

Object Oriented Software Design I

Structural Patterns in Java

Giuseppe Lipari
<http://retis.sssup.it/~lipari>

Scuola Superiore Sant'Anna – Pisa

November 14, 2011

Outline

- 1 Adapter pattern
- 2 Composite pattern
- 3 Decorator
- 4 Proxy
- 5 Comparison

Introduction to Structural Patterns

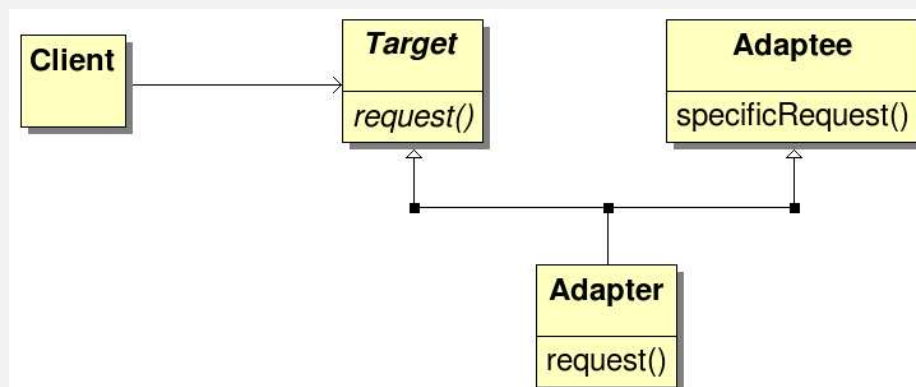
- Structural patterns are concerned with how classes and objects are composed to form larger structures.
- Two kinds of “composition”:
 - static composition through inheritance
 - A class functionality is extended by deriving a new class and overloading polymorphic methods, or by adding new methods
 - A class can mix the functionality of two classes through multiple inheritance
 - A class can implement multiple interfaces through interface inheritance
 - dynamic composition through associations
 - an object holds a pointer to another object
 - an object *delegates* part of its functionality to another object
 - an object can be composed by several smaller objects in many different ways

Motivation

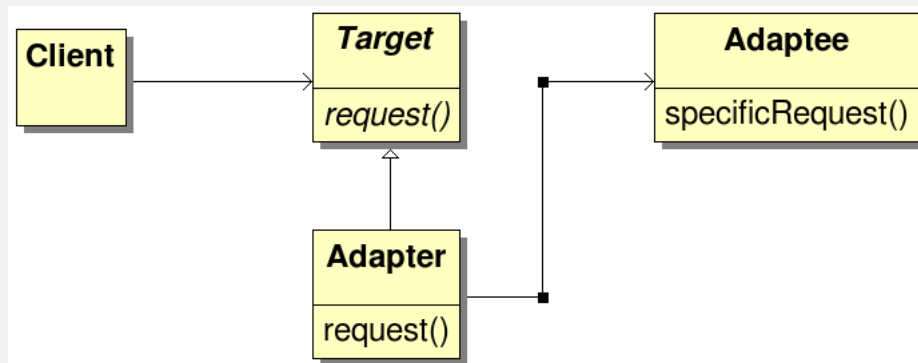
- We have implemented a toolkit to draw graphics objects
 - the base class is *Graphic*, then we have *CompositeGraphic*, and *Line*, *Polygon*, etc.
 - (see the *Composite* example)
- We would like to add a *TextBox*, but it is difficult to implement
- So we decide to buy an off-the-shelf library that implements *TextBox*
- However, *TextBox* does not derive from *Graphic*.
 - and we don't have their source code
- How to include it in our framework?

- We have two options
 - Implement a TextShape class that derives from Graphic and from TextBox, reimplementing part of the interface
 - Implement a TextShape class that derives from Graphic, and holds an object of type TextBox inside
- These two represent the two options for the *Adapter* pattern

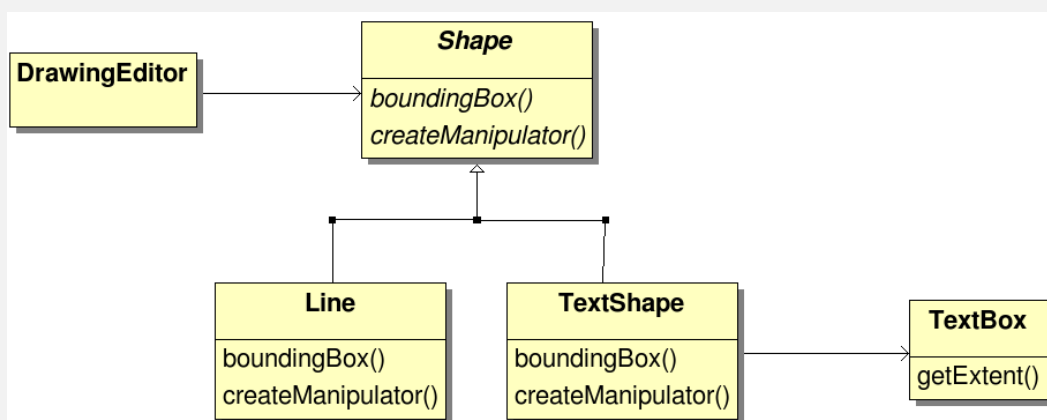
Multiple Inheritance option



Composition Option



Example



Another example

- Suppose we are given the class:

```
class DocManager {  
    ...  
    public void printDocument();  
    public void saveDocument();  
    ...  
};
```

- which is responsible for opening, closing, saving, and printing text documents
- Now suppose we want to build a word-processing application:
 - Graphical user interface that allows, among other things, users to save the file by clicking a Save button and to print the file by clicking a Print button.
 - Details of saving and printing are handled by the class DocManager
- **Key problem:** How to design class Button so that its instances can collaborate with instances of class DocManager

Example – cont.



- Observe: Two instances and one link:
 - “print button” object of class Button
 - “document manager” object of class DocManager
 - Button press should invoke printDocument() operation
- **Question:** How might one design classes Button and DocManager to support this link?

```
class Button {  
    protected DocManager target;  
    protected void monitorMouse() {  
        ...  
        if (/* mouse click */) {  
            target.printDocument();  
        }  
        ...  
    }  
    ...  
}
```

- Why this is a bad design?

Comments

- Observe: To invoke `printDocument()` operation, `Button` needed to know the class (`DocManager`) of the target object.
- However:
 - `Button` should not care that target is a `DocManager`
 - Not requesting information from target
 - More like sending a message to say: "Hey, I've been pressed!. Go do something!"
- **Question:** How can we design `Button` to send messages to objects of arbitrary classes?

Use interface

```
interface ButtonListener {  
    void buttonPressed(String s) ;  
}
```

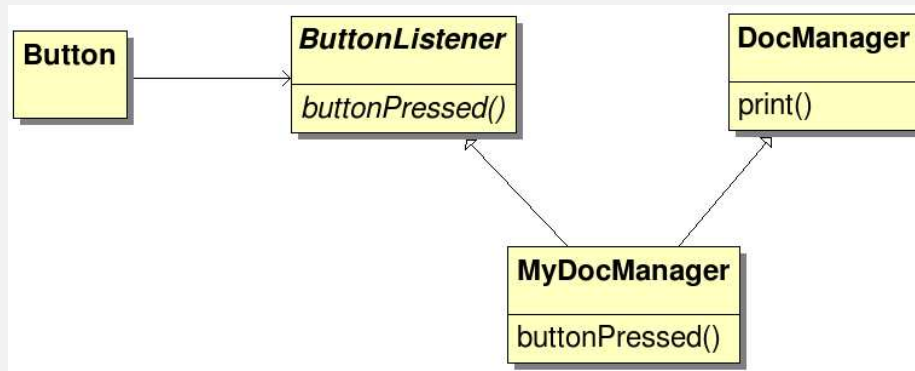
- This interface declares the messages that an object must be able to handle in order to collaborate with a `Button`

A better design

```
class Button {  
    public Button(String lab) {  
        label = lab; target = null;  
    }  
    public void setListener(ButtonListener blis) {  
        target=blis;  
    }  
    private String label;  
    private ButtonListener target;  
  
    public void monitorMouse() {  
        ...  
        if(/* mouse click */) {  
            if (target != null)  
                target.buttonPressed(label);  
        }  
        ...  
    }  
}
```

Collaborator

- *Collaborator*: any object that implements the `ButtonListener` interface
- Collaborator must register interest in press events
- What if `DocManager` is already written and tested and does not implements interface `ButtonListener`?
- Use the adapter interface



Advantages

- Class `Button` very reusable
- We can understand the `Button`–`ButtonListener` collaboration with little knowledge of `Button` and no knowledge of `DocManager`
 - Example of *separation of concerns*
- Clear mechanism for adapting arbitrary class to implement the `ButtonListener` interface
- Rather than thinking of a program as a sequence of steps that compute a function We think of a program as comprising two parts: initialization and then interaction
 - Initialization involves the allocation of objects and the assembly of objects into collaborations;
 - Interaction involves the interaction of objects with other objects by sending messages back and forth

Which structure to use?

- Interface inheritance:
 - Expose interfaces of the adaptee to clients
 - Allow the adapter to be used in places where an adaptee is expected
- Composition
 - Does not expose interfaces of the adaptee to clients (safer)
 - In places that an adaptee is expected, the adapter needs to provide a function `getAdaptee()` to return the pointer to the adaptee object.

Composite pattern

- We must write a complex program that has to treat several object in a hierarchical way
 - Objects are composed together to create more complex objects
 - For example, a painting program treats shapes, that can be composed of more simple shapes (lines, squares, triangles, etc.)
 - Composite objects must be treated like simple ones
 - Another example: a word processor, which allows the user to compose a page consisting of letters, figures, and compositions of more elementary objects
- Requirements:
 - Treat simple and complex objects uniformly in code – move, erase, rotate and set color work on all
 - Some composite objects are defined statically (wheels), while others dynamically (user selection)
 - Composite objects can be made of other composite objects

Solution

- All simple objects inherit from a common interface, say Graphic:

```
interface Graphic {  
    void move(int x, int y);  
    void setColor(Color c);  
    void rotate(double angle);  
};
```

- The classes Line, Circle and others inherit Graphic and add specific features (radius, length, etc.)

```
class CompositeGraphic : extends List<Graphic>, implements Graphic  
{  
    ...  
    public void rotate(double angle) {  
        for (int i=0; i < count(); i++)  
            item(i).rotate();  
    }  
    ...  
}
```

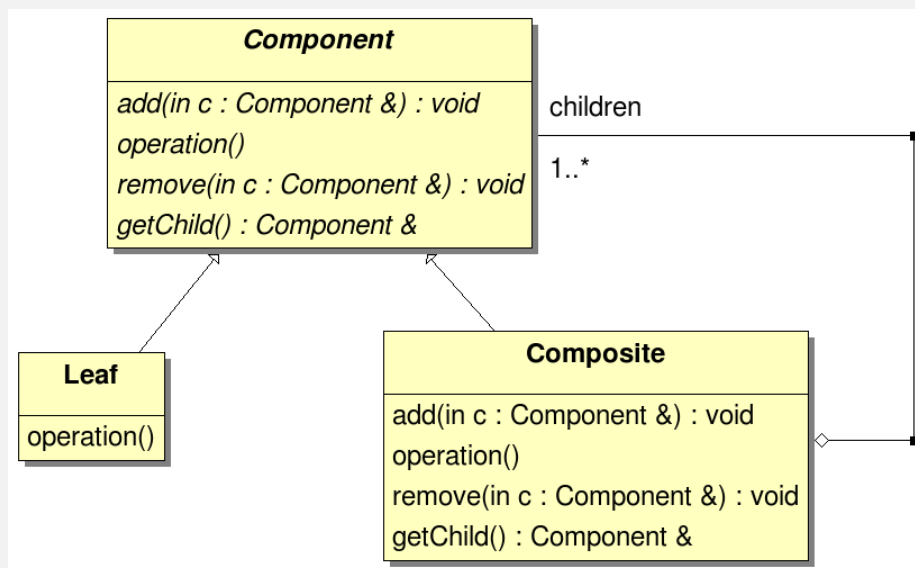
The solution – II

- Since a CompositeGraphic is a list, it had add(), remove() and count() methods
- Since it is also a Graphic, it has rotate(), move() and setColor() too
- Such operations on a composite object work using a “forall” loop
- Works even when a composite holds other composites – results in a tree-like data structure

- Example of creating a composite

```
CompositeGraphic cg new CompositeGraphic();  
cg.add(new Line(0,0,100,100));  
cg.add(new Circle(50,50,100));  
cg.add(t); // dynamic text graphic  
cg.remove(2);
```

UML representation



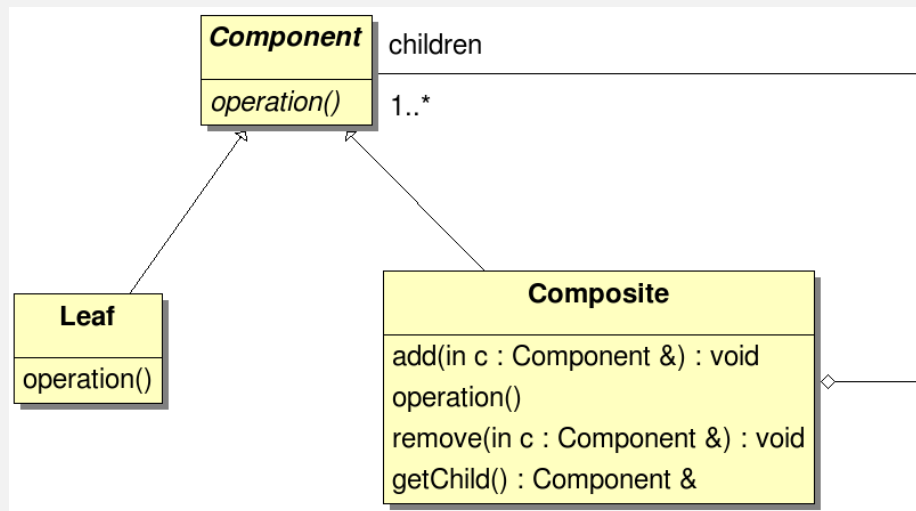
Participants

- Component (Graphic)
 - declares the interface for objects in the composition
 - implements default behaviour for the interface common to all classes, as appropriate
 - declares an interface for accessing and managing its child components
 - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate
- Leaf (Rectangle, Line, Text, etc.)
 - represents leaf objects in the composition. A leaf has no children
 - defines behaviour for primitive objects in the composition
- Composite (Picture)
 - defines behaviour for components having children
 - stores child components
 - implements child-related operations in the Component interface
- Client
 - manipulates objects in the composition through the Component interface

Trade-off between transparency and safety

- Although the Composite class implements the Add and Remove operations for managing children, an important issue in the Composite pattern is which classes declare these operations in the Composite class hierarchy
- For transparency, define child management code at the root of the hierarchy. Thus, you can treat all components uniformly. It costs you safety, however, because clients may try to do meaningless things like add and remove objects from leaves
- For safety, define child management in the Composite class. Thus, any attempt to add or remove objects from leaves will be caught at compile-time. But you lose transparency, because leaves and composites have different interfaces

Composite for safety



Known uses

- Document editing programs
- GUI (a form is a composite widget)
- Compiler parse trees (a function is composed of simpler statements or function calls, same for modules)
- Financial assets can be simple (stocks, options) or a composite portfolio

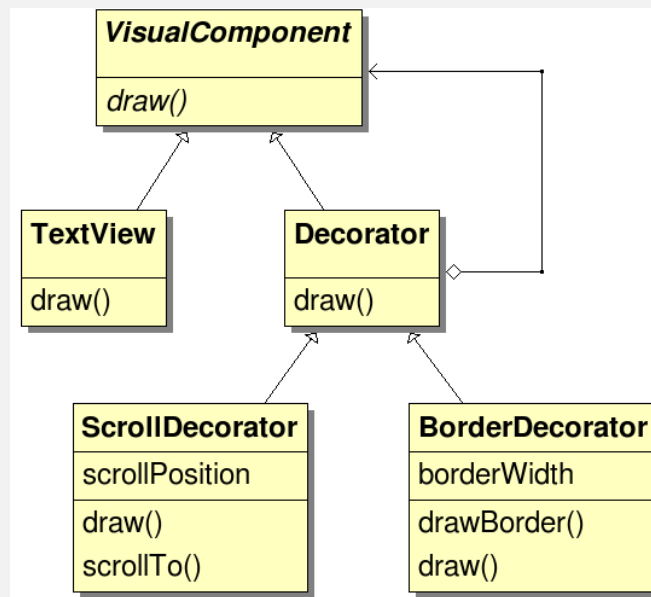
Motivation

- Sometimes we need to add responsibilities to individual objects (rather than to entire classes)
 - For example, in a GUI, add borders, or additional behaviours like scrollbar, to a widget
- *Inheritance approach*
 - We could let the class `Widget` inherit from a `Border` class
 - Not flexible, because for any additional responsibility, we must add additional inheritance
 - Not flexible because is static: the client cannot control dynamically whether to add or not add the border
- *Composition approach*
 - Enclose the component into another object which additionally implements the new responsibility (i.e. draws the border)
 - More flexible, can be done dynamically
 - The enclosing object is called **decorator** (or wrapper)

Example

- A `TextView` object which displays text in a window
- `TextView` class has no scrollbars by default
- we add a `ScrollDecorator` to attach scrollbars to a `TextView` object
- What if we also want a border?
- Well, let's use a `BorderDecorator`!

UML structure for Decorator



Description

- **VisualComponent** is the the abstract class all components must comply to
- From it, many different component classes derive with specific behaviours
- One of these classes is the **Decorator** which does nothing but *wrapping* another component and forwarding all requests to it
 - In the previous example, the **Decorator** class must forward all requests of `draw()` to its component:

```
class Decorator {
    ...
    public draw() {
        component.draw();
    }
    ...
}
```

Decorators internals

- All *decorators* classes must derive from `Decorator`, implementing the additional interface (`scrollTo()`, `drawBorder()`, etc.)
- The `draw()` function is overridden to add the additional behavior

```
class BorderDecorator extends Decorator {  
    ...  
    public draw() {  
        super.draw();  
        drawBorder();  
    }  
}
```

- what the client will see will be the interface of the `VisualComponent`, plus the interface of the most external decorator
- The key property is the any decorator implements the same interface of any other component
- therefore, from the client point of view, it does not matter if an object is decorated or not

Notes

- When to use
 - To add responsibilities to individual object dynamically and transparently
 - When extension by sub-classing is impractical
- Don't use it the component class has large interface
 - Implementation becomes too heavy (all methods must be re-implemented by forwarding to the internal component)
- When there are many "decoration options" you may end up with a lot of small classes
 - The library may become more complex to understand and to use due to the large number of classes
- Where it is used
 - In the Java IO library, the most basic classes perform raw input/output
 - more sophisticated I/O though decorators (i.e. `BufferedInputStream`, `LineNumberInputStream`, are all decorators of `InputStream`)

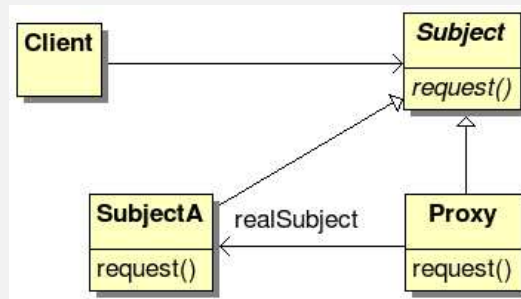
Provide a surrogate or placeholder for another object to control access to it

- Some times it is impossible, or too expensive, to directly access an object
 - Maybe the object is physically on another PC (in distributed systems)
 - In other cases, the object is too expensive to create, so it may be created on demand
- In such cases, we could use a placeholder for the object, that is called *Proxy*

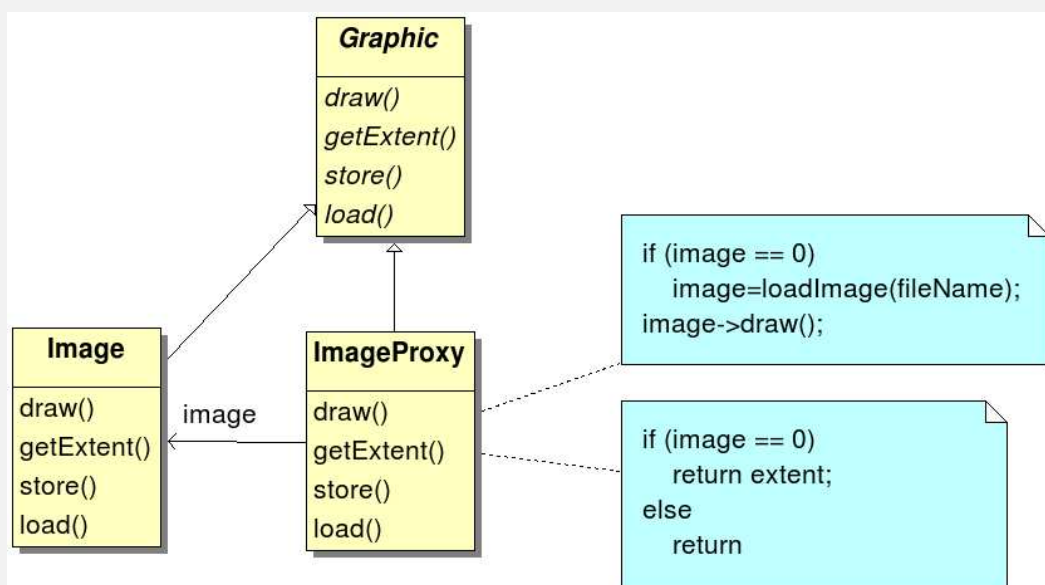
Example

- Suppose that a text document can embed images
- Some of these can be very large, and so expensive to create and visualize
- on the other hand the user wants to load the document as soon as possible
- the solution could be to use a “similar object” that has the same methods as the original image objects
- the object will start loading and visualizing the image only when necessary (for example, on request for `draw()`), and in the meanwhile could draw a simple white box in its placeholder
- it can also return the size of the image (height and width) that it is what is needed to format the rest of the document

The structure



Generalization



- **Proxy** (ImageProxy)
 - maintains a reference that lets the proxy access the real subject. Its interface must be the same as the one of the *RealSubject*.
 - provides an interface identical to *Subject*'s so that a proxy can be substituted for the real subject
 - controls access to the real subject and may be responsible for creating and deleting it
 - other responsibilities depends on the kind of proxy
- **Subject** (Graphic)
 - defines the common interface for *RealSubject* and *Proxy* so that a proxy can be used anywhere a *RealSubject* is expected
- **RealSubject** (Image)
 - defines the real object that the proxy represents

Applicability

- Proxy is applicable whenever there is the need for a more sophisticated reference to an object than a simple pointer
 - A **remote proxy** provides a local representative for an object in a different address space
 - a **virtual proxy** creates an expensive object on demand (like in the example)
 - a **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights
 - A **smart reference** is a replacement for a bare pointer (see *share_ptr* or *auto_ptr*)
 - implement **copy-on-write** behaviour when copying large objects

Comparison of structural patterns

- Many of the patterns we have seen have very similar class diagrams
 - Therefore, the structure is quite similar
 - the basic idea is to favour composition for extending functionality, and use inheritance to make interfaces uniform, and this is why they look similar
 - however, the intent of these patterns is different
- **Adapter** versus **Bridge**
 - Both promote flexibility by providing a level of indirection to another object
 - both involve forwarding requests to this object from an interface other than its own
 - the different is on their *intent*
 - **Adapter** tries to adapt an existing interface to work with other classes; it is used *after* the class is available
 - **Bridge** provides a bridge to a potentially large number of implementations; it is used *before* the classes are available

Composite / Decorator / Proxy

- **Composite** and **Decorator** have similar recursive structure
 - Composite is used to group objects and treat them as one single object
 - Decorator is used to add functionality to classes without using inheritance
 - These intents are complementary
 - therefore, these patterns are sometimes used in concert
- **Decorator** and **Proxy** are also similar
 - They both provide the same interface as the original object
 - However, Decorator is used to add functionalities; Decorators can be wrapped one inside the other
 - **Proxy** is used to control access