Design Patterns, cont.

Outline of Today's Class

- Behavioral patterns
  - Observer
  - Façade
- Dependences and coupling

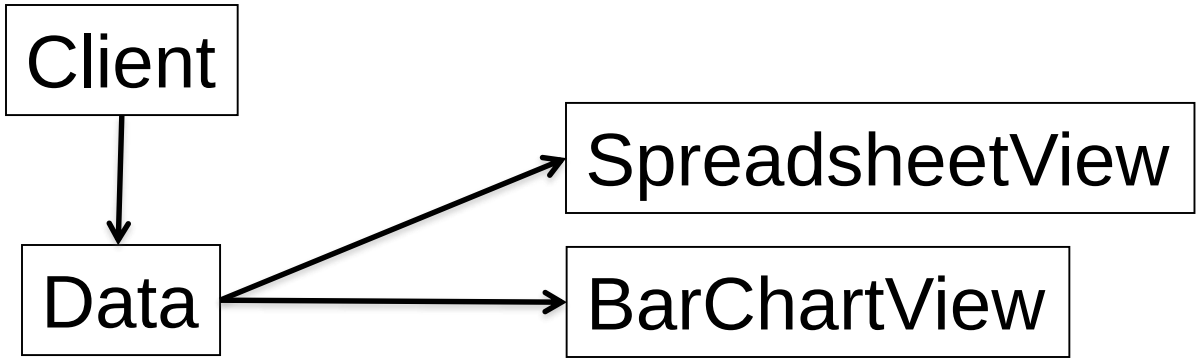
Observer Pattern

- 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`** 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`**!

A Better Design

- Data class has minimal interaction with Views
  - Only needs to update Views when it changes

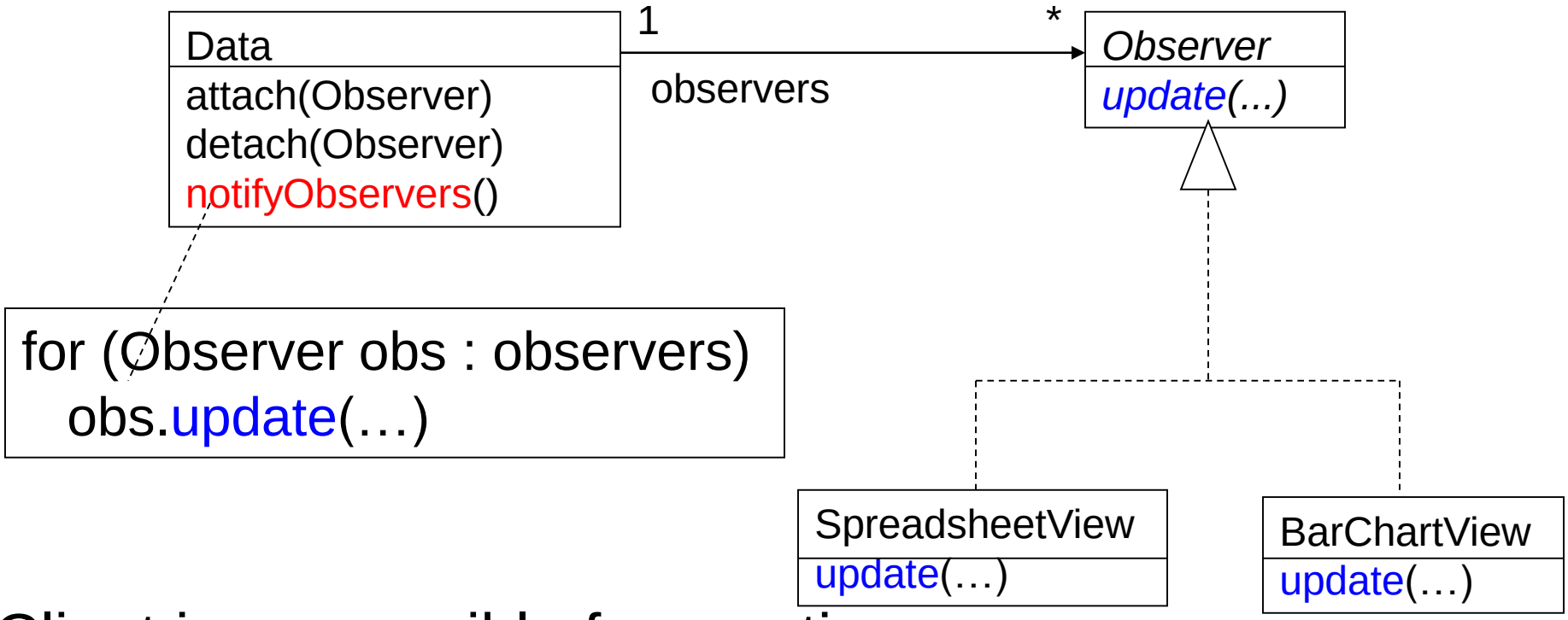
Old, naive design:

```
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



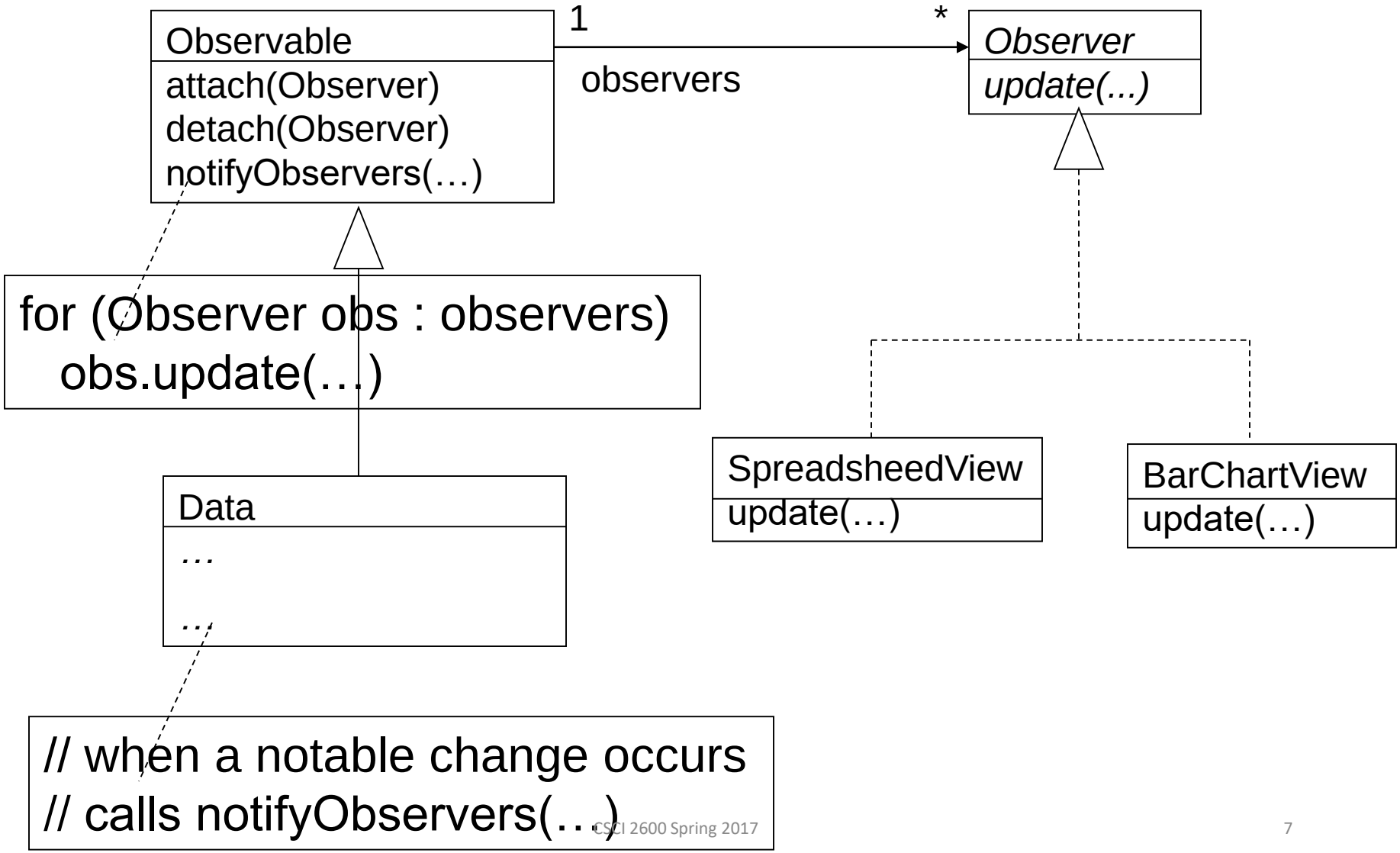
Client is responsible for creation:

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

Data keeps list of Views, notifies them when change.

Data is minimally connected to Views!

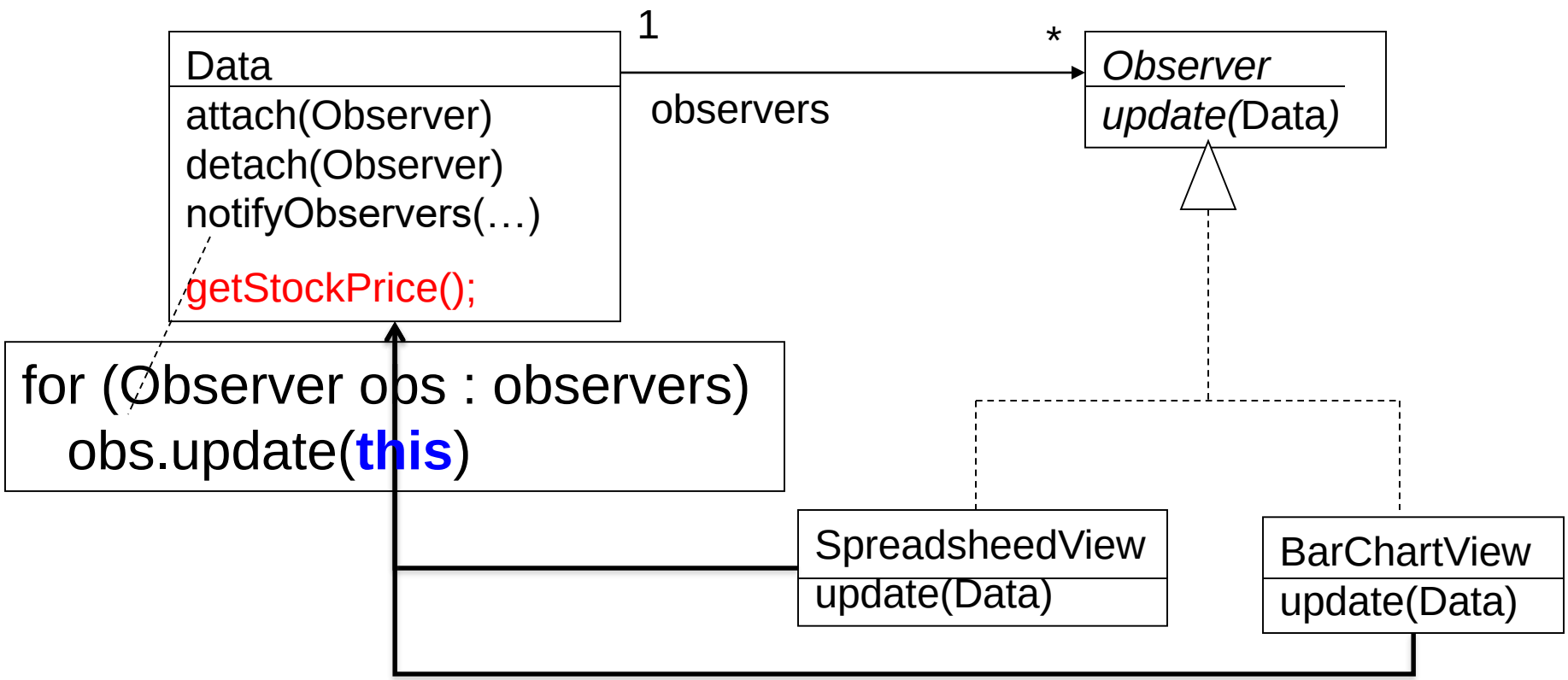
Even Better


Observable
attach(Observer)
detach(Observer)
notifyObservers(…)
1
*
observers
Observer
update(...)
for (Observer obs : observers)
obs.update(…)
Data
…
…
SpreadsheedView
update(…)
BarChartView
update(…)
// when a notable change occurs
// calls notifyObservers(…)

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.

Example of Observer

From JDK

```
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)`.

THE MODEL

An Observer of Name Changes (Push)

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

  public void update(Observable obj, Object arg) {
   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!

THE VIEW

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) and 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!");
   }
}
```

We don’t care for this now.

ANOTHER VIEW

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 VIEW

The Client

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

// Now add observers
si.addObserver(nameObserver);
si.addObserver(priceObserver);

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

JDK. Since s is Observable!

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 addStrudent(String student) {
        students.add(student);
        notifyObservers();
    }
    public int size() {
        return students.size();
    }
}
```


THE MODEL

Example of Observer

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

```
public class SignupObserver extends 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());
    }
}
```

We don't care for arg now.

THE VIEW

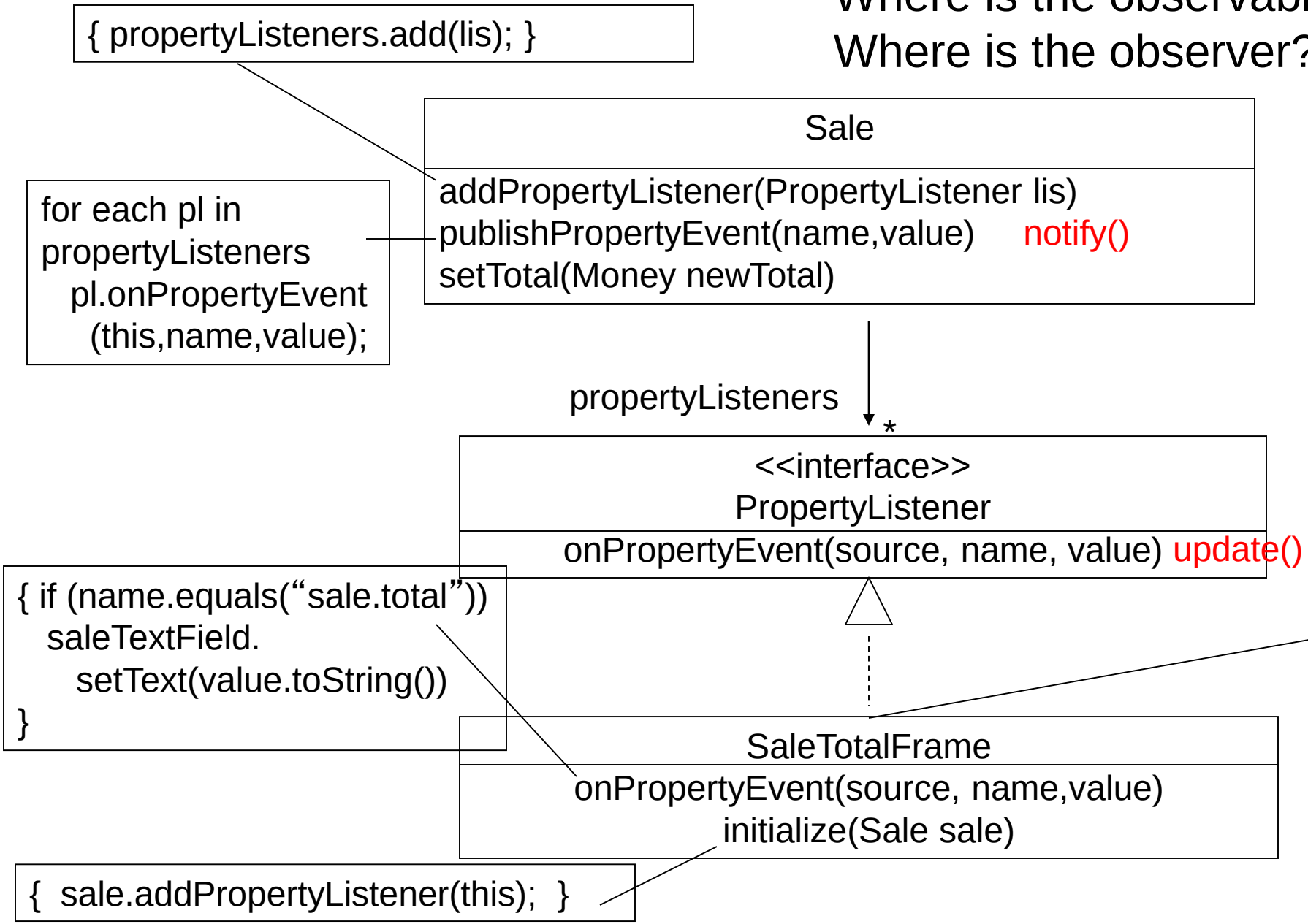
The Client

```
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?

THE CONTROLLER

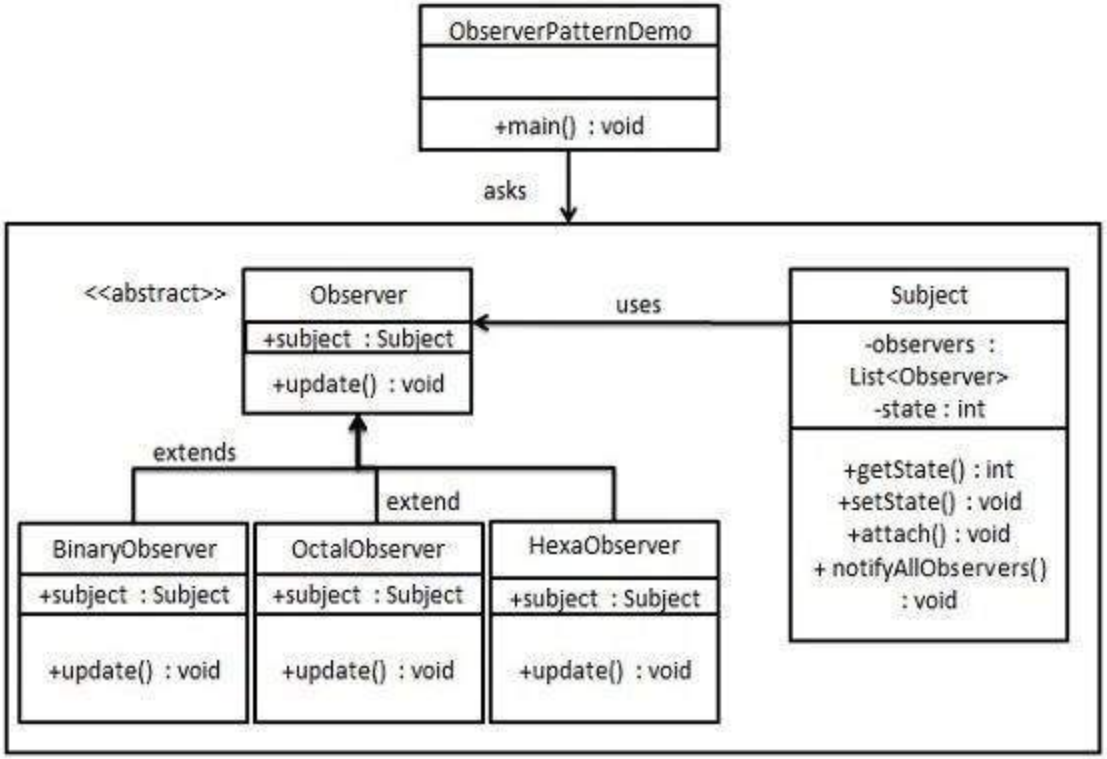
Where is the observable?
Where is the observer?



Model-view Principle

- Observer pattern known as Model-view or Model-view-controller
- "Model" objects (e.g., Sale, SignupSheet) should not know 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 Pattern – simple complete example



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

```
public class Subject {

  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);
  }

  @Override
  public void update() {
    System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );
  }
}
// similar code for Octal and HexObserver
```

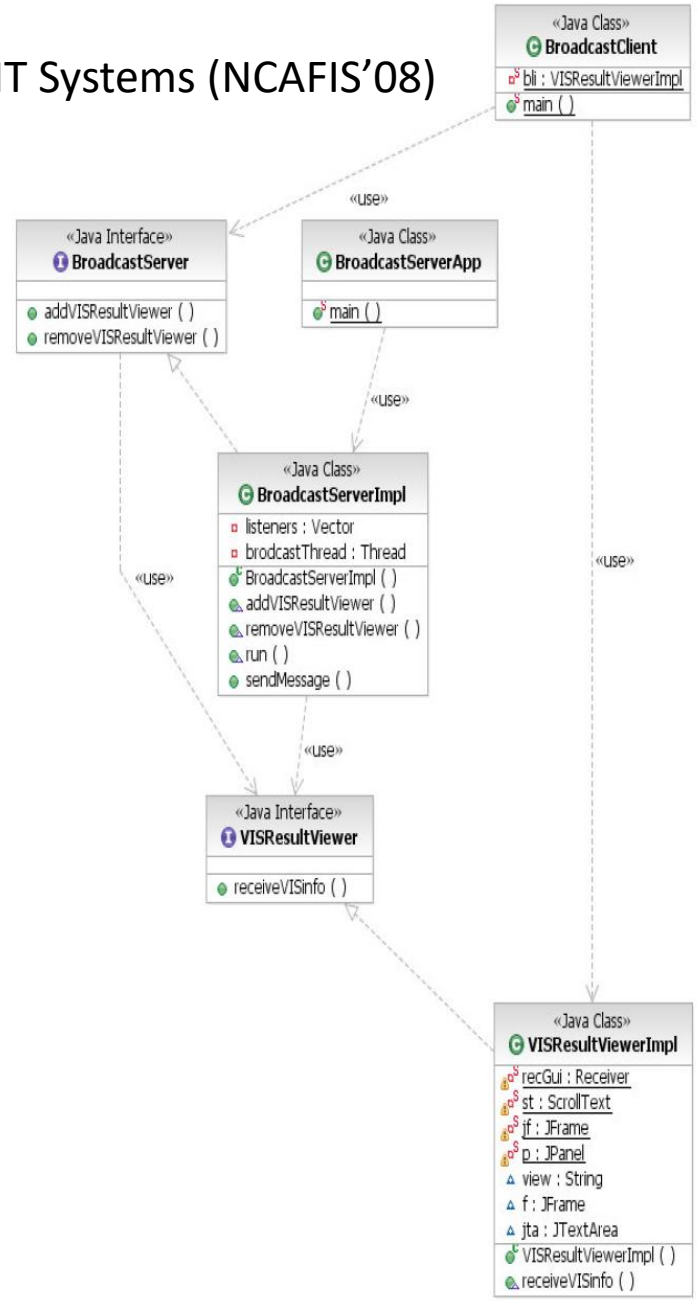
```
public class ObserverPatternDemo {
  public static void main(String[] args) {
    Subject subject = new Subject();

    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
```

E-Governance Solution Based on Observer Design Pattern

Ajay Parikh, Bharat V. Buddhdev

National Conference on Architecturing Future IT Systems (NCAFIS'08)



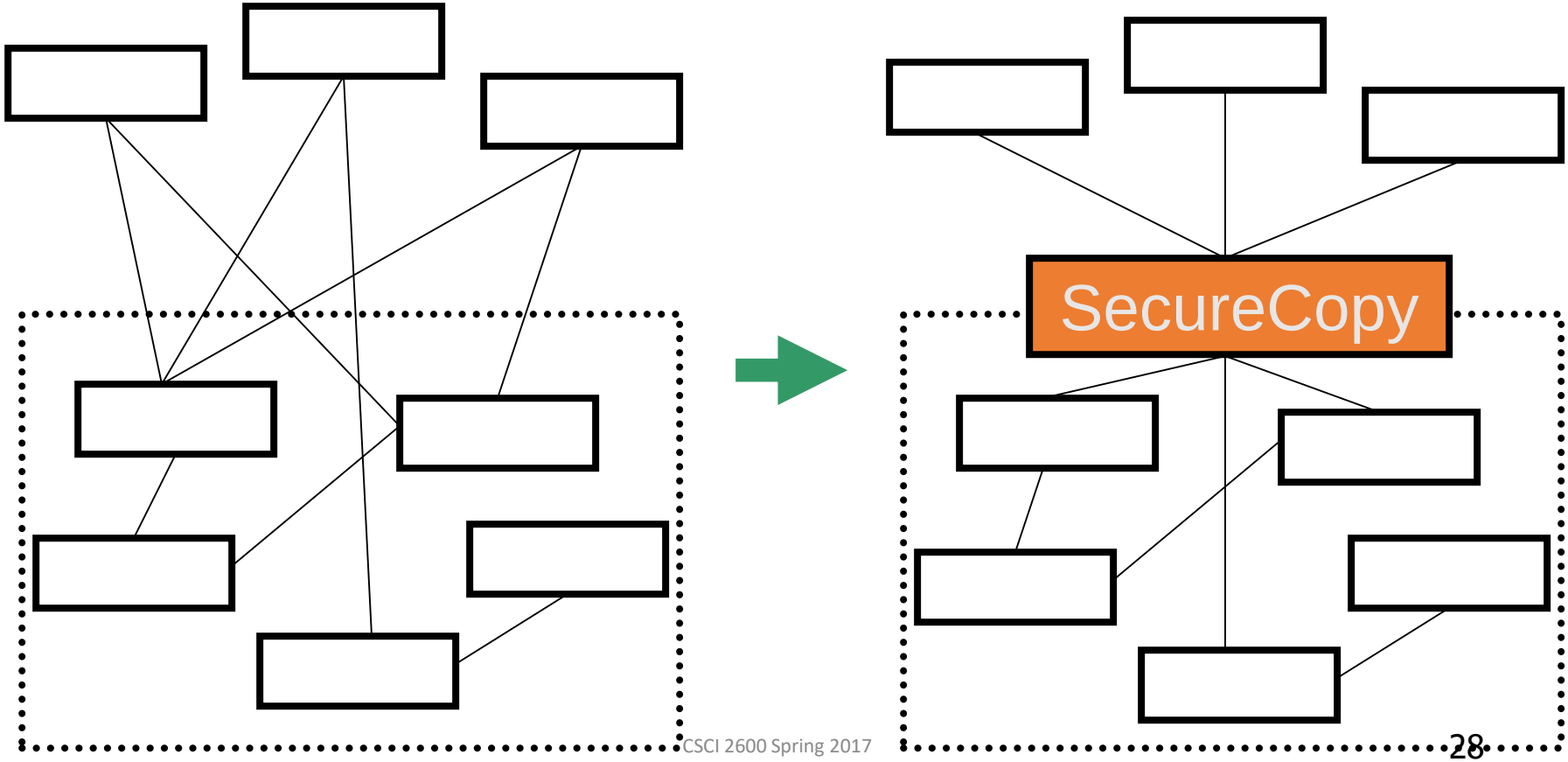
Façade Pattern

- 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?

Façade Pattern

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



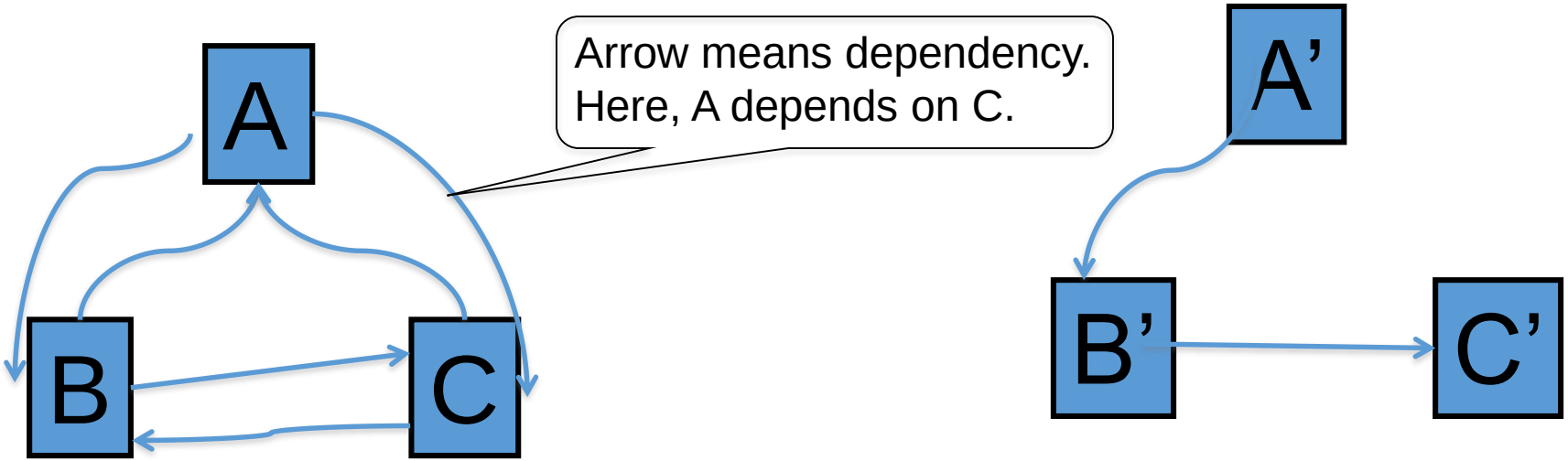
Façade Pattern

- 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

Interactions Between Modules

- Interactions between modules (in our designs, module = class) cause complexity
- To simplify, split design into parts that don't interact much
- 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



Bad Design:
Classes strongly coupled

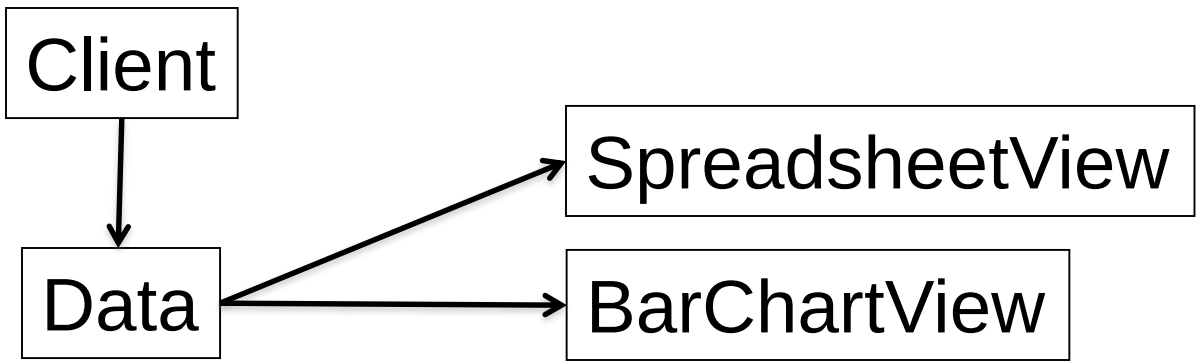
Better Design:
Weaker coupling

Coupling is the Path to the Dark Side

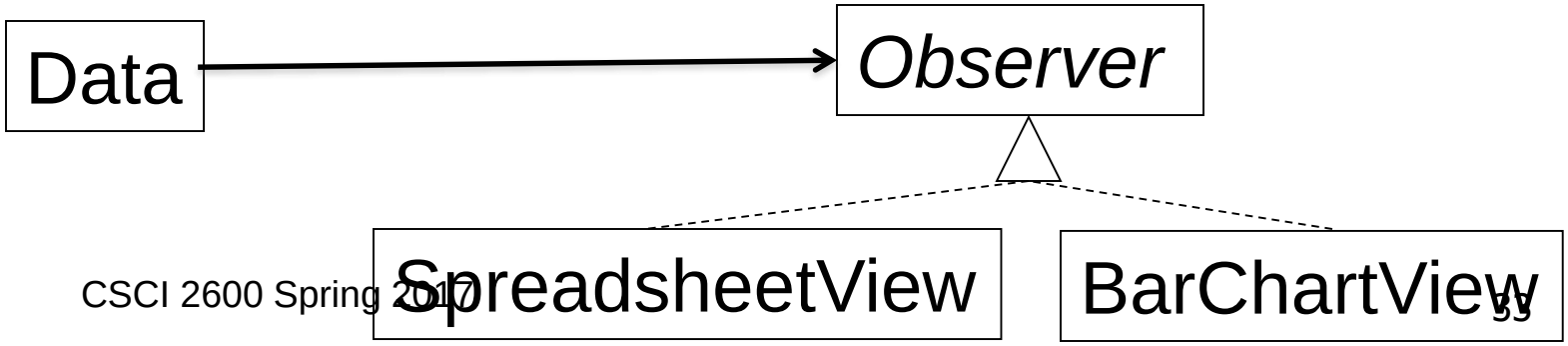
- Coupling leads to complexity
- Complexity leads to confusion
- Confusion leads to suffering
- If once you start down the dark path, forever will it dominate your destiny, consume you it will

Observer promotes low coupling

- Bad. Data does not need to depend on Views

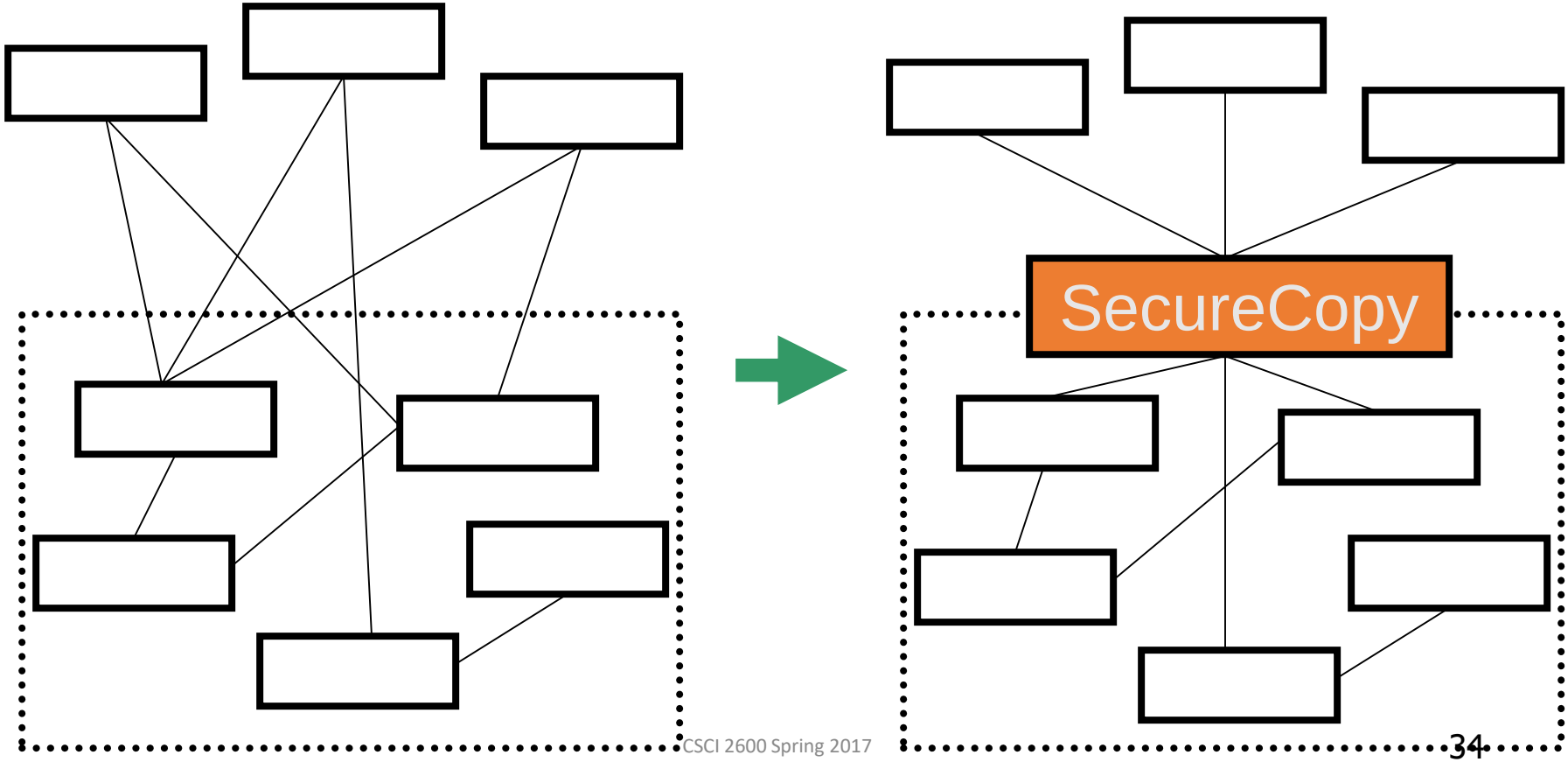


- Better: Weaken dependency of Data on Views
  - Introduce a weaker spec in the form of interface



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
- Neither content or appearance is easily understood or changed
- Destroys flexibility
- Leads to subtle bugs
- Callbacks, listeners, and other patterns can help

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.

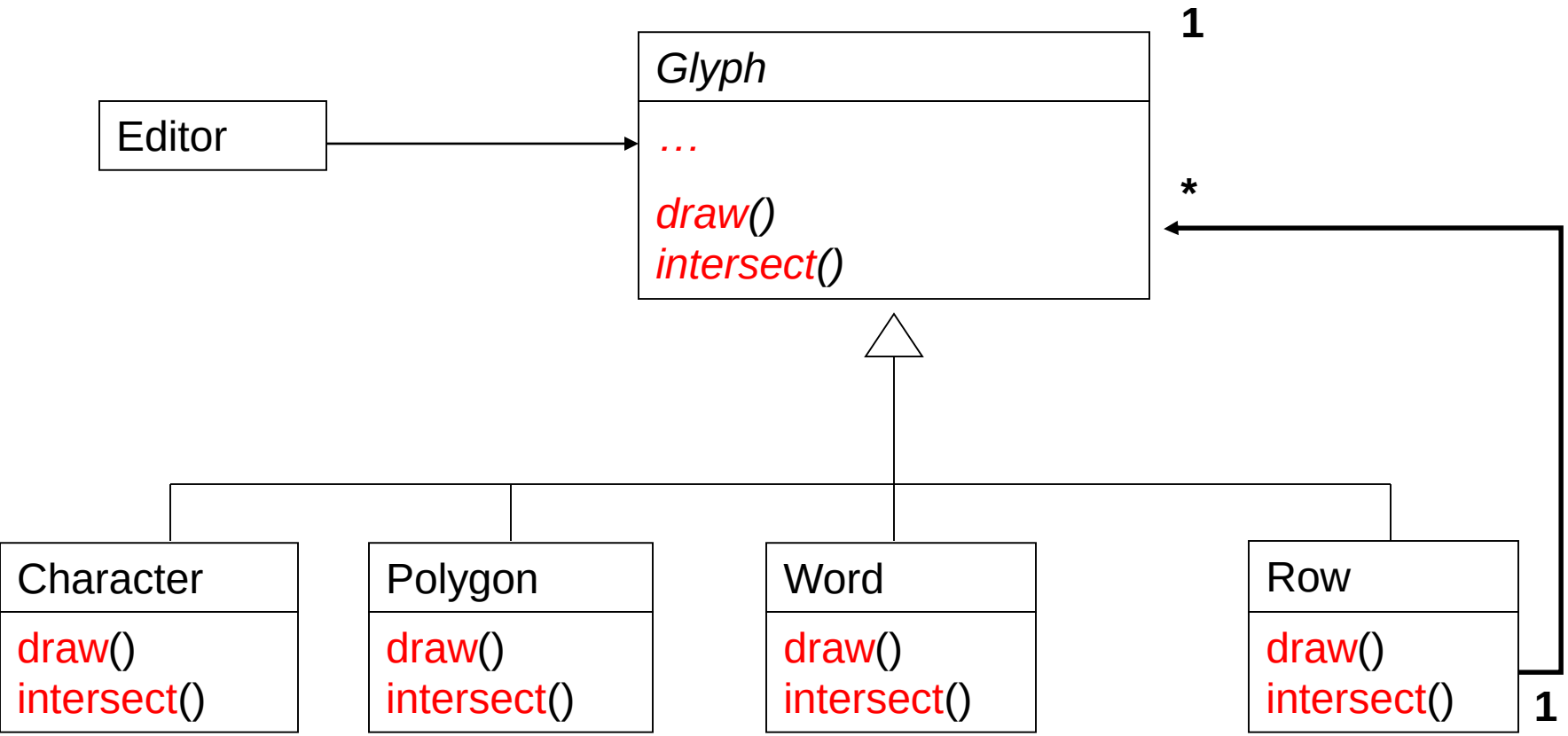
- Structure, Formatting, Embellishing the UI, User commands, Spell checking

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



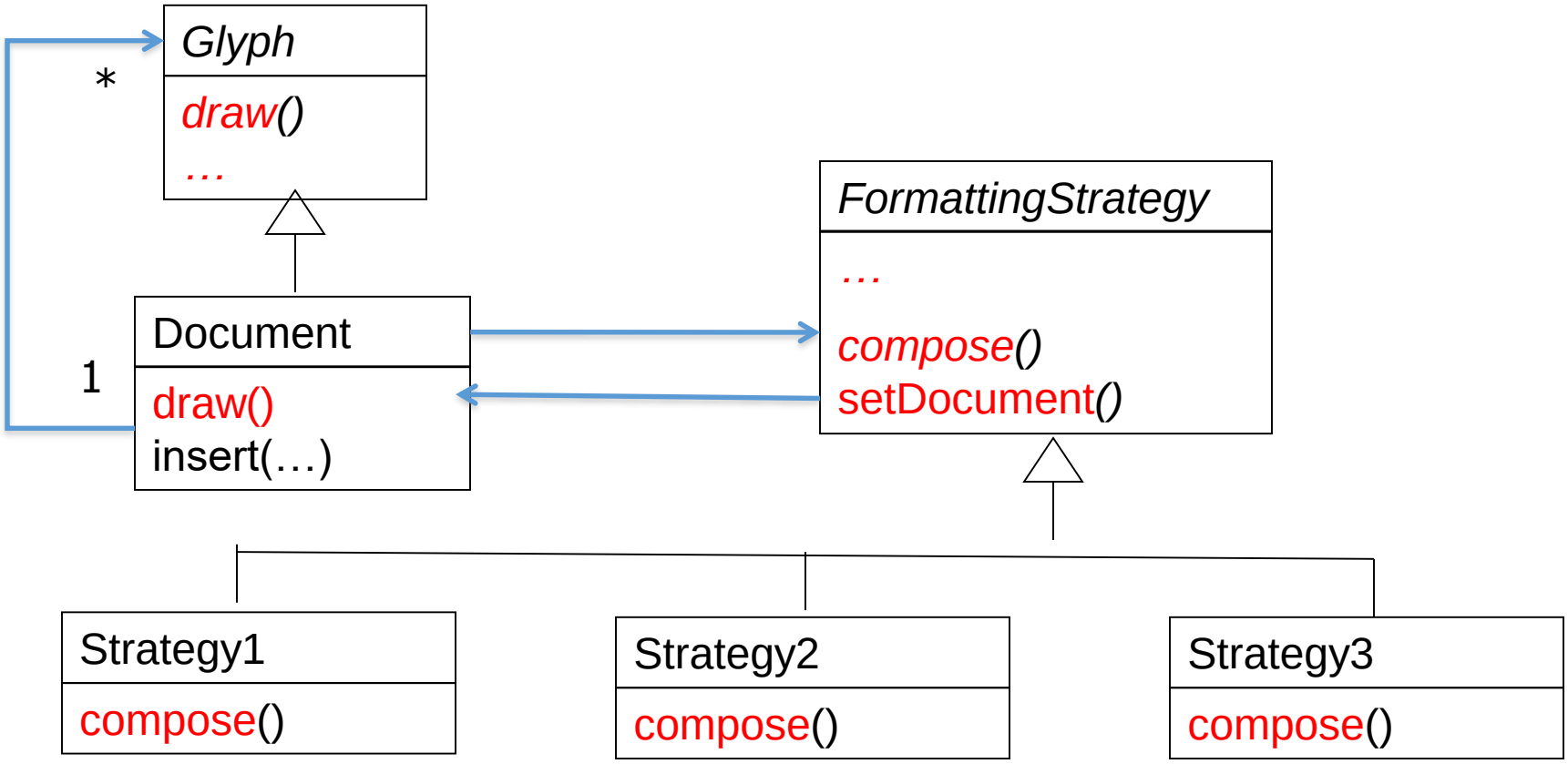
What's missing from this picture?

Formatting

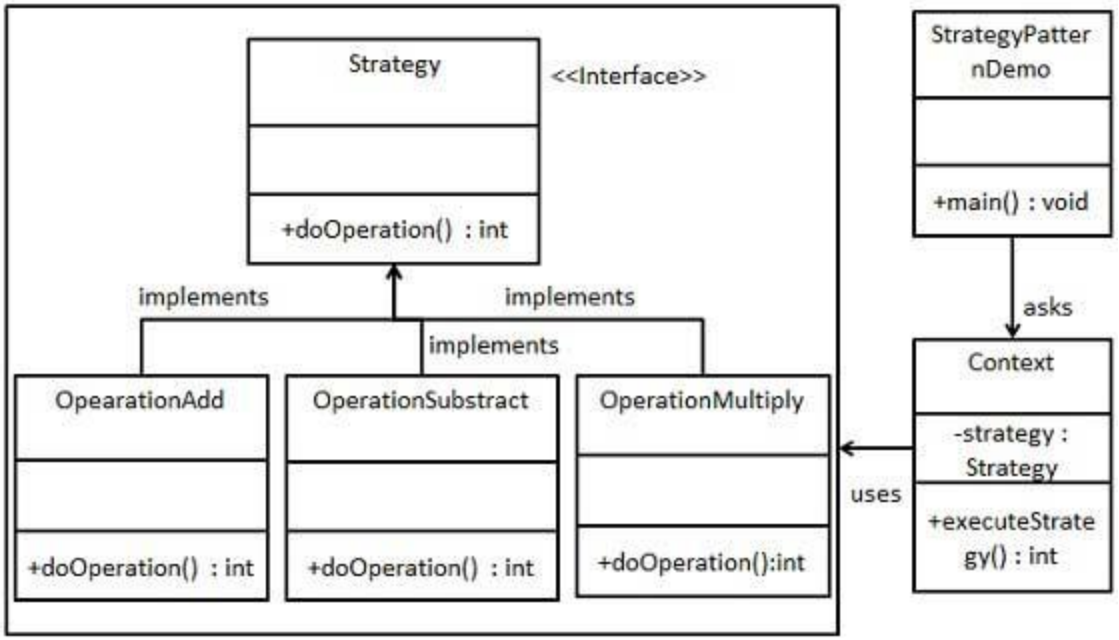
- Formatting displays the document
- Many different formatting strategies are possible
  - We would use <u>different</u> formatting strategies over the <u>same</u> hierarchical structure
- Each formatting strategy is complex

The Strategy Pattern

- Encapsulates an algorithm in an object



Aside: A simple Strategy Pattern Example



```
 public interface Strategy {
   public int doOperation(int num1, int num2);
 }

public class OperationAdd implements Strategy{
  @Override
  public int doOperation(int num1, int num2) {
    return num1 + num2;
  }
}
public class OperationSubstract implements Strategy{
  @Override
  public int doOperation(int num1, int num2) {
    return num1 - num2;
  }
}
public class OperationMultiply implements Strategy{
  @Override
  public int doOperation(int num1, int num2) {
    return num1 * num2;
  }
}
```

```
public class Context {
  private Strategy strategy;

  public Context(Strategy strategy){
    this.strategy = strategy;
  }

  public int executeStrategy(int num1, int num2){
    return strategy.doOperation(num1, num2);
  }
}
```

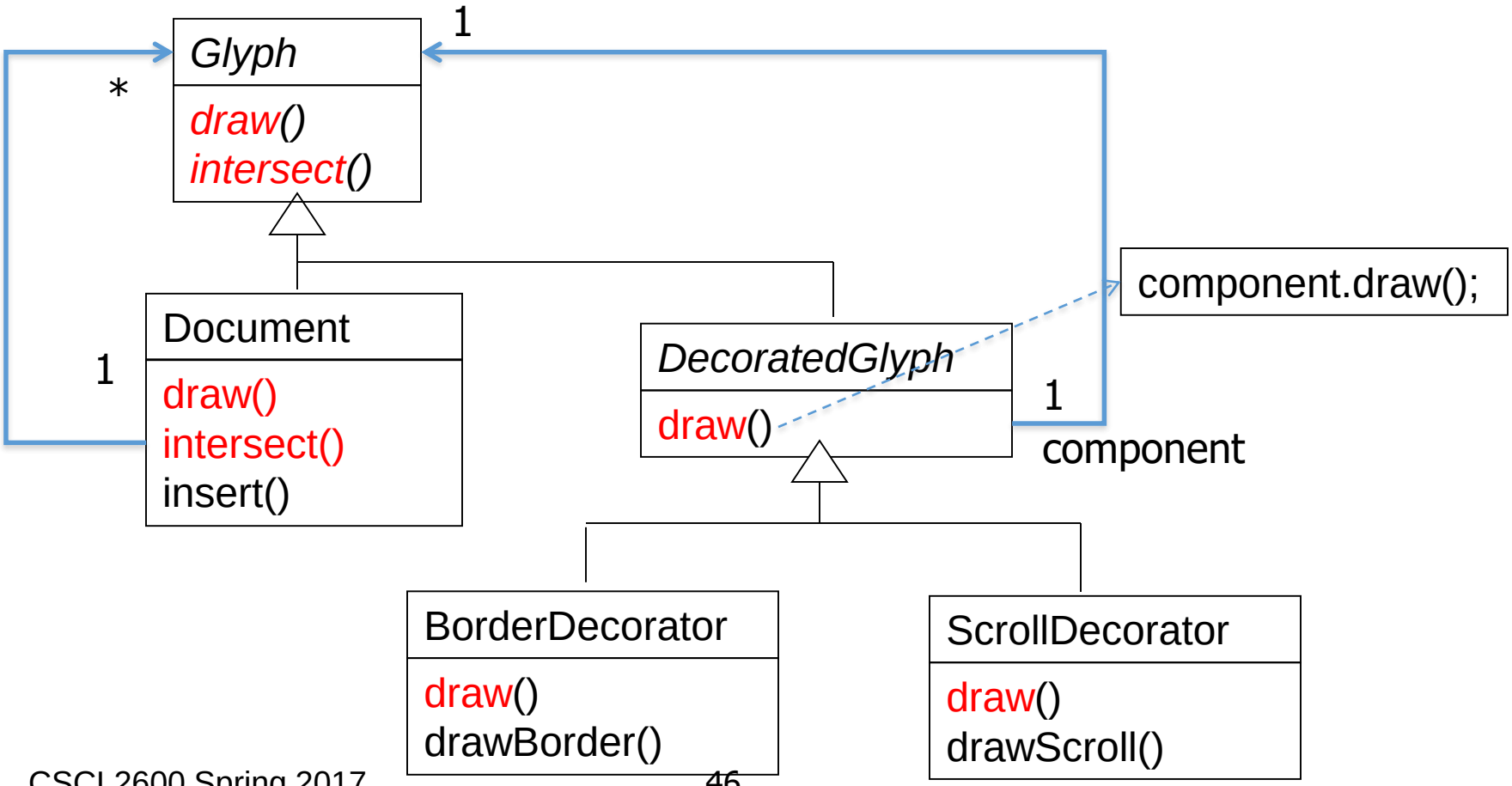
```
public class StrategyPatternDemo {
  public static void main(String[] args) {
    Context context = new Context(new OperationAdd());
    System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

    context = new Context(new OperationSubstract());
    System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

    context = new Context(new OperationMultiply());
    System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
  }
}
```

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



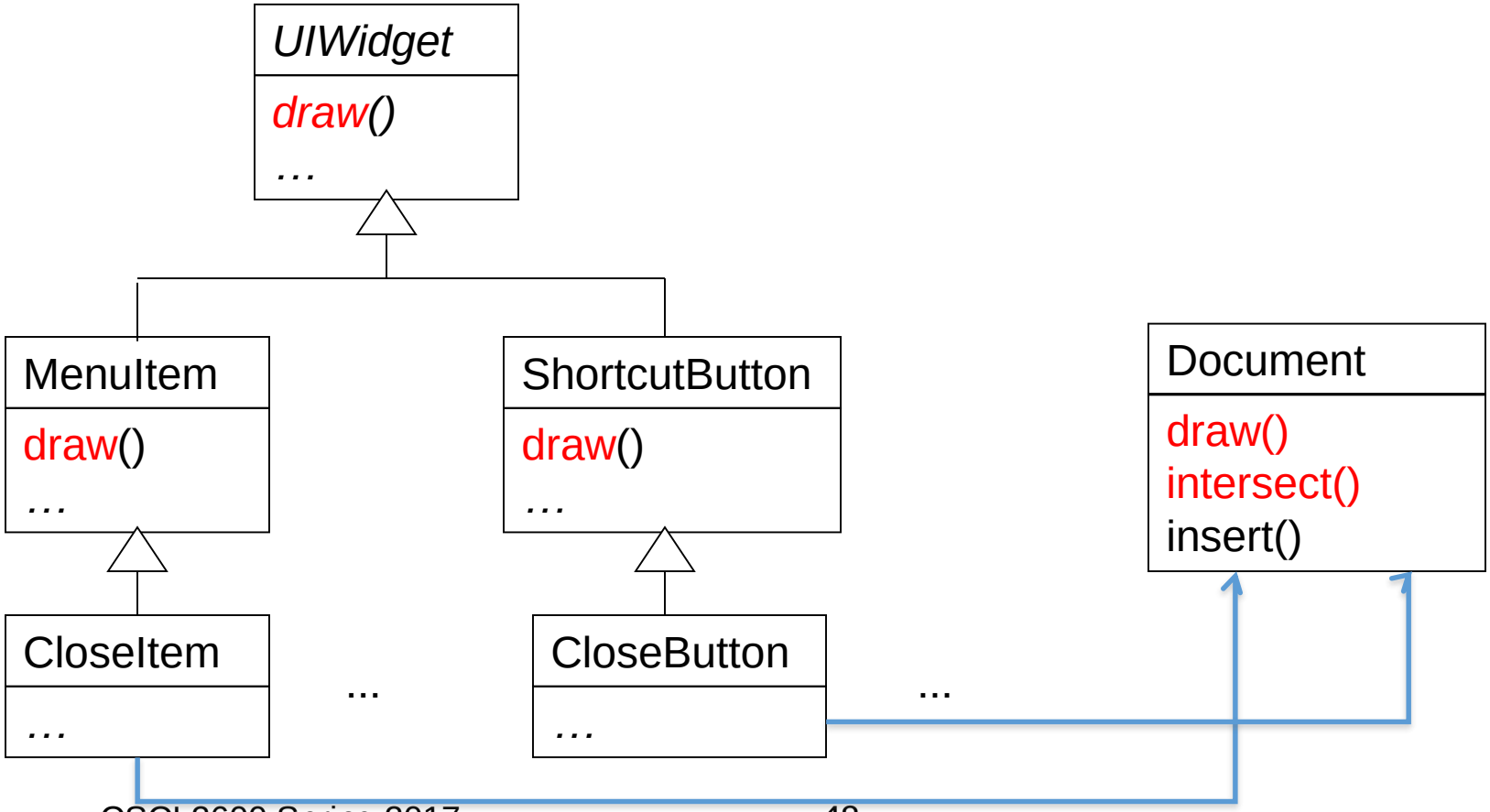
User Commands

- 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!

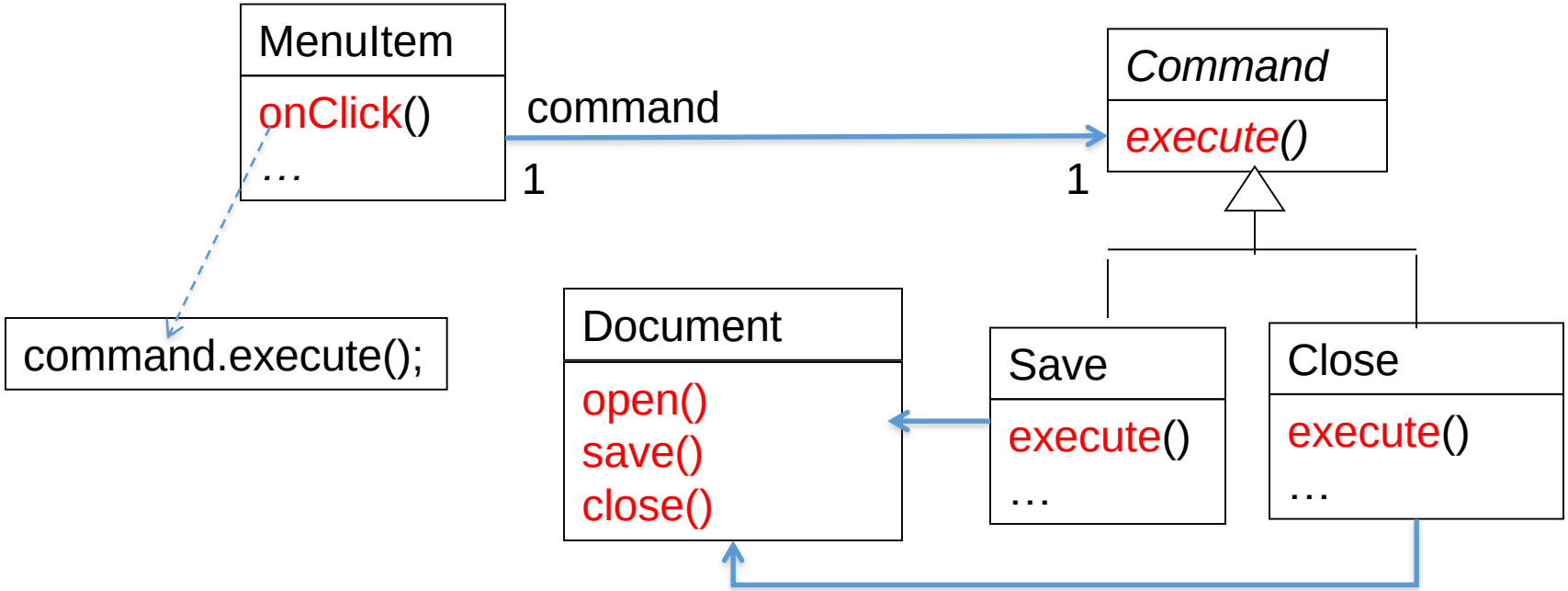
A Naïve Design

- What's wrong with this design?



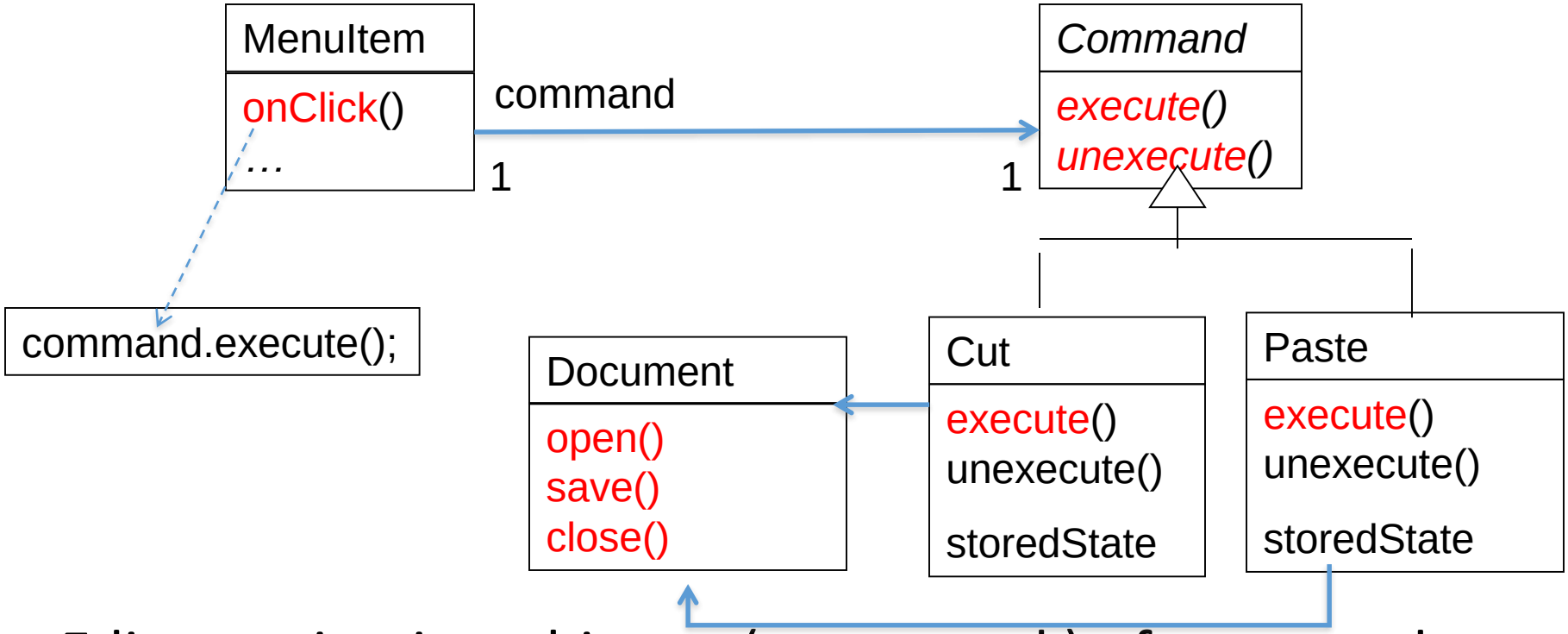
The Command Pattern

- Separates MenuItems from Commands that do the work



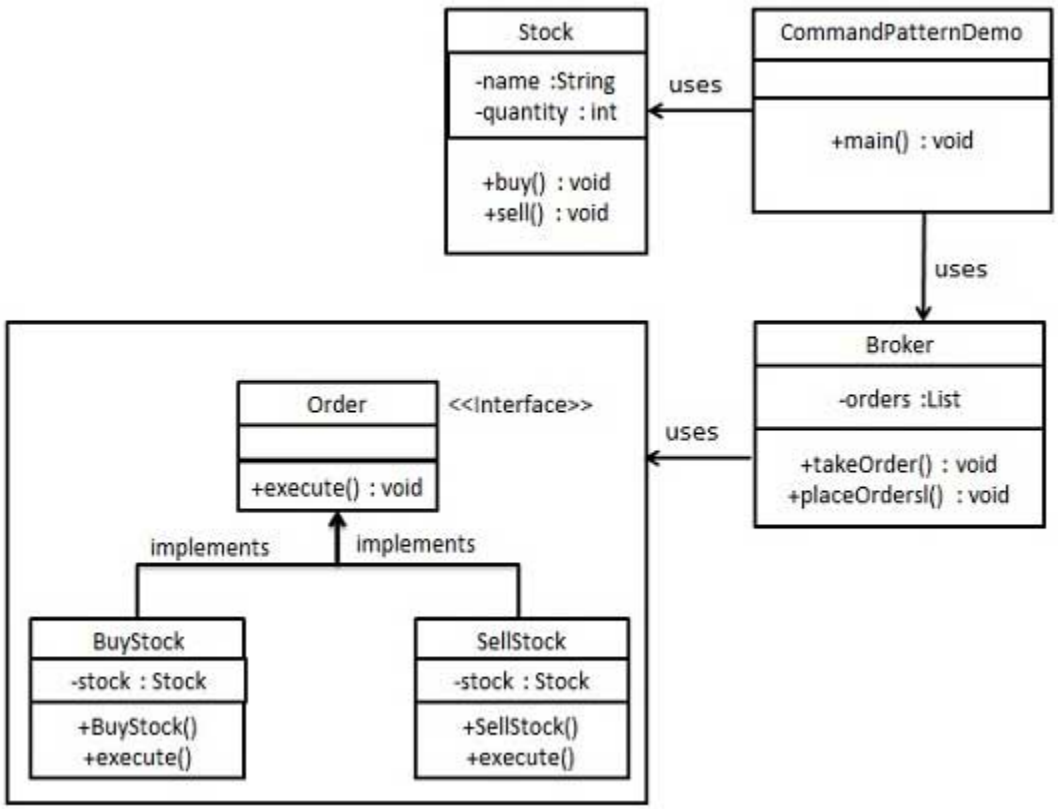

```
… = new MenuItem("Save", new Save(document));
… = new MenuItem("Close", new Close(document));
…
… = new ShortcutButton("Save", new Save(document));
```

Easy to Add Undo/Redo!



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

Aside: Simple Command Design Pattern Example



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

```
public interface Order {
  void execute();
}

public class Stock {

  private String name = "ABC";
  private int quantity = 10;

  public void buy(){
    System.out.println("Stock [ Name: "+name+",
      Quantity: " + quantity +" ] bought");
  }
  public void sell(){
    System.out.println("Stock [ Name: "+name+",
      Quantity: " + quantity +" ] sold");
  }
}
```

```
public class BuyStock implements Order {
  private Stock abcStock;

  public BuyStock(Stock abcStock){
    this.abcStock = abcStock;
  }

  public void execute() {
    abcStock.buy();
  }
}

public class SellStock implements Order {
  private Stock abcStock;

  public SellStock(Stock abcStock){
    this.abcStock = abcStock;
  }

  public void execute() {
    abcStock.sell();
  }
}
```

```
public class Broker {
  private List<Order> orderList = new ArrayList<Order>();

  public void takeOrder(Order order){
    orderList.add(order);
  }

  public void placeOrders(){

    for (Order order : orderList) {
      order.execute();
    }
    orderList.clear();
  }
}
```

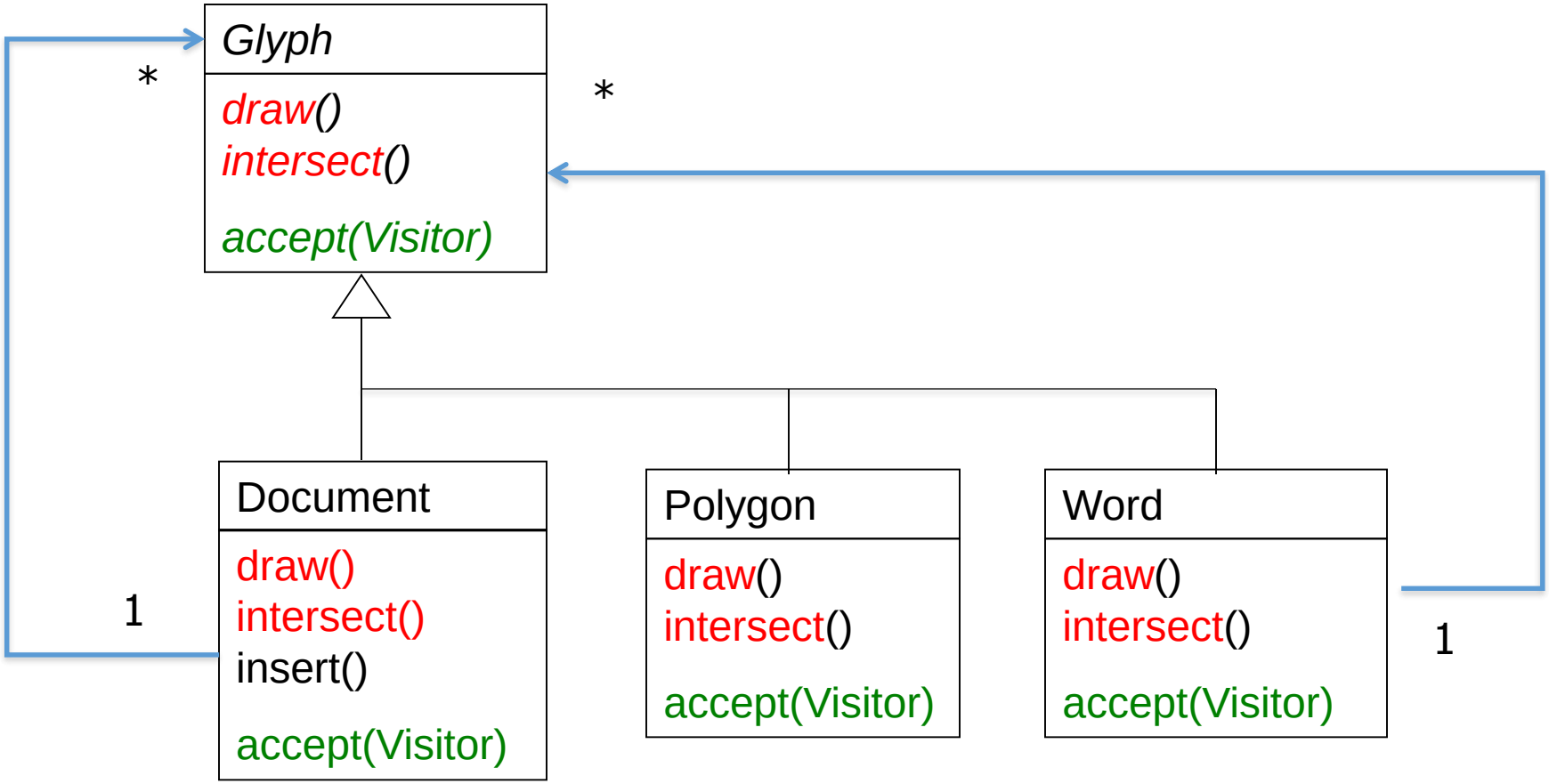
```
public class CommandPatternDemo {
  public static void main(String[] args) {
    Stock abcStock = new Stock();

    BuyStock buyStockOrder = new BuyStock(abcStock);
    SellStock sellStockOrder = new SellStock(abcStock);

    Broker broker = new Broker();
    broker.takeOrder(buyStockOrder);
    broker.takeOrder(sellStockOrder);

    broker.placeOrders();
  }
}
```

Output:
Stock [Name: ABC, Quantity: 10] bought
Stock [Name: ABC, Quantity: 10] sold

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 depenedent 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.