

OVERVIEW **Structural Design Patterns: Adapter, Composite and Facade**

Structural patterns deal with how existing classes and objects are put together. Adapter converts the interface of an existing class (the adaptee) into the interface a client expects (the target), either by wrapping it (object adapter) or by inheriting from it (class adapter). Composite arranges objects into part-whole trees so that a client can treat a single leaf and a whole group the same way. Facade puts one simple, higher-level interface in front of a complicated subsystem.

Adapter

Client → Target interface ← Adapter → Adaptee. The adapter translates calls; the adaptee is unchanged. Example: BirdAdapter makes a Sparrow act as a ToyDuck.

Object vs. class adapter

Object adapter holds an adaptee (composition, run time); class adapter inherits from it (compile time, needs multiple inheritance). Object adapter is preferred.

Composite

Component interface; Leaf implements it; Composite implements it and holds a list of Components, forwarding each call to its children.

Facade

One class (ShopKeeper) offers simple task methods (iphoneSale()) and calls the subsystem classes for the client.

HOW THE LECTURE IS ORGANIZED

Slides 1–3 Structural patterns

Slides 4–16 Adapter: intent, structure, ToyDuck and Bird example, object vs. class adapter, pros and cons

Slides 17–31 Composite: intent, structure, Java example, steps, benefits, file-system example

Slides 32–42 Facade: intent, structure, mobile-shop example, steps, benefits, summary

EXAM FOCUS

Final 252 Q7-1 (2.5 marks) asked you to recognise Composite in a file explorer (files and nested folders, total folder size) and draw its class diagram. Expect Adapter and Facade as “which pattern?” questions and as a comparison (Adapter converts an interface; Facade simplifies a subsystem).

HOW TO READ THIS PAGE

Every original slide is shown unchanged. **Yellow marks** on a slide point to a numbered note: hover or tap a mark to jump to its note, and hover a note to see its marks. Questions tagged **Past exam** appeared on a real paper; **Prediction** questions are likely angles, not confirmed.

Software Design and Architecture

[Structural Design Patterns] – Chapter 07, L01

NOTES

- 1 Chapter 7 moves from creating objects to **structuring** them. Lecture 1: three structural patterns. Lecture 2: behavioural patterns.

SLIDE 2 / 42 Lecture Outlines

Lecture Outlines

- **Previously (Creational Design Patterns)**
- **Structural Patterns in Detailed Design**
 - ✓ **Adaptor**
 - ✓ **Composite**
 - ✓ **Facade**
- **What's next...**

NOTES

- 1 Previously: the creational patterns (Abstract Factory, Factory Method, Builder, Singleton).
- 2 Today: **Adapter**, **Composite** and **Facade**.

SLIDE 3 / 42 Structural Patterns in Detailed Design

Structural Patterns in Detailed Design

- Structural design patterns are patterns that deal with designing the **structure** of **existing classes** or **objects** at run .
- Popular structural design patterns include:
 - ✓ **Adaptor**
 - ✓ **Facade**
 - ✓ **Composite**

NOTES

- 1 **Structural** patterns deal with designing the **structure of existing classes or objects**: how they are connected and combined, so parts fit together or present a simpler boundary.
- 2 The three here: **Adapter** (make incompatible interfaces fit), **Facade** (simplify a subsystem) and **Composite** (build part-whole trees).

Adapter Design Pattern

- The **Adapter** design pattern is a class/object structural design pattern used to adapt an exiting interface that is expected in a software system.
- The **intent** of the Adapter is to:
 - ✓ Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

NOTES

- 1 **Adapter** is a **class or object structural** pattern (it has both variants) used to adapt an existing interface to the one a system expects.
- 2 Its **intent**: **convert the interface of a class into another interface that clients expect**. Adapter lets classes work together that otherwise couldn't because their interfaces are incompatible. Think of a travel plug converter: the device and the socket don't change; the converter between them does the fitting.

EXAM ANGLE

Prediction

Q1

Making an existing class with an incompatible interface usable by a client, without changing either, is the:

- A. Adapter pattern
- B. Facade pattern
- C. Composite pattern
- D. Observer pattern

Answer: A. Adapter pattern

Adapter Design Pattern-



A real-life example for the use of adapters

NOTES

- 1 The real-life version: **your existing system (the client)** has an Ethernet cable and sends its `request()`.
- 2 The **adapter** (an Ethernet-to-USB dongle) offers the **target interface** the cable expects, and passes on a **translated request()**.
- 3 The **vendor class (adaptee)** only understands its own **adaptee interface** (USB). Neither the cable nor the USB port changed; the adapter did the translation.

The **adapter** design pattern solves problems like:^[3]

- How can a class be reused that does not have an interface that a client requires?
- How can classes that have incompatible interfaces work together?
- How can an alternative interface be provided for a class?

Often an (already existing) class can't be reused only because its interface doesn't conform to the interface clients require.

The adapter design pattern describes how to solve such problems:

- Define a separate **adapter** class that converts the (incompatible) interface of a class (**adaptee**) into another interface (**target**) clients require.
- Work through an **adapter** to work with (reuse) classes that do not have the required interface.

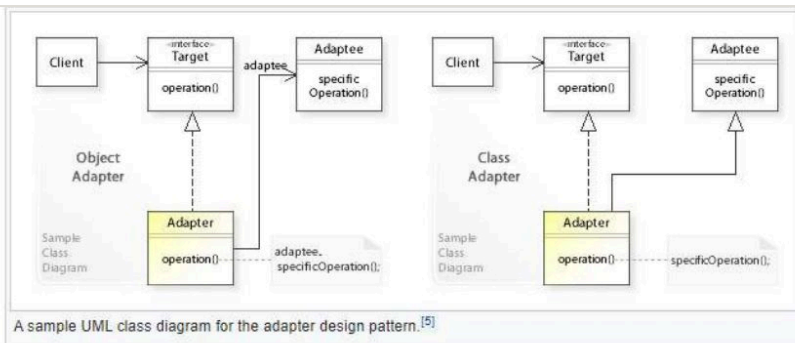
The key idea in this pattern is to work through a separate **adapter** that adapts the interface of an (already existing) class without changing it.

Clients don't know whether they work with a **target** class directly or through an **adapter** with a class that does not have the **target** interface.

See also the UML class diagram below.

NOTES

- 1 The problems Adapter solves: how can a class be **reused** when it doesn't have the interface a client requires? How can classes with **incompatible interfaces** work together? How can an alternative interface be provided?
- 2 The solution: define a separate **adapter** class that converts the (incompatible) interface of a class (the **adaptee**) into the interface clients require (the **target**). Clients work through the adapter and don't know whether they are talking to a target directly or through an adapter.



- In the above UML class diagram, the Client class that requires (depends on) a Target Interface cannot reuse the Adaptee class directly because its interface doesn't conform to the Target Interface. Instead, the Client works through an Adapter class that implements the Target Interface in terms of Adaptee:

The Object Adapter implements the Target Interface by delegating to an Adaptee object at **run-time** (adaptee.specificOperation()). *(during the running time)*

The Class Adapter implements the Target Interface by **inheriting** from an Adaptee class at **compile-time** (specificOperation()). *same target interface*

NOTES

- The UML: the **Client** depends on a **Target** interface; it can't use the **Adaptee** directly because the adaptee's interface doesn't conform. The **Adapter** implements Target in terms of Adaptee.
- Object adapter** (left): implements Target by **delegating** to an Adaptee object held in a field, at **run time** (adaptee.specificOperation()).
- Class adapter** (right): implements Target by **inheriting** from the Adaptee class, at **compile time** (specificOperation() is called on itself).

EXAM ANGLE

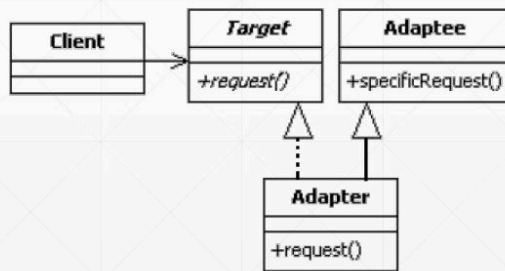
Prediction

Q1

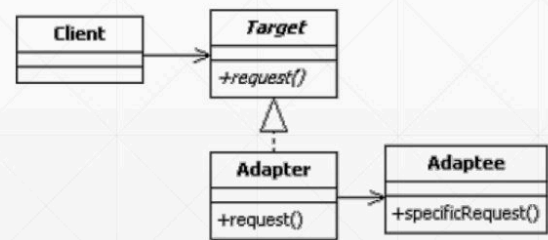
What is the difference between an object adapter and a class adapter?

Answer: An object adapter holds a reference to an adaptee object and delegates to it (composition, decided at run time). A class adapter inherits from the adaptee class and implements the target (inheritance, fixed at compile time; needs multiple inheritance of classes, which Java lacks). Object adapters are preferred.

Adapter Design Pattern - Structure



Class Structural version
of Adapter Pattern



Object Structural version
of Adapter Pattern

To use an adapter:

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request on the adaptee using the adaptee interface.
3. Client receive the results of the call

Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

8

NOTES

- 1 **Class version** (left): Adapter inherits from both Target and Adaptee (two hollow triangles) and its `request()` calls the inherited `specificRequest()`.
- 2 **Object version** (right): Adapter realises Target and has an association arrow to an Adaptee object; its `request()` calls `adaptee.specificRequest()`.
- 3 To use an adapter: (1) the client calls a method on the adapter through the **target** interface; (2) the adapter translates it into calls on the **adaptee** interface; (3) the client gets the results, never knowing an adaptee was involved.

EXAM ANGLE

Prediction

Q1

In the Adapter pattern, the existing class with the incompatible interface is called the:

- A. Adaptee
- B. Target
- C. Client
- D. Composite

Answer: A. Adaptee

In Online class
we will find another example
Find WQY

Adapter Design Pattern – Example*

Example:

- Suppose you have a **Bird** class with `fly()` , and `makeSound()` methods.
- Also a **ToyDuck** class with `squeak()` method.
- Let's assume that you are short on ToyDuck objects and you would like to use Bird objects in their place.
- Birds have some similar functionality but implement a different interface, so we can't use them directly.
- So we will use **adapter pattern**. Here our **target** would be **ToyDuck** and **adaptee** would be **Bird**.

Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

* Source: <http://www.geeksforgeeks.org/adapter-pattern/>

9

NOTES

- 1 The example (from GeeksforGeeks): a **Bird** class with `fly()` and `makeSound()` .
- 2 A **ToyDuck** with `squeak()` .
- 3 We are short on ToyDuck objects and want to use Birds in their place. Birds behave similarly but have a different interface, so they can't be used directly.
- 4 So: **target = ToyDuck**, **adaptee = Bird**, and a BirdAdapter in between.

Adapter Design Pattern – Example



```
interface ToyDuck
```

```
{
    // target interface
    // toyducks dont fly they just make
    // squeaking sound
    public void squeak();
}
```

```
class PlasticToyDuck implements ToyDuck
```

```
{
    public void squeak()
    {
        System.out.println("Squeak");
    }
}
```

```
interface Bird
```

```
{
    // birds implement Bird interface that allows
    // them to fly and make sounds adaptee interface
    public void fly();
    public void makeSound();
}
```

```
class Sparrow implements Bird
```

```
{
    // a concrete implementation of bird
    public void fly()
    {
        System.out.println("Flying");
    }
    public void makeSound()
    {
        System.out.println("Chirp Chirp");
    }
}
```

NOTES

1 The existing code:

CODE

```
interface ToyDuck { public void
squeak(); }
```

```
class PlasticToyDuck implements
ToyDuck { public void squeak() {
System.out.println("Squeak"); } }
```

```
interface Bird { public void fly();
public void makeSound(); }
```

```
class Sparrow implements Bird { fly()
{ ..."Flying"... } makeSound() { ..."Chirp
Chirp"... } }
```

WHAT IT DOES

The target interface: toy ducks don't fly, they just squeak.

A normal toy duck.

The adaptee interface: birds fly and make sounds.

A concrete bird.

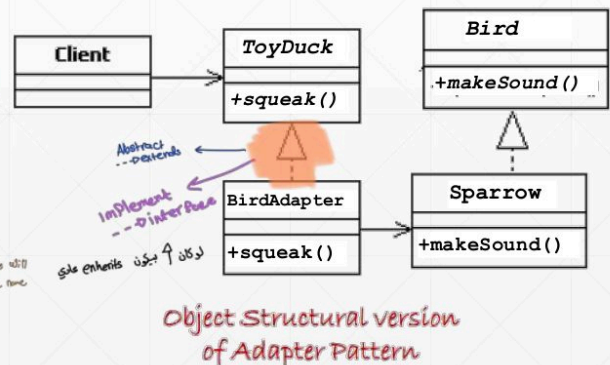
Adapter Design Pattern – Example

```

class BirdAdapter implements ToyDuck
{
    // You need to implement the interface your
    // client expects to use.
    Bird bird; → Object for Adapter
    public BirdAdapter(Bird bird) → Create Constructor with 1 argument
    {                                     as object from Adapter interface
        // this is reference to the bird
        // we need reference to the object we
        // are adapting
        this.bird = bird;
    }

    public void squeak() → Overriden Method / Inside the body of this method we will
    {                                     call the Adapter method by the name
        // translate the methods appropriately                                     of the object (bird)
        bird.makeSound();
    }
}

```



NOTES

1 The adapter:

CODE

```
class BirdAdapter implements ToyDuck {
```

```
    Bird bird;
```

```
    public BirdAdapter(Bird bird) {
        this.bird = bird; }

```

```
    public void squeak() {
        bird.makeSound(); } }

```

WHAT IT DOES

Implements the interface the client expects (ToyDuck).

Holds a reference to the object being adapted (object adapter).

The bird to adapt is passed in.

Translates the target call into the adaptee call.

2 The UML on the right: Client → «ToyDuck» ← BirdAdapter (realization) → Bird ← Sparrow: the object structural version of the pattern.

Adapter Design Pattern – Example

```

class Main
{
    public static void main(String args[]) → Create TestClass (Client)
    {
        Sparrow sparrow = new Sparrow();
        PlasticToyDuck toyDuck = new PlasticToyDuck();

        // Wrap a bird in a birdAdapter so that it
        // behaves like toy duck
        ToyDuck birdAdapter = new BirdAdapter(sparrow);
        name of the class that implements the target interface → new object from class called (Sparrow)

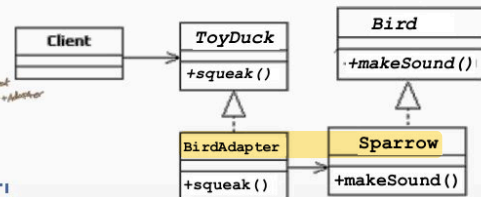
        System.out.println("Sparrow...");
        sparrow.fly();
        sparrow.makeSound();

        System.out.println("ToyDuck...");
        toyDuck.squeak();

        // toy duck behaving like a bird
        System.out.println("BirdAdapter...");
        birdAdapter.squeak();
    }
}

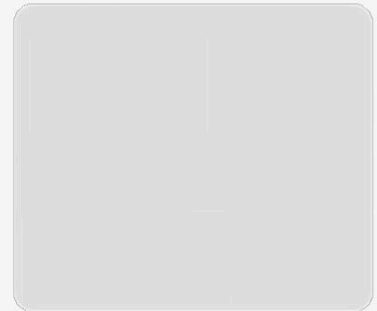
```

Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides



Object Structural version
of Adapter Pattern

Output:



NOTES

1 The client:

CODE

```

Sparrow sparrow = new Sparrow();
PlasticToyDuck toyDuck = new
PlasticToyDuck();

```

WHAT IT DOES

A real bird and a real toy duck.

```

ToyDuck birdAdapter = new
BirdAdapter(sparrow);

```

Wrap the bird so it behaves like a toy duck.

```

sparrow.fly(); sparrow.makeSound();
toyDuck.squeak();

```

Prints "Flying", "Chirp Chirp", "Squeak".

```

birdAdapter.squeak();

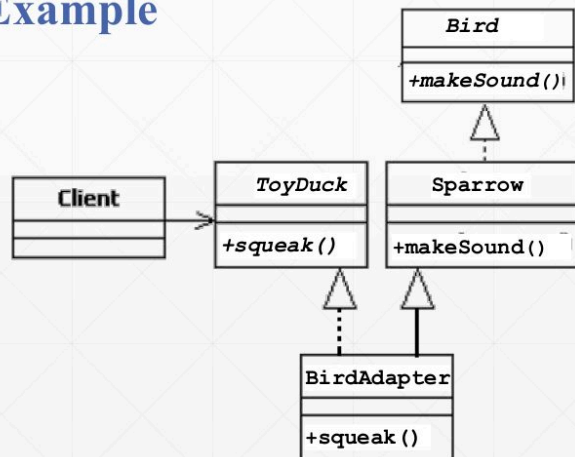
```

A ToyDuck call that prints "Chirp Chirp": the toy duck is behaving like a bird.

Adapter Design Pattern – Example

➤ Object Adapter Vs Class Adapter

- ✓ The adapter pattern we have implemented above is called **Object Adapter Pattern** because the adapter holds an **instance** of adaptee. There is also another type called **Class Adapter Pattern** which use inheritance instead of composition but you require **multiple inheritance** to implement it.



*Class Structural version
of Adapter Pattern*

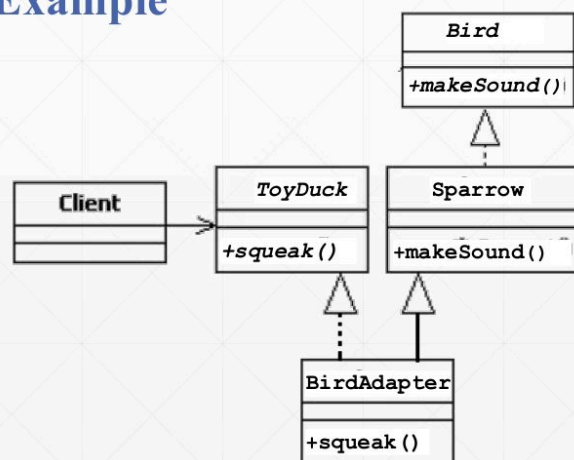
NOTES

- 1 The version just built is the **object adapter**, because the adapter holds an **instance** of the adaptee.
- 2 The alternative, the **class adapter**, uses inheritance instead of composition, which requires **multiple inheritance**: **BirdAdapter** would extend **Sparrow** and implement **ToyDuck** (the diagram on the right).

Adapter Design Pattern – Example

➤ Object Adapter Vs Class Adapter

- ✓ Here instead of having an adaptee object inside adapter (composition) to make use of its functionality the adapter (**BirdAdapter**) class **inherits** the adaptee (**Sparrow**) class.
- ✓ Since **multiple inheritance** is not supported by many languages and is associated with many problems **object adapter pattern** is preferred



*Class Structural version
of Adapter Pattern*

NOTES

- 1 In the class adapter, BirdAdapter **inherits** the adaptee (Sparrow) instead of containing it, and calls the inherited `makeSound()` inside `squeak()`.
- 2 Because many languages don't support multiple inheritance of classes (and it causes problems where they do), the **object adapter is preferred**. In Java this one case works only because ToyDuck is an interface.

SLIDE 15 / 42 Adapter Example: client code and explanation

Step 4: Client Code

```
java
public class Client {
    public static void main(String[] args) {
        // Create a Sparrow
        Sparrow sparrow = new Sparrow();

        // Wrap it inside an Adapter
        ToyDuck birdAdapter = new BirdAdapter(sparrow);

        // Client expects a ToyDuck, but actually uses a Bird
        birdAdapter.squeak(); // Output: "Chirp chirp"
    }
}
```

➤ Explanation

- **ToyDuck** is the expected interface (Target).
- **Bird/Sparrow** is the legacy interface (Adaptee).
- **BirdAdapter** converts `squeak()` into `makeSound()`.
- The **Client** works with ToyDuck but internally a Bird is used.

NOTES

- 1 The client again: create a Sparrow, wrap it in a BirdAdapter, and call `birdAdapter.squeak()`, which outputs "Chirp chirp": the client expects a ToyDuck but actually uses a Bird.
- 2 **ToyDuck** is the expected interface (target); **Bird/Sparrow** the legacy interface (adaptee).
- 3 **BirdAdapter** converts `squeak()` into `makeSound()`; the client works with ToyDuck while a Bird is used internally.

EXAM ANGLE

Prediction

Q1

In the ToyDuck/Bird example, name the target, adaptee and adapter, and say which method call is translated.

Answer: Target: ToyDuck. Adaptee: Bird (Sparrow). Adapter: BirdAdapter, which implements ToyDuck and holds a Bird. Its `squeak()` calls `bird.makeSound()`.

Adapter Design Pattern

➤ Advantages:

- ✓ Helps achieve **reusability and flexibility**.
- ✓ Client class is not complicated by having to use a different interface and can use **polymorphism** to swap between different implementations of adapters.

➤ Disadvantages:

- ✓ All requests are forwarded, so there is a **slight increase in the overhead**.
- ✓ Sometimes many adaptations are required along an **adapter chain** to reach the type which is required.



NOTES

- 1 **Advantage:** reusability and flexibility: existing classes are reused without modification.
- 2 **Advantage:** the client isn't complicated by a different interface and can use **polymorphism** to switch between different adapters.
- 3 **Disadvantage:** every request is forwarded, so there is a **slight overhead**.
- 4 **Disadvantage:** sometimes a **chain** of adapters is needed to reach the required type (like the stacked plug adapters in the photo).

The Composite Design Pattern

- The Composite design pattern is an **object structural pattern** that allows designers to compose tree-like design by strategically **structuring objects** that share a **whole-part relationship**.
- The **intent** of the Composite is to:
 - ✓ Compose objects into **tree structures** to represent **whole-part hierarchies**.
 - ✓ Composite pattern lets clients **treat** individual **objects** and **composites** of objects **uniformly**.

NOTES

- 1 **Composite** is an **object structural** pattern for tree-like designs of objects that share a **whole-part** relationship.
- 2 Intent (1): compose objects into **tree structures** to represent whole-part hierarchies.
- 3 Intent (2): let clients treat individual objects and compositions of objects **uniformly**: the same call works on one item or a whole group.

EXAM ANGLE

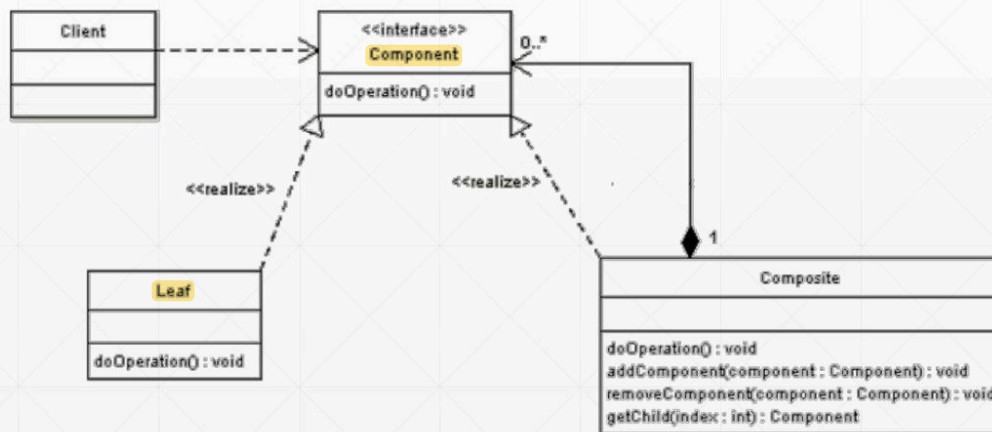
Past exam · Final 252 · Q7-1 · 2.5 marks

Q1

File explorer: Files (leaf, with a size) and Directories (contain files and other folders, nested to any depth). Calculate a folder's total size. Which pattern? Draw its class diagram.

Answer: Pattern: Composite. Treat single files and whole folders the same way through one interface. **FileSystemItem** (component, abstract or interface): `getSize()`. **File** (leaf): returns its own size. **Directory** (composite): holds a list of `FileSystemItems` with `add()` / `remove()`; its `getSize()` sums `child.getSize()` for every child, which recurses into sub-folders to any depth. Diagram: File and Directory both inherit from `FileSystemItem`; Directory has a composition/aggregation to `FileSystemItem` (0..* children).

The Composite Design Pattern – Structure



NOTES

- 1 «interface» **Component** declares the common operation (`doOperation()`). The **Client** depends only on **Component**.
- 2 **Leaf** realises **Component** and does the operation itself; it has no children.
- 3 **Composite** realises **Component** too, and holds `0..*` child **Components** (the filled diamond with multiplicity 1 to `0..*` is composition). It adds `addComponent()`, `removeComponent()` and `getChild(index)`, and its `doOperation()` calls `doOperation()` on each child.

EXAM ANGLE

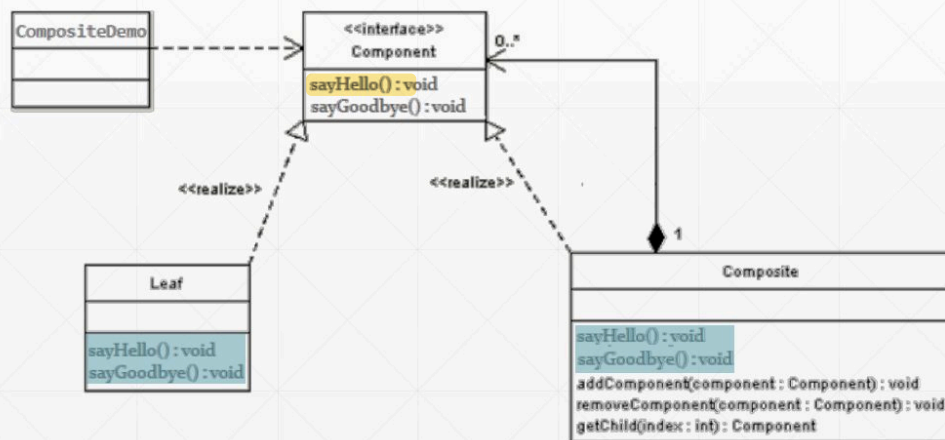
Prediction

Q1

In the Composite pattern, what does the Composite class do when its operation is called?

Answer: It forwards the call to each of its child **Components** (which may be leaves or other composites) and combines the results if needed, so the operation recurses down the whole tree.

The Composite Design Pattern – Example



Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

21

NOTES

- 1 The example: «interface» **Component** with `sayHello()` and `sayGoodbye()`; **Leaf** and **Composite** both realise it; **Composite** holds 0..* **Components**; **CompositeDemo** is the client. The highlighted methods are the ones each class implements.

The Composite Design Pattern – Example*

- The following slides have the Java code that represents the implementation of the classes in the Composite Design Pattern

File: **Component.java**

```

public interface Component {
    public void sayHello();
    public void sayGoodbye();
}
  
```

Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

*source: <https://www.avajava.com/tutorials/lessons/composite-pattern.html?page=1>

22

NOTES

1 Component.java:

CODE

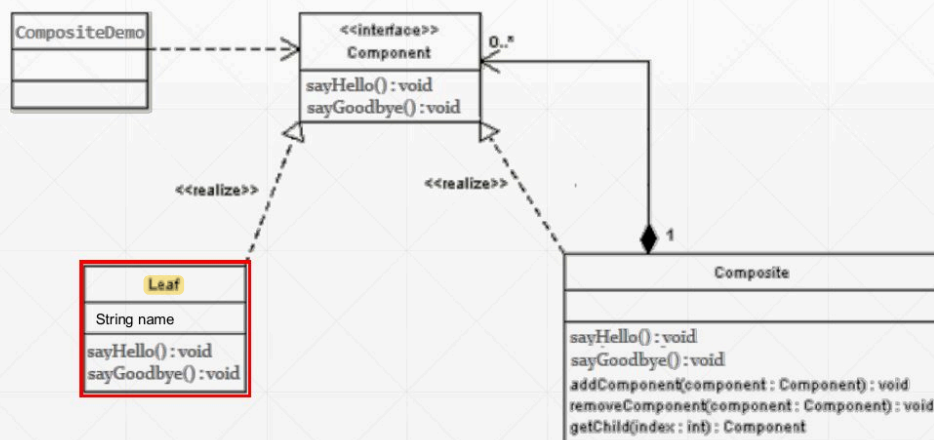
```
public interface Component { public  
void sayHello(); public void  
sayGoodbye(); }
```

WHAT IT DOES

The operations every node, leaf or composite, supports.

SLIDE 21 / 42 The Composite Design Pattern Example

The Composite Design Pattern – Example



NOTES

- 1 The diagram with **Leaf** outlined: it has a `String name` and implements `sayHello()` and `sayGoodbye()`.

The Composite Design Pattern Example

File: **Leaf.java**

```
public class Leaf implements Component {

    String name;

    public Leaf(String name) {
        this.name = name;
    }

    @Override
    public void sayHello() {
        System.out.println(name + " leaf says hello");
    }

    @Override
    public void sayGoodbye() {
        System.out.println(name + " leaf says goodbye");
    }

}
```

NOTES

1 Leaf.java:

CODE

```
public class Leaf implements Component
{ String name;
```

```
    public Leaf(String name) {
        this.name = name; }
}
```

```
    @Override public void sayHello() {
        System.out.println(name + " leaf says
        hello"); }
}
```

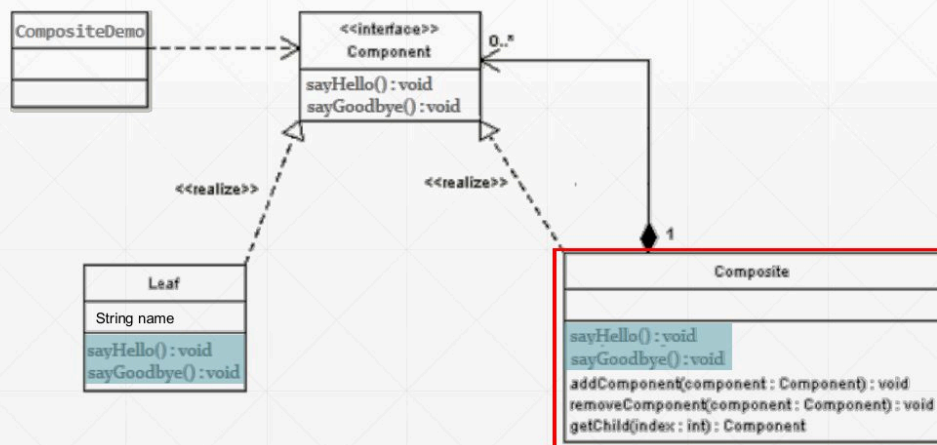
```
    @Override public void sayGoodbye()
{ System.out.println(name + " leaf
  says goodbye"); } }
}
```

WHAT IT DOES

A leaf has its own data and no children.

Does the work itself.

The Composite Design Pattern – Example



NOTES

- 1 The diagram with **Composite** outlined: it implements `sayHello()` and `sayGoodbye()` plus the child-management methods `addComponent()`, `removeComponent()` and `getChild()`.

The Composite Design Pattern Example

File: **Composite.java**

```

import java.util.ArrayList;
import java.util.List;
public class Composite implements Component {
    List<Component> components = new ArrayList<Component>();

    @Override
    public void sayHello() {
        for (Component component : components) {
            component.sayHello();
        }
    }

    @Override
    public void sayGoodbye() {
        for (Component component : components) {
            component.sayGoodbye();
        }
    }

    public void add(Component component) {
        components.add(component);
    }

    public void remove(Component component) {
        components.remove(component);
    }

    public List<Component> getComponents() {
        return components;
    }

    public Component getComponent(int index) {
        return components.get(index);
    }
}
  
```

NOTES

1 Composite.java:

CODE

```
public class Composite implements
Component {
```

WHAT IT DOES

A composite is also a Component, so it can be nested.

```
List<Component> components = new
ArrayList<Component>();
```

Its children: leaves or other composites.

```
@Override public void sayHello() {
for (Component component : components)
{ component.sayHello(); } }
```

Forward the call to every child: this loop is the recursion.

```
@Override public void sayGoodbye()
{ for (...) { component.sayGoodbye(); }
}
```

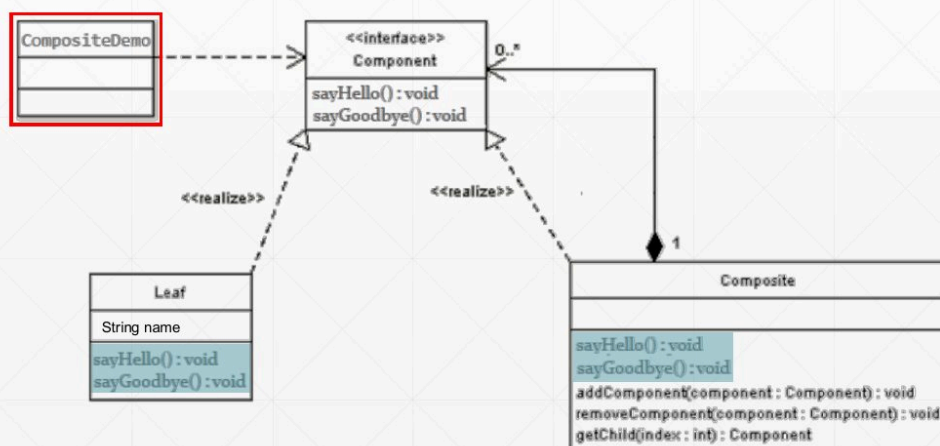
```
public void add(Component
component) {
components.add(component); }
```

Child management.

```
public void remove(Component
component) { ... } getComponents() { ... }
getComponent(int index) { ... } }
```

SLIDE 25 / 42 The Composite Design Pattern Example

The Composite Design Pattern – Example



NOTES

- 1 The diagram with the client **CompositeDemo** outlined: it depends only on Component, though it builds the tree from Leaf and Composite objects.

The Composite Design Pattern – Example

File: **CompositeDemo.java**

```
package com.cakes;

public class CompositeDemo {

    public static void main(String[] args) {

        Leaf leaf1 = new Leaf("Bob");
        Leaf leaf2 = new Leaf("Fred");
        Leaf leaf3 = new Leaf("Sue");
        Leaf leaf4 = new Leaf("Ellen");
        Leaf leaf5 = new Leaf("Joe");

        Composite composite1 = new Composite();
        composite1.add(leaf1);
        composite1.add(leaf2);

        Composite composite2 = new Composite();
        composite2.add(leaf3);
        composite2.add(leaf4);

        Composite composite3 = new Composite();
        composite3.add(composite1);
        composite3.add(composite2);
        composite3.add(leaf5);

        System.out.println("Calling 'sayHello' on leaf1");
        leaf1.sayHello();

        System.out.println("\nCalling 'sayHello' on composite1");
        composite1.sayHello();

        System.out.println("\nCalling 'sayHello' on composite2");
        composite2.sayHello();

        System.out.println("\nCalling 'sayGoodbye' on composite3");
        composite3.sayGoodbye();
    }
}
```

NOTES

1 CompositeDemo.java:

CODE

```
Leaf leaf1 = new Leaf("Bob"); ... leaf5
= new Leaf("Joe");
```

WHAT IT DOES

Five leaves: Bob, Fred, Sue, Ellen, Joe.

```
Composite composite1 = new
Composite(); composite1.add(leaf1);
composite1.add(leaf2);
```

composite1 = {Bob, Fred}.

```
Composite composite2 = new
Composite(); composite2.add(leaf3);
composite2.add(leaf4);
```

composite2 = {Sue, Ellen}.

```
Composite composite3 = new
Composite();
composite3.add(composite1);
composite3.add(composite2);
composite3.add(leaf5);
```

composite3 = {composite1, composite2, Joe}: a tree two levels deep.

```
leaf1.sayHello();
composite1.sayHello();
composite2.sayHello();
composite3.sayGoodbye();
```

The same calls on a leaf and on composites.

The Composite Design Pattern – Example

Output

```
Calling 'sayHello' on leaf1
Bob leaf says hello

Calling 'sayHello' on composite1
Bob leaf says hello
Fred leaf says hello

Calling 'sayHello' on composite2
Sue leaf says hello
Ellen leaf says hello

Calling 'sayGoodbye' on composite3
Bob leaf says goodbye
Fred leaf says goodbye
Sue leaf says goodbye
Ellen leaf says goodbye
Joe leaf says goodbye
```

NOTES

- 1 The output: `sayHello` on leaf1 prints "Bob leaf says hello"; on composite1 prints Bob and Fred; on composite2 prints Sue and Ellen.
- 2 `sayGoodbye` on **composite3** prints goodbye from all five leaves (Bob, Fred, Sue, Ellen, Joe): one call at the root reached every nested leaf, because each composite forwarded it to its children.

The Composite Design Pattern

➤ The steps required to apply the composite design pattern include:

1. Identify, understand, and plan the **tree-like structure** required for the system.
2. Identify and design the **Component** base class:
 - ✓ Includes overridable methods common to both Leaf and Composite objects.
3. **Identify and design the Composite** class, which will:
 - ✓ Overrides common methods in the component interface.
 - ✓ Requires an internal data structure to store objects (nodes) added to the hierarchy.
 - ✓ Includes methods specific for **Composite** objects, which provide capability for adding and removing objects to the internal data structure (i.e. add objects to hierarchy).
4. **Identify and design the Leaf** classes, which overrides methods in the component class
5. Identify and design the **client** that uses both composite and leaf objects.

NOTES

1. Identify and plan the tree-like structure the system needs.
2. Design the **Component** base class with the overridable methods common to leaves and composites.
3. Design the **Composite** class: it overrides the common methods, keeps an internal data structure for its child nodes, and adds methods to add and remove children.
4. Design the **Leaf** classes, which override the component's methods.
5. Design the **client**, which uses both composite and leaf objects.

SLIDE 29 / 42 The Composite Design Pattern

The Composite Design Pattern

➤ Benefits of the Composite design pattern:

- ✓ Provides a design structure that **supports** both composite and primitive objects.
- ✓ **Minimizes complexity on clients** by shielding them from knowing the operational differences between primitive and composite objects.
 - Clients that expect a primitive object will also work with a composite object, since operations are called uniformly on both primitive and composite objects.
- ✓ **Easy to create and add new components** objects to applications.

<https://www.digitalocean.com/community/tutorials/composite-design-pattern-in-java>

NOTES

- 1 It gives one structure that **supports both composite and primitive** (leaf) objects.
- 2 It **minimises complexity for clients**: they don't need to know whether they hold a primitive or a composite, because operations are called the same way on both. A client written for a single object also works with a whole group.
- 3 New component types are **easy to add** without changing client code. (The trade-off: a general Component interface can make it hard to restrict which children a composite may hold.)

Prediction

Q1

Give two benefits of the Composite pattern.

Answer: Clients treat single objects and groups uniformly (simpler client code); it supports both primitive and composite objects in one tree structure; new component types can be added easily.

SLIDE 30 / 42 Extra: Explanation of the Composite pattern

Extra:

➤ Explanation of the Composite Design Pattern

- The **Composite design pattern** is used when we want to **treat individual objects (primitives)** and groups of objects (composites) in the same way.
- It provides a **tree-like structure**, where leaf nodes are simple objects and composite nodes are collections of objects.
- Clients (the user or program that uses the objects) don't need to know whether they're dealing with a single object or a group — they can call the same operations **uniformly**.
- This reduces complexity because the same interface is used for both simple and complex objects.
- It also makes it easier to **add new objects** (either simple or composite) without changing the client code.

NOTES

- 1 Composite is used to **treat individual objects (primitives) and groups (composites) the same way**.
- 2 It gives a **tree**: leaves are simple objects, composite nodes are collections.
- 3 Clients call the same operations **uniformly**, which reduces complexity and makes it easy to add new simple or composite objects without changing client code.

Extra:**Real-Life Example****File System (Folders and Files):**

- A **file** is a primitive object — it doesn't contain other objects.
 - A **folder** is a composite object — it can contain both files and other folders.
 - Whether you open a file or a folder, you're using the same operation (`open`). If it's a file, the content shows directly. If it's a folder, it expands to show its contents.
- 👉 The file explorer treats both **files** (primitives) and **folders** (composites) the same way.

Another Simple Example for Students

Think about a **university course structure**:

- A **single course** (like "Math 101") is a primitive.
- A **program** (like "Computer Science degree") is a composite — it contains many courses.
- Whether you check the **details** of one course or the whole program, you interact with them in a similar way.

33

NOTES

- 1 **File system:** a file is primitive (contains nothing), a folder is composite (contains files and other folders). The same "open" operation shows a file's content or expands a folder: the file explorer treats both the same way. This is exactly the Final 252 question.
- 2 **University:** a single course (Math 101) is primitive; a programme (a CS degree) is composite, containing many courses. Checking the details of one course or a whole programme works the same way.

Facade Design Pattern

- The **Facade** design pattern is an **object structural** pattern that provides a **simplified interface** to complex subsystems.
- The **intent** of the Facade is to:
 - ✓ Provide a **unified interface** to a set of interfaces in a subsystem.
 - ✓ Facade defines a **higher-level interface** that makes the subsystem easier to use.

NOTES

- 1 **Facade** is an **object structural** pattern that provides a **simplified interface** to complex subsystems.
- 2 Intent: provide a **unified interface** to a set of interfaces in a subsystem; the facade defines a **higher-level interface** that makes the subsystem easier to use. Like a hotel reception desk: one place to ask, many departments behind it.

EXAM ANGLE

Prediction

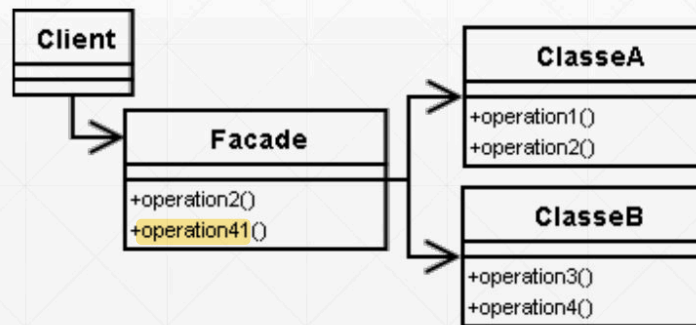
Q1

Providing one simple interface to a set of complex subsystem classes is the:

- A. Facade pattern
- B. Adapter pattern
- C. Composite pattern
- D. Singleton pattern

Answer: A. Facade pattern

Facade Design Pattern - Structure



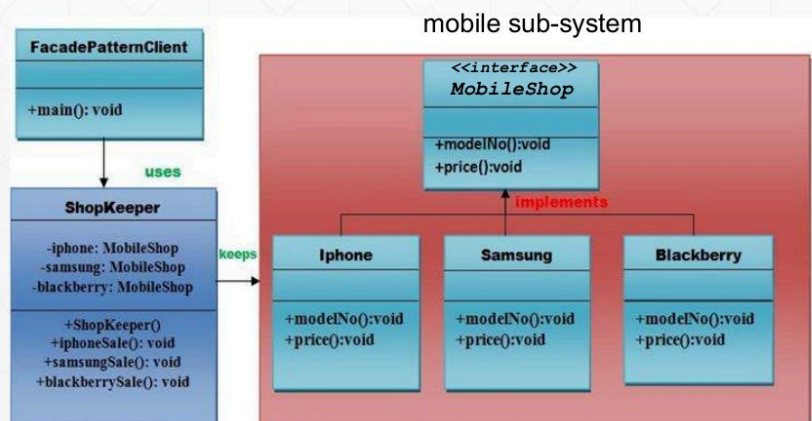
NOTES

- 1 The structure: the **Client** depends only on the **Facade**, whose methods (`operation2()`, `operation41()`) are implemented by calling **ClassA** and **ClassB** in the subsystem (`operation1()` ... `operation4()`). The client never touches ClassA or ClassB directly.

SLIDE 34 / 42 Facade Design Pattern Example*

Facade Design Pattern – Example*

- Consider the following UML example. The client class requires to use the **mobile sub-system** without the need to deal with details of the sub-systems.
- This can be achieved by using a Facade Pattern class (**ShopKeeper**) to allow to client class to use the mobile sub-system easily.



NOTES

- 1 The example (from javatpoint): a **mobile subsystem** with «interface» MobileShop (`modelNo()` , `price()`) implemented by Iphone, Samsung and Blackberry.
- 2 The client (FacadePatternClient) shouldn't deal with those details, so a facade class, **ShopKeeper**, keeps one of each phone and offers simple methods: `iphoneSale()` , `samsungSale()` , `blackberrySale()` .

SLIDE 35 / 42 Facade Example: MobileShop and Iphone

Facade Design Pattern – Example (implementation)

- The following Java code represent the implementation of the classes in the mobile sub-system:

File: *MobileShop.java*

```
public interface MobileShop {  
    public void modelNo();  
    public void price();  
}
```

File: *Iphone.java*

```
public class Iphone implements MobileShop {  
    @Override  
    public void modelNo() {  
        System.out.println(" Iphone 6 ");  
    }  
    @Override  
    public void price() {  
        System.out.println(" Rs 65000.00 ");  
    }  
}
```

NOTES

- 1 The subsystem:

CODE

```
public interface MobileShop { public  
void modelNo(); public void price(); }
```

```
public class Iphone implements  
MobileShop { modelNo() { ..." Iphone 6  
"... } price() { ..." Rs 65000.00 "... } }
```

WHAT IT DOES

The subsystem interface.

One subsystem class.

Facade Design Pattern – Example (implementation)

File: **Samsung.java**

```
public class Samsung implements MobileShop {
    @Override
    public void modelNo() {
        System.out.println(" Samsung galaxy tab 3 ");
    }
    @Override
    public void price() {
        System.out.println(" Rs 45000.00 ");
    }
}
```

File: **Blackberry.java**

```
public class Blackberry implements MobileShop {
    @Override
    public void modelNo() {
        System.out.println(" Blackberry Z10 ");
    }
    @Override
    public void price() {
        System.out.println(" Rs 55000.00 ");
    }
}
```

NOTES

- 1 Samsung prints "Samsung galaxy tab 3" and "Rs 45000.00"; Blackberry prints "Blackberry Z10" and "Rs 55000.00". Same interface, three classes the client would otherwise have to know.

Facade Design Pattern – Example (implementation)

File: **ShopKeeper.java**

```
public class ShopKeeper {
    private MobileShop iphone;
    private MobileShop samsung;
    private MobileShop blackberry;

    public ShopKeeper(){
        iphone= new Iphone();
        samsung=new Samsung();
        blackberry=new Blackberry();
    }

    public void iphoneSale(){
        iphone.modelNo();
        iphone.price();
    }

    public void samsungSale(){
        samsung.modelNo();
        samsung.price();
    }

    public void blackberrySale(){
        blackberry.modelNo();
        blackberry.price();
    }
}
```

NOTES

1 The facade:

CODE

```
public class ShopKeeper { private
MobileShop iphone; private MobileShop
samsung; private MobileShop
blackberry;
```

WHAT IT DOES

It holds the subsystem objects.

```
public ShopKeeper() { iphone = new
Iphone(); samsung = new Samsung();
blackberry = new Blackberry(); }
```

And creates them, so the client doesn't have to.

```
public void iphoneSale() {
iphone.modelNo(); iphone.price(); }
```

One task-level method = the right sequence of subsystem calls.

```
public void samsungSale() { ... }
public void blackberrySale() { ... } }
```

SLIDE 38 / 42 Facade Example: client and output

Facade Design Pattern – Example (implementation)

Output

```
===== Mobile Shop =====
1. IPHONE.
2. SAMSUNG.
3. BLACKBERRY.
4. Exit.
Enter your choice: 1
Iphone 6
Rs 65000.00
```

File: FacadePatternClient.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class FacadePatternClient {
    private static int choice;
    public static void main(String args[]) throws NumberFormatException, IOException{
        do{
            System.out.print("===== Mobile Shop ===== \n");
            System.out.print("1. IPHONE. \n");
            System.out.print("2. SAMSUNG. \n");
            System.out.print("3. BLACKBERRY. \n");
            System.out.print("4. Exit. \n");
            System.out.print("Enter your choice: ");

            BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
            choice=Integer.parseInt(br.readLine());
            ShopKeeper sk=new ShopKeeper();

            switch (choice) {
                case 1:
                    sk.iphoneSale();
                    break;
                case 2:
                    sk.samsungSale();
                    break;
                case 3:
                    sk.blackberrySale();
                    break;
                default:
                    System.out.println("Nothing You purchased");
            }
            return;
        }while(choice!=4);
    }
}
```

Prince Sultan University | Slides prepared from "Software Engineering Design: Theory and Practice" book and slides

42

NOTES

1 The client:

CODE

```
ShopKeeper sk = new ShopKeeper();
```

WHAT IT DOES

The only subsystem-related class it knows.

```
switch (choice) { case 1:
sk.iphoneSale(); break; case 2:
sk.samsungSale(); break; case 3:
sk.blackberrySale(); break; ... }
```

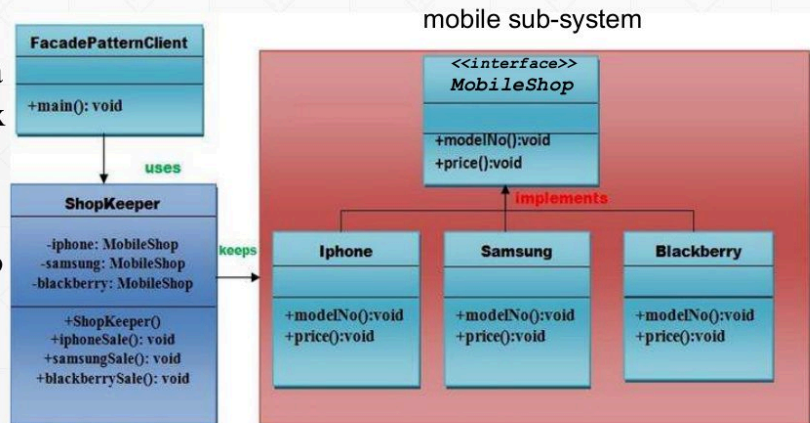
A menu: 1 iPhone, 2 Samsung, 3 Blackberry, 4 Exit.

- 2 Output for choice 1: "Iphone 6" and "Rs 65000.00".

SLIDE 39 / 42 Facade Design Pattern Example*

Facade Design Pattern – Example*

- As noted from the example, the Façade Pattern provided us with a **simplified interface** to a complex subsystems.
- The Facade Pattern class (ShopKeeper) to allow client class to use the mobile sub-system **without knowing anything about the sub-system**.



NOTES

- 1 As the example shows, the facade gave the client a **simplified interface** to a complex subsystem.
- 2 ShopKeeper lets the client use the mobile subsystem **without knowing anything about it**: which classes exist, how they are created, or which methods to call in which order.

Facade Design Pattern

➤ The step-by-step procedure for applying the Facade design pattern include:

1. Identify all **components** involved in carrying out a subsystem operation
2. Create a list of the **operations** required to execute the subsystem operation.
3. **Design a Facade class** that includes an interface method to carry out the subsystem operation.
4. **Implement the Facade** interface method by calling operations on one or more subsystem components, in the list identified in step 2.
5. **Allow** one ore more **clients** to access the objects of the Facade type so that they can gain access to the subsystem operation.

NOTES

1. Identify all **components** involved in carrying out a subsystem operation.
2. List the **operations** required to execute it.
3. **Design a facade class** with an interface method for that subsystem operation.
4. **Implement** that method by calling the operations from step 2 on the subsystem components.
5. Let **clients** use the facade object to reach the subsystem operation.

EXAM ANGLE

Prediction

Q1

True or false: Adapter changes an interface to match what a client expects, while Facade simplifies access to a whole subsystem.

Answer: True

Prediction

Q2

How do Adapter and Facade differ?

Answer: Adapter converts one existing interface into a different interface that a client already expects (making incompatible classes work together). Facade defines a new, simpler, higher-level interface over many subsystem classes so clients don't need to know them. Adapter wraps one class to fit; Facade wraps a subsystem to simplify.

Facade Design Pattern

➤ Benefits of the Facade design pattern include

- ✓ Shields clients from knowing the internals of complex subsystem, therefore minimizing complexity in clients.
- ✓ The facade provides a **stable interface** that hides changes to internal subsystems; therefore, client code is more stable.
- ✓ Promotes **weak coupling** on clients; with facade, clients depend only on one interface instead of multiple interfaces

NOTES

- 1 It **shields clients** from the internals of a complex subsystem, reducing client complexity.
- 2 It provides a **stable interface** that hides internal changes, so client code is more stable.
- 3 It promotes **weak coupling**: clients depend on one interface instead of many. (Keep facades focused on related tasks so they don't become "god objects".)

Summary

- In this session, we presented **structural design patterns**, including:
 - ✓ Adapter
 - ✓ Facade
 - ✓ Composite

- Next ... we will present **behavioral** design patterns in detailed design.

NOTES

- 1 Structural patterns covered: **Adapter** (translate an interface), **Facade** (simplify a subsystem), **Composite** (treat trees uniformly).
- 2 Next: **behavioural** design patterns.

Original slides from the SE322 lecture deck, with study notes. Check exact wording against the [original deck](#).
Notes and questions are study aids. Made by Shoug Alomran.