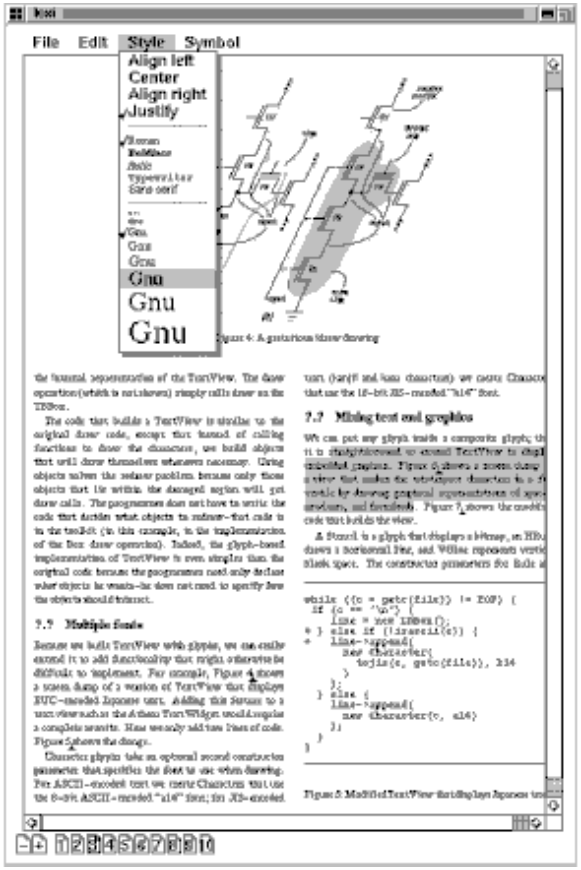
Structural Patterns

Case: Document Editor

Document Editor

Classical example in the GoF book = Lexi (modified)

- Document is mix of graphics and text with formatting styles.
- Surroundings are scrollbars, menus, page icons
- Design problems
  - *Document structure* impacts the design of the rest of the application
  - *Supporting multiple window systems*
  - *Effects and links on items in the document*



Note: we design a document model and its rendering logic, not all interactions

Requirements

- Document structure:
  - Document contains text, images, and drawings
  - Fonts and style settings
- Effects attached to anything
  - Hyperlinks, shadows, backgrounds
- Support multiple window systems
  - portability
  - as independent of the window system as possible
- Document can be processed on demand
  - Able to convert the document to other languages

Document Structure: Req -1.1

- Arrangement of text, lines, images, columns, figures, tables, …
- Should allow manipulations as a group
  - e.g. refer to a table as a whole
  - e.g., treat a diagram as a unit rather than as a collection of individual graphical primitives
  - Simple interface for the client i.e. GUI objects (view)
- Internal representation should support
  - Maintaining the physical structure
  - Generating and presenting the visuals
  - Reverse mapping positions to elements
    - i.e., Given a point, does it intersect with this element?

Document Structure: Req-1.2

- Should treat text and graphics uniformly
  - Avoid treating graphics as special case of text
  - One set of manipulation mechanism should work for both text and graphics
- No distinction between single elements or groups.
  - E.g., the 10^{th} element in line 5 could be a single character, or a complex diagram comprising nested subparts.
  - As long as an element can draw itself and knows its dimensions, its complexity has no effect on how and where it should appear on the page.

Document Structure

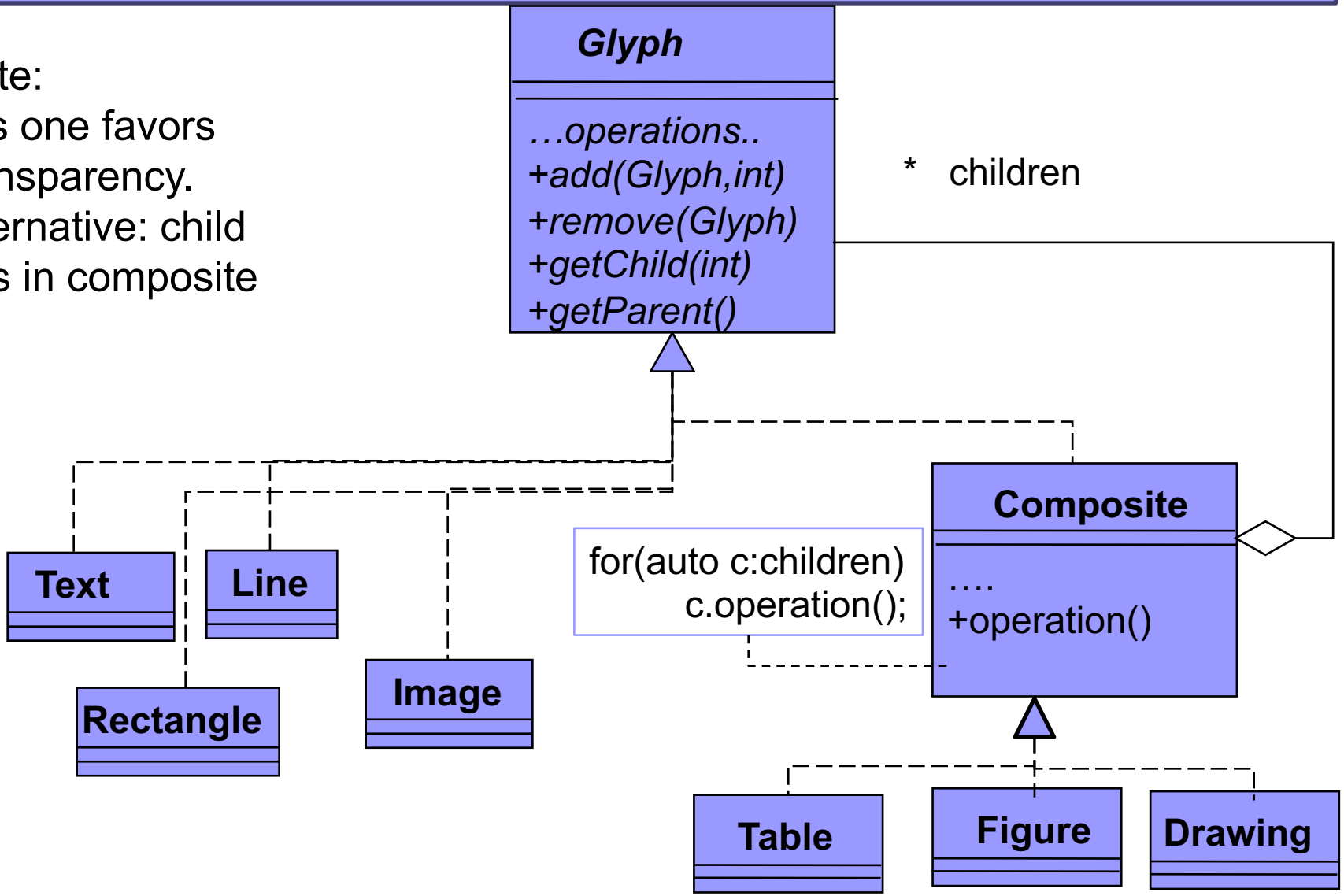
Arrangement of text, lines, images, columns, figures, tables

- Image or Picture
- Text
  - sequence of characters a.k.a letters
  - with different fonts and styles
- Drawing
  - polygons, lines, text
- Figure
  - Image, drawing, text, sub figure
- Table
  - Rows and columns

Let's have a common name: **Glyph**

Document Structure: Composite pattern

Note:
this one favors transparency.
alternative: child ops in composite



Design the Component/Glyph

Requirement

- Internal representation should support
  - Generating and presenting the visuals
  - Reverse mapping positions to elements
  - Maintaining the physical structure

- Responsibility
  - how to draw themselves,
  - what space they occupy, hit detection
  - Structure

Design the Glyph Component

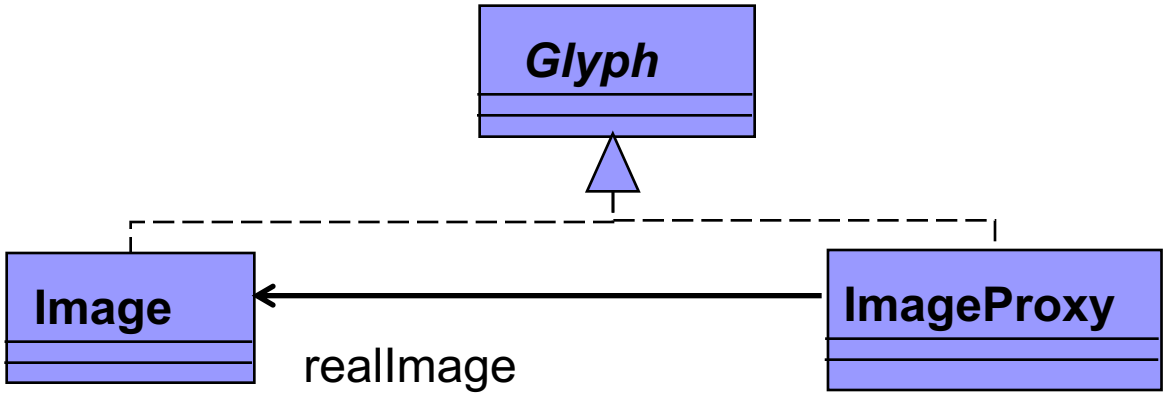
Responsibility	Operations
Appearance	abstract void draw(Window) abstract void bounds(Rect)
Hit detection	abstract boolean intersect(Point)
Structure:	abstract void addChild(Glyph) abstract void remove(Glyph) abstract Glyph getChild(int) Glyph getParent()

Responsibility	Operations
appearance	`virtual void Draw(Window*)` `virtual void Bounds(Rect&)`
hit detection	`virtual bool Intersects(const Point&)`

structure	`virtual void Insert(Glyph*, int)`
	`virtual void Remove(Glyph*)`
	`virtual Glyph* Child(int)`
	`virtual Glyph* Parent()`

Images at startup

- Expensive to create images, opening a document should be fast
- Solution
  - Delay creation and loading of images until they are on demand
    - Load and draw the images that are only in the current page
  - A (virtual) Proxy in place of images in the document



```
class ImageProxy : public Glyph {
  public:
    ImageProxy(const char* fileN);
    virtual ~ImageProxy();

    virtual void Draw(Window * w);
    virtual void Bounds(Rect& r);
    /* ..other methods …*/

  protected:
    Image* GetImage();
  private:
    Image* _image;
    char* _fileName;
  };
```

```
class Image : public Glyph {
 public:
    // loads image from a file
   Image(const char* fileN);
    virtual ~Image();

    virtual void Draw(Window* w);
    virtual void Bounds(Rect& r);
       /*…other methods...*/
};
```

Client code:

```
Glyph doc; //……
doc->Insert(new ImageProxy("anImageFileName"),index);
```

```
class ImageProxy : public Glyph {
  public:
    ImageProxy(const char* fileN);
    virtual ~ImageProxy();

    virtual void Draw(Window * w);
    virtual void Bounds(Rect& r);
    /* ..other methods …*/

  protected:
    Image* GetImage();
  private:
    Image* _image;
    char* _fileName;
  };
```

```
ImageProxy::ImageProxy (const char* fileN)  {
        _fileName = strdup(fileN);
        _image = 0;
}
Image* ImageProxy::GetImage() {
        if (_image == 0) { //do this in a thread
            _image = new Image(_fileName);
        }
        return _image;
}
void ImageProxy::Draw (Window* w) {
        GetImage()->Draw(w);
         //once image is loaded,
        // this is a simple delegation
}
```

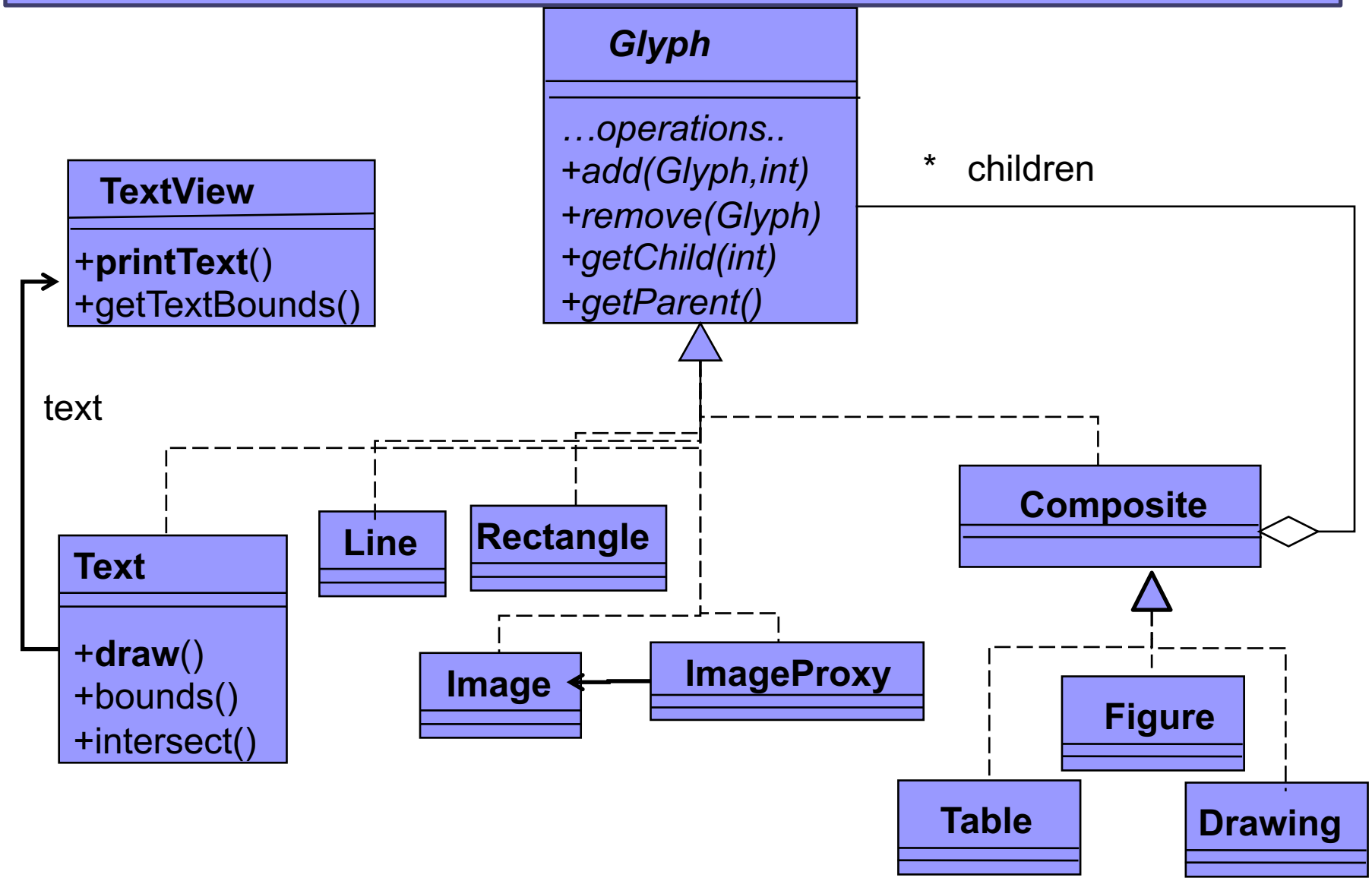
Client code:

```
    Glyph doc; //……
    doc->Insert(new ImageProxy("anImageFileName"),index);
```

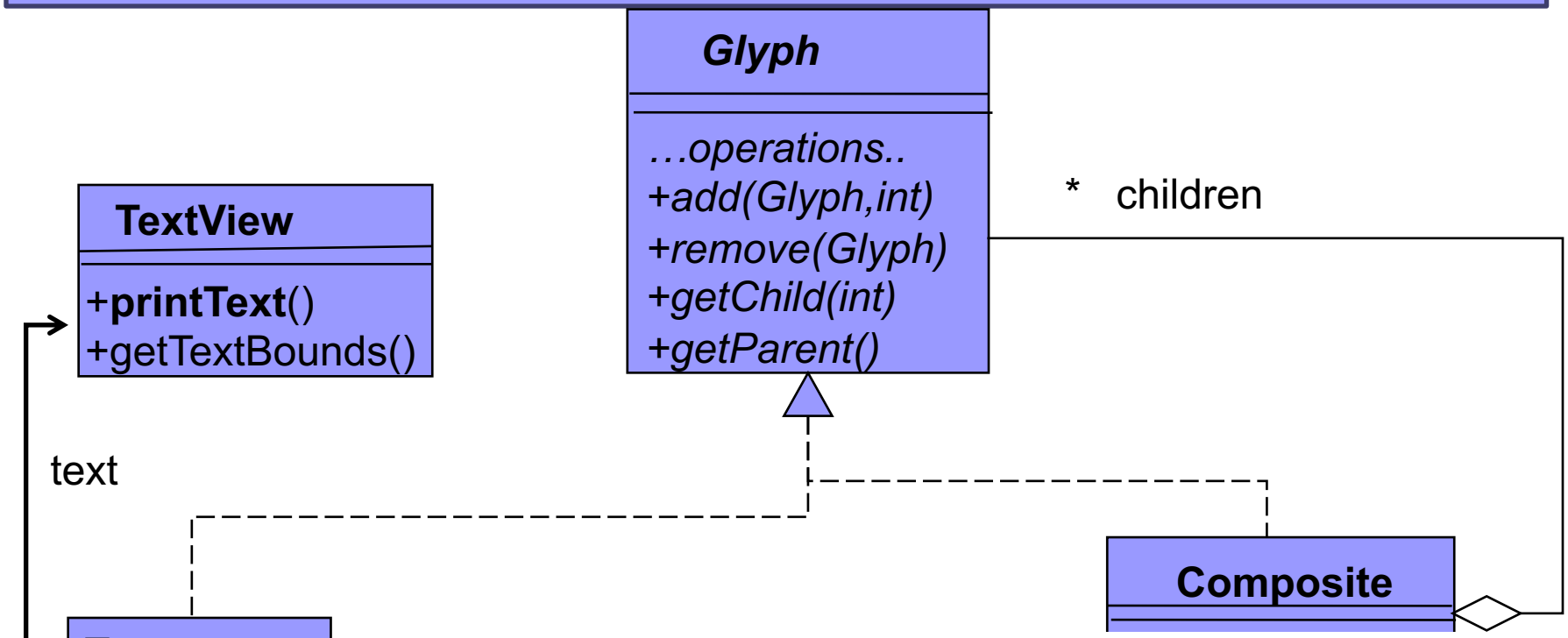
Want to use an existing Text Impl

- There is already a text class (TextView) with fonts associated to each letter
  - Designed earlier or came with a library
- TextView does not support `draw()` and `bounds()`, instead it has `printText()` and `getTextBounds()`
- Solution: Adapter
  - A Text object adapts TextView to Glyph

Adapting TextView



Adapting TextView

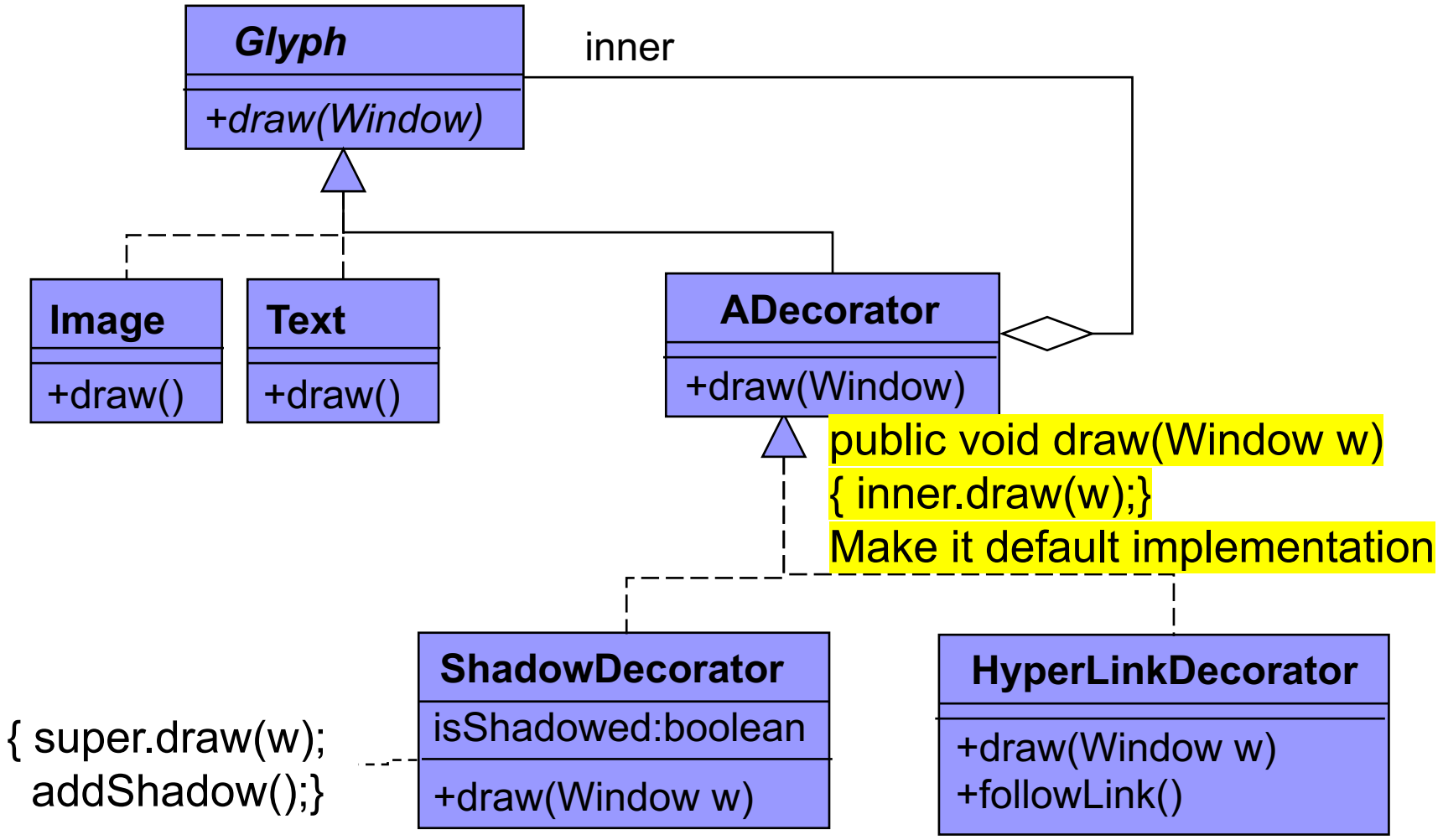



```
// Text class is the Adapter
void Text::draw(Window* w) {
        _adaptee->printText();
    // The adapter forwards the call,
    //translating the interface
}
```

R2:Adding the Effects and Links

- Req: Effects can be attached to anything
  - Any Glyph (text, image, etc.) can have a hyperlink or a shadow
- Hyperlinks, shadows can be added and removed at runtime
  - Inheritance is impractical
- Don’t crowd the Glyph -ISP
- Decorator
  - Hyperlink decorator
  - Shadow decorator

Effects as Decorator



Usage

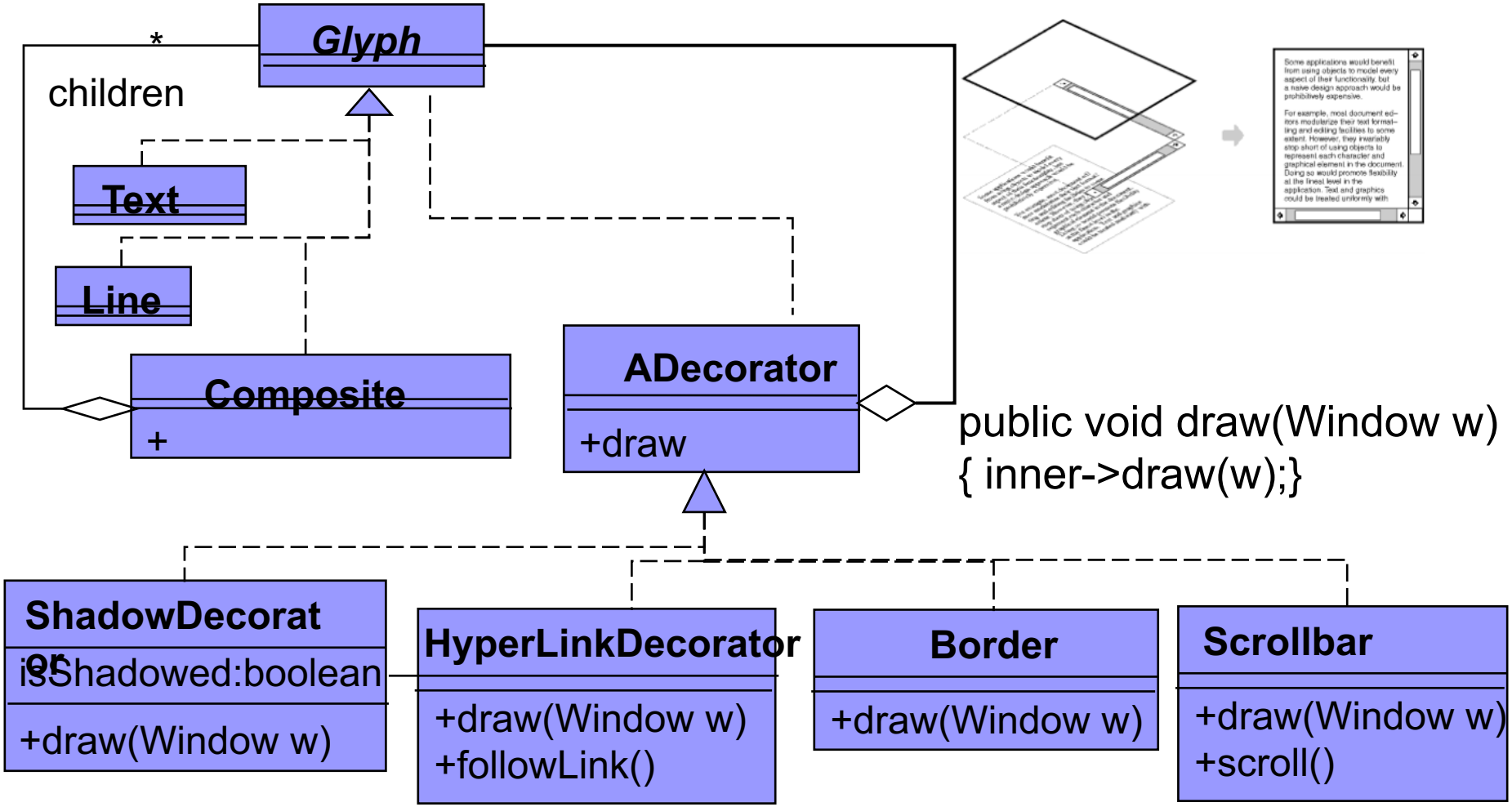
- Create a text with shadow and hyperlink

```
Glyph component=new Text();
component=new ShadowDecorator(component);
  //adding a shadow
component=new HyperlinkDecorator(component);
  //adding a hyperlink
```

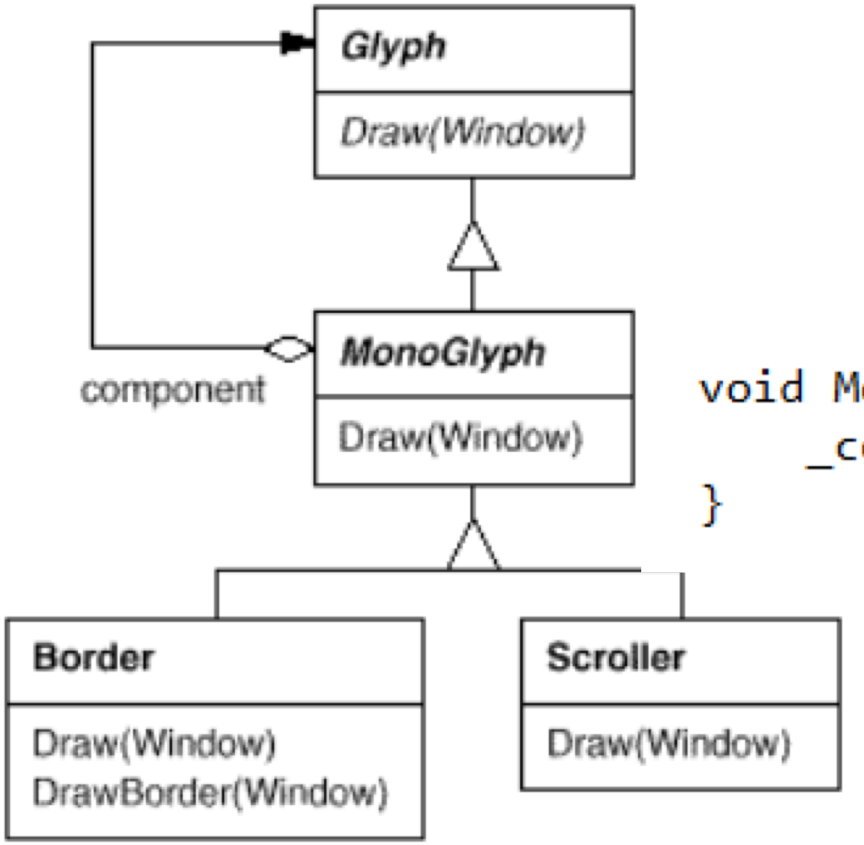
- Now use this `component` in the windowing system. The window only sees it as a Glyph.
- You can directly interact with it and follow the link as well.

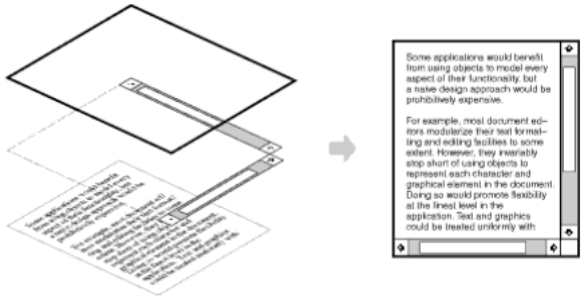
Borders and Scrollbar –GoF ex.

- Add borders to any part: see Decorator lecture



Borders and Scrollbar





```
void MonoGlyph::Draw (Window* w) {
    _component->Draw(w);
}
```

```
void Border::Draw (Window* w) {
    MonoGlyph::Draw(w);
    DrawBorder(w);
}
```

MonoGlyph is the DefaultDecorator

R3: Support Multiple Window Systems

- Window system portability
  - Multiplatform windows sys: Qt, GTK, wxWidgets
  - MS Windows, X Window, Quartz
- We have several class hierarchies from different vendors.
  - highly unlikely these hierarchies are compatible
  - different window systems have incompatible programming interfaces.
- Same problem for multiple DBMS

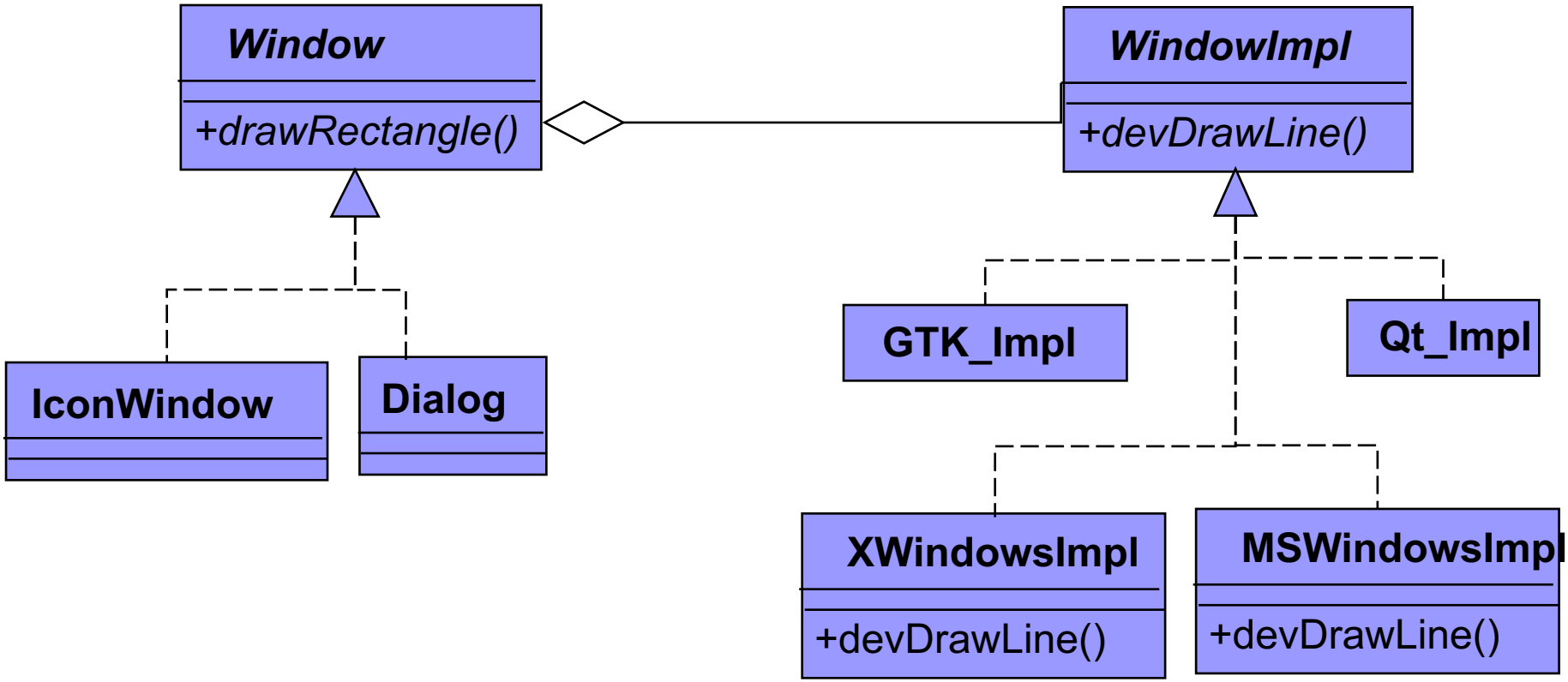
R3: Support Multiple Window Systems

- Window system portability
- We have several class hierarchies from different vendors.
  - Let's say Qt and GTK
- All window systems do the same thing.
  - We want a uniform set of windowing abstraction that lets us use any window system implementation
  - Use any one of them under a common interface

Portable to Window Systems

- Bridge Pattern

```
// Client code that sets up the bridge
Window* myWindow = new ApplicationWindow();
WindowImp* imp = new XWindowImp(); //choose 1
myWindow->setImplementation(imp);
```

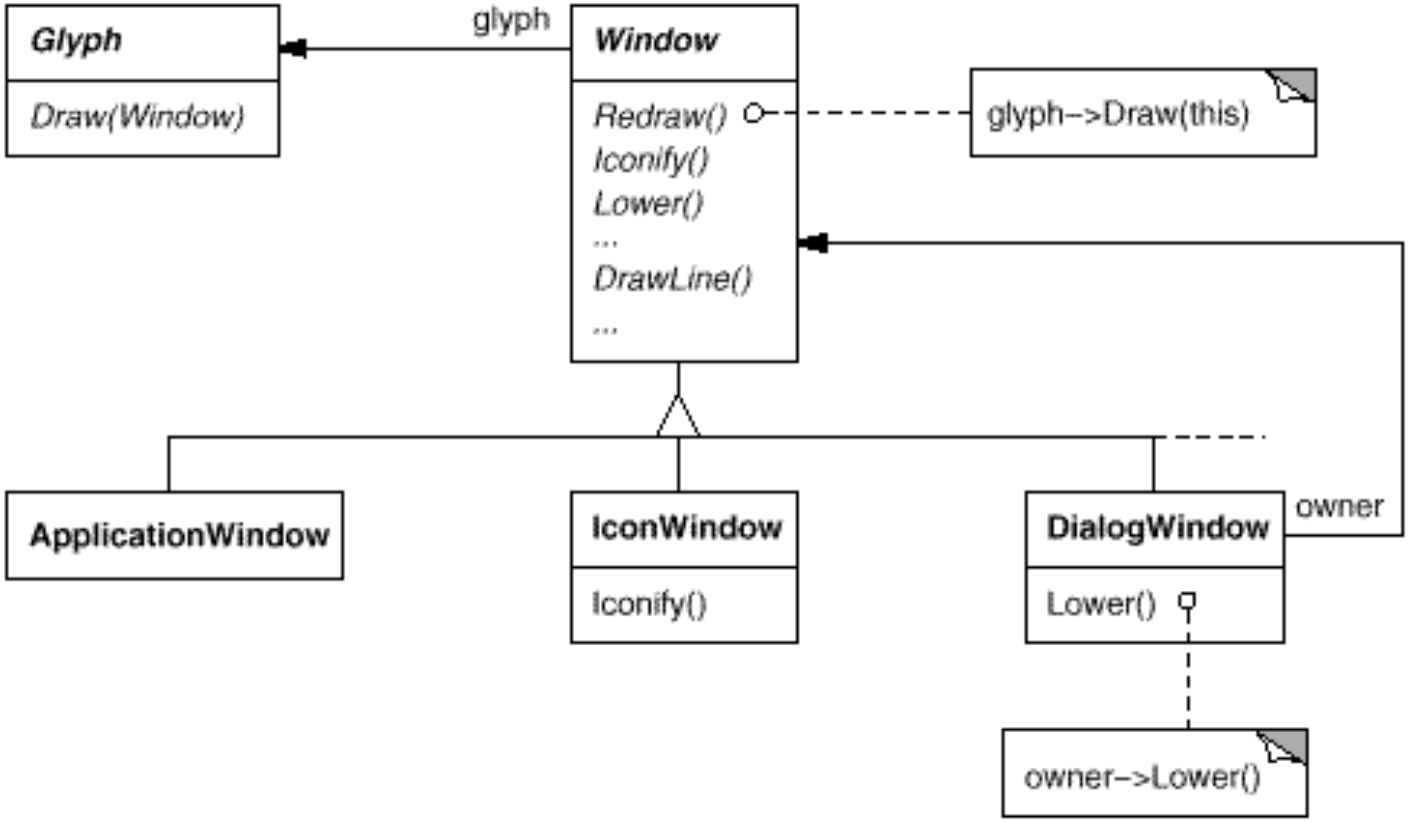


Window abstraction

- Recall Glyph::draw(Window *w)
- Window abstraction
  - provide operations for drawing basic geometric shapes. (graphics)
  - resize themselves. (window management)
  - iconify and de-iconify themselves. (window mng)
  - (re)draw their contents on demand,
    - for example, when they are de-iconified or when an overlapped and obscured portion of their screen space is exposed. (window management)
- Alternatively, we may choose to intersect or to unify functionalities of each windowing system

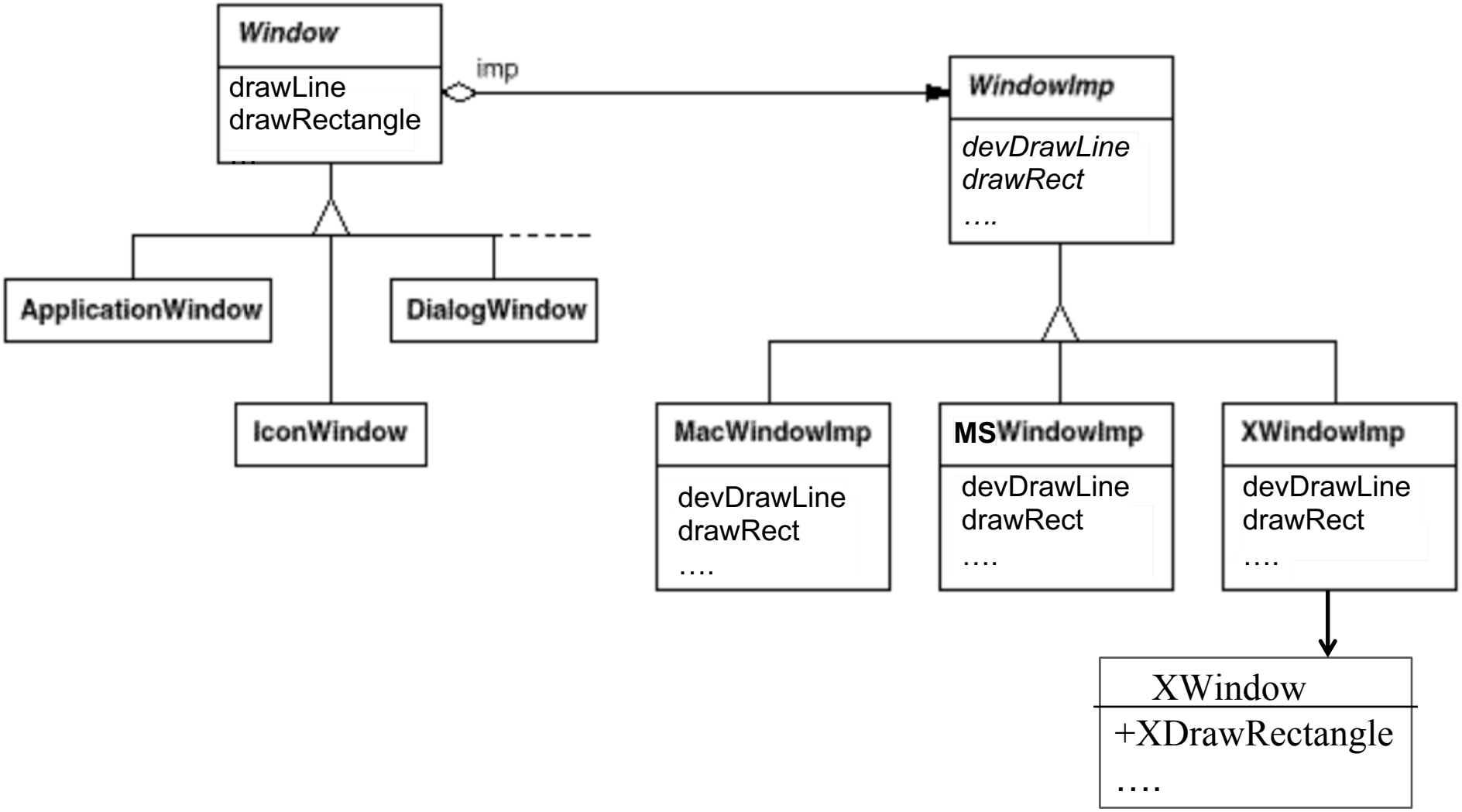
Window Abstraction

Client:



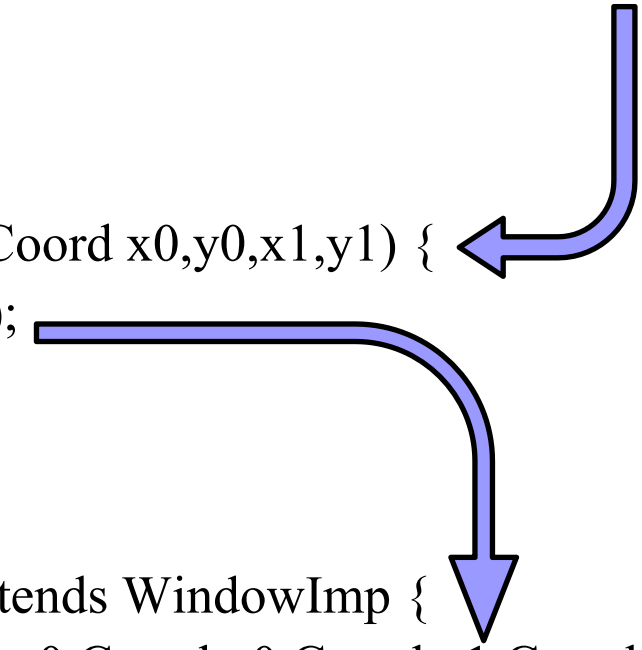
We could specialize Window abstraction as application windows, icon window, and warning dialogs

Bridge Pattern



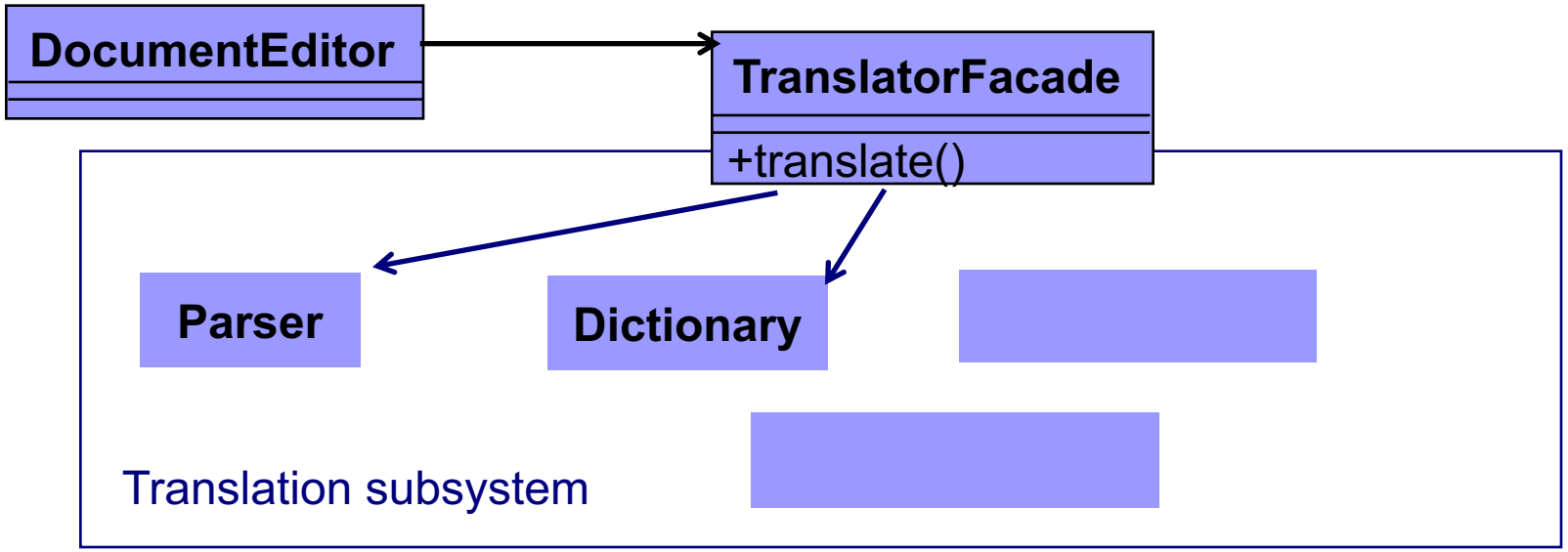
Bridge in Action

```
public class Rectangle extends Glyph {
  public void draw(Window w) { w.drawRectangle(x0,y0,x1,y1); }
  ...
}
public class Window {
  public void drawRectangle(Coord x0,y0,x1,y1) {
    imp.drawRect(x0,y0,x1,y1);
  }
  ...
}
public class XWindowImp extends WindowImp {
  public void drawRect(Coord x0,Coord y0,Coord x1,Coord y1) {
    ...
    XDrawRectangle(display, _winId, graphics, x,y,w,h);
  }
}
```



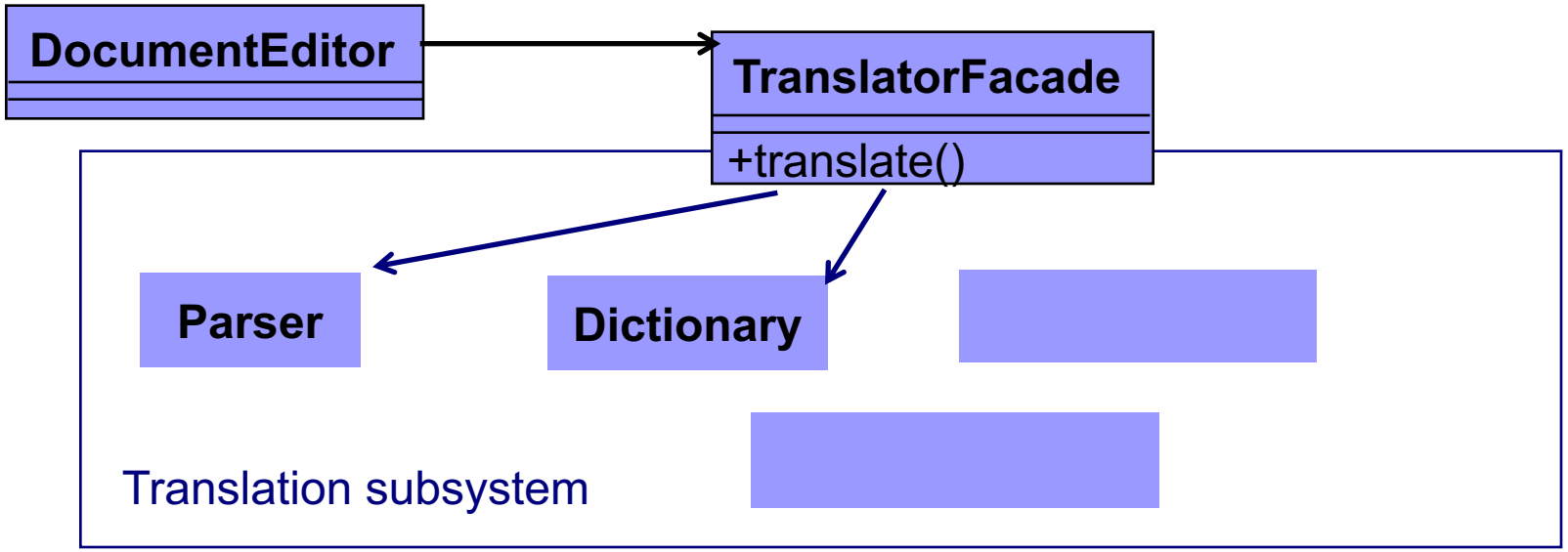
R4: Document Processing

- R4:Process document on demand
- There is have a subsystem that converts a document to other languages
  - Parser, dictionary, grammar rules, etc
- Façade: Translator
  - Have a TranslatorFaçade to provide a unified interface of translation subsystem to the document editor



R4: Document Processing

- Façade: Translator
  - provides a unified interface of translation subsystem to the editor
- Goal: **decouple** the Editor from the complexity of the translation subsystem.
  - we can completely replace the translation subsystem with a different one, and as long as the new one is wrapped in the same Façade, the document editor code would not need to change



Formatting Problem

- **Requirement:** Each character in the document can have its own font, size, style (italic), and color.
- Create a "Style" object for every single character?

- Suppose we do not use an existing the Text manipulation class, TextView anymore.

Formatting Problem

- **Requirement:** Each character in the document can have its own font, size, style (italic), and color.
- Create a "Style" object for every single character?

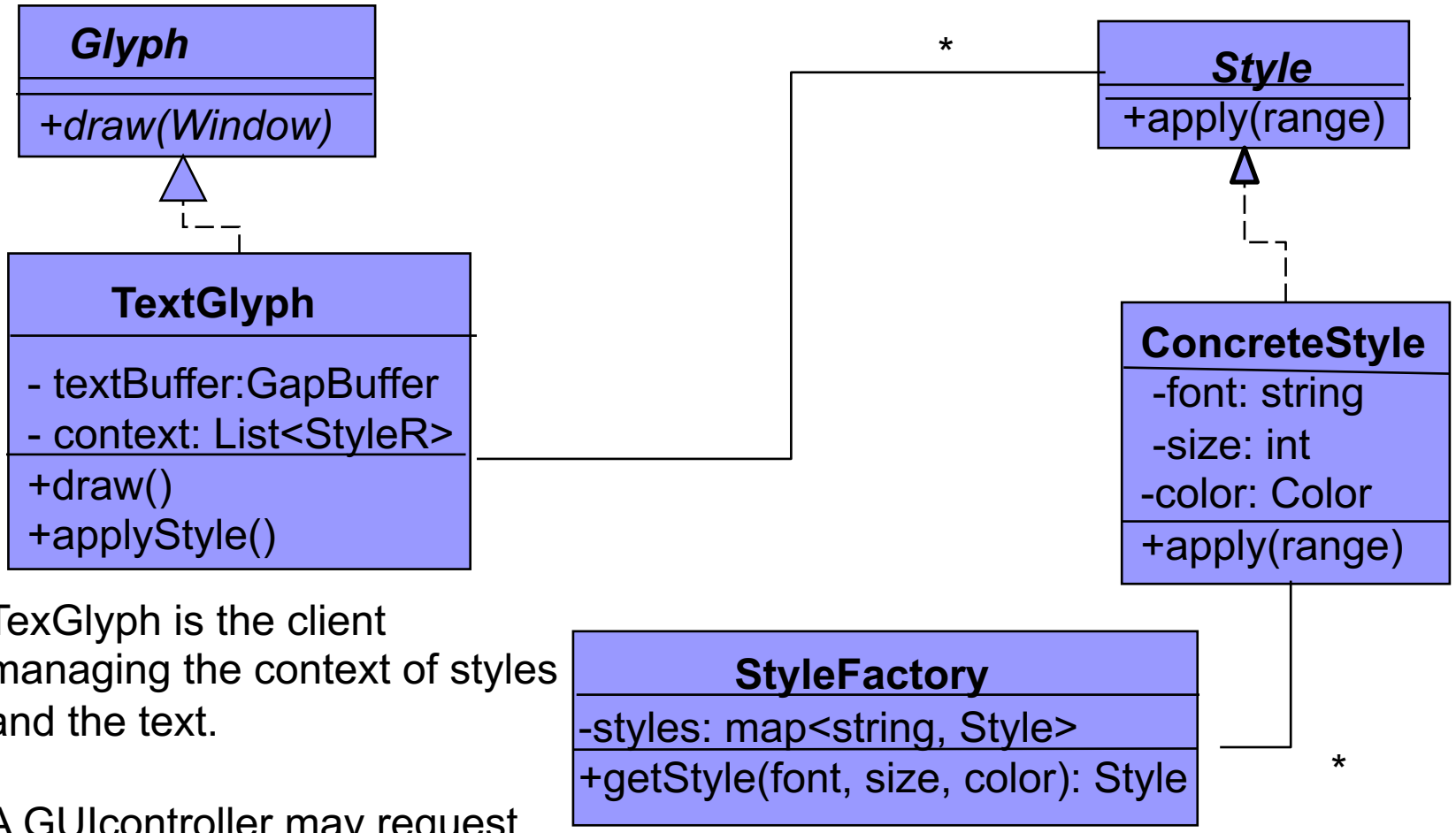
- **Problem:** This is incredibly wasteful.
  - If a paragraph of 500 characters is styled as "Arial 12pt," you would create 500 identical "Arial 12pt" objects.
  - This leads to massive memory consumption.

  - Suppose we do not use an existing the Text manipulation class, TextView anymore.

Flyweight for Formatting

- # of unique formatting combinations is very small
  - share formatting objects. E.g. an object for Arial12pt.
- Flyweight: Style objects.
- Intrinsic State: use definition of the style
  - font name, size, color, italic, bold
  - "Arial", 12, "Bold", "Black"
- Extrinsic State:
  - the range of characters that use this style
  - e.g., from character position 5,328 to 5,828.
  - this Context is managed by the document model

Flyweight



TexGlyph is the client managing the context of styles and the text.

A GUIcontroller may request Styles from the StyleFactory

```
public class TextGlyph extends Glyph {
    private static class StyleR {
        Style style; // The shared flyweight object
        int start; int end;  //extrinsic state
    }
    // context: list of StyleR mapping flyweights to character ranges
    private List<StyleR> styleRuns;

    // The raw character data, stored in a performance-oriented structure
    private GapBuffer textBuffer;

   //see the next slide for the use of context to draw the text.

    public void applyStyle(int start, int end, Style stylefromfactory) {
        // Logic to insert a new style-range for the given range.
         styleRuns.add(new StyleR(stylefromfactory, start, end));
      // A real implementation would need to handle splitting/merging existing runs.
    }
    public TextGlyph() {…} //initialize textBuffer and context
}
```

```
public void draw(Window w) {
        // 1. Iterate through each StyleRun (the extrinsic context)
        for (StyleR  run : styleRuns) {
            // 2. Get the shared Style flyweight for this run
            Style currentStyle = run.style;

            // 3. Iterate through the characters in this run's range
            for (int i = run.start; i < run.end; i++) {
                char currentChar = textBuffer.charAt(i); // Get char from buffer
                Position currentPos = calculatePosition(i);

                // 4. Pass extrinsic state (char, position) to the flyweight's method
                currentStyle.draw(w, currentChar, currentPos);
            }
        }
    }
```

Side note

- editors do not use flyweights for induvial character objects
  - no per-character object
  - previously, I used a Letter class as a flyweight for characters and shared font objects as flyweights
- Word uses **flyweights** for formatting objects (**style**) and keeps them separate from text
- VS Code uses rope for string manipulation

DISCUSSION

Structural Patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures.
- Structural class patterns use *inheritance* to compose classes
  - Adapter Class pattern
- While the Structural object patterns describe ways to *assemble objects*
  - Composite, decorator, flyweight, façade, Adapter object, bridge, proxy

Adapter vs Bridge

- Adapter and Bridge, both promote flexibility by providing a level of indirection to another object.
- Adapter and Bridge, both involve forwarding requests to this object from an interface other than its own.
- Difference in their intents!

Adapter vs Bridge

- Difference in their intents!
- Adapter : making two independently designed classes **work together** without reimplementing them
  - Ignore how those interfaces are implemented, nor does it consider how they might evolve independently.
- Bridge: bridges an abstraction and its **potentially numerous implementations**.
  - It provides a stable interface to clients even as it lets you vary the classes that implement it.
  - It also accommodates new implementations as the system evolves.

Adapter vs Bridge

- Adapter and Bridge are often used at different points in the software lifecycle.
- An adapter often becomes necessary when you discover that two incompatible classes should work together, generally to avoid replicating code. *The coupling is unforeseen.*
- In contrast, the user of a bridge understands *up-front* that an abstraction must have several implementations, and both may evolve independently.
- The Adapter pattern makes things work *after* they're designed; Bridge makes them work *before* they are.

Adapter vs Facade

- Is façade an adapter to a set of other objects?
- This interpretation overlooks that a facade defines a *new* interface
  - whereas an adapter reuses an old interface.
- An adapter makes two *existing* interfaces work together as opposed to defining an *entirely* new one.

Composite vs Decorator

- Decorator and Composite structure are similar but differ in intent.
- Decorator: *add responsibilities* to objects without subclassing.
  - It avoids the explosion of subclasses that can arise from trying to cover every combination of responsibilities statically.
- Composite: structuring classes so that many related objects can be treated *uniformly*, and *multiple objects can be treated as one.*
  - Its focus is not on embellishment but on representation.

Composite vs Decorator

- Composite and Decorator patterns are often used together.
  - Their intents are distinct but complementary.
- From the point of view of the Decorator pattern, a composite is a ConcreteComponent.
- From the point of view of the Composite pattern, a decorator is a Leaf.
- Both lead to a design in which you can build applications just by plugging objects together without defining any new classes.

Decorator vs Proxy

- Both patterns introduce a level of indirection to an object,
- Both proxy and decorator implementations keep a reference to another object to which they forward requests.
- Both patterns compose an object and provides an identical interface to clients.

Difference in Intent

- Unlike Decorator, the Proxy pattern is not concerned with *attaching* or detaching *properties dynamically*,
- Proxy is a *placeholder* for a subject when it's inconvenient or *undesirable to access the subject directly*
  - because, for example, it lives on a remote machine, has restricted access, or is persistent.

Decorator vs Proxy

- Decorator: the Component provides only part of the functionality, and one or more decorators furnish the rest.
- Proxy: the Subject defines the key functionality, and the proxy provides (or refuses) access to it.