CSCI 4050/6050
Software Engineering

Design Patterns

Design Patterns

- A *design pattern* is a recurring solution to a standard problem, in a context.
- Christopher Alexander, a professor of architecture, wrote:

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

- Patterns can be applied to many different areas of endeavor...

Design Patterns are NOT…

- NOT data structures that can be encoded in classes and reused *as is* (i.e., linked lists, hash tables)
- NOT complex domain-specific designs (for an entire application or subsystem)
- NOT libraries or middleware system
- If they are not familiar data structures or complex domain-specific subsystems,

what are they?

They are:

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

Patterns in software development

- Experienced designers reuse solutions that were successfully used in the past
- Well-structured object-oriented systems have recurring patterns of classes and objects.
- Designers knowing patterns that have worked in the past can be more productive; their designs are more flexible and reusable in their own right.
- Software design patterns have been cataloged:

The “Gang of Four” (GoF)

- *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (Addison-Wesley, 1995)
  - *Design Patterns* book catalogs 23 different patterns as solutions to different classes of problems, in C++ & Smalltalk
  - The problems and solutions are broadly applicable, used by many people over many years
  - Patterns suggest opportunities for reuse in analysis, design and programming
  - GOF presents each pattern in a structured format

Design Patterns Elements

- Design patterns have 4 basic elements:
  - Pattern name:
    - meaningful name identifying the pattern
    - increases vocabulary of designers
  - Problem
    - describes when to apply the pattern
    - Describes problem and its context
    - May describe class or object structures
  - Solution
    - Describes elements making up a design, their relationships, responsibilities and collaborations; frequently using UML and abstract code
  - Consequences: results and tradeoffs

Design Patterns Description

Design pattern template

- Pattern name
- Intent – what does the pattern do
- Also known as
- Motivation – an illustrating scenario
- Applicability – in what situations to apply the pattern
- Structure – a graphical representation (UML, OMT)
- Participants – classes and objects and their roles
- Collaborations – how the participants collaborate
- Consequences
- Implementation – critical hints concerning the implement.
- Sample code
- Known uses – examples of the pattern in real systems
- Related patterns

Singleton Pattern Design Pattern

- Sometimes it's appropriate to have exactly one instance of a class:
  - a window manager
  - a system’s configuration (parameters)
  - a logging system
  - a print spooler
  - an object with large data (state)
- Typically, those types of objects — known as singletons — are accessed by different clients (other objects) throughout a software system, and therefore require a global point of access

Singleton Pattern Design Pattern

Name

- Singleton

Intent

- Ensure a class only has one instance, and provide a global point of access to it

Motivation

- It is important for some classes to have exactly one instance
- Make it illegal to have more than one instance, to be safe

Singleton Pattern Design Pattern

Motivation (cont.)

- Examples: there can be many printers in a system, there should be only one printer spooler
- there should be only one object with a large state (internal data)
- creating lots of objects can take a lot of time
- extra objects take up memory
- it is a cumbersome to deal with different objects “floating” around if they are essentially the same

Singleton Pattern Design Pattern

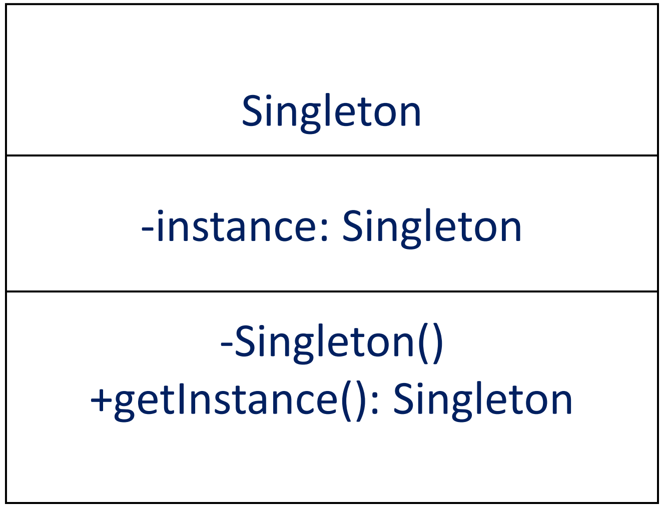
Applicability

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point

- when the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code

Singleton Pattern Design Pattern

Structure



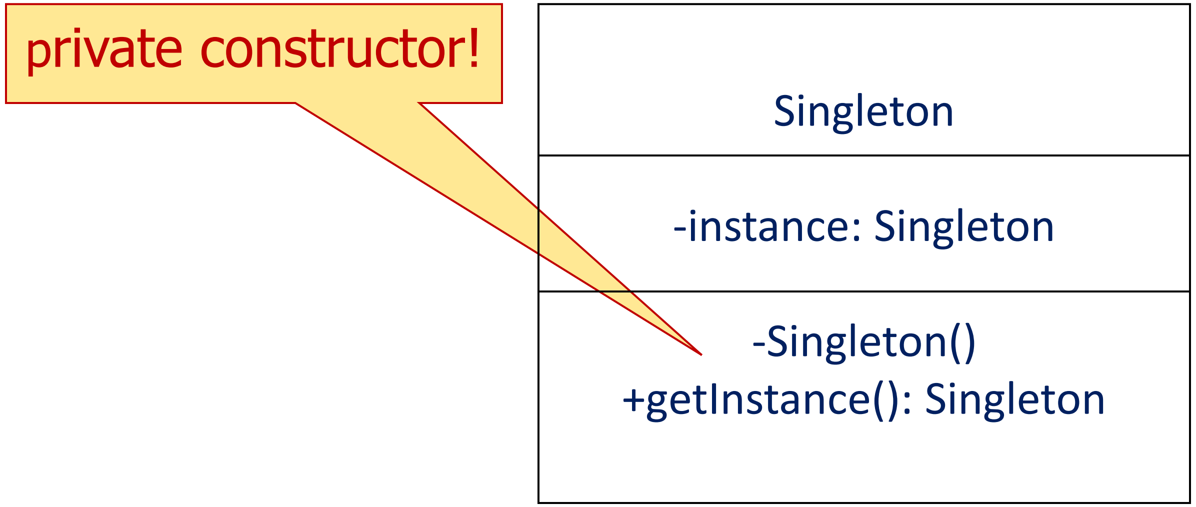
Singleton

-instance: Singleton

-Singleton()
+getInstance(): Singleton

Singleton Pattern Design Pattern

Structure



private constructor!

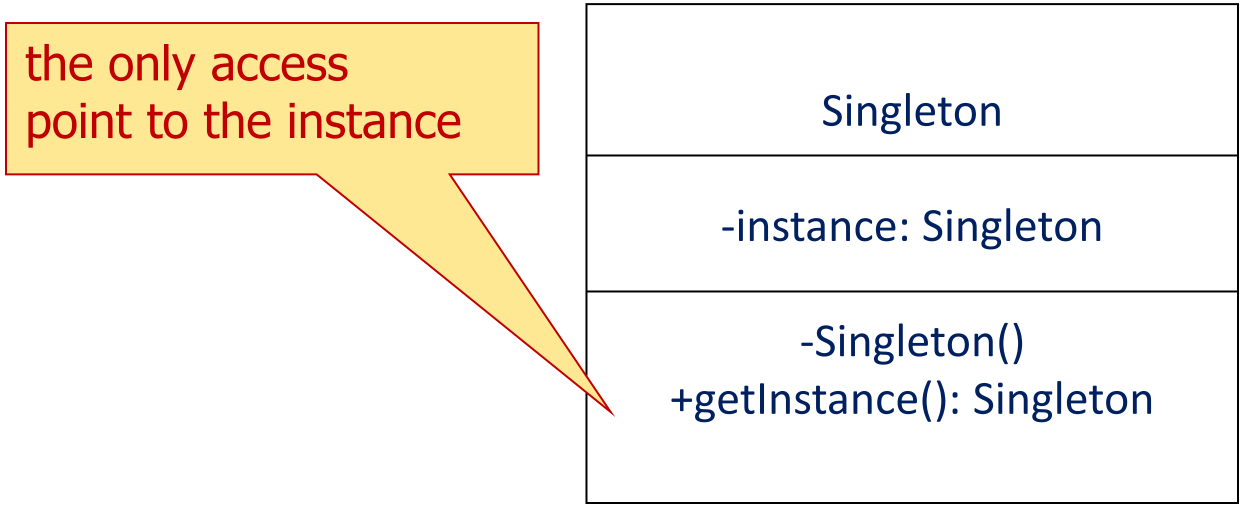
Singleton

-instance: Singleton

-Singleton()
+getInstance(): Singleton

Singleton Pattern Design Pattern

Structure



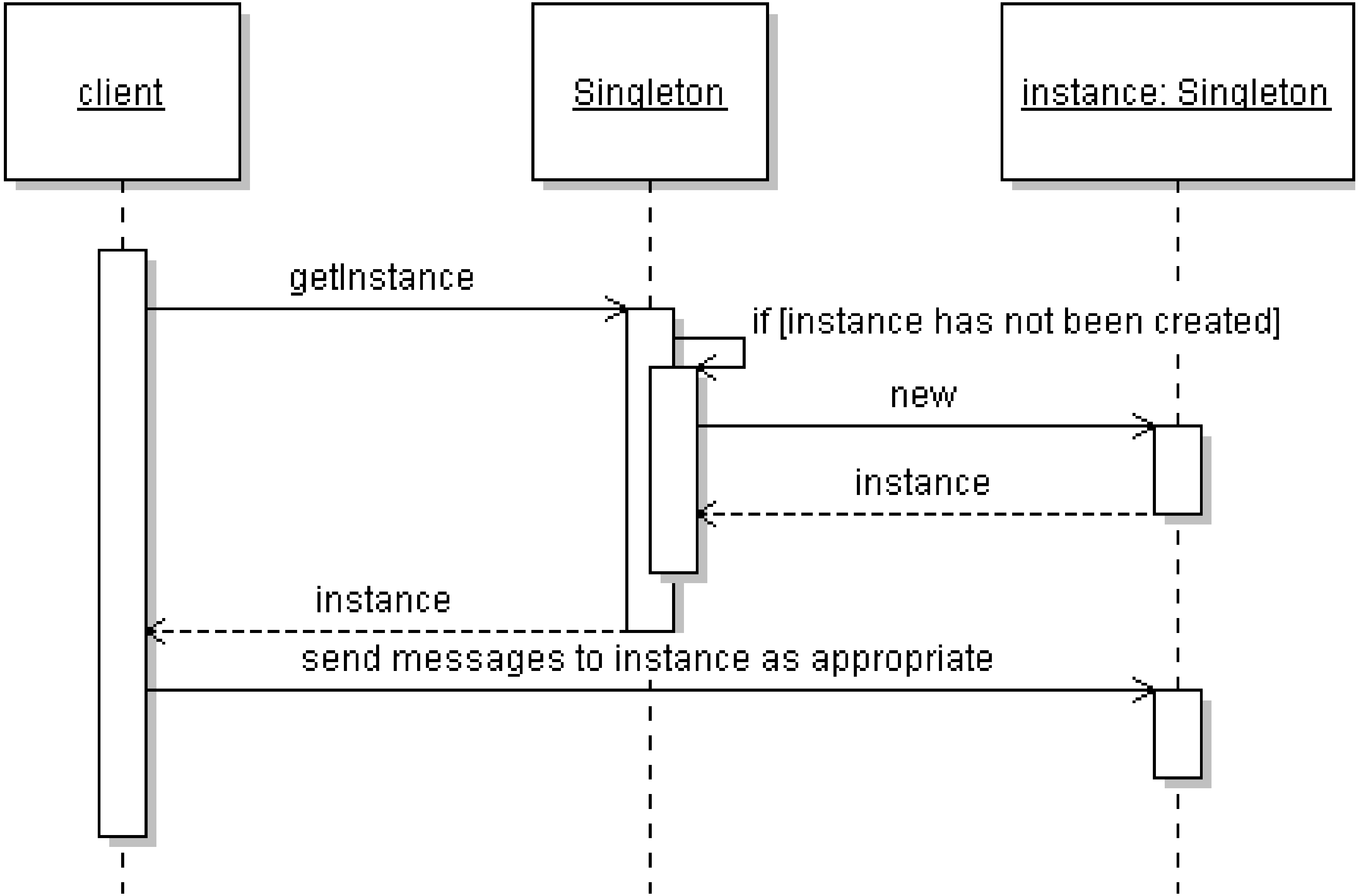
the only access point to the instance

Singleton

-instance: Singleton

-Singleton()
+getInstance(): Singleton

Singleton Pattern Design Pattern


client
Singleton
instance: Singleton
getInstance
if [instance has not been created]
new
instance
instance
send messages to instance as appropriate

Singleton Pattern Design Pattern

Consequences

- Only a single instance exists
- Controlled access to the only instance (the single instance is encapsulated)
- Reduced name space (better than a global variable!)
- Permits a variable (but controlled) number of instances (it is easy to permit more instances)
- More flexible than class scope operations (hard to change the design to allow more instances, for example)

Singleton Pattern Design Pattern

Implementation

- make constructor(s) private so that they can not be called from outside
- declare a single static private instance of the class
- write a public `getInstance()` or similar method that allows access to the single instance;
- possibly protect / synchronize this method to ensure that it will work in a multi-threaded program

Singleton Pattern Design Pattern

Example code

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

public class TestSingleton{
  public static void main( String[] args ){
        Singleton s = Singleton.getInstance();
        …
  }
}
```

Benefits of Design Patterns

- Design patterns enable large-scale reuse of software architectures and also help document systems.

- Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available

- Patterns help improve developer communication

- Pattern names form a common vocabulary

Three Types of Patterns

- **Creational patterns:**
  - Deal with initializing and configuring classes and objects

  Singlton. Factory, Abstract Factory,….

- **Structural patterns:**
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects
  - Proxy, Adaptor, Bridge, Façade, decorator

- **Behavioral patterns:**
  - Deal with dynamic interactions among ensembles of classes and objects
  - How they distribute responsibility
  - Examples: Chain of responsibility, Command, Interpreter, memento

Structural patterns

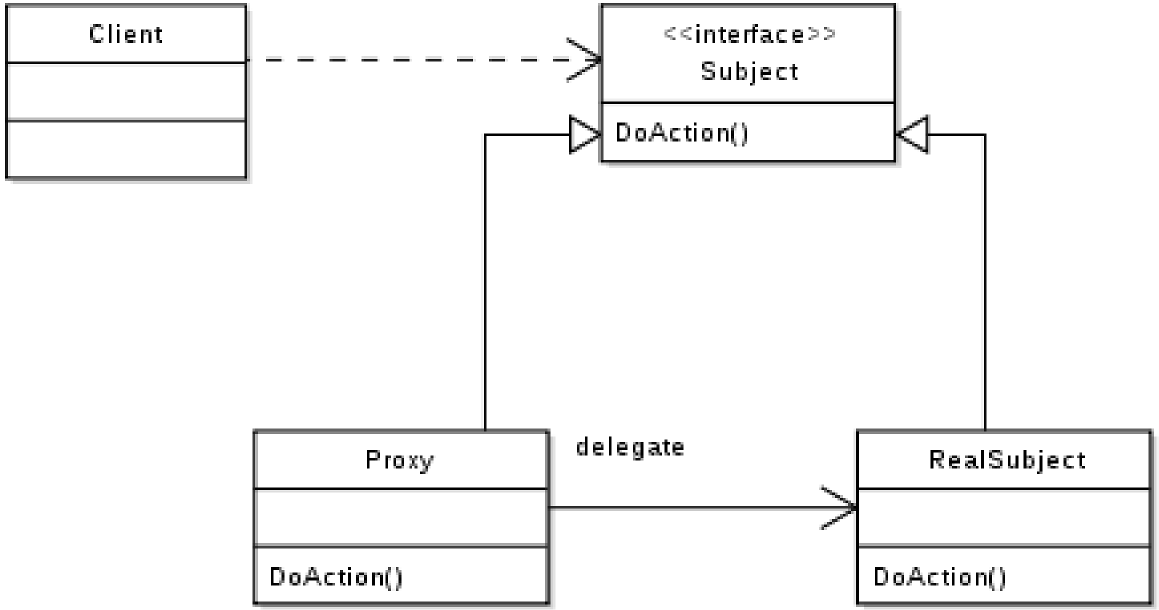
- Describe ways to assemble objects to realize new functionality
  - Added flexibility inherent in object composition due to ability to change composition at run-time
  - not possible with static class composition
- Example: The Proxy Pattern

Proxy Pattern

Proxy: acts as convenient surrogate or placeholder for another object.

- **Remote Proxy:** local representative for object in a different address space.
- **Virtual Proxy**: represent large object that should be loaded on demand
- **Protected Proxy:** protect access to the original object

Proxy Pattern



Proxy Example

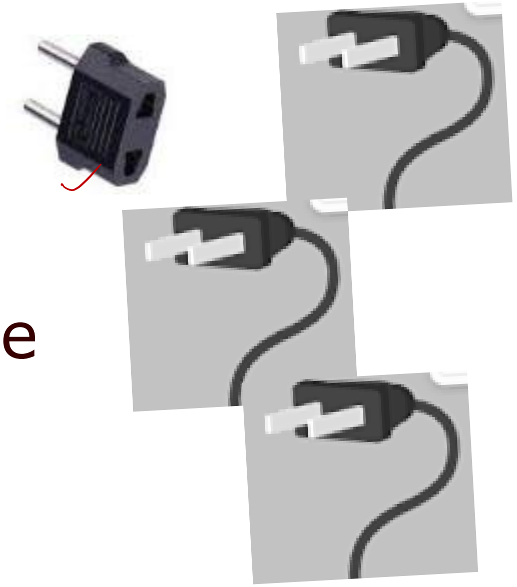
```
package com.java2novice.dp.proxy;

public interface Internet {

    public void connectTo(String host) throws
Exception;
}


  package com.java2novice.dp.proxy;

  public class RealInternet implements Internet {

      @Override
      public void connectTo(String host) {
          System.out.println("Connecting to "+host);
      }
  }
```

```java
// Internet proxy class
 package com.java2novice.dp.proxy;

 import java.util.ArrayList;
 import java.util.List;

 public class InternetProxy implements Internet {

   private Internet internet = new RealInternet(); // composition
   private static List<String> restrictedSites;

   static {
       restrictedSites = new ArrayList<String>();
       restrictedSites.add("jumbxyz.com");
       restrictedSites.add("testme.com");
       restrictedSites.add("adult-site.com");
       restrictedSites.add("bad-site.com");
   }
   @Override
   public void connectTo(String host) throws Exception {

       if(!restrictedSites.contains(host.toLowerCase())){
           internet.connectTo(host);
       }
       throw new Exception("Company restricted this site view");
   }
}
```

```java
package com.java2novice.dp.proxy;

public class ProxyDemo {

    public static void main(String a[]){

        Internet intConn = new InternetProxy();
        try {
            intConn.connectTo("java2novice.com");
            intConn.connectTo("adult-site.com");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Adapter Pattern

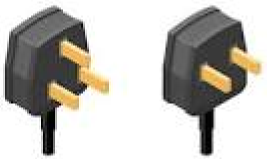
- The adapter pattern lets classes work together that could not otherwise because of incompatibl interfaces
  - “Convert the interface of a class into another interface expected by a client class.”
  - Used to provide new interfaces to existing legacy components (Interface engineering, reengineering).



Object adapter:

- Uses single inheritance and delegation

Adapter Pattern



- The adapter pattern lets classes work together that could not otherwise because of incompatibl interfaces
  - "Convert the interface of a class into another interface expected by a client class."
  - Used to provide new interfaces to existing legacy components (Interface engineering, reengineering).

Object adapter:

- Uses single inheritance and delegation

Adapter Pattern

Name

- Adapter

Intent

- Convert the interface of a class into another interface clients expect.
- Adapter lets classes work together, that could not otherwise because of incompatible interfaces.

Also Known As:

- Wrapper

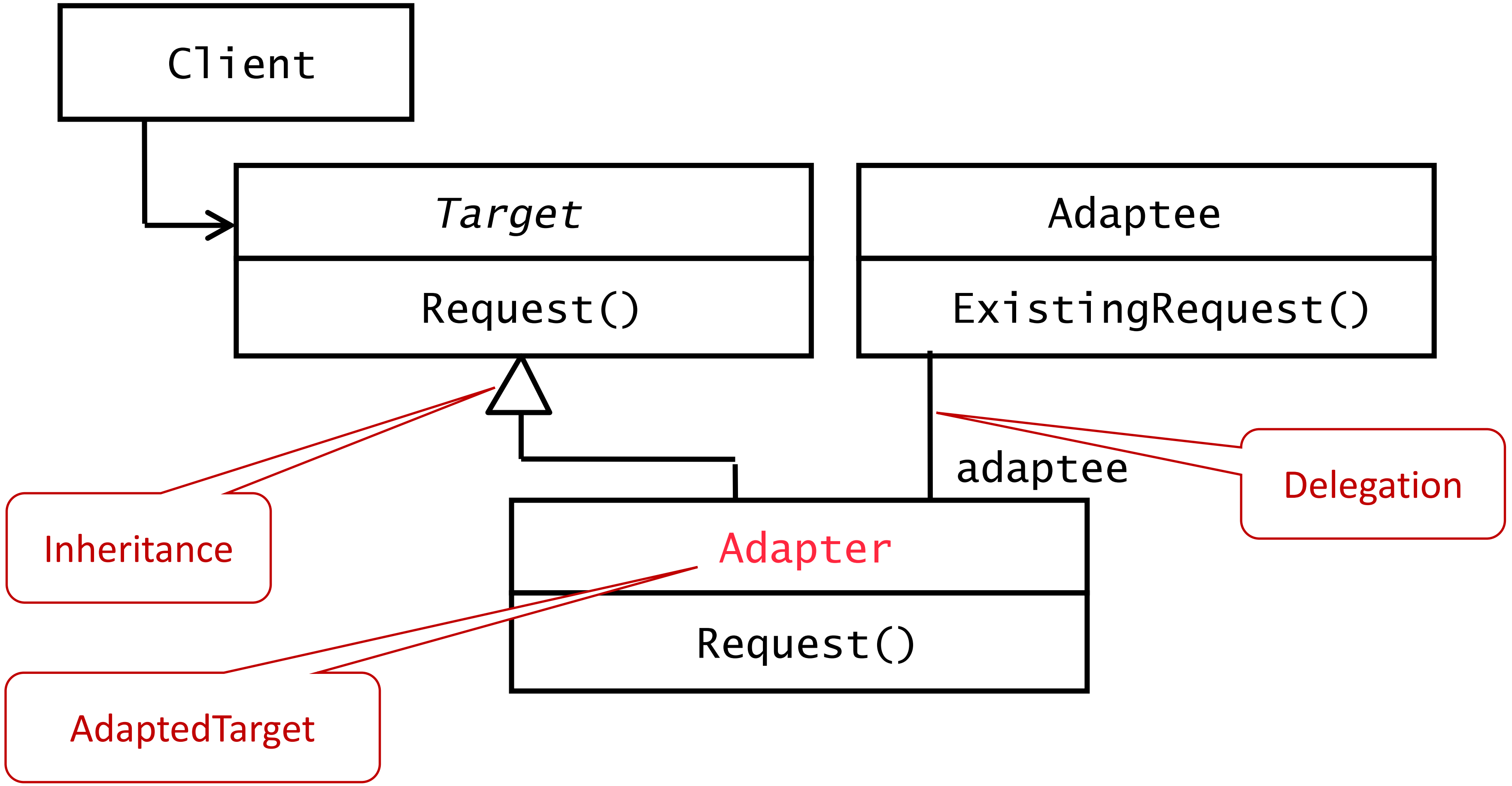
Adapter Pattern

Motivation

- sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application
- we can not change the library interface, since we may not have its source code
- even if we did have the source code, we ✓ probably should not change the library for each domain-specific application

Adapter Pattern

Structure



Adapter Pattern

Participants

- Target - defines the domain-specific interface that Client uses.
- Adapter - adapts the interface Adaptee to the Target interface.
- Adaptee - defines an existing interface that needs adapting.
- Client - collaborates with objects conforming to the Target interface.

Applicability

Use the Adapter pattern when:

- You want to use an existing class, and its interface does not match the one you need
- You want to create a reusable class that cooperates with unrelated classes with incompatible interfaces

The Adapter pattern, Example

- Here, we have two incompatible interfaces : MediaPlayer and MediaPackage. MP3 class is an implementation of the MediaPlayer interface and we have VLC and MP4 as implementations of the MediaPackage interface.
- We want to use MediaPackage implementations as MediaPlayer instances. So, we need to create an adapter to help to work with two incompatible classes.
- The Adapter will be named FormatAdapter and must implement the MediaPlayer interface. Furthermore, the FormatAdapter class must have a reference to MediaPackage, the incompatible interface.

The Adapter pattern: Example

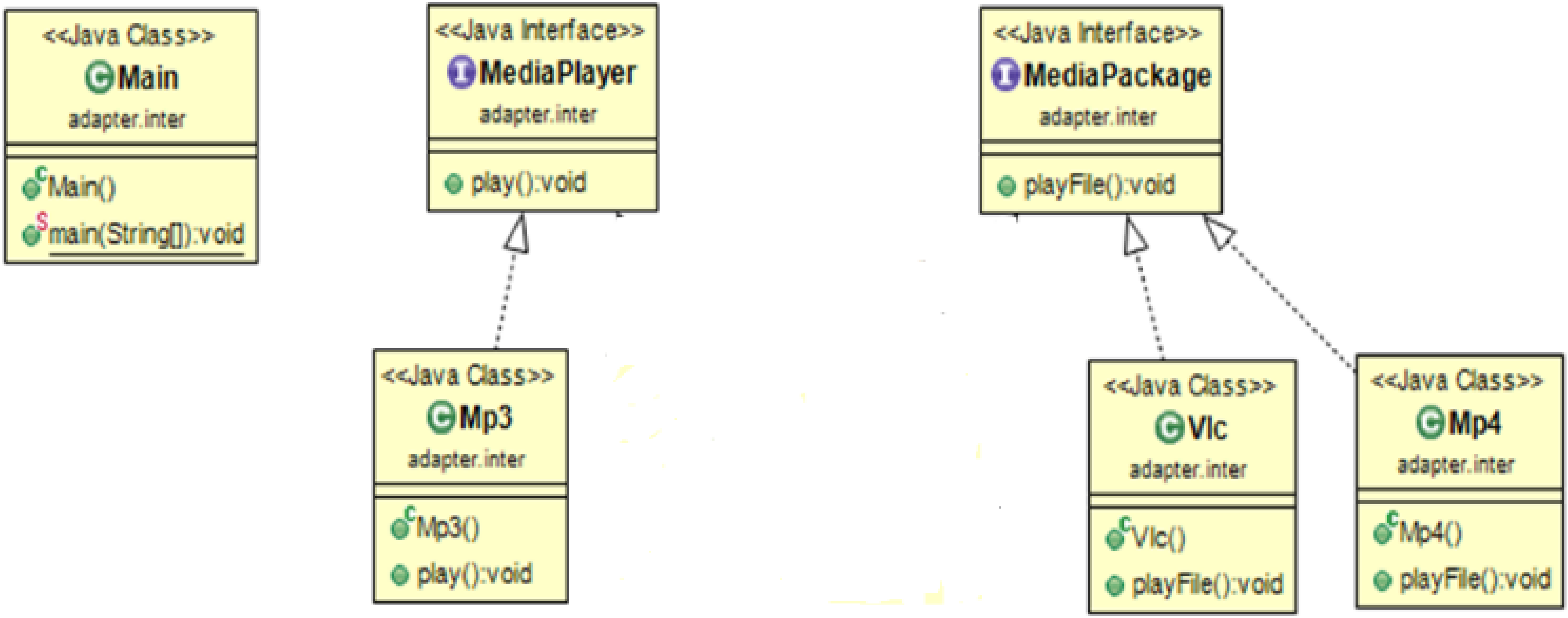


Figure from medium.com

The Adapter pattern: Example

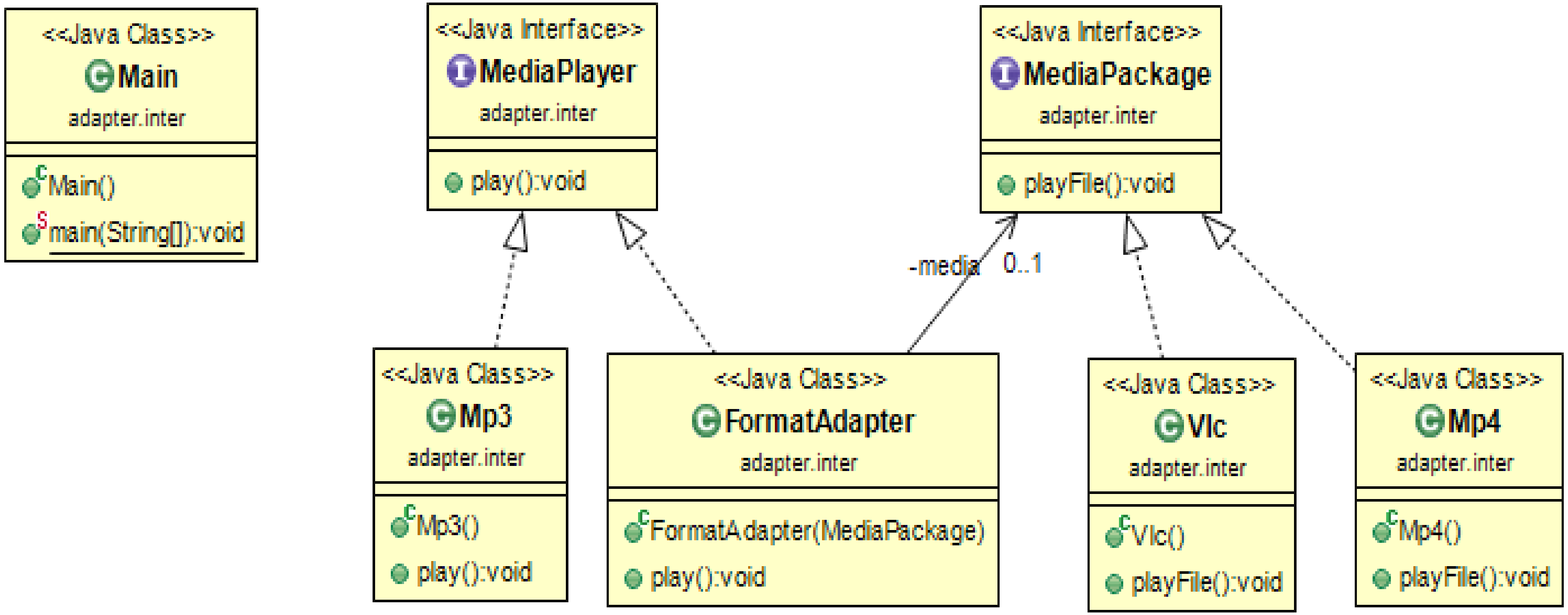


Figure from medium.com

Java Code:

```
//MediaPlayer.java
public interface MediaPlayer {
 void play(String filename);}
```

```
//MediaPackage.java
 public interface MediaPackage {
 void playFile(String filename);
}
```

```
//MP3.java
public class MP3 implements MediaPlayer {
 @Override
 public void play(String filename) {
   System.out.println("Playing MP3 File " + filename);
 }
}
```

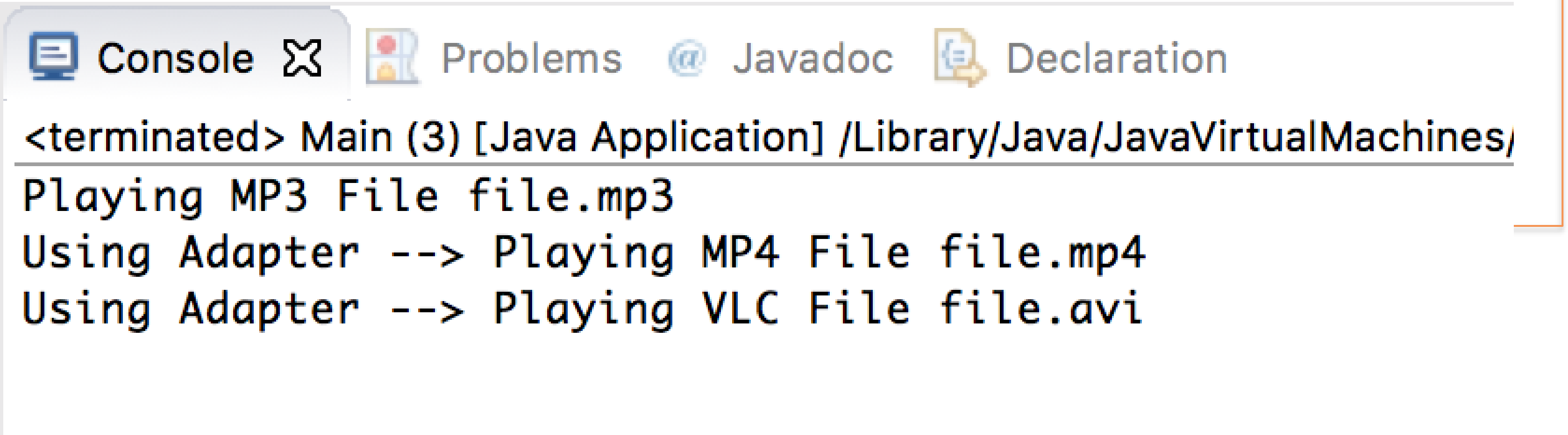
```java
//MP4.java
public class MP4 implements MediaPackage {
 @Override
 public void playFile(String filename) {
   System.out.println("Playing MP4 File " + filename);
 }
}
```

```java
//VLC.java
public class VLC implements MediaPackage {
 @Override
 public void playFile(String filename) {
   System.out.println("Playing VLC File " + filename);
 }
}
```

//FormatAdapter.java

```
public class FormatAdapter implements MediaPlayer {
 private MediaPackage media; //composition
 public FormatAdapter(MediaPackage m) {
    media = m;
 }
 @Override
 public void play(String filename) {
   System.out.print("Using Adapter --> ");
   media.playFile(filename); // deligation
 }
```

```java
public class Main {
 public static void main(String[] args) {
   MediaPlayer player = new MP3();
   player.play("file.mp3");
   player = new FormatAdapter(new MP4());
   player.play("file.mp4");
   player = new FormatAdapter(new VLC());
   player.play("file.avi");
 }
}
```



```
<terminated> Main (3) [Java Application] /Library/Java/JavaVirtualMachines/
Playing MP3 File file.mp3
Using Adapter --> Playing MP4 File file.mp4
Using Adapter --> Playing VLC File file.avi
```

POP Quiz:

What is the **open/closed principle in Object Oriented design?**

The Open-closed principle is one of the five SOLID principles of object-oriented design. SOLID is an **acronym** for the five **object-oriented design** principles. What are they?

SOLID – OOD PRINCIPLES

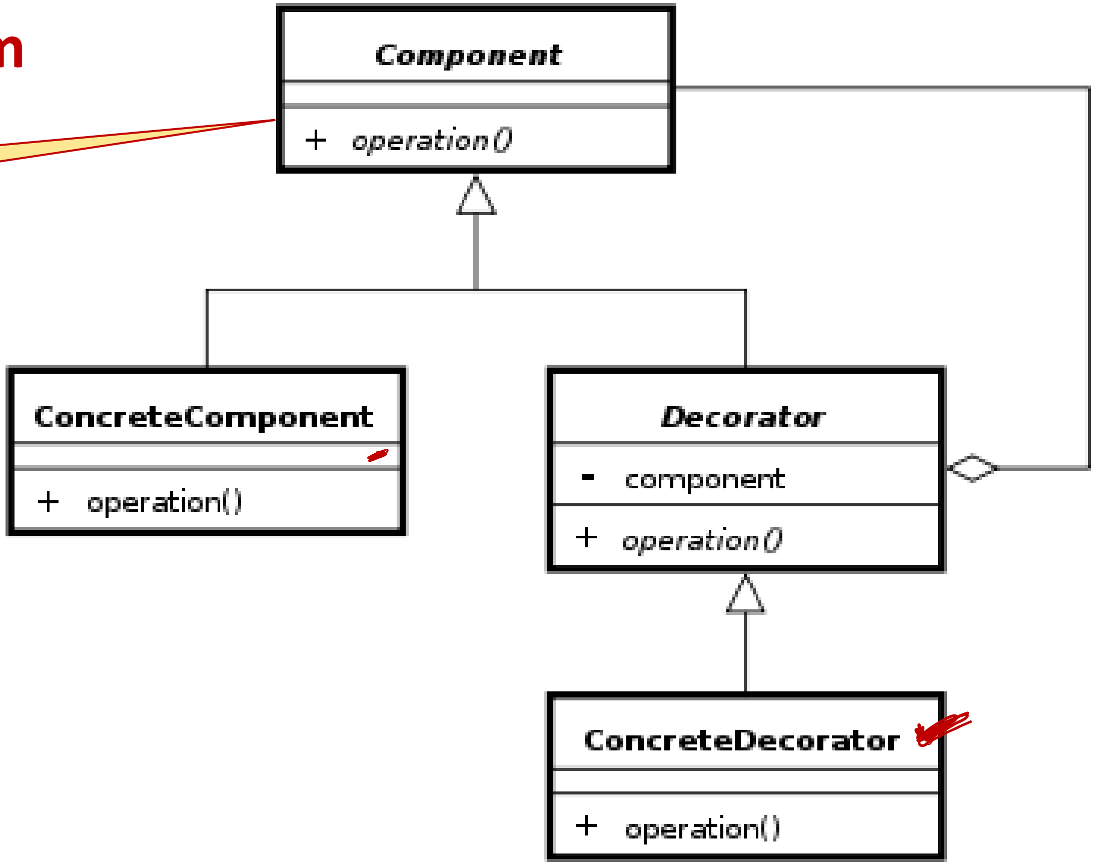
- **S** - Single-responsiblity principle
- **O** - Open-closed principle
- **L** - Liskov substitution principle
- **I** - Interface segregation principle
- **D** - Dependency Inversion Principle

Decorator Pattern (Structural Pattern)

- **Definition**

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

- **UML Class Diagram**



Abstract class,

if you need to define initial behavior

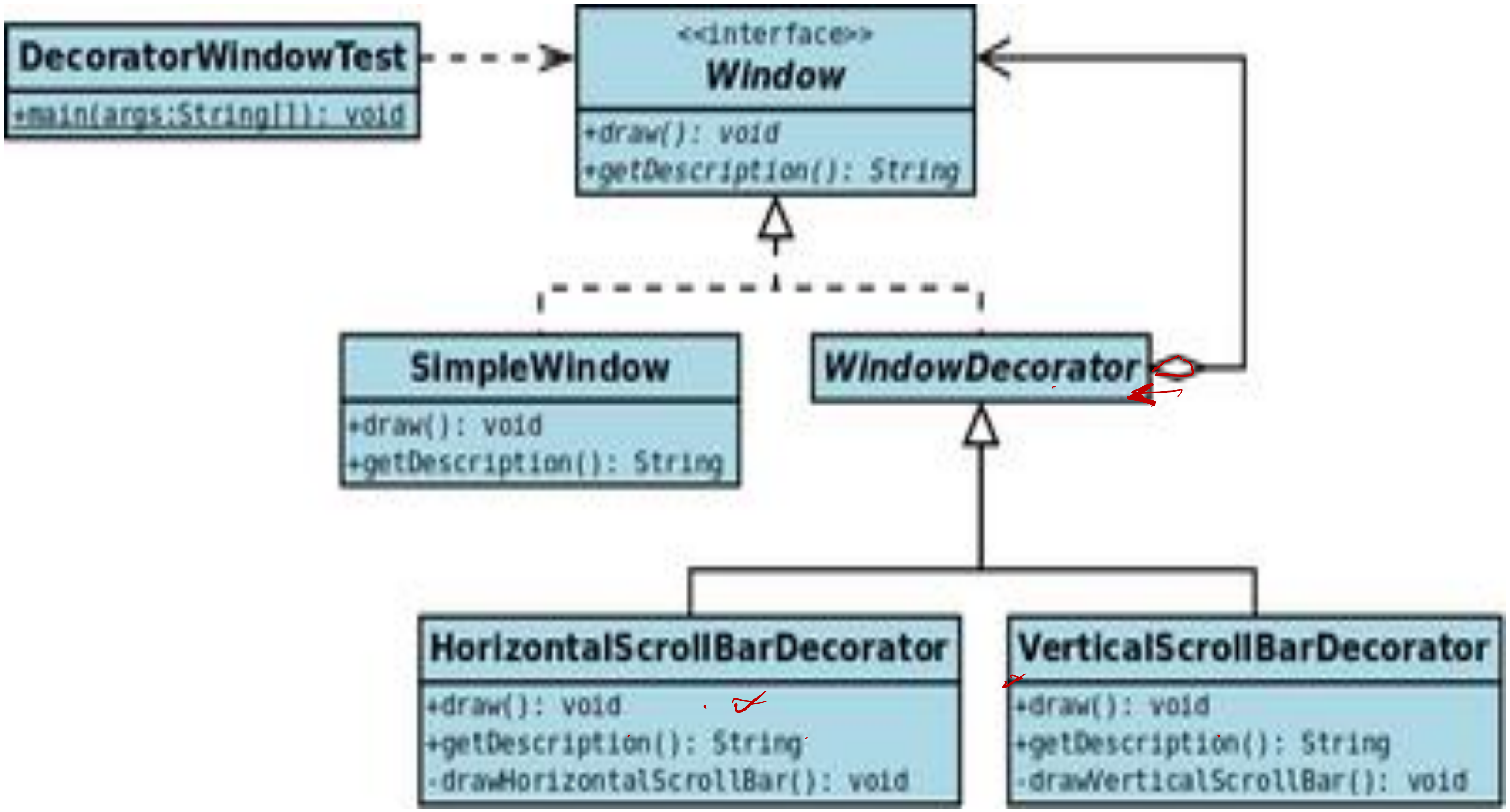
```
public abstract class ShapeDecorator implements Shape
{
protected Shape decoratedShape;//composition,i.e has-a

 public ShapeDecorator(Shape decoratedShape) {
     super();
     this.decoratedShape = decoratedShape;
  }
}
```

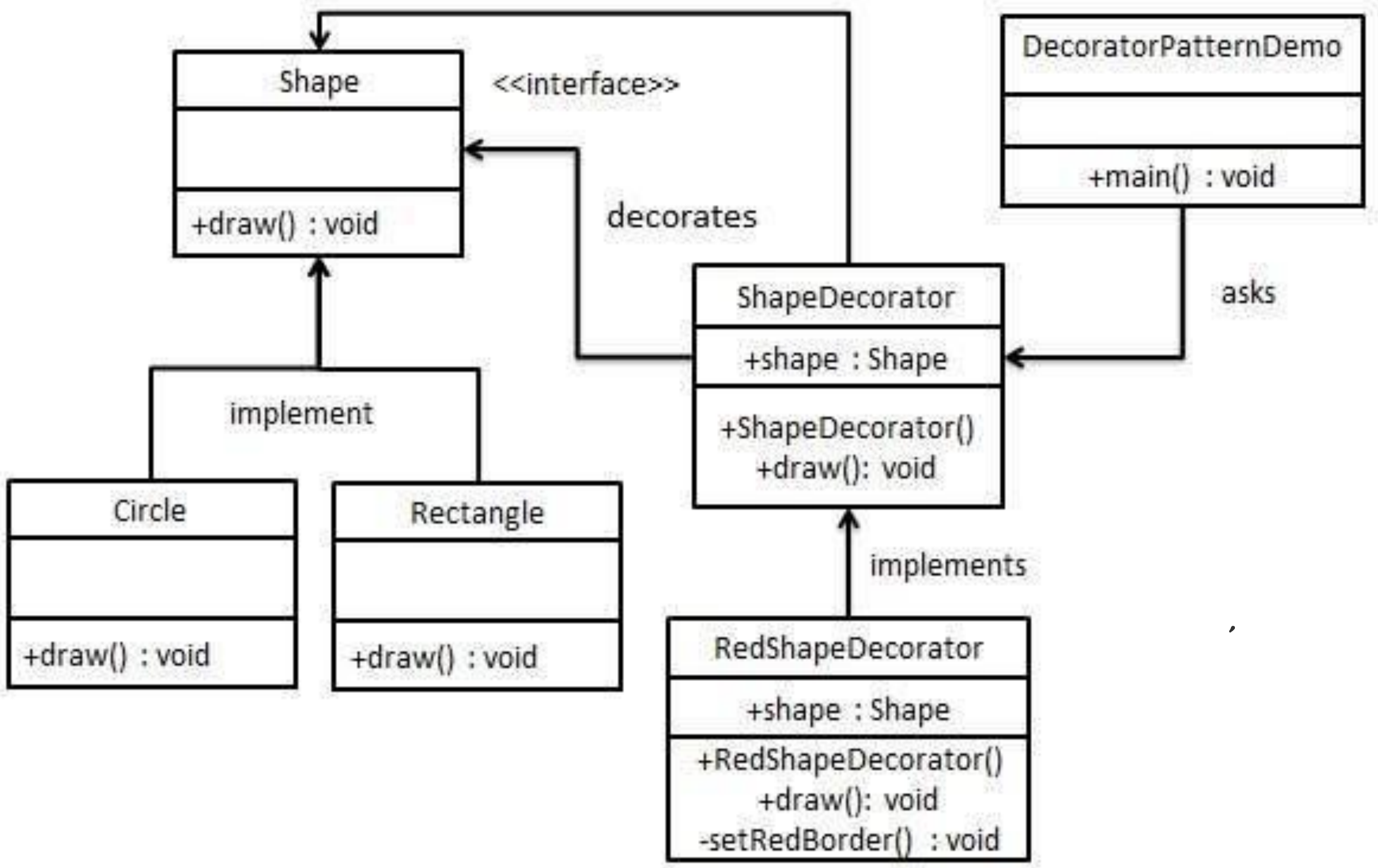
Client :-

```
Shape circle1 = new FillColorDecorator(new LineColorDecorator(new
LineStyleDecorator(new LineThinknessDecorator(new Circle(), 2.0d), LineStyle.DASH),
Color.BLUE), Color.RED);

circle1.draw();
```

```
Circle c = new Circle();

LineThinknessDecorator lt = new LineThinknessDecorator(c, 2.0d);
LineStyleDecorator ls = new LineStyleDecorator(lt, LineStyle.DASH);
LineColorDecorator lc = new LineColorDecorator(ls, Color.BLUE);
FillColorDecorator fc = new FillColorDecorator(lc, Color.RED);
Shape circle3 = fc;

circle3.draw();
```

Decorator Pattern, Example: we may want to add both a horizontal and a vertical scroll bar to a window object.


DecoratorWindowTest
+main(args:String[]): void
<<interface>>
Window
+draw(): void
+getDescription(): String
SimpleWindow
+draw(): void
+getDescription(): String
WindowDecorator
HorizontalScrollBarDecorator
+draw(): void
+getDescription(): String
-drawHorizontalScrollBar(): void
VerticalScrollBarDecorator
+draw(): void
+getDescription(): String
-drawVerticalScrollBar(): void



Example from Tutorialpoint.com

```
//Shape.java
public interface Shape { void draw(); }
```

```
//Create concrete classes implementing the
//same interface.
//Rectangle.java
public class Rectangle implements Shape { @Override
public void draw() { System.out.println("Shape:
Rectangle"); } }
```

```
// class Circle
public class Circle implements Shape {
@Override public void draw() {
 System.out.println("Shape: Circle");
 }
 }
```

```java
//ShapeDecorator.java
public abstract class ShapeDecorator implements Shape {
 protected Shape decoratedShape; // has-a (composition)
public ShapeDecorator(Shape decoratedShape){
 this.decoratedShape = decoratedShape;
}
public void draw(){ decoratedShape.draw(); }
 }
```

RedShapeDecorator.java

```
public class RedShapeDecorator extends ShapeDecorator {
public RedShapeDecorator(Shape decoratedShape)
{ super(decoratedShape); }
@Override public void draw() {
decoratedShape.draw();
setRedBorder(decoratedShape); }
private void setRedBorder(Shape decoratedShape){
System.out.println("Border Color: Red"); }
 }
```

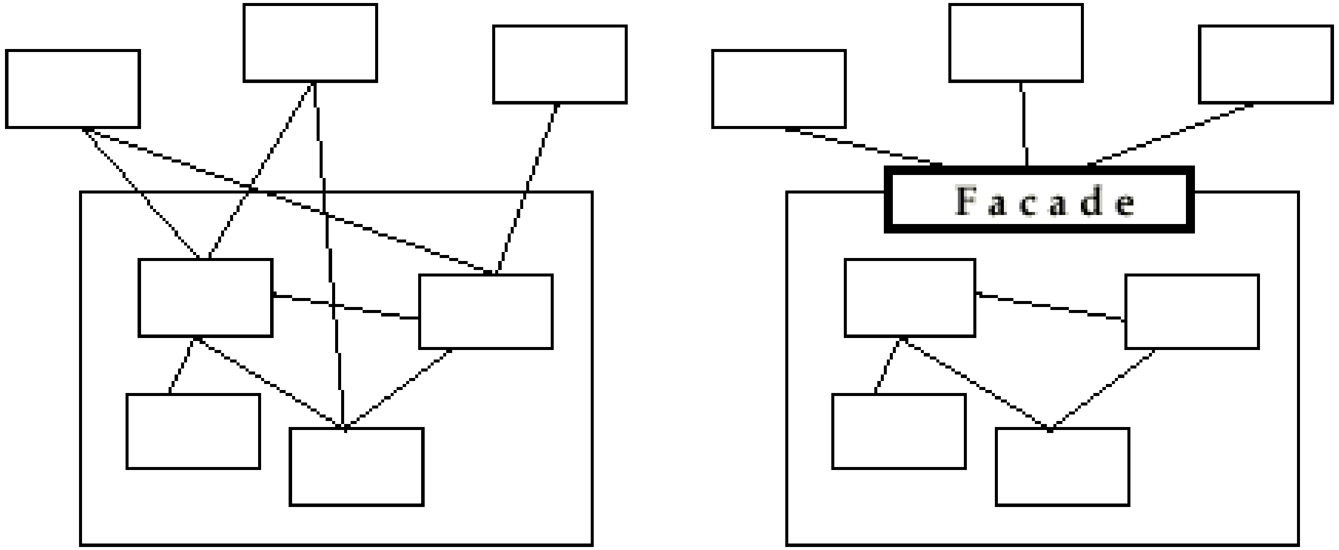
Outline

We will take a closer look at:

- Façade Pattern
- Factory Method
- Abstract Factory Pattern

Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the "gory details")
- Facades allow us to provide a closed architecture



When to use the facade pattern?

- A facade should be offered by all subsystems in a software system that offer services to other subsystems
  - The facade delegates requests to the appropriate components within the subsystem. The facade usually does not have to be changed, when the components are changed.

When to use the facade pattern?

Consequences

Benefits

- It hides the implementation of the subsystem from clients, making the subsystem easier to use
- It promotes weak coupling between the subsystem and its clients. This allows you to change the classes the comprise the subsystem without affecting the clients.
- It reduces compilation dependencies in large software systems
- It simplifies porting systems to other platforms, because it's less likely that building one subsystem requires building all others

When to use the facade pattern?

Benefits (cont.)

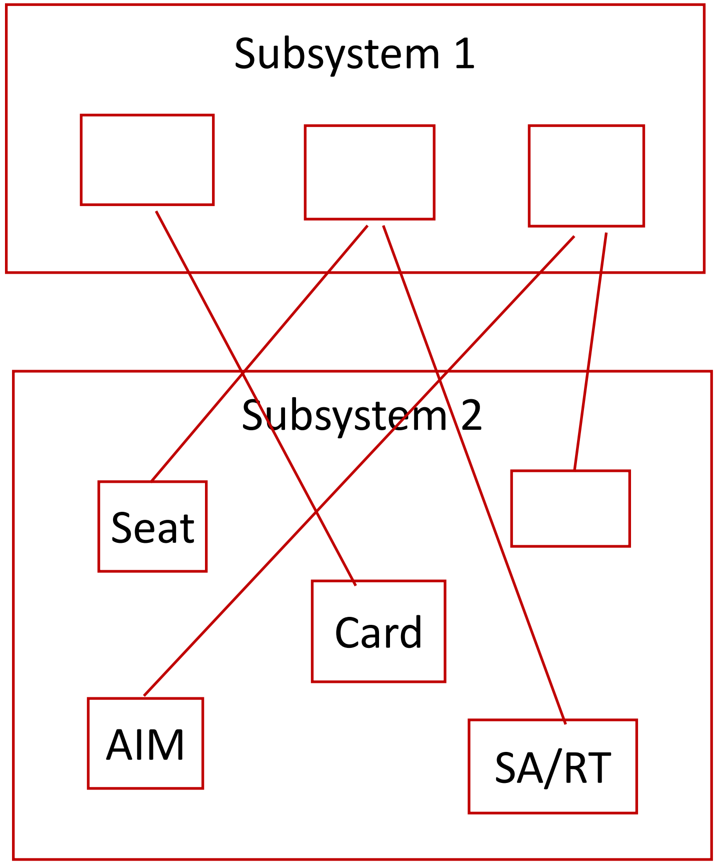
- It does not prevent sophisticated clients from accessing the underlying classes
- Note that Facade does not add any functionality, it just simplifies interfaces

Liabilities

- It does not prevent clients from accessing the underlying classes!

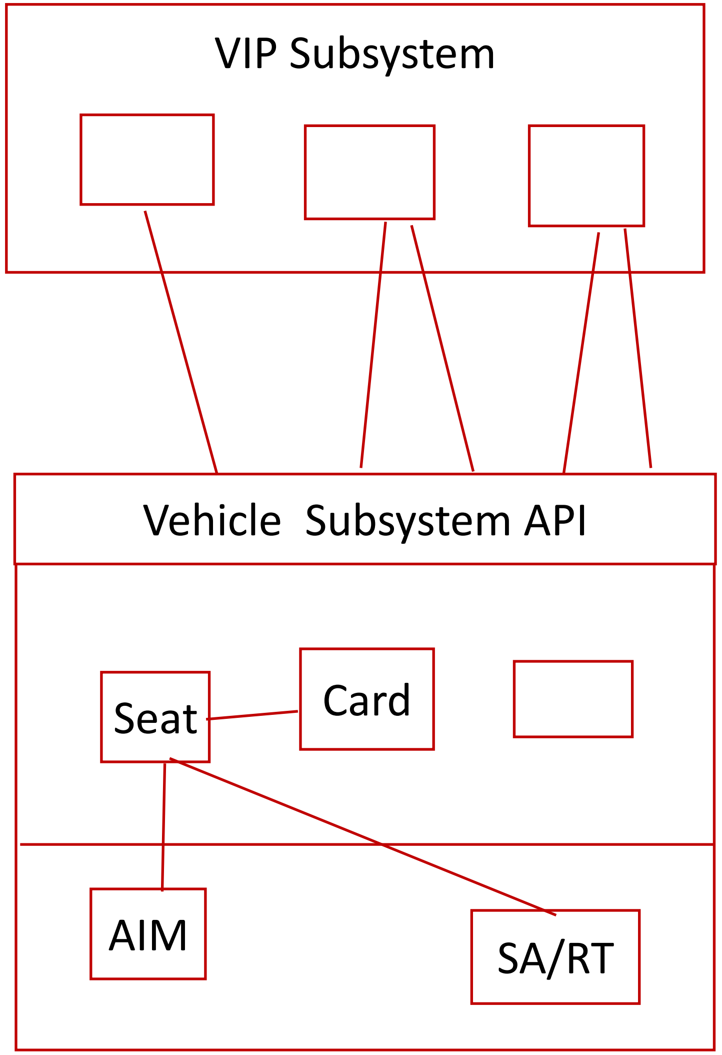
Design Example

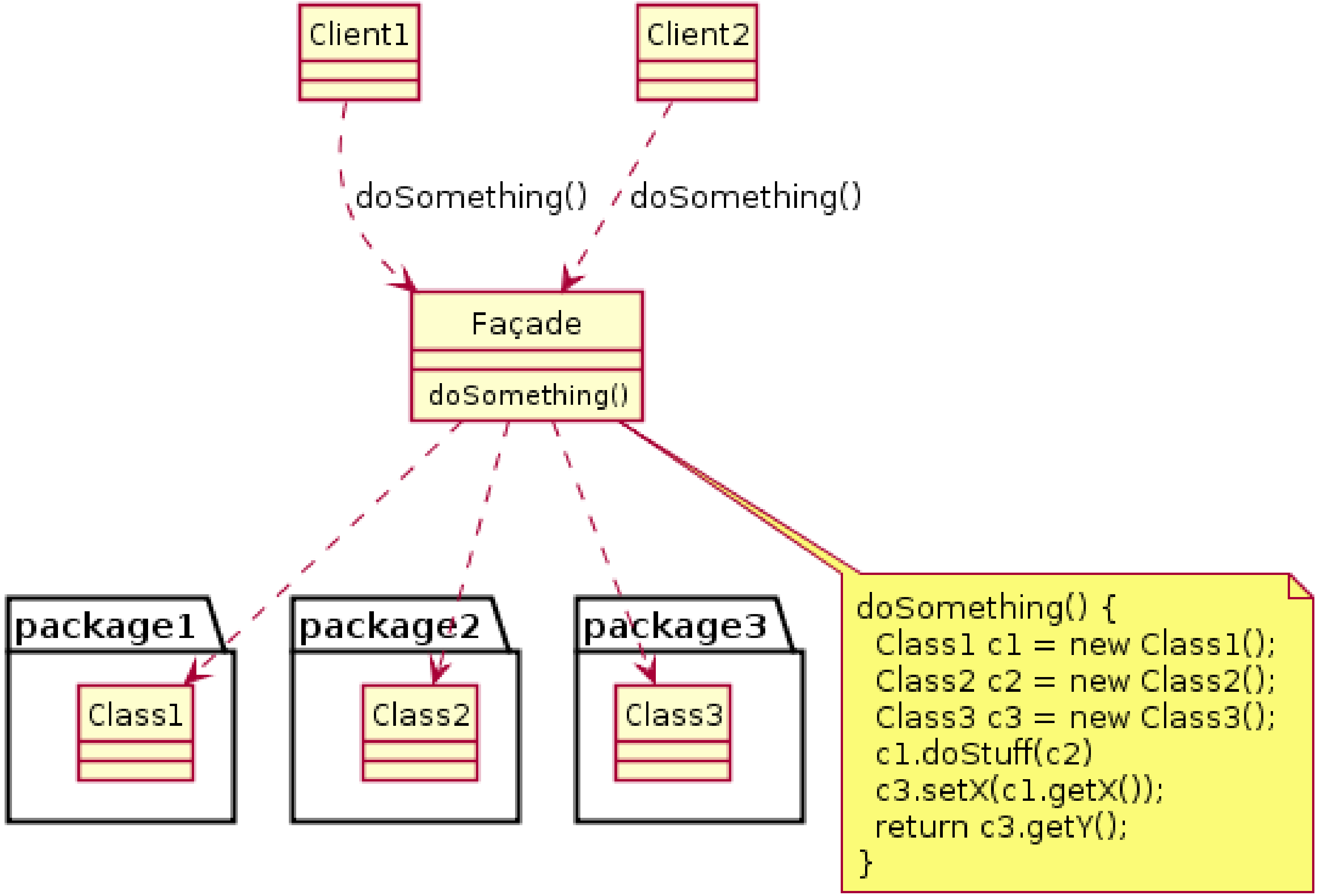
- Subsystem 1 can look into the Subsystem 2 (vehicle subsystem) and call on any component or class operation at will.
- This is a “Spaghetti Design”
- Why is this good?
  - Efficiency
- Why is this bad?
  - Can’t expect the caller to understand how the subsystem works or the complex relationships within the subsystem.
  - We can be assured that the subsystem will be misused, leading to non-portable code!



Realizing an Opaque Architecture with a Facade

- The subsystem decides exactly how it is accessed
- No need to worry about misuse by callers
- If a facade is used, the subsystem can be used in an early integration test
  - We need to write only a driver





```
doSomething() {
  Class1 c1 = new Class1();
  Class2 c2 = new Class2();
  Class3 c3 = new Class3();
  c1.doStuff(c2)
  c3.setX(c1.getX());
  return c3.getY();
}
```

When to use ……..

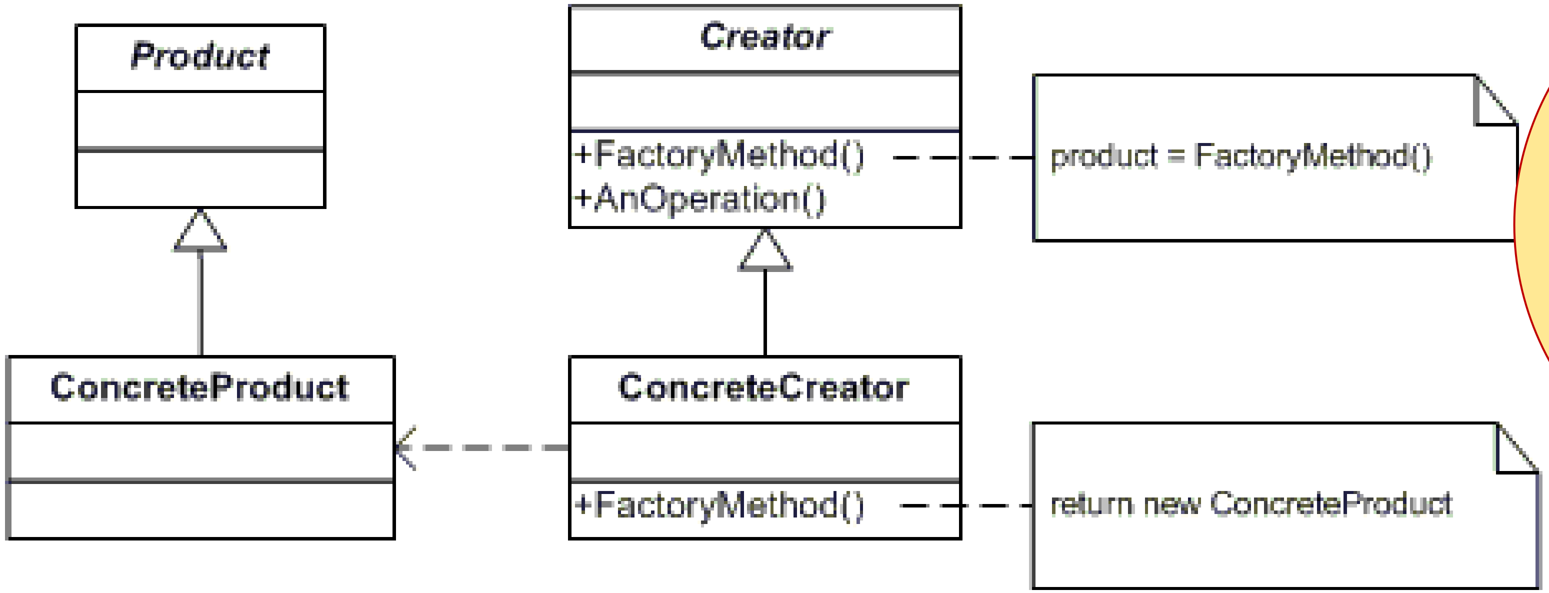
- A **decorator** makes it possible to add or alter behavior of an interface at run-time.
- An **Adapter** can be used when the wrapper must respect a particular interface and must support polymorphic behavior, and
- **Facade** when an easier or simpler interface to an underlying object is desired

Factory Method Pattern

- The factory pattern is used to replace class **constructors,** abstracting the process of object generation so that the type of the object instantiated can be determined at run-time.

Factory Method Pattern

- Related to Abstract Factory
- Product is an interface
- ConcreteProduct instances can be created
- FactoryMethod() of the Creator interface returns a Product object, but which ConcreteProduct is actually created (the actual constructor call) is hidden in the ConcreteCreator



```
//Object instantiation
is here.
if (cond)
{ product=
new ConcreateProduct1();
if (….)
product=
new ConcreateProduct1();
```

Factory Method Pattern: Example 1

- Assume you have different types of users.

You can create a user object in two ways:

1- User user = new user(?????);

2- user = DataFactory.create(???);

For the second case you need to define the factory class

```
Class DataFactory{
Public static Object create(UserType objType) // User type is enumeration
Switch (objType)
Case user: return new User();
             break;
Case admin: return new Admin();
               break;
//cases for other object types here…….}
```

```
public class ShapeFactory {
//use getShape method to get object of type shape
public Shape getShape(String shapeType)
  { if(shapeType == null)
            { return null; }
   if(shapeType.equalsIgnoreCase("CIRCLE"))
       { return new Circle(); }
  else if(shapeType.equalsIgnoreCase("RECTANGLE"))
       { return new Rectangle(); }
  else if(shapeType.equalsIgnoreCase("SQUARE"))
       { return new Square(); }
  return null; } }
```

Abstract Factory Pattern

Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- The Abstract Factory pattern is very similar to the Factory Method pattern.
- One difference between the two is that with the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object via **composition** whereas the Factory Method pattern uses **inheritance** and relies on a subclass to handle the desired object instantiation.
- Actually, the delegated object frequently uses factory methods to perform the instantiation!

Abstract Factory Pattern

Motivation

- Each platform is represented by a Factory class, with concrete subclasses under it. Each concrete subclass support a platform concept (e.g, window, button, slider, menu).

- The Factory class contains methods for ‘creating’ or ‘instantiating’ a concrete type below it. Thus, when the platform is changed, only the Factory class methods have to be reworked to conform to the new platform(s) concepts.

Abstract Factory Pattern

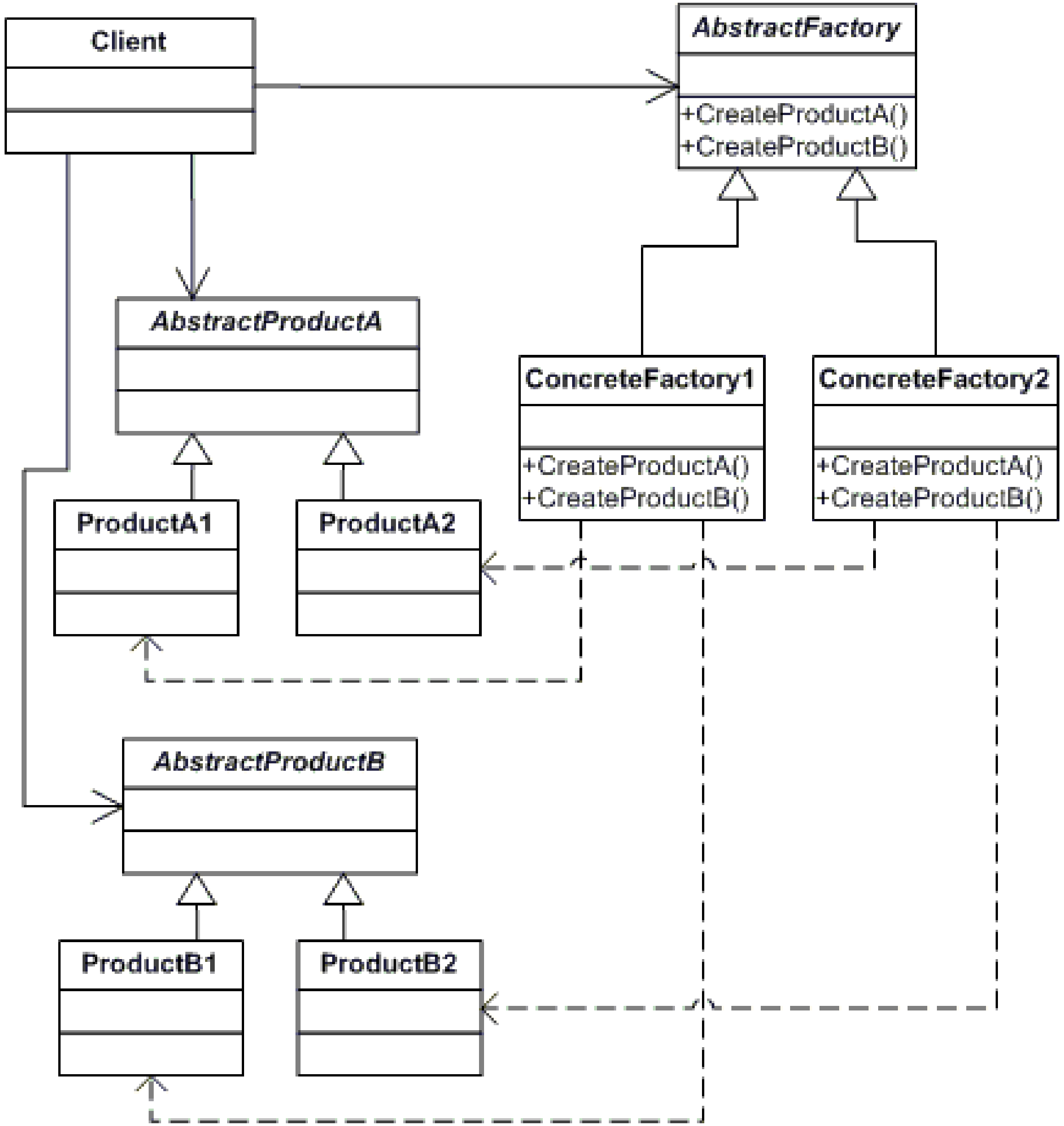
Applicability

Use the Abstract Factory pattern in any of the following situations:

- A system should be independent of how its products are created, composed, and represented
- A class can't anticipate the class of objects it must create
- A system must use just one of a set of families of products
- A family of related product objects is designed to be used together, and you need to enforce this constraint

Abstract Factory Pattern: UML Diagram

Structure



Abstract Factory Pattern

Participants

- AbstractFactory
  - Declares an interface for operations that create abstract product objects
- ConcreteFactory
  - Implements the operations to create concrete product objects

- AbstractProduct
  - Declares an interface for a type of product object
- ConcreteProduct
  - Defines a product object to be created by the corresponding concrete factory
  - Implements the AbstractProduct interface

- Client
  - Uses only interfaces declared by AbstractFactory and AbstractProduct classes

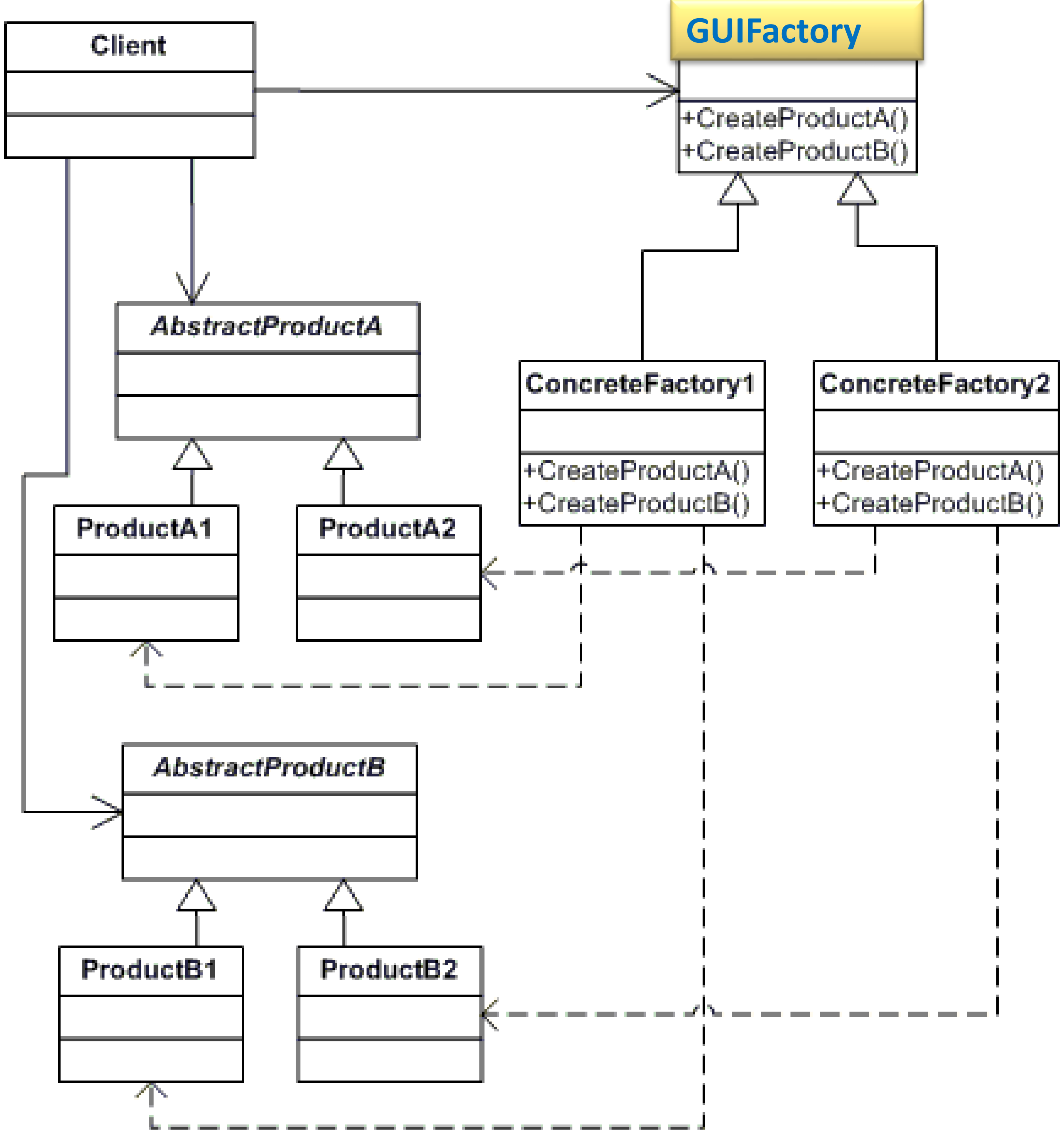
Abstract Factory Pattern

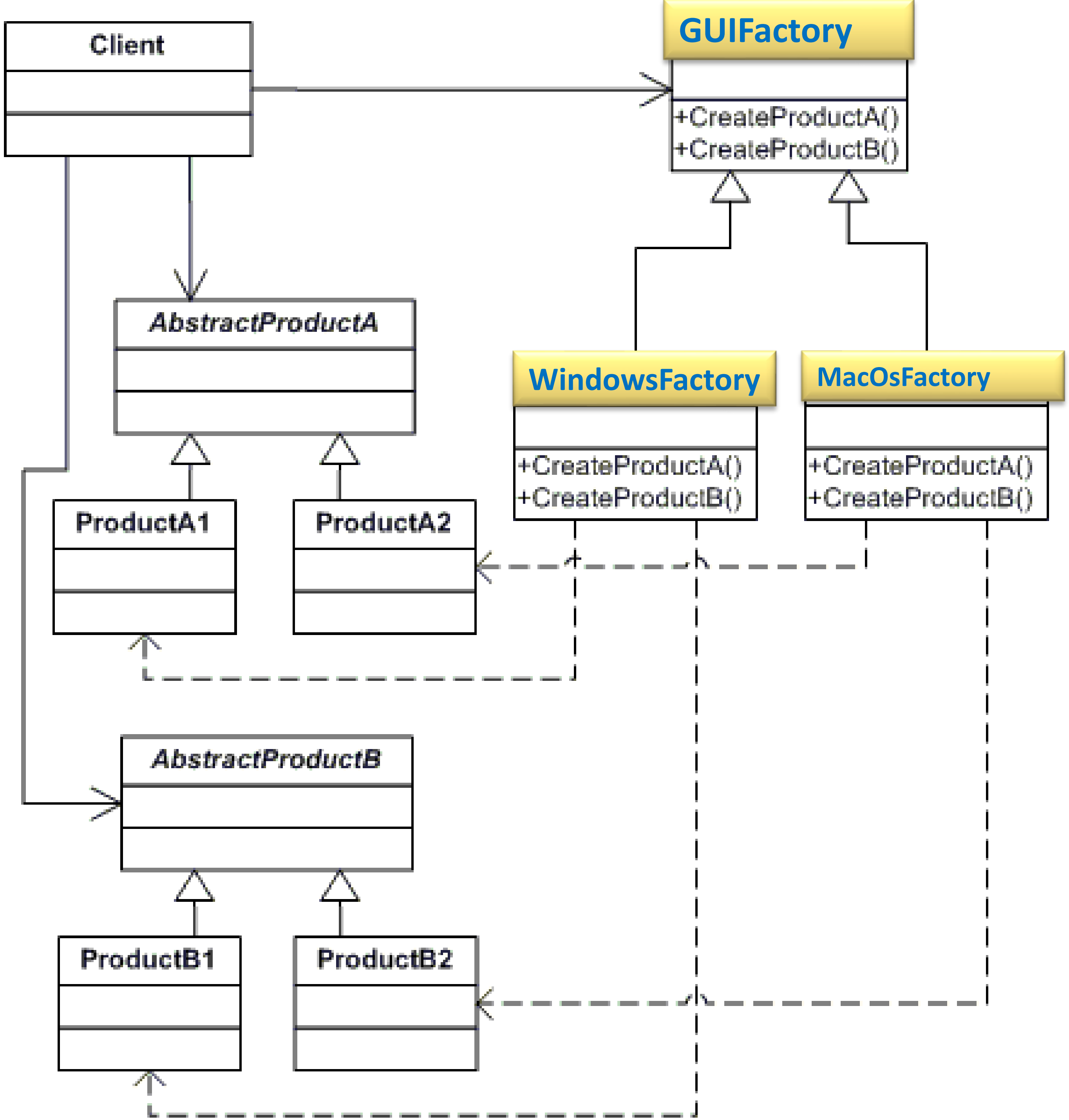
Collaborations

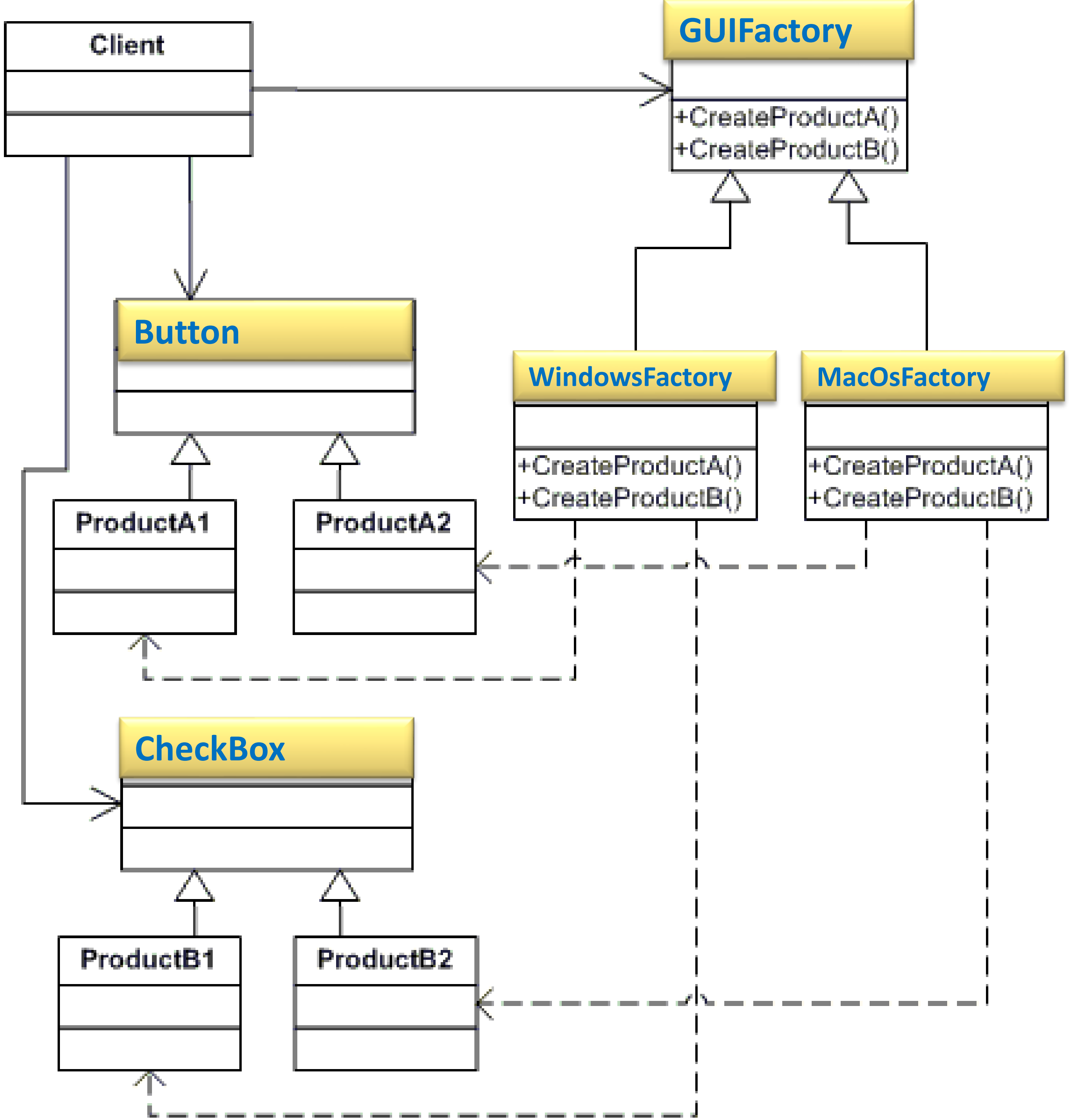
- Normally a single instance of a ConcreteFactory class is created at runtime. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.

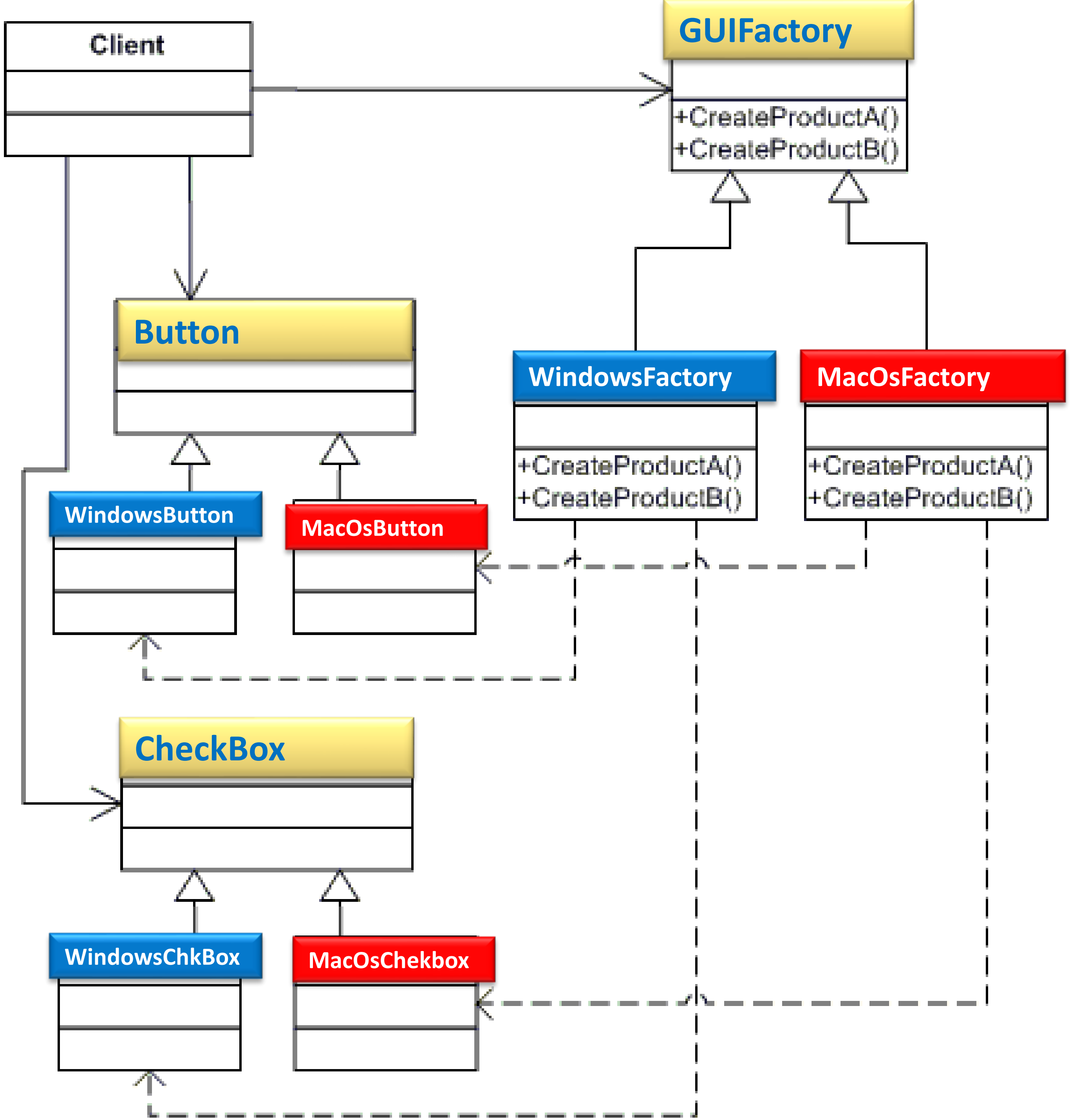
- AbstractFactory defers creation of product objects to its ConcreteFactory.

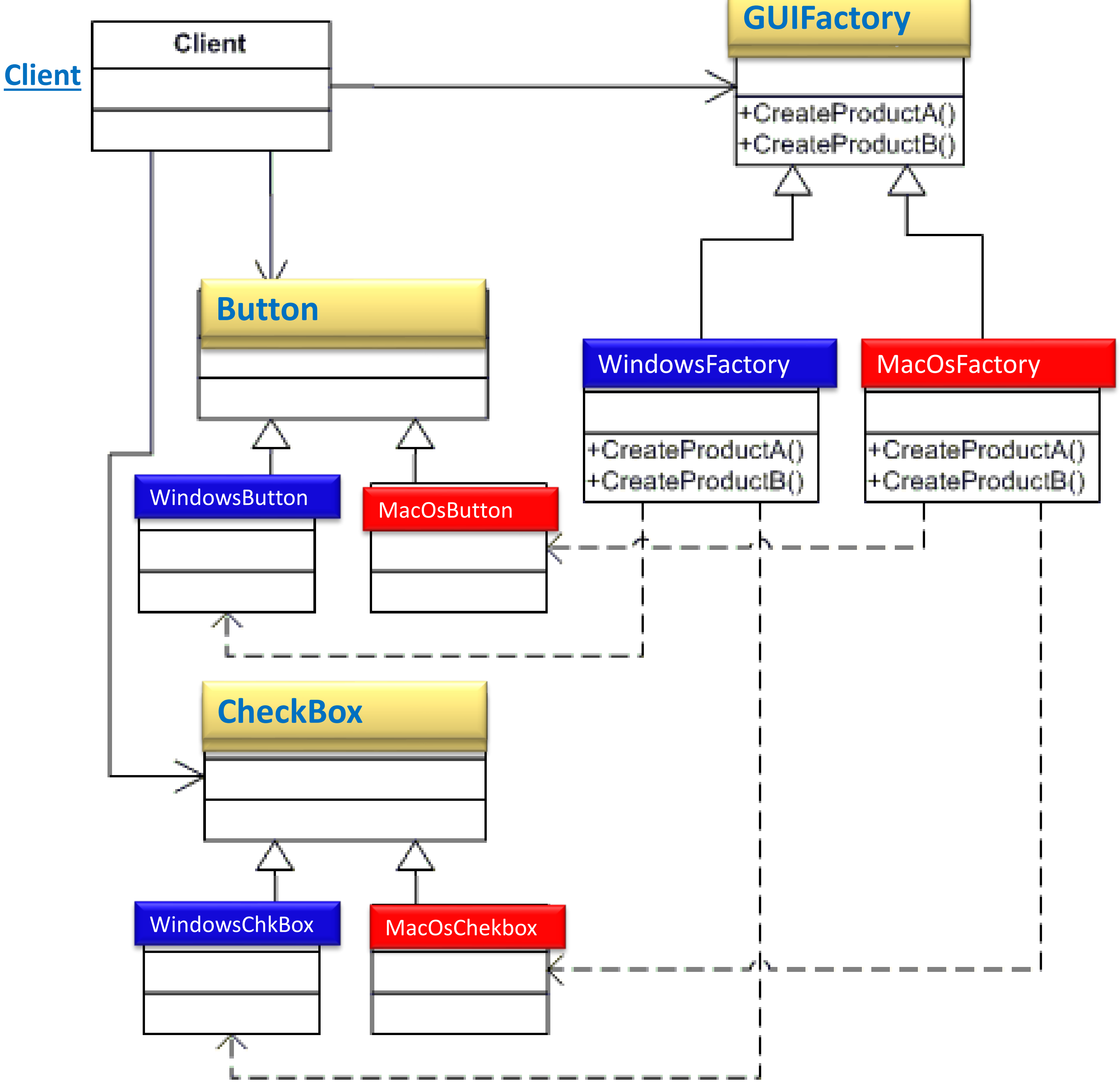
Abstract Factory Example




GUIFactory
+CreateProductA()
+CreateProductB()
Client
AbstractProductA
WindowsFactory
+CreateProductA()
+CreateProductB()
MacOsFactory
+CreateProductA()
+CreateProductB()
ProductA1
ProductA2
AbstractProductB
ProductB1
ProductB2

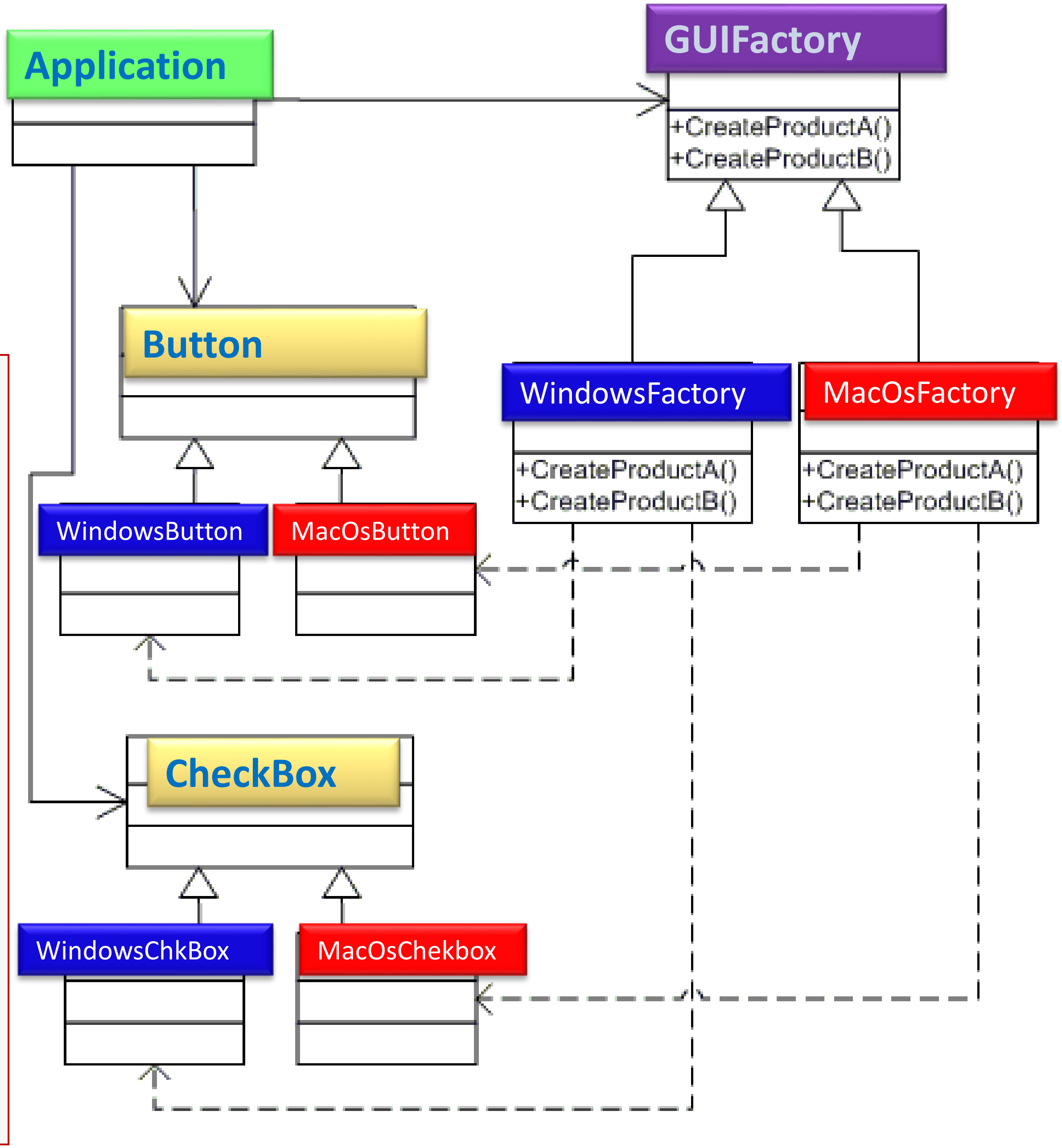

Client
GUIFactory
+CreateProductA()
+CreateProductB()
Button
WindowsFactory
+CreateProductA()
+CreateProductB()
MacOsFactory
+CreateProductA()
+CreateProductB()
ProductA1
ProductA2
CheckBox
ProductB1
ProductB2


Client
GUIFactory
+CreateProductA()
+CreateProductB()
Button
WindowsFactory
+CreateProductA()
+CreateProductB()
MacOsFactory
+CreateProductA()
+CreateProductB()
WindowsButton
MacOsButton
CheckBox
WindowsChkBox
MacOsChekbox


Client
Client
GUIFactory
+CreateProductA()
+CreateProductB()
Button
WindowsFactory
MacOsFactory
+CreateProductA()
+CreateProductB()
+CreateProductA()
+CreateProductB()
WindowsButton
MacOsButton
CheckBox
WindowsChkBox
MacOsChekbox

```
public Application (GUIFactory factory)
{private Button button;
private Checkbox checkbox;
button = factory.createButton();
checkbox = factory.createCheckbox();     }
public void paint() {
button.paint();
checkbox.paint();     }
```

```
public class Demo {    /**      * Application picks the
factory type and creates it in run time (usually at     *
initialization stage), depending on the configuration or
* environment variables.      */
 private static Application configureApplication() {
Application app;
GUIFactory factory;
String osName = System.getProperty("os.name").toLowerCase();
if (osName.contains("mac")) {
factory = new MacOSFactory();
app = new Application(factory);
}
else {
factory = new WindowsFactory();
app = new Application(factory);        }
return app;     }
public static void main(String[] args) {
Application app = configureApplication();
app.paint();     }}
```



```
/** * Each concrete factory extends basic factory and
responsible for creating * products of a single variety.
*/
public class MacOSFactory implements GUIFactory {
@Override
public MacOSButton createButton() {
return new MacOSButton();    }
@Override
public MacOSCheckbox createCheckbox() {
return new MacOSCheckbox();    }}
```

Additional readings

A collection of links to pages on Design Patterns:

- **Design patterns tutorial** from dofactory.com.
- **Overview of Design Patterns**, Mark Grand.
- *The Design Patterns. Java Companion*, a book by James Cooper, available on-line. Several web sites have a copy of this book free of charge.
- **The Hillside Group's Patterns Home Page.** contains a wide collection of pattern resources (papers, books, software, and other information).

Summary

- Design patterns are partial solutions to common problems such as:
  - separating an interface from a number of alternate implementations,
  - wrapping around a set of legacy classes,
  - protecting a caller from changes associated with specific platforms.
- A design pattern consists of a small number of classes
  - uses delegation and inheritance,
  - provides a modifiable design solution.
- These classes can be adapted and refined for the specific system under construction
  - customization of the system,
  - reuse of existing solutions.

Books

- *Design Patterns*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995.
- *Design Patterns in Java*, by Steven John Metsker and William C. Wake, Addison-Wesley, 2006.
- *Head First Design Patterns*. By Eric Freeman, Bert Bates , Kathy Sierra, Elisabeth Robson, O'Reilly, 2004