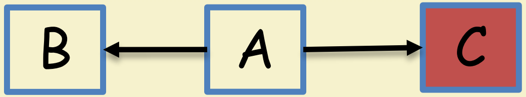
Adapter Pattern

Adapter Pattern

Intent:



- **Convert the interface of a class to the interface expected by the users of the class.**
- Allows classes to work together even when they have incompatible interfaces.

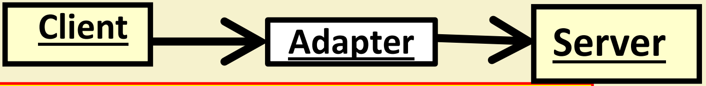
Example (non-software):

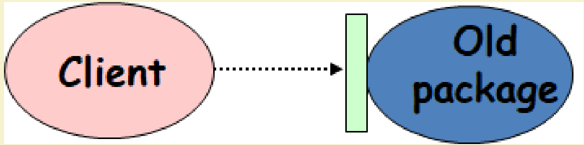
- You went to U.S.
- Had an Indian electrical appliance.
- How can you use it in U.S.?
- **Use Adapters!**

Also universal adapters?

Adapter Pattern

- A **wrapper pattern**

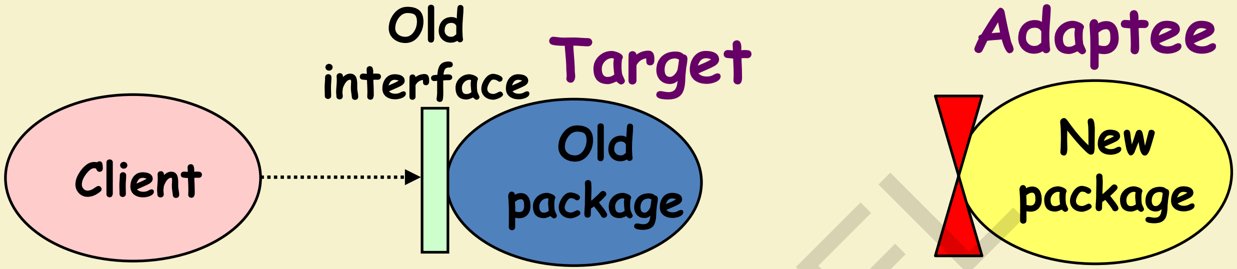


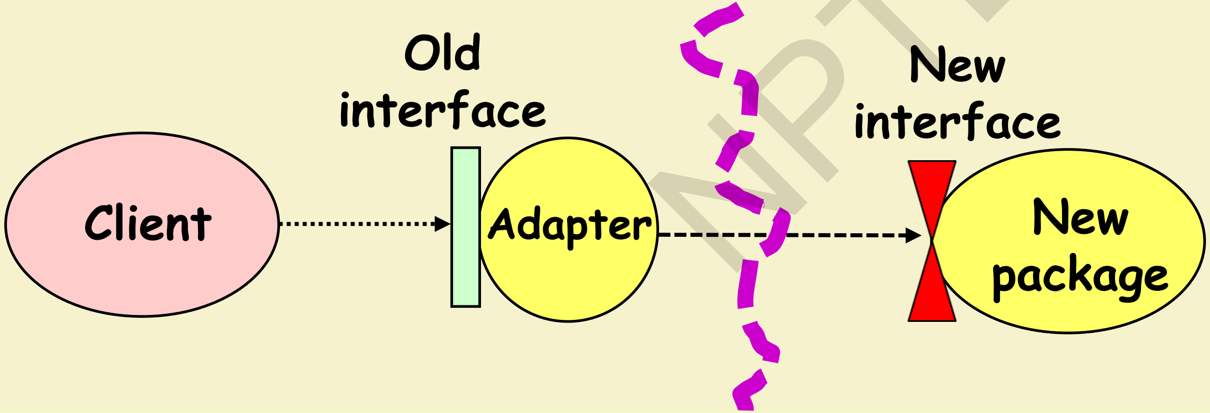


- **Problem: Convert the interface of a class into one that a client expects.**
  - Lets classes work together --- that couldn't otherwise --- because of incompatible interfaces
  - Used to provide a new interface to existing legacy components.

- Two main adapter variants:
  - **Class adapter:**
    - Uses interface implementation and inheritance mechanisms
  - **Object adapter:**
    - Uses delegation to adapt one interface to another
- **Object adapters are much more common.**

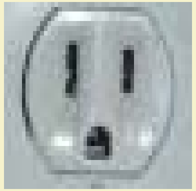
Essential Idea Behind Adapter Pattern





Client

Target



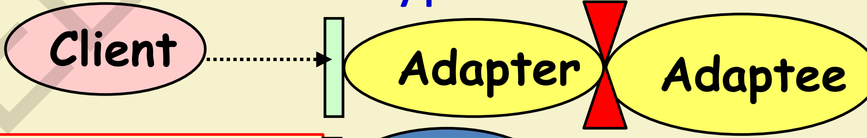
Adaptee

Adapter

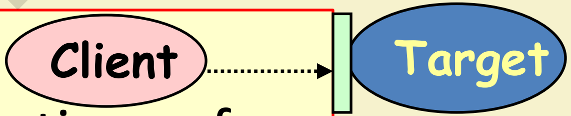
Adapter Pattern

- **Helps two incompatible types to communicate.**
  - When a client class expects an interface ---but that is not supported by a server class,
  - The adapter acts as a translator between the two types.

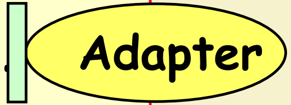
- 3 essential classes involved:



  - o **Target** – Interface that client uses.



  - o **Adapter** - class that wraps the operations of the Adaptee in interfaces familiar to client.

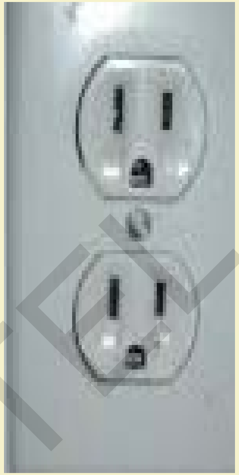


  - o **Adaptee** - class with operations that the client class desires to use.

Recap: Terminology

Target

Adapter

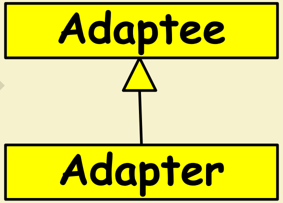


Adaptee

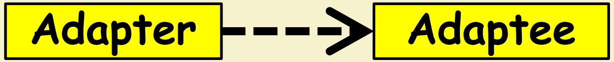
Class and Object Adapters

An adaptee may be given a new interface by an adapter in two ways:

- **Inheritance**
  - Known as **Class Adapter pattern**
  - The adapter is a sub-class of adaptee;

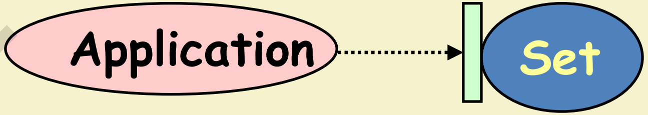


- **Delegation**
  - Known as **Object Adapter pattern**
  - The adapter holds a reference to an adaptee object and delegates work to it.



Example 1 – Sets

- There are many ways to implement a set
- Assume:
  - Your existing set implementation has poor performance.
- You got hold of a more efficient set class,

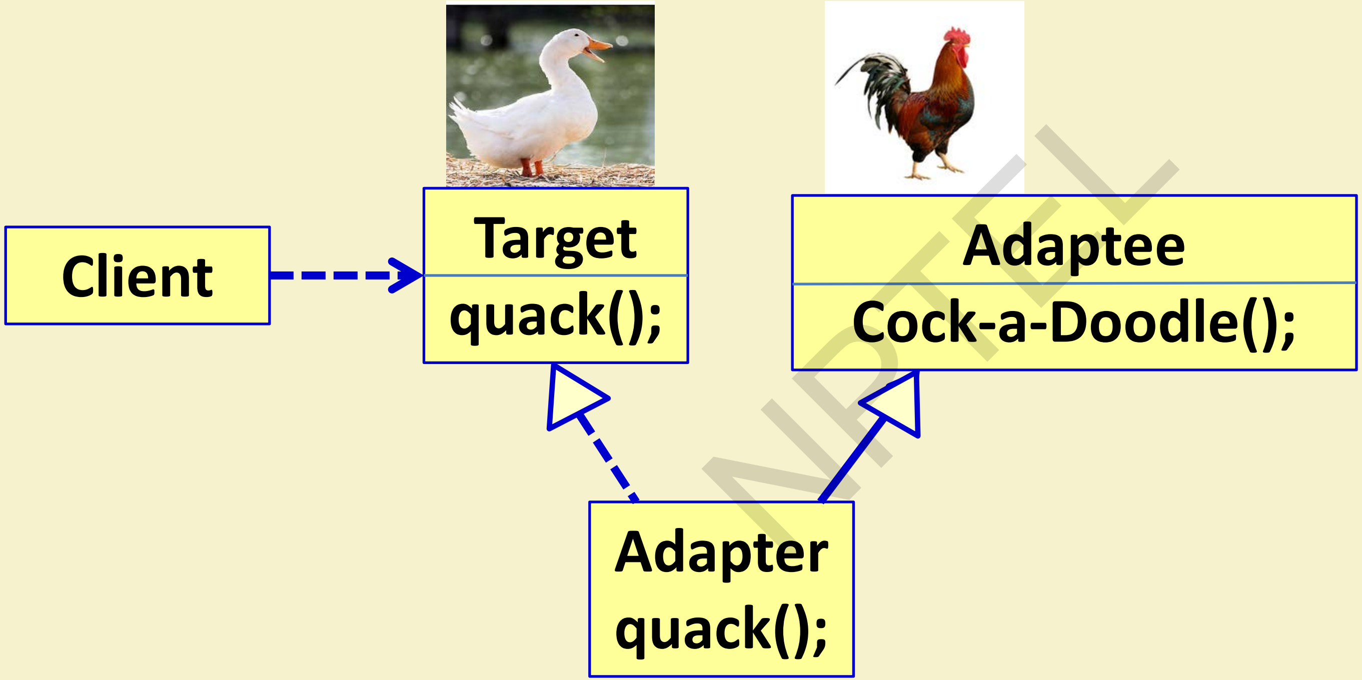


  - BUT: The new set has a different interface.
  - Do not want to change voluminous client code

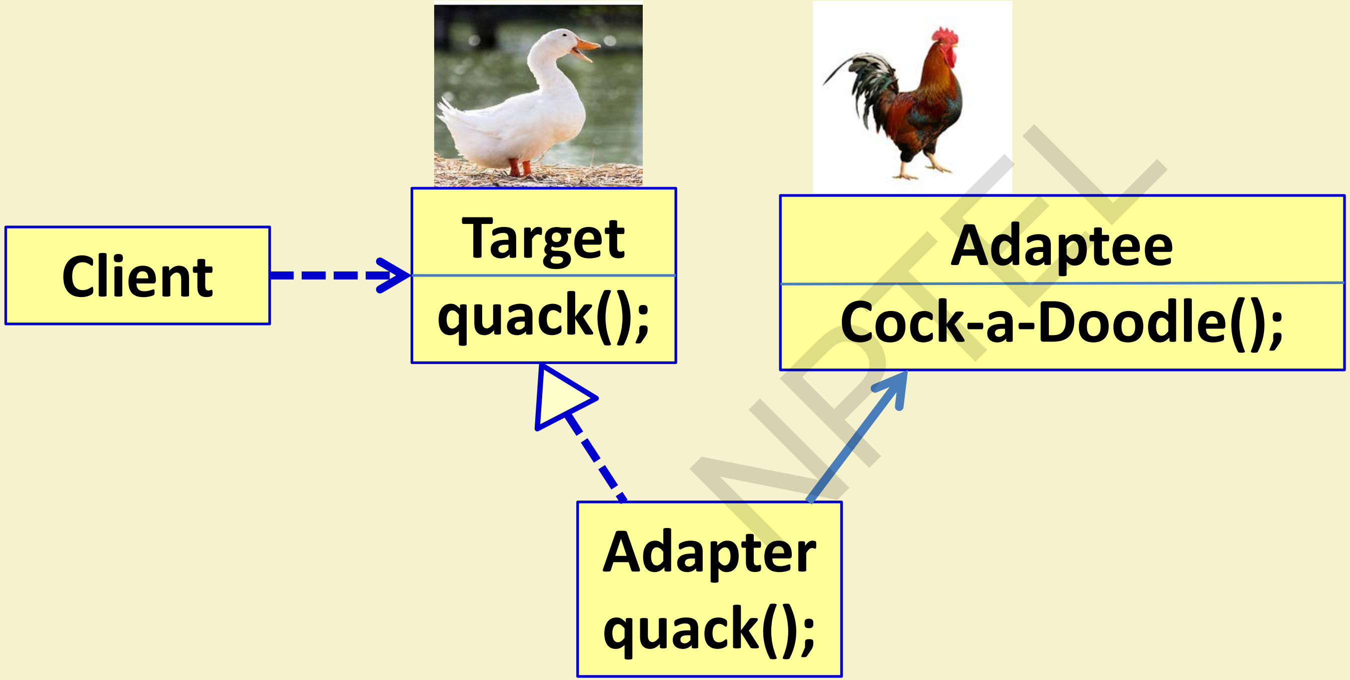
- **Solution: Design a setAdapter class :**
  - Same interface as the existing set..

  - Simply translates to the new set's interface.

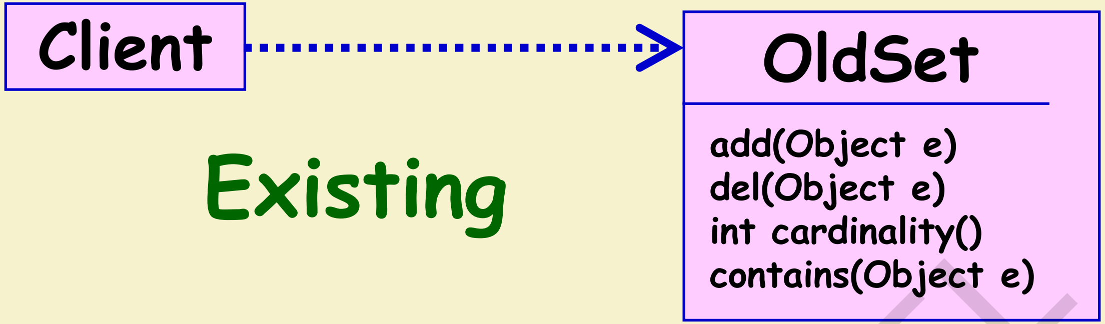
Class Adapter: Main Idea



Object Adapter: Main Idea



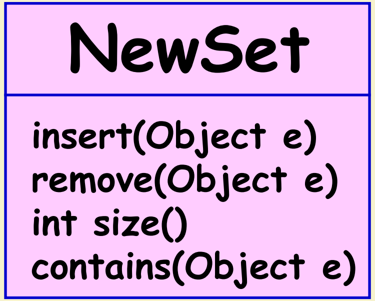
Example: Problem



Existing

Target

Got hold of Newset…



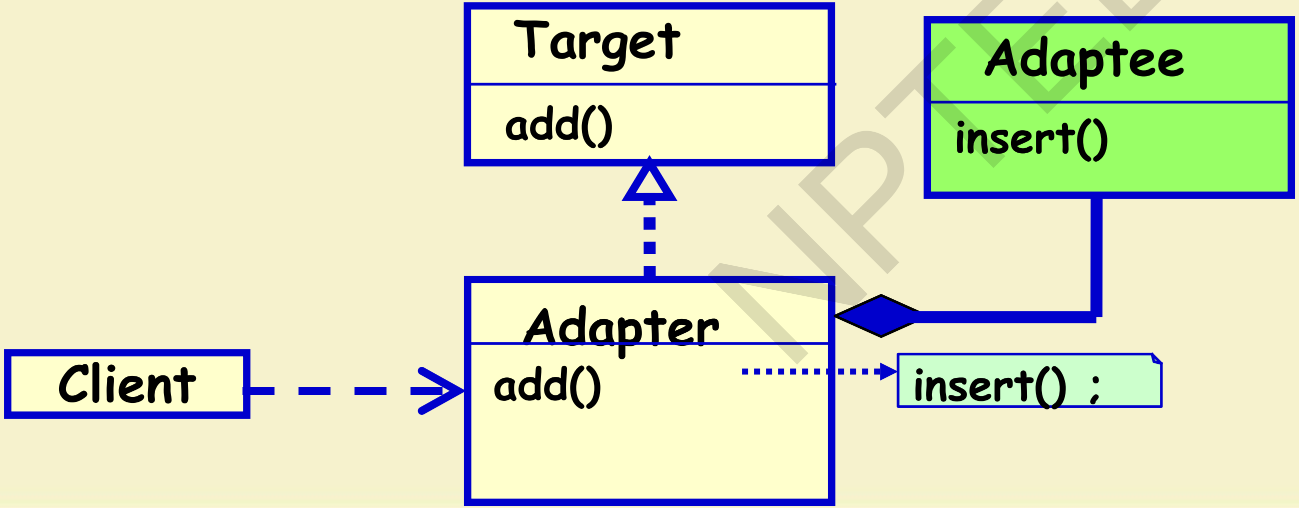
Want use this with client…

Adaptee

But, do not want to change Client code…

Object Adapter --- main idea delegation

- Adapter internally holds an instance of the Adaptee
- Uses it to call Adaptee operations from within operations supported by the Target.



Object Adapter - Code

Client Code:

```
   Adaptee a =new Adaptee(); Target t = new Adapter(a);
     public void test() {  t.add();  }
```

Target Code:

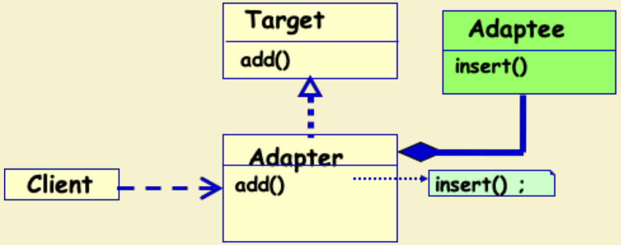
```
interface Target {
 public void  add(){}
}
```

Adaptee Code:

```
class Adaptee {
  public void insert(){}
}
```

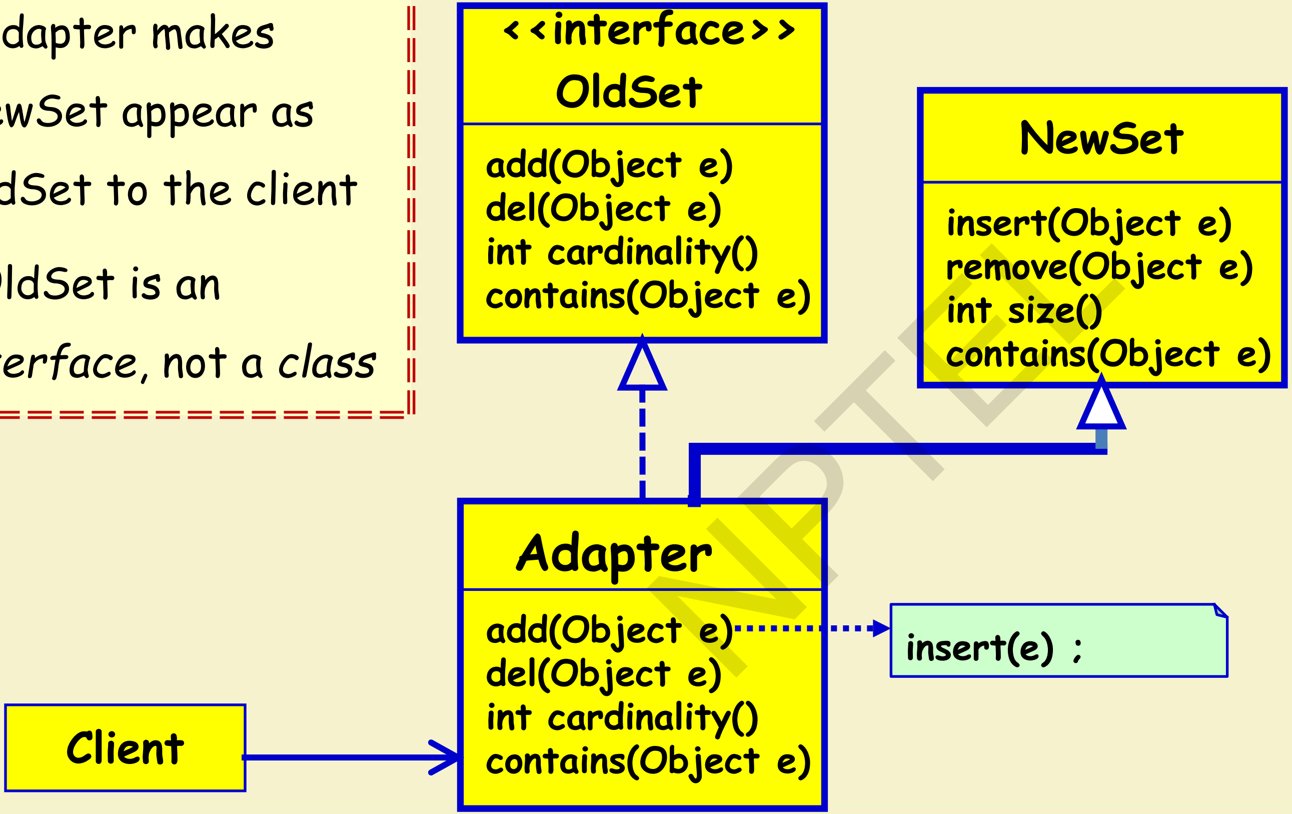
Adapter Code:

```
class Adapter implements Target {
  private Adaptee adaptee;
  public Adapter(Adaptee a) { adaptee = a;}
  public void add() { adaptee.insert();}
}
```



Class Adaptation

- Adapter makes NewSet appear as OldSet to the client
- OldSet is an *interface*, not a *class*



Class Adapter - Code

Client Code:

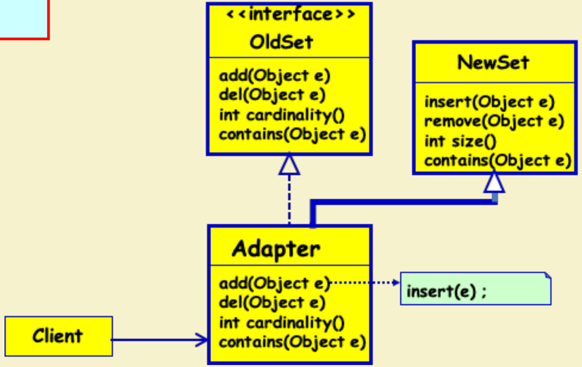
```
    Target t = new SetAdapter();
     public void test() {  t.add();  }
```

Target Code:

```
interface Target {
 public void  add(){}
}
```

Adaptee Code:

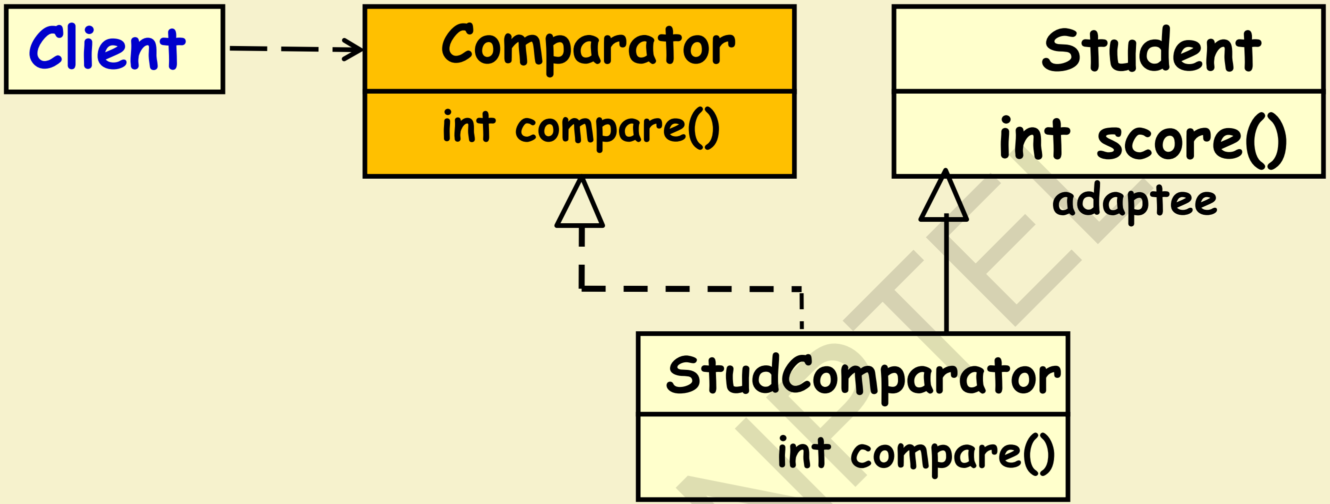
```
class SetAdaptee {
   public void insert(){}
}
```



Adapter Code:

```
class SetAdapter extends SetAdaptee implements Target{
    public void request() { specificRequest();}
}
```

Adapter design pattern for comparing Objects



Adapter pattern has been used in implementing the “Comparable” interface in Java

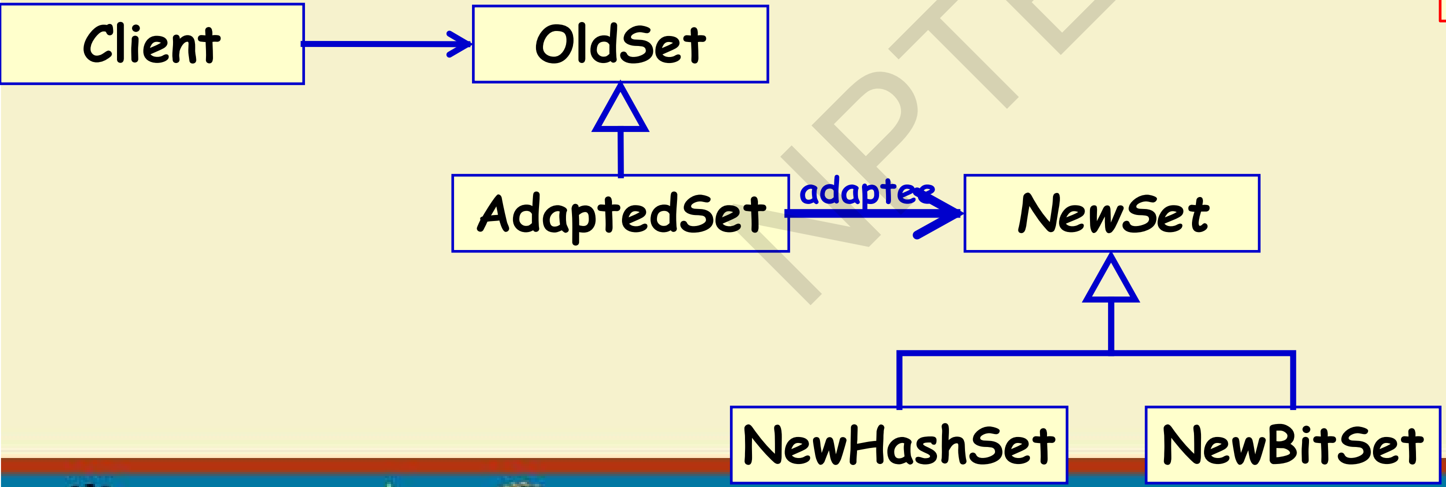
Outline of StudentComparator

```
public class StudentComparator
            implements Comparator<Student> {
   public int compare(Student s1, Student s2){
            return s1.score() – s2.score();}
     }
```

Solution: Object only ---Many subclasses to adapt:

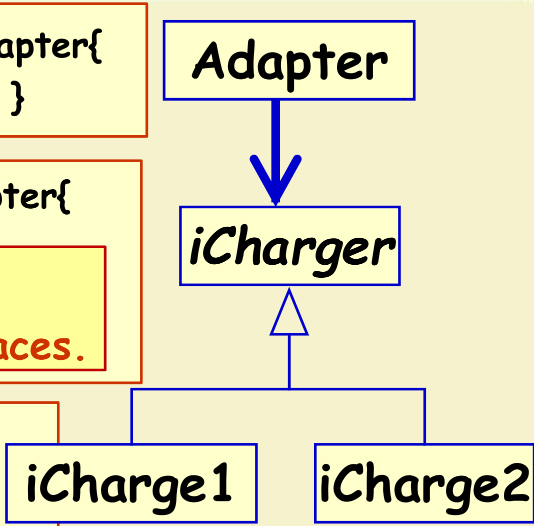
- Too expensive to adapt each subclass.
- Create single adapter to superclass interface.
- Configure the **AdaptedSet** with the specific **NewSet** at run-time.

Variant: Universal Adapter--Adapt Multiple Versions of NewSet



```
public class IPhoneCharger {
public void applePhoneCharge(){
 System.out.println("The iPhone is charging ..."); }  }
```

```
public interface ChargeAdapter{
public void phoneCharge(); }
```



```
public class UniversalCharger extends IPhoneCharger implements ChargeAdapter{
public void phoneCharge() {       ✗
super.applePhoneCharge(); } }
```

Class adapter:

Adapter object has at most two faces.

```
public class UniversalCharger implements ChargeAdapter{
IPhoneCharger iphoneCharger;
public UniversalCharger(IPhoneCharger iphoneCharger){
this.iphoneCharger = iphoneCharger;  }
public void phoneCharge() {
iphoneCharger.applePhoneCharge(); }
```

Object adapter

Class Adapters: Consequences

- A concrete adapter created for a specific Adaptee (e.g., **NewSet**)

● − Cannot adapt a class and all its subclasses

● + Can override Adaptee (e.g., **NewSet**) behavior:

  – After all, Adapter is a subclass of Adaptee

Object Adapters: Consequences

- Single Adapter can handle many Adaptees **(Universal adaptor):**
  - Can adapt the Adaptee class and all its subclasses.
- **Hard to override Adaptee behavior**
  - Because the Adapter *uses* but does not *inherit from* Adaptee interface.

Other Issues

- How much adapting does adapter do?
  - Simple forwarding of requests (renaming)?
  - Different set of operations and semantics?
  - **At some point do the Adaptee and Adapter interfaces and functionality diverge so much that "adaption" is no longer the correct term…**

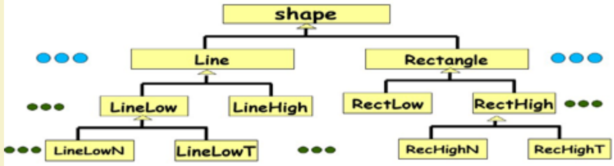
Advantages of Adapter Pattern

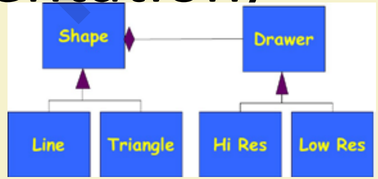
- Can help change behavior of existing software:
  - Without changing its source code.
- Can help maintain legacy software:
  - Without making any modifications to aging source code…

Bridge Pattern

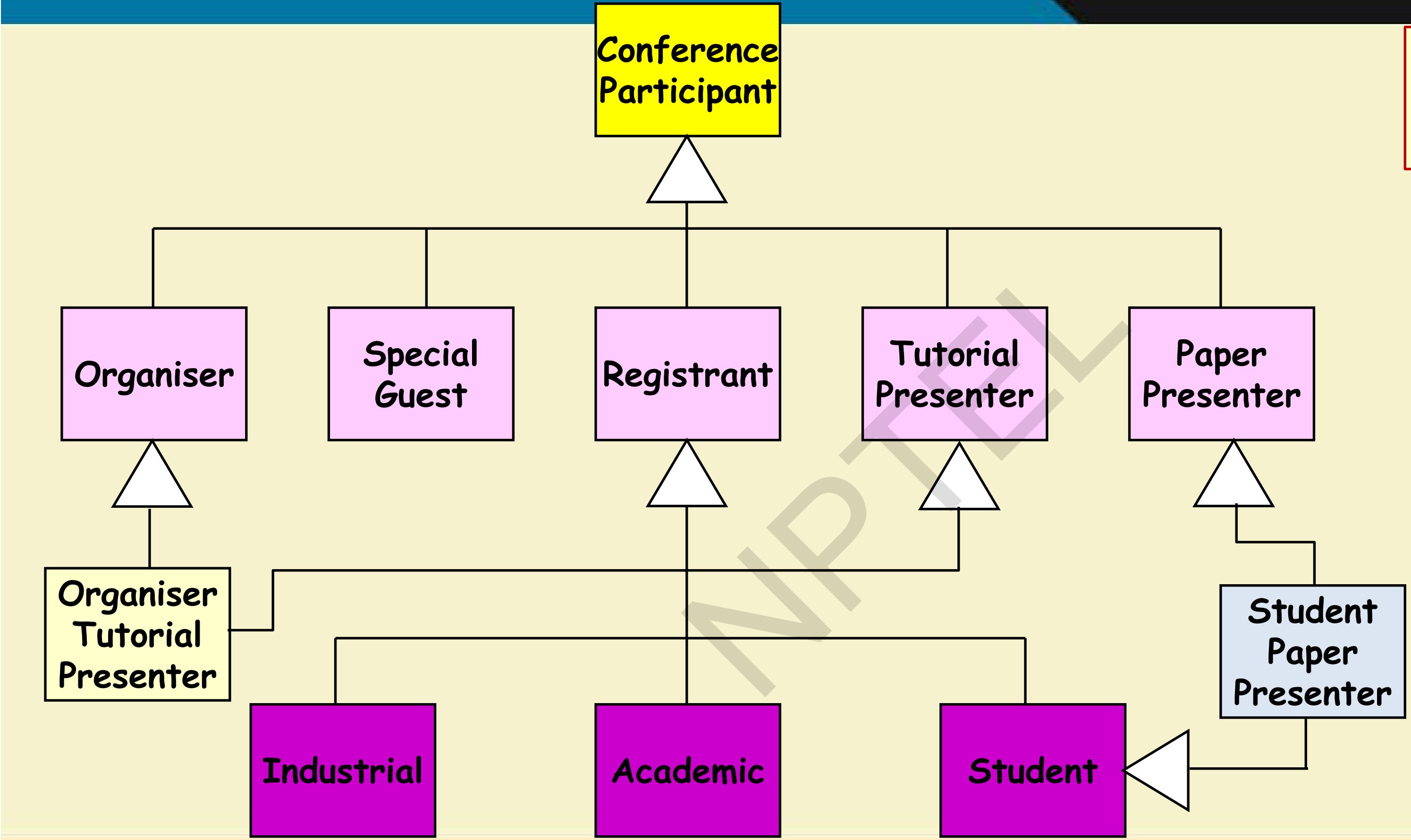
Bridge Pattern

- Also called **Handle/Body pattern**.
  - Split a complex class hierarchy into two hierarchies.
  - One represents the abstraction (called the handle).
  - The other is the implementation, and is called the body.
  - The handle forwards any invocations to the body.



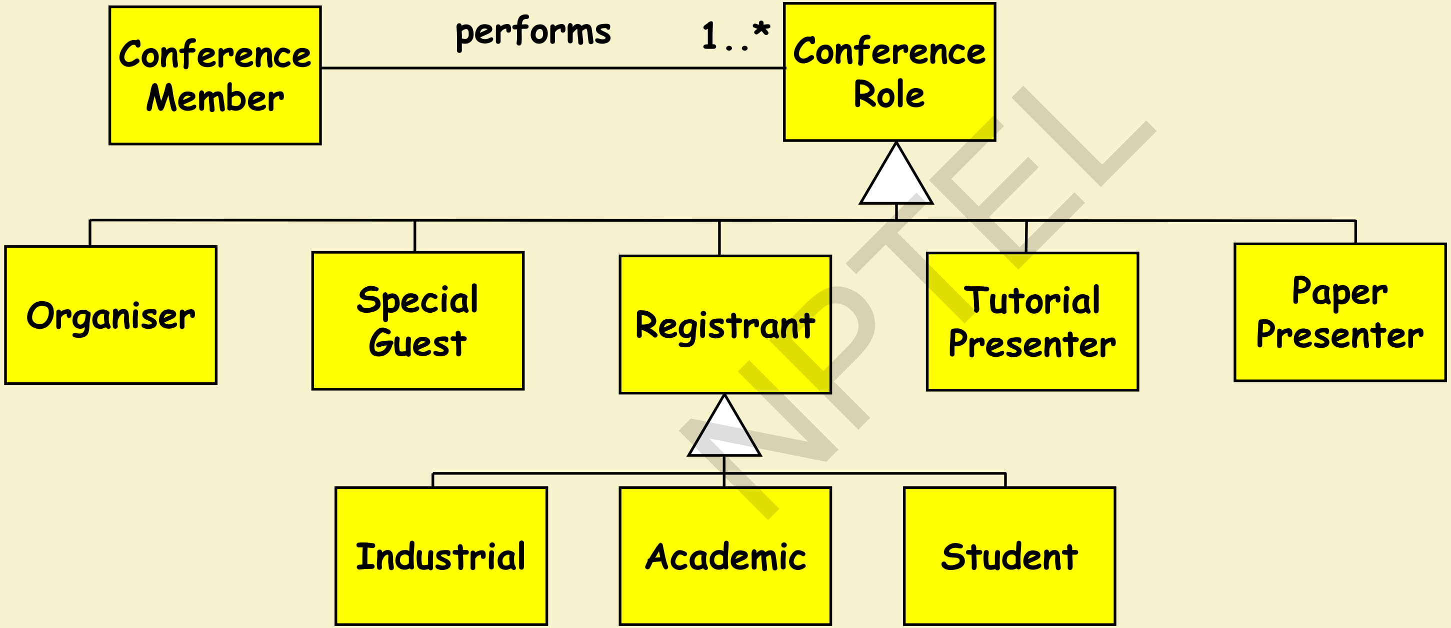


Motivation: Example 1



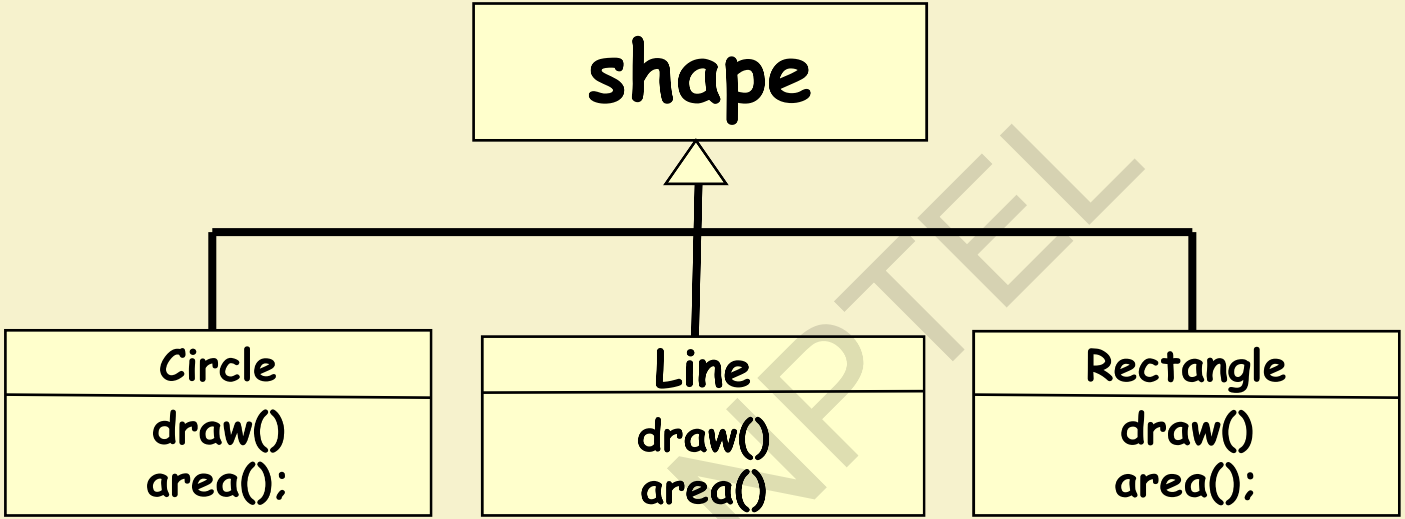
Solution: Roles

- In this case, multiple inheritance is not a particularly good solution
- ***Roles* are a much better solution as they are more flexible**



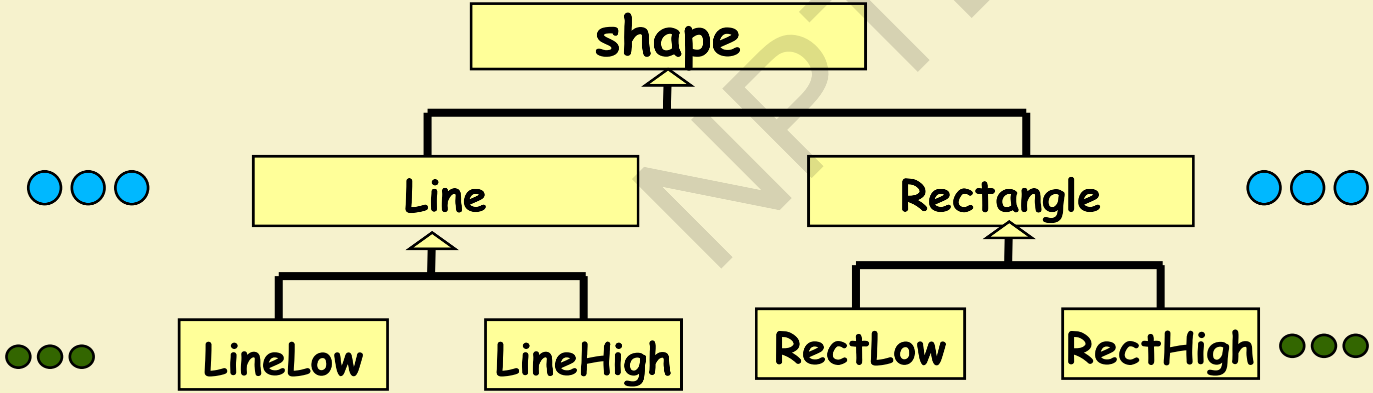
- You designed a graphics package…

Example 2



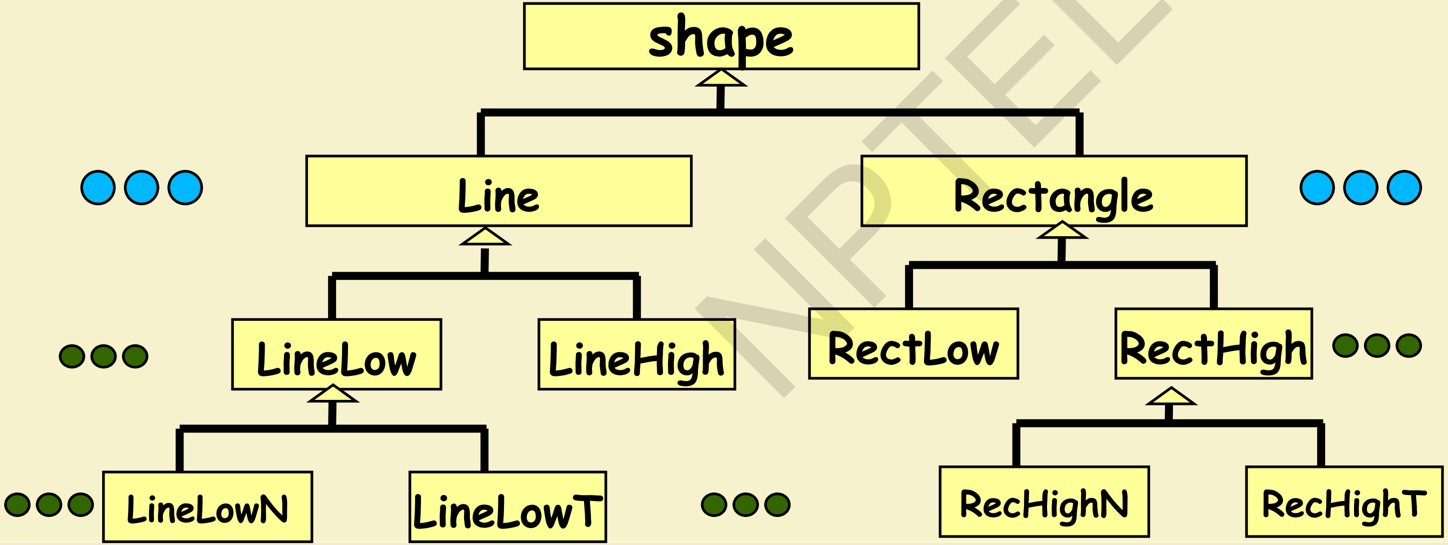
Example 2

- Things worked fine until you had to support mobile phones that can draw only low precision shapes.
  - **You extended your design….**



- You soon had to support a different drawing primitive for efficient display of transient views for animation…
  - You extended your design again ….

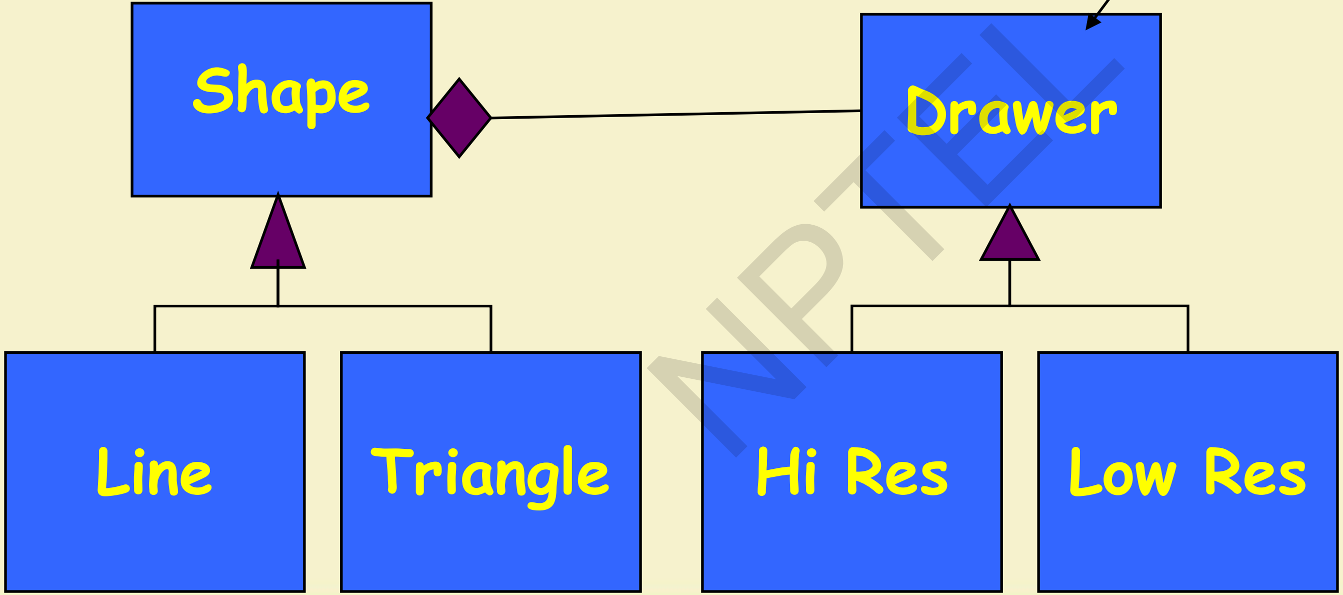
Example 2



- You soon needed a different way of drawing on Smartphones…
- Things were becoming pretty complicated …

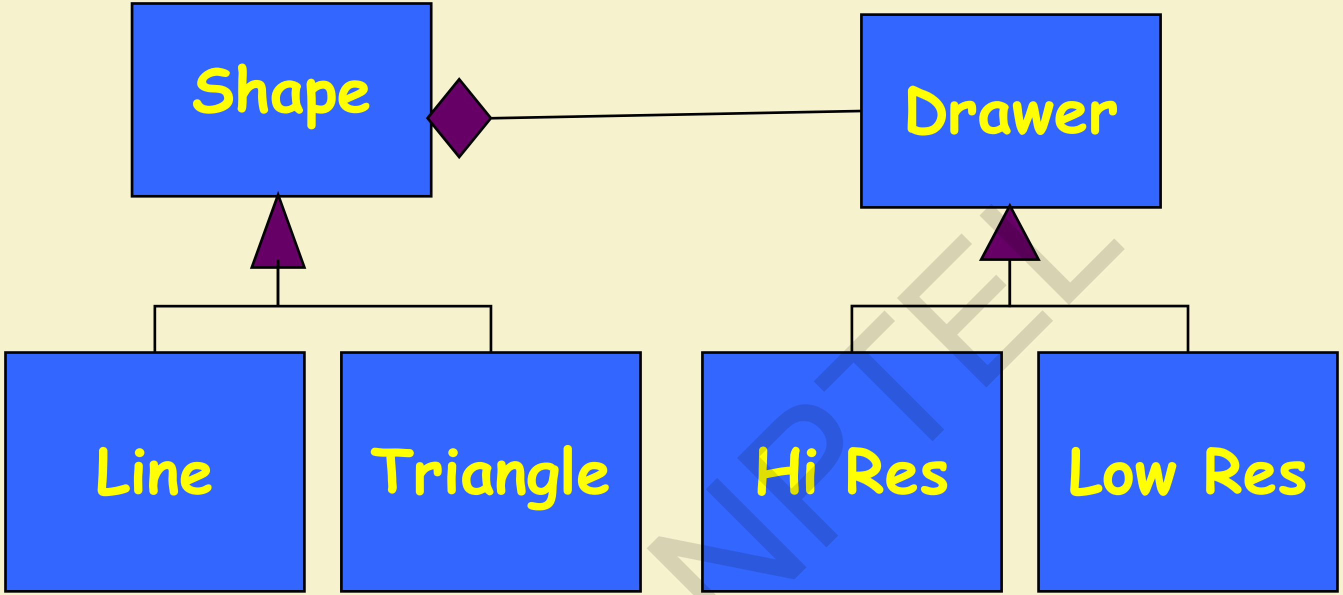
until you decided to use bridge design pattern…

Bridge Design Pattern

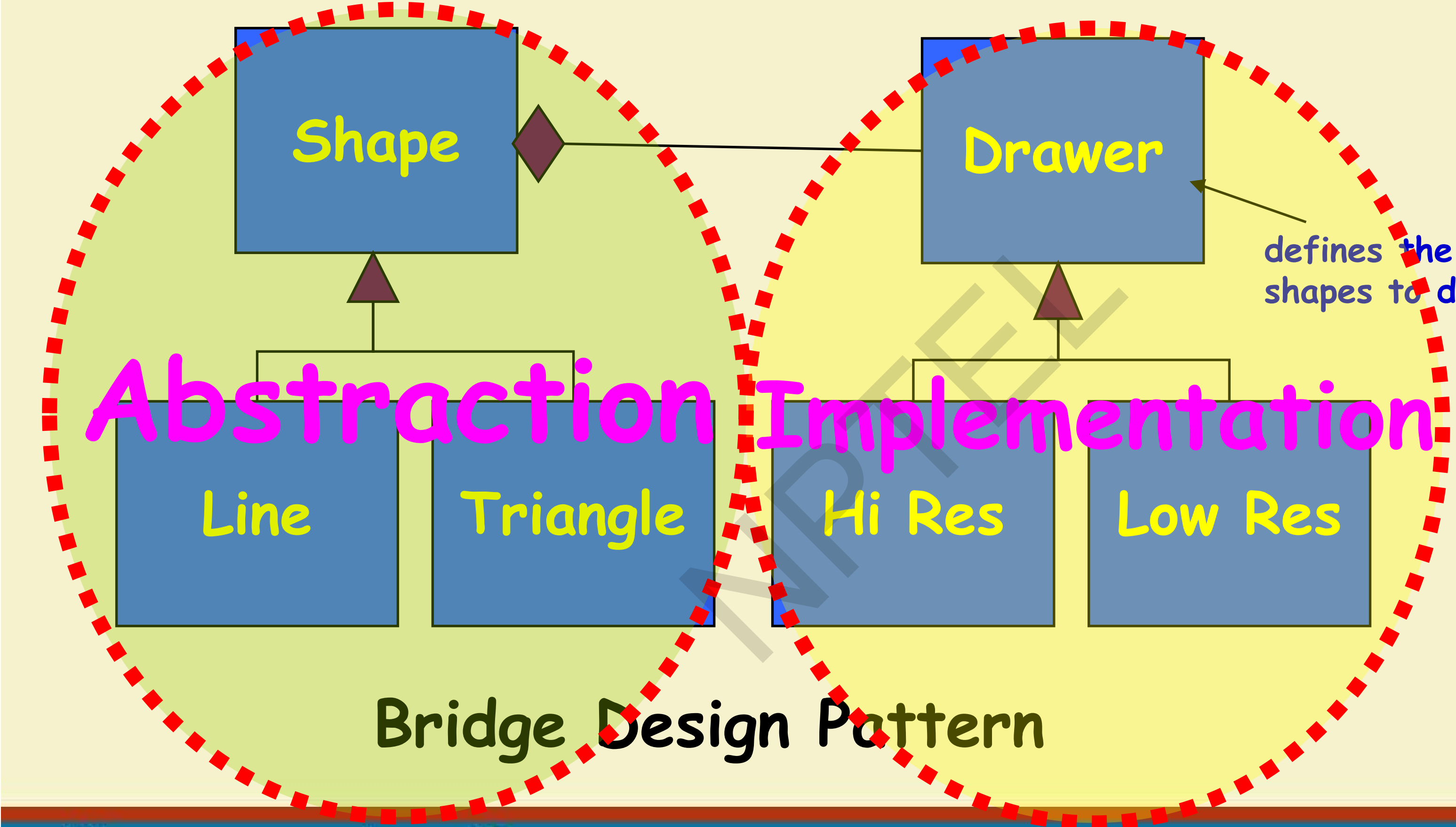


defines the interface shapes to draw

Bridge Design Pattern

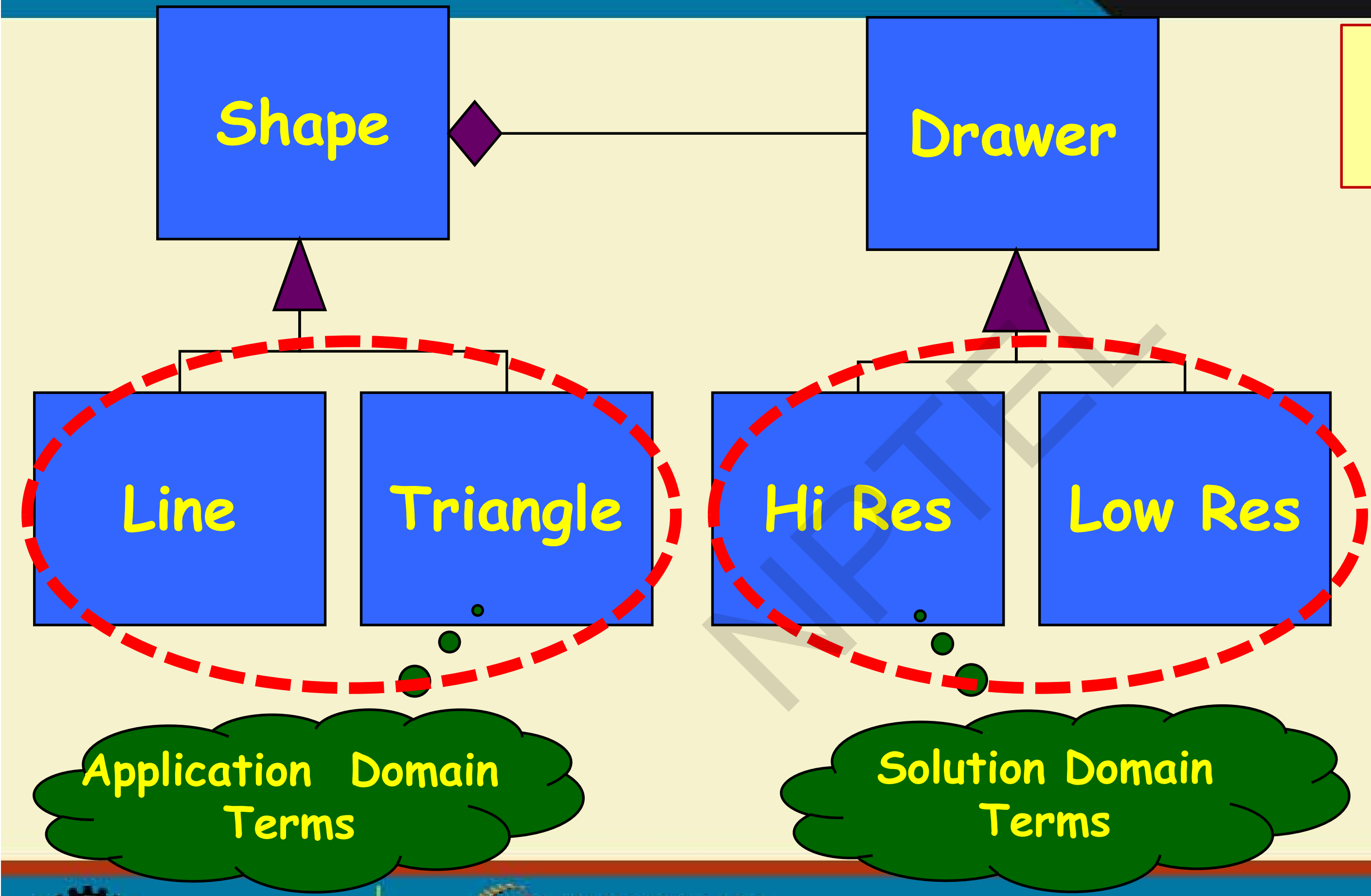


Separate the abstraction from implementation



defines the interface shapes to draw

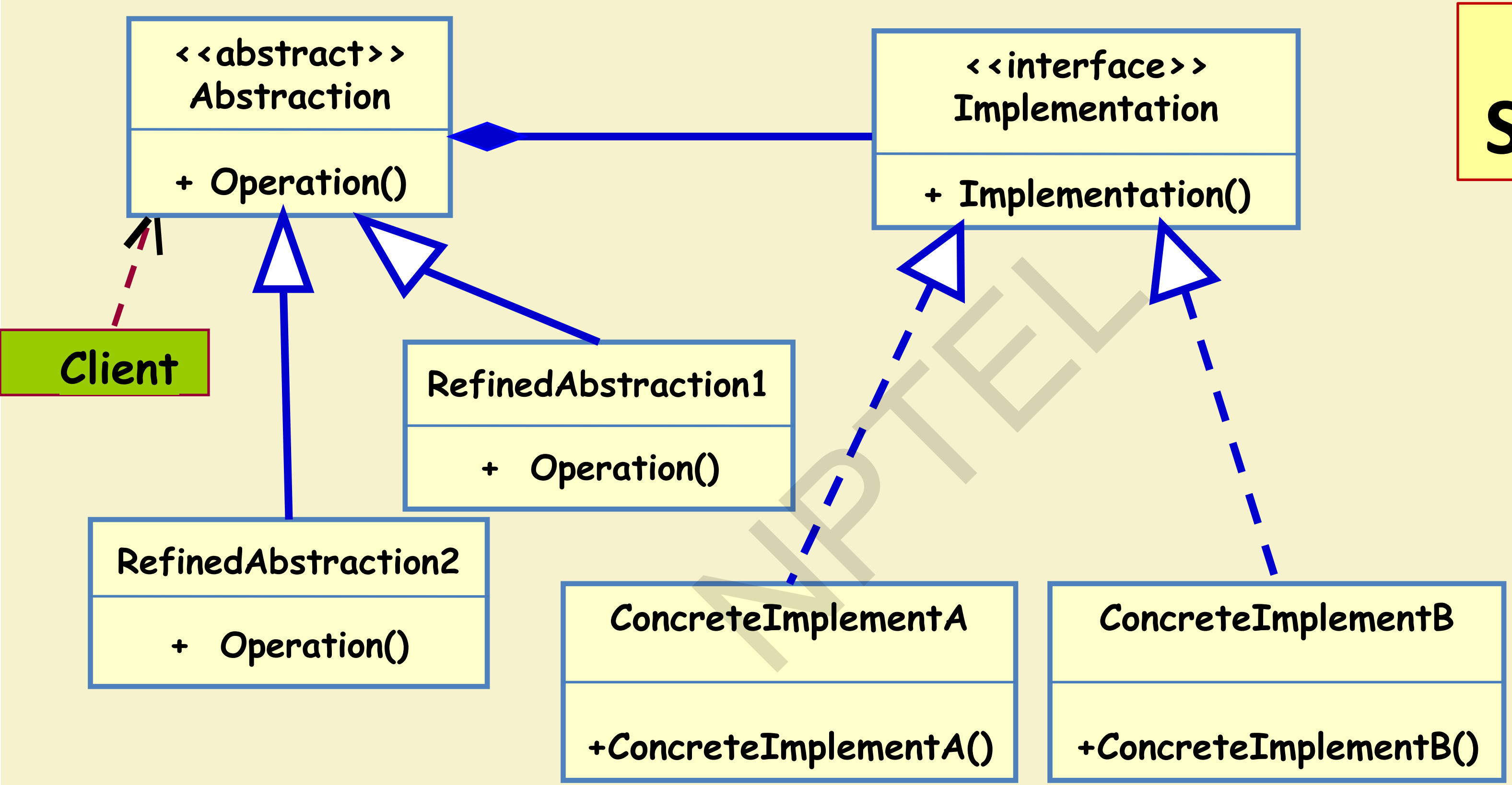
Bridge Design Pattern



Why the Name Bridge?

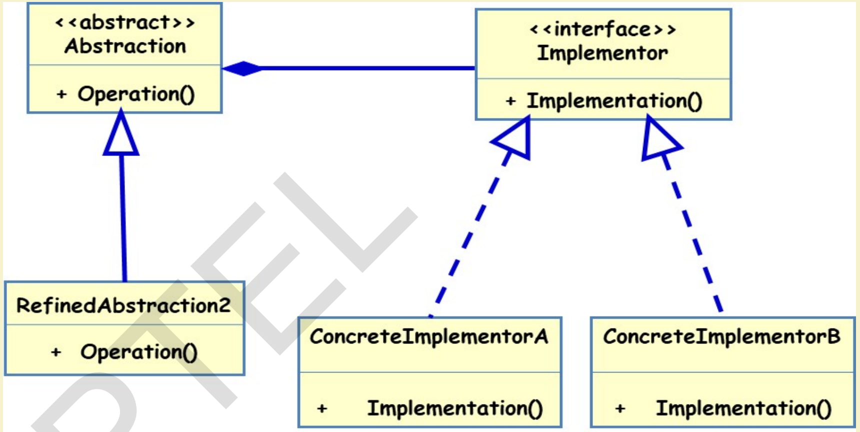
Provides A Bridge Between the Application and Solution domains…

Bridge: Structure

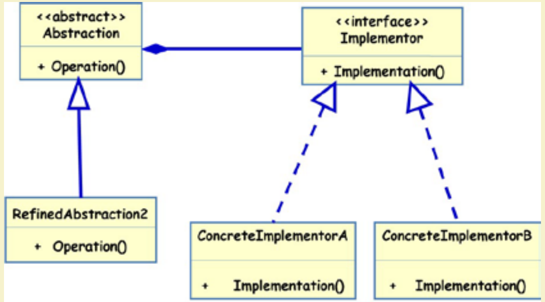


Participants

- Abstraction
- RefinedAbstraction
- Implementor
- ConcreteImplementers



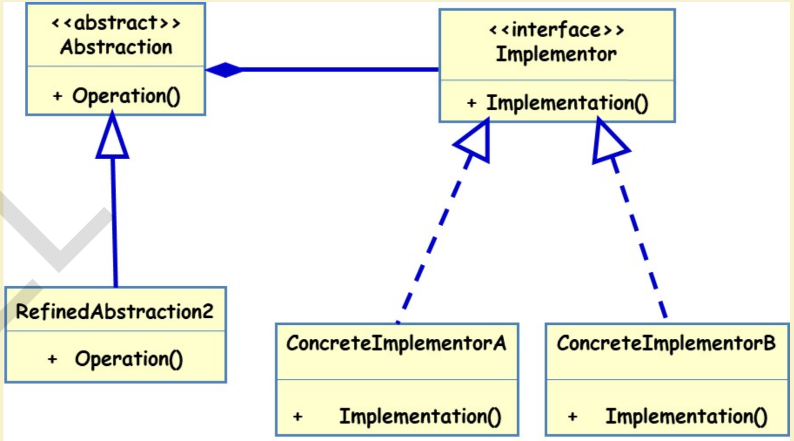
Participants contd…



- **Abstraction**
  - Defines the abstract interface
  - Maintains a reference to the implementer
- **RefinedAbstraction**
  - Extends the interface defined by Abstraction

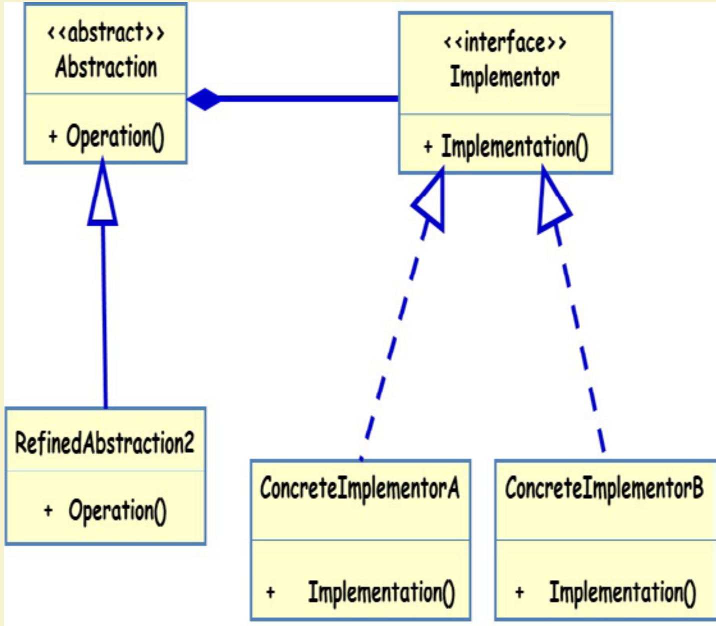
Participants contd…

- **Implementer**
  - Defines the interface for the implementation classes



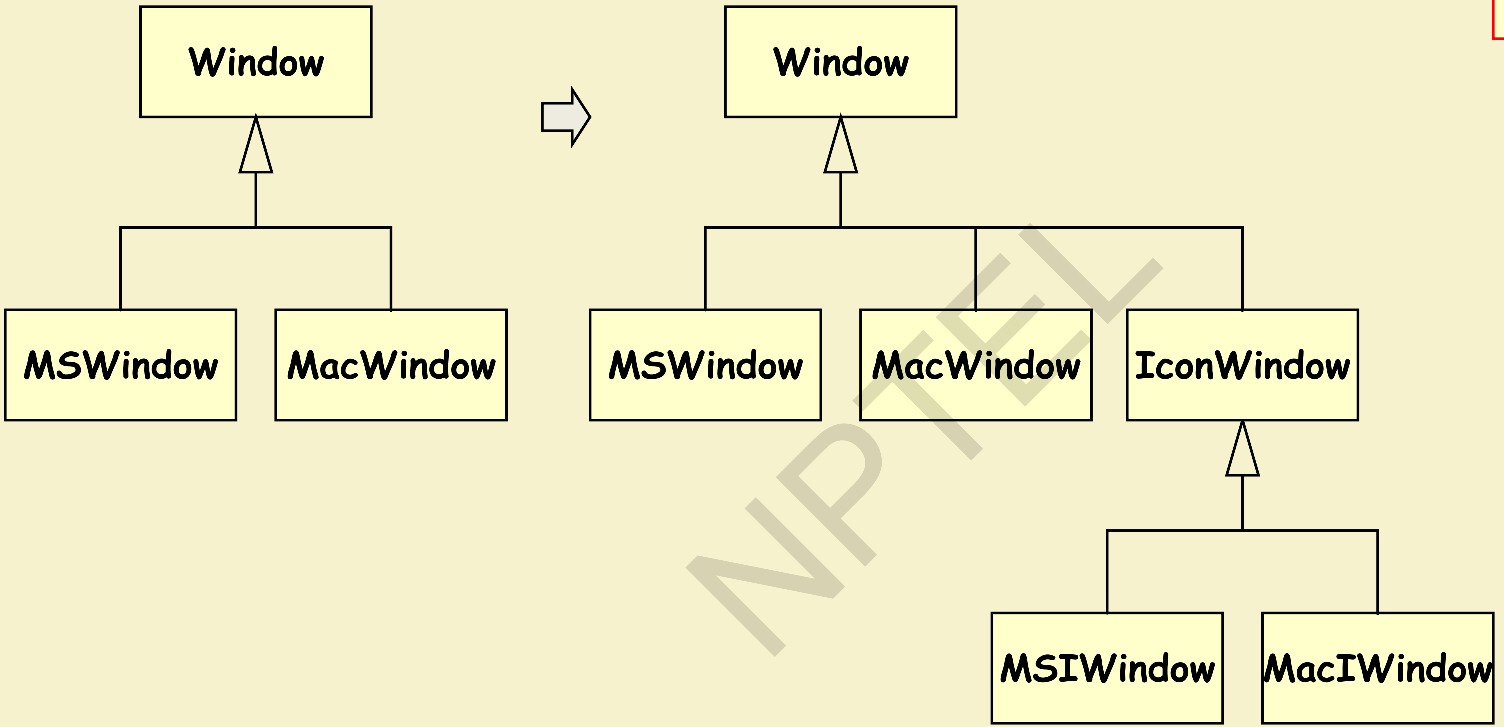
- **ConcreteImplementer**
  - Implements the implementer interface

Collaborators

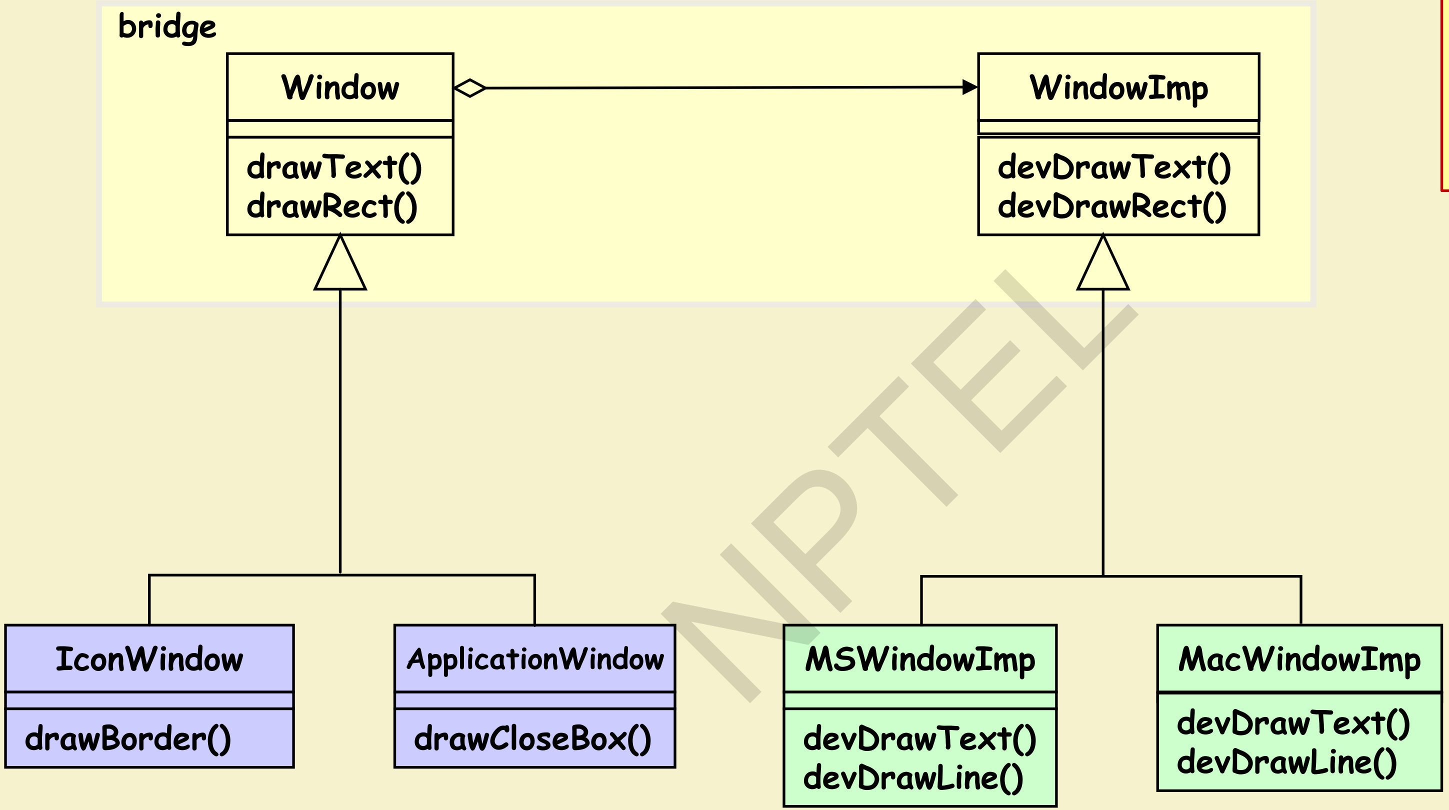


- **Abstraction forwards client requests to Implementer object.**
  - Clients interface with abstraction class.
  - Abstraction class forward any requests to the implementer class.

Example 1

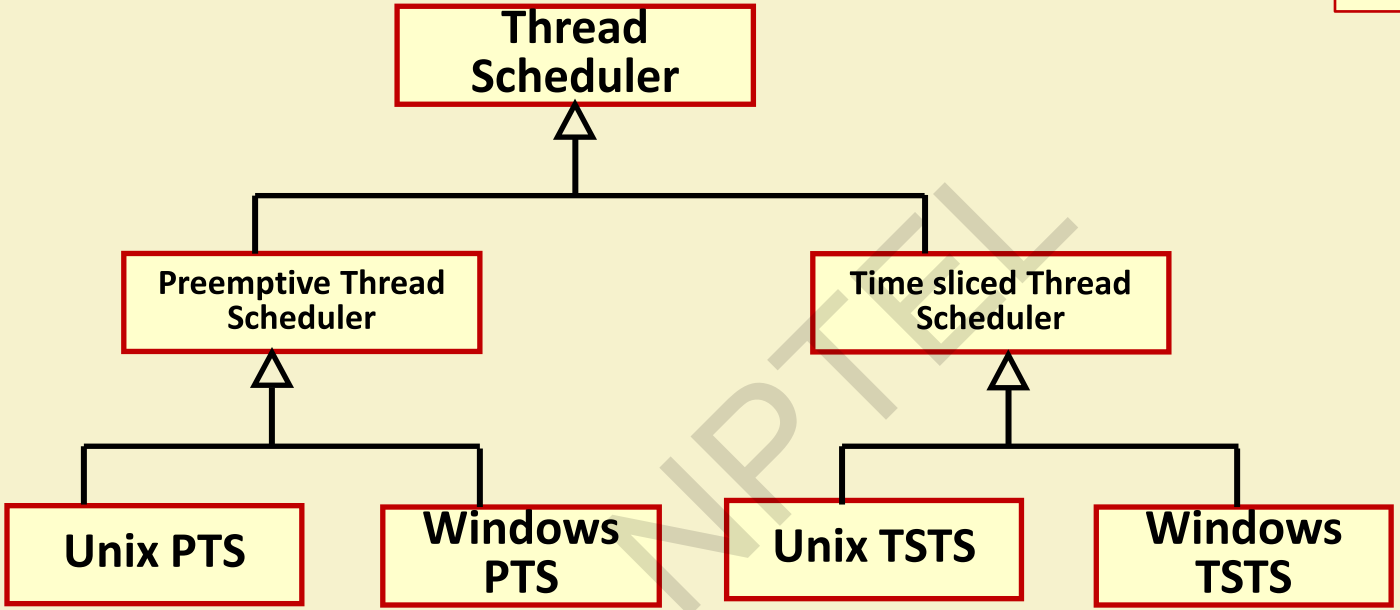

Window
MSWindow
MacWindow
Window
MSWindow
MacWindow
IconWindow
MSIWindow
MacIWindow

Example 1: Solution



Consider thread scheduling :

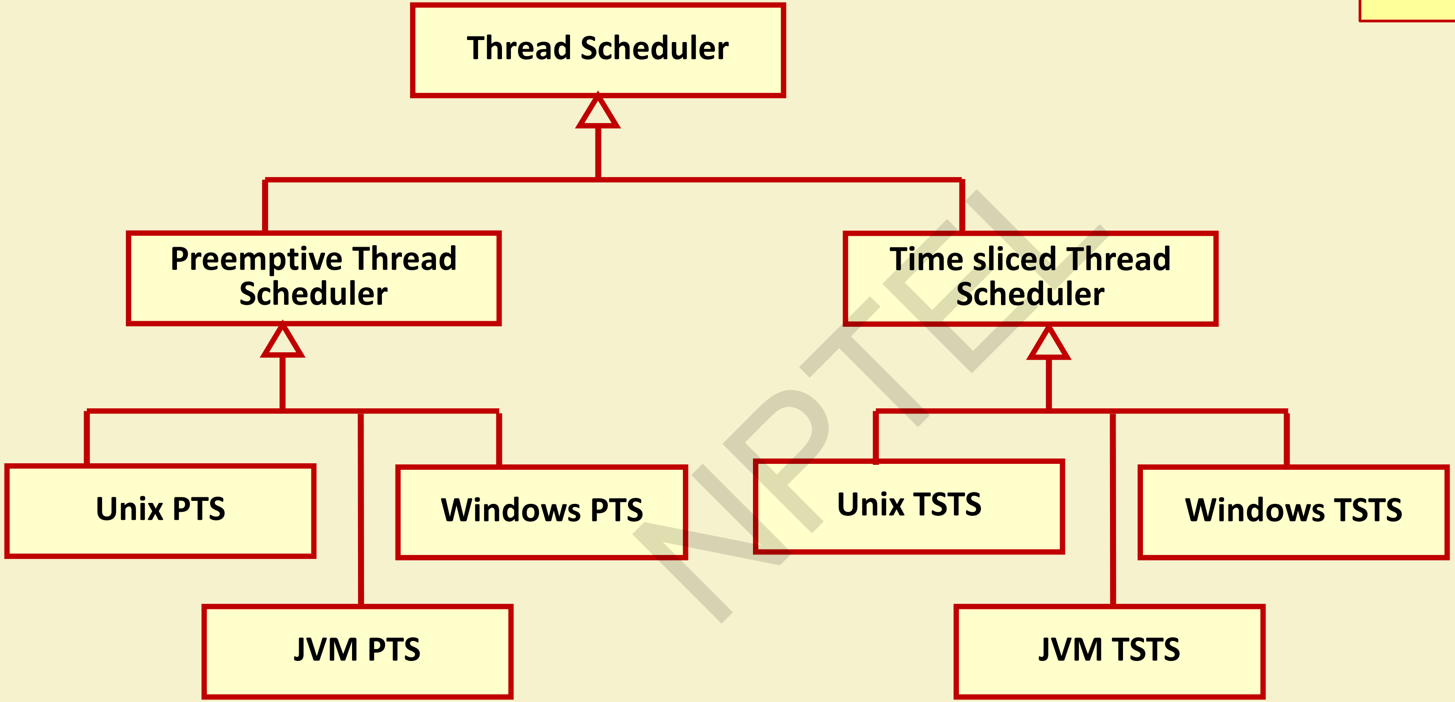
Example 2



A class for each permutation of dimensions!

We need to add support for Java platform also…

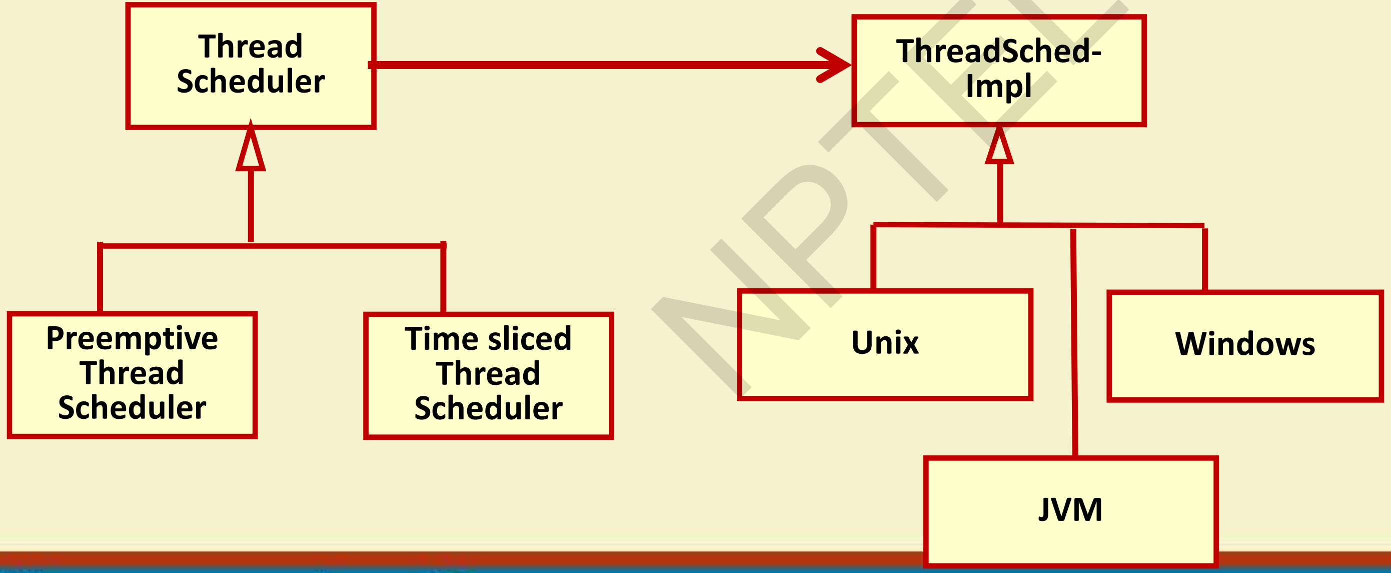
Example 1



- Explosive Class Hierarchy!

- Refactoring into two orthogonal hierarchies:
  - One for platform-dependent abstractions a nd other for platform independent implementations

Example 4: Solution



- Suppose an abstraction has several implementations:
  - **Inheritance is commonly used to accommodate these!!!**

Observation

1. **Inheritance binds an implementation to the abstraction permanently:**
   - It becomes difficult to modify and reuse abstraction and implementations independently.
2. Inheritance without a **Bridge**:
   - Leads to violation of single responsibility principle (SRP)

Overuse of inheritance.

“As a beginning object-oriented analyst, I had a tendency to solve the kind of problem I have seen here by using special cases, taking advantage of inheritance. I loved the idea of inheritance because it seemed new and powerful. I used it whenever I could. This seems to be normal for many beginning analysts, but it is naive: given this new “hammer,” everything seems like a nail. “

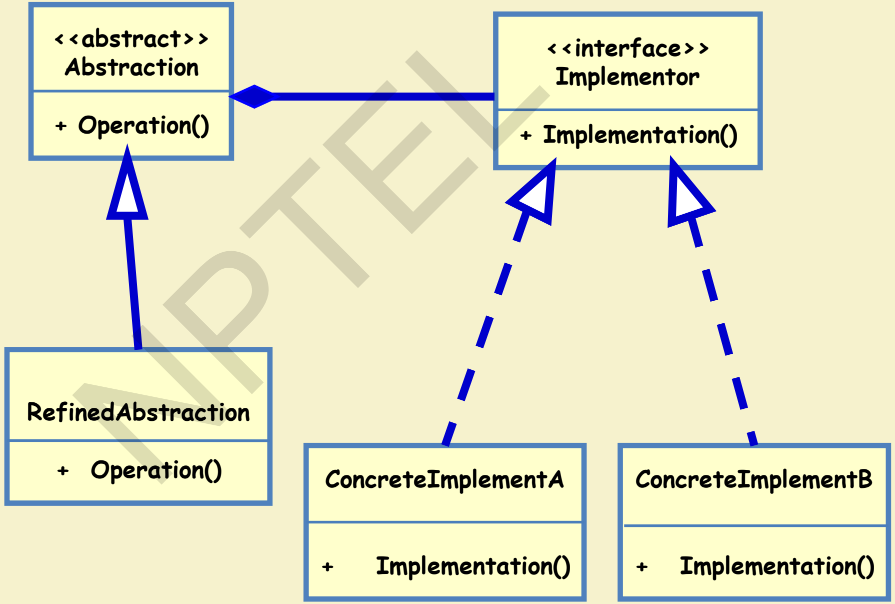
Bridge: Applicability

Use bridge Pattern when:

- You want to avoid a permanent binding between an abstraction and its implementation.
  - **Implementation may be selected or switched at run time.**
- Both the abstraction and their implementation should be extensible by subclassing without impacting the clients:
  - **Even code should not be recompiled.**

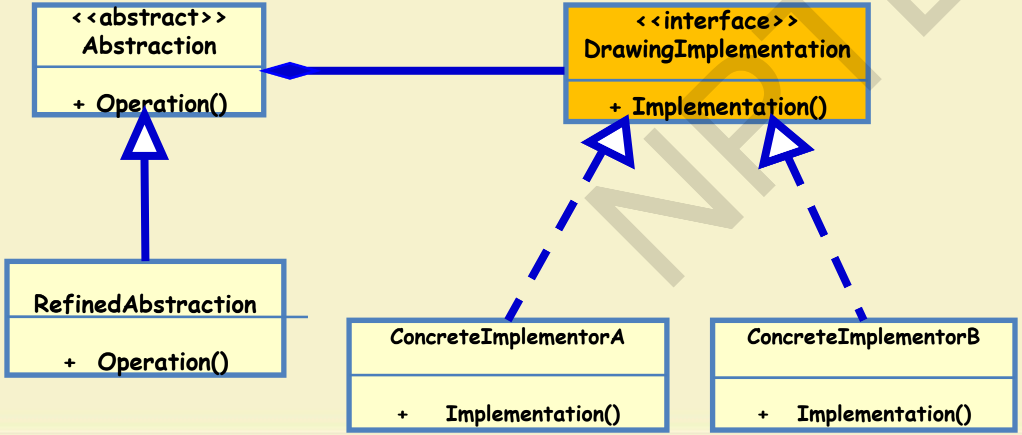
Bridge Pattern: Example 3

- **Circle Shape problem**
  - Different implementations for drawing circle
  - A method for changing the circle abstractly
- **Participants**
  - **Abstraction**
    - Interface Shape
  - **RefinedAbstraction**
    - Class CircleShape
  - **Implementation**
    - Interface DrawingAPI
  - **ConcreteImplementations**
    - Class DrawingAPI1
    - Class DrawingAPI2



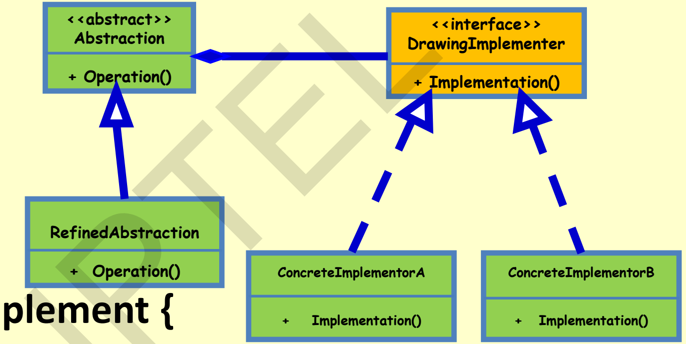
Bridge Pattern Example 3

```
Interface DrawingImplementation{
   public void drawCircle(double x, double y,
  double radius);
 }
```



Bridge Pattern Example 3

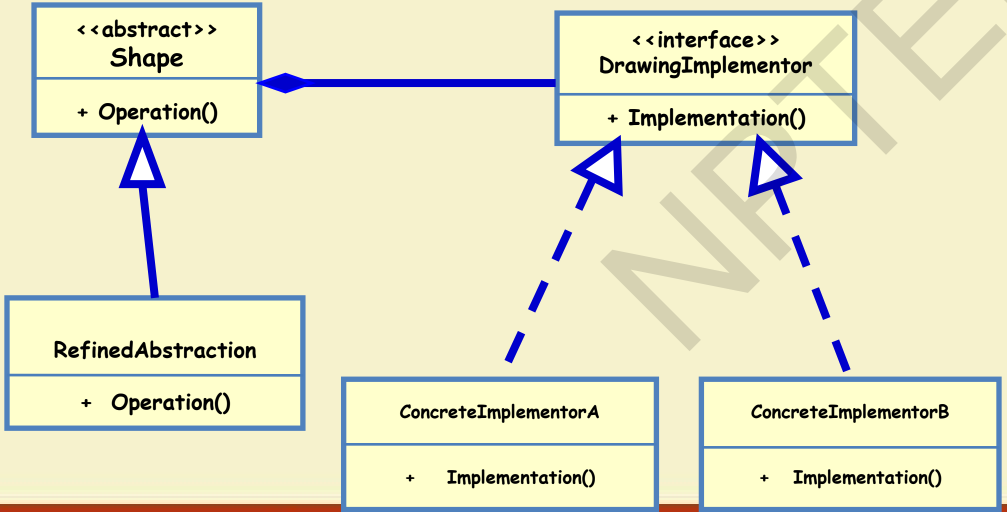
```
class ConcreteImplementerA implements DrawingImplementer{
    public void drawCircle(double x, double y, double radius) {
            System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
    }
}
```



```
class DrawingAPI2 implements DrawingImplement {
   public void drawCircle(double x, double y, double radius){
            System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);
    }
}
```

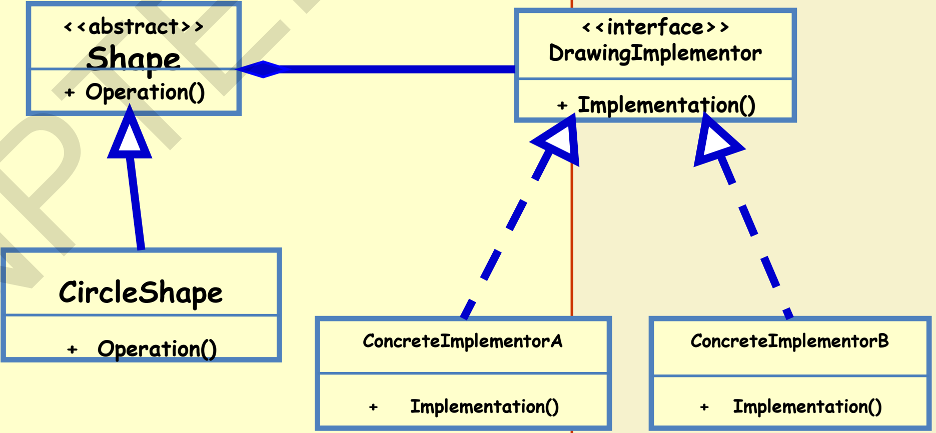
Bridge Pattern: Example 3

```
interface Shape {
   public void draw();
   public void resizeByPercentage(double pct);
}
```



Bridge Pattern Example 3

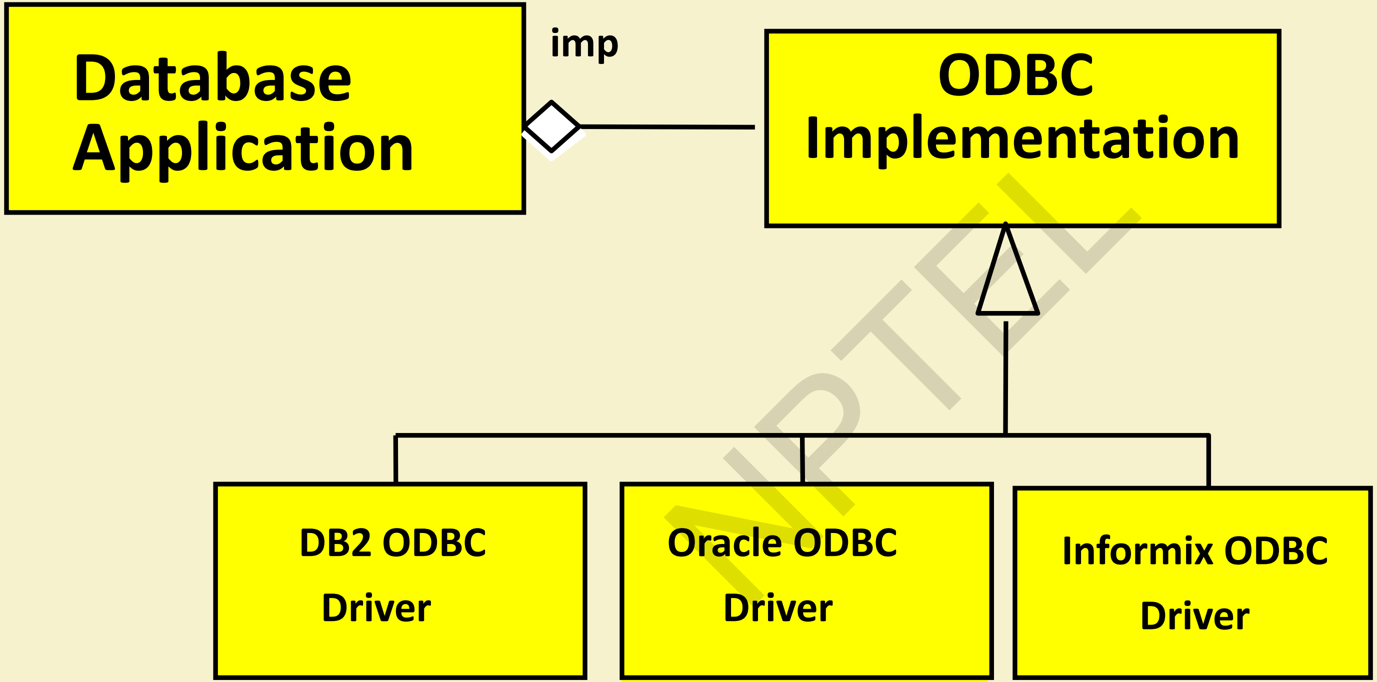
```
class CircleShape implements Shape {  /** "Refined Abstraction" */
    private double x, y, radius;
    private DrawingAPI drawingAPI;
    public CircleShape(double x, double y, double radius, DrawingAPI
    drawingAPI)
    { this.x = x; this.y = y; this.radius = radius; this.drawingAPI = drawingAPI;
    }
    // low-level i.e. Implementation specific
    public void draw() {
    drawingAPI.drawCircle(x, y, radius);
    }
    // high-level i.e. Abstraction specific
    public void resizeByPercentage(double pct)
     radius *= pct;
     }
}
```



Bridge Pattern Example 3

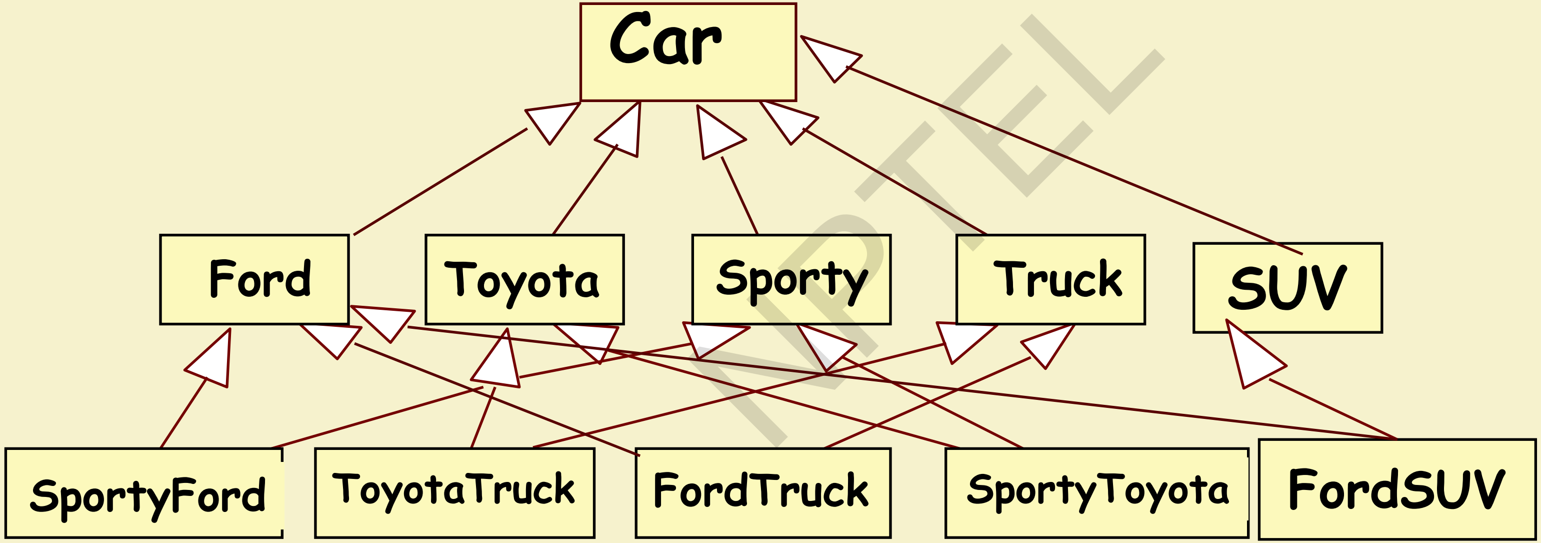
```
/** "Client" */
class BridgePattern {
   public static void main(String[] args) {
          Shape[] shapes = new Shape[] {
          new CircleShape(1, 2, 3, new DrawingAPI1()), new CircleShape(5, 7,
   11, new DrawingAPI2()) }
          for (Shape shape : shapes)    {
                    shape.resizeByPercentage(2.5);
                    shape.draw();
          }
   }
}
```

Example 4


Database Application
imp
ODBC Implementation
DB2 ODBC Driver
Oracle ODBC Driver
Informix ODBC Driver

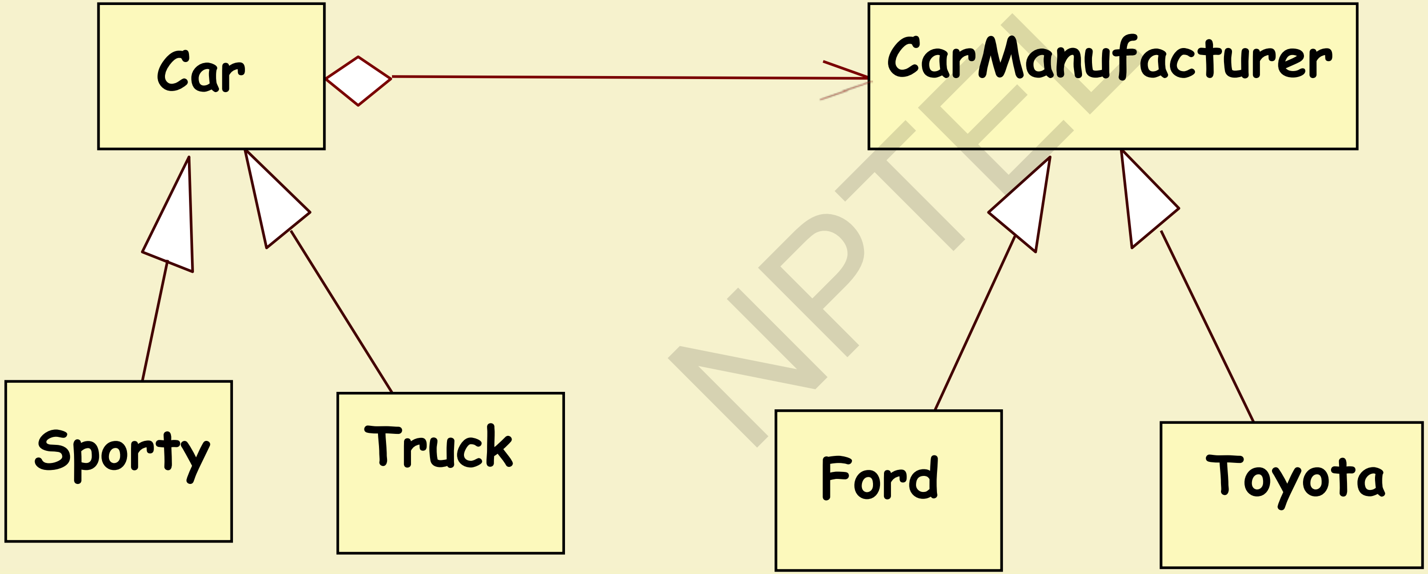
Bridge Pattern: Exercise 1

- How do we simplify the following design?



Exercise 1: Solution…

Use Bridge when you might otherwise be tempted to use multiple inheritance…



When should we apply Bridge Pattern?

- **We want run-time binding of implementation.**
- **We need to overcome a proliferation of classes:**
  - –Resulted from a coupled interface and numerous implementations
  - –We need to map these into orthogonal class hierarchies

Implementation Issues

- How and when to decide which implementer to instantiate?
- **Depends**:
  - if Abstraction knows about a concrete implementer, then it can instantiate it.
  - Or it can delegate the decision to another object (to an abstract factory for example)
- **“Find what varies and encapsulate it” and “favor object composition over class inheritance”**

Benefits

- Decoupling abstraction from implementation
- Reduction in number of sub-classes
- **Reduced complexity and reduction in executable size**
- Interface and Implementation can be varied independently
- Improved extensibility:
  - Abstraction and Implementation can be extended independently

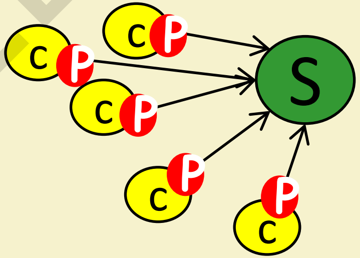
Drawbacks?

- Increased Complexity???
- Double Indirection :
  - Abstraction→ Implementation
    →ConcreteImplementation

Proxy Pattern

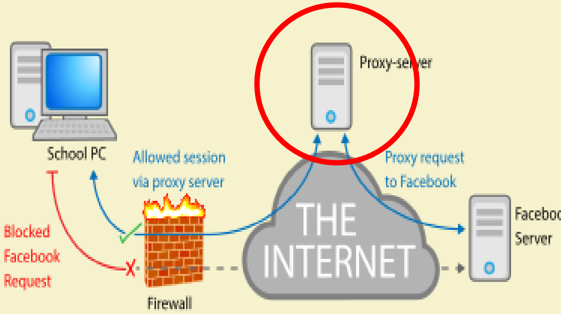
Proxy (Surrogate) Pattern

- **Problem:** **How should a client invoke the services of a server when access to the server should be managed in some way?**
- **Solution:** A proxy object should be created at the client side.
- **Proxy Role:** The proxy can help to
  - Authenticate
  - Hide details of network operations.
    - Determine server address,
    - Communicate with the server, obtain server response and seamlessly pass that to the client, etc.



Digression: Network Proxy Server

- **What is the role of a network proxy server?**
  - To keep machines behind it anonymous, mainly for security.
  - To speed up access to resources using caching.
  - To prevent downloading the same content multiple times and save bandwidth (caching).
  - To log usage to support reporting of company employee-wise Internet usage.
  - Firewall: Scan for malware



- **Encryption / SSL acceleration:** Secure Sockets Layer (SSL) encryption is usually not done by the web browser itself, but by the proxy that is equipped with SSL acceleration hardware.
- **Load balancing:** can distribute the load to several web servers, each web server serving its own application area.
- **Compression:** proxy server can optimize and compress the content to speed up the load time.
- **Spoon feeding:** Address the problem caused by slow clients by caching the response from the web servers and slowly "spoon feeding" it to the client.

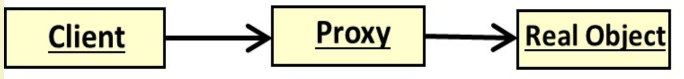
Proxy: Some Insights

- **Use Proxy pattern whenever the services provided by a supplier need to be managed in some way without disturbing the supplier interface.**

- **Example:** You require some additional conditions to be satisfied before the actual object is accessed:
  - **Consider loading an image from disk only when it is actually needed.**

Proxy Pattern

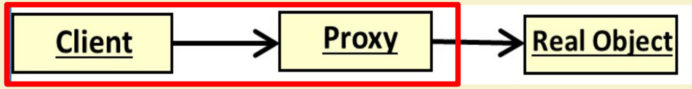
- Proxy object has the same interface as the target object:
  - **Proxy stores a reference to the target object**
  - **Forwards (delegates) requests to it.**



- Sometimes more sophistication needed than just a simple reference to an object:
  - **That is, we want to wrap code around references to an object…**

Proxy Pattern

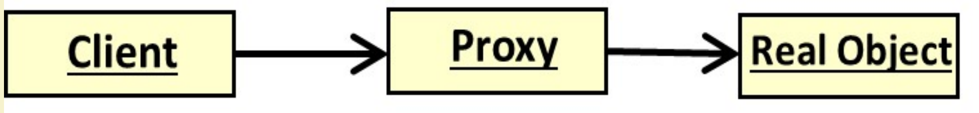
- **A Structural Pattern:**
  - Provides surrogate for some object,
  - Controls access to real target object.



- Real target may not always be instantiated immediately:
  - Due to performance, location, or access restrictions.

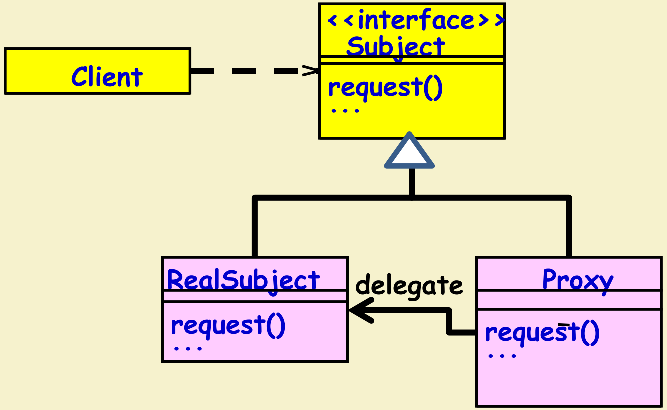
Proxy Usage 1: Help Reduce Expensive Steps

- Example of what is expensive…
  - Heavy weight object Creation
  - Object Initialization

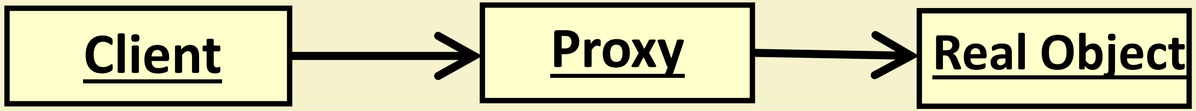


- **Defer object creation and initialization to the time the object is actually need.**
- **Proxy pattern:**
  - Reduces the cost of accessing objects
  - The proxy object acts as a stand-in for the real object
  - The proxy creates the real object only if the user asks for it

Proxy Solution

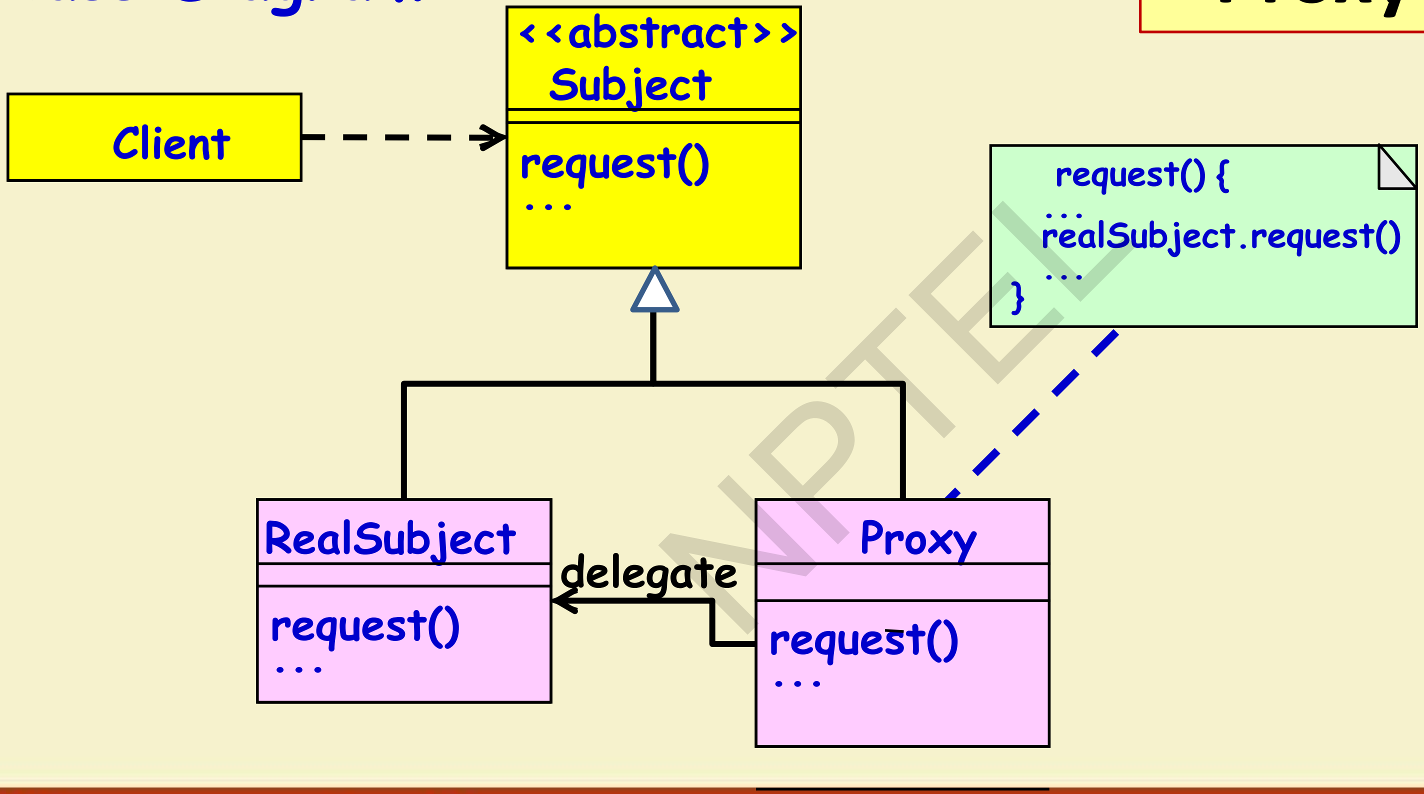


- Create a Proxy object that implements the same interface as the real object…
- The Proxy object contains a reference to the real object …
- Clients have a reference to the Proxy:
  - Not the real object
- **Client invokes operations only on the Proxy:**
  - **Proxy may perform additional processing before delegating…**



Proxy Structure

Class Diagram

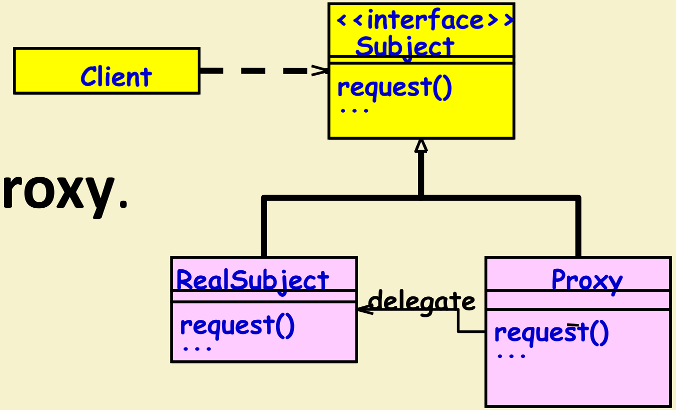


```
request() {
 ...
realSubject.request()
 ...
}
```

Proxy Pattern

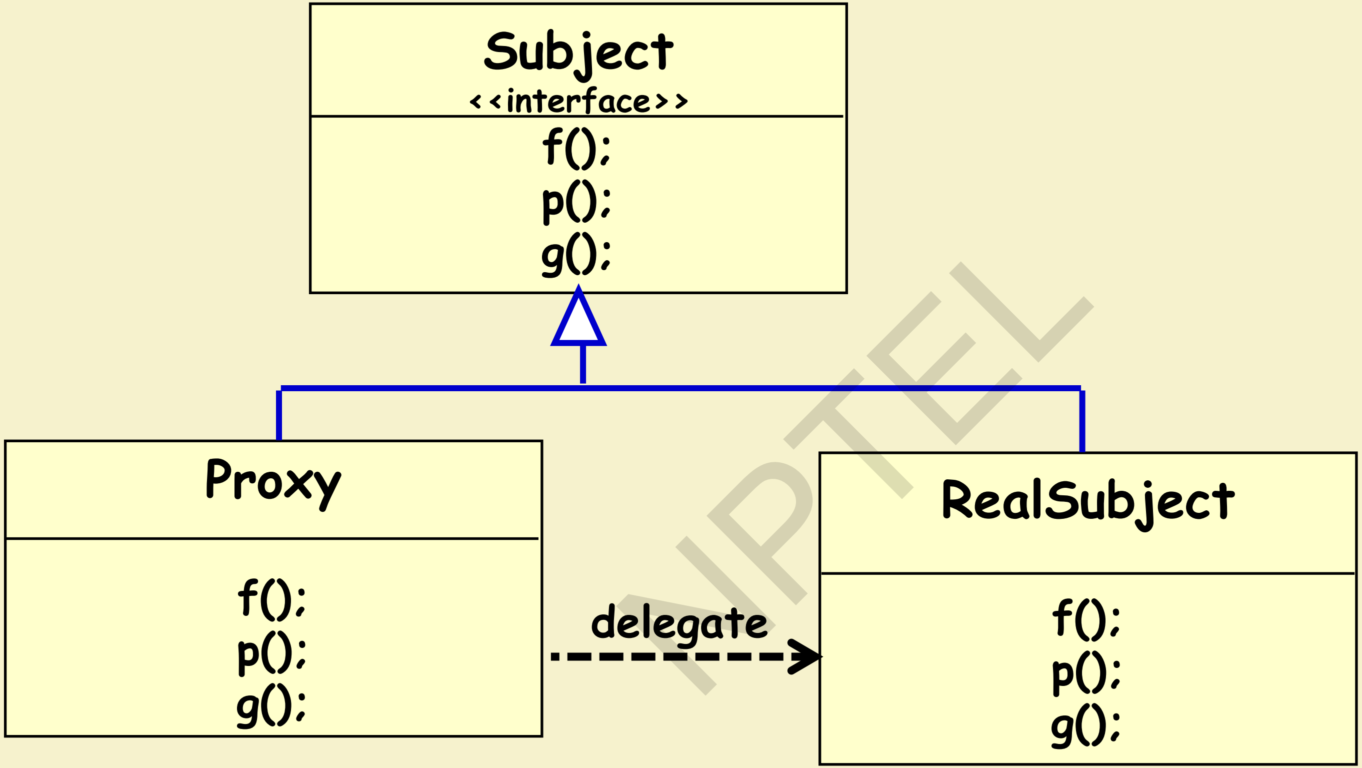
- Three Classes: **Subject**, **RealSubject**, and **Proxy**.

- Interface **Subject:**
  - –Implemented both by **RealSubject**, and **Proxy**.



- **Proxy:**
  - – Delegates any calls to **RealSubject**.

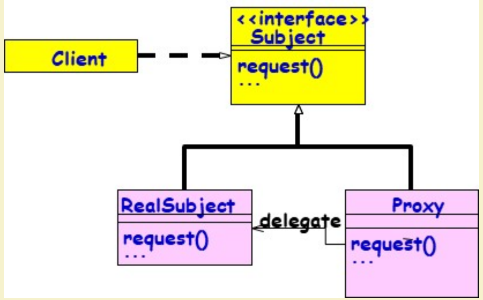
- **Client:**
  - – Always uses **Proxy**.


Subject
<<interface>>
f();
p();
g();
Proxy
f();
p();
g();
delegate
RealSubject
f();
p();
g();

Proxy Code

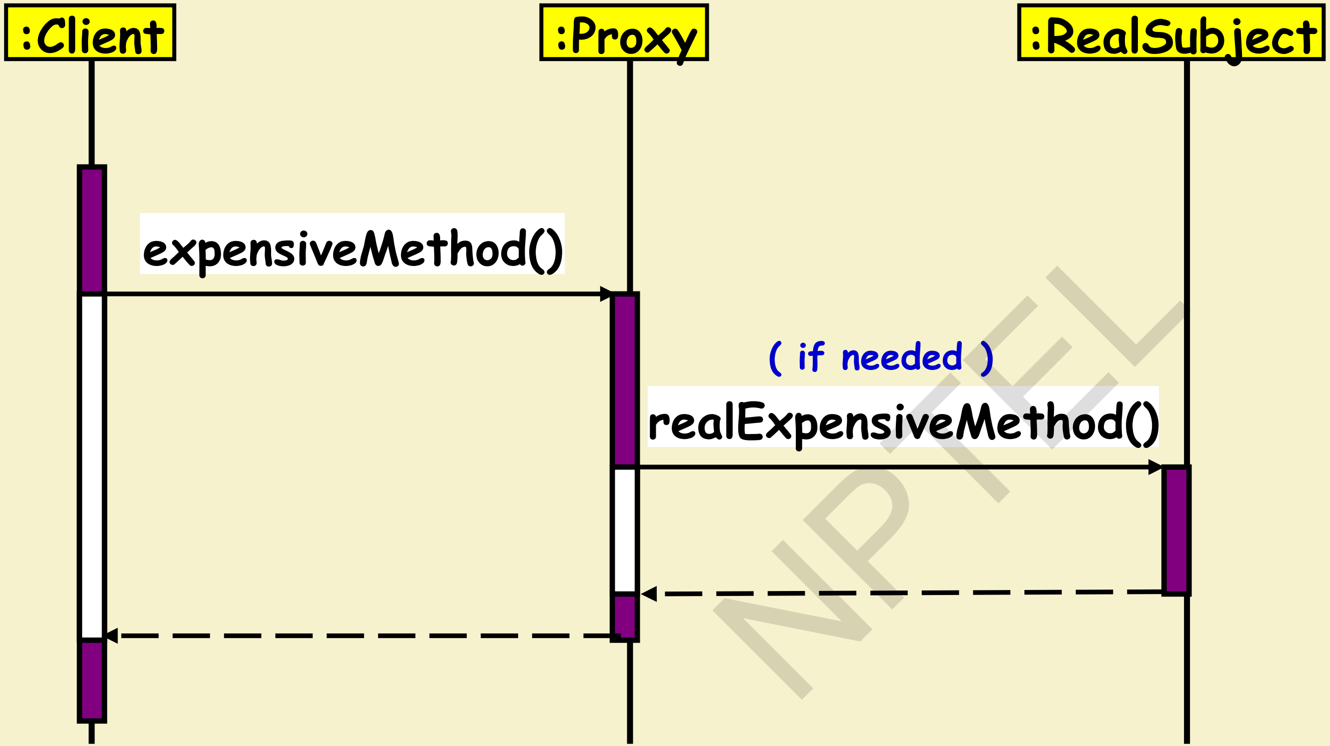
```
interface Subject { void f();  void g();  void h();}
class Proxy implements Subject {
  private Subject implementation;
  public Proxy() { implementation = new RealSubject(); }
  public void f() {implementation.f();}
  public void g() {implementation.g();}
  public void h() {implementation.h();}
}

class RealSubject implements Subject {
  public void f() {System.out.println("Implementation.f()");}
  public void g() {System.out.println("Implementation.g()");}
  public void h() {System.out.println("Implementation.h()");}
}
```



```
class Client{
  Proxy p = new Proxy();
  p.f(); p.g(); p.h(); }
```

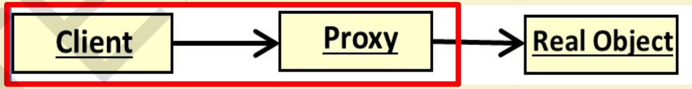
Sequence Diagram for Proxy



Proxy forwards requests to **RealSubject** when necessary.

Many Kinds of Proxies…

- **Virtual proxy:**
  - Delays creation or loading of large or computationally expensive objects (**lazy construction**)
  - Proxy is a standin --- **postpones accessing the real subject.**
- **Remote proxy:**



  - Use a local representative for a remote object (different address space)
  - **Hides the fact that an object is not local**
  - Encode and send the request to the real subject in a different address space.

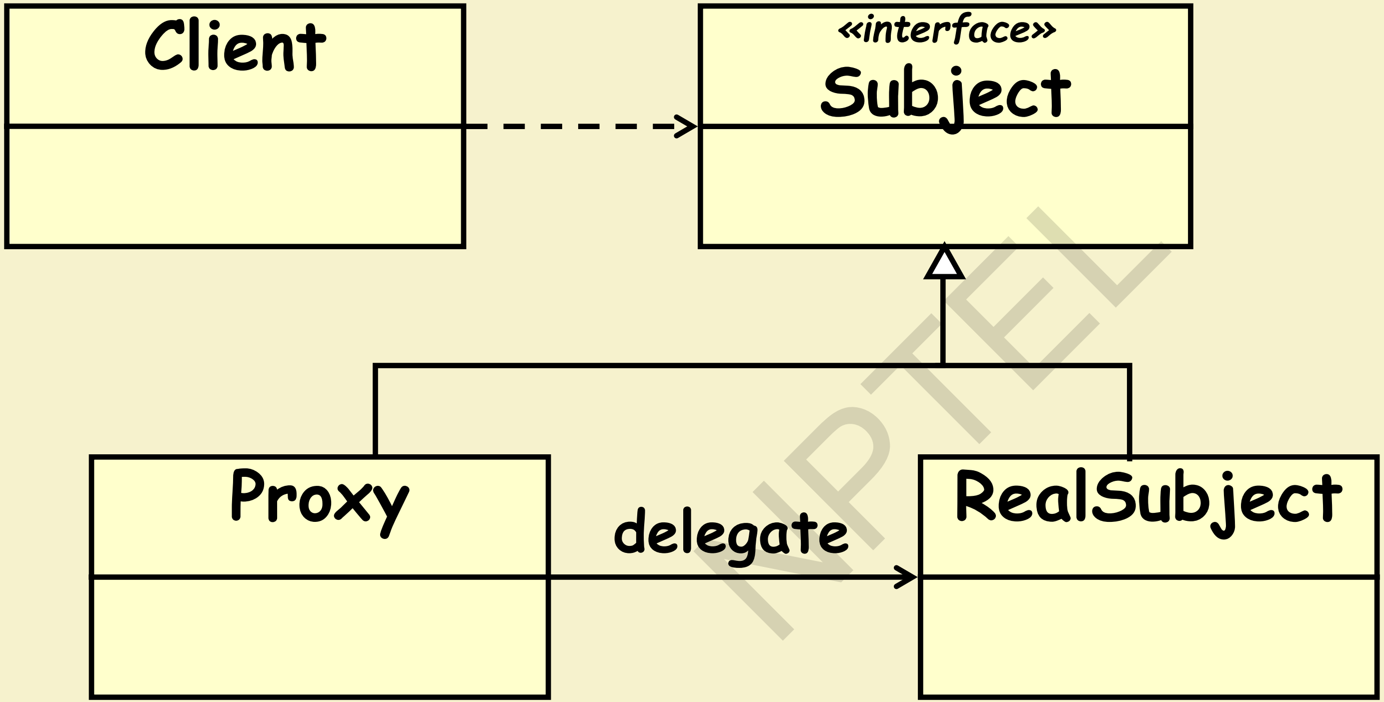
Kinds of Proxies

- **Synchronization Proxy**
  - Controls access to a target object when multiple objects access it.
- **Cache Proxy**
  - Hold results temporarily
  - Saves data for clients to share so data is only fetched or calculated once
  - **Caching of information: Helpful if information does not change too often.**
- **Copy-on-write**:
  - Postpones creation of a copy of an object until it is really necessary.

Kinds of Proxies

- **Protection Proxy**
  - Checks access permission to the real object when it is accessed.
  - **Ensures that only authorized clients access a supplier in legitimate ways**
  - Useful when different objects should have different access and viewing rights for the same document.
  - **Example:** Grade information shared by administrators, teachers and students.

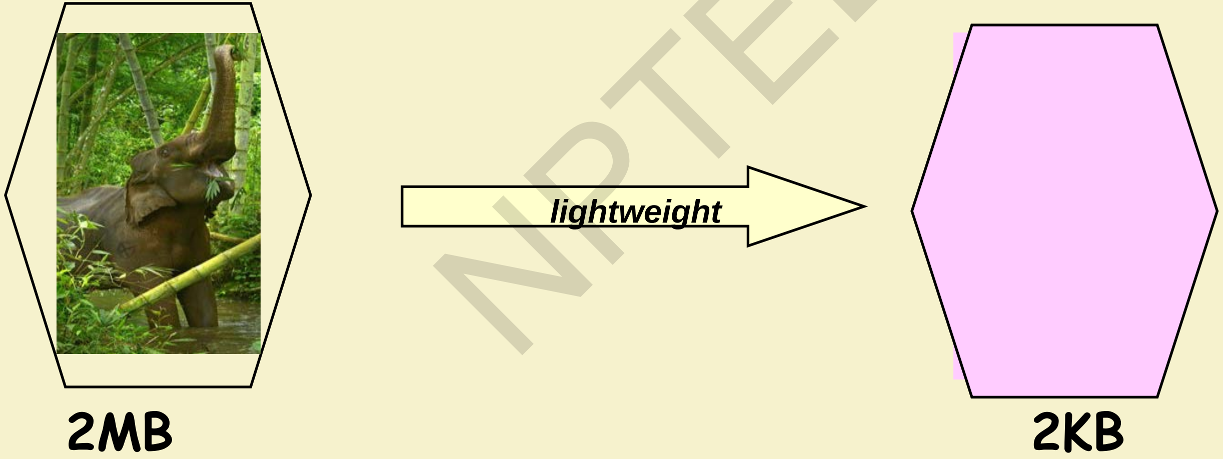
Proxy Pattern



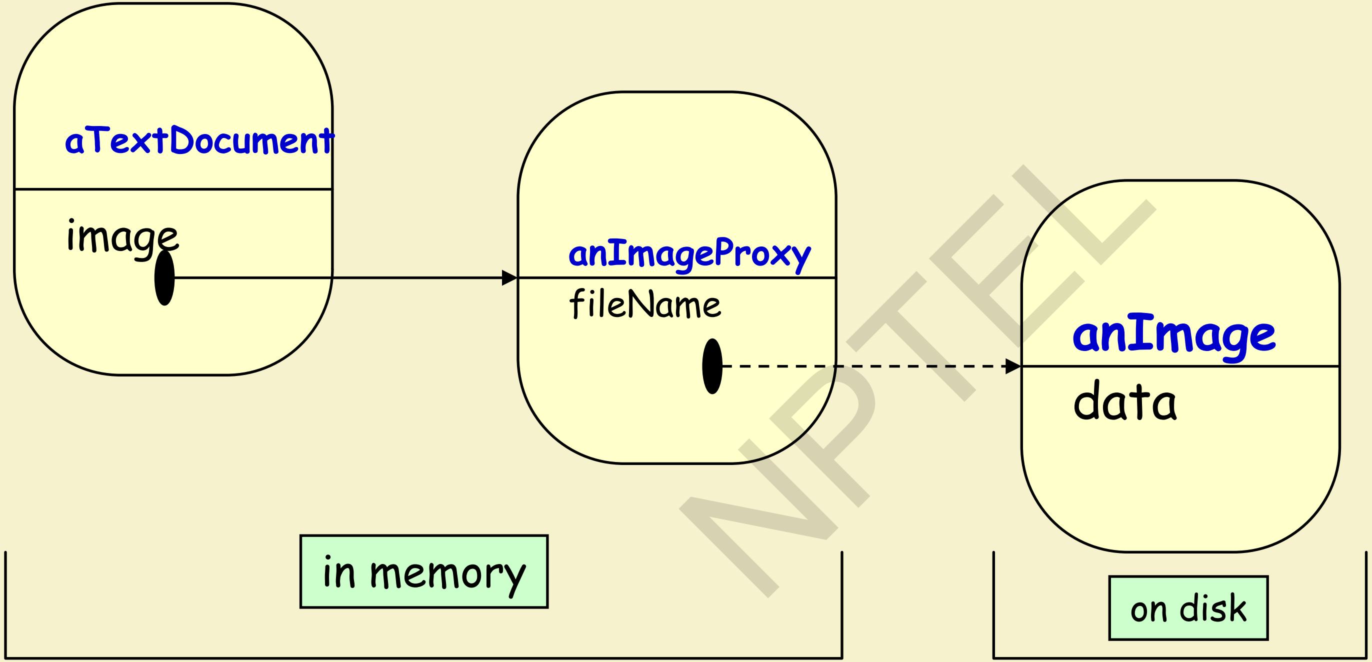
Virtual Proxy: Why Stand-in?

1. The image is expensive to load

2. The complete image is not always necessary

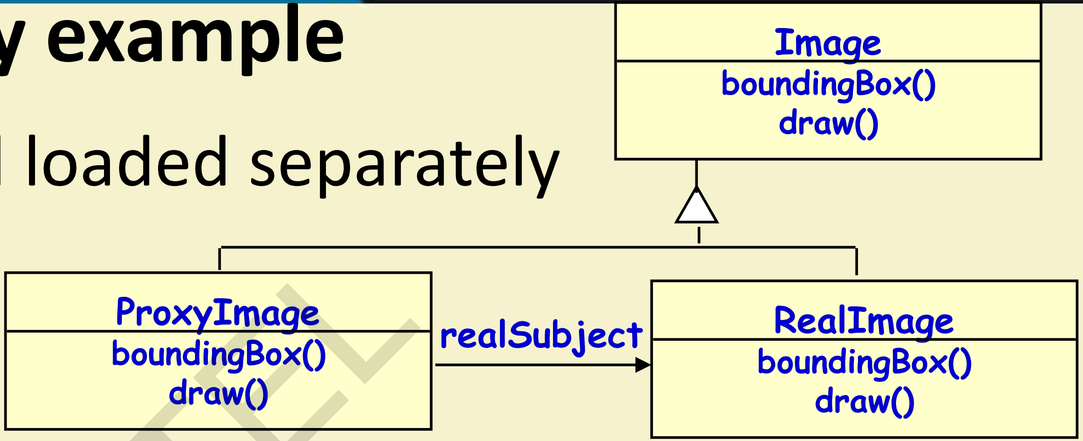


Virtual Proxy Motivation: Image Viewer



Virtual Proxy example

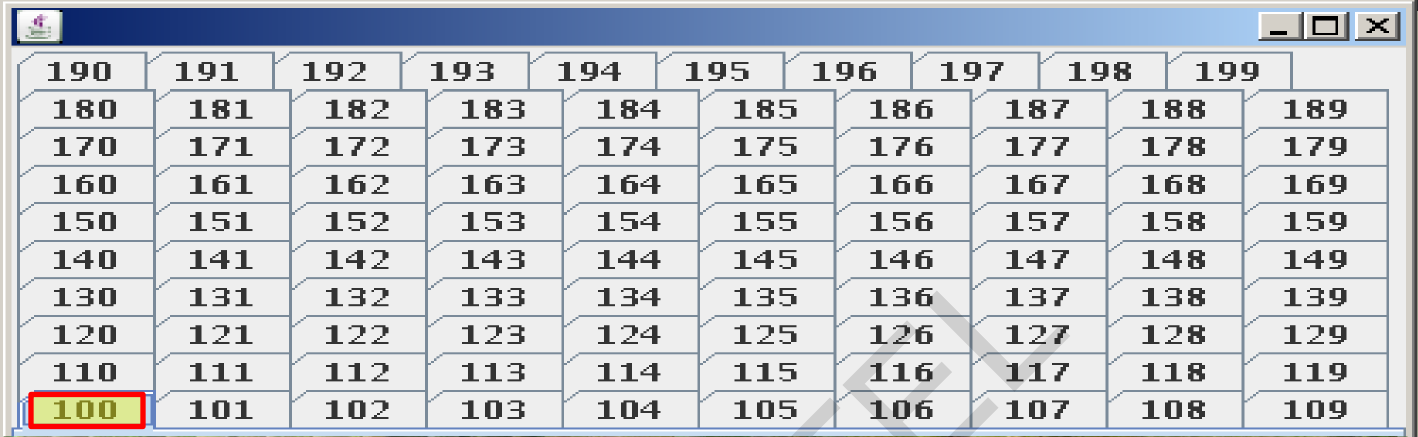
- Images are stored and loaded separately from text



- If a RealImage is not yet loaded, a ProxyImage displays a grey rectangle in place of the image
- The client cannot tell whether it is dealing with a ProxyImage instead of a RealImage

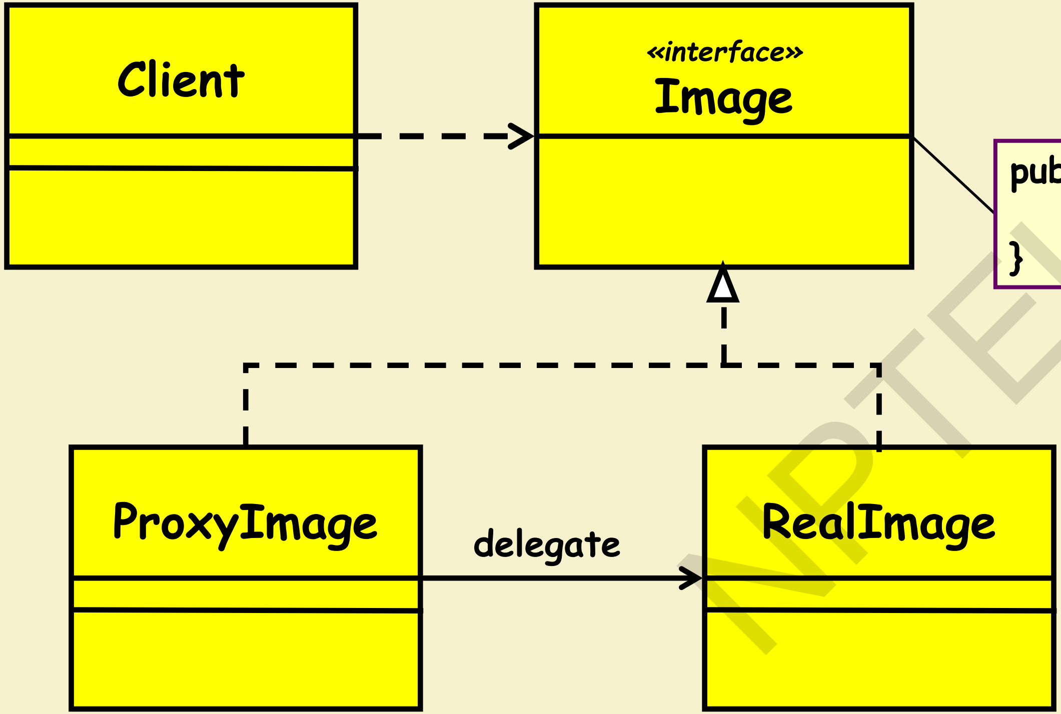
Example Application: Picture Viewer

- A picture viewer is to be developed that can should handle displaying many high quality images.
- Requirement: Opening the viewer should be fast:
  - Becomes slow if all images must be first loaded.
- No need to load all at the start:
  - All pictures not viewable in the display Window.
- Create images on demand !



190	191	192	193	194	195	196	197	198	199
180	181	182	183	184	185	186	187	188	189
170	171	172	173	174	175	176	177	178	179
160	161	162	163	164	165	166	167	168	169
150	151	152	153	154	155	156	157	158	159
140	141	142	143	144	145	146	147	148	149
130	131	132	133	134	135	136	137	138	139
120	121	122	123	124	125	126	127	128	129
110	111	112	113	114	115	116	117	118	119
100	101	102	103	104	105	106	107	108	109

Proxy Pattern Example 1



```
public interface Image {
      public void displayImage();
}
```

Similar Uses: Lazy Loading

- Word Processor:
  - Suppose a text document contains lots of multimedia objects and yet should load fast
  - Create proxies that represent large images, movies, etc. --- only load objects on demand as they become visible on the screen (only a small part of the document is visible at a time)

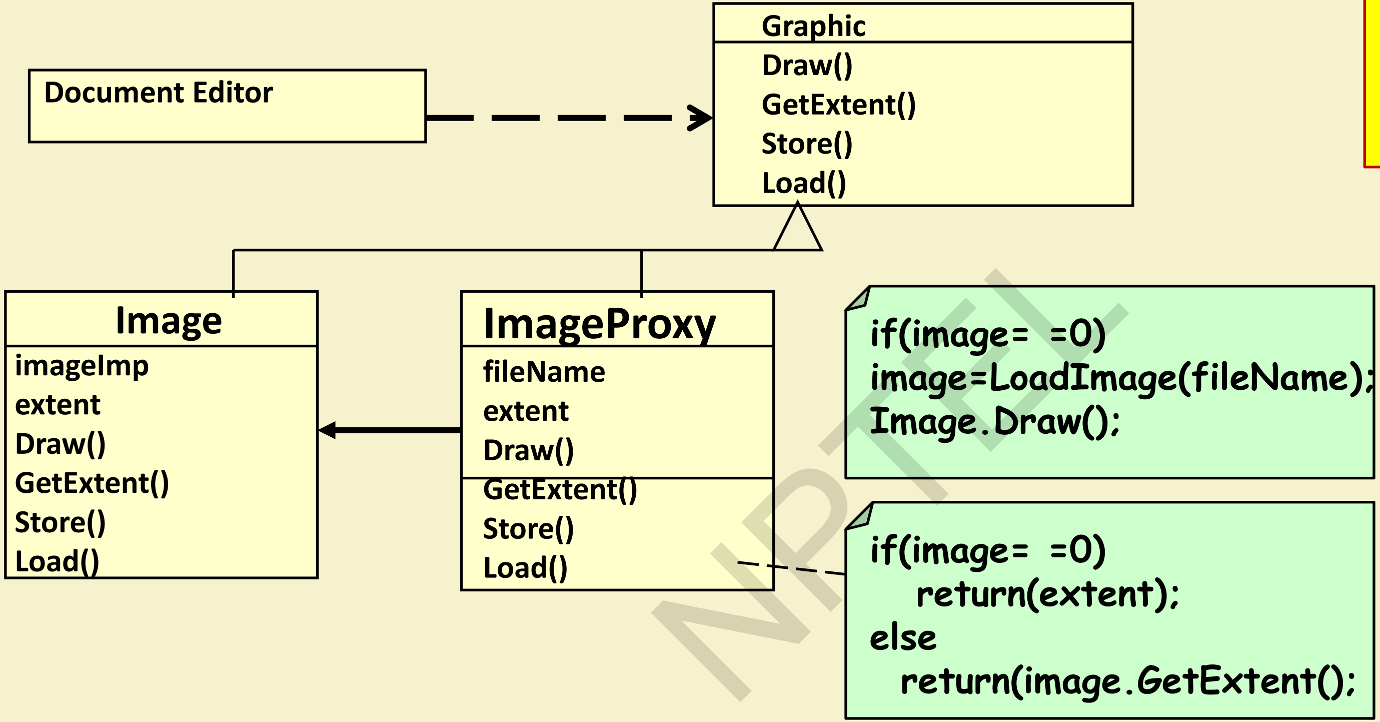
Lazy Loading

- Rather than instantiating an expensive object right away:
  - Create a proxy instead, and give the proxy to the client
- The proxy creates the object on demand when the client first uses it
  - **If the client never uses the object, the expense of creating it is never incurred…**

Lazy Loading

- A hybrid approach can be used:
  - **The proxy implements some operations itself.**
  - Create the real object only if the client invokes one of the operations it cannot perform
- A proxy stores necessary information to create the object on-the-fly:
  - file name, network address, etc.

Image Proxy: Code Example

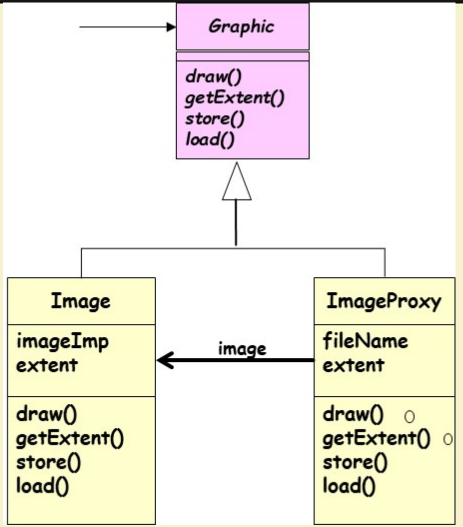


```
if(image= =0)
image=LoadImage(fileName);
Image.Draw();
```

```
if(image= =0)
   return(extent);
else
  return(image.GetExtent();
```

Image Proxy: Code Example

```
interface  Graphic {
   public void displayImage();
   public void loadImage();
}
class Image implements Graphic {
   private String filename;
   public Image(String filename) {
       this.filename = filename;
       System.out.println("Loading   "+filename);
     }
   public void displayImage() {
     System.out.println("Displaying "+filename); }
}
```



Proxy Pattern Example 1

```
public class ImageProxy implements Graphic {
private String filename;
private Image image;

    public ImageProxy (String filename){
        this.filename = filename;
    }
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename); //load only on demand
        }
        image.displayImage();
    }
}
```

delegate request to real subject

Proxy Example --- Client

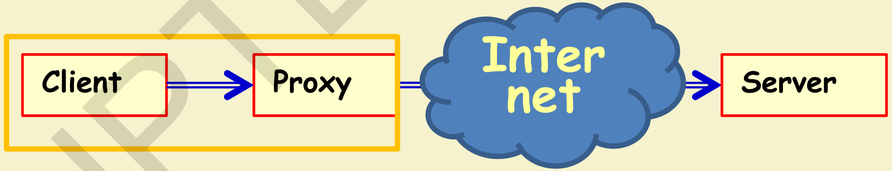
```
public class ProxyExample {
    public static void main(String[] args) {

        ArrayList<Image> images = new ArrayList<Image>();
        images.add(new ProxyImage("HiRes_10MB_Photo1"));
        images.add(new ProxyImage("HiRes_10MB_Photo2"));
        images.add(new ProxyImage("HiRes_10MB_Photo3"));

        images.get(0).displayImage();
        images.get(1).displayImage();
        images.get(0).loadImage();
    }
}
```

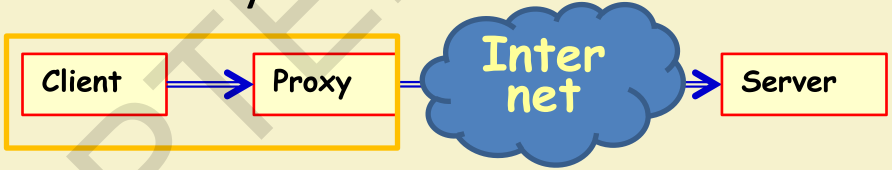
Remote Proxy

- The proxy object:
  - – Assembles data into a network message
  - – Sends it to the remote object (Server)
- A remote receiver:
  - – Receives this data,
  - – Turns it into a local message that is sent to the remote object



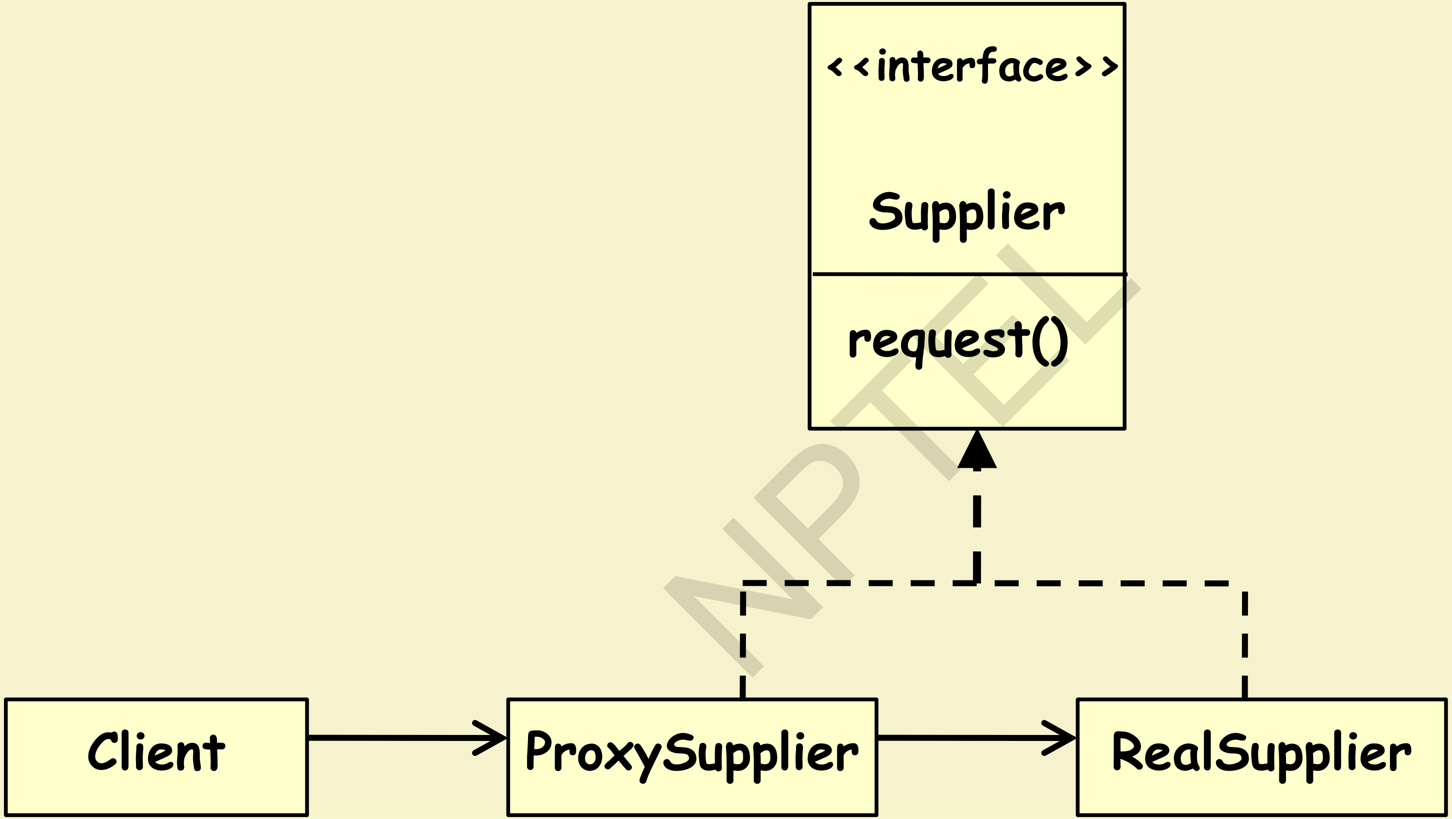
- **Stand-in for an object often needed because it is:**
  - Not locally available;
  - Instantiation and access are complex
  - Needs protected access for security.

Remote Proxy

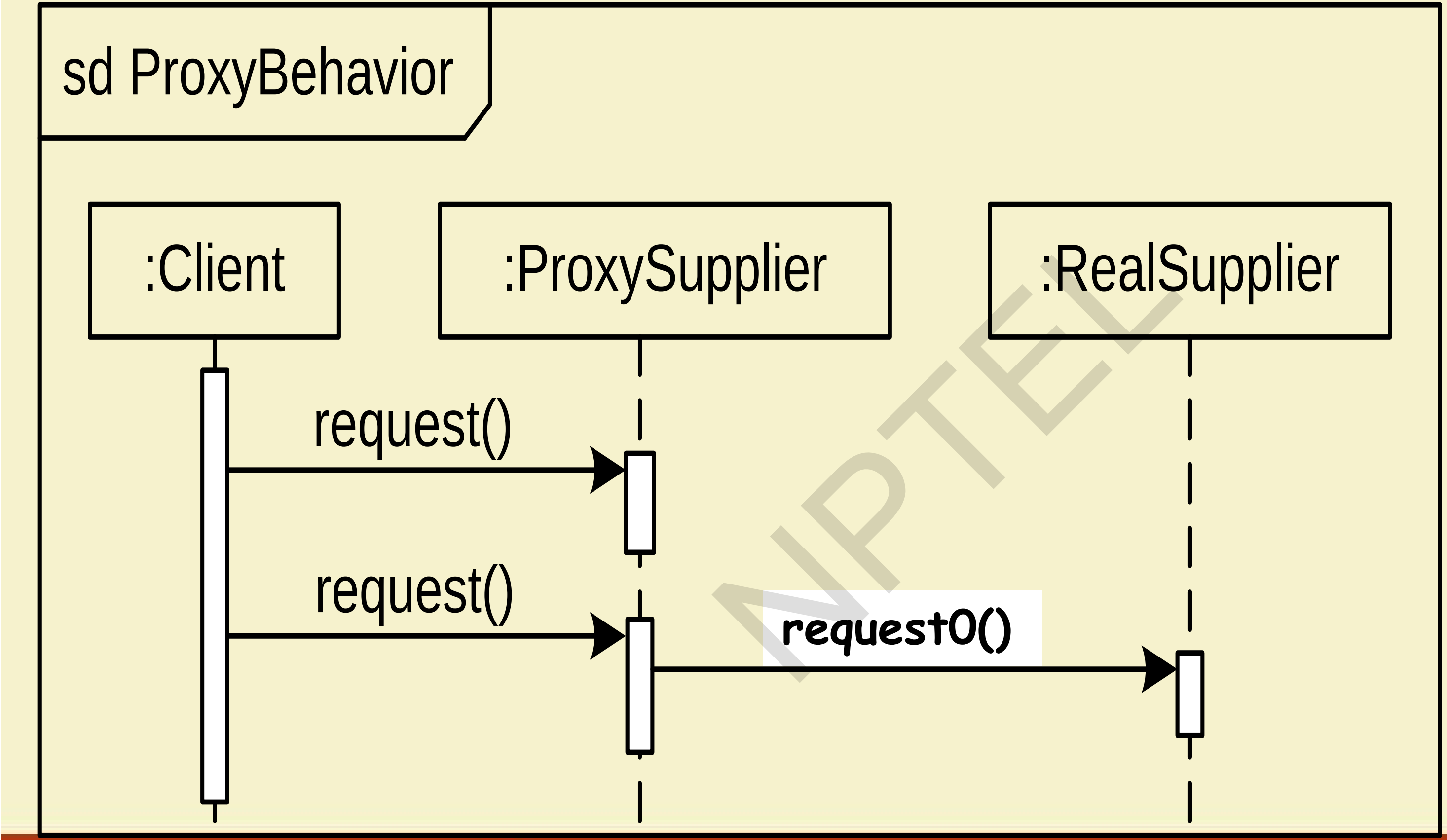


- **The stand-in must:**
  - Have the same interface as the real object;
  - Handle as many messages as it can;
  - Delegate messages to the real object when necessary.

Remote Proxy Pattern Structure



Proxy Pattern Behavior



Uses in Java Collection

- ## Read-only Collections
  - Wrap a collection object in a proxy that only allows read-only operations to be invoked on the collection
  - All other operations throw exceptions
  - `List unmodifiableList=Collections.unmodifiableList(List list);`
    - Returns read-only List proxy
- ## Synchronized Collections
  - Wrap collection object in a proxy that ensures only one thread at a time is allowed to access the collection
  - **Proxy acquires lock before calling a method, and releases lock after the method completes**
  - `List Collections.synchronizedList(List list);`
    - Returns a synchronized List proxy

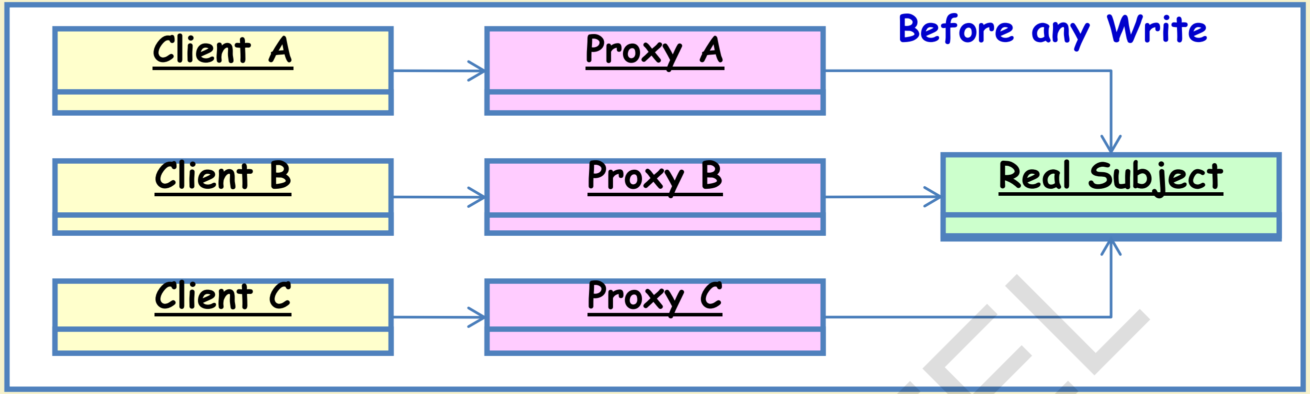
Other Proxy Uses: Copy-on-Write

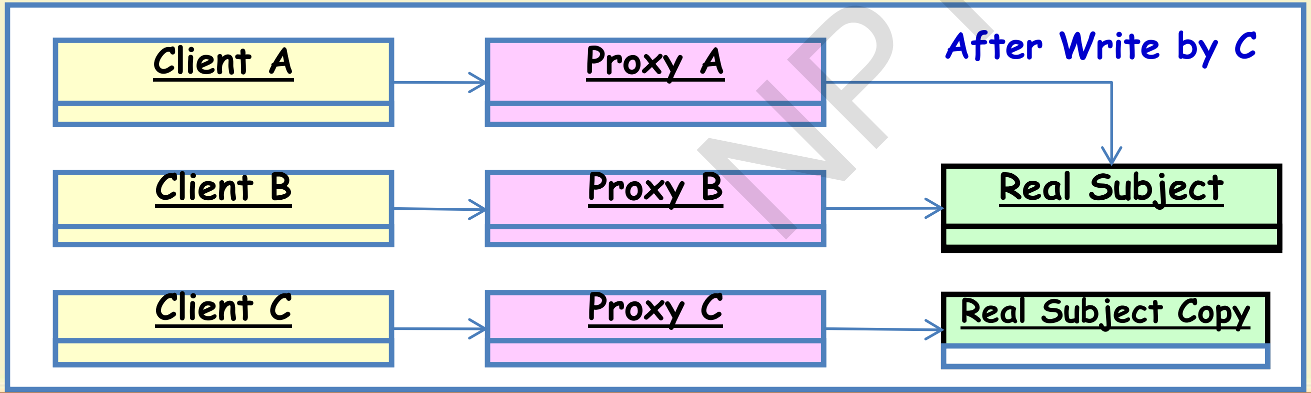
- Multiple clients share the same object:
  - as long as nobody tries to change it
- When a client attempts to change the object:
  - They get their own private copy of the object
- Read-only clients continue to share the original object:
  - while writers get their own copies
- **Maximize resource sharing, while making it look like everyone has own object.**

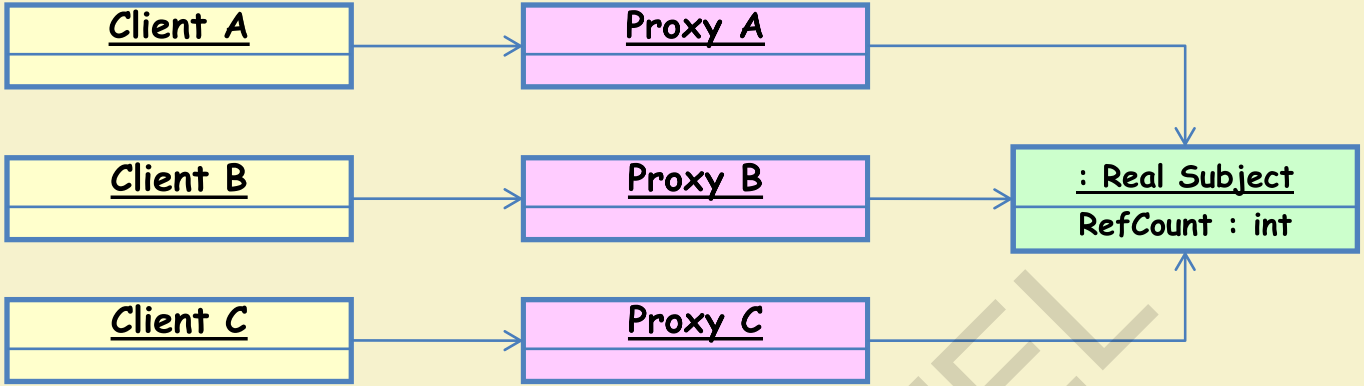
Uses: Copy-on-Write

- Highly optimized String classes often use this approach
- To make this work:
  - Clients are given proxies rather than direct references to the object
- When a write operation occurs:
  - **A proxy makes a private copy of the object on-the-fly to insulate other clients from the changes**

Decorator Known Uses: Copy-on-Write







Uses: Reference Counting

- Proxies maintain the reference count inside the object
- The last proxy to go away is responsible for deleting the object:
  - That is, when the reference count goes to 0, delete the object.

Proxy: Final Analysis

- A Proxy decouples clients from servers.
  - A Proxy introduces a level of indirection.
- **Proxy differs from Adapter in that it does not change the object’s interface.**

Proxy: Related Patterns

- **Adapter:**
  - Provides a different interface to an object,
- **Decorator:**
  - Similar structure as proxy, but different purpose.
  - **Decorator adds responsibilities whereas proxy controls access.**