# Swing issues
# Actually, many GUI systems have them

Simple *live* example:


# Swing issues

```
public class MainFrame extends JFrame
   implements ActionListener {
   ...
   private JButton bt1;      // Swing stuff
   private JButton bt2;
   ...
   int count;


   public MainFrame() {
      setSize(WIDTH,HEIGHT); // Swing Stuff!
      ...
      count = 0;
   }
```

## Swing issues

```
public class MainFrame extends JFrame
    implements ActionListener {
    ...
    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == bt1) {
            count++;
        } else if (e.getSource() == bt2) {
            count += 2;
        }
        textArea.setText("");
        for (int i = 0; i < count; i++) {
            textArea.append("@");
        }
    }
}
```

## Swing issues

Critique 1:

- Exposes the **`ActionListener`** Interface

```
public class BadApp {
    public static void main(String[] args) {
        JFrame mainFrame = new MainFrame();
        mainFrame.setVisible(true);
        // Trying to access the ActionListener method
        if (mainFrame instanceof ActionListener) {
            ActionListener al = (ActionListener)mainFrame;
            try { // For the surprise effect :> >
                Thread.sleep(5000); // Pause it for 5 seconds
            } catch (InterruptedException e) {}
            al.actionPerformed(null); // ← We can call it!
        }
    }
}
```

## Swing issues

Use a separate class to implement it

```
??? class ClickListener implements ActionListener {
    private int count = 0;

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == bt1) {  // ← NO ACCESS!!
            count++;
        } else
        if (e.getSource() == bt2) {  // ← NO ACCESS!!
            count += 2;
        }
        textArea.setText("");        // ← NO ACCESS!!
        for (int i = 0; i < count; i++) {
            textArea.append("@");
        }
    }
}
```

## Swing issues

Make it an *inner class:*

```
public class MainFrame extends JFrame {

    public MainFrame() {
        ...
        ActionListener al = new ClickListener();
        bt1 = new JButton("+1");
        bt1.addActionListener(al);
        ...
    }

    // Private inner class
    private class ClickListener implements ActionListener {
        ...
    }
}
```

## Swing issues

Critique 2:

- JFrame needs extra variables
- JFrame needs extra initializations
- Listener needs extra code

→ Becomes more and more complicated if the App Logic becomes more and complicated

Inner Listeners help, but only so much...

## Swing issues – Overburdened Listener

```
@Override
public void actionPerformed(ActionEvent e) {
   if (e.getSource() == bt1) {
      Class.forName(driverName);
      Connection conn = DriverManager.getConnection("");
      DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
      factory.setValidating(false);
      factory.setIgnoringElementContentWhitespace(true);
      DocumentBuilder builder =
            factory.newDocumentBuilder();
      Document doc = builder.parse(new
      ByteArrayInputStream(xml.getBytes(ENCODING)));
      XPathFactory xpfactory =
            XPathFactory.newInstance();
      // and so on and so on and so on...
```

# Swing issues – Factoring!

- Isolate Logic from Listening
- Develop up a specific class that doesn't know about Jframes, Listeners, … (Swing *agnostic*, context *agnostic*)

```
public class Incrementer {
    int count;

    public Incrementer() {
        count = 0;
    }

    public int incrementBy(int incr) {
        count += incr;
        return count;
    }
}
```

# Swing issues – Factoring!

- Call this from the Listener (trivial)

```
  private class ClickListener implements ActionListener {
      private Incrementer inc;
      public ClickListener() { inc = new Incrementer(); }
      @Override
      public void actionPerformed(ActionEvent e) {
          int result;
          if (e.getSource() == bt1) {
              result = inc.incrementBy(1);
          } else if (e.getSource() == bt2) {
              result = inc.incrementBy(2);
          }
          textArea.setText("");
          for (int i = 0; i < result; i++) {
              textArea.append("@");
          }
      }
  }
```

## More about Factoring

- The interface of our **`Incrementer`** class is still app dependent.
- Idea (doesn't always work...): Hide this behind an Interface

```
public interface Model {
    Object update(Object arg);
}


public class Incrementer implements Model {
    ...
    @Override
    public Object update(Object incr) {
        count += (Integer)incr;   // auto (un)boxing at work!
        return count;
    }
}
```

## Swing issues – Factoring!

- Remove app dependent code from the Listener

```
  private class ClickListener implements ActionListener {
      private Model m;
      public ClickListener() { m = new Incrementer(); }
      @Override
      public void actionPerformed(ActionEvent e) {
          int result;
          if (e.getSource() == bt1) {
              result = (Integer)(m.update(1)); // bit clumsy!
          } else if (e.getSource() == bt2) {
              result = (Integer)(m.update(2));
          }
          textArea.setText("");
          for (int i = 0; i < result; i++) {
              textArea.append("@");
          }
      }
  }
```

# How to get a Model into the Listener?

• You don't! Place it in the Jframe over the Constructor

```
public class MainFrame extends JFrame {
    private Model model;

    public MainFrame(Model model) {
        this.model = model;
        ...
    }

    private class ClickListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            int result;
            if (e.getSource() == bt1) {
                result = (Integer)(model.update(1));
            }
            ...
}
```

# How to get a Model into the Listener?

• Now the App controls which model to use

```
public class App {
    public static void main(String[] args) {
        Model model = new Incrementer();
        JFrame mainFrame = new MainFrame(model);
        ...
```

• That's better: How would a Jframe decide that?
→ *Plug-and-Play architecture*

Remark: Looks way better than it actually is:
- Still depends on type conversions (all over **`Object`**)
  → Susceptible to run-time casting problems!
  → Make everything generic (→ Advanced Programming)

## Factoring: Returning or calling back?

```
public void actionPerformed(ActionEvent e) {
    int result;
    // Mixes call to model with the return
    if (e.getSource() == bt1) {
        result = (Integer)(m.update(1));
    } else if (e.getSource() == bt2) {
        result = (Integer)(m.update(2));
    }
    // More or less forces me to program the
    // GUI update in the Listener (overburdening!)
    textArea.setText("");
    for (int i = 0; i < result; i++) {
        textArea.append("@");
    }
}
```

Can we separate that? Oh yeah!

## Factoring: Views to separate concerns

```
public interface View {
    void notify(Object result);
}

public class MainFrame extends JFrame implements View
    ...
    @Override
    public void notify(Object result) {
        textArea.setText("");
        for (int i = 0; i < (Integer)result; i++) {
            textArea.append("@");
        }
    }
    ...
}
```

## Factoring: View is called back from Model

```
public interface Model {
   Object update(View v, Object arg); // Must know
                                      // who called this
}
```

```
   @Override
   public void update(view v, Object arg) {
      if (arg instanceof Integer) {
         count += (Integer)arg;
         v.notify(count);
      }
   }
```

Hang on: That's still **not** the final version!

## Factoring: View is called back

This would mean the Listener has to know the View it's running in → Clumsy code alert!

```
public class MainFrame extends JFrame implements View {

   private View me;
   ...
      me = this; // In the constructor
   ...
   private class ClickListener implements ActionListener {
      @Override
      public void actionPerformed(ActionEvent e) {
         int result;
         if (e.getSource() == bt1) {
            model.update(me,1)); // this won't work here
         }
         ...
```

# Instead of passing: Registering!

A model has a list of Views – managed in an abstract class rather than an Interface.

```
public abstract class Model {
   private List<View> views;

   public Model () { this.views = new ArrayList<View>(); }

   public void addView(View v) { views.add(v); }

   protected void notifyViews(Object changed) {
      for (View v : views) {
         v.notify(changed);
      }
   }

   public abstract void update(Object arg);
}
```

# Instead of passing: Registering!

a) Every View registers itself with the model:

```
  public MainFrame(Model model) {
      this.model = model;
      model.addView(this);
      ...
```

b) The Model calls all Views back:

```
   @Override
   public void update(Object arg) {
      if (arg instanceof Integer) {
         count += (Integer)arg;
         notifyViews(count);
      }
   }
```

## Multiple views

Now any Model can have multiple views! Some might not even update the model, but are notified by changes.

```
public class SubFrame extends JFrame implements View {

    private JTextArea textArea;
    private Model model;

    public SubFrame(Model model) {
        this.model = model; // Technically not necessary
        model.addView(this);
        ...
    }

    @Override
    public void notify(Object result) {
        textArea.append(" " + result);
    }
}
```

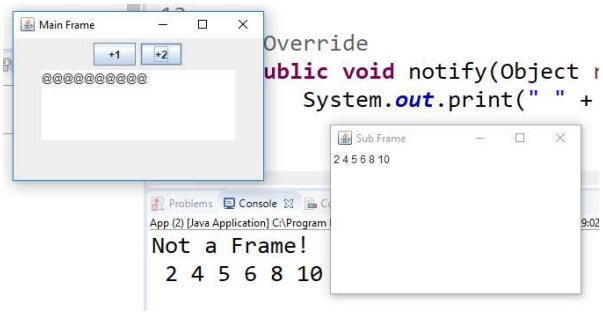
## Multiple views

Don't have to use GUIs anymore :-)

```
public class SysOutView implements View {

    private Model model;

    public SysOutView(Model model) {
        this.model = model;
        model.addView(this);
        System.out.println("Not a Frame!");
    }

    @Override
    public void notify(Object result) {
        System.out.print(" " + result);
    }
}
```

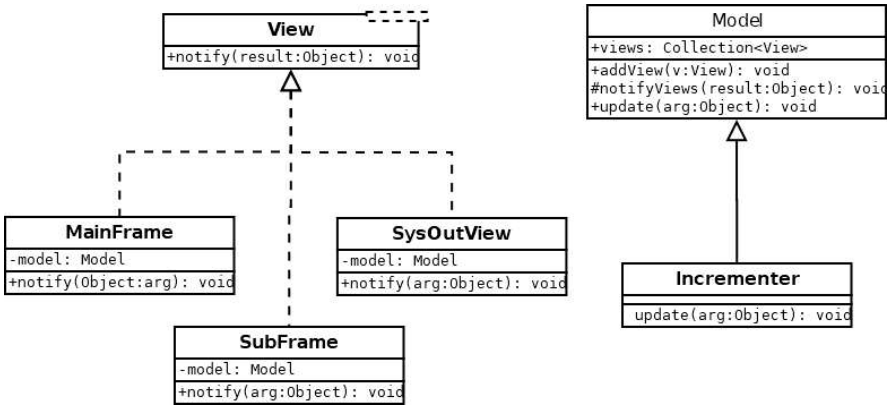
# Multiple views

Wiring it all up!




```
public class App {
    public static void main(String[] args) {
        Model model = new Incrementer();

        JFrame mainFrame = new MainFrame(model);
        JFrame subFrame = new SubFrame(model);
        SysOutView sov = new SysOutView(model);

        subFrame.setVisible(true);
        mainFrame.setVisible(true);
    }
}
```

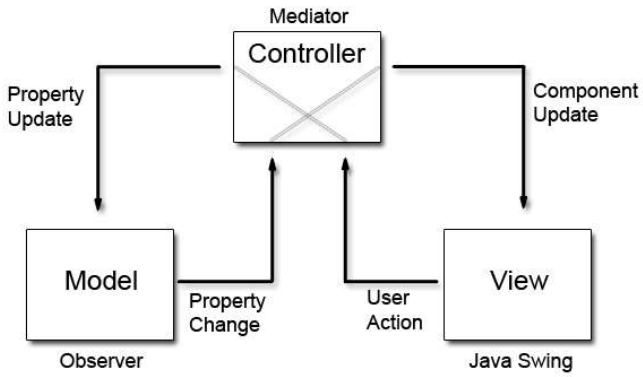
# Model-View architecture




→ Observer pattern

# Model-View-Controller architecture

# → Advanced Programming




If App flow logic becomes too complicated, move it into a Controller and keep Model agnostic of it.