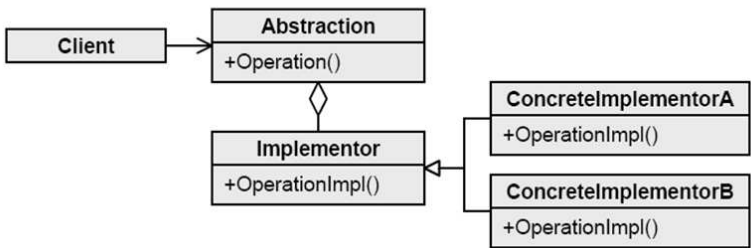


## Structural Design Patterns

- *Why?*
- In some domains there is a requirement to reconfigure structures to give effect to behavioral characteristics that are not native to a given design or arrangement of classes. In those domains, that requirement presents a problem, so to overcome the problem, structural patterns are incorporated into the design of the program.
- *How?*
- Commonly, a structural pattern is a manipulation process that centers around the implementation of an interface.
- *Programming?*
- These patterns will be implemented during lab sessions. Follow hyperlink to see sample scenarios and relevant code examples.


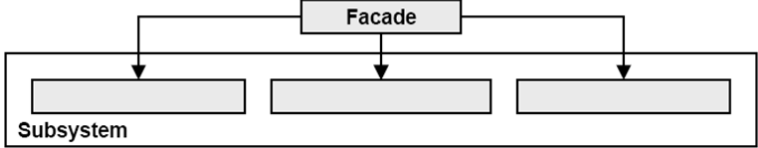
6


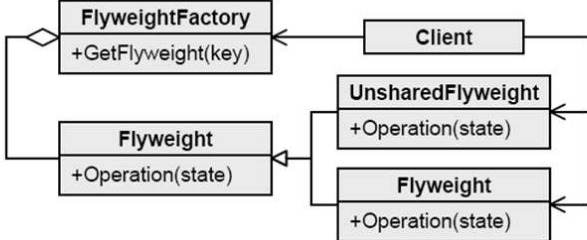
 <b>Catalog of Design Patterns</b>	
Pattern Name	<b>Adapter</b>
Category	Structural
Purpose	<i>Match interfaces of different classes</i> This pattern lets classes work together that could not otherwise because of incompatible interfaces.
Diagram	 <pre> classDiagram     class Client     class Target {         +Request()     }     class Adapter {         +Request()     }     class Adaptee {         +SpecificRequest()     }     Client --&gt; Target     Adapter -- &gt; Target     Adapter --&gt; Adaptee           </pre> <p>The diagram illustrates the Adapter pattern. A <b>Client</b> depends on a <b>Target</b> interface, which defines a <code>+Request()</code> method. An <b>Adapter</b> class implements the <b>Target</b> interface, also providing a <code>+Request()</code> method. The <b>Adapter</b> class holds a reference to an <b>Adaptee</b> object, which implements the <code>+SpecificRequest()</code> method. The <b>Adapter</b>'s <code>+Request()</code> method delegates the call to the <b>Adaptee</b>'s <code>+SpecificRequest()</code> method.</p>
<b>Code &amp; Examples:</b> <a href="https://www.dofactory.com/net/adapter-design-pattern">https://www.dofactory.com/net/adapter-design-pattern</a>	
7	


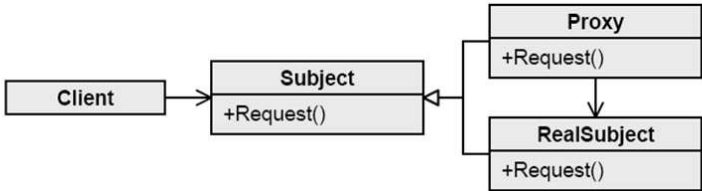
 <b>Catalog of Design Patterns</b>	
Pattern Name	<b>Bridge</b>
Category	Structural
Purpose	<i>Separates an object's interface from its implementation</i> This pattern lets classes work together that could not otherwise because of incompatible interfaces.
Diagram	 <pre> classDiagram     class Client     class Abstraction {         +Operation()     }     class Implementor {         +OperationImpl()     }     class ConcreteImplementorA {         +OperationImpl()     }     class ConcreteImplementorB {         +OperationImpl()     }     Client --&gt; Abstraction     Abstraction o-- Implementor     Implementor &lt; -- ConcreteImplementorA     Implementor &lt; -- ConcreteImplementorB           </pre> <p>The diagram illustrates the Bridge pattern. A <b>Client</b> depends on an <b>Abstraction</b> interface, which defines a <code>+Operation()</code> method. The <b>Abstraction</b> interface has a <b>Implementor</b> interface, which defines a <code>+OperationImpl()</code> method. The <b>Abstraction</b> interface holds a reference to an <b>Implementor</b> object. There are two concrete implementations of the <b>Implementor</b> interface: <b>ConcreteImplementorA</b> and <b>ConcreteImplementorB</b>, both of which implement the <code>+OperationImpl()</code> method.</p>
<b>Code &amp; Examples:</b> <a href="https://www.dofactory.com/net/bridge-design-pattern">https://www.dofactory.com/net/bridge-design-pattern</a>	
8	


Catalog of Design Patterns	
Pattern Name	<b>Composite</b>
Category	Structural
Purpose	<i>A tree structure of simple and composite objects</i> This pattern lets classes work together that could not otherwise because of incompatible interfaces.
Diagram	<pre> classDiagram     class Client     class Component {         +Operation()         +Add(component)         +Remove(component)         +GetChild(index)     }     class Composite {         +Operation()         +Add(component)         +Remove(component)         +GetChild(index)     }     class Leaf {         +Operation()     }     Client --&gt; Component     Component &lt; -- Composite     Component &lt; -- Leaf     Composite o-- Component           </pre> <p>The diagram illustrates the Composite Design Pattern. It shows a <b>Client</b> class that interacts with a <b>Component</b> interface. The <b>Component</b> interface defines methods: <code>+Operation()</code>, <code>+Add(component)</code>, <code>+Remove(component)</code>, and <code>+GetChild(index)</code>. Two classes, <b>Composite</b> and <b>Leaf</b>, implement the <b>Component</b> interface. The <b>Composite</b> class has the same methods as the <b>Component</b> interface and holds a collection of <b>Component</b> objects (indicated by a hollow diamond on the arrow from <b>Composite</b> to <b>Component</b>). The <b>Leaf</b> class also implements the <b>Component</b> interface but only has the <code>+Operation()</code> method.</p>
<b>Code &amp; Examples:</b> <a href="https://www.dofactory.com/net/composite-design-pattern">https://www.dofactory.com/net/composite-design-pattern</a>	
9	

Catalog of Design Patterns	
Pattern Name	<b>Decorator</b>
Category	Structural
Purpose	<i>Add responsibilities to objects dynamically</i> This pattern attaches additional responsibilities to an object dynamically while keeping the interface same.
Diagram	<pre> classDiagram     class Component {         +Operation()     }     class ConcreteComponent {         +Operation()     }     class Decorator {         +Operation()     }     class ConcreteDecorator {         +Operation()         +AddedBehavior()     }     Component &lt; -- ConcreteComponent     Component &lt; -- Decorator     Decorator o-- Component     Decorator o-- Decorator     Decorator &lt; -- ConcreteDecorator           </pre> <p>The diagram illustrates the Decorator Design Pattern. It shows a <b>Component</b> interface with a <code>+Operation()</code> method. Two classes, <b>ConcreteComponent</b> and <b>Decorator</b>, implement the <b>Component</b> interface. The <b>Decorator</b> class also has a <code>+Operation()</code> method and holds a collection of <b>Component</b> objects (indicated by a hollow diamond on the arrow from <b>Decorator</b> to <b>Component</b>). The <b>ConcreteDecorator</b> class implements the <b>Decorator</b> interface and has an additional <code>+AddedBehavior()</code> method. The <b>ConcreteComponent</b> class implements the <b>Component</b> interface directly.</p>
<b>Code &amp; Examples:</b> <a href="https://www.dofactory.com/net/decorator-design-pattern">https://www.dofactory.com/net/decorator-design-pattern</a>	
10	

 <b>Catalog of Design Patterns</b>	
Pattern Name	<b>Façade</b>
Category	Structural
Purpose	<i>A single class that represents an entire subsystem</i> This pattern provides a simpler interface to a larger and more complex system such as a class library or a complex API.
Diagram	
<b>Code &amp; Examples:</b> <a href="https://www.dofactory.com/net/facade-design-pattern">https://www.dofactory.com/net/facade-design-pattern</a>	
11	

 <b>Catalog of Design Patterns</b>	
Pattern Name	<b>Flyweight</b>
Category	Structural
Purpose	<i>A fine-grained instance used for efficient sharing</i> This pattern minimizes memory usage by sharing common data between objects.
Diagram	
<b>Code &amp; Examples:</b> <a href="https://www.dofactory.com/net/flyweight-design-pattern">https://www.dofactory.com/net/flyweight-design-pattern</a>	
12	

 <b>Catalog of Design Patterns</b>	
Pattern Name	<b>Proxy</b>
Category	Structural
Purpose	<p><i>An object representing another object</i></p> <p>Proxy is a surrogate or placeholder class for another class mostly done with an intention of intercepting access to the said class.</p>
Diagram	 <pre> classDiagram     class Client     class Subject {         +Request()     }     class Proxy {         +Request()     }     class RealSubject {         +Request()     }     Client --&gt; Subject     Proxy &lt; -- RealSubject     Proxy &lt; -- Subject     </pre>
<p><b>Code &amp; Examples:</b> <a href="https://www.dofactory.com/net/proxy-design-pattern">https://www.dofactory.com/net/proxy-design-pattern</a></p>	
13	

 <b>References</b>
<ol style="list-style-type: none"> <li>1. <i>Software Architecture, Perspectives on an Emerging Discipline</i> By Mary Shaw &amp; David Garlan</li> <li>2. <i>The Art of Software Architecture, Design Methods &amp; Techniques</i> By Stephen T. Albin</li> <li>3. <i>Essential Software Architecture</i> By Ian Gorton</li> <li>4. <i>Design Patterns, Elements of Reusable Object-Oriented Software</i> By Erich Gamma, Richard Helm, Ralph Johnson &amp; John Vlissides</li> </ol>
14