Outline

Introduction to Design Patterns

- **What are Design Patterns ?**
- **History of design patterns ?**
- **How to describe design patterns ?**
- **Why study design patterns ?**
- **Classification of Design Patterns**

Design Patterns

- **Singleton, …**

What are Design Patterns ?

- **Design Patterns provide a vocabulary for understanding and discussing designs.**

-- from book of B. Liskov and J. Guttag

- **Design Patterns can improve performance or flexibility of code. They can increase complexity, and should be used only when analysis indicates the benefits outweigh the disadvantages**.

History of Design Patterns

Christopher Alexander says:

"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 even doing it the same way twice."

Pattern: solution to a problem in a context.

A Design Pattern has four essential elements:

- **Pattern name:** is a handle we can use to describe a design problem, its solutions and its consequences.

- **Problem:** the problem that the pattern is trying to solve. It describes when to apply the pattern

- **Solution:** how the pattern provides a solution to the problem in context.

- **Consequences:** are the results and trade-offs of applying the pattern.

Describing Design Patterns:

- **Pattern name:** all patterns have a unique name

- **Intent:** purpose of the pattern

- **Motivation:** a scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem.

- **Applicability:** what are the situations in which the design pattern can be applied ?

Describing Design Patterns: (contd.)

- **Generic Structure:** a graphical representation of the classes in the pattern using UML.

- **Participants and Collaborations:** the classes and/or objects participating in the design pattern, their responsibilities and collaborations

- **Consequences:** trade-offs and results of using the pattern.

- **Implementation:** how the pattern can be implemented.

Why study Design Patterns ?

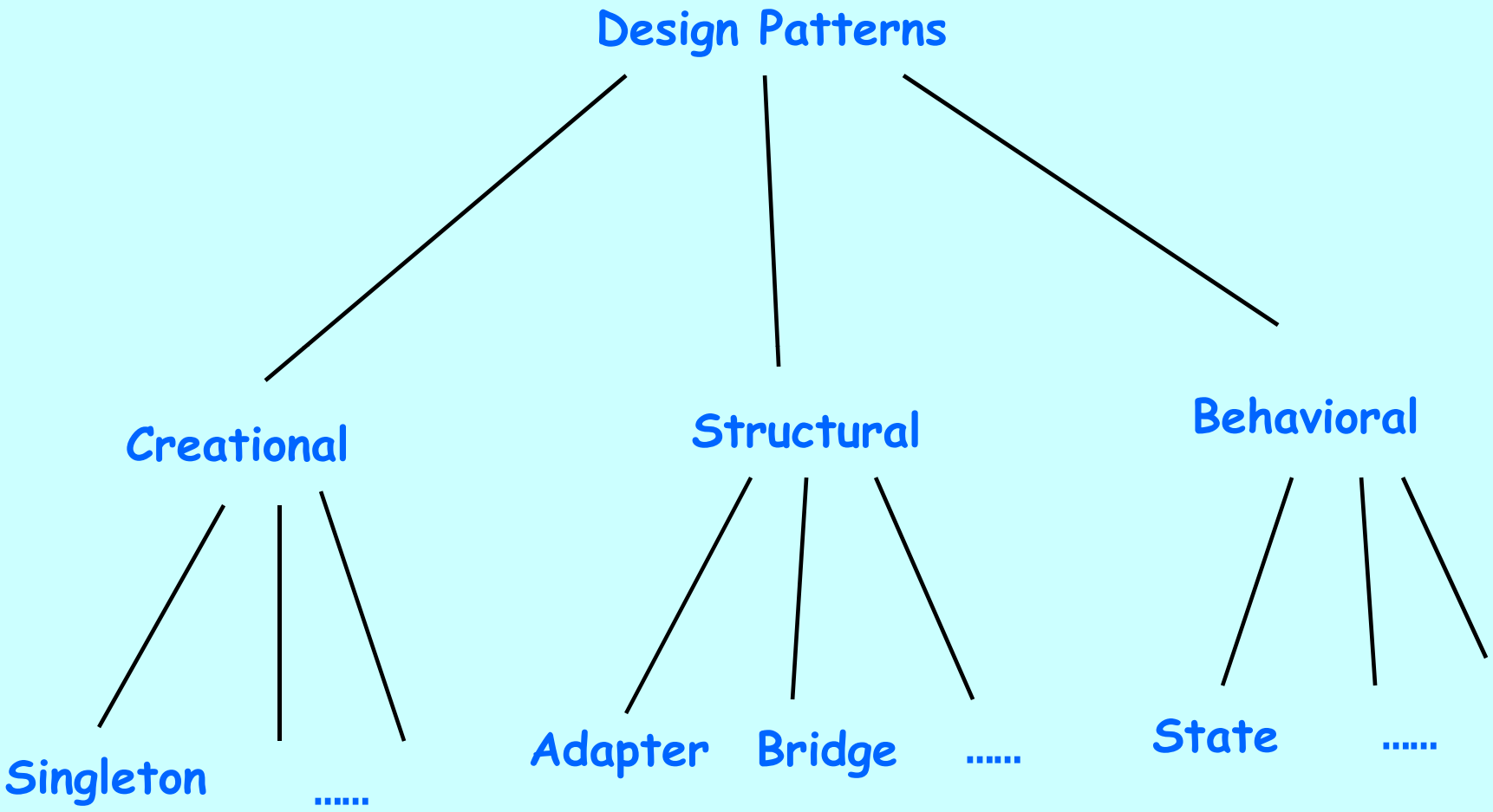
- **enable us to Reuse solutions**

- **enable us to establish common terminology**

- **give a higher level perspective on the problem and on the process of design**

- **improve modifiability and maintainability of code**

Classification of Design Patterns (Gang of Four book)



Singleton Pattern

Singleton Pattern (Creational Pattern):

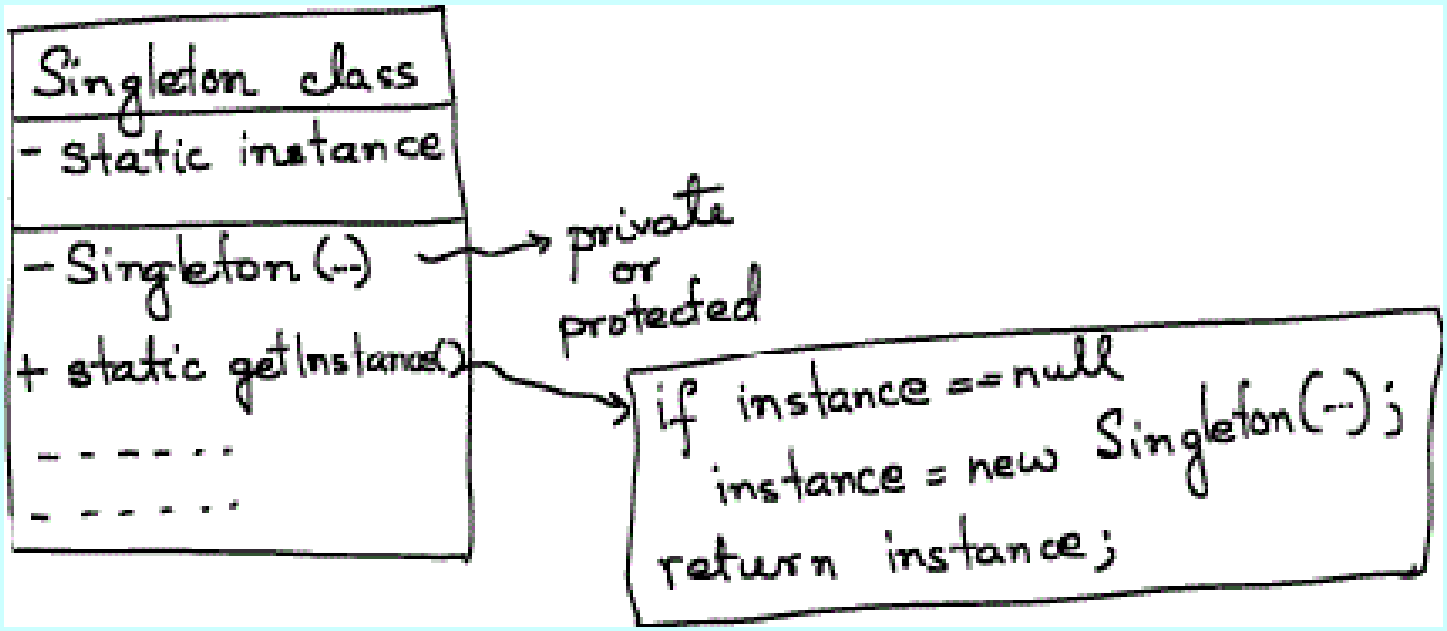
- **Intent:** Ensure a class only has one instance and provide a global point of access to it.

- **Motivation:** It is important for some classes to have exactly one instance. A good solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created and it provides a way to access the instance.

Singleton Pattern (contd.):

- **Applicability:** **there must be exactly one instance of a class.**

- **Generic Structure:**



Singleton Pattern (contd.):

- Participants and Collaborations: Clients create an instance of the Singleton solely through the static getInstance method.

- Consequences: several benefits:
  - controlled access to the sole instance
  - permits a variable number of instances easily

Singleton Pattern (contd.):

- Implementation (from Shalloway and Trot book):
  - Add a private static member of the class that refers to the desired object
  - Add a public static method that instantiates this class if this member is null and then returns the value of this member
  - Set the constructor private or protected so that no one can directly instantiate this class

Singleton Pattern (contd.):

- **Implementation (example):**

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {…… }
    public static Singleton getInstance() {
        if( instance == null )
            instance = new Singleton();
        return instance;
    }
    // instance data and instance methods
}
```

Adapter Pattern

Adapter Pattern (Structural Pattern):

- **Intent:** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

- **Motivation:** sometimes a class that's designed for reuse is not reusable only because its interface does not match the domain specific interface an application requires...

Example:

- Create classes for points, lines an squares that have the behavior "display".

- The client objects should not have to know whether they actually have a point, a line, or a square. They just want to know that they have one of these shapes.

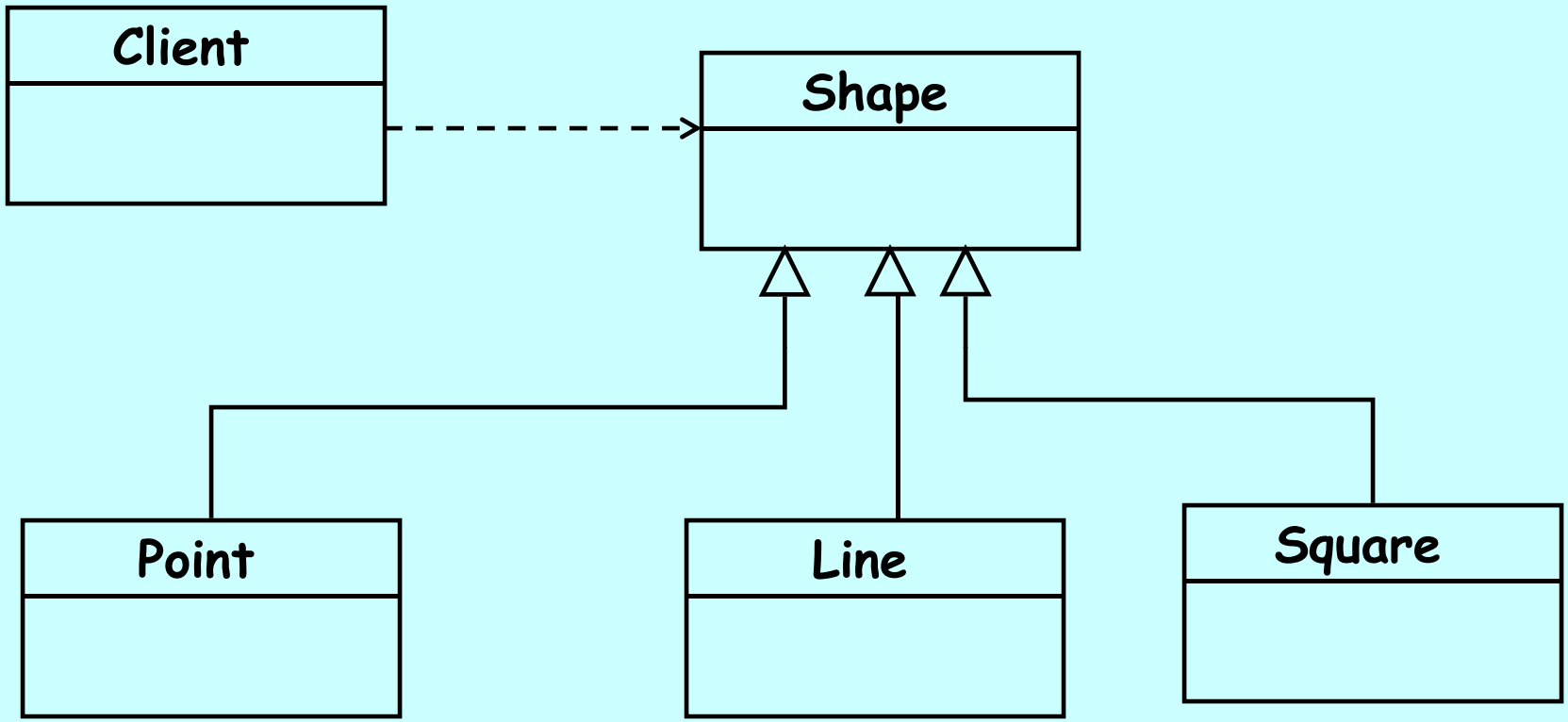
(from the book of Shalloway and Trott)

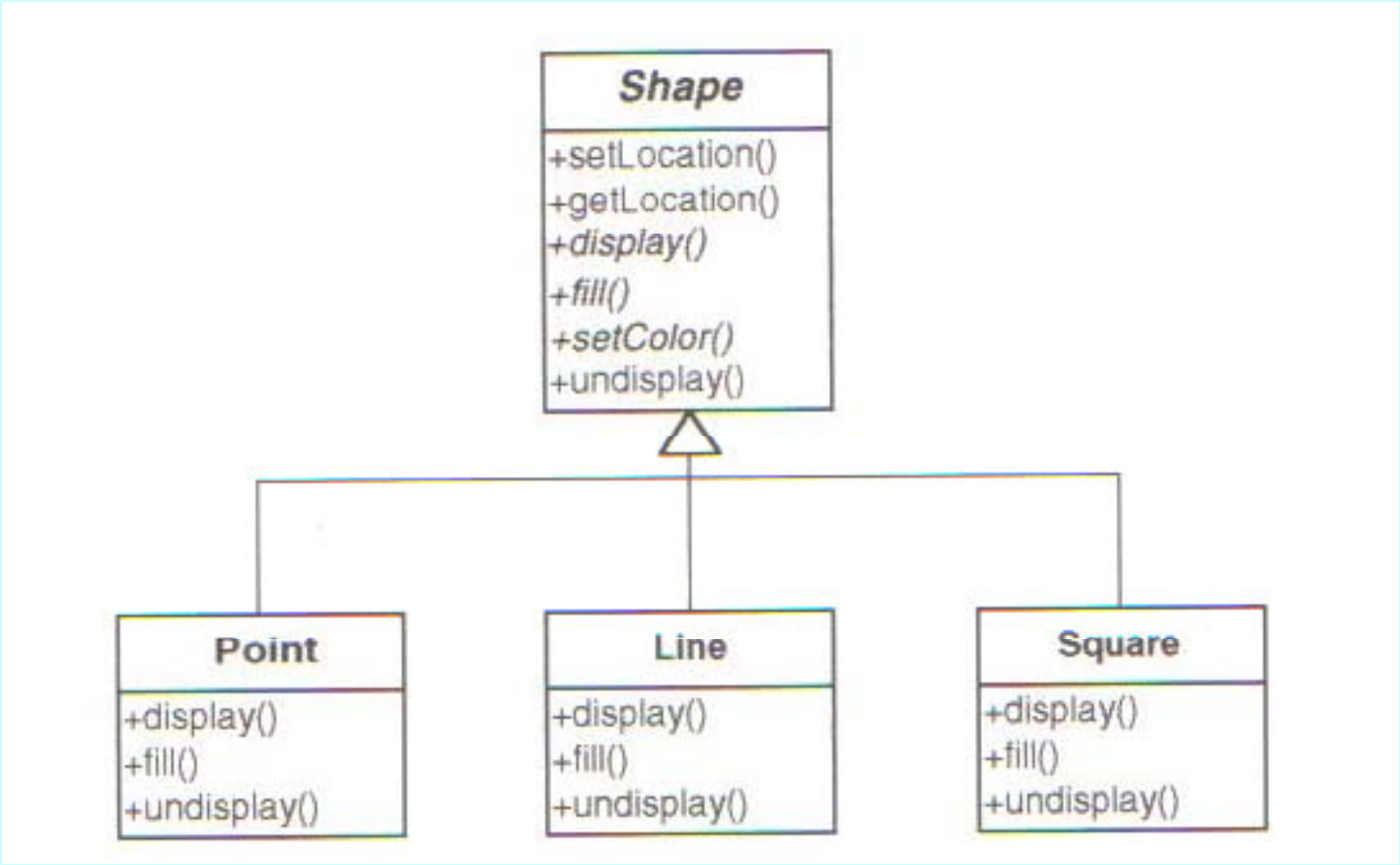
- Although the system will have points, lines and squares, the client objects want to think that they only have shapes.

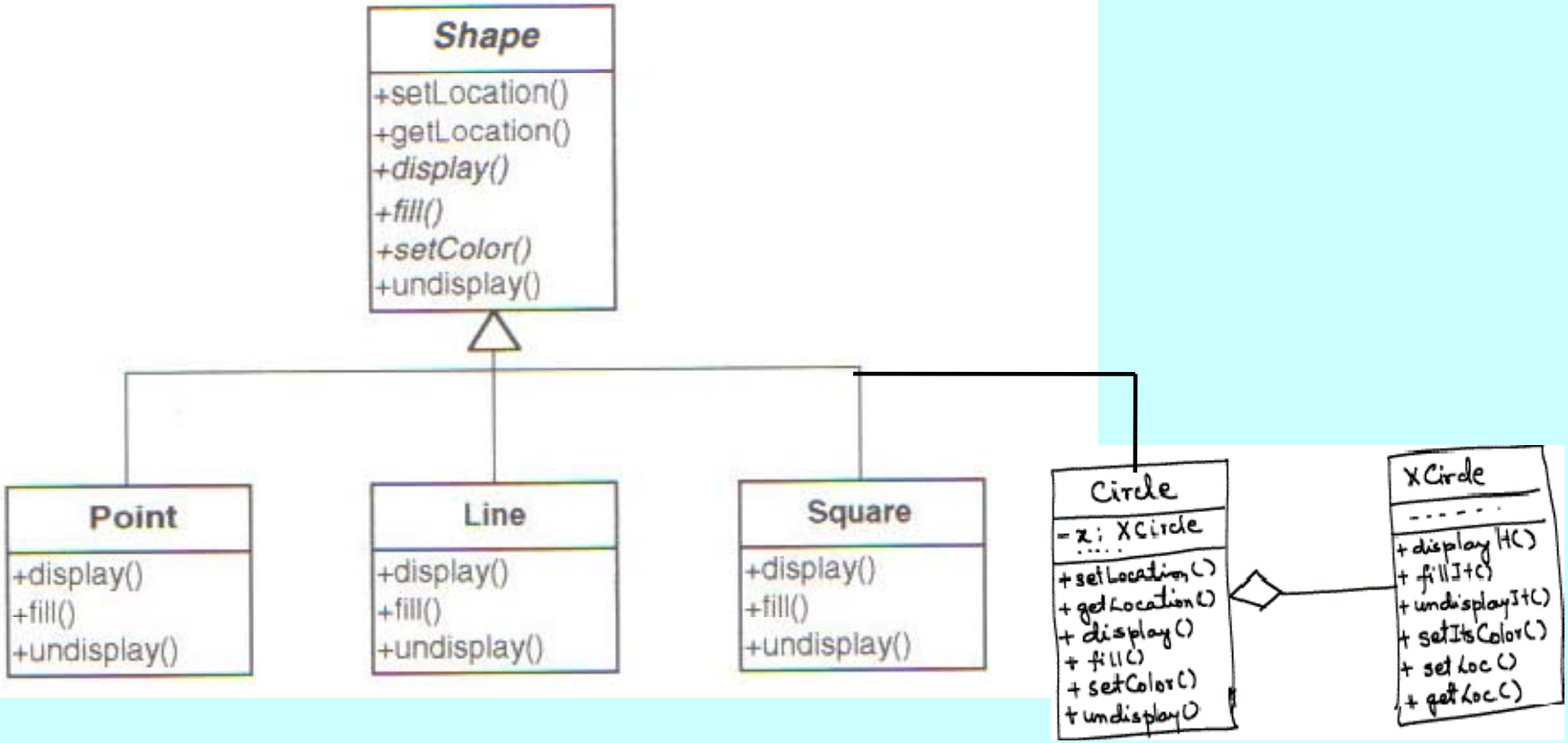
  - this allows client objects to deal with all these objects in same way

  - it enables us to add different kind of shapes in future without having to change the clients

Use Polymorphism


Client
Shape
Point
Line
Square


Shape
+setLocation()
+getLocation()
+display()
+fill()
+setColor()
+undisplay()
Point
+display()
+fill()
+undisplay()
Line
+display()
+fill()
+undisplay()
Square
+display()
+fill()
+undisplay()


Shape
+setLocation()
+getLocation()
+display()
+fill()
+setColor()
+undisplay()
Point
+display()
+fill()
+undisplay()
Line
+display()
+fill()
+undisplay()
Square
+display()
+fill()
+undisplay()
Circle
- x: XCircle
+ setLocation()
+ getLocation()
+ display()
+ fill()
+ setColor()
+ undisplay()
XCircle
+ displayH()
+ fillH()
+ undisplayH()
+ setItsColor()
+ setLoc()
+ getLoc()

```
public class Circle extends Shape {
    …
    private XCircle x;
    …
    public Circle() { …
        x = new XCircle();
    }
    public void display() {
        x.displayIt();
    }
    …
}
```

Adapter Pattern (contd.):

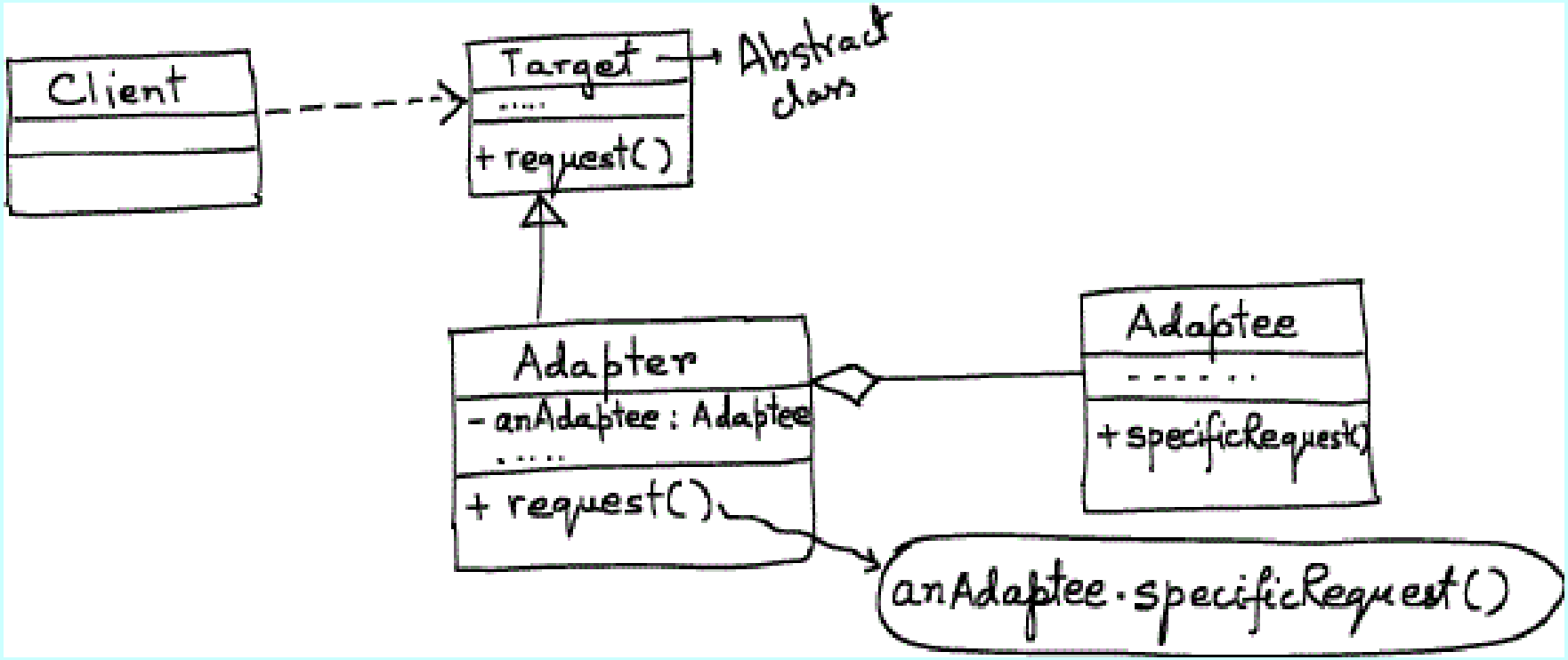
- **Applicability: Use it when:**

- **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, that is, classes don't necessarily have compatible interfaces.**

Adapter Pattern (contd.):

- **Generic Structure:**



The Adapter provides a wrapper with the desired interface

Adapter Pattern (contd.):

- Participants and Collaborations: Target, Client, Adaptee, Adapter.

Adapter adapts the interface of an Adaptee to match that of an Adapter's Target. This allows Client to use the Adaptee as if it were a type of Target.

Clients call operations on an Adapter instance. In turn, adapter calls Adaptee methods that carry out the request..

Adapter Pattern (contd.):

- **Consequences:**
  - the Adapter pattern allows for preexisting objects to fit into new class structures without being limited by their interfaces.

Adapter Pattern (contd.):

- **Implementation (from Shalloway and Trot book):**
  - **Contain the existing class in another class.**
  - **Have the containing class match the required interface and call the methods of the contained class.**

Composite Pattern

Composite Pattern (Structural Pattern):

- **Intent (GOF):** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- **Motivation:** we often must manipulate composite objects exactly the same way we manipulate primitive objects..

Composite objects: objects that contain other objects

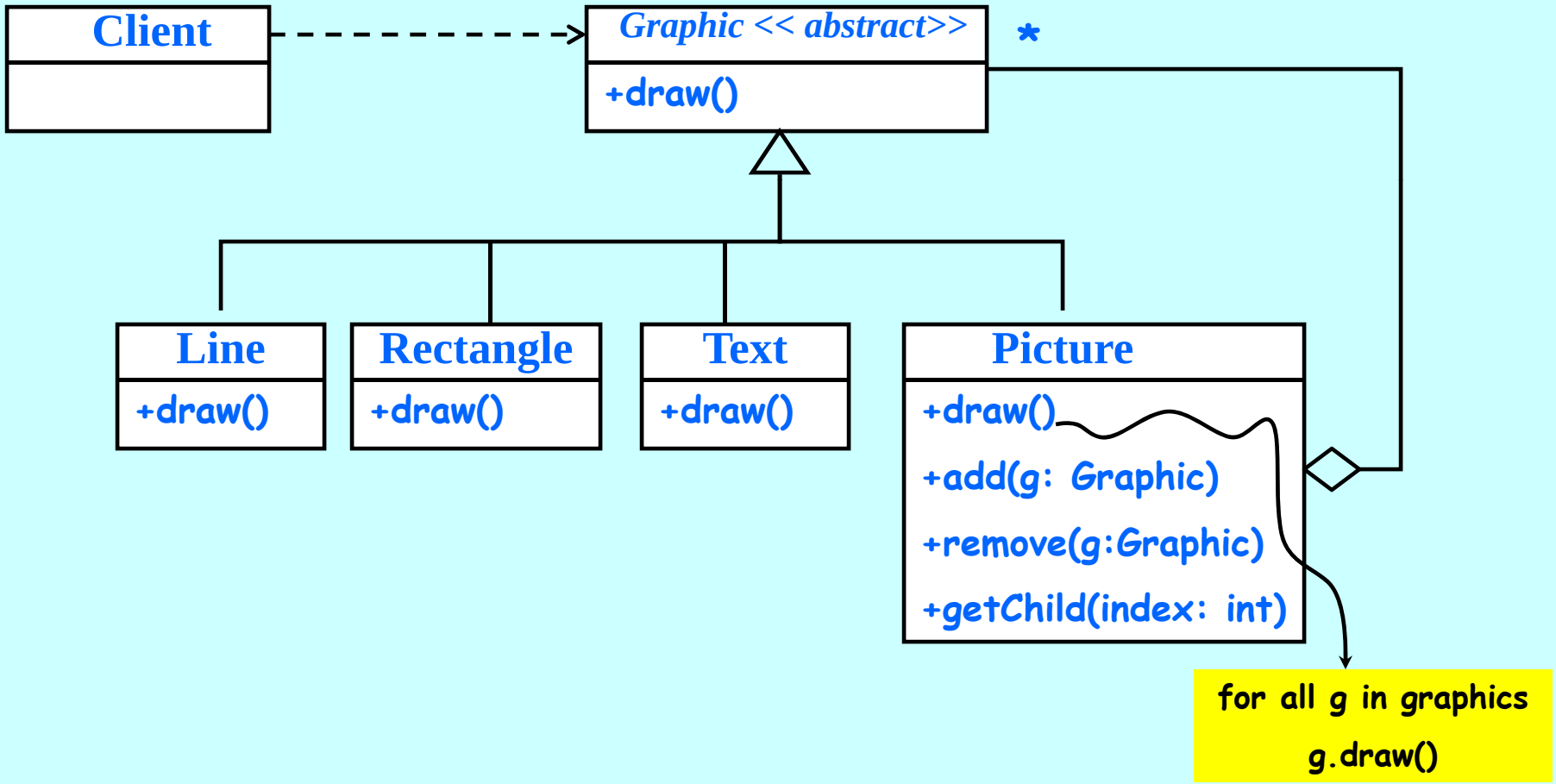
Example:

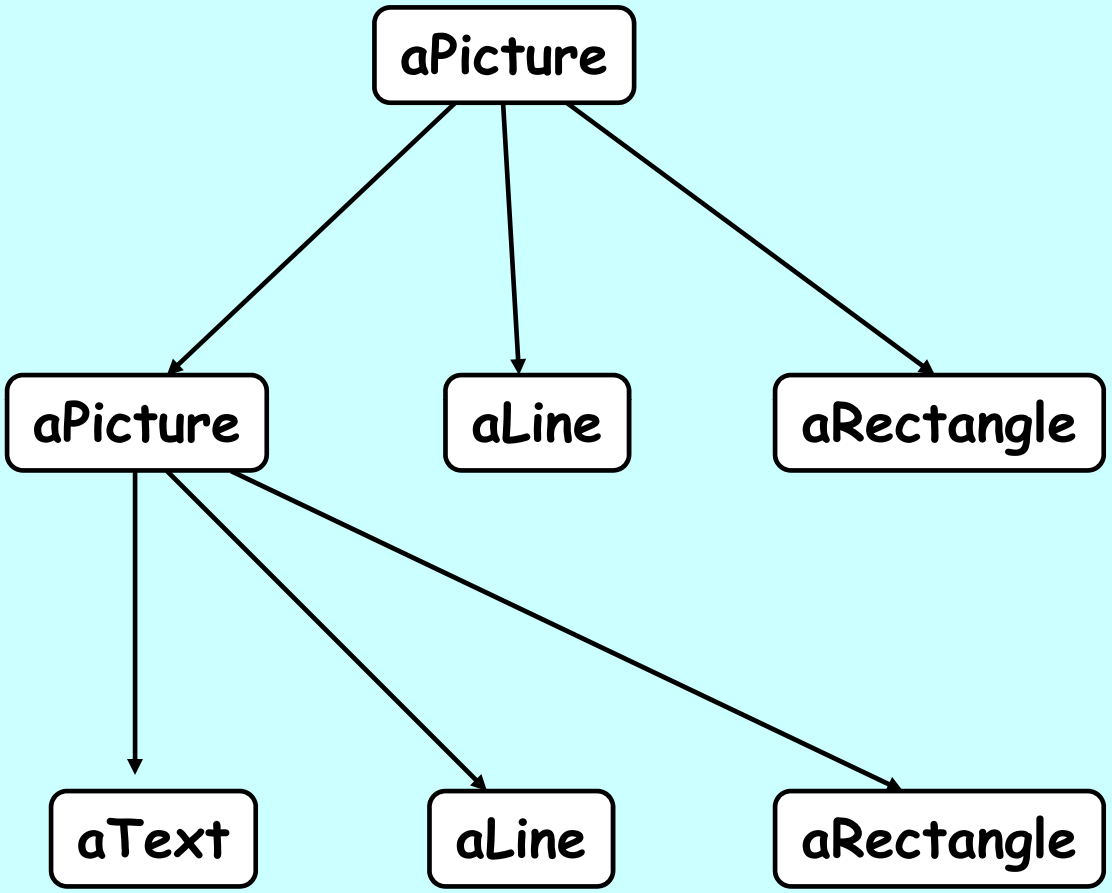
Graphics applications like drawing editors let users build complex diagrams. The user can group component to form larger components.

Simple implementation: define classes for graphical primitives like Text, Line plus other classes that act as containers for these primitives..

Problem with this approach: Code that uses these classes must treat primitive and container objects differently..

The Composite pattern describes how to use recursive composition so that clients do not have to make this distinction..




aPicture
aPicture
aLine
aRectangle
aText
aLine
aRectangle

Composite Pattern (contd.):

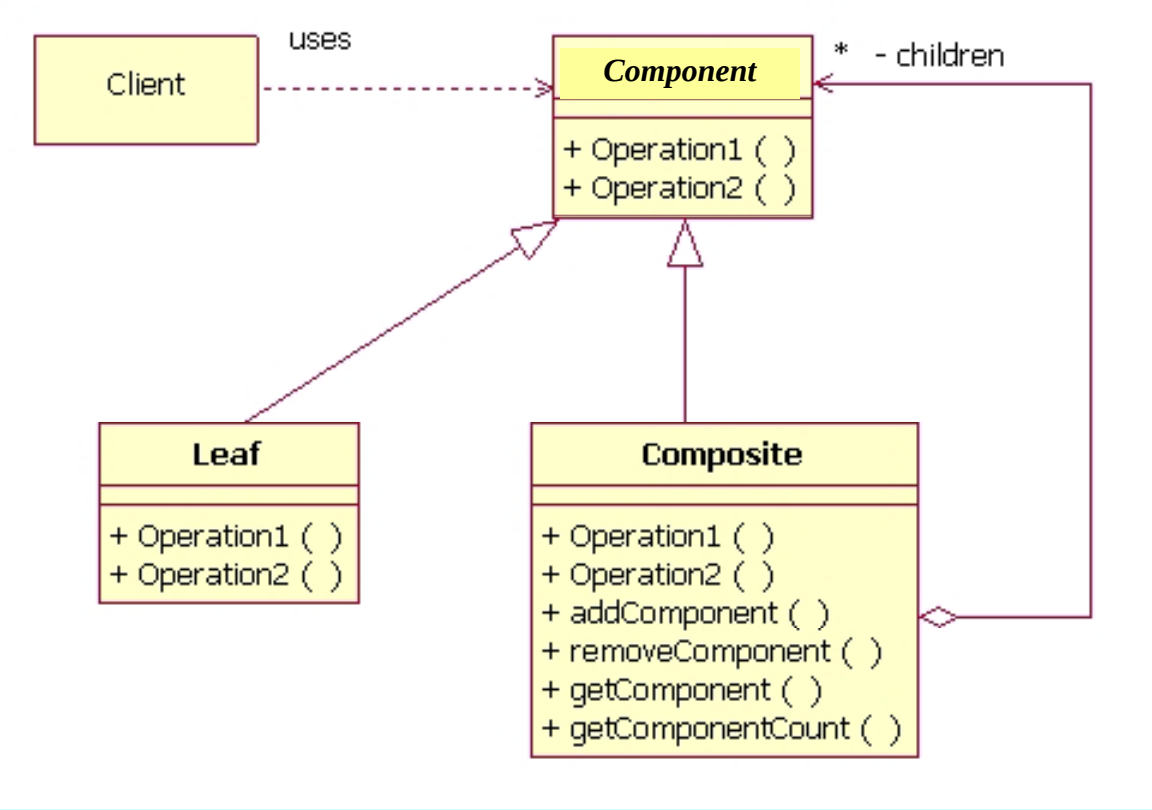
- **Applicability: Use it when:**

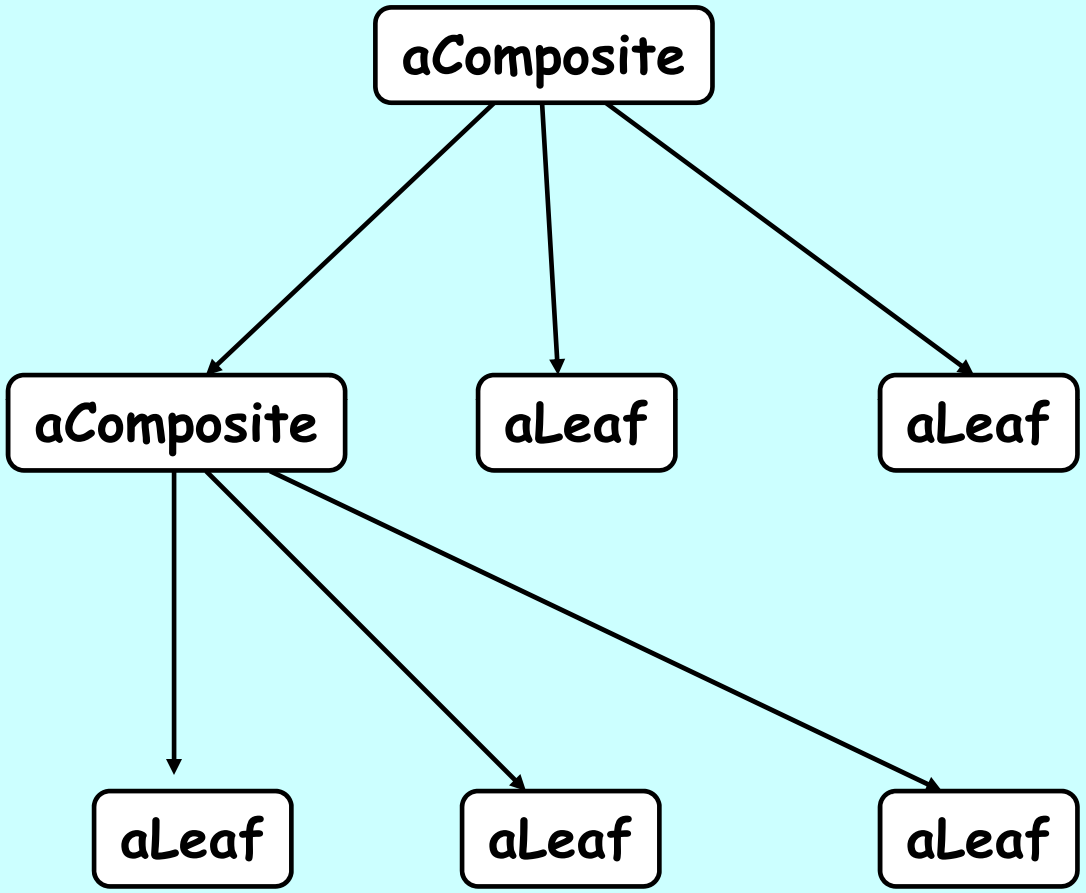
  - **you want to represent part-whole hierarchies of objects**

  - **you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.**

Composite Pattern (contd.):

- **Generic Structure (Alternative 1): Children managing operations are in the Composite class; *Component* is abstract**




aComposite
aComposite
aLeaf
aLeaf
aLeaf
aLeaf
aLeaf

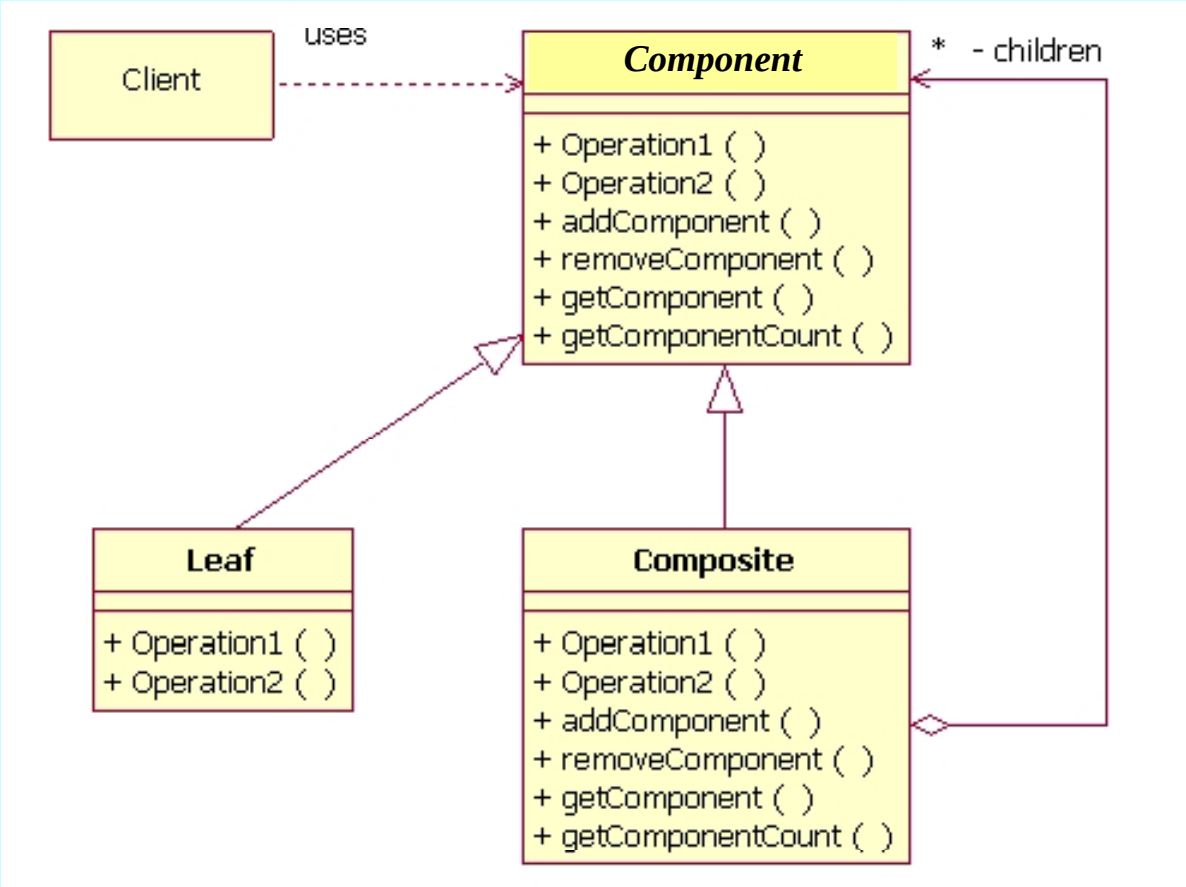
Composite Pattern (contd.):

- Participants and Collaborations: Component, Leaf, Composite, Client.

Clients use the Component class to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components.

Composite Pattern (contd.):

- Generic Structure (Alternative 2): Children managing operations are in Component and Composite (possibly in Leaf as well) ; *Component* is abstract



Composite Pattern (contd.):

- **Consequences:**
  - makes it easire to add new kinds of components. Clients do not have to be changed for new component classes.
  - makes the Client cimple. Clients can treat composite structures and individual objects uniformly.
  - defines class hierarchies consisting of primitive objects and composite objects.
  - can make the design overly general..

Composite Pattern (contd.):

- Example Implementation:

```
public abstract class Component  {
   public abstract void operation();

   public abstract void addComponent(Component c);
   public abstract void removeComponent(Component c);
   public  abstract Component getComponent( int j );
   public abstract int getComponentCount();
}
```

```
public class Leaf  extends Component {
   public void operation() {
      System.out.println( “Leaf: operation();” );
   }

   public void addComponent(Component c) { }
   public void removeComponent(Component c) { }
   public Component getComponent(int j) { return null; }
   public int getComponentCount() {return 0;}
}
```

```
public class Composite  extends Component {
  private Vector<Component> components;
  public Composite() {
    components = new Vector<Component>();
  }
  public void operation() {
    System.out.println( “Composite: operation();” );
    Enumeration<Component> list = components.elements();
    while( list.hasMoreElements() ) {
       Component c = (Component)list.nextElement();
       c.operation();
    }
  }
```

```
    public void addComponent(Component c) {
        components.addElement( c );
    }
    public void removeComponent(Component c) {
        components.removeElement( c );
    }
    public Component getComponent(int j) {
        return components.get( j );
    }
    public int getComponentCount() {
        return components.size();
    }
}
```

```
public class Client {
   public static void main(String args[]) {
      Component c1 = new Composite();
      Component c2 = new Leaf();
      Component c3 = new Leaf();
      c1.addComponent(c2);
      c1.addComponent(c3);

      c1.operation();
   }
}
```

Output:

```
Composite: operation();
Leaf: operation();
Leaf: operation();”
```

Output ??