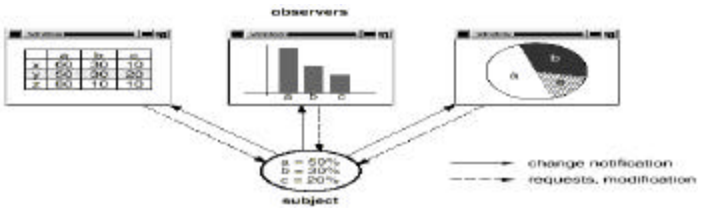
The Observer Pattern

The Observer Pattern

- Intent
  - ⇒ Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Also Known As
  - ⇒ Dependents, Publish-Subscribe, Model-View
- Motivation
  - ⇒ The need to maintain consistency between related objects without making classes tightly coupled

The Observer Pattern

- Motivation



The Observer Pattern

- Applicability

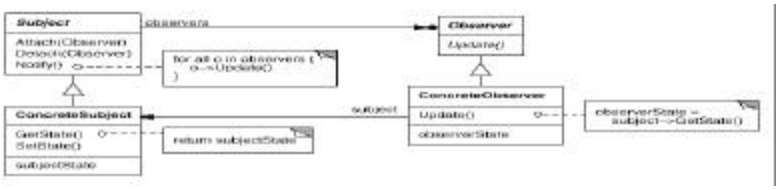
  Use the Observer pattern in any of the following situations:

  ⇒ When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.

  ⇒ When a change to one object requires changing others

  ⇒ When an object should be able to notify other objects without making assumptions about those objects

- Structure

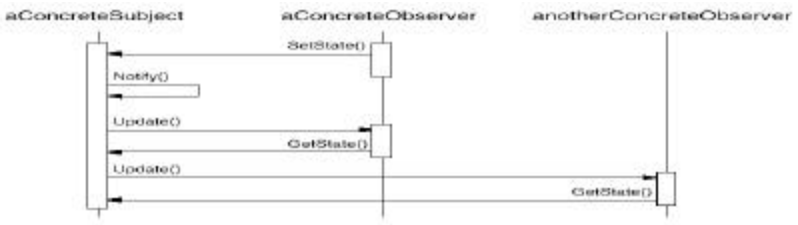


The Observer Pattern

- Participants
  - ⇒ Subject
    - ➡ Keeps track of its observers
    - ➡ Provides an interface for attaching and detaching Observer objects
  - ⇒ Observer
    - ➡ Defines an interface for update notification
  - ⇒ ConcreteSubject
    - ➡ The object being observed
    - ➡ Stores state of interest to ConcreteObserver objects
    - ➡ Sends a notification to its observers when its state changes
  - ⇒ ConcreteObserver
    - ➡ The observing object
    - ➡ Stores state that should stay consistent with the subject's
    - ➡ Implements the Observer update interface to keep its state consistent with the subject's

The Observer Pattern

- Collaborations



The Observer Pattern

- Consequences
  - ⇒ Benefits
    - ➙ Minimal coupling between the Subject and the Observer
      - Can reuse subjects without reusing their observers and vice versa
      - Observers can be added without modifying the subject
      - All subject knows is its list of observers
      - Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
      - Subject and observer can belong to different abstraction layers
    - ➙ Support for event broadcasting
      - Subject sends notification to all subscribed observers
      - Observers can be added/removed at any time

The Observer Pattern

- Consequences
  - ⇒ Liabilities
    - ➙ Possible cascading of notifications
      - Observers are not necessarily aware of each other and must be careful about triggering updates
    - ➙ Simple update interface requires observers to deduce changed item

The Observer Pattern

- Implementation Issues
  - ⇒ How does the subject keep track of its observers?
  - ⇒ What if an observer wants to observe more than one subject?
    - ➡ Have the subject tell the observer who it is via the update interface
  - ⇒ Who triggers the update?
    - ➡ The subject whenever its state changes
    - ➡ The observers after they cause one or more state changes
    - ➡ Some third party object(s)
  - ⇒ Make sure the subject updates its state before sending out notifications
  - ⇒ How much info about the change should the subject send to the observers?
    - ➡ Push Model - Lots
    - ➡ Pull Model - Very Little

The Observer Pattern

- Implementation Issues
  - ⇒ Can the observers subscribe to specific events of interest?
  - ⇒ Can an observer also be a subject?
  - ⇒ What if an observer wants to be notified only after several subjects have changed state?
    - ➡ Use an intermediary object which acts as a mediator
    - ➡ Subjects send notifications to the mediator object which performs any necessary processing before notifying the observers

The Observer Pattern

- Sample Code
  - ⇒ We'll see some Java soon!
- Known Uses
  - ⇒ Smalltalk Model/View/Controller user interface framework
    - ➡ Model = Subject
    - ➡ View = Observer
    - ➡ Controller is whatever object changes the state of the subject
  - ⇒ Java 1.1 AWT/Swing Event Model
- Related Patterns
  - ⇒ Mediator
    - ➡ To encapsulate complex update semantics

Java Implementation Of Observer

- We could implement the Observer pattern “from scratch” in Java
- But Java provides the Observable/Observer classes as built-in support for the Observer pattern
- The java.util.Observable class is the base Subject class. Any class that wants to be observed extends this class.
  - ⇒ Provides methods to add/delete observers
  - ⇒ Provides methods to notify all observers
  - ⇒ A subclass only needs to ensure that its observers are notified in the appropriate mutators
  - ⇒ Uses a Vector for storing the observer references
- The java.util.Observer interface is the Observer interface. It must be implemented by any observer class.

The java.util.Observable Class

- public Observable()
  - ⇒ Construct an Observable with zero Observers
- public synchronized void addObserver(Observer o)
  - ⇒ Adds an observer to the set of observers of this object
- public synchronized void deleteObserver(Observer o)
  - ⇒ Deletes an observer from the set of observers of this object
- protected synchronized void setChanged()
  - ⇒ Indicates that this object has changed
- protected synchronized void clearChanged()
  - ⇒ Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change. This method is called automatically by notifyObservers().

The java.util.Observable Class

- public synchronized boolean hasChanged()
  - ⇒ Tests if this object has changed. Returns true if setChanged() has been called more recently than clearChanged() on this object; false otherwise.
- public void notifyObservers(Object arg)
  - ⇒ If this object has changed, as indicated by the hasChanged() method, then notify all of its observers and then call the clearChanged() method to indicate that this object has no longer changed. Each observer has its update() method called with two arguments: this observable object and the arg argument. The arg argument can be used to indicate which attribute of the observable object has changed.
- public void notifyObservers()
  - ⇒ Same as above, but the arg argument is set to null. That is, the observer is given no indication what attribute of the observable object has changed.

The java.util.Observer Interface

- public abstract void update(Observable o, Object arg)
  - ⇒ This method is called whenever the observed object is changed. An application calls an observable object's notifyObservers method to have all the object's observers notified of the change.
  - ⇒ Parameters:
    - ➥ o - the observable object
    - ➥ arg - an argument passed to the notifyObservers method

Observable/Observer Example

```
/**
 * A subject to observe!
*/
public class ConcreteSubject extends Observable {

  private String name;
  private float price;

  public ConcreteSubject(String name, float price) {
    this.name = name;
    this.price = price;
    System.out.println("ConcreteSubject created: " + name + " at "
      + price);
  }
```

Observable/Observer Example (Continued)

```
  public String getName() {return name;}

  public float getPrice() {return price;}

  public void setName(String name) {
    this.name = name;
    setChanged();
    notifyObservers(name);
  }

  public void setPrice(float price) {
    this.price = price;
    setChanged();
    notifyObservers(new Float(price));
  }

}
```


Observable/Observer Example (Continued)

```
// An observer of name changes.
public class NameObserver implements Observer {
  private String name;

  public NameObserver() {
    name = null;
    System.out.println("NameObserver created: Name is " + name);
  }

  public void update(Observable obj, Object arg) {
    if (arg instanceof String) {
      name = (String)arg;
      System.out.println("NameObserver: Name changed to " + name);
    }
  }
}
```

Observable/Observer Example (Continued)

```
// An observer of price changes.
public class PriceObserver implements Observer {
  private float price;

  public PriceObserver() {
    price = 0;
    System.out.println("PriceObserver created: Price is " + price);
  }

  public void update(Observable obj, Object arg) {
    if (arg instanceof Float) {
      price = ((Float)arg).floatValue();
      System.out.println("PriceObserver: Price changed to " +
                         price);
    }
  }
}
```


Observable/Observer Example (Continued)

```
// Test program for ConcreteSubject, NameObserver and PriceObserver
public class TestObservers {
  public static void main(String args[]) {
    // Create the Subject and Observers.
    ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();
    // Add those Observers!
    s.addObserver(nameObs);
    s.addObserver(priceObs);
    // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Surge Crispies");
  }
}
```

Observable/Observer Example (Continued)

- Test program output

```
ConcreteSubject created: Corn Pops at 1.29
NameObserver created: Name is null
PriceObserver created: Price is 0.0
NameObserver: Name changed to Frosted Flakes
PriceObserver: Price changed to 4.57
PriceObserver: Price changed to 9.22
NameObserver: Name changed to Surge Crispies
```


A Problem With Observable/Observer

- Problem:
  - ⇒ Suppose the class which we want to be an observable is already part of an inheritance hierarchy:

    ```
    class SpecialSubject extends ParentClass
    ```

  - ⇒ Since Java does not support multiple inheritance, how can we have ConcreteSubject extend both Observable and ParentClass?
- Solution:
  - ⇒ Use Delegation
  - ⇒ We will have SpecialSubject contain an Observable object
  - ⇒ We will delegate the observable behavior that SpecialSubject needs to this contained Observable object

Delegated Observable

```
/**
  * A subject to observe!
  * But this subject already extends another class.
  * So use a contained DelegatedObservable object.
  * Note that in this version of SpecialSubject we do
  *   not duplicate any of the interface of Observable.
  *   Clients must get a reference to our contained
  *   Observable object using the getObservable() method.
*/
public class SpecialSubject extends ParentClass {

  private String name;
  private float price;

  private DelegatedObservable obs;
```


Delegated Observable (Continued)

```
  public SpecialSubject(String name, float price) {
    this.name = name;
    this.price = price;
    obs = new DelegatedObservable();
  }

  public String getName() {return name;}

  public float getPrice() {return price;}

  public Observable getObservable() {return obs;}
```

Delegated Observable (Continued)

```
  public void setName(String name) {
    this.name = name;
    obs.setChanged();
    obs.notifyObservers(name);
  }

  public void setPrice(float price) {
    this.price = price;
    obs.setChanged();
    obs.notifyObservers(new Float(price));
  }

}
```


Delegated Observable (Continued)

- What's this DelegatedObservable class?
- Two methods of java.util.Observable are protected methods: setChanged() and clearChanged()
- Apparently, the designers of Observable felt that only subclasses of Observable (that is, "true" observable subjects) should be able to modify the state-changed flag
- If SpecialSubject contains an Observable object, it could not invoke the setChanged() and clearChanged() methods on it
- So we have DelegatedObservable extend Observable and override these two methods making them have public visibility
- Java rule: An subclass can changed the visibility of an overridden method of its superclass, but only if it provides more access

Delegated Observable (Continued)

```
// A subclass of Observable that allows delegation.
public class DelegatedObservable extends Observable {

  public void clearChanged() {
    super.clearChanged();
  }

  public void setChanged() {
    super.setChanged();
  }

}
```


Delegated Observable (Continued)

```
// Test program for SpecialSubject with a Delegated Observable.
public class TestSpecial {
  public static void main(String args[]) {
    // Create the Subject and Observers.
    SpecialSubject s = new SpecialSubject("Corn Pops", 1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();
    // Add those Observers!
    s.getObservable().addObserver(nameObs);
    s.getObservable().addObserver(priceObs);
    // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Surge Crispies");
  }
}
```

Delegated Observable (Continued)

- This version of SpecialSubject did not provide implementations of any of the methods of Observable. As a result, it had to allow its clients to get a reference to its contained Observable object using the getObservable() method. This may have dangerous consequences. A rogue client could, for example, call the deleteObservers() method on the Observable object, and delete all the observers!
- Let's have SpecialSubject not expose its contained Observable object, but instead provide "wrapper" implementations of the addObserver() and deleteObserver() methods which simply pass on the request to the contained Observable object.

Delegated Observable (Continued)

```
/**
  * A subject to observe!
  * But this subject already extends another class.
  * So use a contained DelegatedObservable object.
  * Note that in this version of SpecialSubject we
  *   provide implementations of two of the methods
  *   of Observable: addObserver() and deleteObserver().
  *   These implementations simply pass the request on
  *   to our contained DelegatedObservable reference.
  *   Now clients can use the normal Observable semantics
  *   to add themselves as observers of this object.
*/
public class SpecialSubject2 extends ParentClass {

  private String name;
  private float price;
  private DelegatedObservable obs;
```

Delegated Observable (Continued)

```
  public SpecialSubject2(String name, float price) {
    this.name = name;
    this.price = price;
    obs = new DelegatedObservable();
  }

  public String getName() {return name;}

  public float getPrice() {return price;}

  public void addObserver(Observer o) {
    obs.addObserver(o);
  }

  public void deleteObserver(Observer o) {
    obs.deleteObserver(o);
  }
```


Delegated Observable (Continued)

```
  public void setName(String name) {
    this.name = name;
    obs.setChanged();
    obs.notifyObservers(name);
  }

  public void setPrice(float price) {
    this.price = price;
    obs.setChanged();
    obs.notifyObservers(new Float(price));
  }

}
```

Delegated Observable (Continued)

```
// Test program for SpecialSubject2 with a Delegated Observable.
public class TestSpecial2 {
  public static void main(String args[]) {
    // Create the Subject and Observers.
    SpecialSubject2 s = new SpecialSubject2("Corn Pops", 1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();
    // Add those Observers!
    s.addObserver(nameObs);
    s.addObserver(priceObs);
    // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Surge Crispies");
  }
}
```


Java 1.1 Event Model

- Java 1.1 introduced a new GUI event model based on the Observer Pattern
- GUI components which can generate GUI events are called *event sources*
- Objects that want to be notified of GUI events are called *event listeners*
- Event generation is also called *firing the event*
- Comparison to the Observer Pattern:

  ConcreteSubject => event source

  ConcreteObserver => event listener
- For an event listener to be notified of an event, it must first register with the event source

Java 1.1 Event Model (Continued)

- An event listener must implement an interface which provides the method to be called by the event source when the event occurs
- Unlike the Observer Pattern which defines just the one simple Observer interface, the Java 1.1 AWT event model has 11 different listener interfaces, each tailored to a different type of GUI event:
  - ⇨ Listeners For Semantic Events
    - ➡ ActionListener
    - ➡ AdjustmentListener
    - ➡ ItemListener
    - ➡ TextListener

Java 1.1 Event Model (Continued)

  - ⇨ Listeners For Low-Level Events
    - ➡ ComponentListener
    - ➡ ContainerListener
    - ➡ FocusListener
    - ➡ KeyListener
    - ➡ MouseListener
    - ➡ MouseMotionListener
    - ➡ WindowListener
- Some of these listener interfaces have several methods which must be implemented by an event listener. For example, the WindowListener interface has seven such methods. In many cases, an event listener is really only interested in one specific event, such as the Window Closing event.

Java 1.1 Event Model (Continued)

- Java provides “adapter” classes as a convenience in this situation. For example, the WindowAdapter class implements the WindowListener interface, providing “do nothing” implementation of all seven required methods. An event listener class can extend WindowAdapter and override only those methods of interest.

AWT Example 1

```
import java.awt.*;
import java.awt.event.*;

/**
  * An example of the Java 1.1 AWT event model.
  * This class not only builds the GUI, but it is the
  *   listener for button events.
*/
public class ButtonExample1
  extends WindowAdapter
  implements ActionListener {

  Frame frame;
  Panel buttonPanel;
  Button redButton, greenButton;
```

AWT Example 1 (Continued)

```
// Build the GUI and display it.
public ButtonExample1(String title) {
  frame = new Frame(title);
  buttonPanel = new Panel(new FlowLayout());

  redButton = new Button("Red");
  redButton.setBackground(Color.red);
  redButton.setActionCommand("Change To Red");
  redButton.addActionListener(this);
  buttonPanel.add(redButton);

  greenButton = new Button("Green");
  greenButton.setBackground(Color.green);
  greenButton.setActionCommand("Change To Green");
  greenButton.addActionListener(this);
  buttonPanel.add(greenButton);
```


AWT Example 1 (Continued)

```
  frame.add("Center", buttonPanel);
  frame.addWindowListener(this);
  frame.pack();
  frame.setVisible(true);
}

// Since we are a WindowAdapter, we already implement the
// WindowListener interface.  So only override those methods
// we are interested in.
public void windowClosing(WindowEvent e) {
  System.exit(0);
}
```

AWT Example 1 (Continued)

```
  // Since we handle the button events, we must implement
  // the ActionListener interface.
  public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    if (cmd.equals("Change To Red")) {
      System.out.println("Red pressed");
      buttonPanel.setBackground(Color.red);
    }
    else if (cmd.equals("Change To Green")) {
      System.out.println("Green pressed");
      buttonPanel.setBackground(Color.green);
    }
  }

  public static void main(String args[]) {
    new ButtonExample1("Button Example");
  }
}
```


AWT Example 2

```
import java.awt.*;
import java.awt.event.*;

/**
  * An example of the Java 1.1 AWT event model.
  * This class uses anonymous inner classes as the
  *   listeners for button events.  As a result, we
  *   do not need to implement ActionListener.
*/
public class ButtonExample2
  extends WindowAdapter {

  Frame frame;
  Panel buttonPanel;
  Button redButton, greenButton;
```

AWT Example 2 (Continued)

```
// Build the GUI and display it.
public ButtonExample2(String title) {
  frame = new Frame(title);
  buttonPanel = new Panel(new FlowLayout());

  redButton = new Button("Red");
  redButton.setBackground(Color.red);
  redButton.setActionCommand("Change To Red");
  redButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      System.out.println("Red pressed");
      buttonPanel.setBackground(Color.red);
    }
  } );
  buttonPanel.add(redButton);
```


AWT Example 2 (Continued)

```
  greenButton = new Button("Green");
  greenButton.setBackground(Color.green);
  greenButton.setActionCommand("Change To Green");
  greenButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      System.out.println("Green pressed");
      buttonPanel.setBackground(Color.green);
    }
  } );
  buttonPanel.add(greenButton);

  frame.add("Center", buttonPanel);
  frame.addWindowListener(this);
  frame.pack();
  frame.setVisible(true);
}
```

AWT Example 2 (Continued)

```
  // Since we are a WindowAdapter, we already implement the
  // WindowListener interface.  So only override those methods
  // we are interested in.
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }

  public static void main(String args[]) {
    new ButtonExample2("Button Example");
  }

}
```


MVC Example 1

- This example shows the model and the view in the same class

```
/**
  * Class CounterGui demonstrates having the model and view
  *   in the same class.
*/
public class CounterGui extends Frame {

  // The counter.  (The model!)
  private int counter = 0;

  // The view.
  private TextField tf = new TextField(10);
```

MVC Example 1 (Continued)

```
public CounterGui(String title) {
  super(title);
  Panel tfPanel = new Panel();
  tf.setText("0");
  tfPanel.add(tf);
  add("North", tfPanel);

  Panel buttonPanel = new Panel();

  Button incButton = new Button("Increment");
  incButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      counter++;
      tf.setText(counter + "");
    }
  } );
  buttonPanel.add(incButton);
```


MVC Example 1 (Continued)

```
  Button decButton = new Button("Decrement");
  decButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      counter--;
      tf.setText(counter + "");
    }
  } );
  buttonPanel.add(decButton);

  Button exitButton = new Button("Exit");
  exitButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      System.exit(0);
    }
  } );
  buttonPanel.add(exitButton);

  add("South", buttonPanel);
```

MVC Example 1 (Continued)

```
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    } );
  }

  public static void main(String[] argv) {
    CounterGui cg = new CounterGui("CounterGui");
    cg.setSize(300, 100);
    cg.setVisible(true);
  }
}
```

- Where is the controller in this example?? The controllers are the instances of the anonymous classes which handle the button presses.

MVC Example 2

- This example shows the model and the view in separate classes
- First the view class:

```
/**
  * Class CounterView demonstrates having the model and view in the
  *   separate classes.  This class is just the view.
*/
public class CounterView extends Frame {

  // The view.
  private TextField tf = new TextField(10);

  // A reference to our associated model.
  private Counter counter;
```

MVC Example 2 (Continued)

```
public CounterView(String title, Counter c) {
  super(title);
  counter = c;

  Panel tfPanel = new Panel();
  tf.setText(counter.getCount()+ "");
  tfPanel.add(tf);
  add("North", tfPanel);
  Panel buttonPanel = new Panel();

  Button incButton = new Button("Increment");
  incButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      counter.incCount();
      tf.setText(counter.getCount() + "");
    }
  } );
  buttonPanel.add(incButton);
```


MVC Example 2 (Continued)

```
  Button decButton = new Button("Decrement");
  decButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      counter.decCount();
      tf.setText(counter.getCount()+ "");
    }
  } );
  buttonPanel.add(decButton);

  Button exitButton = new Button("Exit");
  exitButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      System.exit(0);
    }
  } );
  buttonPanel.add(exitButton);

  add("South", buttonPanel);
```

MVC Example 2 (Continued)

```
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    } );
  }

  public static void main(String[] argv) {
    Counter counter = new Counter(0);
    CounterView cv1 = new CounterView("CounterView1", counter);
    cv1.setSize(300, 100);
    cv1.setVisible(true);
    CounterView cv2 = new CounterView("CounterView2", counter);
    cv2.setSize(300, 100);
    cv2.setVisible(true);
  }
}
```


MVC Example 2 (Continued)

- Next the model class:

```
/**
  * Class Counter implements a simple counter model.
*/
public class Counter {

  // The model.
  private int count;

  public Counter(int count) { this.count = count; }

  public int getCount() { return count; }
  public void incCount() { count++; }
  public void decCount() { count--; }

}
```

MVC Example 2 (Continued)

- Note that we instantiated one model and two views in this example:

- But we have a problem! When the model changes state, only one view updates!
- We need the Observer Pattern here!

MVC Example 3

- This example shows the model and the view in separate classes with the model being observable
- First the model class:

```
import java.util.Observable;

/**
  * Class ObservableCounter implements a simple observable
  *   counter model.
*/
public class ObservableCounter extends Observable {

  // The model.
  private int count;

  public ObservableCounter(int count) { this.count = count; }
```

MVC Example 3 (Continued)

```
  public int getCount() { return count; }

  public void incCount() {
    count++;
    setChanged();
    notifyObservers();
  }

  public void decCount() {
    count--;
    setChanged();
    notifyObservers();
  }

}
```


MVC Example 3 (Continued)

- Next the view class:

```
/**
  * Class ObservableCounterView demonstrates having the model
  * and view in the separate classes.  This class is just the view.
*/
public class ObservableCounterView extends Frame {

  // The view.
  private TextField tf = new TextField(10);

  // A reference to our associated model.
  private ObservableCounter counter;
```

MVC Example 3 (Continued)

```
  public ObservableCounterView(String title, ObservableCounter c) {
    super(title);
    counter = c;

    // Add an anonymous observer to the ObservableCounter.
    counter.addObserver(new Observer() {
      public void update(Observable src, Object obj) {
        if (src instanceof ObservableCounter && src == counter) {
          tf.setText(((ObservableCounter)src).getCount() + "");
        }
      }
    } );

    // Same GUI code as Example 2 not shown...
```


MVC Example 3 (Continued)

```
  public static void main(String[] argv) {
    ObservableCounter counter = new ObservableCounter(0);
    ObservableCounterView cv1 = new
      ObservableCounterView("ObservableCounterView1", counter);
    cv1.setSize(300, 100);
    cv1.setVisible(true);
    ObservableCounterView cv2 = new
      ObservableCounterView("ObservableCounterView2", counter);
    cv2.setSize(300, 100);
    cv2.setVisible(true);
  }

}
```

MVC Example 3 (Continued)

- Looking good now!