

More Design Patterns

Design Patterns Stories



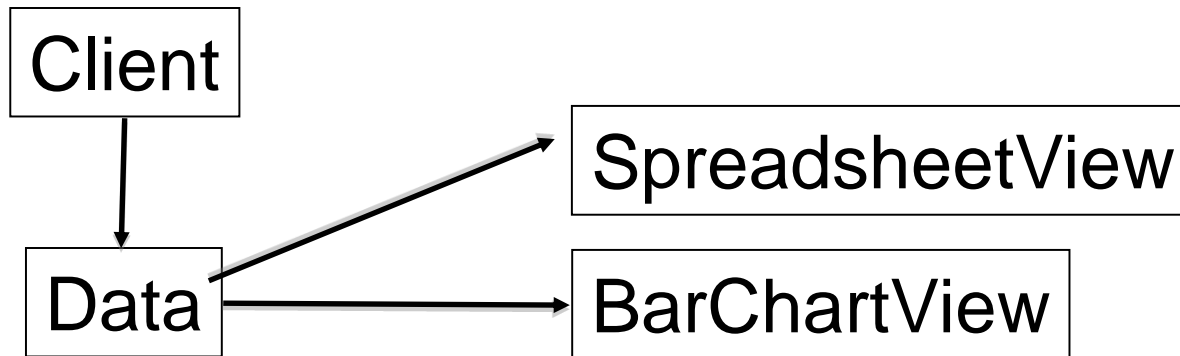


Observer Pattern

- Question: how to handle an object (model), which has many “observers” (views) that need to be notified and updated when the object changes state
- For example, an interface toolkit with various presentation formats (spreadsheet, bar chart, pie chart). When application data, e.g., **stocks data** (**model**) changes, all presentations (**views**) should change accordingly

A Naïve Design

- Client stores information in **Data**
- Then **Data** class updates the views accordingly



- Problem: to add a view, or change a view, we must change **Data**. Better to insulate **Data** from changes to **Views**!
- Single responsibility principle

A Better Design

- Data class has minimal interaction with Views
 - Only needs to [update](#) Views when it changes

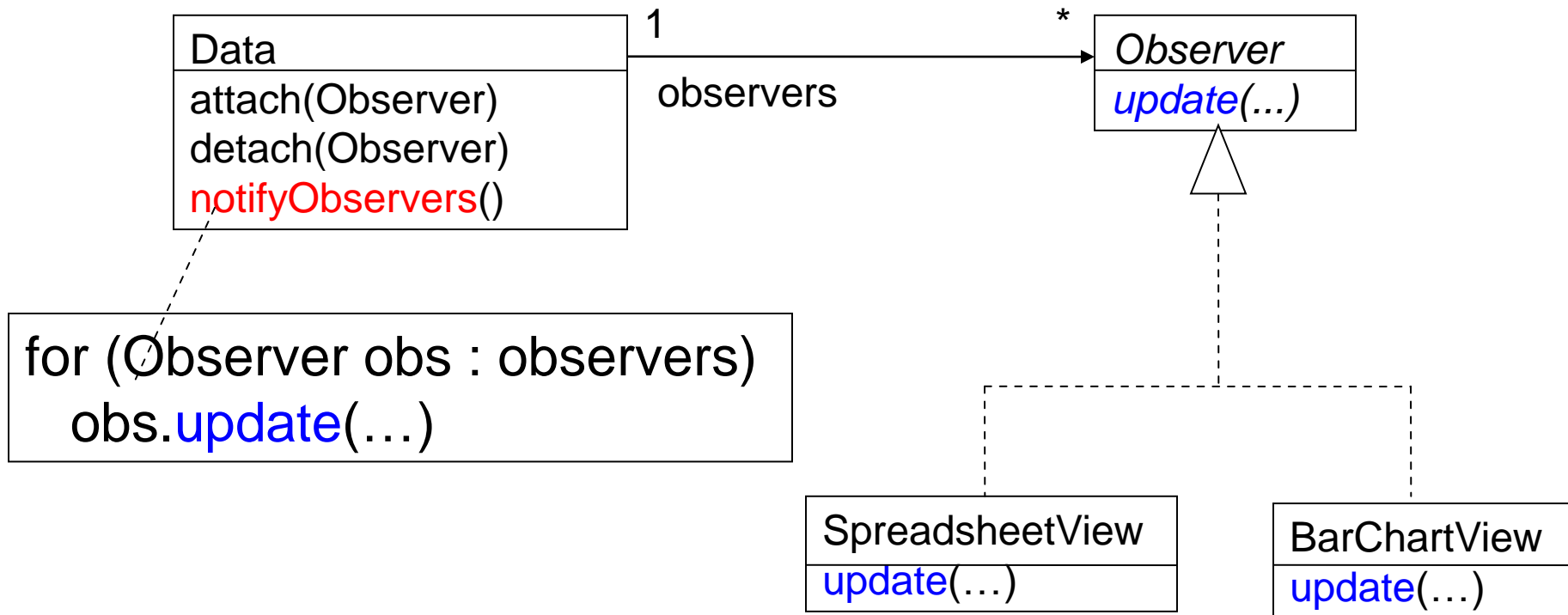
Old, naïve design:

```
class Data {  
    ...  
    void updateViews() {  
        spreadsheet.update(newData);  
        barChart.update(newData);  
        // Edit this method when  
        // different views are added.  
        // Bad!  
    }  
}
```

Better design:

```
class Data {  
    List<Observer> observers;  
    void notifyObservers() {  
        for (obs : observers)  
            obs.update(newData);  
    }  
}  
  
interface Observer {  
    void update(...);  
}
```

Class Diagram



Client is responsible for creation:

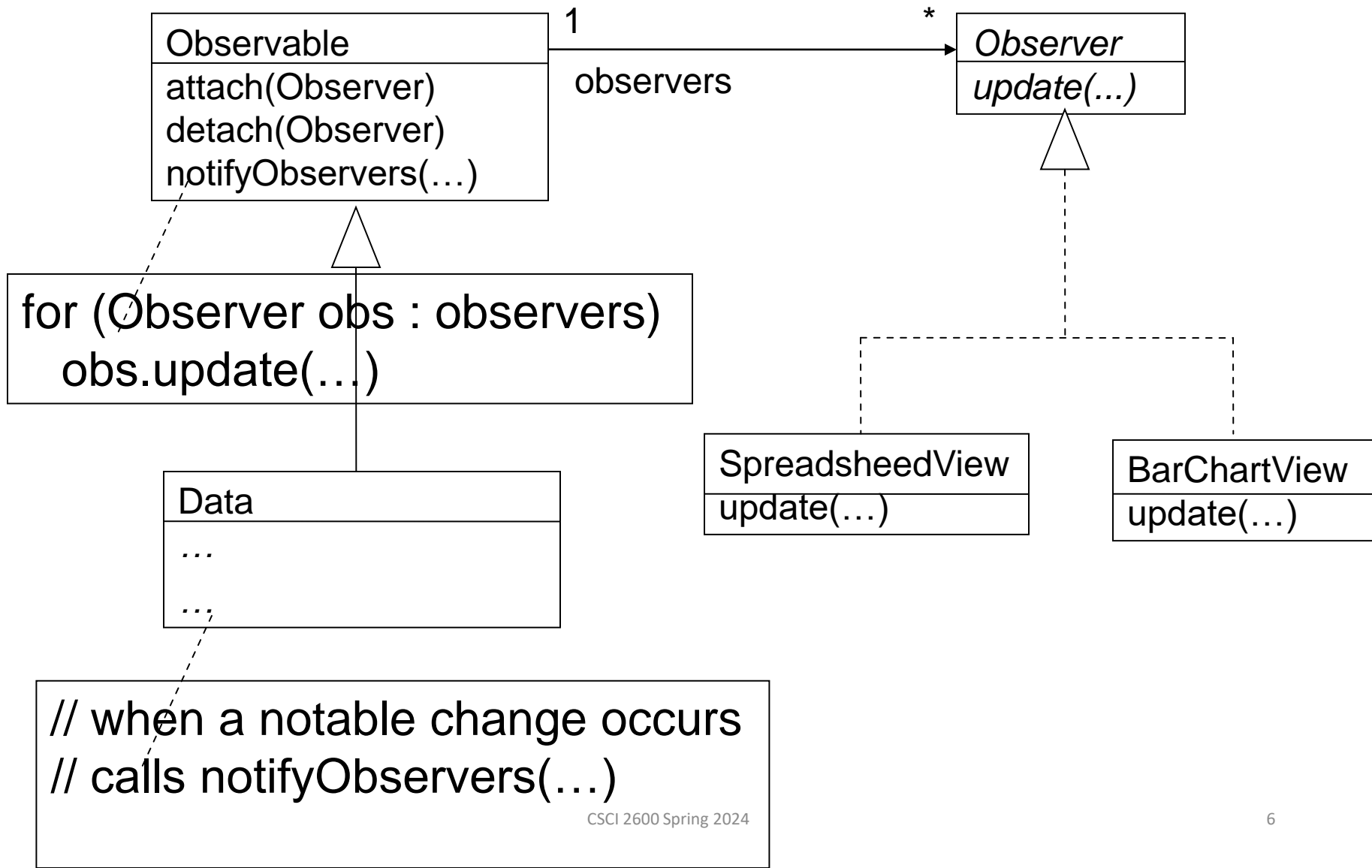
```
data = new Data();
```

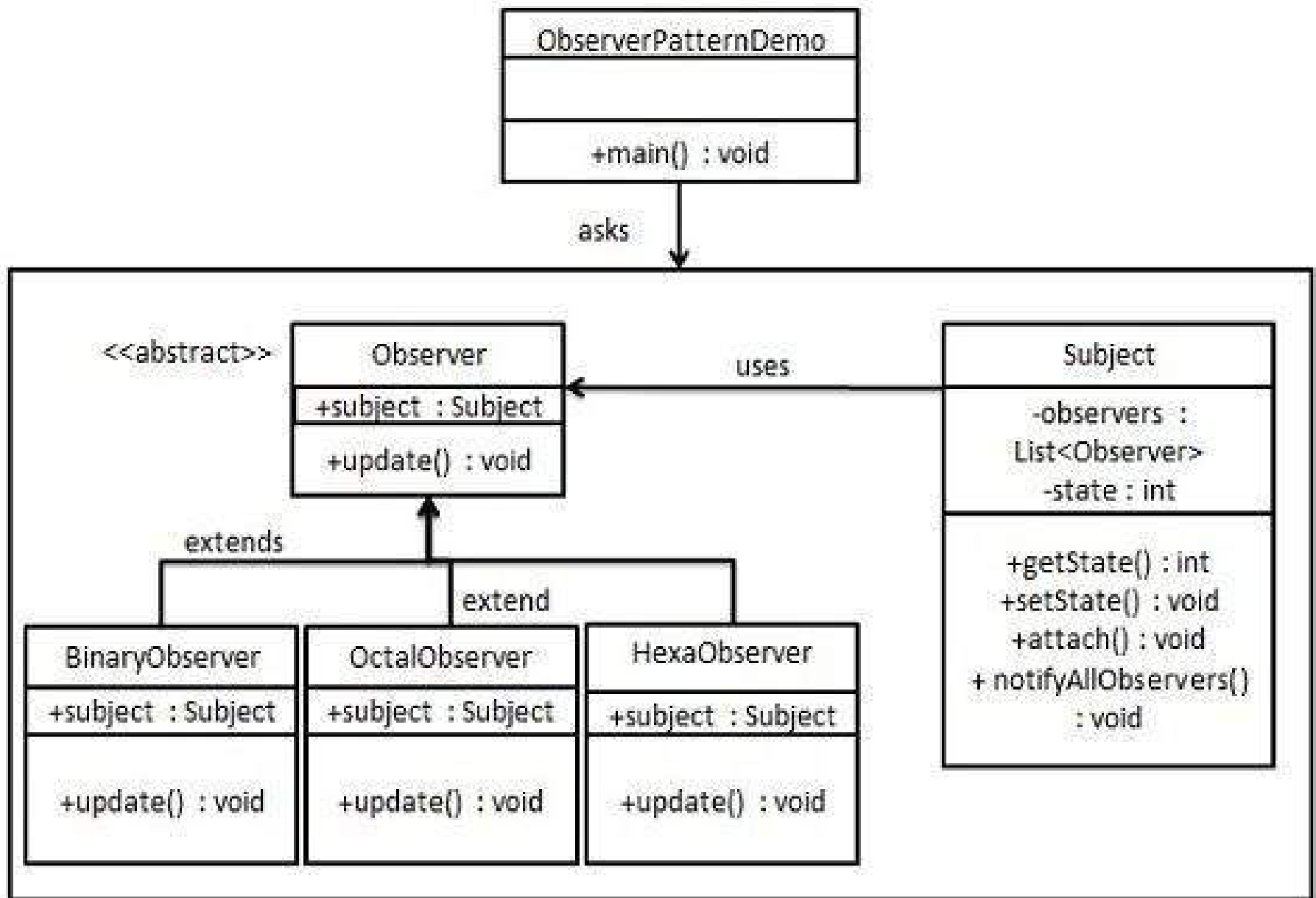
```
data.attach(new BarChartView());
```

Data keeps list of Views, notifies them when change.

Data is minimally connected to Views!

Even Better





https://www.tutorialspoint.com/design_pattern/observer_pattern.htm

```

import java.util.ArrayList;
import java.util.List;

public class Subject { // an Observable

    private List<Observer> observers = new ArrayList<Observer>();
    private int state;

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;
        notifyAllObservers();
    }

    public void attach(Observer observer){
        observers.add(observer);
    }

    public void notifyAllObservers(){
        for (Observer observer : observers) {
            observer.update();
        }
    }
}

```



```
public abstract class Observer {  
    protected Subject subject;  
    public abstract void update();  
}
```

```
public class BinaryObserver extends Observer{
```

```
    public BinaryObserver(Subject subject){  
        this.subject = subject;  
        this.subject.attach(this);  
    }
```

```
    // get the state of the subject
```

```
    @Override
```

```
    public void update() {  
        System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );  
    }  
}
```

```
public class OctalObserver extends Observer{
```

```
    public OctalObserver(Subject subject){  
        this.subject = subject;  
        this.subject.attach(this);  
    }
```

```
    @Override
```

```
    public void update() {  
        System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );  
    }  
}
```

```
public class HexaObserver extends Observer{
```

```
    public HexaObserver(Subject subject){  
        this.subject = subject;  
        this.subject.attach(this);  
    }
```

```
    @Override
```

```
    public void update() {  
        System.out.println( "Hex String: " + Integer.toHexString( subject.getState() ).toUpperCase() );  
    }  
}
```

```
public class ObserverPatternDemo {  
    public static void main(String[] args) {  
        Subject subject = new Subject();  
  
        new HexaObserver(subject);  
        new OctalObserver(subject);  
        new BinaryObserver(subject);  
  
        System.out.println("First state change: 15");  
        subject.setState(15);  
        System.out.println("Second state change: 10");  
        subject.setState(10);  
    }  
}
```

Output:

First state change: 15

Hex String: F

Octal String: 17

Binary String: 1111

Second state change: 10

Hex String: A

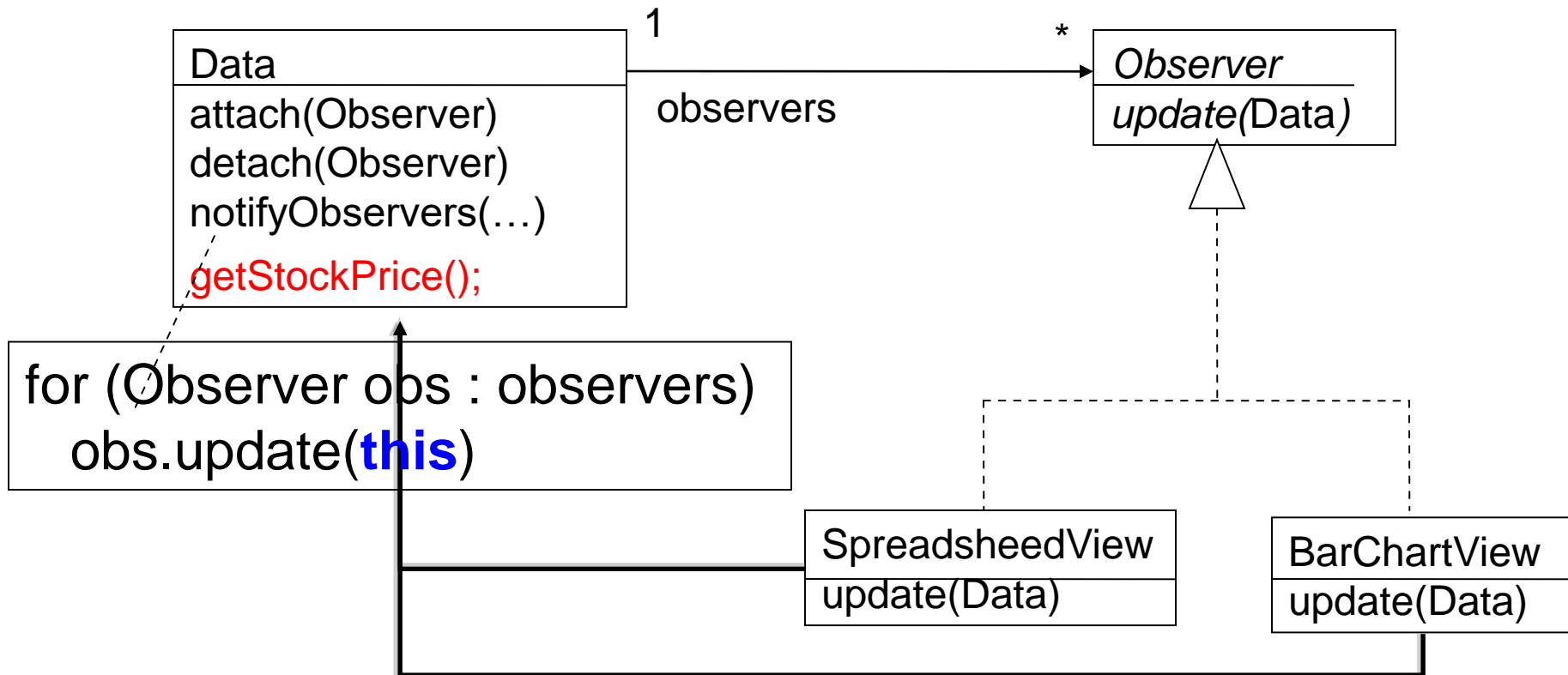
Octal String: 12

Binary String: 1010

Push vs. Pull Model

- Question: How does the object (Data in our case) know what info each observer (View) needs?
- **Push** model: Object sends the info to Observers
- **Pull** model: Object does not send info directly. It gives access to itself to the Observers and lets each Observer extract the data they need

Observer Pattern



Pull model: observers have access to Data, they can pull the info they need.

THE MODEL

From JDK

Example of Observer

```
public class SaleItem extends Observable {  
    private String name;  
    private float price;  
    public SaleItem(String name, float price) {  
        this.name = name;  
        this.price = price;  
    }  
    public void setName(String name) {  
        this.name = name;  
        setChanged() ;  
        notifyObservers(name) ;  
    }  
    public void setPrice(float price) {  
        // analogous to setName  
    }  
}
```

From JDK. Marks that object has changed.

From JDK. If object has changed, calls **obs.update(this, name)**

An Observer of Name Changes (Push)

THE VIEW

```
public class NameObserver implements Observer {
    private String name;

    public void update(Observable obj, Object arg) {
        // we ignore obj in this example
        if (arg instanceof String) {
            name = (String) arg;
            System.out.println("NameObserver:
                               Name changed to " + name);
        }
        else
            System.out.println("NameObserver:
                               Some other change to observable!");
    }
}
```

From JDK

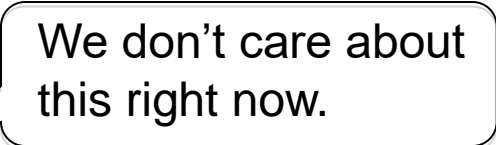
Implements update from JDK.
Results in **callback!**

Observables and Observers

- `update(Observable obj, Object arg)` allows for both Push and Pull models.
- `update` is called from `notifyObservers`, which is a library method!
- `notifyObservers()` calls `update(this, null)`
- `notifyObservers(arg)` calls `update(this, arg)`.
- An **Observer** (such as `NameObserver`) can choose to use the first argument of `update`, the `Observable obj`, cast it to the appropriate type and extract the info it needs (the Pull model)
- or it can choose to ignore `Observable obj`, and use the argument `Object arg`, which the data sends (the Push model).

An Observer of Price Changes (Push)

```
public class PriceObserver implements Observer {  
    private Float price;  
  
    public void update(Observable obj, Object arg) {  
        if (arg instanceof Float) {  
            price = (Float) arg;  
            System.out.println("PriceObserver:  
                               Price changed to " + price);  
        }  
        else  
            System.out.println("PriceObserver:  
                               Some other change to observable!");  
    }  
}
```



An Observer of Changes (Pull)

```
public class PriceObserver implements Observer {  
    private String name;  
    private float price;  
  
    public void update(Observable obj, Object arg) {  
        name = ((SaleItem) obj).name;  
        price = ((SaleItem) obj).price;  
        System.out.println("NameObserver:  
                           Name changed to " + name +  
                           "Price is " + price);  
    }  
}
```

From
JDK

Implements update from JDK.
Results in **callback**!

The Client

```
// SaleItem implements Observable
SaleItem si = new SaleItem("Corn Pops", 1.29f);
// Observers
NameObserver nameObs = new NameObserver();
PriceObserver priceObs = new PriceObserver();

// Now add observers
si.addObserver(nameObs);
si.addObserver(priceObs);

// Make changes to the Subject.
si.setName("Frosted Flakes");
si.setPrice(4.57f);
si.setPrice(9.22f);
si.setName("Sugar Crispies");
```

JDK. Since si is
Observable!

MVC

We have just implemented the **Mode-View-Controller** (MVC) method

SaleItem – The **Model**

PriceObserver, NameObserver – The **View**

Client – The **Controller**

Another Example

- An application that computes a path on a map and displays the path.
When user requests different path, display changes
- Initially, application displays using a simple text-based UI
 - Therefore, a text-based View (i.e., Observer)
- Later, application will display using a GUI interface
 - A GUI-based View (another Observer)

Another Example of Observer

// Represents sign-up sheet of students

```
public class SignupSheet extends Observable {  
    private List<String> students =  
        new ArrayList<String>();  
    public void addStudent(String student) {  
        students.add(student);  
        notifyObservers();  
    }  
    public int size() {  
        return students.size();  
    }  
}
```

THE VIEW

Example of Observer

```
public class SignupObserver implements Observer
{
    // called from notifyObservers, which
    // was called when SignupSheet changed
    public void update(Observable o,
                        Object arg) {
        System.out.println("Signup count: "
            + ((SignupSheet)o).size());
    }
}
```

The SignupSheet observable was sent when notifyObservers called update(this,...)

We don't care about arg now.

The Client

```
SignupSheet s = new SignupSheet();  
s.addStudent("Ana");  
// nothing visible happens. Why?  
  
s.addObserver(new SignupObserver());  
s.addStudent("Katarina");  
// what happens now?
```

What model's used here? Push model or pull model?

Where is the observable?
Where is the observer?

```
{ propertyListeners.add(lis); }
```

```
for each pl in  
propertyListeners  
    pl.onPropertyEvent  
        (this,name,value);
```

```
Sale  
-----  
addPropertyListener(PropertyListener lis)  
publishPropertyEvent(name,value) notify()  
setTotal(Money newTotal)
```

propertyListeners ↓ *

```
<<interface>>  
PropertyListener  
-----  
onPropertyEvent(source, name, value) update()
```

```
{ if (name.equals("sale.total"))  
    saleTextField.  
        setText(value.toString())  
}
```

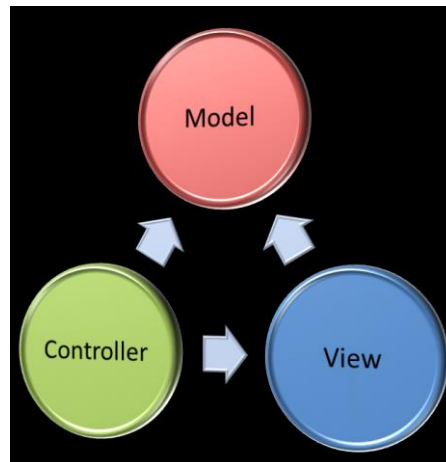
SaleTotalFrame

```
onPropertyEvent(source, name,value)  
initialize(Sale sale)
```

```
{ sale.addPropertyListener(this); }
```

Model-view Principle

- **Observer** pattern known as **Model-view** or **Model-view-controller**
- “Model” objects (e.g., Sale, SignupSheet) should not know much about concrete “view” objects (e.g., SaleTotalFrame, SignupObserver)
- Domain layer should be minimally connected with presentation layer
 - Open/closed principle: if user decides to change/upgrade interface, the change shall trigger no modification to domain layer



Observer and Observable are deprecated in Java 9 – why?

- They didn't provide a rich enough event model for many applications.
 - For example, they could support only the notion that something has changed, but only convey indirect information about what has changed.
- All Observables are the same.
 - You have to implement the logic that is based on instanceof and cast object to concrete type in Observable.update() method.
- Not Serializable
 - Since, Observable doesn't implement Serializable. So, you can't Serialize Observable nor its subclass.
- Not thread safe
 - The methods can be overridden by its subclasses, and event notification can occur in different orders and possibly on different threads, which is enough to disrupt any "thread safety".
- Observable is a class not an interface

Another Example – Functional This Time

```
package twitterDemo; // This is the O-O version

import java.util.List;
import java.util.ArrayList;

interface Observer {
    void inform(String tweet);
}

class NYTimes implements Observer {
    @Override
    public void inform(String tweet) {
        if (tweet != null && tweet.contains("money")) {
            System.out.println("Breaking news in NY!" + tweet);
        }
    }
}

... other news sources – similar to above
}
```

```
interface Subject {  
    void registerObserver(Observer o);  
    void notifyObservers(String tweet);  
}  
  
class Feed implements Subject {  
    private final List<Observer> observers = new ArrayList < > ();  
  
    public void registerObserver(Observer o) {  
        this.observers.add(o);  
    }  
  
    public void notifyObservers(String tweet) {  
        observers.forEach(o -> o.inform(tweet));  
    }  
}  
  
public class TwitterDemo {  
  
    public static void main(String[] args) {  
        Feed f = new Feed();  
        f.registerObserver(new NYTimes());  
        f.registerObserver(new Guardian());  
        f.registerObserver(new LeMonde());  
        f.notifyObservers("Save money!");  
    }  
}
```

A Functional Observer

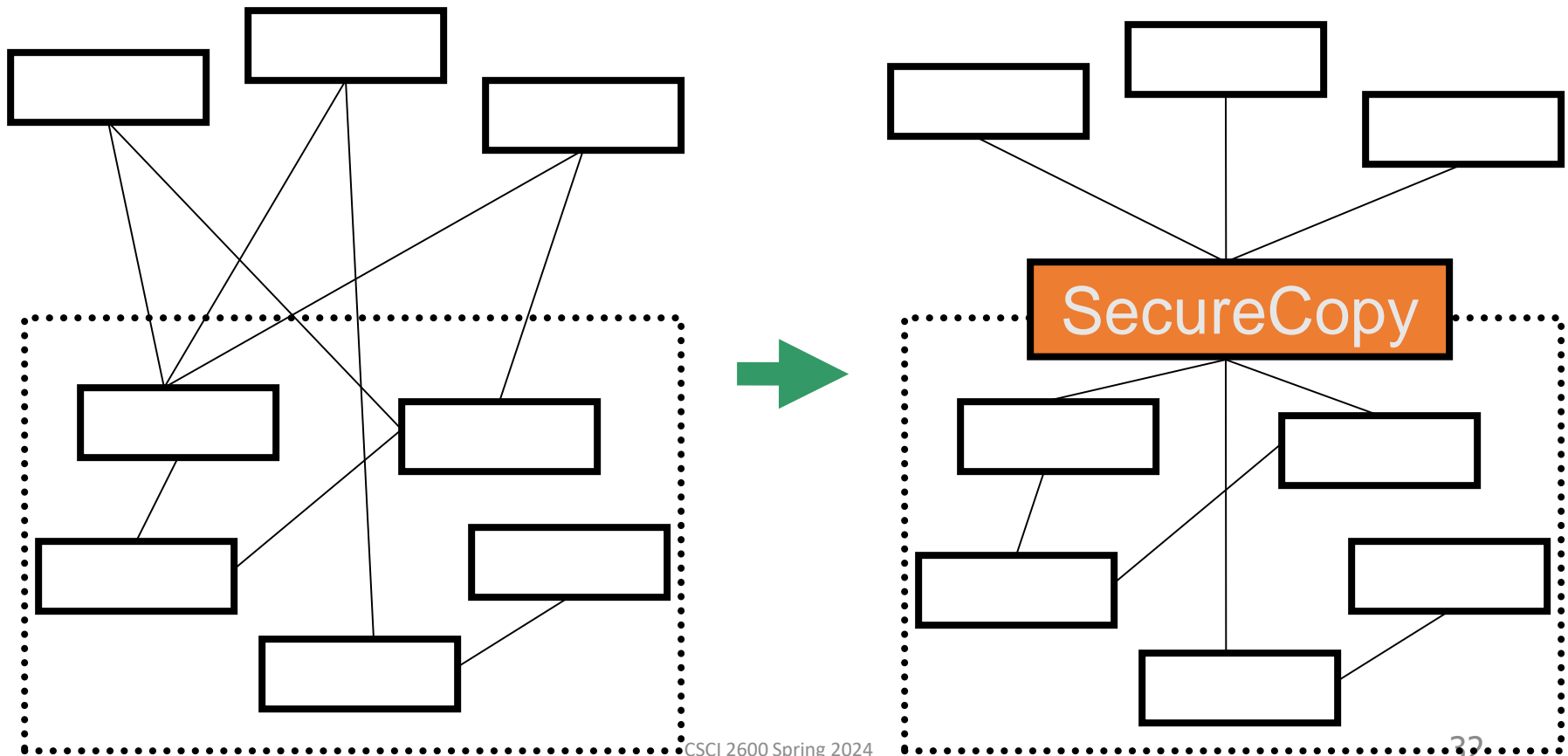
```
public class FunTwitterDemo {  
    public static void main(String[] args) {  
        Feed feedLambda = new Feed();  
        feedLambda.registerObserver((String tweet) -> {  
            if (tweet != null && tweet.contains("money")) {  
                System.out.println("Breaking news in NY! " + tweet);  
            }  
        });  
  
        feedLambda.registerObserver((String tweet) -> {  
            if (tweet != null && tweet.contains("queen")) {  
                System.out.println("Yet another news in London... " + tweet);  
            }  
        });  
  
        feedLambda.notifyObservers("Money money money, give me money!");  
    }  
}
```

Façade Pattern

- Question: how to handle the case, when we need a subset of the functionality of a powerful, extensive and complex library
- Example: We want to perform secure file copies to a server. There is a powerful and complex general-purpose security library. What is the best way to interact with this library?
- Example: Protect code against unstable/in progress API.
- Example: Travel booking system
 - Must interact with airline, limo, hotel systems, etc.
 - Systems have different APIs
 - Provide uniform interface

Façade Pattern

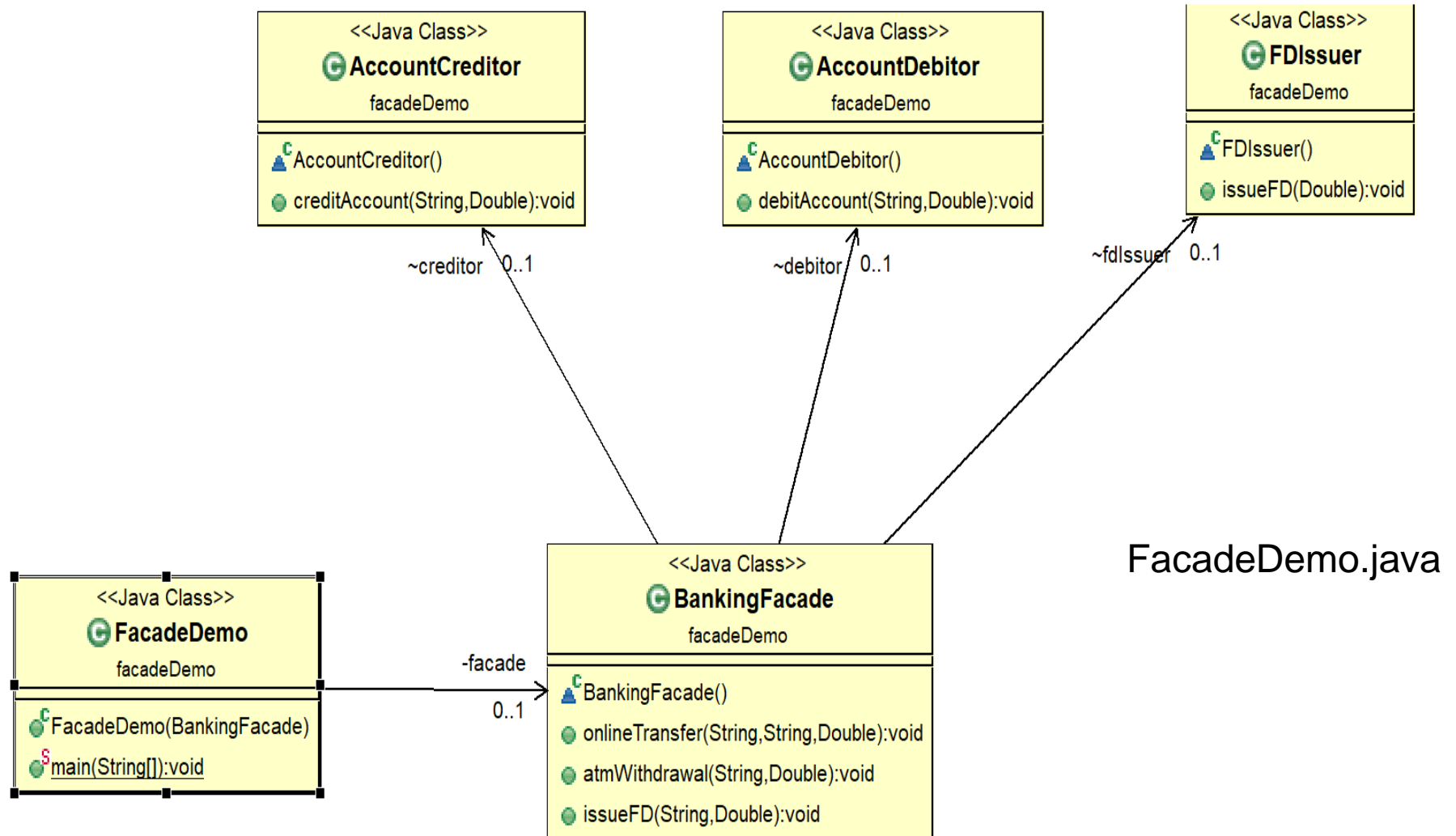
Build a Façade to the library, to hide its (mostly irrelevant) complexity. SecureCopy is the Façade.



Façade Pattern

- Façade reduces interactions between client and the complex library
- Façade hides (mostly irrelevant) complexity of the library
- If library changes, we'll only need to change the Façade, the client remains insulated
 - Open/closed principle: when change happens, the change has minimal impact
- Kind of like a proxy or even an adapter pattern
 - Main difference is that Façade is designed to hide complexity or underlying system

Façade Demo



Façade Pattern

- Façade pattern decreases coupling between the client and subsystems
 - The higher-level interface can absorb any changes which might happen in the lower subsystems interfaces
 - Provide a consistent interface to client
- Subsystem interfaces are not aware of Façade and they shouldn't have any reference of the Façade interface.
- The purpose of the Façade Pattern is to provide a single interface rather than multiple interfaces that do similar kinds of jobs.

Similar Patterns

Adapter

adapts a given class/object to a new interface.
Solves the problem of non-compatible interfaces

Façade

a simple gateway to a complicated set of functionality.
You make a black-box so your clients worry less
i.e. make interfaces simpler.
seems a bit like an adapter

Proxy

provides the same interface as the proxied-for class
and typically does some housekeeping stuff on its own.
Used to solve the problem of the client having to manage a heavy and/or complex object.
Used to restrict access to object
Has aspects of Singleton and Interning.

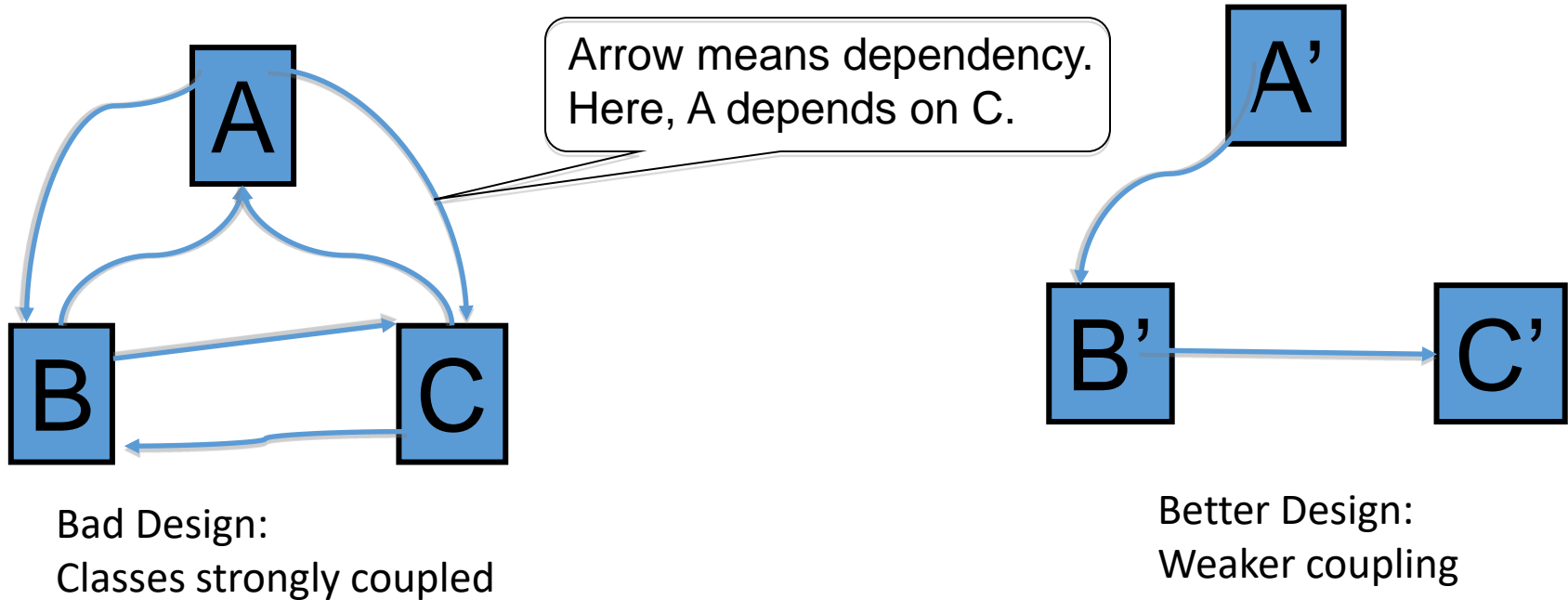
Decorator

used to add more functionality to your objects.
You do not hide the existing interfaces of the object but rather extend it at runtime.

Interactions Between Modules

- Interactions between modules (in our designs, module = class) cause complexity
- To simplify, split design into parts that don't interact as much
 - refactor
- **Coupling** is the amount of interaction among classes
 - Roughly, if class **A** calls methods/uses fields of class **B**, then there is **coupling** from **A** to **B**
- In design, we strive towards **low (weak) coupling**, i.e., minimal, necessary interactions

Low Coupling





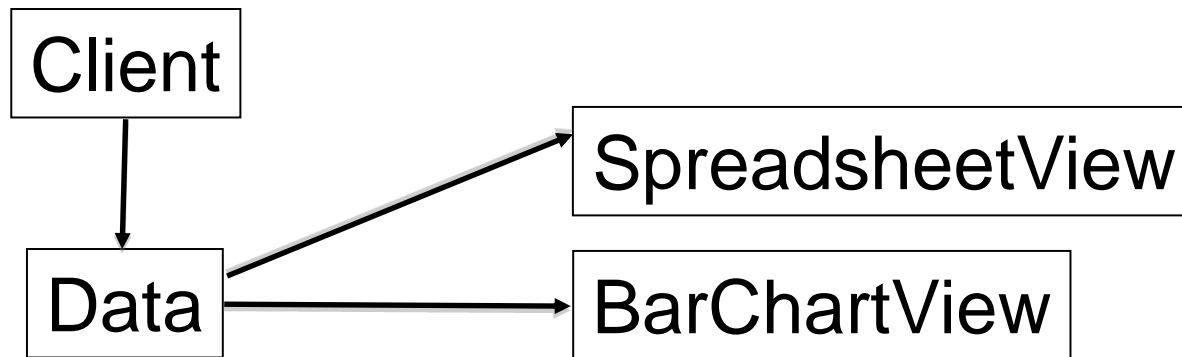
Coupling is the Path to the Dark Side

- Coupling leads to complexity
- Complexity leads to confusion
- Confusion leads to suffering
- If once you start down the dark path, forever will it dominate your destiny, consume you it will

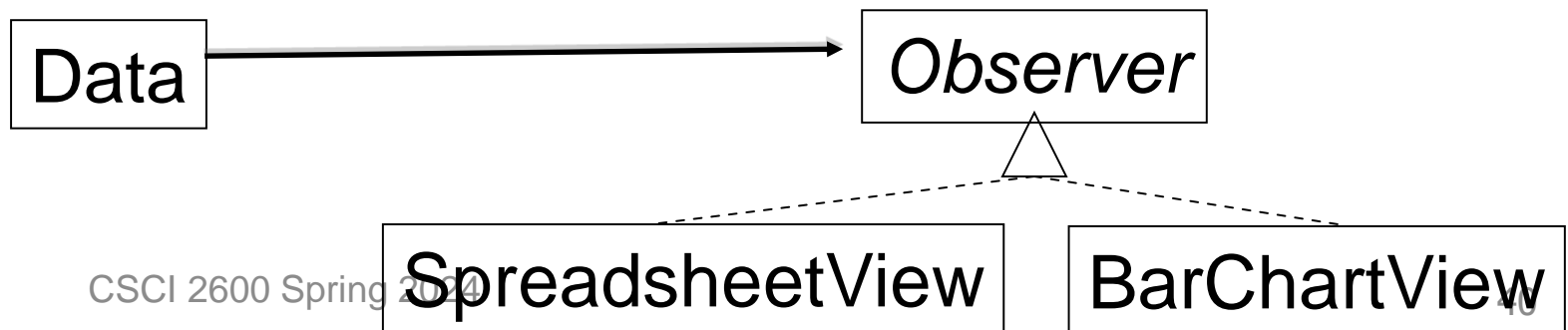


Observer promotes low coupling

- Bad. Data does not need to depend on Views

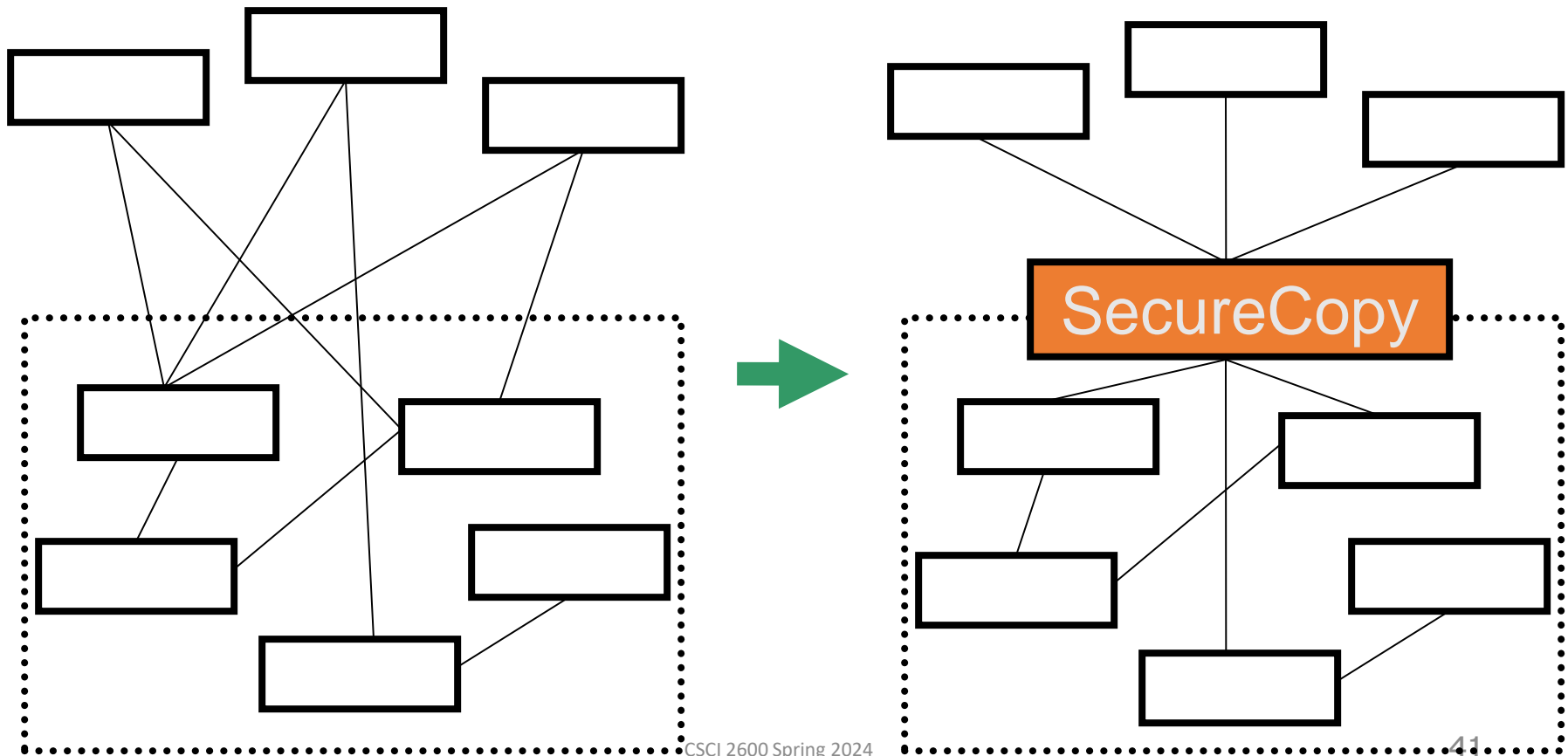


- Better: Weaken dependency of Data on Views
 - Introduce a weaker spec in the form of interface



Façade promotes low coupling

Façade weakens the dependency between Client and library.
Introduce Façade object: reduce #dependences from 3×5 to $3 + 5$



User interfaces: appearance vs. content

- It's easy to tangle up appearance and content
 - E.g. in dragging a line in a drawing program
 - Where are endpoints stored
 - Program state stored in widgets in dialog boxes
 - Not a good idea
 - Dialog boxes are transient, state is more general
- Neither content nor appearance is easily understood or changed
- Destroys flexibility
- Leads to subtle bugs
- Callbacks, listeners, and other patterns can help
- Stick to single purpose principle

Shared Constraints

- Coupling can arise from shared constraints
 - A module that writes a file and a module that reads a file in the same format
 - Even if there's no dependency on each other's code
 - If one fails to write the correct format, the other fails
- Shared constraints are easier to reason about/debug if they are encapsulated
 - Place all format information in a single module used by both read and write module

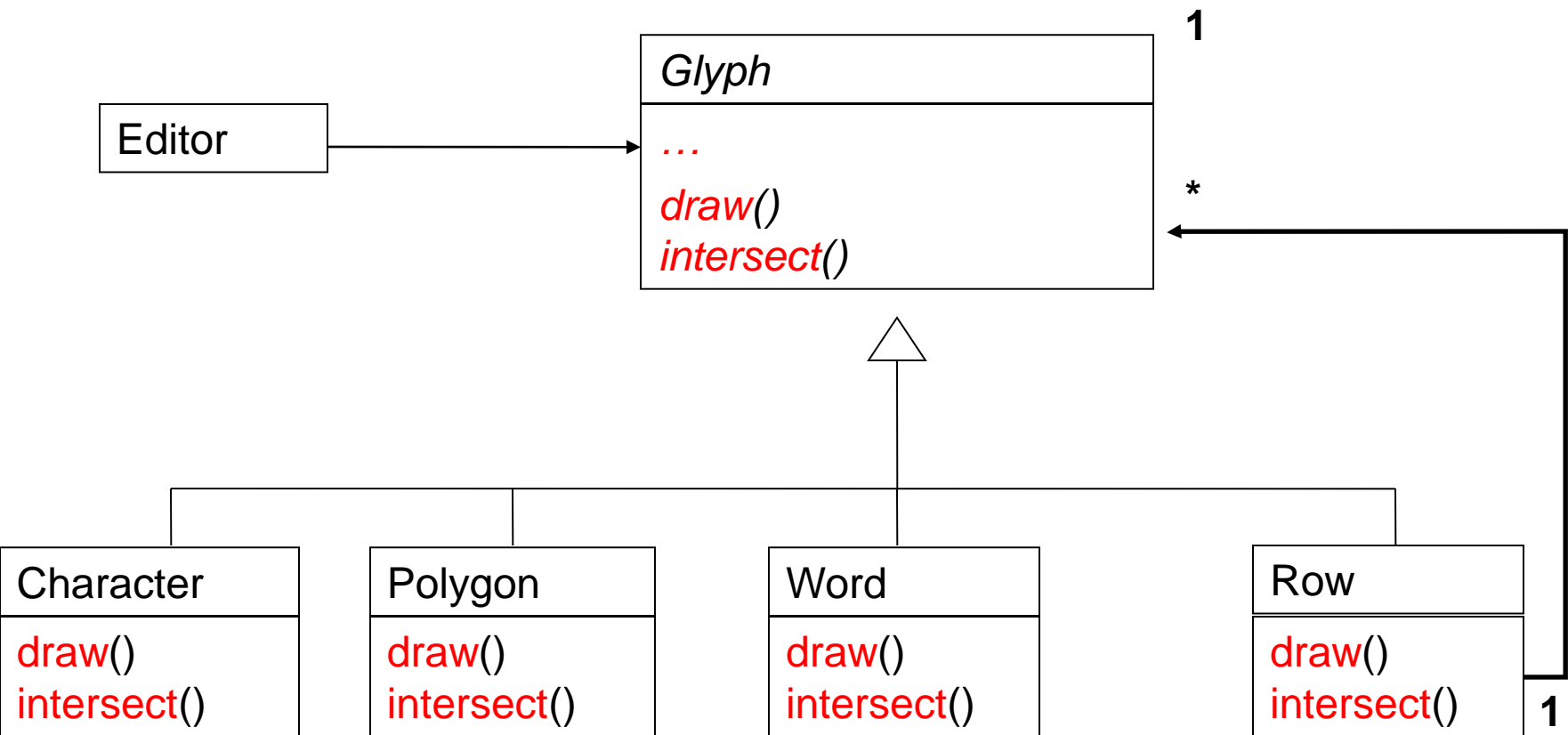
A Design Exercise

- We are building a document editor --- a large rectangle that displays a document. Document can mix text, graphical shapes, etc.
Surrounding the document are the menu, scrollbars, borders, etc.
- Structure, Formatting, Embellishing the UI, User commands, Spell checking

Structure

- Hierarchical structure --- document is made of columns, a column is made up of rows, a row is made up of words, images, etc.
- Editor should treat text and graphics uniformly. Editor should treat simple and complex elements uniformly
- What design pattern?

The Composite Pattern



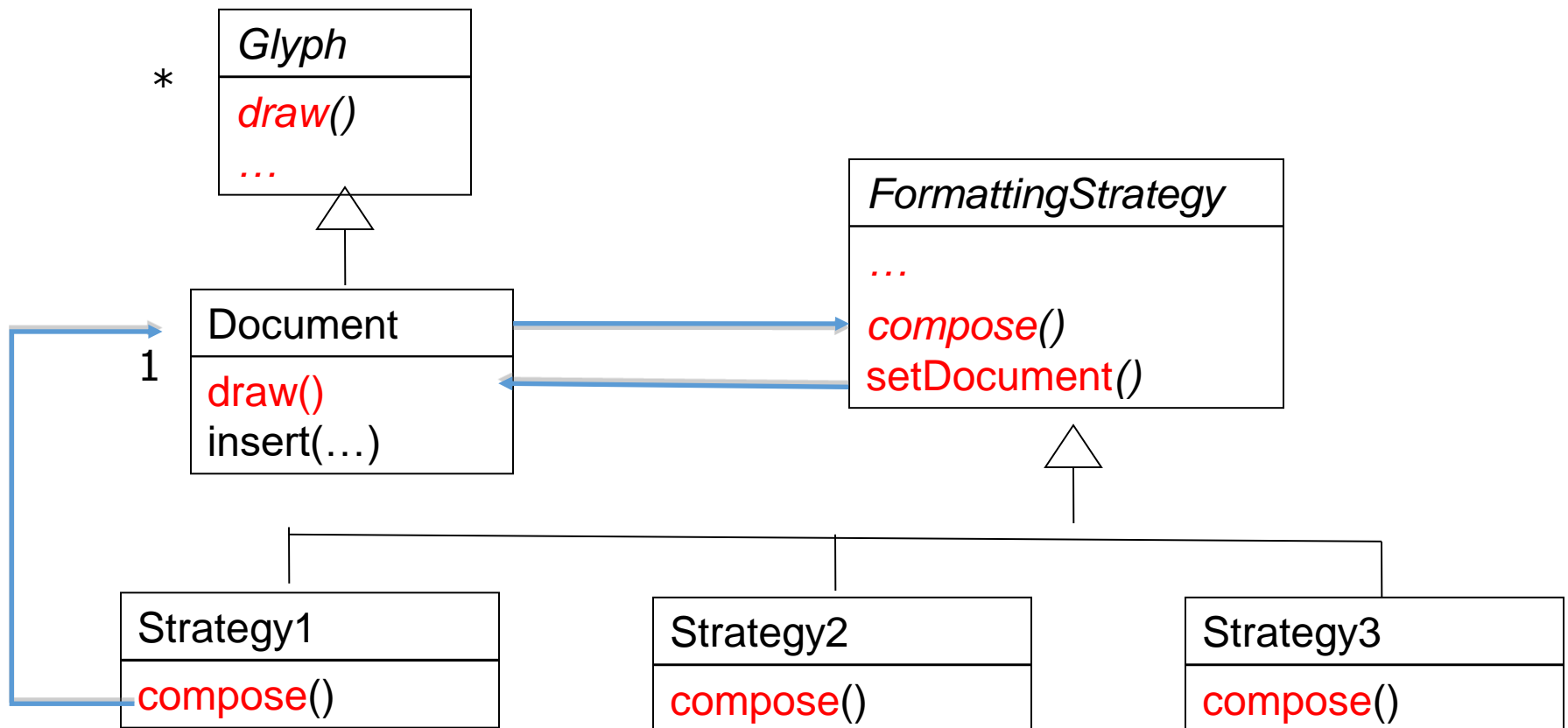
What's missing from this picture?

Formatting

- Formatting displays the document
- Many different formatting strategies are possible
 - We would use different formatting strategies over the same hierarchical structure
- Each formatting strategy is complex

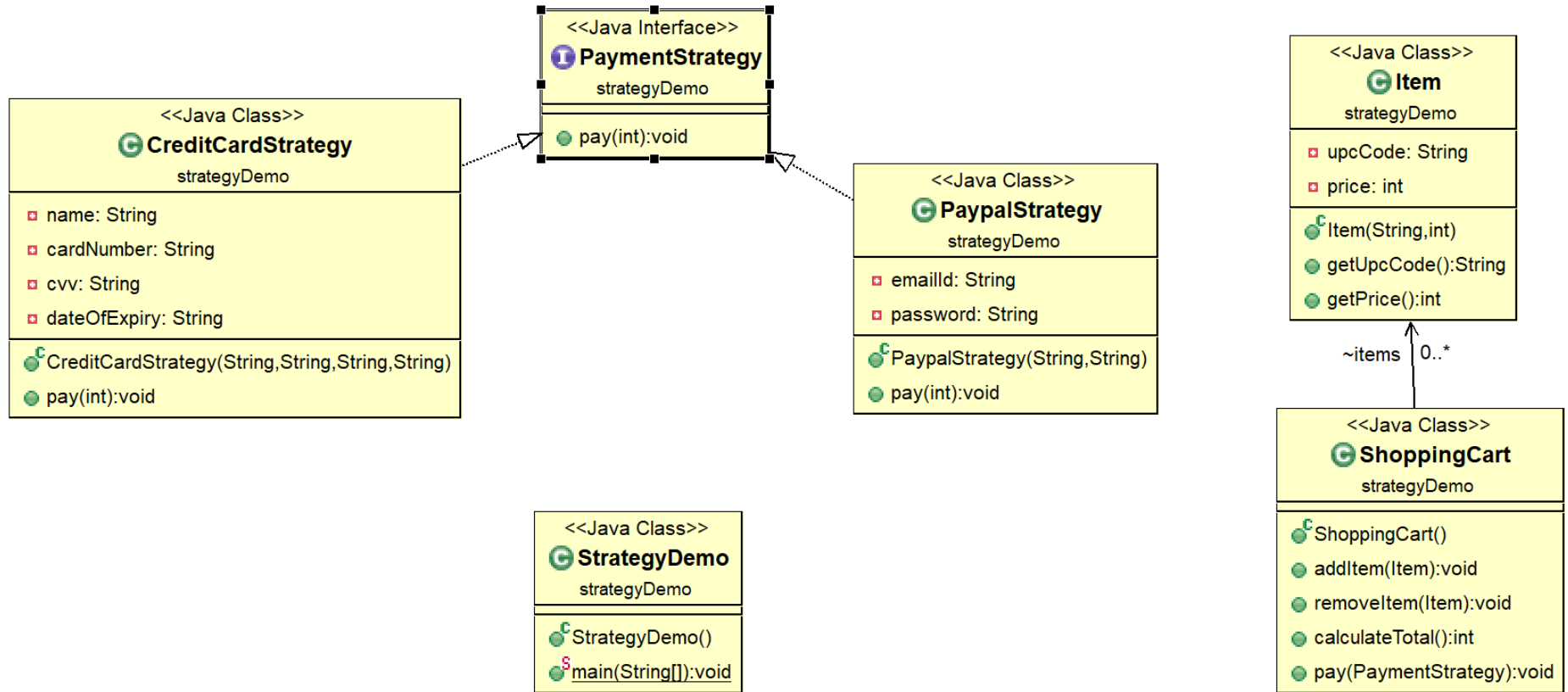
The Strategy Pattern

- Encapsulates an algorithm in an object



Strategy Pattern

- Also known as the **Policy Pattern**
- Define multiple algorithms and let client application pass the algorithm to be used as a parameter
- Reduces coupling
 - Program to an interface, not an implementation
- Collections.sort is an example
 - It takes Comparator parameter.
 - Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways.
- Strategy pattern is very similar to State and Command Patterns.
- Strategy pattern is useful when we have multiple algorithms for specific task and we want our application to be flexible to chose any of the algorithm at runtime for specific task.



StrategyDemo.java

Functional Strategy Pattern

```
@FunctionalInterface
interface Strategy {
    double apply(double a, double b);
};

public class FunStrategyDemo {
    public static double execute(double a, double b,
                                Strategy strategy) {
        return strategy.apply(a, b);
    }

    public static void main(String[] args) {
        System.out.println(execute(10, 5, (a, b) -> a + b));
        System.out.println(execute(10, 5, (a, b) -> a - b));
        System.out.println(execute(10, 5, (a, b) -> a * b));
    }
}
```

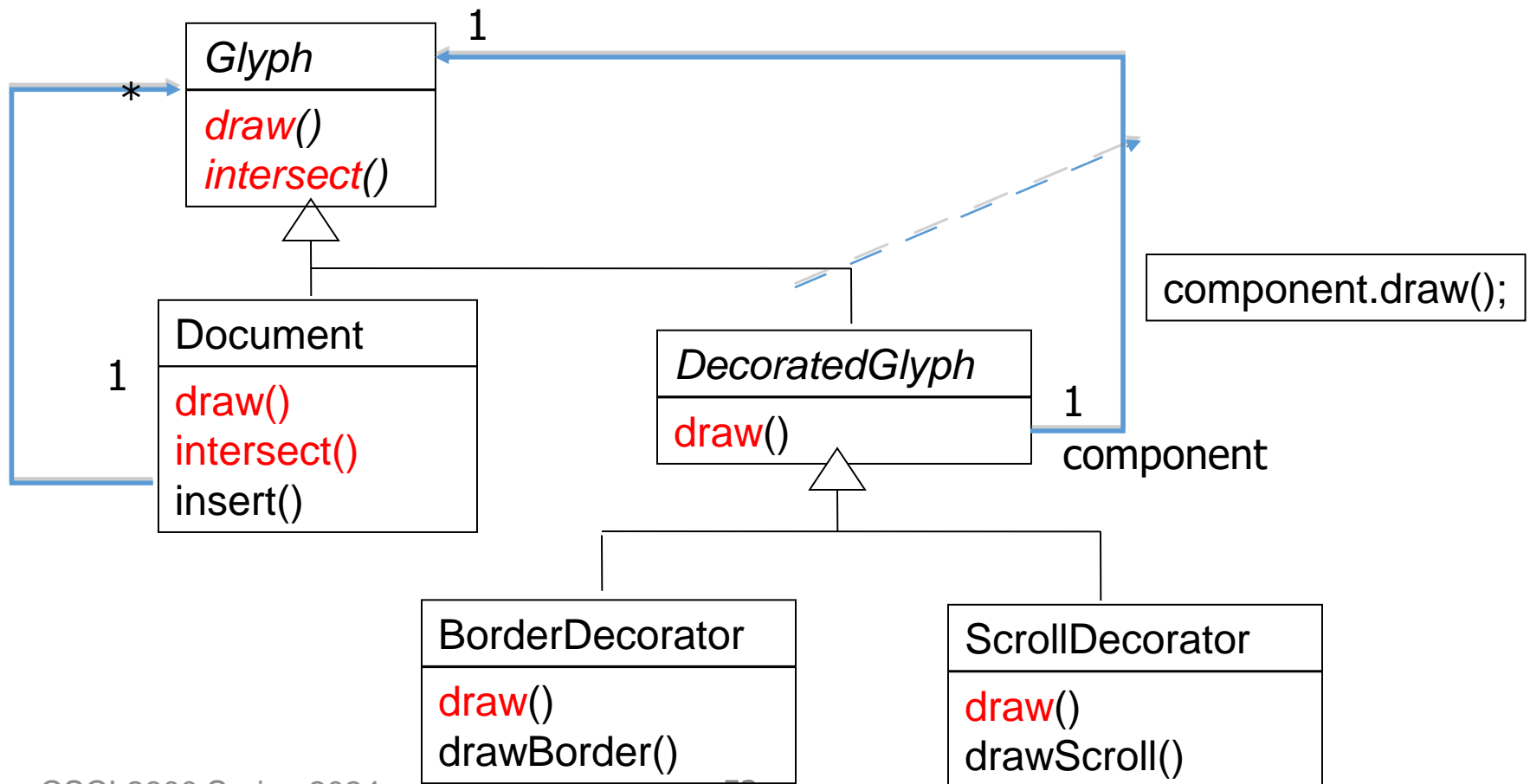
FunStrategyDemo.java

Embellishing the UI

- We would like to embellish the document display. One embellishment adds a border around the document area, another one adds a horizontal scroll bar and a third one adds a vertical scroll bar
- We would like to do this dynamically --- one can create any combination of embellished documents
- What pattern?

The Decorator Pattern

- Adds functionality, preserves interface

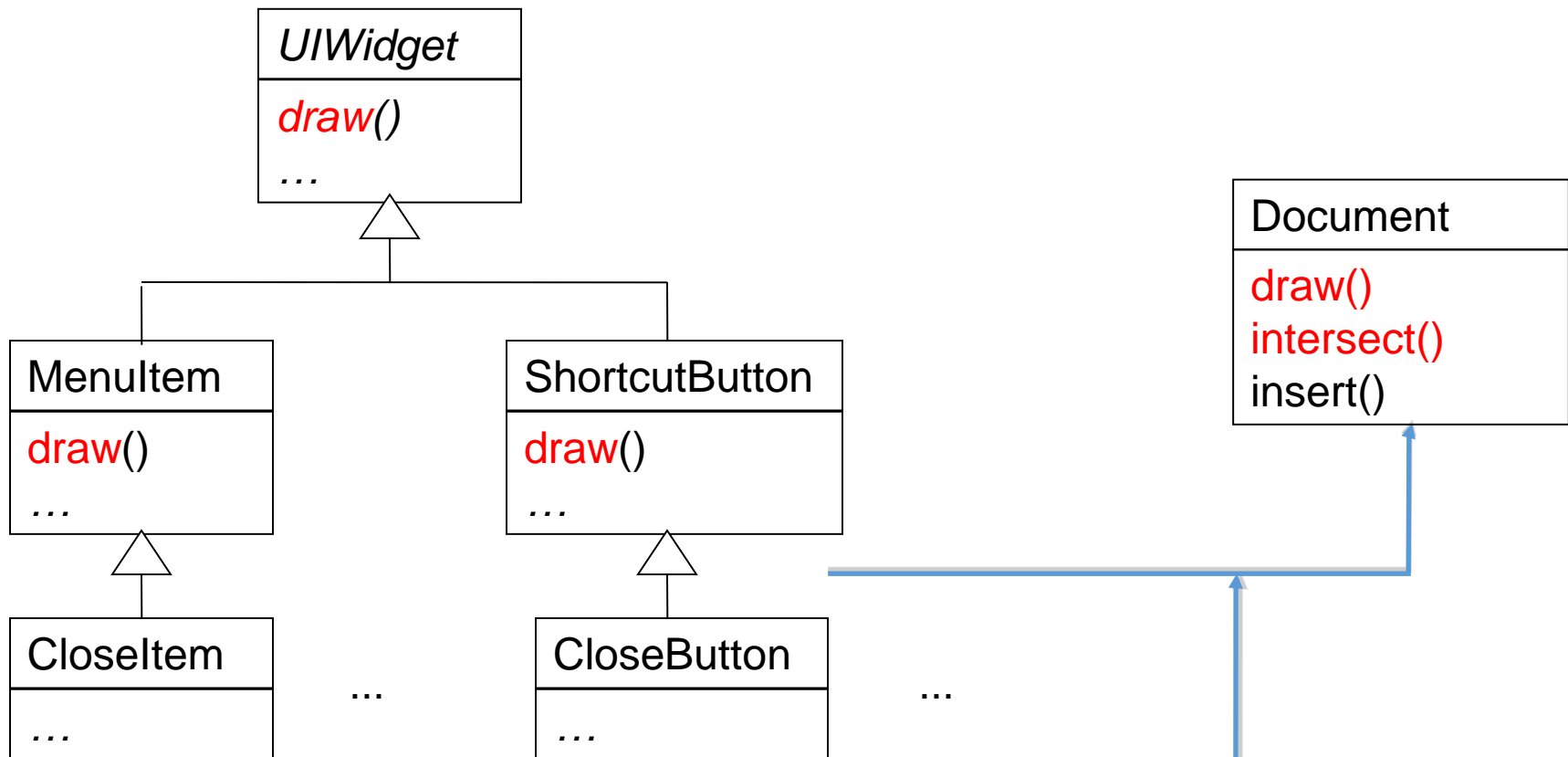


User Commands

- Editor supports many user “commands”: open and close, cut and paste, etc.
- There is different user interface for the same command
 - E.g., close document through a pull-down menu item, close button, key shortcut, other
- Supports undo and redo!

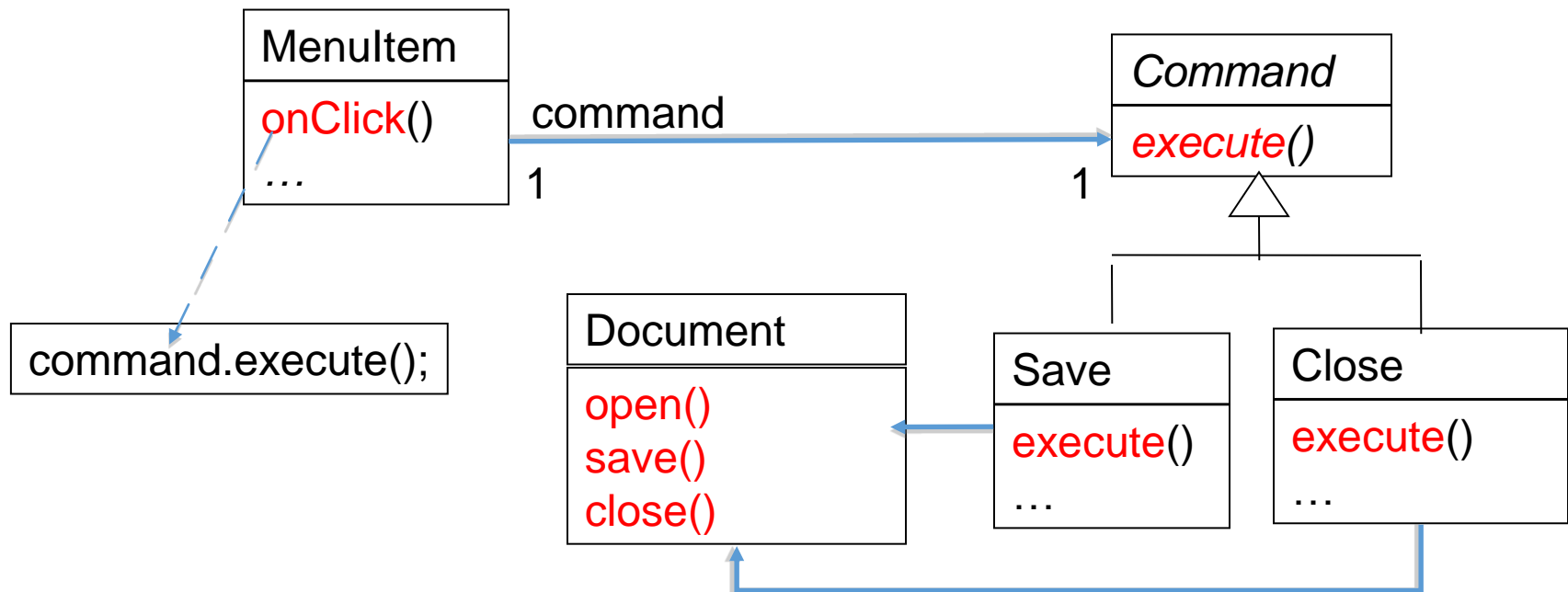
A Naïve Design

- What's wrong with this design?



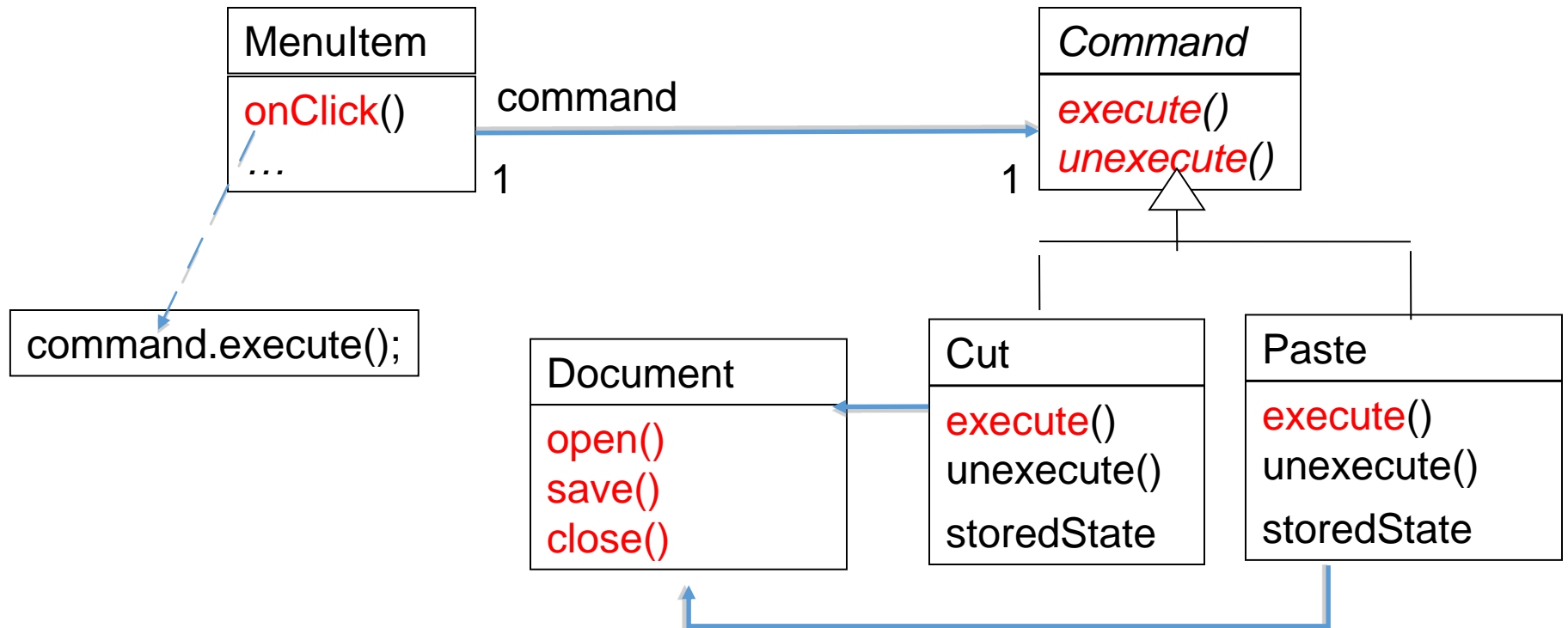
The Command Pattern

- Separates MenuItems from Commands that do the work



```
... = new MenuItem("Save", new Save(document));
... = new MenuItem("Close", new Close(document));
...
... = new ShortcutButton("Save", new Save(document));
```


Easy to Add Undo/Redo!



- Editor maintains a history (e.g., a stack) of commands that have been executed

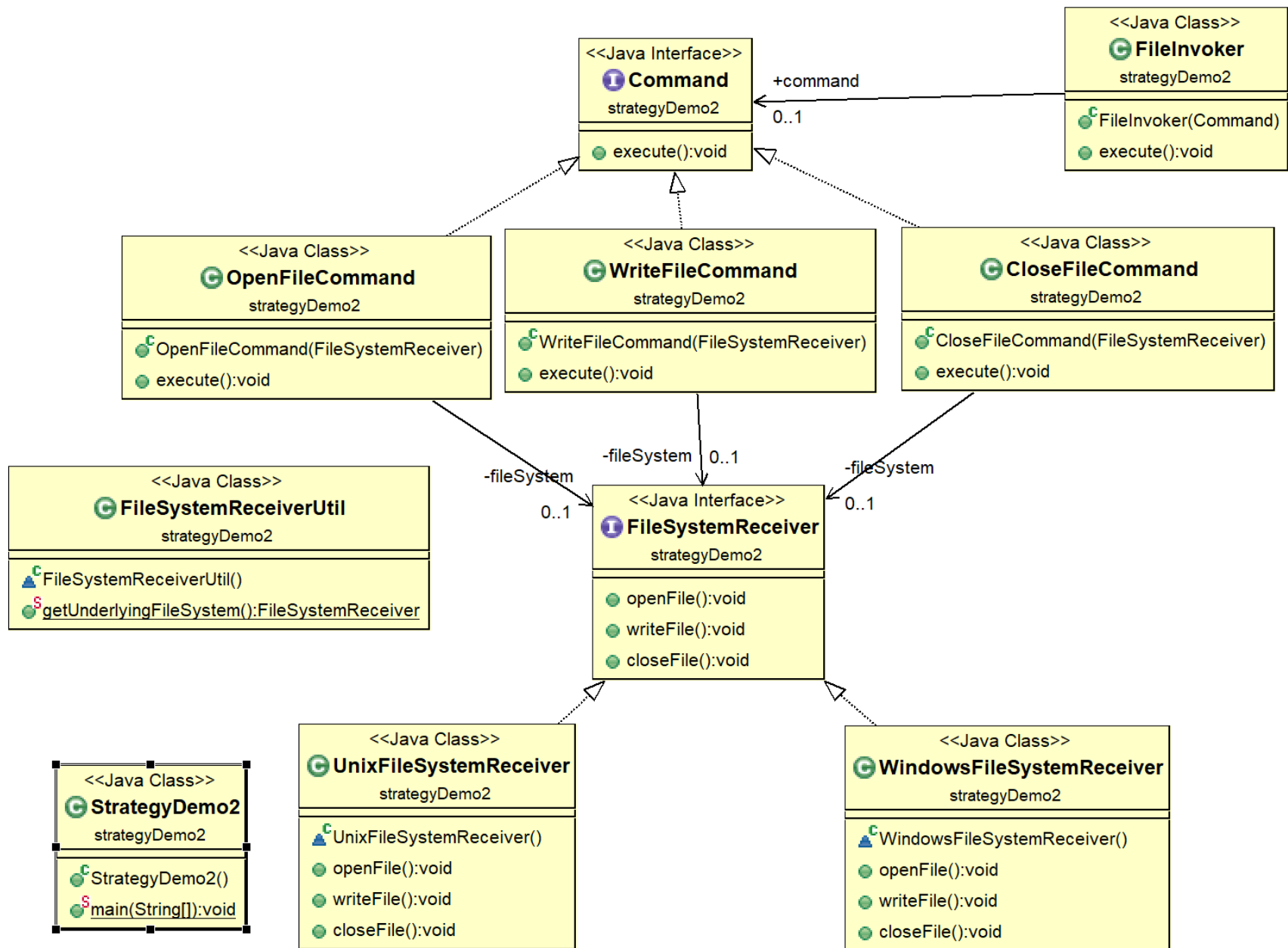
Aside: Command Pattern



- Command pattern is a data driven design pattern
 - behavioral pattern
- A request is wrapped in an object and passed to an invoker object.
- Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

Command Pattern

- In the Command Pattern, the request is sent to the invoker object and the invoker passes it to the encapsulated command object.
- The client creates the receiver object and then attaches it to the Command.
- Then it creates the invoker object and attaches the command object to perform an action.



CommandDemo.java

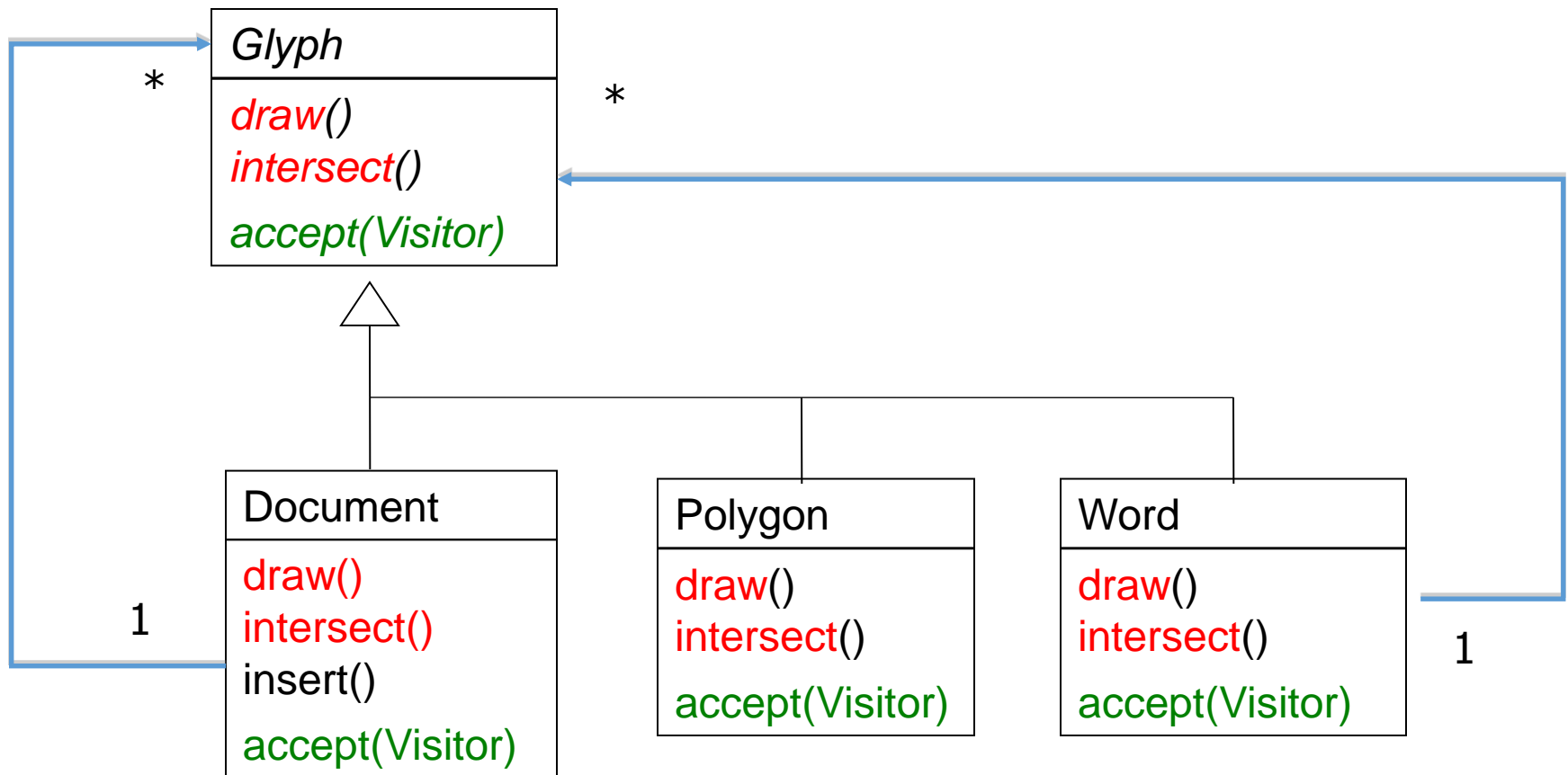
Command Pattern

- Command class is the core of the command design pattern. It defines the contract for implementation.
- Receiver implementation is separate from command implementation.
- Command implementation class chooses the method to invoke on receiver object, for every method in receiver there will be a command implementation. It works as a bridge between receiver and action methods.
- Invoker class just forwards the request from client to the command object.
- Client is responsible for instantiating the appropriate command and receiver implementations and associating them.
- Client is also responsible for instantiating the invoker object and associating the command object with it and executing the action method.
- Command design pattern is easily extendible, we can add new action methods in receivers and create new Command implementations without changing the client code.
- The drawback is that the code can get huge and confusing with high number of action methods and because of so many associations.

Adding Spell Checking

- Requires traversal of document hierarchy
- We want to avoid writing this functionality into the document structure
- We would like to add other traversals in the future, e.g., search, word count, hyphenation
- What pattern?

The Visitor Pattern



Design Patterns so Far

- **Factory method, Factory class, Prototype**
 - Creational patterns: address problem that constructors can't return subtypes
- **Singleton, Interning**
 - Creational patterns: address problem that constructors always return a new instance of class
- **Wrappers: Adapter, Decorator, Proxy**
 - Structural patterns: when we want to change interface or functionality of an existing class, or restrict access to an object

Design Patterns so Far

- **Composite**
 - A structural pattern: expresses whole-part structures, gives uniform interface to client
- **Interpreter, Procedural, Visitor**
 - Behavioral patterns: address the problem of how to traverse composite structures
- **Observer**
 - Model-View
 - Model-View-Controller
 - here is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically

Design Patterns So Far

- Façade

- we need a subset of the functionality of a powerful, extensive and complex library
- Somewhat like proxy

- Strategy

- create objects which represent various strategies and a context object whose behavior varies as its strategy object.

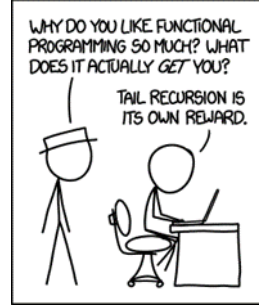
- Command

- Request are passed to an invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

Are Design Patterns The Last Word in Software Design?

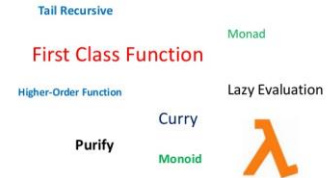
- Design Patterns are elements of reusable object-oriented software
- There are other programming approaches
 - Procedural
 - Functional
- Is the world made of nouns or verbs?
- Object oriented programming is about nouns
- Procedural programming is about verbs
 - Imperative programming
 - Do this, do this, etc.,
- Functional programming is about nouns and verbs
 - Verbs (functions) are *first class objects*
 - Many of the GoF patterns are part of the language
 - Pure functional programming is stateless

Imperative vs Functional Programming



	Imperative approach	Purely Functional approach
Programmer focus	How to perform tasks (algorithms) and how to track changes in state.	What information is desired and what transformations are required.
State changes	Important.	Non-existent.
Order of execution	Important.	Low importance.
Primary flow control	Loops, conditionals, and function (method) calls.	Function calls, including recursion.
Primary manipulation unit	Instances of structures or classes.	Functions as first-class objects and data collections.

Functional Programming in Java



- Java 8 added functional programming constructs.
- For example, `java.util.Function<T, R>` interface and lambda
- Example: Factory function
 - `FunBicycleFactory.java`
- Example: Lambda functions
 - `LambdaDemo.java`

Functional Strategy Example

```
package funStrategyDemo;

@FunctionalInterface
interface Strategy {
    double apply(double a, double b);
};

public class FunStrategyDemo {
    public static double execute(double a, double b, Strategy strategy) {
        return strategy.apply(a, b);
    }

    public static void main(String[] args) {
        System.out.println(execute(3, 4, (a,b) -> a > b ? a : b ));
        System.out.println(execute(3, 4, (a,b) -> a < b ? a : b ));
        System.out.println(execute(3, 4, (a,b) -> (a + b) / 2 ));
    }
}
```

Functional version uses fewer lines of text.

Code intention is clearer.

No side effects. Context is not needed.