Foundations of Software Engineering

Structural Design Patterns

Fall 2017
Department of Computer Science Ben-Gurion university

Based on slides of:

Nurit Gal-oz, Department of Computer Science Ben-Gurion University
Mira Balaban Department of Computer Science Ben-Gurion university
F. Tip. IBM T J Watson Research Center.
Structural Patterns

- concerned with how classes and objects are composed to form larger structures
  - **structural class patterns**: use inheritance to compose interfaces or implementations.
  - **structural object patterns**: describe ways to compose objects to realize new functionality.
- structural patterns:
  - Adapter
  - Composite
  - Proxy
  - Flyweight
  - Façade
  - Bridge
  - Decorator
Adapter
Adapter – Motivation

- Build a *drawing editor* for manipulation of graphical objects like lines, polygons, text.
  - A graphical object has an editable shape.
  - A graphical object can draw itself.
- Key abstraction: an abstract class *Shape*.
- Implementation subclasses: A subclass of *Shape* for each kind of graphical object – *Lineshape, PolygonShape, TextShape*.
- *TextShape* implementation is difficult, but there are many off-the-shelf tools for that, e.g., *TextView*.
- **Problem:**
  - Interface of existing tools is different from *Shape*. 
Adapter – Solution

- **Bad solution:** change the *TextView* class so to conform to the *Shape*:
  - *TextView*'s source code is not available.
  - Does not make sense to change *TextView* to adopt domain-specific interfaces.

- **Good solutions:** *TextShape* *adapts* the *TextView* interface to *Shape*'s.
  - Inherit *Shape*'s interface and *TextView*'s implementation.
  - or
  - Compose *TextShape* a *TextView* instance within a *TextShape* and implement in terms of *TextView*'s interface.

- **Two approaches:** Class and object versions of the Adapter pattern.

- *TextShape* is an *adapter*. 
Adapter – Solution (the object version)

- *TextShape* adapts *TextView* to *Shape*:
  - Reuses *TextView*’s implementation for computing a bounding box.
  - Adds a new functionality: CreateManipulator.
Adapter Design Pattern

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

- Wrap an existing class with a new interface.

- Impedance match an old component to a new system.
Adapter: Participants

- **Target**
  - defines the interface that you need to implement
- **Client**
  - collaborates with objects conforming to the *Target* interface
- **Adaptee**
  - defines an existing interface that needs adapting
- **Adapter**
  - adapts the interface of *Adaptee* to the *Target* interface

**Adapter - a.k.a. Wrapper**
Class Adapter: Class Diagram
Object Adapter: Class Diagram
Adapter: intent and context

• converts the interface of a class into another interface that clients expect

• two variations:
  • Class Adapter: uses multiple inheritance
  • Object Adapter: relies on object composition

• use Adapter when:
  • you want to use an existing class, and its interface does not match the one you need
  • (object adapter) you need to use several existing subclasses, but it’s impractical to adapt their interface by subclassing every one.
Adapter: Example

• Suppose we have a *Client* application that uses a *Stack*, with operations *push*, *pop*, *nrElements*.  
• Instead of implementing *Stack* from scratch, we want to use an existing *Vector* class that provides almost the right functionality.  
• **Solution:** Create a *StackAdapter* class  
  • (class adapter) extends *Vector*, implements *Stack* interface.  
  • (object adapter) has pointer to a *Vector*, implements *Stack* interface.
Client and Target Classes

```java
public class Client {
    public static void main(String[] args) {
        Stack s = new StackAdapter();
        s.push("foo");
        s.push("bar");
        s.push("baz");
        System.out.println(s.pop());
        System.out.println(s.pop());
        System.out.println(s.pop());
    }
}

interface Stack {
    public void push(Object o);
    public Object pop();
    public int nrElements();
}
```
class StackAdapter extends Vector implements Stack {
    StackAdapter() {
        super();
    }
    public void push(Object o) {
        insertElementAt(o, size());
    }
    public Object pop() {
        Object o = elementAt(size() - 1);
        removeElementAt(size() - 1);
        return o;
    }
    public int nrElements() {
        return size();
    }
}

Class StackAdapter (class adapter)
Class StackAdapter (object adapter)

class StackAdapter implements Stack {
    StackAdapter(){ _adaptee = new Vector(); }
    public void push(Object o){
        _adaptee.insertElementAt(o, _adaptee.size());
    }
    public Object pop(){
        Object o =
            _adaptee.elementAt(_adaptee.size()-1);
        _adaptee.removeElementAt(_adaptee.size()-1);
        return o;
    }
    public int nrElements(){
        return _adaptee.size();
    }
    private Vector _adaptee;
}
Adapter: Tradeoffs

• class adapters:
  • adapts *Adaptee* to *Target* by **committing** to a specific *Adapter* class; will not work when we want to adapt a class and its subclasses
  • lets *Adapter* override/reuse some of *Adaptee*’s behavior
  • introduces only one object, no additional pointer indirection is needed to get to *Adaptee*

• object adapters:
  • lets a single *Adapter* work with many *Adaptees* – a single adapter for a whole adaptees hierarchy.
  • makes it harder to override *Adaptee* behavior (requires subclassing of *Adaptee*, and making *Adapter* refer to the subclass)
Bridge
Bridge: Motivation

- Implementation of a portable *Window* abstraction in a user interface toolkit.
- several possible platforms: *X Window System, IBM's Presentation Manager (PM)*, …
- Different kinds of windows: *IconWindow, TransientWindow*, …

→ Need to extend *Window* with hierarchies in multiple “dimensions”.
Bridge: Motivation

Regular solution -- Subclassing:

Problems:

- a combinatorial explosion in number of classes
- difficulties in sharing of implementations
- exposure of platform dependencies to clients
Bridge: Solution

```
Window
+drawText()
+drawRect()

IconWindow
+drawBorder()

TransientWindow
+drawCloseBox()

drawRect();
drawText();

drawRect();

XWindowImpl
+devDrawText()
+devDrawLine()

XdrawLine();
XdrawString();

imp.devDrawLine();
imp.devDrawLine();
imp.devDrawLine();
imp.devDrawLine();
```
Bridge: Participants

- **Abstraction**
  - defines the abstraction’s interface
  - maintains a reference to an object of type *Implementor*

- **RefinedAbstraction**
  - extends the interface defined by *Abstraction*

- **Implementor**
  - defines the interface for the implementation classes; doesn’t have to match interface of *Abstraction*

- **ConcreteImplementor**
  - implements the *Implementor* interface and defines its concrete implementation
Bridge: Class Diagram
Bridge: intent and context

- decouple an abstraction from its implementation so that the two can vary independently

- use *Bridge* when:
  - you want to avoid a permanent binding between an abstraction and its implementation
  - both the abstractions and implementations need to be subclassed
  - changes in the implementation should have no impact on clients (no recompilation). Implementation is hidden from clients.
  - you want to avoid a proliferation of classes caused by extension in multiple, orthogonal extensions
  - you want to share an implementation among multiple objects, and hide this fact from the client
In the bridge pattern, there are 2 parts - the first part is the Abstraction, and the second part is the Implementation. The bridge pattern allows the Abstraction and the Implementation to be developed independently, and the client code can access only the Abstraction part without being concerned about the Implementation part.

Bridge Pattern separates the abstraction hierarchy and the implementation hierarchy in two different layers so that change in one hierarchy will not affect the development or functionality of the other hierarchy.
Bridge: Example

*Stack* that lets us select one of several different implementations:

- *linked list Stack*
- *array-based Stack*
Abstraction -- Class Stack

class Stack {
    Stack(String implType) {
        if (implType.equals("array")) {
            _impl = new ArrayBasedStack();
        } else if (implType.equals("linkedlist")) {
            _impl = new LinkedListBasedStack();
        }
    }
    public void push(Object o) { _impl.push(o); }
    public Object pop() { return _impl.pop(); }
    public boolean isEmpty() { return _impl.isEmpty(); }
    public boolean isFull() { return _impl.isFull(); }

    private StackImpl _impl;
}
Implementation - Class StackImpl

interface StackImpl {
    public void push(Object o);
    public Object pop();
    public boolean isEmpty();
    public boolean isFull();
}

Class ArrayBasedStack

class ArrayBasedStack implements StackImpl {
    public void push(Object o) {
        if (!isFull()) {
            _elements[++_size] = o;
        }
    }
    public boolean isEmpty() {
        return (_size == -1);
    }
    public boolean isFull() {
        return (_size == MAX_SIZE - 1);
    }
    public Object pop() {
        if (isEmpty()) {
            return null;
        }
        return _elements[_size--];
    }
    private final int MAX_SIZE = 100;
    private Object[] _elements = new Object[MAX_SIZE];
    private int _size = -1;
}
class LinkedListBasedStack implements StackImpl {

    // use an inner class for linked list nodes
    private class Node {
        Node(Object o) {
            value = o;
            next = null;
            prev = null;
        }
        public Object value;
        public Node next;
        public Node prev;
    }

    public boolean isEmpty() { return _tail == null; }
    public boolean isFull() { return false; }
}
public void push(Object o) {
    if (_tail == null) {
        _tail = new Node(o);
    } else {
        _tail.next = new Node(o);
        _tail.next.prev = _tail;
        _tail = _tail.next;
    }
}

public Object pop() {
    if (isEmpty()) return null;
    Object ret = _tail.value;
    _tail = _tail.prev;
    return ret;
}

private Node _tail;
public class Main {

    public static void main(String[] args) {
        Stack s = new Stack("linkedlist");
        s.push("foo");
        s.push("bar");
        s.push("baz");
        s.push("zip");
        s.push("zap");
        while (!s.isEmpty()){
            System.out.println(s.pop());
        }
    }
}

Detail the abstraction

- Stack
  + push
  + pop
  + isEmpty
  + isFull

- Stack With Swapping
  ...
  + swap

- LinkedListBasedStack

- ArrayBasedStack

- <<StackImpl >>
Bridge vs. Adapter

- *Object Adapter* and *Bridge* lead to code that looks quite similar. However, they serve different purposes:
  - *Adapter* is retrofitted to make *existing* unrelated classes work together.
  - *Bridge* is designed up-front to let the abstraction and the implementation vary independently.
Bridge: Implementation

- Only one Implementer – Abstraction-Implementer separation is still useful when a change in the implementation must not affect existing clients

- Creating the right implementation object: How, when, where to decide on concrete implementation object?
  - Abstraction knows about all concrete implementation classes:
    - Parameterized constructor.
    - Default implementer.
  - An Abstract Factory object is handled to the Abstraction constructor – abstraction is decoupled from all implementer classes.
1. Client will “set” the look and feel
2. LookAndFeelFactory will retrieve the look and feel from this and will create the appropriate factory

client code: Gets type of UI LookAndFeel to create as Parameter or reads from config.

... 
UILookAndFeelController.setLookAndFeel(WINDOW);
LookAndFeelFactory factory = LookAndFeelFactory.createFactory();
Window wWindow = factory.create(WINDOW) as Window;
Button button = factory.create(BUTTON) as Button;
Label label = factory.create(LABEL) as Label;
...
... or better if (LookAndFeelFactory is of Generic type :: create<T>(T)
...
Window wWindow = factory.create<Window>();
Button button = factory.create<button>();
Label label = factory.create<label>();
...

Composite
Composite: Motivation

- Graphics applications build complex diagrams out of simple components.
- Components can be repeatedly grouped to form larger components.
  - There are graphics primitives: *Text*, *Lines*, …
  - Containers for primitives: *Picture*.
- Clients **treat** primitive and container objects **indifferently** -- Distinguishing these objects makes client applications more complex.
Composite: Solution

- Insert an abstract class that represents both primitives and their containers.

- *Picture* objects can compose other *Pictures* recursively.

- *Composite* structure can be a tree or a graph.
Composite: Solution
Composite: Participants

• **Component**
  - declares common interface
  - implements default behavior common interface
  - declares interface for accessing/managing child components and (optional) for accessing parent

• **Leaf**
  - represents leaf objects in the composition
  - defines behavior for primitive objects

• **Composite**
  - defines behavior for components having children
  - stores child components
  - implements child-related operations in Component

• **Client**
  - manipulates objects via the Component interface
Composite: Class Diagram

```
Client  Component
       +operation()
       +add(Component)
       +remove(Component)
       +getChild(int)

Leaf
+operation()

Composite
+operation()
+add(Component)
+remove(Component)
+getChild(int)

forall g in children
g.operation()
```
Composite: Intent and context

- Compose objects into tree (directed graph) structures to represent **part-whole hierarchies**.
- *Composite* lets you treat individual objects and compositions of objects uniformly.
- Apply *Composite* when:
  - you want to model **part-whole** hierarchies of objects
  - you want clients to be able to ignore the difference between compositions of objects and individual objects.
Composite Example: Unix file systems

- Participants: a Node (Component) is a File (Leaf) or a Directory (Composite).
Composite Example: Unix file systems

- Operations: the `find` command can be used to find and print files with a particular name
  - uses auxiliary operation `getAbsolutePath()`.
- usage: `find <directory> -name <pattern>`
  - `find . -name "*.java"` finds all Java source files in the current directory and its subdirectories and prints their absolute name.
- The example is a somewhat simplified version: we will study a method `Node.find(s)` that
  
  \textit{finds all the files whose name contains \texttt{s} as a substring}

\begin{itemize}
  \item
  \end{itemize}
class Node

abstract class Node {
    Node(String n, Directory p) {
        _name = n; _parent = p;
        if (_parent != null) { p.add(this); }
    }
    public String getName() { return _name; }

    public String getAbsolutePathName() { 
        if (_parent != null) {
            return _parent.getAbsolutePathName() + getName();
        }
        return getName();
    }

    public abstract Vector find(String s);
    protected String _name;
    protected Directory _parent;
}
class File

class File extends Node {
    private String _contents;

    File(String n, Directory p, String c) {
        super(n, p);
        _contents = c;
    }

    public Vector find(String s) {
        Vector result = new Vector();
        if (getName().indexOf(s) != -1) {
            // s is found
            result.add(getAbsoluteName());
        }
        return result;
    }
}

class Directory (1)

class Directory extends Node {
  private Vector _children;
  Directory(String n) { this(n, null); }
  Directory(String n, Directory p) {
    super(n, p);
    _children = new Vector();
  }
  public String getAbsoluteName() {
    return super.getAbsoluteName() + "/";
  }
  public void add(Node n) {
    _children.addElement(n);
  }
  ...
}
public Vector find(String s) {
  Vector result = new Vector();
  if (getName().indexOf(s) != -1) {
    result.add(getAbsoluteName());
  }
  for (int t = 0; t < _children.size(); t++) {
    Node child = (Node) _children.elementAt(t);
    result.addAll(child.find(s));
  }
  return result;
}
class Main

public class Main {
    public static void main(String[] args) {
        Directory root = new Directory("");
        File core = new File("core", root, "hello");
        Directory usr = new Directory("usr", root);
        File adm = new File("adm", usr, "there");
        Directory foo = new Directory("barIsMyName", usr);
        File bar1 = new File("bar1", usr, "abcdef");
        File bar2 = new File("xbar2", usr, "abcdef");
        File bar3 = new File("yybarzz3", usr, "abcdef");
        System.out.println(root.find("bar"));
    }
}

output
[/usr/barIsMyName/, /usr/bar1, /usr/xbar2, /usr/yybarzz3]
Composite: Considerations

- composite makes clients more uniform
- composite makes it easy to add new kinds of components

Disadvantages:

- Some operations only make sense for **Leaf** or **Composite** classes, but not for both.
- Cannot restrict a component to have only components of a certain type. Cannot rely on the type system for that.
  - Need run time checks instead.
Composite: Implementation

• Explicit parent reference in *Component*.
  • Invariant: inverse child-parent references.

• Maximize *Component* interface – adds **transparency**.

• Child management operations:
  • Best maintained in *addition/removal* operations of *Component*.
  • Implies a uniform view of *Composite* and *Leaf*: A *Leaf* is a *Composite* with an empty children collection.

• Transparency-safety tradeoff:
  • In *Component*: increased transparency, less safety.
  • In *Composite*: Less transparency, increased safety.

• Sharing components -- for correctness (structure is a directed graph) or efficiency (→ Flyweight).

• Child ordering relevant or not (→ Iterator).
Proxy
Proxy: Motivation

• A document editor that can embed graphical objects in a document.
• Some graphical objects are expensive to create.
• Not all of these objects are visible at the same time.
• Opening a document should be fast.

→ Avoid creating all the expensive objects at once.
→ Create each expensive object on demand.
Proxy: Solution (1)

- Use another object, an image **proxy**, that acts as a stand-in for the real image.
- The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation.
- The image proxy might store information about the real image.

```
\begin{center}
\begin{tikzpicture}
  \node [draw] (aTextDocument) at (0,0) {aTextDocument};
  \node [draw] (anImageProxy) at (3,0) {anImageProxy};
  \node [draw] (anImage) at (6,0) {anImage};
  \draw [->] (aTextDocument) -- node {image} (anImageProxy);
  \draw [->] (anImageProxy) -- node {fileName} (anImage);
  \draw [dashed] (anImage) -- node {data} (anImageProxy);
  \draw [->] (aTextDocument) -- node {in memory} (anImageProxy);
  \draw [dashed] (anImage) -- node {on disk} (anImageProxy);
\end{tikzpicture}
\end{center}
```
Proxy: Solution (2)

- The Document Editor should be unaware of the proxy:

```
if (image == 0) {
    image = loadImage(fileName);
} image->Draw()
```

```
if (image == 0) {
    return extent;
} else {
    return image->GetExtent();
}
```
Proxy: Class Diagram

Client → Subject

Subject
+request()

RealSubject
+request() → realSubject

Proxy
+request()

realSubject.request()
Proxy: Participants

- **Subject**
  - Defines the *common interface* for *RealSubject* and *Proxy* so that *Proxy* can be used anywhere *RealSubject* is used.

- **Proxy**
  - Maintains reference that lets proxy access real subject.
  - Provides an *interface identical to the subject’s*.
  - **Controls access** to the real subject, and may be responsible for creating & deleting it.
  - Other responsibilities depend on the kind of proxy:
    - remote proxies: encoding and transferring request. A local representative.
    - virtual proxies: caching information (like *ImageProxy*)
    - protection proxies: check access permissions

- **RealSubject**
  - Defines the real object represented by the *Proxy*. 
Proxy: Object diagram

- A possible object diagram at run time:
Proxy: Intent and context

• *Proxy* provides a surrogate or placeholder for another object to control access to it

• Apply *Proxy* when:
  • you need a local representative for an object that lives in a different address space (*remote proxy*).
  • you want to avoid the creation of expensive objects until they are really needed (*virtual proxy*).
  • you want to control access to an object (*protection proxy*).
Proxy Example: Symbolic Links

- in Unix, you can create symbolic links to files and directories with the “ln” command.
- syntax: `ln -s <directory> <linkName>`
- after this command, you can access the directory also via the link.
- you can tell the `find` command to follow symbolic links by specifying the `-follow` option.
- we will now extend the *File System* example with symbolic links, implemented using *Proxy*.
class Link (1)

class Link extends Node {

    private Node _realNode;

    Link(String n, Node w, Directory p) {
        super(n, p); _realNode = w;
    }

    public String getAbsoluteName() {
        return super.getAbsoluteName() + "@";
    }

    ...

public Vector find(String s){
    Vector result = new Vector();
    if (getName().indexOf(s) != -1){
        result.add(getAbsoluteName());
    }
    Vector resultsViaLink = _realNode.find(s);
    String realNodePath = _realNode.getAbsoluteName();
    int n = realNodePath.length();
    for (int t=0; t < resultsViaLink.size(); t++){
        String r = (String)resultsViaLink.elementAt(t);
        String rr = super.getAbsoluteName() + "/" +
                    r.substring(n);
        result.add(rr);
    }
    return result;
}
public class Main {
    public static void main(String[] args) {
        Directory root = new Directory("" );
        File core = new File("core", root, "hello");
        Directory usr = new Directory("usr", root);
        File adm = new File("adm", usr, "there");
        Directory foo = new Directory("foo", usr);
        File bar1 = new File("bar1", foo, "abcdef");
        File bar2 = new File("xbar2", foo, "abcdef");
        File bar3 = new File("yybarzz3", foo, "abcdef");
        Link link = new Link("link-to-usr", usr, root);
        Link linkToLink =
            new Link("link-to-link", link, root);
        System.out.println(root.find("bar"));
    }
}
output

[/usr/foo/bar1,
 /usr/foo/xbar2,
 /usr/foo/yybarzz3,
 /link-to-usr/foo/bar1,
 /link-to-usr/foo/xbar2,
 /link-to-usr/foo/yybarzz3,
 /link-to-link/foo/bar1,
 /link-to-link/foo/xbar2,
 /link-to-link/foo/yybarzz3]
Proxy vs. Adapter

- An Adapter provides a different interface to the object it adapts.

- In contrast, a Proxy provides the same interface as its subject.

- However, a Proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be effectively a subset of the subject's.
Other structural patterns

- **Decorator**
  - Attach additional responsibilities to an object dynamically.
  - Basically a “wrapper” around an object with the same interface as the object.

- **Facade**
  - Provide a unified interface to a set of interfaces in a subsystem.

- **Flyweight**
  - Use sharing to support large numbers of fine-grained objects efficiently.
Decorator
Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with
Decorator: Motivation

- A graphical user interface toolkit: enable addition of properties like borders or behaviors like scrolling to any user interface component.
- Bad solution: inheritance (e.g., a border) from another class. Inflexible!

Inflexible!
Decorator: Motivation

- A graphical user interface toolkit: enable addition of properties like borders or behaviors like scrolling to any user interface component.
- Bad solution: inheritance (e.g., a border) from another class. Inflexible!
- **Decorator** solution: Enclose the component in another object — the *Decorator* -- that adds the border.
  - **Transparency**: The decorator conforms to the interface of the component it decorates.
  - The decorator forwards requests to the component and may perform additional actions.
Decorator: Solution

- A `TextView` object that displays text in a window.
- Composing decorators: compose a `TextView` object with `BorderDecorator` and `ScrollDecorator` objects to produce a bordered, scrollable text view:
Decorator: Solution

![UML Diagram]

- `VisualComponent`
  - `Draw()`
  - `component`

- `TextView`
  - `Draw()`

- `Decorator`
  - `Draw()`

- `ScrollDecorator`
  - `Draw()`
  - `ScrollTo()`
  - `scrollPosition`

- `BorderDecorator`
  - `Draw()`
  - `DrawBorder()`
  - `borderWidth`

- `Decorator::Draw(); DrawBorder();`
package decorator;

public interface VisualComponent {
    public void draw();
    public void resize();
}

public class TextView implements VisualComponent {
    @Override
    public void draw() { // implementation of draw
    }
    public void resize() { // implementation of resize
    }
}
Decorator Example

```java
package decorator;
public class Decorator implements VisualComponent {
    /** *
    * private reference to the VisualComponent being decorated
    */
    private VisualComponent privateVisualComponentReference = null;

    public Decorator (VisualComponent VisualComponentRefernce) {
        this.privateVisualComponentReference = VisualComponentRefernce;
    }
    @Override
    public void draw() {
        privateVisualComponentReference.draw();
    }
    public void resize() {
        privateVisualComponentReference.resize();
    }
}
```
Decorator Example

// Concrete Decorator with extended state
public class ScrollDecorator extends Decorator {

    // Additional State
    private Object scrollBarObjectRepresentation = null;

    public ScrollDecorator (VisualComponent VisualComponentRefernce) {
        super(VisualComponent Refernce);
    }

    @Override
    public void draw() {
        // draw scroll bar
        drawScrollBarObject();
        //draw decorated VisualComponent
        super.draw();
    }

    private void drawScrollBarObject() {
        // prepare scroll bar
        scrollBarObjectRepresentation = new Object();
        // drawscrollbar
    }

    ...
}
Decorator Example

```java
public class GUIDriver {
    public static void main(String[] args) {
        // create a new VisualComponent
        VisualComponent vComponent = new TextView();
        vComponent.draw();
        // at some point later - maybe text size becomes larger than the VisualComponent
        // thus the scrolling behavior must be added
        // decorate the exiting VisualComponent
        vComponent = new ScrollDecorator(vComponent);
        // now VisualComponent object has additional behavior / state
        vComponent.draw();
    }
}
```
Decorator: Class diagram

http://www.developerfusion.com/article//rotaroced-htiw-lortnoc-ecirp-cimanyd/84898
Decorator: Participants

- **Component** (*VisualComponent*)
  defines the interface for objects that can have responsibilities added to them dynamically.

- **ConcreteComponent** (*TextView*)
  defines an object to which additional responsibilities can be attached.

- **Decorator**
  maintains a reference to a Component object and defines an interface that conforms to Component's interface.

- **ConcreteDecorator** (*BorderDecorator, ScrollDecorator*)
  adds responsibilities to the component.
Decorator: Intent and context

- **Decorator** forwards requests to its **Component** object. May perform additional operations before and after.

- **Apply** Decorator when:
  - you need to add responsibilities to individual objects dynamically and transparently, without affecting other objects.
  - extension by subclassing is impractical.
Decorator: Considerations

- Responsibilities can be added and removed at run-time by attaching and detaching them.

- Functionality is composed from simple pieces. Functionality is added incrementally with Decorator objects.

- A Decorator and its Component aren't identical.

- Lots of little objects: Easy to customize, hard to learn and debug.
Decorator: Implementation

- A Decorator's interface must conform to the interface of the Component it decorates.

- The abstract Decorator class can be omitted in case of a single added responsibility.

- Component class should be kept lightweight — little data, little functionality (otherwise, use Strategy).
Decorator vs. other structural patterns

- **Adapter:**
  A *Decorator* only **changes an object's responsibilities**, not its interface; an *Adapter* gives an object a completely **new interface**.

- **Composite:**
  A *Decorator* can be viewed as a degenerate *Composite* with only one *Component*. A *Decorator* **adds additional responsibilities** — it is not intended for **object aggregation**.

- **Proxy:**
  Implementation is similar, but *Decorator* has a different purpose. A *Decorator* **adds responsibilities** to an object, whereas a *Proxy* **controls access** to an object.

  Implementation similarity:
  - A *protection Proxy* might be implemented exactly like a decorator.
  - A *remote Proxy* contains an indirect reference to its real subject.
  - A *virtual Proxy* starts with an indirect reference but eventually obtains a direct reference.
Facade
Facade: Motivation

- Structuring a system into subsystems helps reduce complexity.
- A common design goal is to minimize the communication and dependencies between subsystems.
Intent and Context

• **Intent**

  Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

• **Use the Facade pattern when**
  • you want to provide a simple interface to a complex subsystem
  • there are many dependencies between clients and the implementation classes of an abstraction.
  • you want to layer your subsystems.

Use a facade to define an entry point to each subsystem level.
If subsystems are dependent - simplify the by making them communicate with each other solely through their facades.
Façade Example
Facade: participants

- **Facade** (*Compiler*)
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.

- **subsystem classes** (*Scanner, Parser*)
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade; that is, they keep no references to it.
Facade: Class diagram
Flyweight
Flyweight: Motivation

- Document editor: Characters are objects that occur plenty of times.
- Efficiency Problem: Too many character occurrences.

- Solution: *Flyweight* objects: Shared objects that can be used in multiple contexts simultaneously.
  - Intrinsic state is stored in the flyweight;
  - Extrinsic state varies with the flyweight's.
    Client objects pass extrinsic state to the flyweight when it needs it.
Flyweight: Solution

- *Flyweight* operations that may depend on extrinsic state have it passed to them as a parameter.
Flyweight example
Flyweight: Participants

- **Flyweight**
  - declares an interface through which flyweights can receive and act on extrinsic state.

- **ConcreteFlyweight (Character)**
  - implements the *Flyweight* interface and adds storage for intrinsic state, if any.

- **UnsharedConcreteFlyweight (Row, Column)**
  - not all *Flyweight* subclasses need to be shared. A *UnsharedConcreteFlyweight* object can have ConcreteFlyweight objects as children.

- **FlyweightFactory**
  - creates and manages flyweight objects.
  - ensures that flyweights are shared properly.

- **Client**
  - maintains a reference to *flyweight*(s).
  - computes or stores the extrinsic state of *flyweight*(s).
Flyweight: Class diagram

- **Factory**
  - GetFlyweight(key)

- **Flyweight**
  - Operation(extrinsicState)

- **ConcreteFlyweight**
  - Operation(extrinsicState)
  - intrinsicState

- **UnsharedConcreteFlyweight**
  - Operation(extrinsicState)
  - allState

- **Client**
Flyweight: Object diagram
Flyweight: intent and context

- Use sharing to support large numbers of fine-grained objects efficiently.
- Apply *Flyweight* when *all* of the following are true:
  - An application uses a large number of objects.
  - Storage costs are high because of the sheer quantity of objects.
  - Most object state can be made extrinsic.
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
  - The application doesn't depend on object identity. Since *flyweight* objects may be shared, identity tests will return true for conceptually distinct objects.
Flyweight: Considerations

- *Clients* must obtain *ConcreteFlyweight* objects exclusively from the *FlyweightFactory* object to ensure they are shared properly.

- Time/space tradeoffs.

- The *Flyweight* pattern is often combined with the *Composite* pattern to represent a hierarchical structure as a graph with shared leaf nodes.