Structural Patterns

- They include all the patterns which are concerned with how classes and objects are composed to form larger structures.
- We will study the following ones:

  - **Adapter**
  - **Composite**
  - **Façade**
  - **Decorator**
  - **Proxy**

Adapter Pattern

Motivation

Consider a graphic editor which manages text, lines, polygons, etc. Every object must be displayed, edited and moved. The editor "sees" objects through an interface:

```
interface Shape {
    public void display();
    public void edit();
    // create an object which moves the shape
    public Manipulator createManipulator();
    }
```

Adapter Pattern (2)

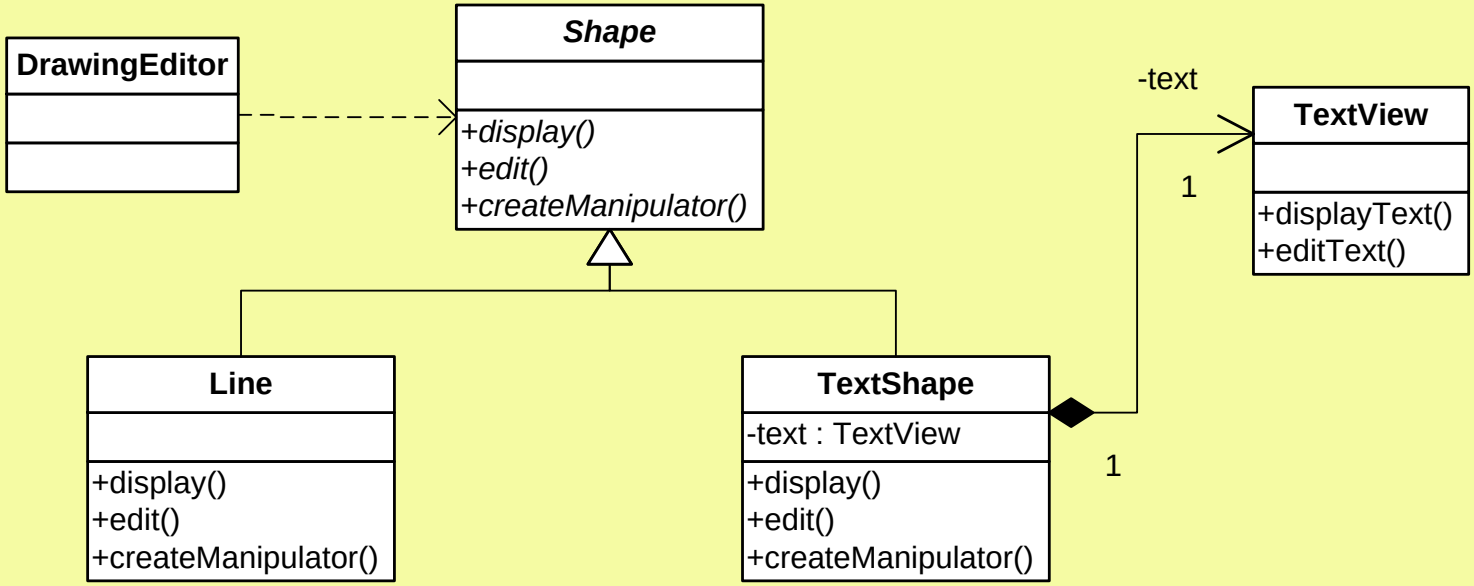
- Suppose an existing toolkit provides a sophisticated TextView class for displaying and editing text through two operations: displayText() and editText().
- We'd like to use TextView to implement TextShape but:
  - toolkit wasn't designed with Shape classes in mind
  - we can't adapt TextView class to conform TextShape interface: we don't have source code and it doesn't make sense anyway
- What we can do: define a TextShape class which implements Shape and uses the TextView class

Adapter Pattern (3)

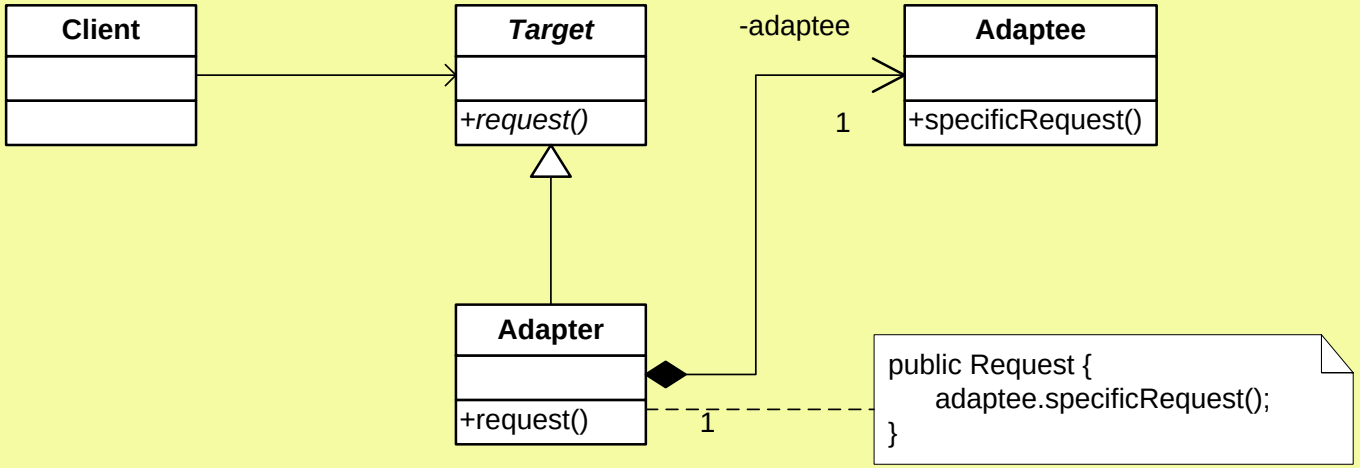
Intent

- Converts the interface of an existing class into another interface adapted to the user's needs.
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Adapter Pattern (4)



Adapter Pattern (5)

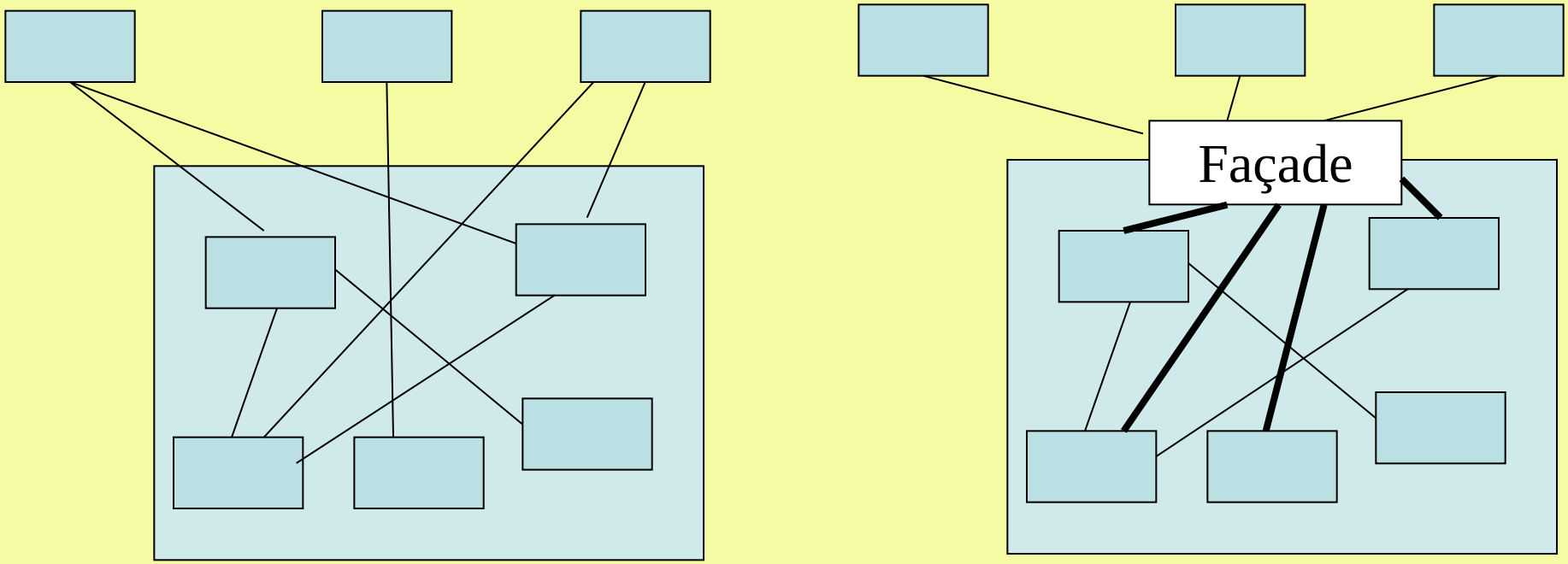


- **Target** (Shape): defines the interface that Client uses.
- **Client** (DrawingEditor): collaborates with objects conforming to the Target interface
- **Adaptee** (TextView): defines an existing interface that needs adapting
- **Adapter** (TextShape): adapts the interface of Adaptee to the Target interface

Façade Pattern

Motivation

Structuring a system into subsystems helps reduce complexity.

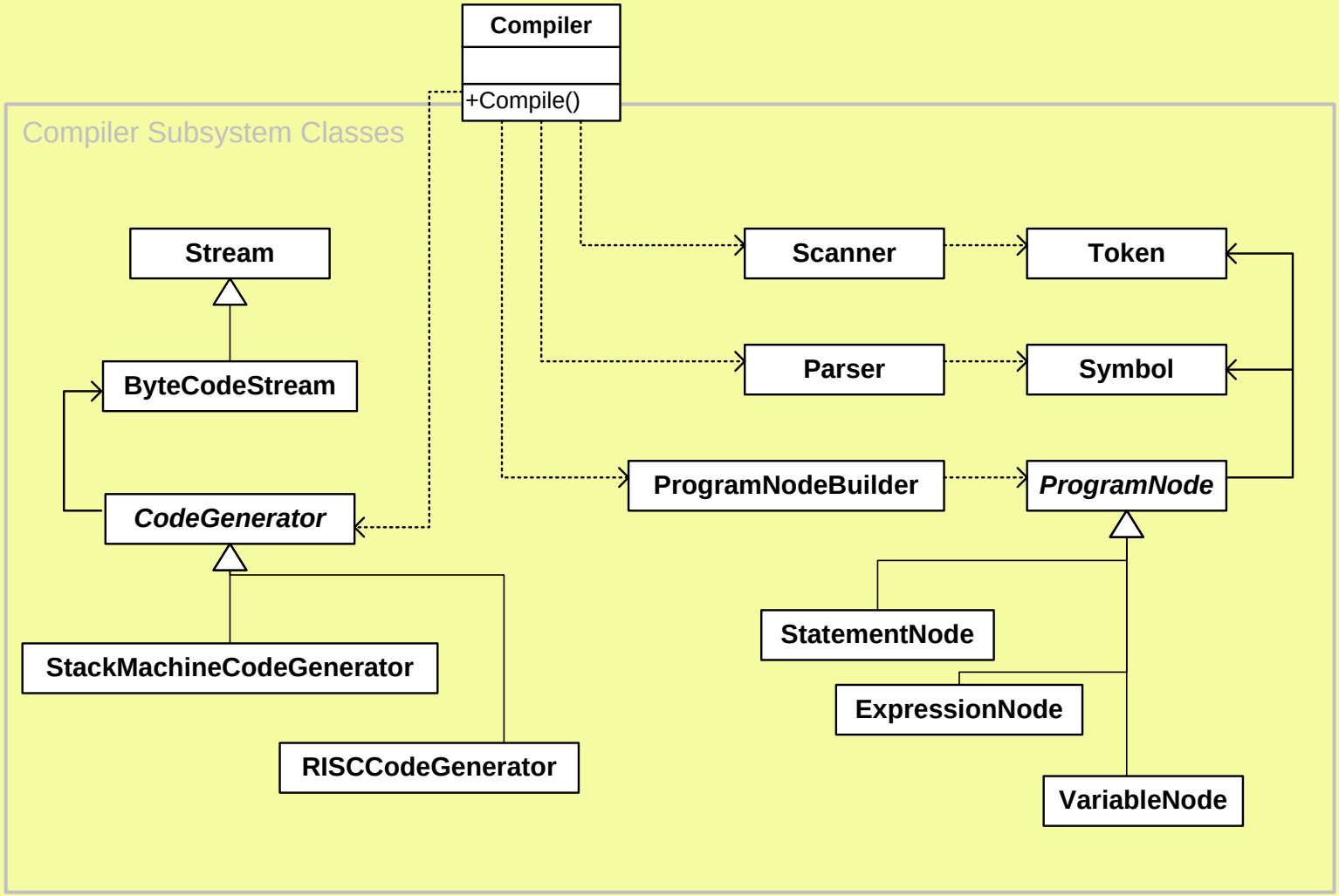


Façade Pattern: Intention

Intention

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

Façade Pattern: Example



Façade Pattern (3)

Consequences

- subsystem gets easier to use
- shield client from subsystem components (weak coupling)
- subsystem can be updated without any impact on the activity of the client
- applications can still access subsystem classes if they need to

Façade Pattern

Participants

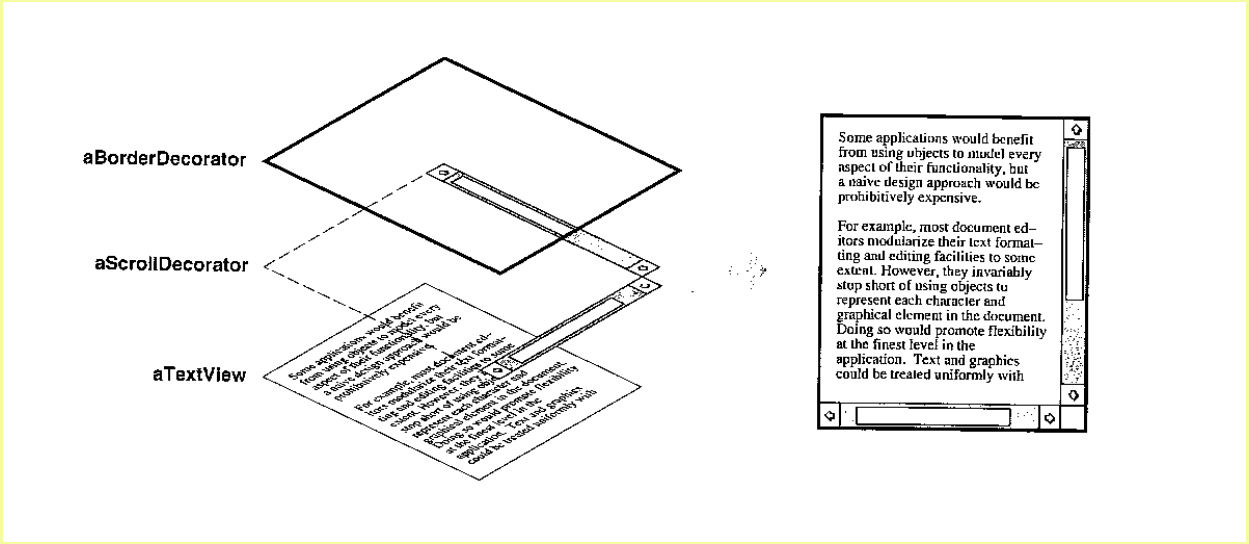
- Façade:
  - knows which subsystem classes are concerned by a request from the client
  - delegates client requests to appropriate subsystems objects
- Subsystem classes
  - implement subsystem functionality
  - handle work assigned by the Façade object
  - have no knowledge of the Façade

Decorator Pattern: Motivation

- Suppose we have a TextView object that display text in a window
- We want to add scroll bars, but not by default (we might not always want them)
- We also want to add a thick black borders around TextView (but not always)
- Inheritance? No, because
  - Inflexible: the choice of borders is made statically (compile-time and not run-time)
  - Too many ad-hoc classes: increase the complexity of a system
  - A client can't control how and when to decorate the component with a border/scrollbar

Decorator Pattern: Motivation (2)

- A more flexible approach: enclose the TextView component in another object that adds the border, the Decorator
- The Decorator
  - conforms to the interface of the component it decorates so that it's transparent to the component's clients
  - forwards requests to the component and may perform additional actions

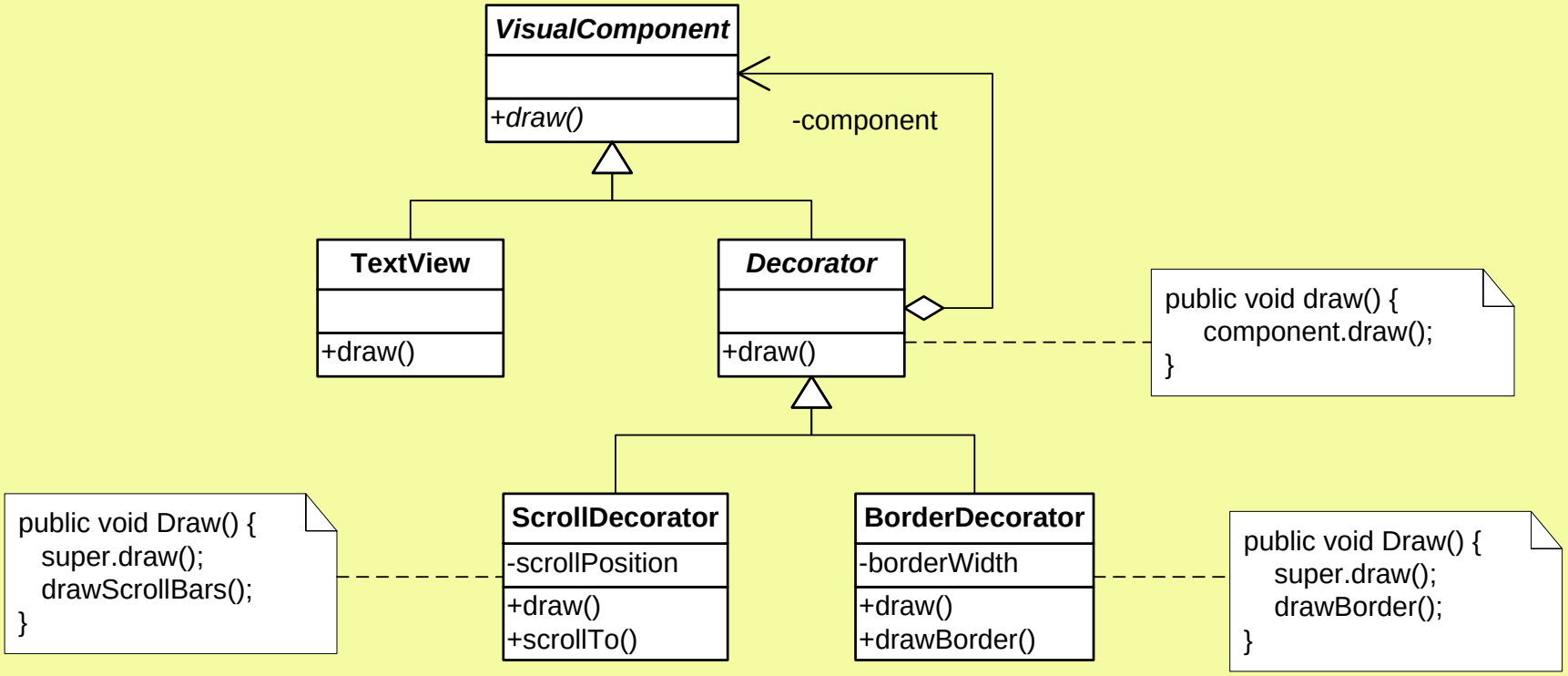


Decorator Pattern

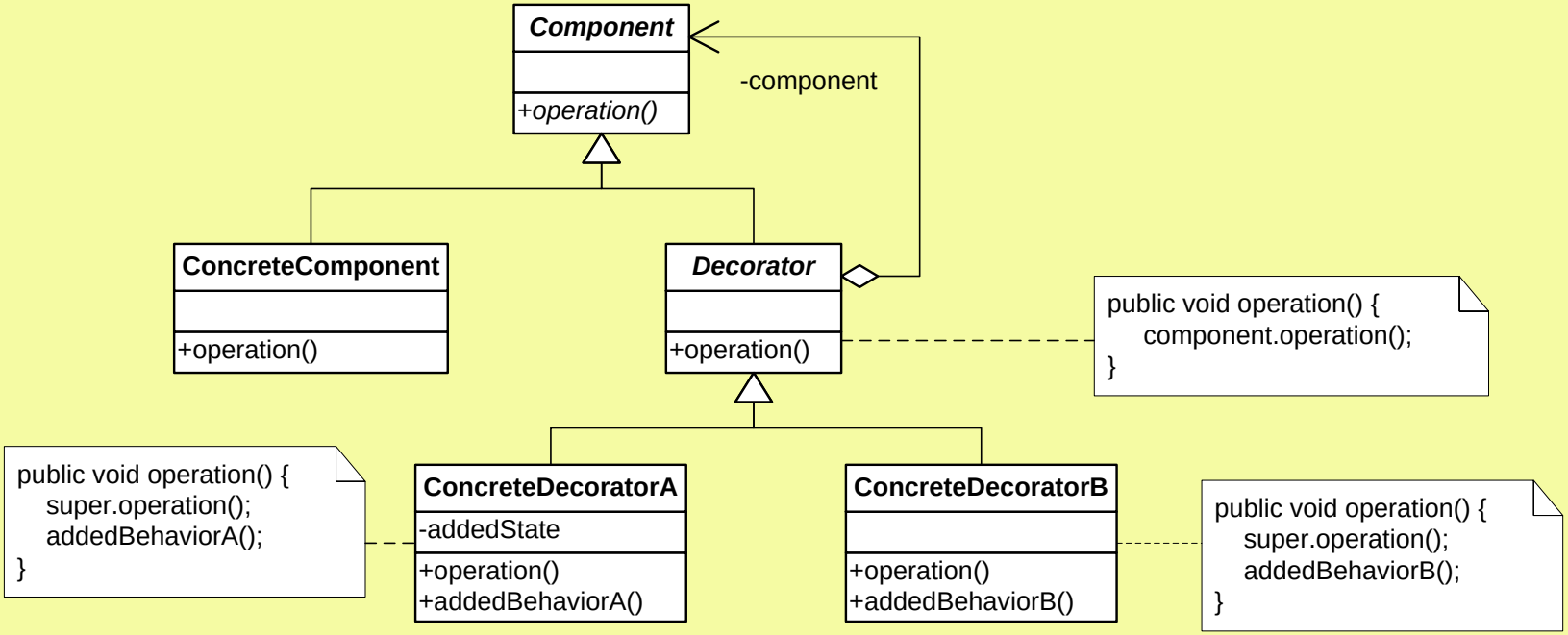
Intent

- Attach additional responsibilities to an object dynamically (i.e. to individual objects, not to entire classes!)
- Provide a flexible alternative to subclassing for extending functionality

Decorator Pattern



Decorator: Structure & Participants



- Component: defines the interface for objects that can have responsibilities added to them dynamically
- ConcreteComponent: object to which responsibilities can be attached
- Decorator: maintains a reference to a component object and defines an interface to conform to Component's interface
- ConcreteDecorator: adds specific responsibilities to the component

Decorator Pattern in Java

```
interface Component {
    public void operation();
}

class ConcreteComponent implements Component {
    public void operation() {
        System.out.println("root");
    }
}
```

Decorator Pattern in Java

```
abstract class Decorator implements Component {
    private Component component

    public Decorator(Component component) {
        this.component = component;
    }

    public Component getComponent() {
        return component;
    }
}
```

Decorator Pattern in Java

```
class ConcreteDecoratorA extends Decorator {
    public ConcreteDecoratorA(Component component) {
        super(component);
    }
    public void operation() {
        try {
            System.out.print("Decorator A");
            this.getComponent().operation();
        }
        catch(Exception e) {
            System.out.println(“Unknown component");
        }
    }
}
```

Decorator Pattern in Java

```
class TestDecorator {
    public static void main(String [] args) {
        Decorator dec =
        new ConcreteDecoratorB(
                (Component)new ConcreteDecoratorA(
                        (Component)new ConcreteComponent()
                )
        );
        dec.operation();
    }
}
```

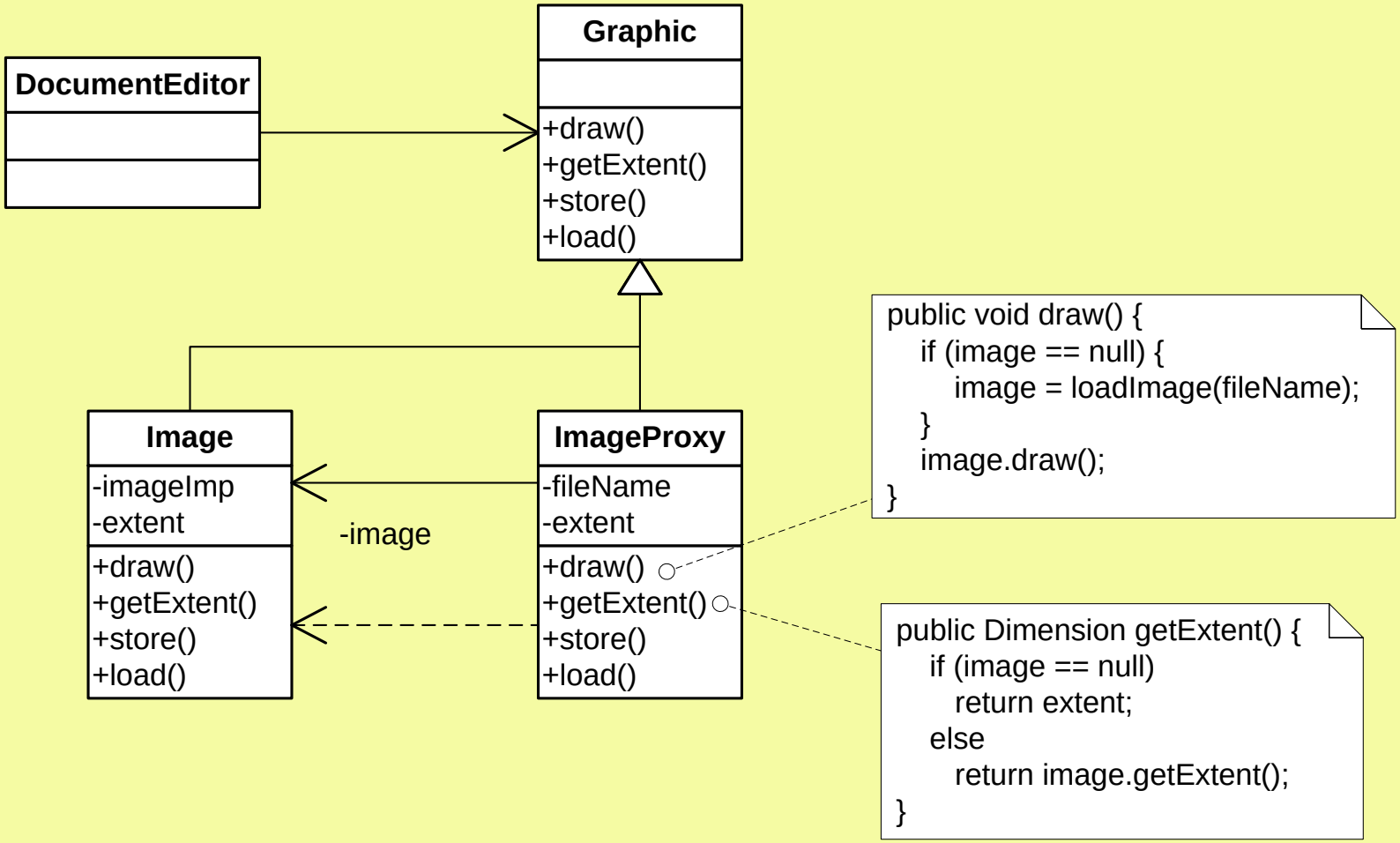
Proxy Pattern: Motivation

- Consider a document editor which can embed graphical objects in a document
- Some graphical objects can be expensive to create, but opening a document should be fast
- Creating large objects is necessary only when they become visible
- Therefore, we put a Proxy in place of the image when it is not visible, which acts as a placeholder for the image when it is not visible

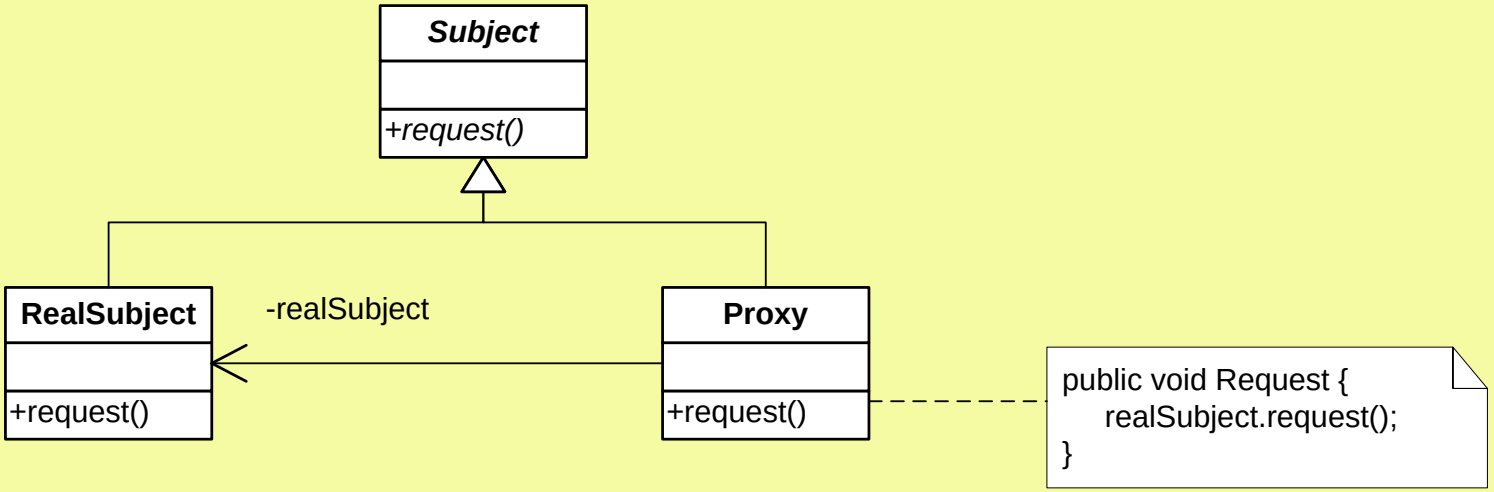
Proxy Pattern: Intent

Provide a surrogate or an intermediate object between the Client and object X to control access to object X

Proxy Pattern : Motivation



Proxy Pattern: structure & participants



- Proxy
  - maintains a reference to access the real subject
  - provides an interface identical to Subject's
  - controls access to real Subject (can create & destroy)
- Subject
  - Defines common interface for RealSubject and Proxy
- RealSubject
  - Defines the real object that the proxy represents