



Java™ Education & Technology Services

Design Principles and Patterns



Table of Contents

- ☐ **Chapter 1: Design Principles**
- ☐ **Chapter 2: Welcome to Design Patterns**
- ☐ **Chapter 3: Strategy Pattern**
- ☐ **Chapter 4: The Observer Pattern**
- ☐ **Chapter 5: The Decorator Pattern**
- ☐ **Chapter 6: Factory Patterns**
- ☐ **Chapter 7: The Singleton Pattern**
- ☐ **Chapter 8: The Adapter Pattern**
- ☐ **Chapter 9: Template Method Pattern**



Chapter 1

Design Principles



Chapter 1: Design Principles

- ☐ Low level design principles
- ☐ High level design principles



LOW LEVEL DESIGN PRINCIPLES



Low Level Class Design Principles

- ☐ Tell don't ask
- ☐ Once and only once
- ☐ Law of Demeter
- ☐ Favor composition over inheritance
- ☐ Command Query Separation

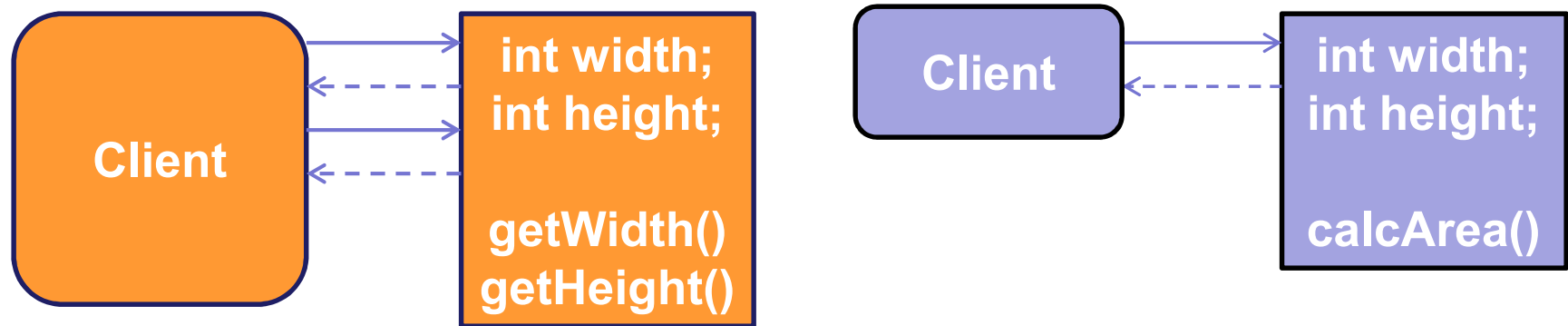


Low Level Class Design Principles

- ☒ Tell don't ask
- ☐ Once and only once
- ☐ Law of Demeter
- ☐ Favor composition over inheritance
- ☐ Command Query Separation

❑ Tell don't ask

- Instead of asking an object for data and acting on that data, tell an object what to do.
- Object Orientation bundles data with behavior (functions acting on that data) in one class.



- **Note that:** co-locating data and behavior should sometimes be dropped in favor of other concerns such as layering.



Low Level Class Design Principles

- ☐ Tell don't ask
- ☒ Once and only once
- ☐ Law of Demeter
- ☐ Favor composition over inheritance
- ☐ Command Query Separation



□ Once and only once Don't Repeat Yourself (DRY)

- Say anything in your program only once.
- If you use a hard-coded value more than one time make them public final constant.
- When two blocks of code are the same. You define a subroutine and call it from both places, so you change in one place later.
- If similar but not Identical: parameterize the subroutine.
- If 2 routines have the same flow but differ in the actual steps, use Interface and Abstraction for common behavior (Template Method Design Pattern)



□ Once and only once Example

Tea class

```
public class Tea{  
    public void prepare ( )  
    {  
        boilWater();  
        putTeaBag();  
        addSugar();  
    }  
    void boilWater ( ) { ... }  
    void putTeaBag() {....}  
    void addSugar ( ) { ... }  
}
```

Coffee class

```
public class Coffee {  
    public void prepare ( )  
    {  
        boilWater();  
        putCoffe ();  
        addSugar();  
    }  
    void boilWater ( ) { ... }  
    void putCoffe() {....}  
    void addSugar ( ) { ... }  
}
```



□ Once and only once Suggested Solution

Drink class

```
public class Drink {  
    public void prepare ( )  
    {  
        boilWater();  
        putIngredient ();  
        addSugar();  
    }  
    void boilWater ( ) { ... }  
    void addSugar ( ) { ... }  
    abstract void putIngredient ();  
}
```

Tea class

```
public class Tea {  
    void putIngredient() {...}  
}
```

Coffee class

```
public class Coffee {  
    void putIngredient() {...}  
}
```



Low Level Class Design Principles

- ☐ Tell don't ask
- ☐ Once and only once
- ☒ Law of Demeter
- ☐ Favor composition over inheritance
- ☐ Command Query Separation



❑ Law of Demeter

- Only talk to your immediate friends
- Avoid method chains
`objectA.getObjectB().getObjectC().doSomething()`
;
- Why?
 - Your classes will be "loosely coupled"; your dependencies are reduced.
 - Reusing your classes will be easier.
 - Your classes are less subject to changes in other classes.
 - Your code will be easier to test.



❑ Law of Demeter (cont.)

- Alternatives:
 - require that Object C be passed in to your Java method instead of objectA
 - Or create wrapper methods that pass your request on to a delegate.



❑ Law of Demeter (cont.)

- Alternatives (cont.): Delegate Example

```
class RealPrinter { // the "delegate"
    void print() {
        System.out.println("something");    }
}

class Printer { // the "delegator"
    RealPrinter p = new RealPrinter(); // create the delegate
    void print() {
        p.print(); // delegation }
}

public class Main { // to the outside world it looks like Printer actually prints.
    public static void main(String[] args) {
        Printer printer = new Printer();
        printer.print(); } }
```




❑ Law of Demeter (Least Knowledge)

- A method *m* of an object *O* may only invoke the methods of the following kinds of objects
 1. *O* itself
 2. *m*'s parameter
 3. Any objects created within *m*
 4. *O*'s direct component objects
 5. Global variables accessible by *O* in the scope of *m*



Low Level Class Design Principles

- ☐ Tell don't ask
- ☐ Once and only once
- ☐ Law of Demeter
- ☒ Favor composition over inheritance
- ☐ Command Query Separation



❑ **Favor composition over inheritance**

- When extending a class, you only get facilities which are available at compile time while having private members of the classes you want gives you the flexibility to replace implementation of Composed classes with better versions in runtime.
- Composition offers better testability of a class than Inheritance (e.g. using Mock object representing composed class)



Favor composition over inheritance

- Strategy and Decorator patterns are good example of using composition over inheritance.
- Inheritance can make your code fragile. If sub class is depending on super class behavior for its operation, when the behavior of super class changes, functionality in sub class may get broken.



Favor composition over inheritance

Example

Using Inheritance

```
public class EncryptedHashSet extends HashSet {  
    ....  
    public boolean add (Object o) {  
        return super.add ( encrypt (o) );  
    }  
}
```

Using Composition

```
public class EncryptedHashSet implements Set {  
  
    private HashSet container;  
  
    public boolean add ( Object o ) {  
        return container.add ( encrypt ( o ) );  
    }  
    public boolean addAll ( Collection c ) {  
        return container.add ( encrypt ( c ) );  
    } ..... }  
}
```



Low Level Class Design Principles

- ☐ Tell don't ask
- ☐ Once and only once
- ☐ Law of Demeter
- ☐ Favor composition over inheritance
- ☒ Command Query Separation



❑ Command Query Separation (CQS)

- The fundamental idea is that we should divide an object's methods into two sharply separated categories:
 - **Queries:** Return a result and do not change the observable state of the system (are free of side effects).
 - **Commands / Modifiers:** Change the state of a system but do not return a value.

Command Query Separation (CQS)

Using Inheritance

```
public interface BookService {  
    public void updateBook ( Book book );  
    public Book getBook ( int bookId );  
}
```



- Think of a common data structure that violates that principal.



HIGH LEVEL DESIGN PRINCIPLES (SOLID PRINCIPLES)



High Level Class Design Principles

- **S**ingle Responsibility Principle **SRP**
- **O**pen Closed Principle **OCP**
- **L**iskov Substitution Principle **LSP**
- **I**nterface Segregation Principle **ISP**
- **D**ependency Inversion Principle **DIP**



High Level Class Design Principles

By Robert Martin (Uncle Bob)

- **Single Responsibility Principle SRP**
- Open Closed Principle OCP
- Liskov Substitution Principle LSP
- Interface Segregation Principle ISP
- Dependency Inversion Principle DIP



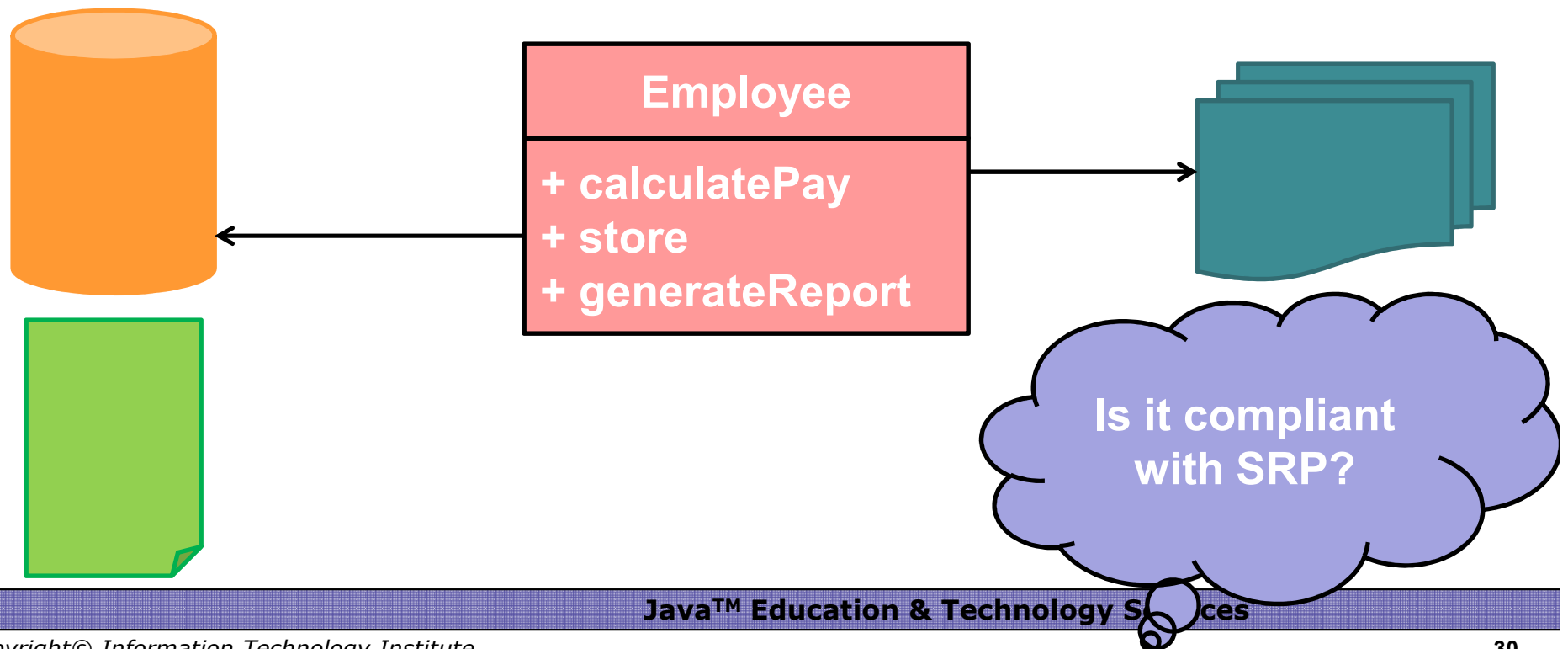
Single Responsibility Principle SRP

- A class should only one reason to change
- Defined by Robert C. Martin in his book Agile Software Development, Principles, Patterns, and Practices



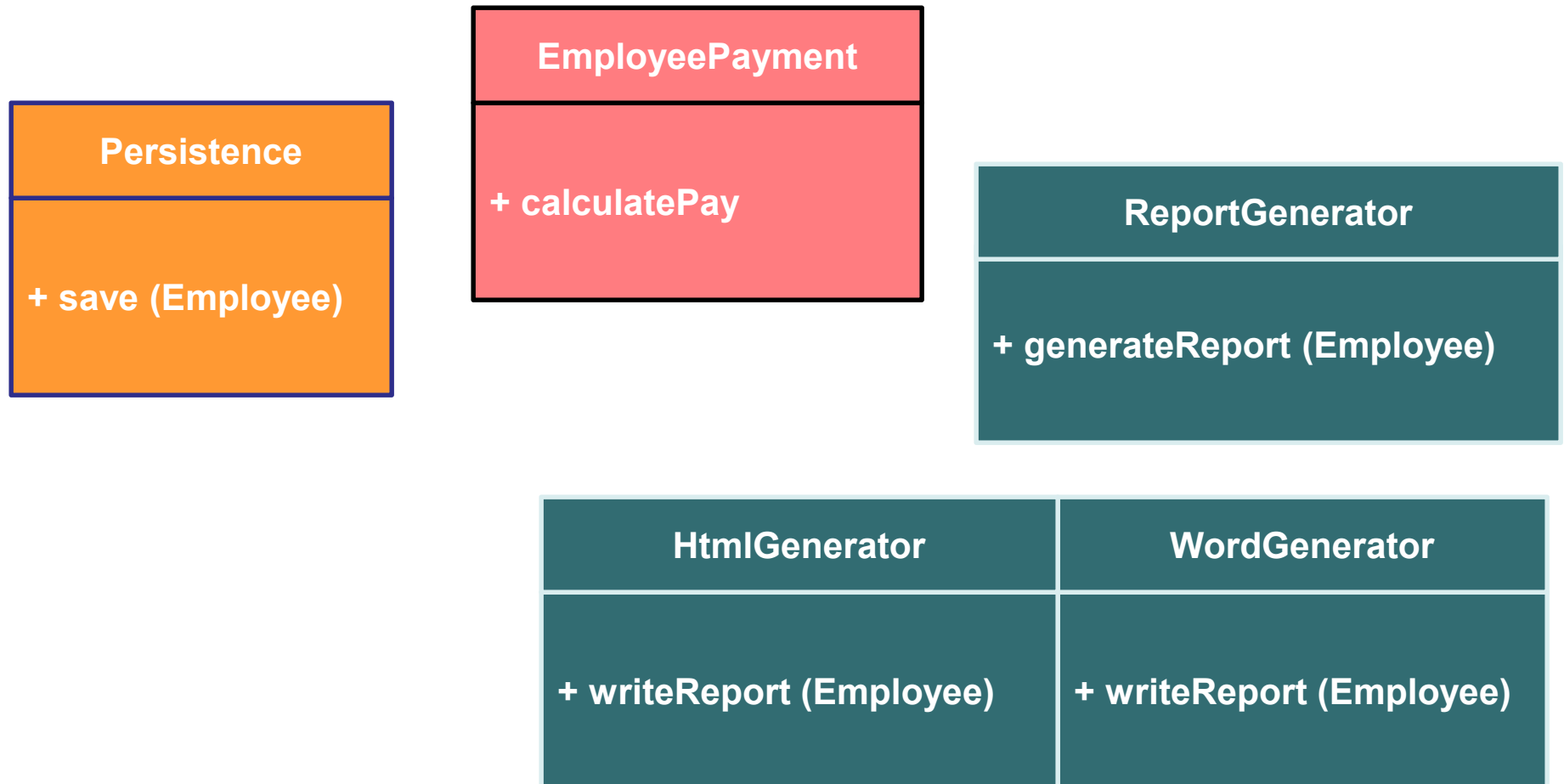
Single Responsibility Principle SRP

- The Employee class contains business rules, report writing and persistence control





Single Responsibility Principle SRP





Single Responsibility Principle SRP

Pros:

- Low coupling
- Light dependency
- Cohesion
- Flexibility to change

Cons: (Excessive use)

- Scattered design , hard to understand code



Single Responsibility Principle SRP

- This is the reason we do not put SQL in JSPs.
- This is the reason we do not generate HTML in the modules that compute results.
- This is the reason that business rules should not know the database schema.
- This is the reason we separate concerns.

*Gather together the things that change for the same reasons.
Separate those things that change for different reasons.*

Source:

<http://blog.8thlight.com/uncle-bob/2014/05/08/SingleResponsibilityPrinciple.html>



High Level Class Design Principles

- Single Responsibility Principle SRP
- **Open Closed Principle OCP**
- Liskov Substitution Principle LSP
- Interface Segregation Principle ISP
- Dependency Inversion Principle DIP

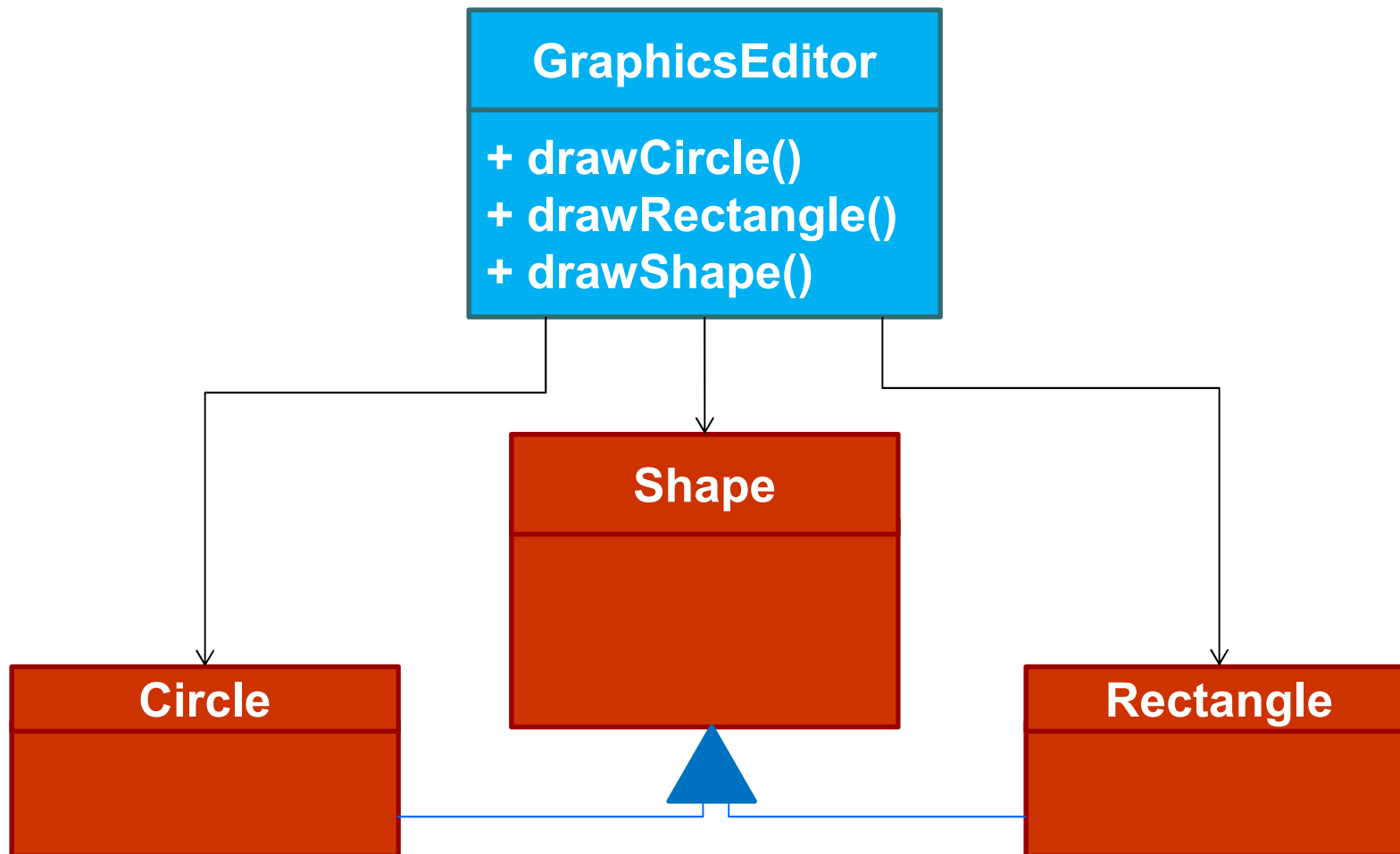


Open Closed Principle OCP

- You should be able to extend the behavior of a system without having to modify that system.
- Plugin systems are the ultimate example.

Open Closed Principle OCP

Example



Source: <http://www.oodeesign.com/open-close-principle.html>



Open Closed Principle OCP - Example

GraphicEditor

```
// Open-Close Principle - Bad example
class GraphicEditor {

    public void drawShape (Shape s) {
        if (s.type==1)
            drawRectangle(s);
        else if (s.type==2)
            drawCircle(s);
    }
    public void drawCircle (Circle r) {...}
    public void drawRectangle (Rectangle r)
    {...}
}
```

Shape

```
class Shape {
    int type;
}
```

Rectangle

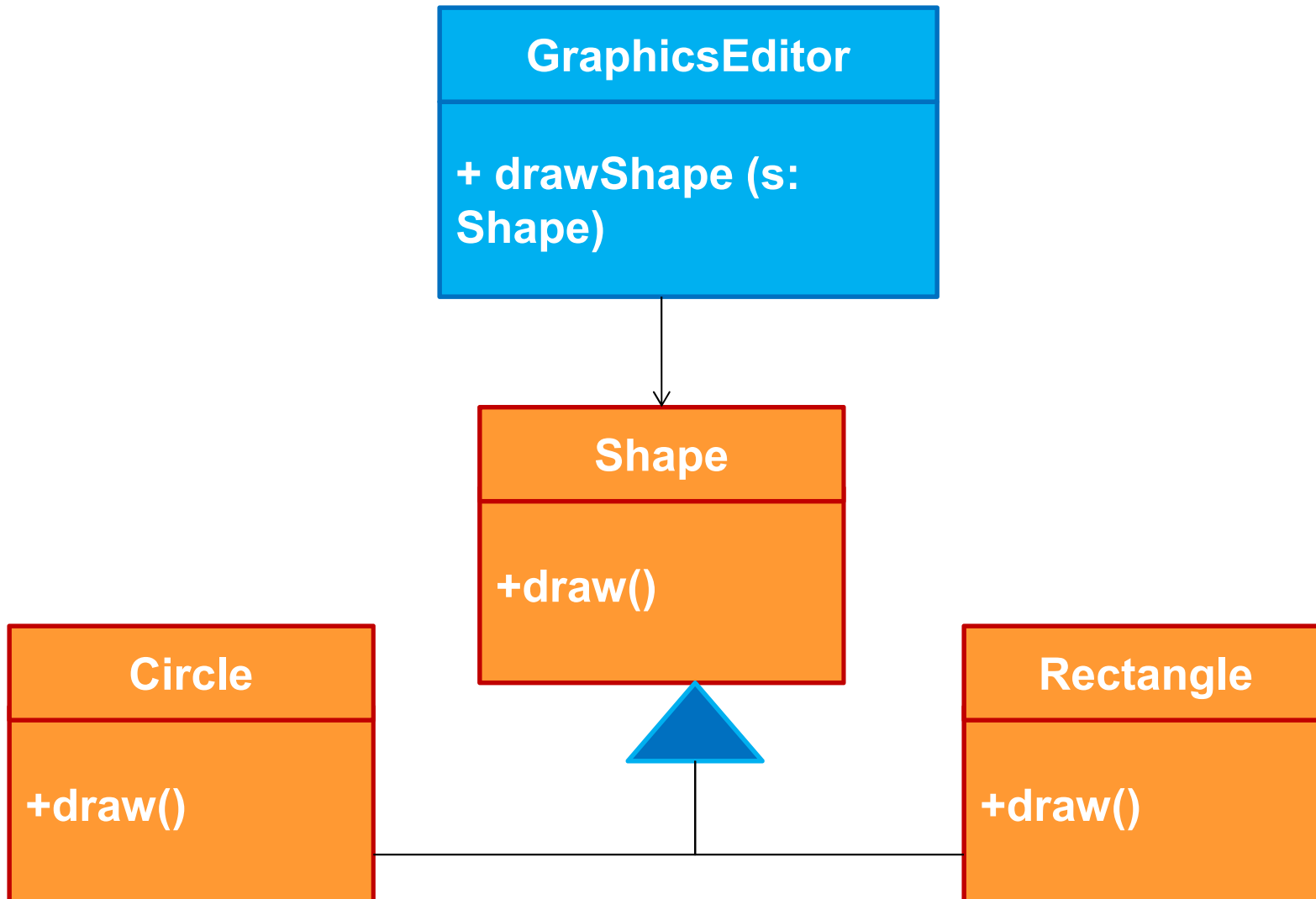
```
class Rectangle extends
Shape {
    Rectangle() {
        super.type=1;
    }
}
```

Circle

```
class Circle extends Shape
{
    Circle() {
        super.type=2;
    }
}
```

Open Closed Principle OCP

Suggested Solution





Single Responsibility Principle SRP

Suggested Solution

GraphicEditor

```
// Open-Close Principle – Good example
class GraphicEditor {

    public void drawShape(Shape s) {
        s.draw();
    }
}
```

Shape

```
class Shape {
    abstract void draw();
}
```

Rectangle

```
class Rectangle extends
Shape {
    public void draw() {
        // draw the rectangle
    }
}
```

Circle

```
class Circle extends Shape {
    public void draw() {
        // draw the circle
    }
}
```

Which design
patterns implement
this principle?



High Level Class Design Principles

- Single Responsibility Principle SRP
- Open Closed Principle OCP
- **Liskov Substitution Principle LSP**
- Interface Segregation Principle ISP
- Dependency Inversion Principle DIP



Liskov Substitution Principle LSP

- The concept of this principle was introduced by Barbara Liskov in a 1987 conference keynote.
- Derived classes must be substitutable for their base classes.
- The reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.
- A subclass should override the parent class' methods in a way that does not break functionality from a client's point of view.



Liskov Substitution Principle LSP Example

Rectangle

// Violation of Likov's Substitution Principle

```
class Rectangle
```

```
{
```

```
    protected int m_width;  
    protected int m_height;
```

```
    setters and getters
```

```
    public int getArea() {  
        return m_width * m_height;  
    }
```

```
}
```

Square

```
class Square extends Rectangle
```

```
{
```

```
    public void setWidth(int width) {  
        m_width = width;  
        m_height = width;
```

```
    }
```

```
    public void setHeight(int height) {  
        m_width = height;  
        m_height = height;
```

```
    }
```

```
}
```



Liskov Substitution Principle LSP Example

```
class LspTest
{
    private static Rectangle getNewRectangle() {
        // it can be an object returned by some factory ...
        return new Square();
    }
    public static void main (String args[]) {
        Rectangle r = LspTest.getNewRectangle();
        r.setWidth(5);
        r.setHeight(10); // user knows that r it's a rectangle.
        System.out.println(r.getArea());
        // now he's surprised to see that the area is 100 instead of 50.
    }
}
```



High Level Class Design Principles

- Single Responsibility Principle SRP
- Open Closed Principle OCP
- Liskov Substitution Principle LSP
- **Interface Segregation Principle ISP**
- Dependency Inversion Principle DIP



Interface Segregation Principle ISP

- Clients should not be forced to depend upon interfaces that they don't use.
- ISP teaches us to respect our clients needs.
- This principle deals with the disadvantages of “fat” interfaces or “interface pollution”.
- If every time a derivative needs a new interface, that interface will be added to the base class. This will further pollute the interface of the base class, making it “fat”.



Interface Segregation Principle ISP

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



Interface Segregation Principle ISP

Example (violates ISP)

Animal

```
public interface Animal
{
    void fly();
    void swim();
    void walk();
}
```

Dolphin

```
public class Dolphin implements Animal
{
    public void Ffy ()
    {
        throw new UnsupportedOperationException();
    }

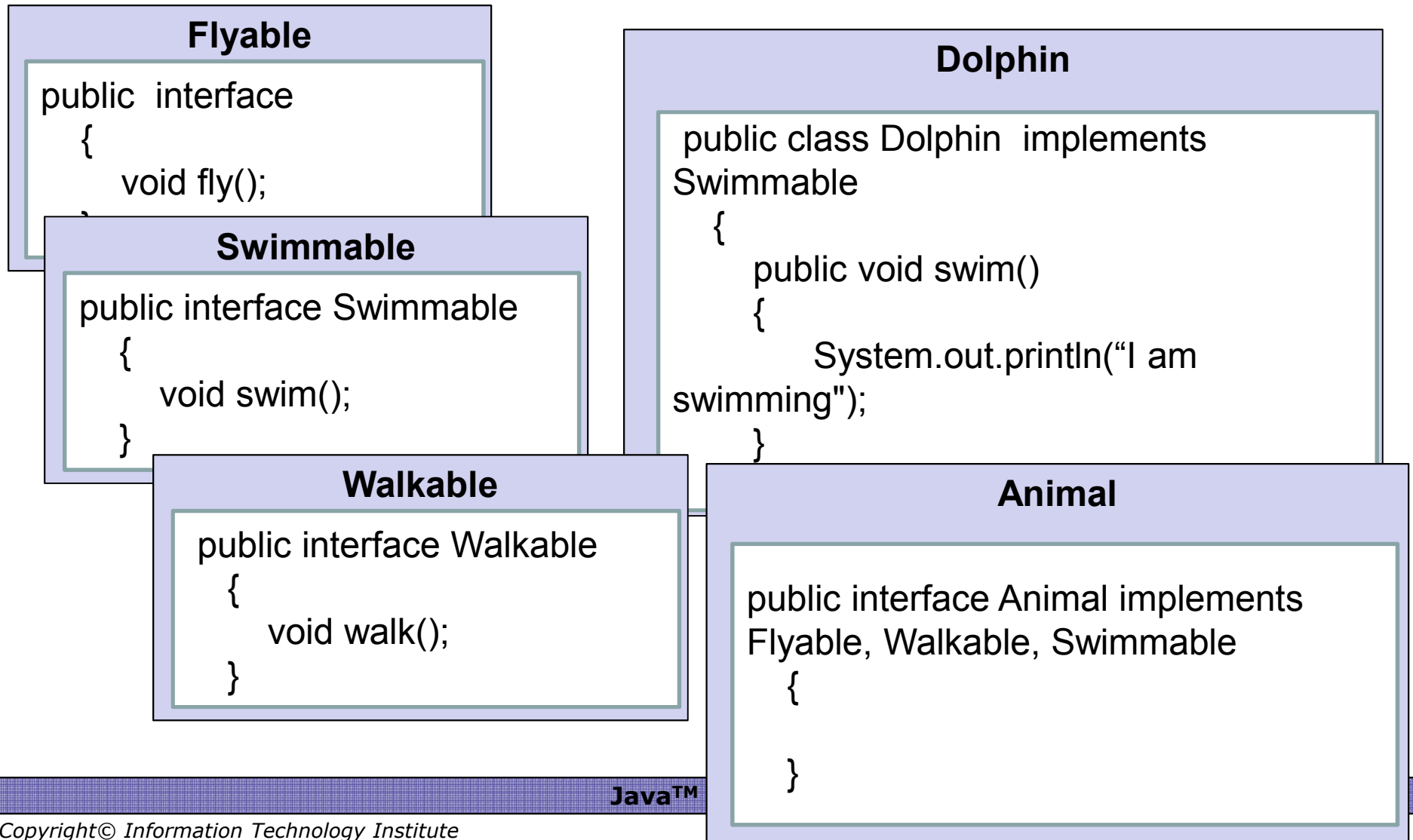
    public void swim()
    {
        System.out.println (" I am Swimming code");
    }

    public void walk()
    {
        throw new UnsupportedOperationException();
    }
}
```



Interface Segregation Principle ISP

Example (ISP)





High Level Class Design Principles

- Single Responsibility Principle SRP
- Open Closed Principle OCP
- Liskov Substitution Principle LSP
- Interface Segregation Principle ISP
- **Dependency Inversion Principle DIP**



Dependency Inversion Principle DIP

- A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

- B. Abstractions should not depend upon details. Details should depend upon abstractions.



Dependency Inversion Principle DIP

Example (violates DIP)

Copier

```
public class Copier
{
    void copy ( KeyBoardReader
reader, PrintWriter writer ) {

        // complex code
    }
```

KeyBoardReader

```
public class KeyBoardReader
{
    String read (....) {
        // reading code
    }
}
```

PrintWriter

```
public class PrintWriter
{
    void write (String text)
    {

        // writing code
    }
}
```



Dependency Inversion Principle DIP

Example (suggested solution)

Copier

```
public class Copier
{
    void copy (Reader reader,
    Writer writer ) {

        // complex code

    }
```

Reader

```
public interface Reader
{
    String read (....) ;
}
```

Writer

```
public interface Writer
{
    void write (String text);

}
```

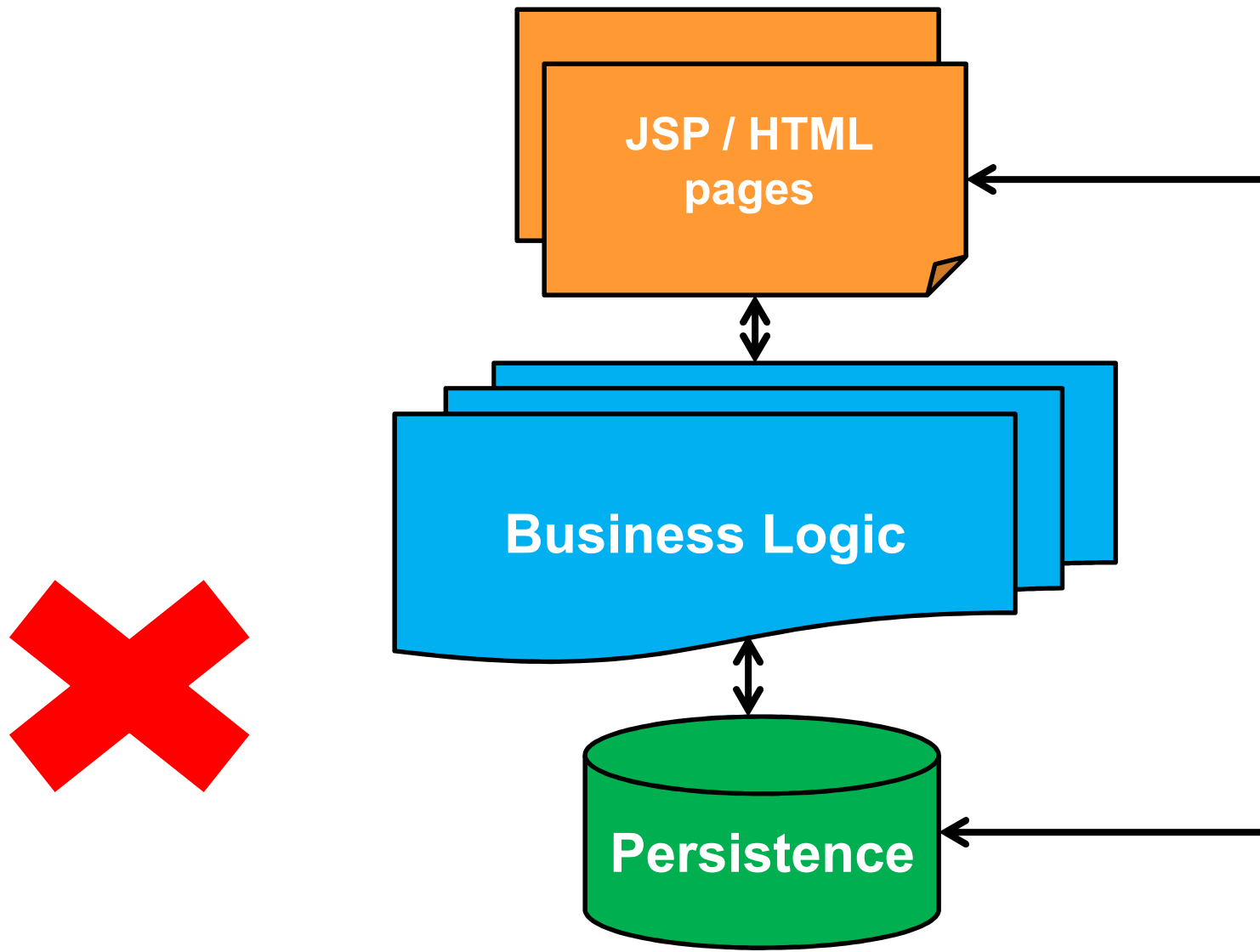


Dependency Inversion Principle DIP

- When this principle is applied it means the high level classes are not working directly with low level classes, they are using interfaces as an abstract layer.
- In this case instantiation of new low level objects inside the high level classes(if necessary) can not be done using the operator new. Instead, some of the Creational design patterns can be used, such as Factory Method, Abstract Factory, Prototype.
- The Template Design Pattern is an example where the DIP principle is applied.



Dependency Inversion Principle DIP





Dependency Inversion Principle DIP

The Dependency Inversion Principle is one that leads or helps us respect all the other principles. Respecting DIP will help us respect all the other principles:

- Almost force you into respecting OCP.
- Allow you to separate responsibilities.
- Make you correctly use subtyping.
- Offer you the opportunity to segregate your interfaces.



Summary of SOLID principles

SRP	<u>The Single Responsibility Principle</u>	A class should have one, and only one, reason to change.
OCP	<u>The Open Closed Principle</u>	You should be able to extend a classes behavior, without modifying it.
LSP	<u>The Liskov Substitution Principle</u>	Derived classes must be substitutable for their base classes.
ISP	<u>The Interface Segregation Principle</u>	Clients should not be forced to depend upon interfaces that they don't use.
DIP	<u>The Dependency Inversion Principle</u>	Depend on abstractions, not on concretions.

Source: <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>



Chapter 2

Welcome to Design Patterns



Chapter 2 Outline

- ☐ What is Design Pattern?
- ☐ History of Design Patterns.
- ☐ Why you need Design Patterns?
- ☐ What do you need to start?



What is Design Pattern?

- In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design.
- “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 using it the same way twice.”

(Christopher Alexander)

- “Reusable solutions to recurring problems that we encounter during software development.”

(Mark Grand - author of **Patterns in Java**)



History of Design Patterns

- Patterns originated as an **architectural** concept by **Christopher Alexander** in 1979.
- In 1987, **Kent Beck** and **Ward Cunningham** began experimenting with the idea of applying patterns to programming.
- In 1994, **Erich Gamma**, **Richard Helm**, **Ralph Johnson**, and **John Vlissides** (the Gang of Four, or **GoF**) published the **Design patterns: Elements of Reusable Object-Oriented Software** book.



History of Design Patterns (cont')

- The GoF book describes a pattern using the following four attributes:
 - The **name** to describes the pattern in a word or two
 - The **problem/intent** describes when to apply the pattern
 - The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations
 - The **consequences** are the results and trade-offs in applying the pattern



Why you need Design Patterns?

- To have **shared vocabulary** with other developers to say what you want in a precise, clear and short way.

*Pattern = set of qualities,
characteristics and constrains.*

- To keep the discussion at the **design level** without diving down to the **implementation details**.
- To minimize the **misunderstanding** between the development team and improve **code readability** for coders and architects .
- To **speed up** the development process by providing tested, proven development paradigms.

What do you need to start?

Object Oriented Basics

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

Design Principles

- Low Level
- SOLID

Design Patterns

Goal

To create flexible designs that are **maintainable** and cope with **changes**



Chapter 3

The Strategy Pattern



Chapter 3 Outline

- ☐ Problem
- ☐ Solution
- ☐ Class Diagram
- ☐ Definition of Strategy Design Pattern
- ☐ Why Strategy Design Pattern

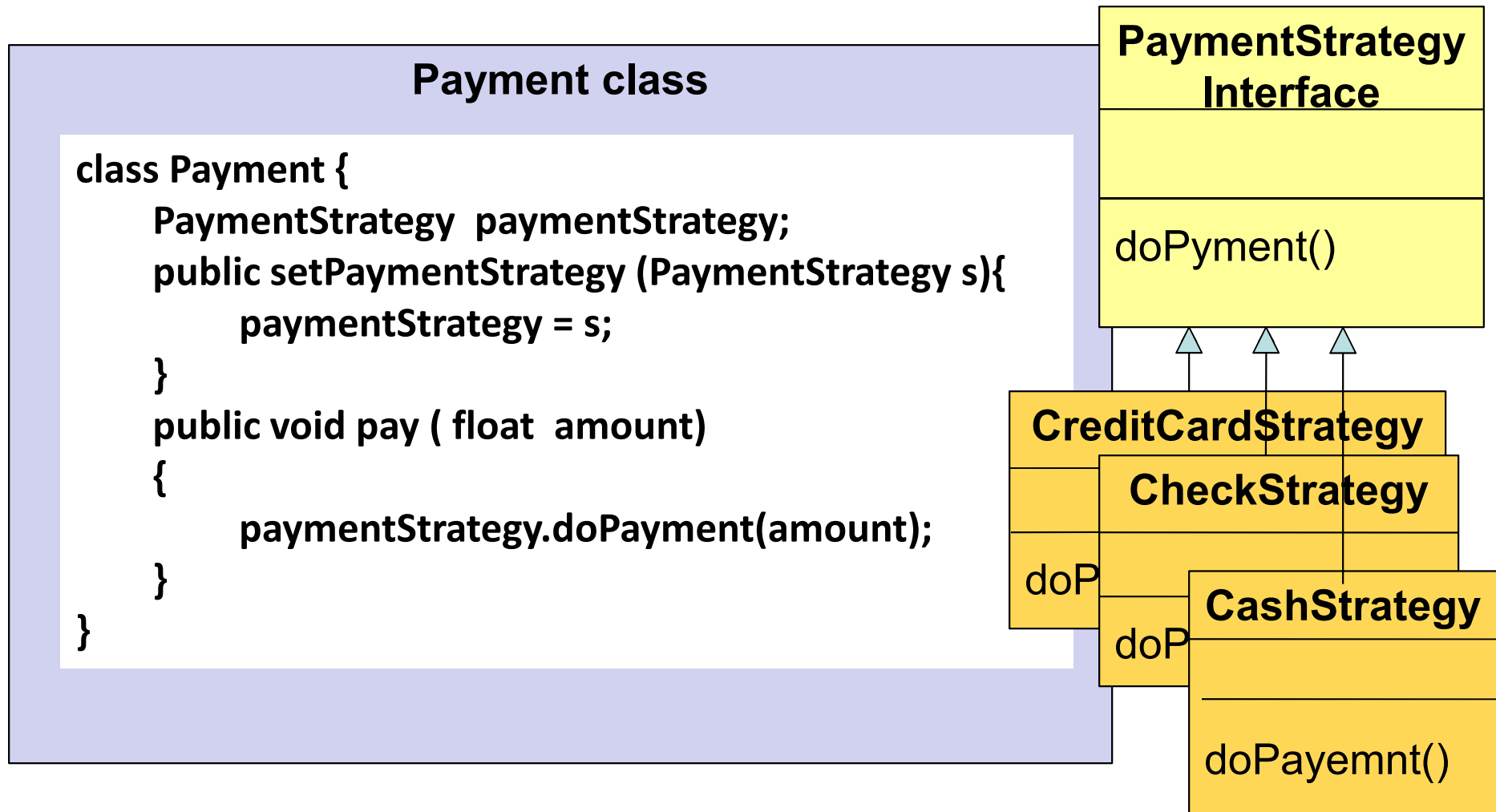
Problem

Payment class

```
class Payment {  
    public void pay( float amount)  
    {  
        if (type.equals (CREDIT_CARD)  
            payUsingCreditCard (amount);  
  
        else if ( type.equals (CHECK)  
            payUsingCheck (amount);  
  
        else  
            payInCash(amount);  
    }  
}
```

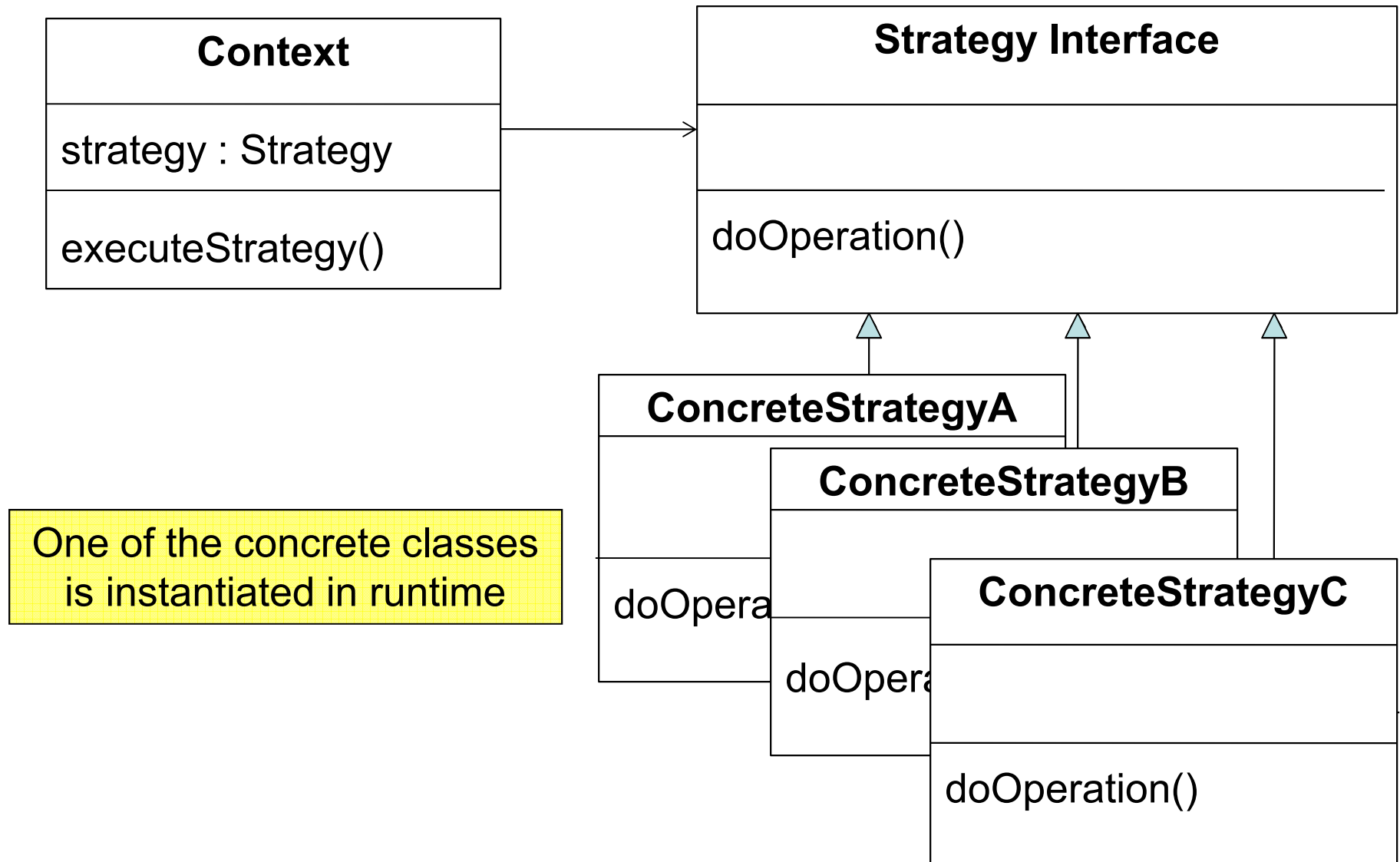
**What do
you think
of this
solution?**

Solution





Class Diagram of Strategy Pattern





Strategy Design Pattern Definition

The strategy pattern

- defines a family of algorithms,
- encapsulates each algorithm, and
- makes the algorithms interchangeable within that family.
- Strategy lets the algorithm vary independently from clients that use it.
- It is also known as Policy Pattern.



Why Strategy Design Pattern

- Strategy pattern is useful when we have different variants of an algorithm or multiple algorithms for a specific task.
- It enables the client to choose any of the algorithms at runtime without knowing the details of each.
- It reduces client code complexity by avoiding the multiple nested conditions.



Chapter 4

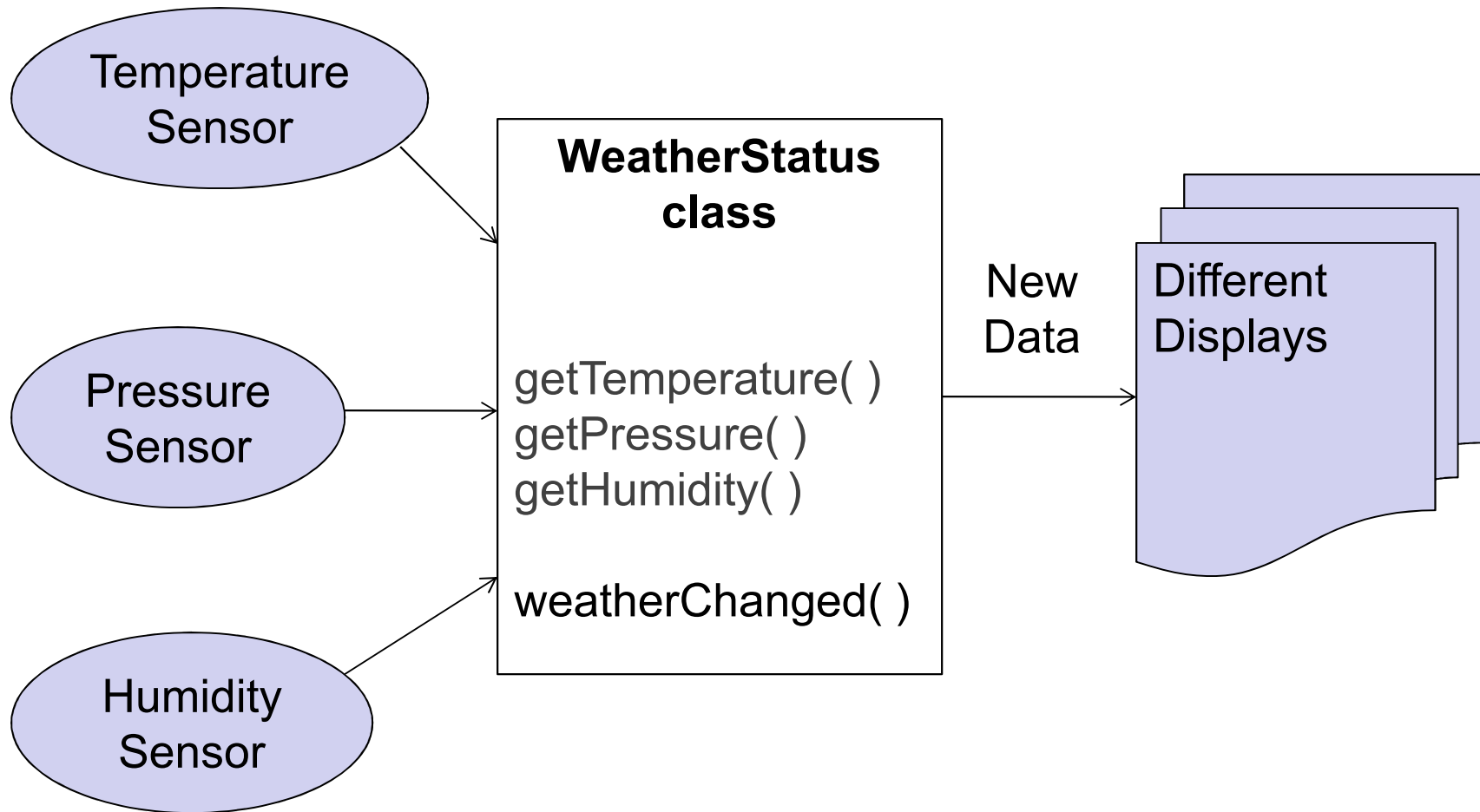
The Observer Pattern



Chapter 4 Outline

- ☐ **Case Study**
- ☐ **Problem**
- ☐ **Suggested Solution**
- ☐ **Definition of Observer Design Pattern.**
- ☐ **Entities of Observer Design Pattern**
- ☐ **Class Diagram of Observer Design Pattern.**
- ☐ **How to Apply Observer Pattern**
- ☐ **Why Observer Pattern?**
- ☐ **Notes on Observer Design Pattern.**
- ☐ **Observer in JDK.**

Case Study





Problem

- The WeatherStatus class has 3 getters for the 3 measurement values
- The weatherChanged method is called each time one of the measurements changes.
- Your role is to implement the different displays (Weather today, Statistics, etc...).
- The System must be expandable (new displays can be added or removed)

Suggested Solution

WeatherStatus class

```
public void weatherChanged( )
{
    float temp=getTemperature();
    float pressure=getPressure();
    float humidity=getHumidity();

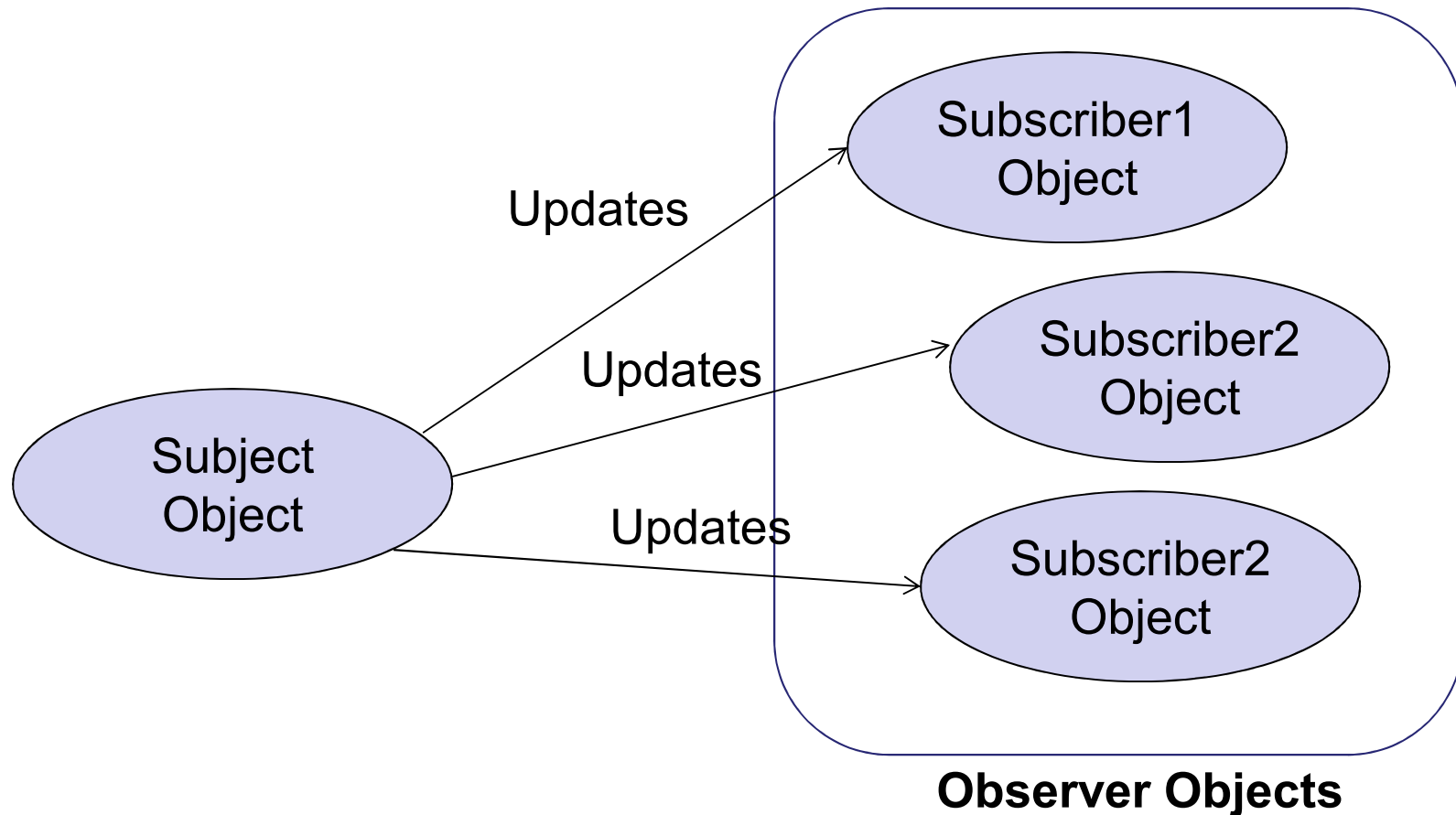
    display1.update(temp,pressure,humidity);

    display2.update(temp,pressure,humidity);

    display3.update(temp,pressure,humidity);
}
```

**What do
you think
of this
solution?**

Observer Design Pattern





Observer Design Pattern Definition

- The Observer Design Pattern is a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.
- Event Driven Architecture

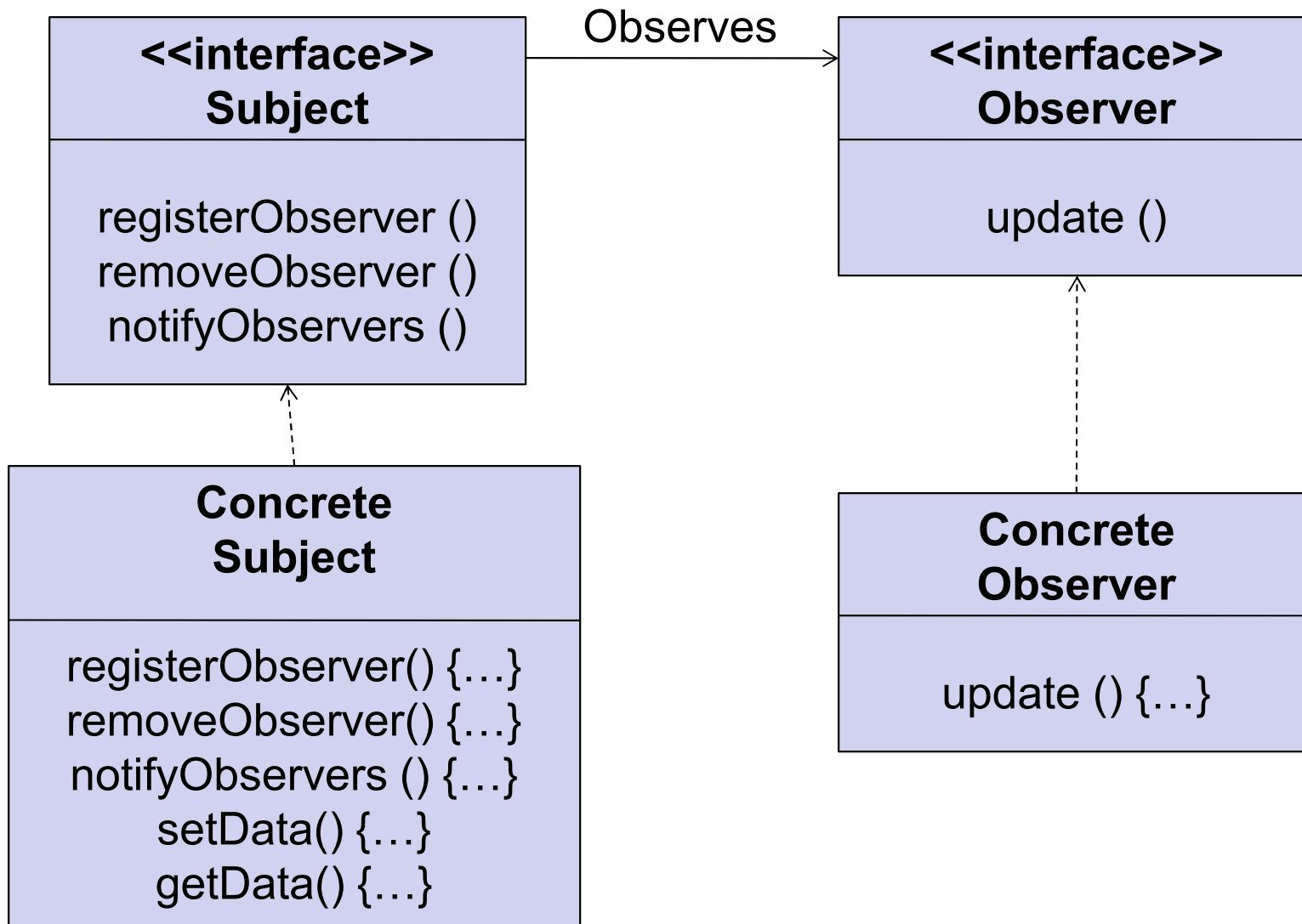


Observer Design Pattern Entities

- **Subject:** The object that carries the data (state) and controls it
- **Observers:** The objects that use the data (e.g. display it) and depend on the **Subject** to tell them when its state changes.
- Each of these **observers** registers its interest in the data by calling a public method in the **Subject** (e.g. registerObserver).
- Each **observer** has a known interface that the **Subject** calls when the data change (e.g. update).



Observer Design Pattern Class Diagram






How to apply Observer Design Pattern?

```
public interface Subject
{
    public void registerObserver (Observer o);
    public void removeObserver (Observer o);
    public void notifyObservers ();
}
```

```
public interface Observer
{
    public void update (float temp, float pressure, float humidity);
}
```



How to apply Observer Design Pattern?(cont.)

```
public class WeatherStatus implements Subject
{
    private ArrayList observers; 
    private float temperature;
    private float humidity;
    private float pressure;
    } state (data)

    public WeatherStatus() {
        observers = new ArrayList();
    }

    public void registerObserver (Observer o) {
        observers.add(o);
    }

    public void removeObserver (Observer o) {
        observers.remove(o);
    }
}
```





How to apply Observer Design Pattern?(cont.)

```
public void notifyObservers() {  
    for (int i = 0; i < observers.size(); i++) {  
        Observer observer = (Observer) observers.get(i);  
        observer. update (temperature, humidity, pressure);  
    }  
}  
  
// This method is called when any of the measurements changes  
public void measurementsChanged() {  
    notifyObservers();  
}  
  
// other WeatherStatus getters and setters methods here  
}
```

How to apply Observer Design Pattern?(cont.)

```
public class TemperatureTodayDisplay implements Observer {
    private float temperature;
    private Subject weatherStatus;

    public TemperatureTodayDisplay (Subject weatherStatus) {
        this. weatherStatus = weatherStatus;
        weatherStatus.registerObserver (this); 
    }

    public void update (float temperature, float humidity, float pressure)
    {
        this.temperature = temperature;
        System.out.println("Current conditions: " + temperature
            + "C degrees ");
    }
}
```



Why Observer Pattern?

- Don't miss out when something interesting happens.
- Keep your Objects in the know when something they may care about happens.
- One of the most used patterns in the JDK.



Chapter 5

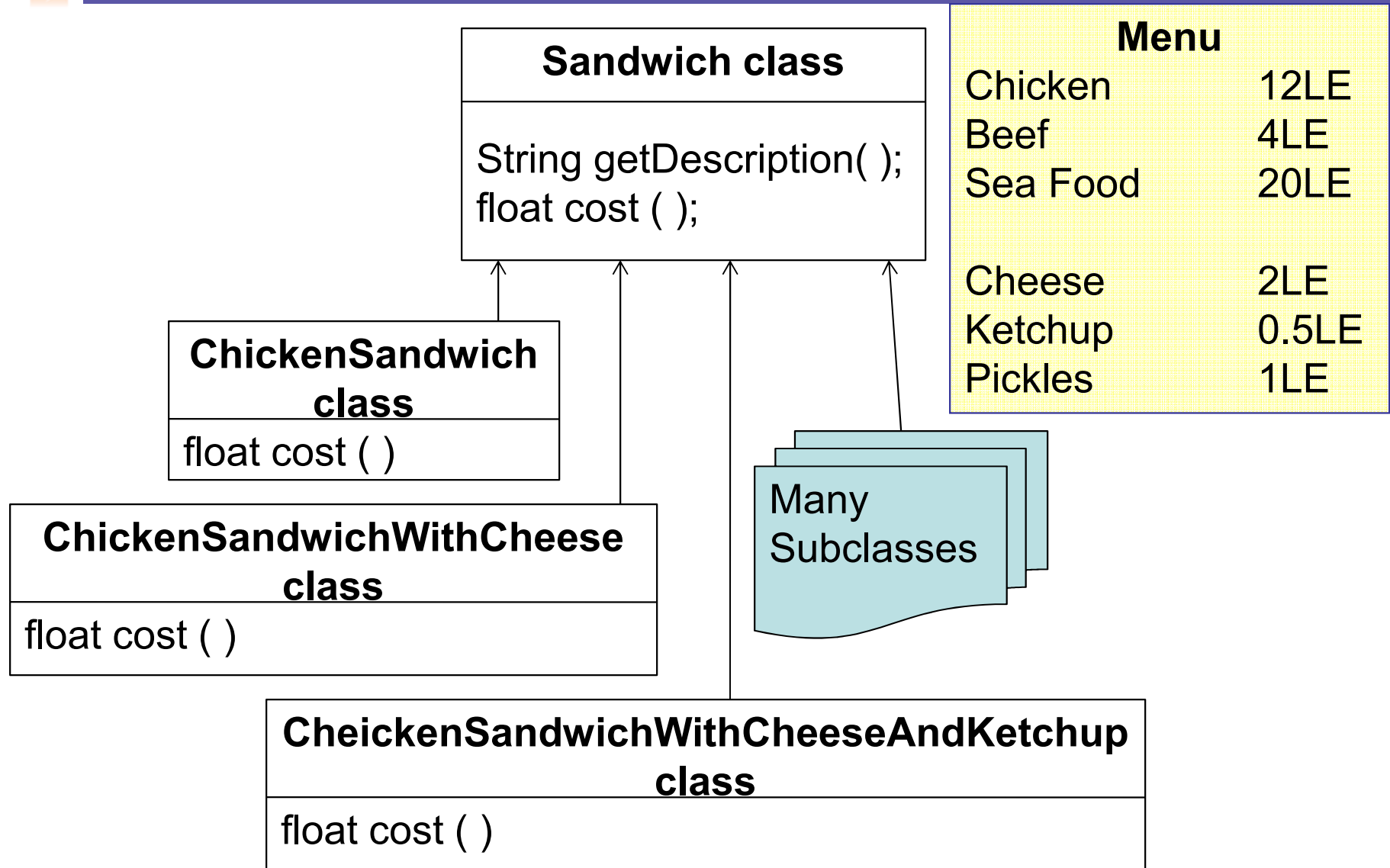
The Decorator Pattern



Chapter 5 Outline

- ☐ Case Study
- ☐ Suggested Solutions
- ☐ Disadvantages of the suggested solution
- ☐ Definition of Decorator Design Pattern
- ☐ Class Diagram of Decorator Design Pattern
- ☐ Decorator Pattern Implementation
- ☐ Why Decorator Pattern?
- ☐ Decorator Pattern in JDK.

Case Study



Case Study (cont')

- **Sandwich** is an abstract class, sub-classed by all sandwiches offered in the restaurant.
- The **cost ()** method of **Sandwich** is abstract and must be implemented by the subclasses to return the cost of each sandwich according to the condiments added to the sandwich (e.g. cheese, ketchup, pickles, etc...)

What do you think of such a design? And what will happen when the price of cheese increases or a new condiment is added?



Suggested Solution

Sandwich class

```
boolean cheese, ketchup,...;  
float cheeseCost, ketchupCost,...;  
public float cost( )  
{  
    float condimentCost = 0.0;  
    if (hasCheese()) {  
        condimentCost += cheeseCost;  
    }  
    if (hasKetchup()) {  
        condimentCost += ketchupCost;  
    }  
    .... // for all possible condiments  
    return condimentCost;  
}
```

ChickenSandwich class extends Sandwich

```
public ChickenSandwich ( ) {  
    description = "Chicken Sandwich  
    With Cheese ";  
    setCheese(true);  
}  
public float cost( )  
{  
    return chickenCost+ super.cost();  
}
```

**What do you think of
this solution?**



Disadvantages of the previous design

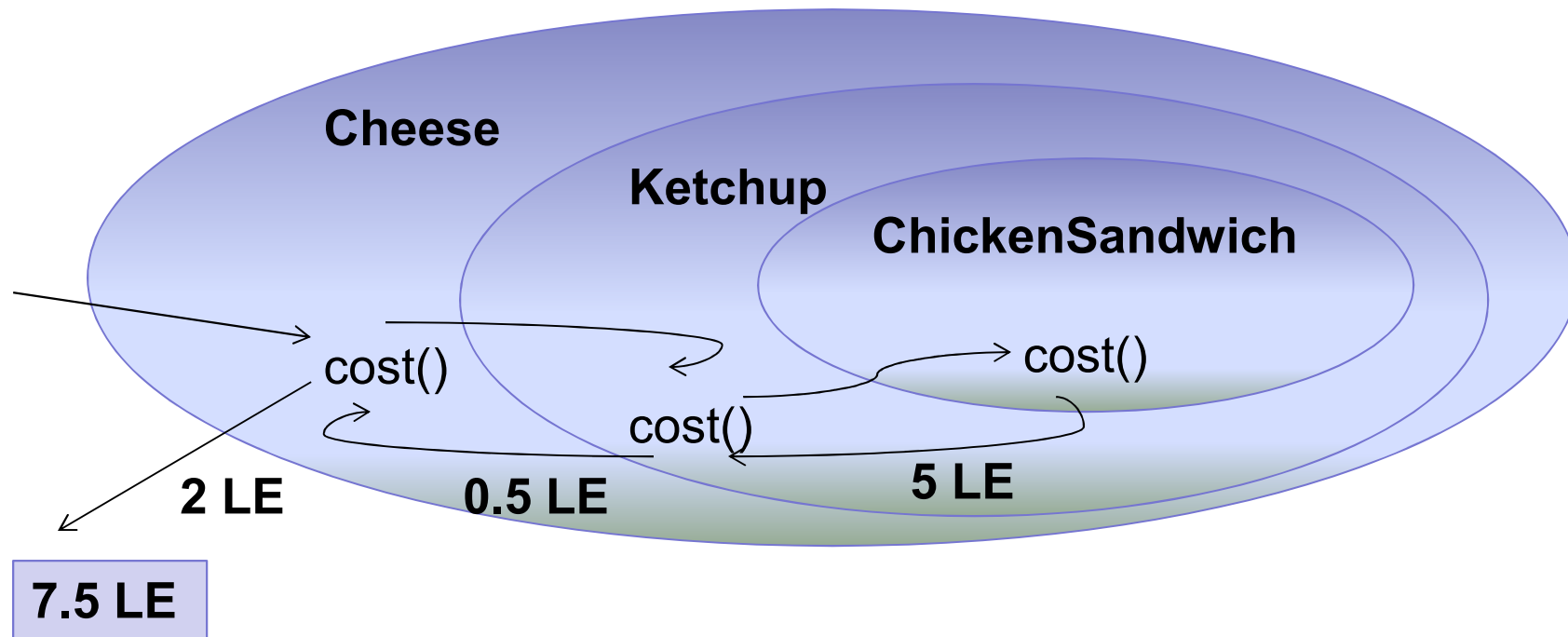
- Price changes for condiments require modifying the Sandwich class.
- New condiments require adding new methods and changing the cost method of the Sandwich class.
- All subclasses inherit all getters and setters of all condiments which may be inappropriate for some subclasses.
- What if a customer wants a double cheese sandwich.



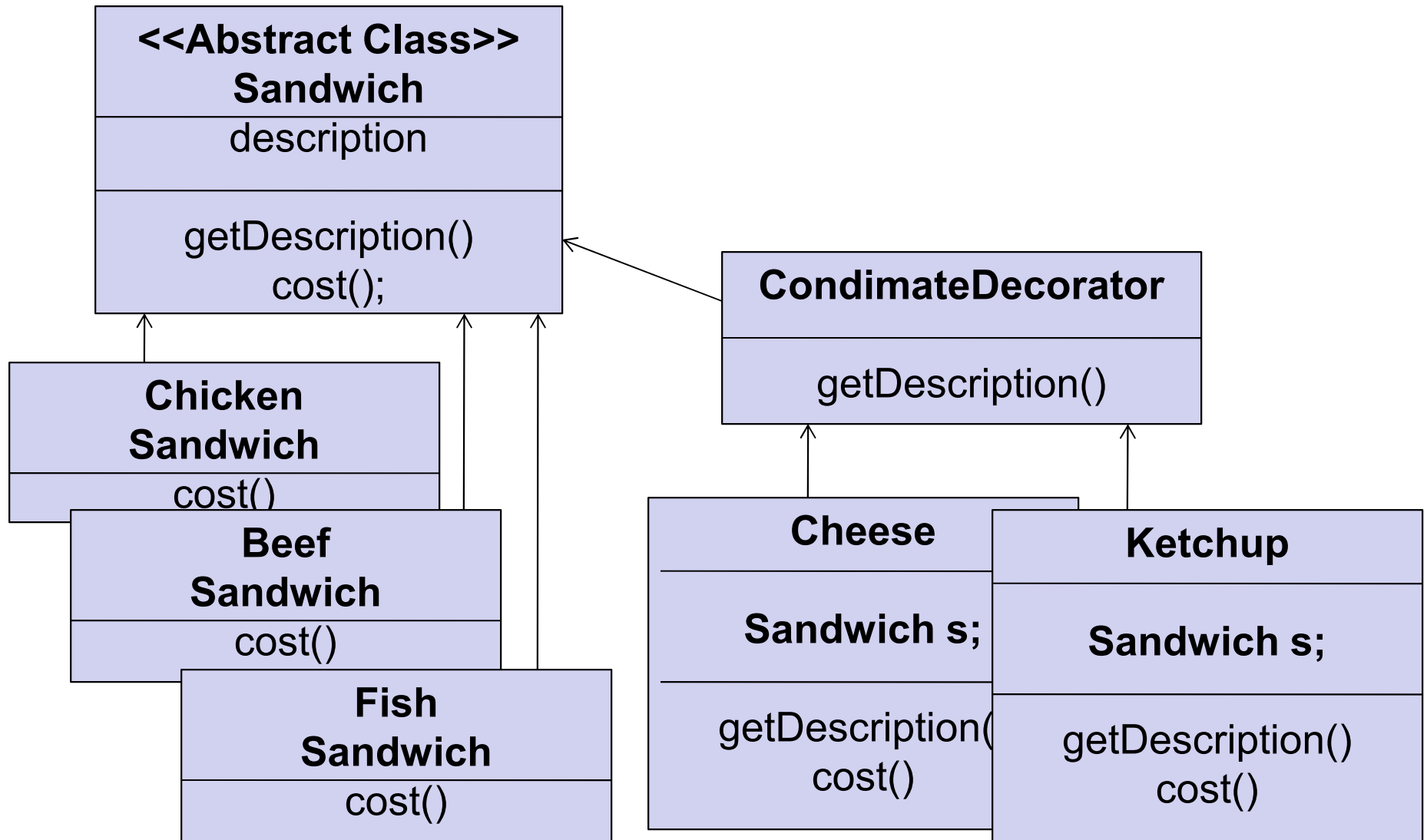
Decorator Design Pattern Definition

- The **Decorator** Pattern attaches additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator Design Pattern



Decorator Pattern Class Diagram





Decorator Pattern Entities

- The **Decorators** have the same type as the objects they are going to decorate (**Component**) to be able to stand in place of the component.
- The **decorators** inherited the **Component** to achieve the *type matching*, not to get *behavior*.
- The behavior comes in through the **composition** of **decorators** with the base **components** (as well as other decorators).



Decorator Pattern Implementation

```
public abstract class Sandwich {  
    String description = "Unknown";  
    public String getDescription() {  
        return description;  
    }  
    public abstract float cost();  
}
```

```
public abstract class CondimentDecorator extends Sandwich {  
    public abstract String getDescription();  
}
```



Decorator Pattern Implementation(cont')

```
public class ChickenSandwich extends Sandwich {  
    public ChickenSandwich () {  
        description = "Chicken Sandwich ";  
    }  
    public float cost() {  
        return 5;  
    }  
}
```

```
public class Cheese extends CondimentDecorator {  
    Sandwich sandwich ;  
    public Cheese (Sandwich sandwich ) {  
        this.sandwich = sandwich ;  
    }  
    public String getDescription() {  
        return sandwich.getDescription() + ", with cheese";  
    }  
    {  
        public float cost() {  
            return 2 + sandwich.cost(); } }  
}
```



Decorator Pattern Implementation(cont')

```
public class Order {  
    public static void main(String args [ ]) {  
        Sandwich sandwich = new ChickenSandwich();  
        System.out.print (sandwich.getDescription());  
        System.out.println (sandwich.cost()+ "LE");  
  
        Sandwich sandwich2 = new BeefSandwich();  
        sandwich2 = new Cheese ( sandwich2 );  
        sandwich2 = new Ketchup (sandwich2 );  
        System.out.println ( sandwich2.getDescription() );  
        System.out.print ( sandwich2.getCost() + "LE" );  
    }  
}
```

Chicken Sandwich 5LE

Beef Sandwich, with Cheese, with Ketchup 8.5

LE



Why Decorator Pattern?

- How to decorate your classes at runtime using composition.
- Give your objects new responsibilities without making any code changes.



Decorator in JDK

- Java I/O API classes applies the Decorator pattern.
- `FileInputStream` is the component that's being decorated.
- `BufferedInputStream` is a concrete decorator that adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method `readLine()` for reading character-based input, a line at a time.



Chapter 6

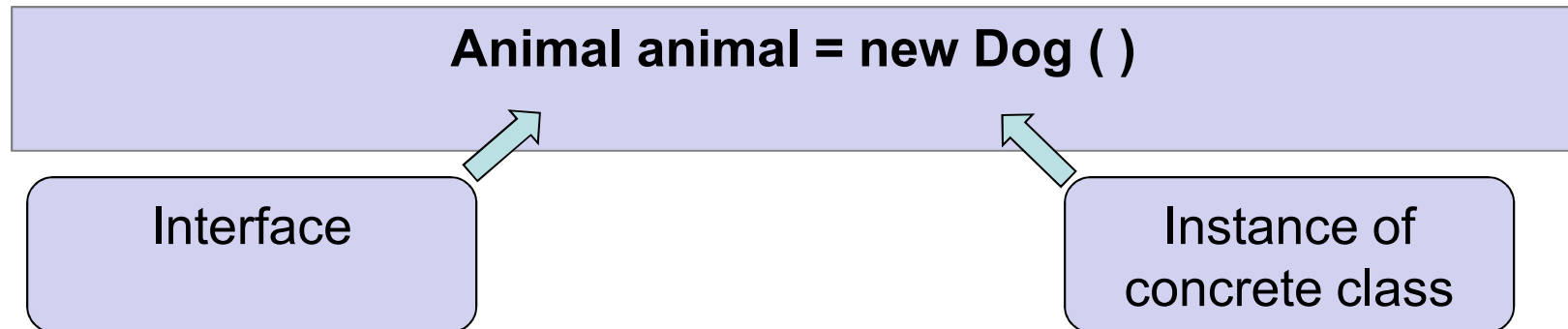
The Factory Method and Abstract Factory Patterns



Chapter 6 Outline

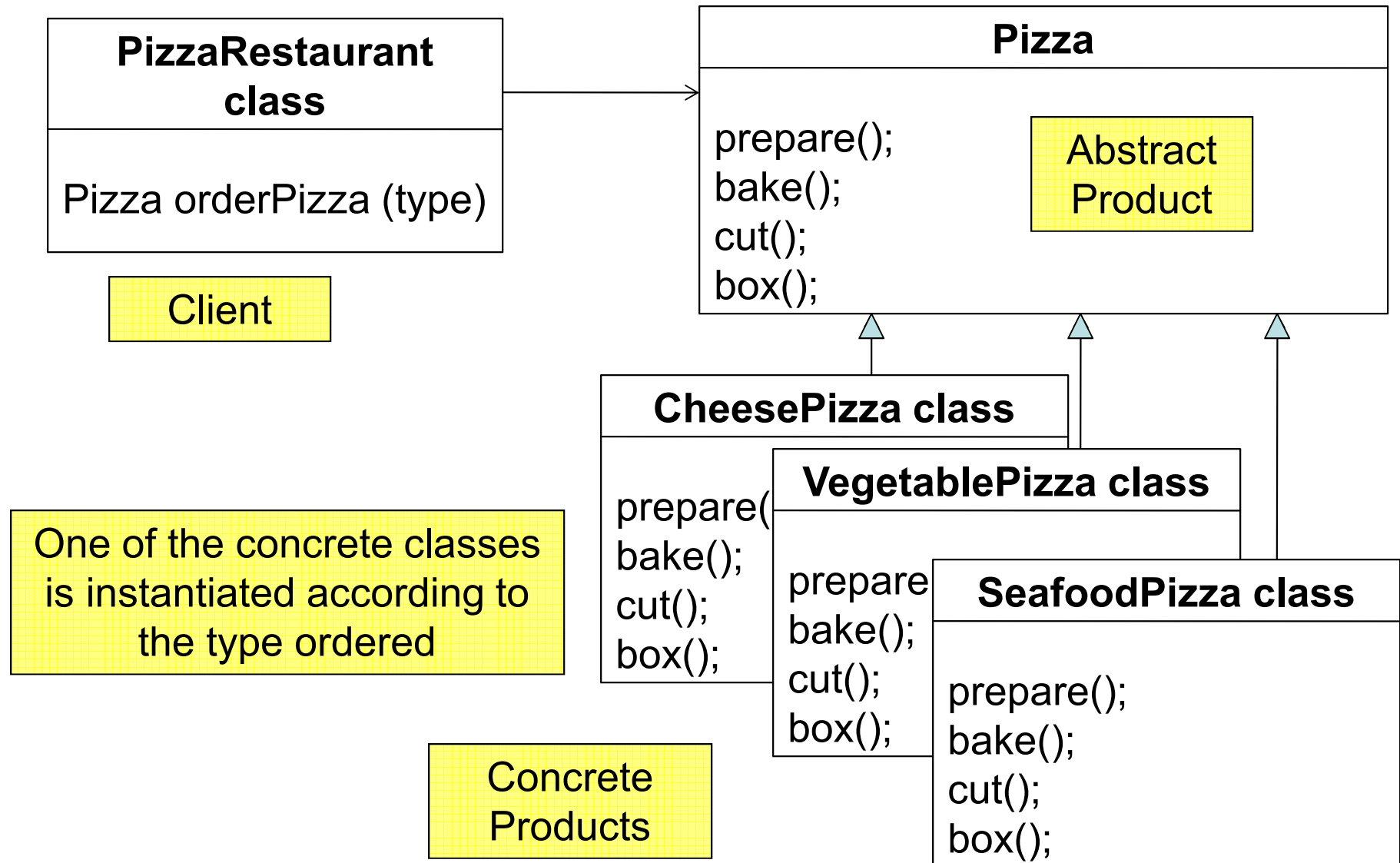
- ☐ Case Study.
- ☐ Solution 1: The Simple Factory.
- ☐ Factory Method Design Pattern.
- ☐ Factory Method Pattern Class Diagram.
- ☐ Factory Method Pattern Definition.
- ☐ Why Factory Pattern?
- ☐ Abstract Factory Pattern Definition.
- ☐ Abstract Factory Pattern Class Diagram.
- ☐ Why Abstract Factory Pattern?
- ☐ Factory Method vs. Abstract Factory.

Case Study



- Sometimes it is needed to decide on the concrete class to instantiate in runtime according to some conditions.

Case Study (cont')



Case Study (cont')

```
public class PizzaRestaurant {
    public Pizza orderPizza (String type) {
        Pizza pizza;

        if ( type.equals ("Cheese") )
            pizza =new CheesePizza ();
        else if ( type.equals ("Vegetable") )
            pizza =new VegetablePizza ();
        else if ( type.equals ("Seafood") )
            pizza =new SeafoodPizza ();

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        return pizza;
    }
}
```

} Object Creation

There is nothing
wrong with such
design, is there?



Case Study (cont')

- The previous design is not closed for modification.
- If the restaurant offers new Pizza types or stops existing type, the object creation code must be modified.
- Consider that the object creation code is needed by different use cases (Clients).
- Remember: Encapsulate what varies.



Solution 1: The Simple Factory

```
public class PizzaRestaurant {  
    SimpleFactory factory;  
    public PizzaRestaurant (SimpleFactory factory){  
        this.factory=factory;  
    }  
    public Pizza orderPizza (String type) {  
        Pizza pizza=null;  
        pizza=factory.createPizza(type)  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        return pizza;  
    }  
}
```

```
public class SimpleFactory {  
    public Pizza createPizza (String type) {  
        if ( type.equals ("Cheese") )  
            pizza =new CheesePizza ();  
        else if (type.equals ("Vegetable"))  
            pizza =new VegetablePizza ();  
        else if ( type.equals ("Seafood") )  
            pizza =new SeafoodPizza ();  
        return pizza;  
    }  
}
```

Java™



The Simple Factory (cont')

- The Simple Factory gives a simple solution for the object creation problem.
- It is just a programming expression not a real design pattern.
- The PizzaRestaurant client and all other clients now deal with the abstract Pizza.
- Changes in the concrete Pizza classes will affect only the Factory class.
- We can have more factories for the different styles of pizza ItalianStyleFactory, EgyptianStyleFactory, etc... each can create one of the set of concrete Pizza classes of this style.



The Simple Factory (cont')

```
public class ItalianPizzaRestaurant {  
    SimpleFactory factory;  
    public PizzaRestaurant (SimpleItalianFactory factory){  
        this.factory=factory;  
    }  
    public Pizza orderPizza (String type) {  
        Pizza pizza=null;  
        pizza=factory.createPizza(type);  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        return pizza;  
    }  
}
```

```
public class SimpleItalianFactory {  
    public Pizza createPizza (String type) {  
        if ( type.equals ("Cheese") )  
            pizza =new ItalianCheesePizza ();  
        else if (type.equals ("Vegetable"))  
            pizza =new ItalianVegetablePizza ();  
        else if ( type.equals ("Seafood") )  
            pizza =new ItalianSeafoodPizza ();  
        return pizza;  
    }  
}
```

Ja



The Simple Factory (cont')

- Some clients can use the available factory to get a pizza object **but** don't stick to the procedure of orderPizza method (prepare then bake then cut and finally box Pizza object)
- It is needed to have a framework that ties the creation and the procedure together in a flexible way.
- That is what the **Factory Method Design** Pattern provides.



Factory Method Design Pattern

```
public abstract class PizzaRestaurant {  
    public Pizza orderPizza (String type) {  
        Pizza pizza=null;  
        pizza=createPizza(type);  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
    abstract Pizza createPizza (String type);  
}
```

The Factory method



Factory Method Design Pattern (cont')

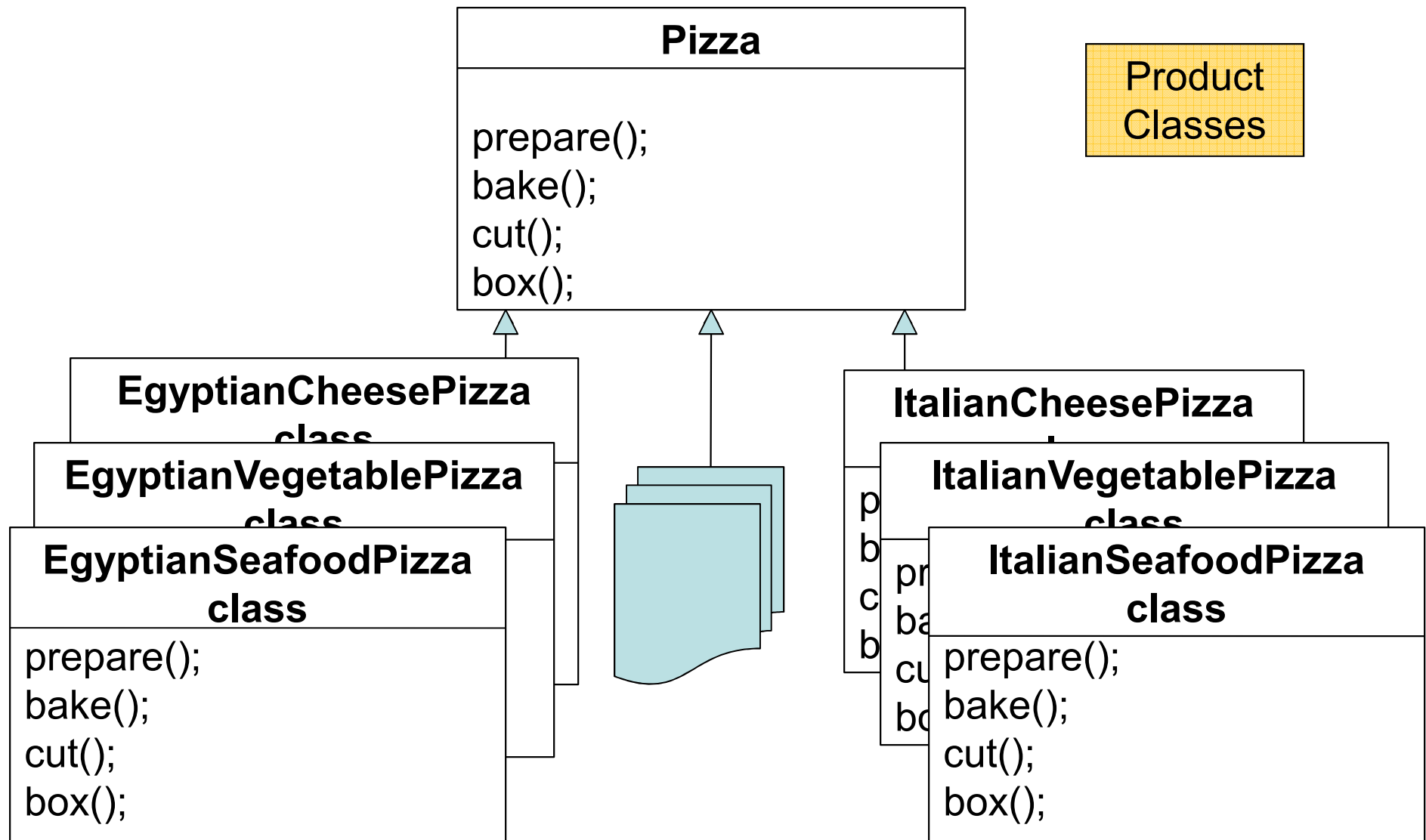
```
Public class ItalianPizzaRestaurant extends  
PizzaRestaurant {  
    public Pizza createPizza (String type) {  
        Pizza pizza=null;  
        if ( type.equals ("Cheese") )  
            pizza =new ItalianCheesePizza ();  
        else if (type.equals ("Vegetable"))  
            pizza =new ItalianVegetablePizza ();  
        else if ( type.equals ("Seafood") )  
            pizza =new ItalianSeafoodPizza ();  
  
        return pizza;  
    }  
}
```

The Factory method
implementation



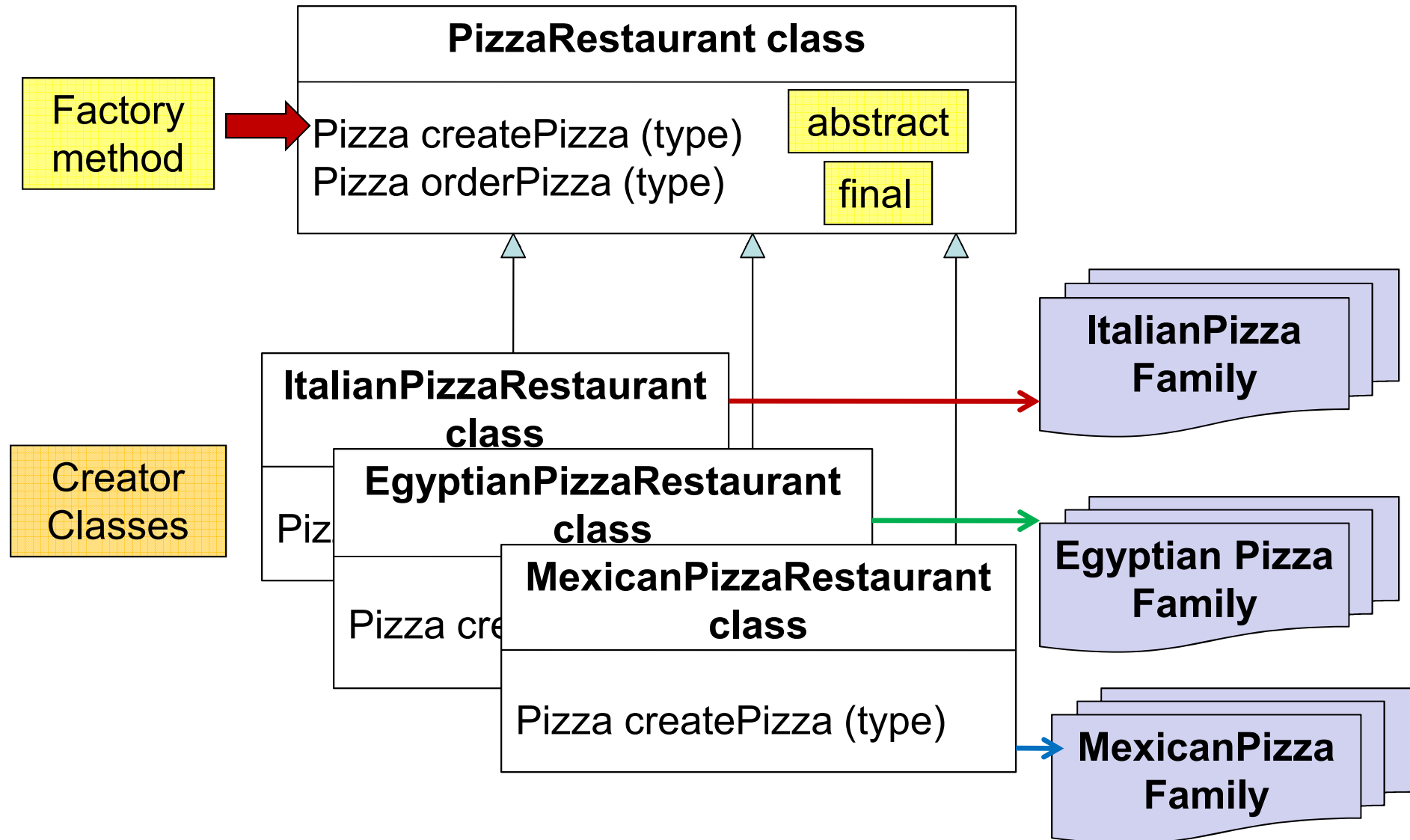


Factory Method Pattern Class Diagram





Factory Method Pattern Class Diagram



Factory Method Design Pattern (cont')

```
PizzaRestaurant italianResturant = new ItalianPizzaResturant ();
italianResturant.orderPizza("cheese");
```

The method of the Abstract **PizzaRestaurant** class runs

Inside orderPizza method

```
Pizza pizza = createPizza ("cheese");
```

The method of the concrete **ItalianPizzaResturant** class runs and returns an **ItalianCheesePizza** object

```
pizza.prepare();
pizza.bake();
pizza.cut();
Pizza.box();
```

The methods of the concrete **ItalianCheesePizza** object run



Factory Method Pattern Definition

- The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses.
- Factory method is a special type of Template method (chapter 8)

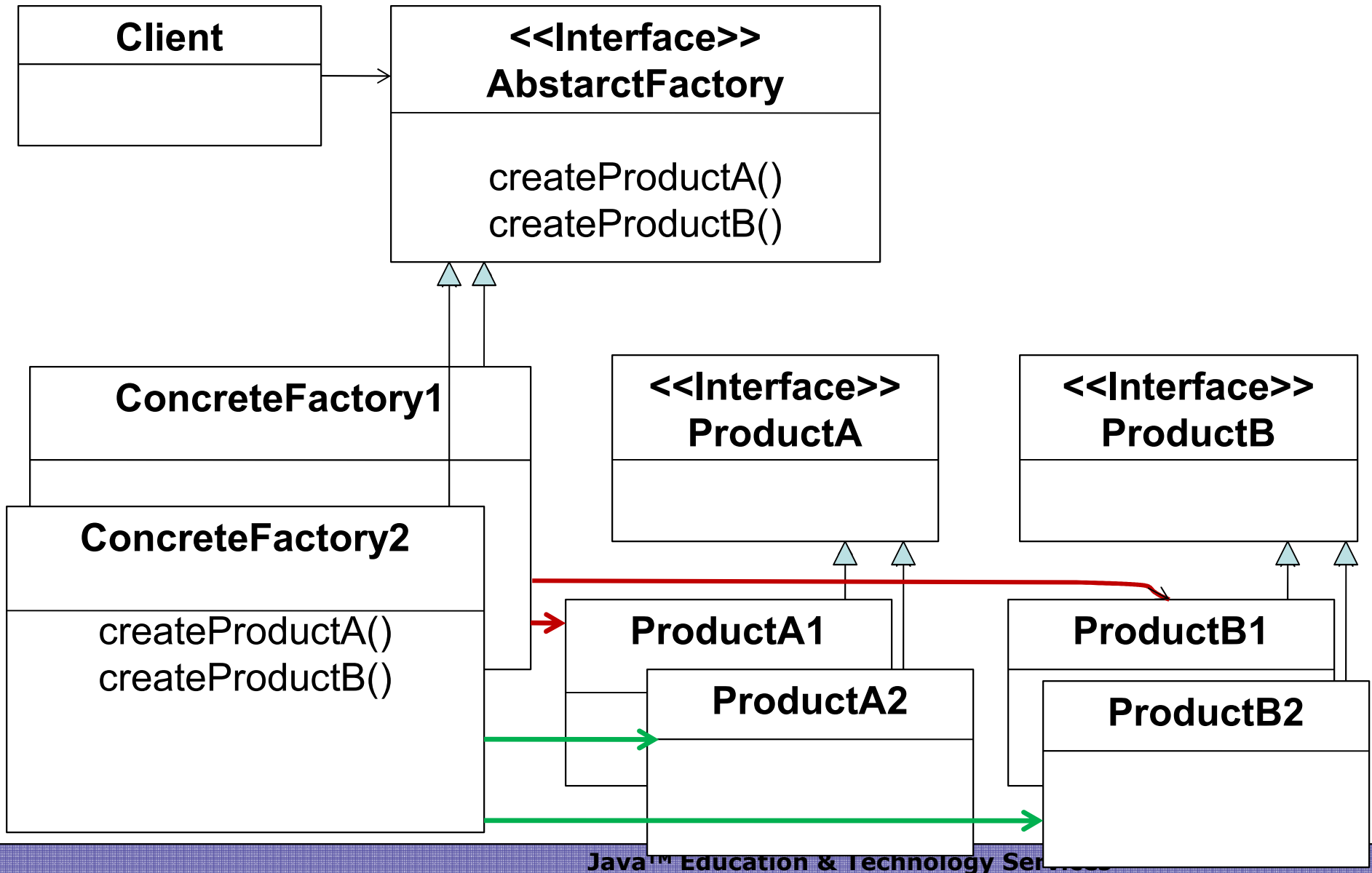


Why Factory Pattern?

- There are ways to make objects other than using the **new** operator.
- Encapsulating object creation keeps application loosely coupled and less dependent on implementations.



Abstract Factory Pattern Class Diagram





The Abstract Factory Pattern Definition

- The Abstract Factory Pattern provides an interface for creating families of related objects without specifying their concrete classes.
- The methods of the Abstract Factory are often implemented as Factory methods.



The Abstract Factory Participants

- Abstract Factory
 - declares an interface for operations that create abstract products objects
- Concrete Factory
 - implements the operations to create concrete product objects
- Abstract Product
 - declares an interface for a type of product object
- Concrete Product
 - defines a product object to be created by the corresponding concrete factory implements the AbstractProduct interface
- Client
 - uses interfaces declared by AbstractFactory and AbstractProduct classes



Why Abstract Factory Pattern?

- As we have seen the Factory method enables clients to use an object without specifying its concrete class.
- Sometimes it is needed to create a group of related objects of different type without specifying their concrete classes.



When to use Abstract Factory Pattern?

- ☐ The system should be independent of how its products are created, composed, and represented.
- ☐ The system should be configured with one of multiple families of products—for example, Microsoft Windows or Apple OSX classes.
- ☐ The family of related product objects is designed to be used together, and you must enforce this constraint. This is the key point of the pattern; otherwise, you could use a Factory Method.



Factory Method vs. Abstract Factory

Factory Method	Abstract Factory
Both are used to create objects Clients are decoupled from the actual concrete classes	
It is used to create one object.	It provides an interface to create a family of objects.
You need to extend a class and implement the factory method to create a specific concrete class.	You need to instantiate one of the concrete subclasses of the Factory and pass it to the client.
It uses inheritance (object creation is delegated to the subclasses)	It uses Composition (objects creation is implemented in the methods of the Factory)



Chapter 7

The Singleton Pattern



Chapter 7 Outline

- ☐ **Why Singleton?**
- ☐ **The Singleton Pattern Class Diagram.**
- ☐ **Simple Singleton Pattern Implementation**
- ☐ **The Singleton Design Pattern Definition.**
- ☐ **Singleton and Multithreading.**
- ☐ **Different implementations of Singleton.**



Why Singleton?

- Sometime you need to make sure that all your application is using the same global resource not multiple copies of it.
- Resource pooling is a typical example of singleton pattern.

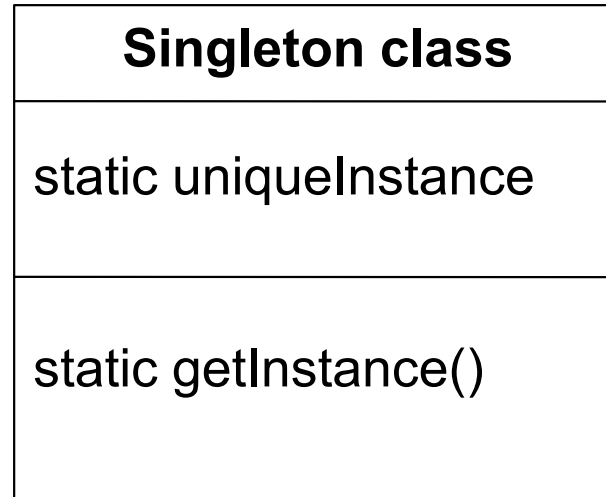


The Singleton Pattern

- How to make sure that only one object is instantiated for your class?
 - Does making your class NOT public solve the problem?
 - What about making its constructor private?
 - If the Constructor is private who can write such a line of code: `new MyClass()` ?
 - Can a static method in your class help?



The Singleton Pattern Class Diagram





Simple Singleton Pattern Implementation

Singleton class

```
public class MyClass{  
    private static MyClass uniqueInstance;  
  
    private MyClass() {}  
  
    public static MyClass getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new MyClass();  
        }  
        return uniqueInstance;  
    }  
}
```

What do you think of
this solution?



The Singleton Design Pattern Definition

- The **Singleton** Pattern ensures a class has only one instance, and provides a global point of access to it.



Singleton and Multithreading

- Consider the case of multithreaded application:

Time
↓

Thread 1

```
public static MyClass getInstance() {
```

```
    if (uniqueInstance == null) {
```

```
        uniqueInstance = new MyClass();
```

```
    return uniqueInstance;
```

Thread 2

```
public static MyClass getInstance() {
```

```
    if (uniqueInstance == null) {
```

```
        uniqueInstance = new MyClass();
```

```
    return uniqueInstance;
```



Suggested Solutions : Synchronized Method

Singleton class

```
public class MyClass{  
    private static MyClass uniqueInstance;  
    private MyClass() {}  
    public static synchronized MyClass getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new MyClass();  
        }  
        return uniqueInstance;  
    }  
}
```

What do you think of
this solution?



Suggested Solutions : Eager Instantiation

Singleton class

```
public class MyClass{  
    private static MyClass uniqueInstance=new MyClass();  
    private MyClass() { }  
    public static MyClass getInstance() {  
        return uniqueInstance;  
    }  
}
```

What do you think of
this solution?



Suggested Solutions: Double Check

Singleton class

```
public class MyClass{  
    private volatile static MyClass uniqueInstance;  
    private MyClass() { }  
    public static MyClass getInstance() {  
        if (uniqueInstance == null) {  
            synchronized (MyClass.class) {  
                if (uniqueInstance == null) {  
                    uniqueInstance = new MyClass();  
                }  
            }  
        }  
        return uniqueInstance;  
    }  
}
```

Volatile must have
its data
synchronized
across all threads

What do you think of
this solution?



Different implementations of Singleton

- **Synchronized Method:**
 - The simplest and most straightforward way.
 - It forces every thread to wait for its turn to enter the method.
 - Performance decreases if high traffic part of your code uses the method
 - Note that we need that Synchronization only in the first time of the method.
 - `uniqueInstance` is instantiated in a lazy manner (Only when needed)
- **Eager Instantiation:**
 - It relies on JVM to create the unique instance when the class is loaded before any thread accesses the static variable.
 - It is not as clear to the reader as the previous way.
- **Double Check:**
 - The Synchronization is used for the first time only.
 - This will not work for jdk 1.4 or earlier due to the improper implementation of volatile keyword



Chapter 8

The Adapter and Façade Patterns



Chapter 8 Outline

- ❑ Why Adapter Pattern?**
- ❑ Overview of the Adapter Pattern.**
- ❑ The Adapter Pattern Class Diagram.**
- ❑ The Adapter Design Pattern Definition.**
- ❑ Object Adapter vs. Class Adapter.**
- ❑ Example on Adapter Pattern.**
- ❑ Decorator vs. Adapter.**
- ❑ Façade Design Pattern.**
- ❑ Façade Design Pattern Class Diagram.**
- ❑ Façade Design Pattern Definition.**

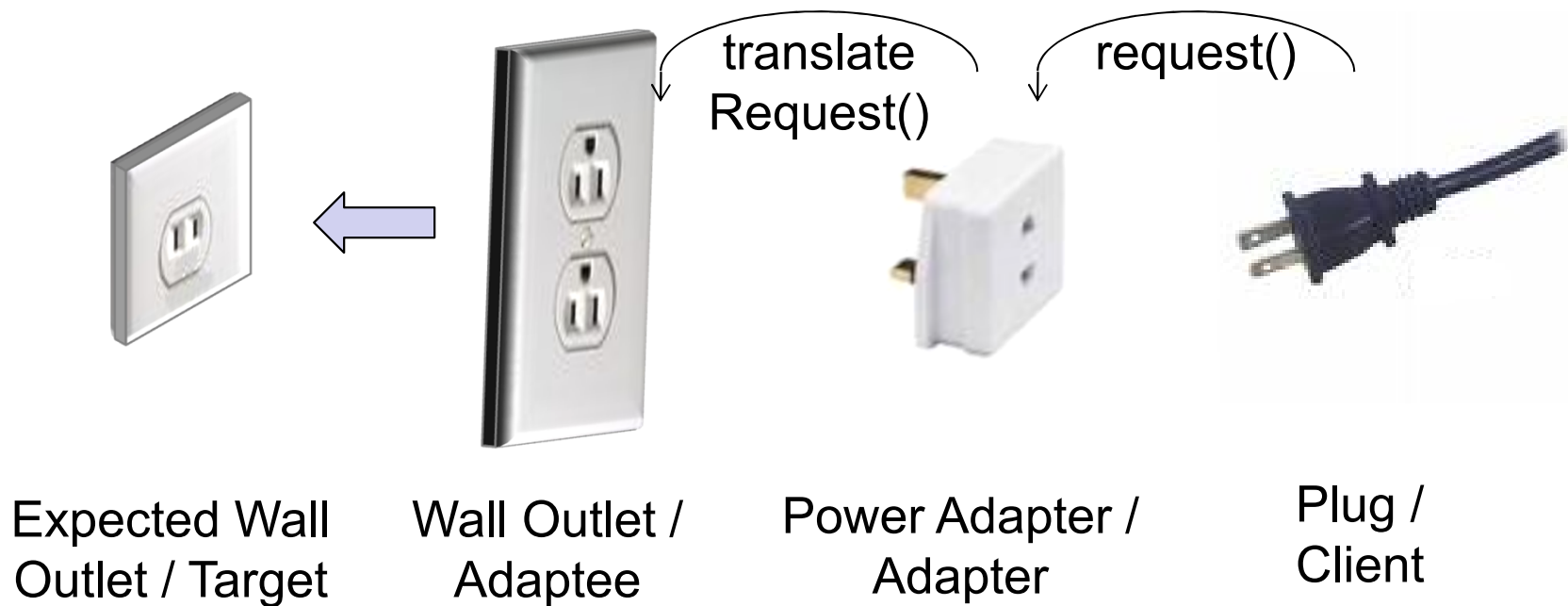


Why Adapter Pattern ?

- Sometimes you need to use an available object as if it is of another type.
- In this case you couldn't use the available object directly as it has different interface.
- You should wrap the object you have in an adapter to look like the type you want.

Overview of the Adapter Pattern

- The AC adapters change the shape of the outlet to match your plug.

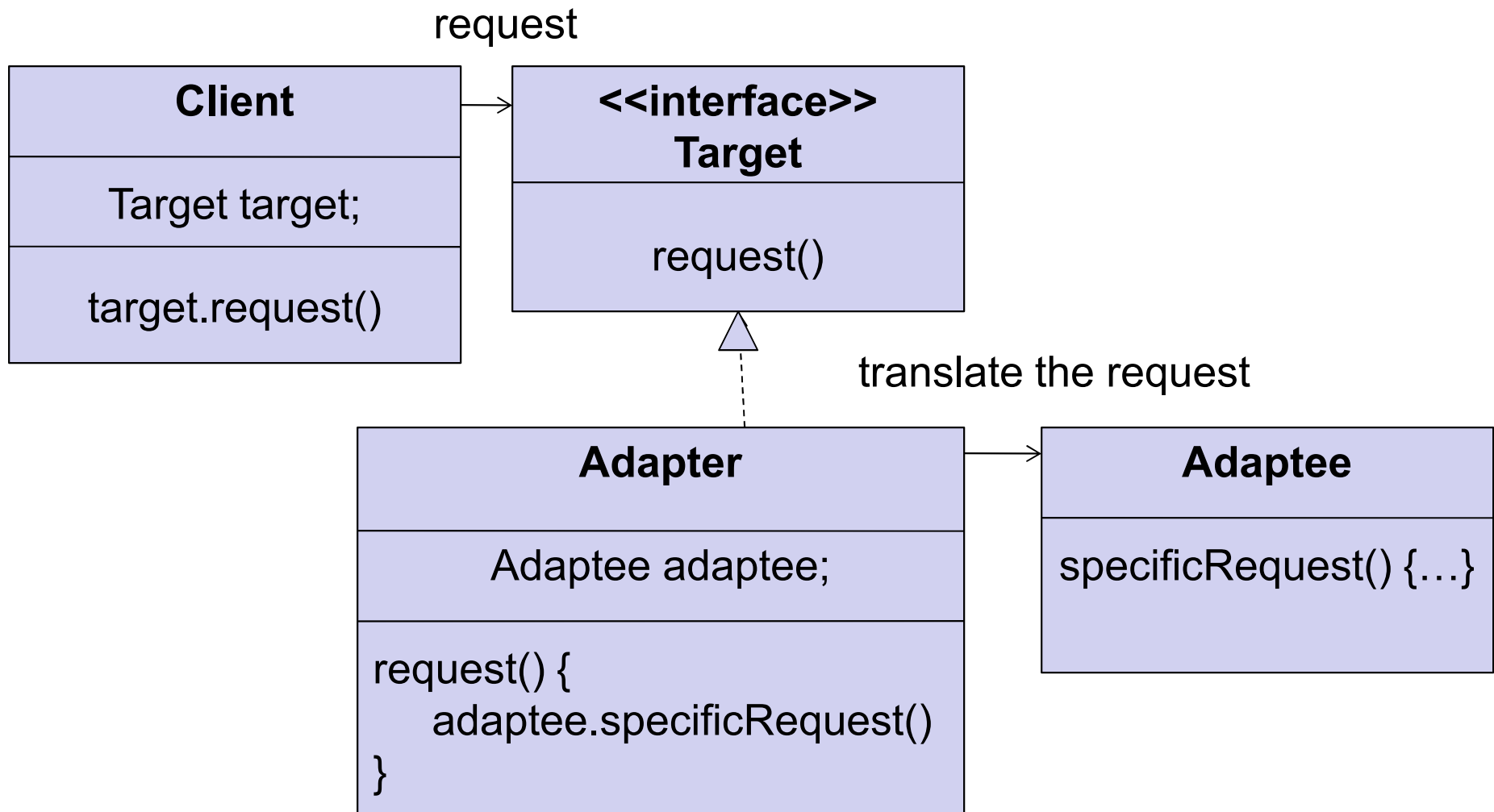


- OO Adapters take an interface and adapt it to what clients expect.



The Adapter Pattern Class Diagram

1- Object Adapter Pattern Class Diagram





The Adapter Pattern Class Diagram (cont')

- The **Adapter** implements the **Target** interface and wraps the **Adaptee** class.
- The **Client** requests a service from the Adapter (by calling a method on the Target interface)
- The Adapter translates this request into one or more calls on the **Adaptee** Interface.
- Notes:
 - ✓ The Adapter may wrap one or more Adaptee classes to translate the Client request.
 - ✓ The Adapter may implement one or more Target classes (e.g. to support different behaviors of different versions of the Target).



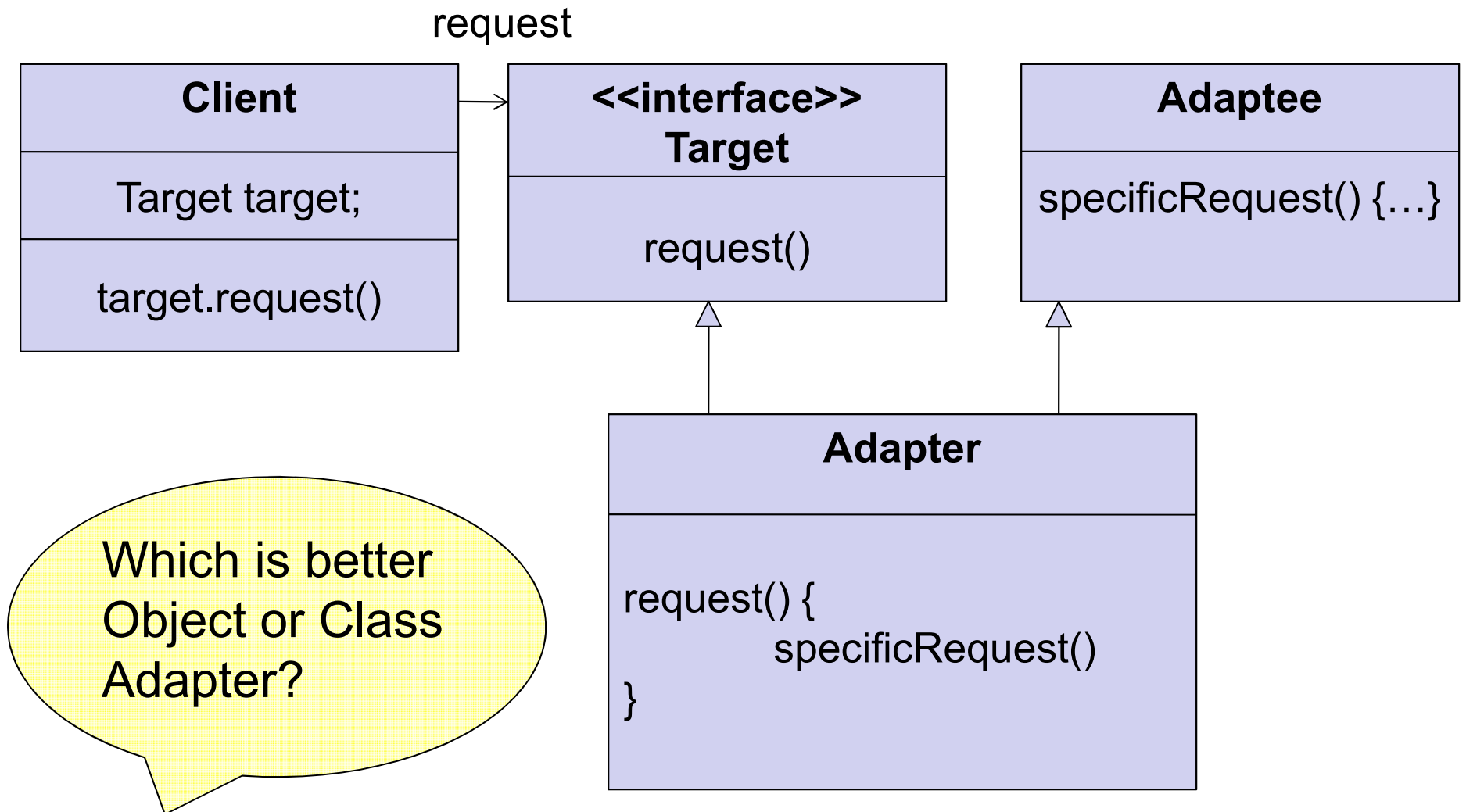
The Adaptor Design Pattern Definition

- The Adaptor Design Pattern converts the interface of a class into another interface which the clients expect.
- There are 2 types of adapters Object adapters and Class Adapters.
- The Object Adapter delegates the request to the Adaptee through composition while Class Adapter uses multiple inheritance.



The Adapter Pattern Class Diagram (cont')

2- Class Adapter Pattern Class Diagram





Object Adapter vs. Class Adapter

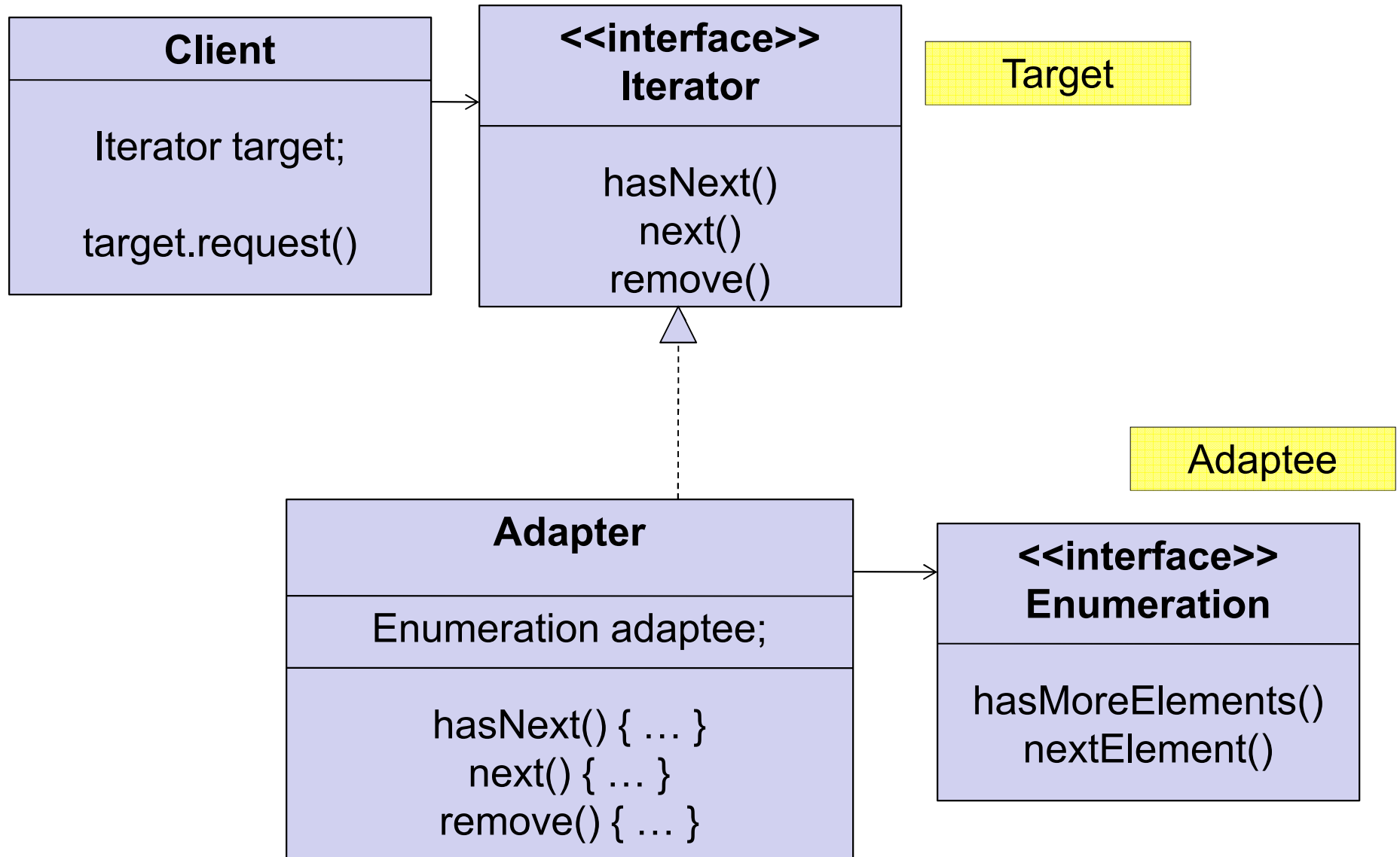
- Object Adapter :
 - Using composition it adapts Adaptee class and all its subclasses.
 - More flexible (Any change in the adapter applies to the Adaptee and all its subclasses).
- Class Adapter
 - Using inheritance it needn't implement all the Adaptee methods.
 - More Efficient (one object acts as both Adapter and Adaptee)



Example on Adapter Pattern

- In early Java Collections , Enumeration interface was used to step through the elements of a collection.
- Recent Collection classes use Iterator interface.
- If you have a legacy system build on Enumeration and you want to use it by a new Client which uses Iterator, you will need an Adapter to wrap Enumeration to look like Iterator

Example on Adapter Pattern (cont')





Example on Adapter Pattern (cont')

class EnumerationTolteratorAdapter

```
public class EnumerationTolteratorAdapter implements Iterator
{
    Enumeration enumm;
    public EnumerationTolteratorAdapter (Enumeration enumm) {
        this.enumm=enumm;
    }

    public boolean hasNext() {
        return enumm.hasMoreElements();}

    public Object next() {
        return enumm.nextElement(); }

    public void remove() {
        throw UnsupportedOperationException(); }
}
```



Decorator vs. Adapter

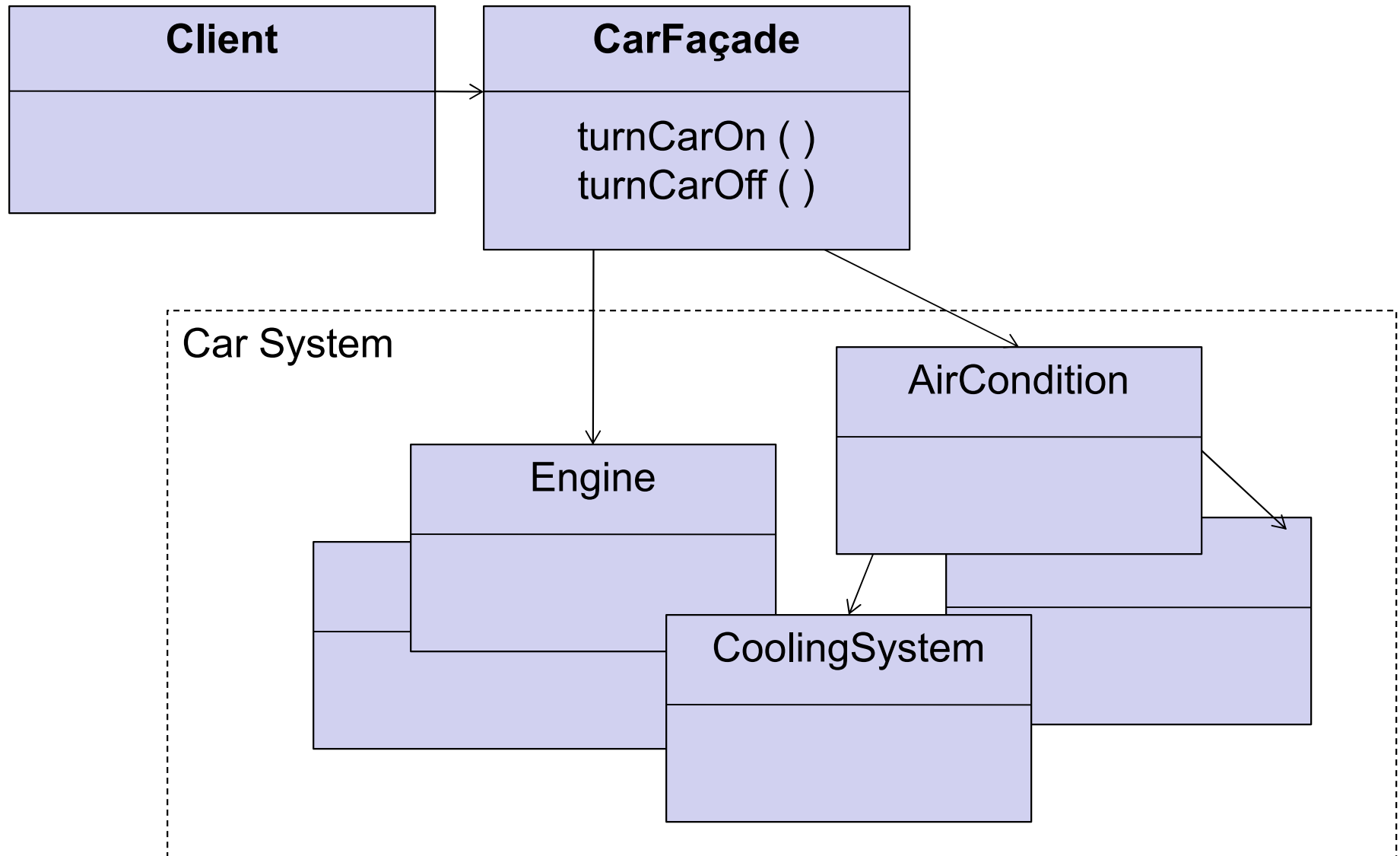
Decorator	Adapter
Adds new behavior or new responsibilities to your class	Convert the interface of a class into another interface which the clients expect.
Client wrap the objects using one or more decorators.	Client is uncoupled



Façade Design Pattern

- Façade Design Pattern is used to convert one interface to another but to simplify the interface to the clients.
- It decouples clients from a subsystem of components.
- Both Façade and Adapter design patterns may wrap multiple classes but for different purposes.

Façade Design Pattern Class Diagram





Façade Design Pattern Definition

- Façade Design Pattern provides a unified and higher interface to a set of interfaces in a subsystem to make it easier to use.
- A single subsystem may have more than one façade.



Chapter 9

The Template Method Pattern



Chapter 9 Outline

- ❑ Why Template Method Pattern?**
- ❑ Case Study.**
- ❑ Suggested Solution.**
- ❑ Template Method.**
- ❑ Template Method Pattern Class Diagram.**
- ❑ Template Method Pattern Example.**
- ❑ Template Method Pattern Definition.**
- ❑ Notes on Template Method Pattern.**



Why Template Method Pattern?

- Sometimes you want to encapsulate pieces of algorithms so that subclasses are fully responsible to provide implementation for these pieces.



Case Study

Tea class

```
public class Tea{  
    public void prepare ( )  
    {  
        boilWater();  
        putTeaBag();  
        addSugar();  
    }  
    void boilWater ( ) { ... }  
    void putTeaBag() {....}  
    void addSugar ( ) { ... }  
}
```

Coffee class

```
public class Coffee {  
    public void prepare ( )  
    {  
        boilWater();  
        putCoffe ();  
        addSugar();  
    }  
    void boilWater ( ) { ... }  
    void putCoffe() {....}  
    void addSugar ( ) { ... }  
}
```

Suggested Solution

Drink class

```
public class Drink {
    public void prepare ( )
    {
        boilWater();
        putIngredient ();
        addSugar();
    }
    void boilWater ( ) { ... }
    void addSugar ( ) { ... }
    abstract void putIngredient ();
}
```

Tea class

```
public class Tea {
    void putIngredient() {....}
}
```

Coffee class

```
public class Coffee {
    void putIngredient() {....}
}
```



Template Method

- The **prepare** method above is a template method because:
 - It defines an algorithm (the algorithm of making a hot drink)
 - Each step of the algorithm is represented by a method.
 - Some methods are handled by the super class.
 - Some methods are abstract to enforce the subclasses to handle them.
 - Some methods may be hooks.

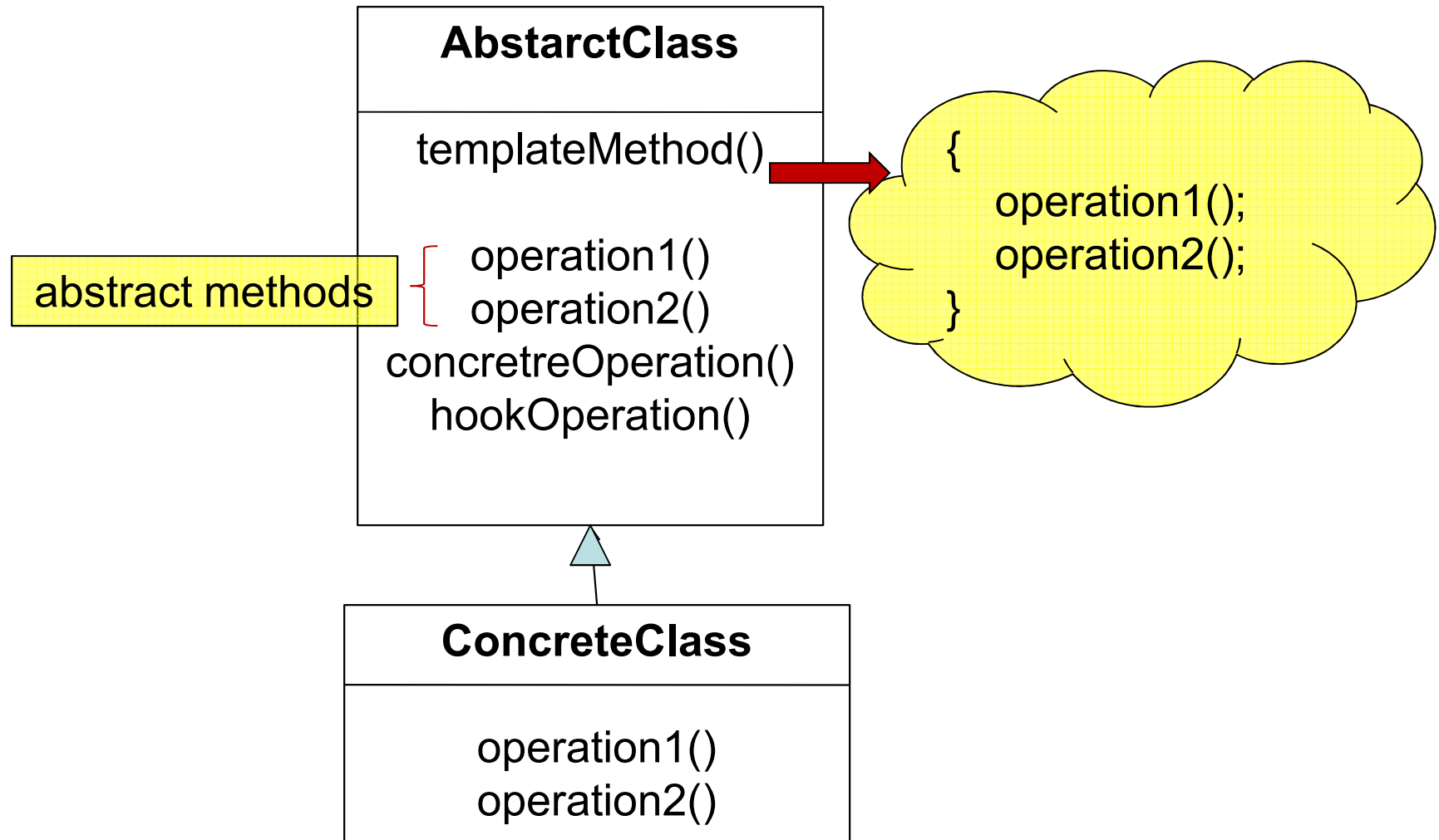


Template Method (cont')

- Hooks are methods that are not abstract in the super class and provide default behaviors.
- They are used for optional steps so subclasses are free to override them or apply the default behavior provided by the super class
- Abstract method makes the super class control and protects the algorithm providing a **framework** that any subclass can be plugged into.



Template Method Pattern Class Diagram





Template Method Pattern Example

AbstractClass

```
public class AbstractClass
{
    public void template( )
    {
        operation1 ();
        operation2 ();
        concreteOperation ();
        hookOperation();
    }
    void concreteOperation ( ) { ... }
    void hookOperation ( ) { ... }
    abstract void operation1 ();
    abstract void operation2 ();
}
```



Template Method Pattern Definition

- The Template Method Pattern defines the skeleton of an algorithm in a method deferring some steps to the subclasses.



Notes on the Template Method Pattern

- The Template Method Pattern applies the Hollywood principal: Don't call us , we will call you.
- The abstract super class controls the algorithms and calls methods of the subclasses when they are needed for implementation.
- Template Method Pattern in JDK:
 - Applet and JFrame.
 - The **sort** method of the **Arrays** class and compareTo method and override compareTo method.



Notes on the Template Method Pattern (cont')

- To prevent the subclasses from changing the algorithm in the template method, declare it as final.
- The Factory method is a special Template method (Encapsulating object creation)



References & Recommended Reading

- Head First Design Patterns *by Eric Freeman & Elisabeth Freeman.*
- The Design Patterns Java Companion *by James W. Cooper.*
- Wikipedia website
[http://en.wikipedia.org/wiki/Design_pattern_\(computer_science\)](http://en.wikipedia.org/wiki/Design_pattern_(computer_science))