# Adapter

## **Intent**

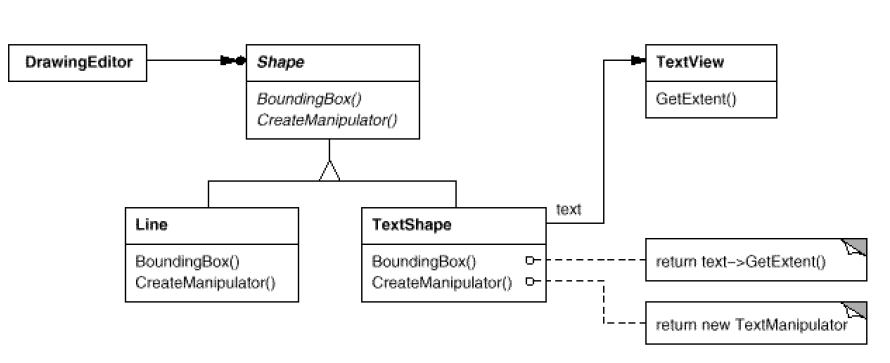
* Create a class that is responsible to join functionalities of independent or incompatible interfaces
* Or GoF: Convert the interface of a class into another interface clients expect

## **Motivation**

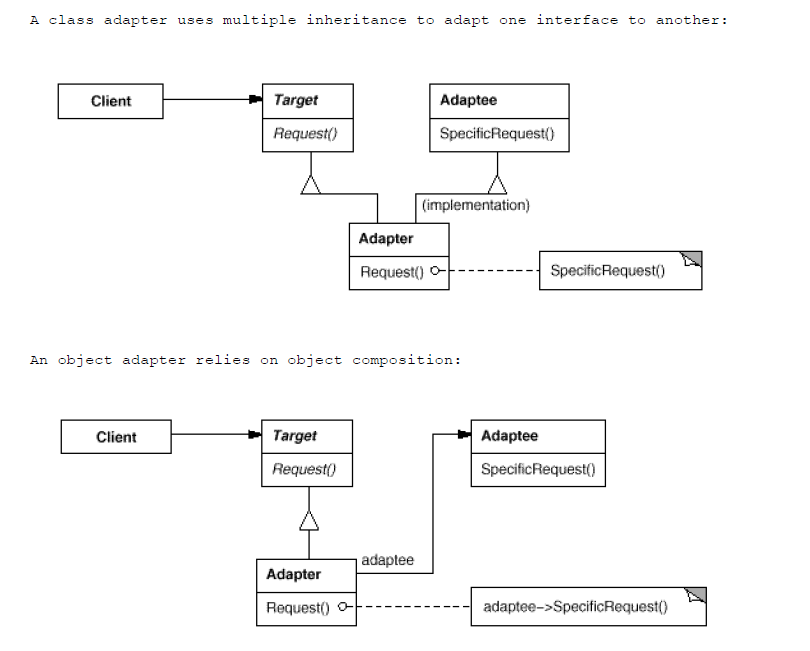
* Suppose that you a drawing editor working with Shape objects
* Line, etc are already implemented but you don’t have a text view object!
* However, on off the shelve library gives you a text implementation but you can’t use this directly because it doesn’t conform with your Shape type (remember polymorphism, liskov principle, etc ☺ ).
* Solution: create an **adapter** class that either: (1) derives the TextView and Shape, and implements the shape functionality by directing calls to TextView or (2) compose a TextView and derive the Shape.
* Second is commonly more suitable when you try to adapt more than one object at the same time ; but first doesn’t hve indirection

The adapter can:

* + Reuse functionality already implemented: check the BoundingBox and GetExtent redirect.
  + Add additional functionality that TextView doesn’t have: see createManipulation func.
  + Adapt more than one object at the same time !



## **Structure**



## **Example**

* See the code attached

# Bridge

## **Intent**

## Decouple the implementation from its abstraction and let them vary independently.

## **Motivation**

* You want to let abstraction and implementation be changed event at runtime without having client know or recompile
* You want run-time binding of the implementation not compile-time.
* Share an implementation among different objects
* You need to map orthogonal (unrelated) class hierarchies
* When mapping, you don’t want to create permutations of classes.

Check the simple example below where we have two hierarchies (except from stackoverflow example):

Let’s say you have Shapes (Rectangle, Circle) and Color (Blue, Red) implementations. Then you want to assign Color functionality to Shapes. Two solutions:

Inheritance:

----Shape---

/ \

Rectangle Circle

/ \ / \

BlueRectangle RedRectangle BlueCircle RedCircle

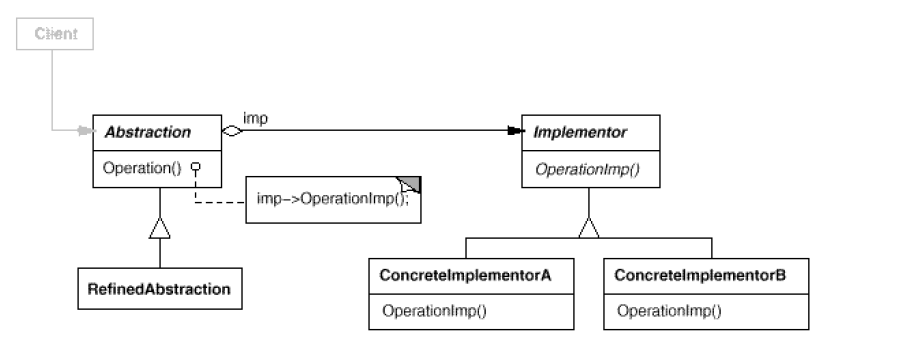
Let them vary independently and bring the bridge between them:

----Shape--- Color

/ \ / \

Rectangle(Color) Circle(Color) Blue Red

## **Structure**



## **Example**

* See the code attached
* Note 1: In the code, the Color interface is set a-priori but you can change it at run-time!
* Note: check GoF example, **you can use a factory to create the implementation.**

# Composite

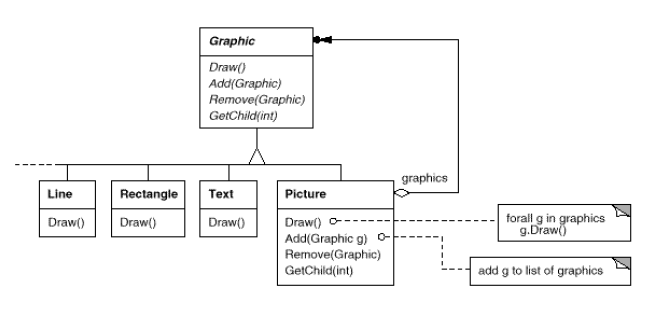
## **Intent**

## Composite lets clients treat individual objects and compositions of objects

## uniformly.

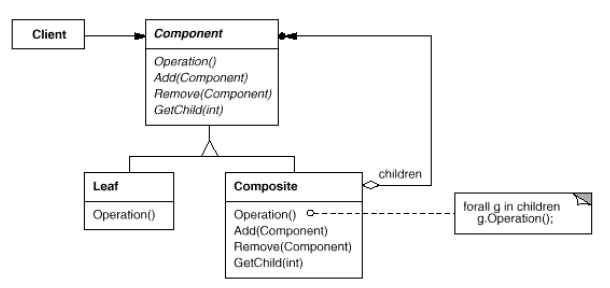
## **Motivation**

* Suppose you have a hierarchy of graphical objects and some of them represent composites
* You want to control them with the same interface and allow clients to see all these objects uniformly: they must ignore the different between composition of objects and individual objects.



* The interface must have the most common denominator and in general has management function for children.

## **Structure**



## **Example**

* See the code attached

# Decorator

## **Intent**

* Attach responsibilities to an object dynamically
* Better alternative than sub classing for extending functionality

## **Motivation**

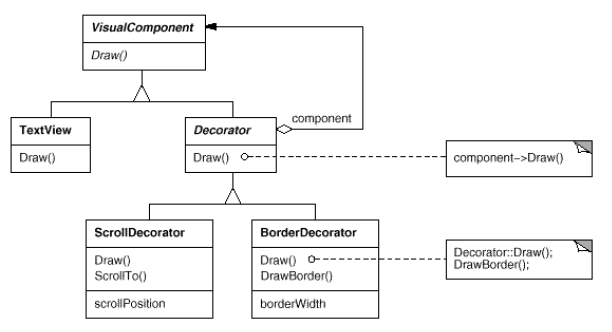
* Suppose that you want additional functionality to an existing component.

E.g.: you have a TextView class and want to add a scrolling behavior or/and a border draw effect. One solution: derive classs like: BorderTextView, ScrollingTextView and BorderAndScrollingTextView.

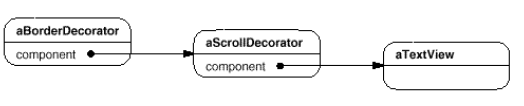
The number of classes explode because of the various combinations and Decorator can help you with that.

* The component you want to attach dynamic responsibilities too can be enclosed in a **decorator** with the same interface as component => Client doesn’t have to make the distinction between component and decorator.

## **Structure**



* Decorator aggregates a component and by default forwards the events and operations to it.
* Our use case:



## **Example**

* See the code attached
* Think about java reading from streams

Eg. **br = new BufferedReader(new FileReader(FILENAME));**

# Proxy

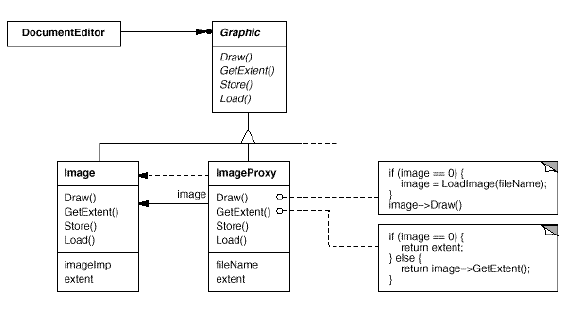
## **Intent**

* Control the access to an object through another class

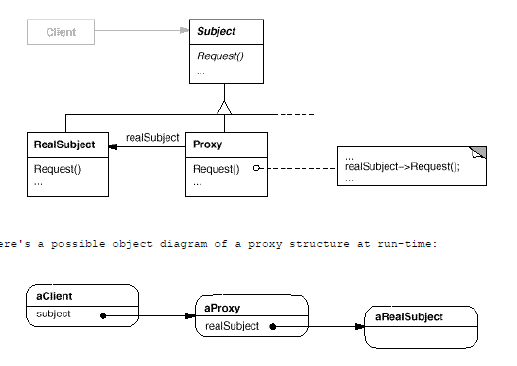
## **Motivation**

* Lazy initialization. Think about Word or Web Browsers. Instead of loading images you can provide a proxy that knows the link (filename) of that object but doesn’t load it right away unless requested explicitly, because it’s expensive to do so. (E.g. what if a word document of 100 pages would load all images in it at the same time? Instead, leave all resources as proxy and when you get to a page containing an image load the resource).
* Smart pointers / references: Hide a pointer and delete it only when ref count is 0.
* Provide local representative for an object in a different address space: for example, think about how SO are handling memory accesses.
* Protection: multithreading, make an object access thread safe if the initial object was not.
* CPU cache is actually a proxy…”copy on write” concept.

The lazy initialization example:



## **Structure**



## **Example**

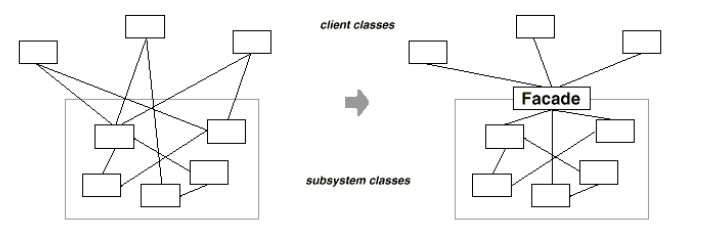
* See the code attached

# Facade

## **Intent**

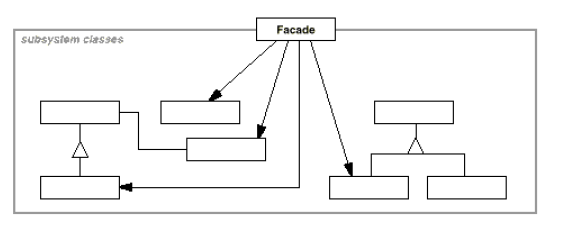
* Provide a unified interface to a set of interface in the system
* Make a collection of subsystems easier to use through a higher-level interface (façade).

## **Motivation**



* You want to provide a simple interface for a complex collection of systems to your clients.
* Hide layers between facades. Example: MVC pattern.
* Decouple as much as possible clients from implementation of subsystem. This way you can modify the subsystems without recompiling or changing client code [why it’s so important ?].

## **Structure**



## **Example**

* See the code attached

# Flyweight

**(quick discussion ; continue next class)**