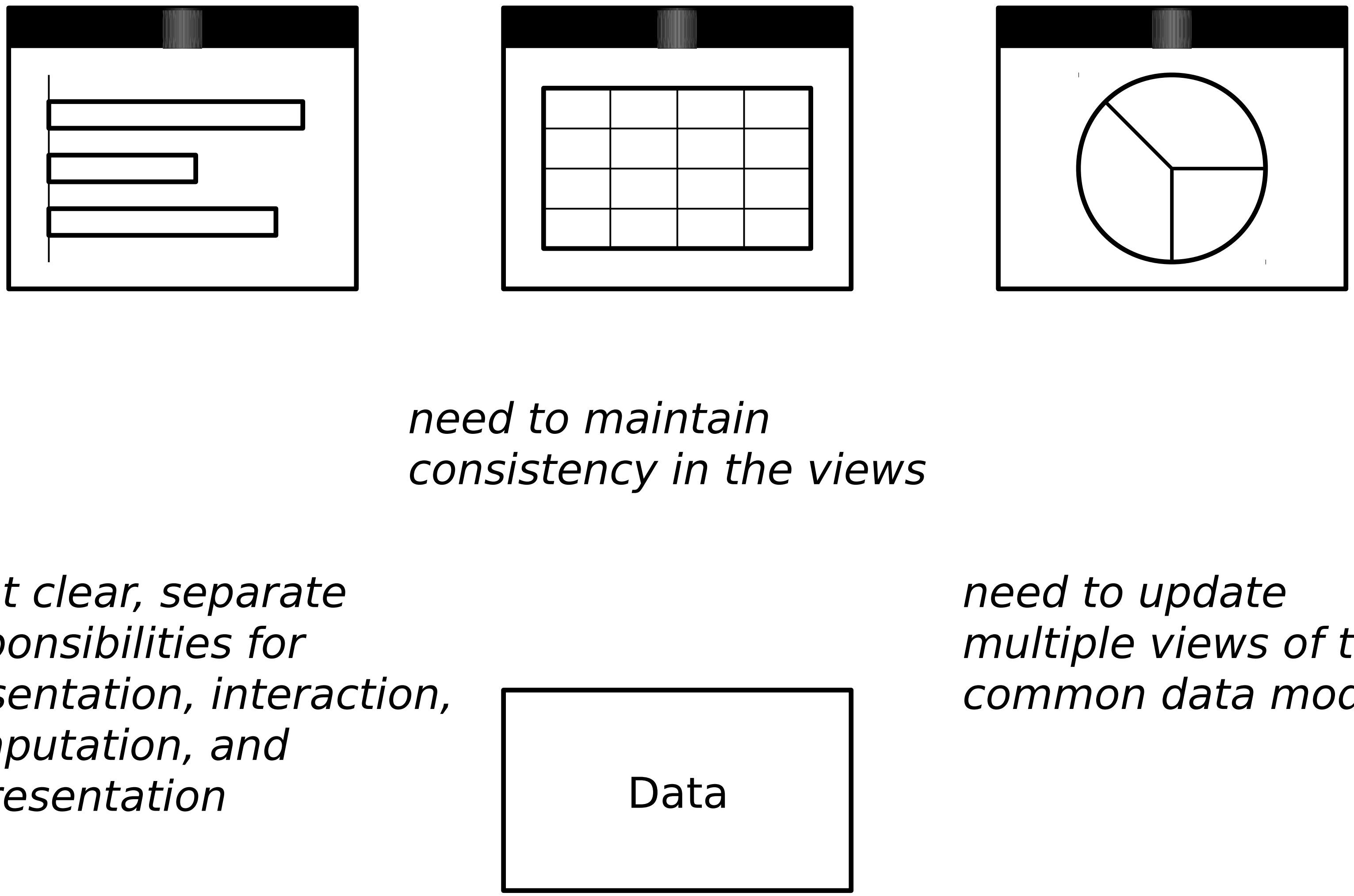
Abram Hindle
Department of Computing Science
University of Alberta

MVC and Android

Slides originally by Ken Wong

- **Model/VieW/Controller**

views



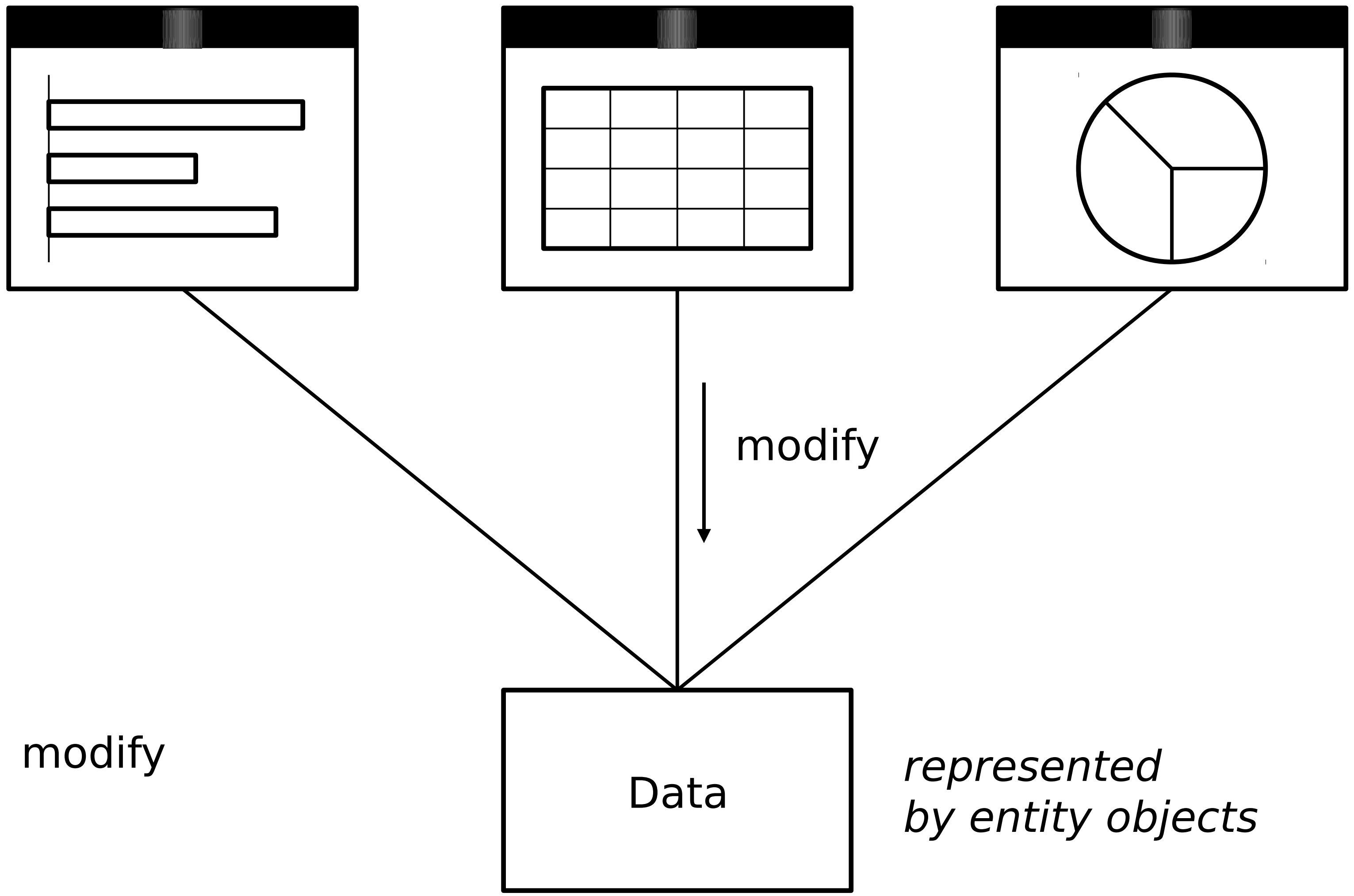
need to maintain consistency in the views

want clear, separate responsibilities for presentation, interaction, computation, and representation

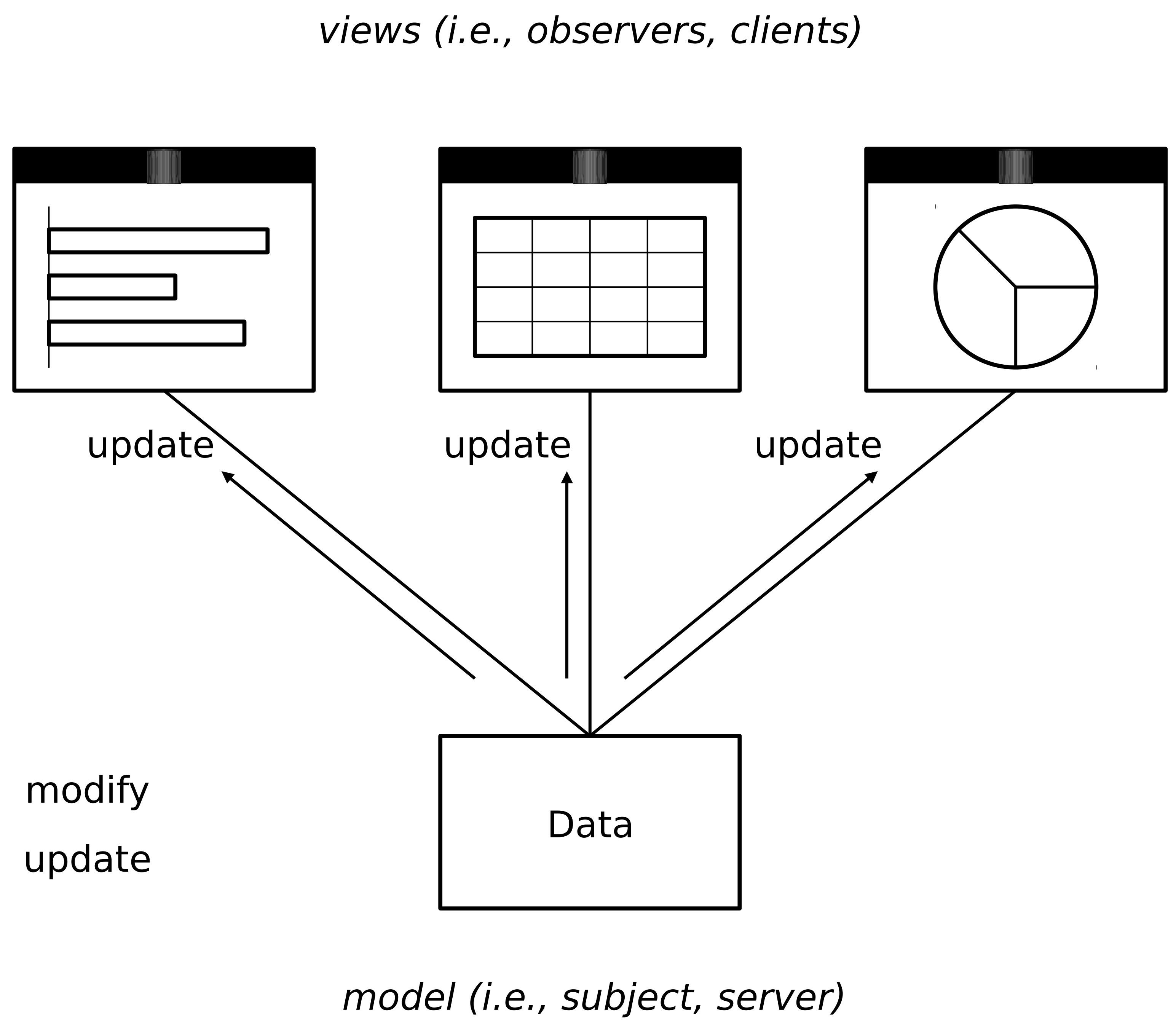
need to update multiple views of the common data model

model

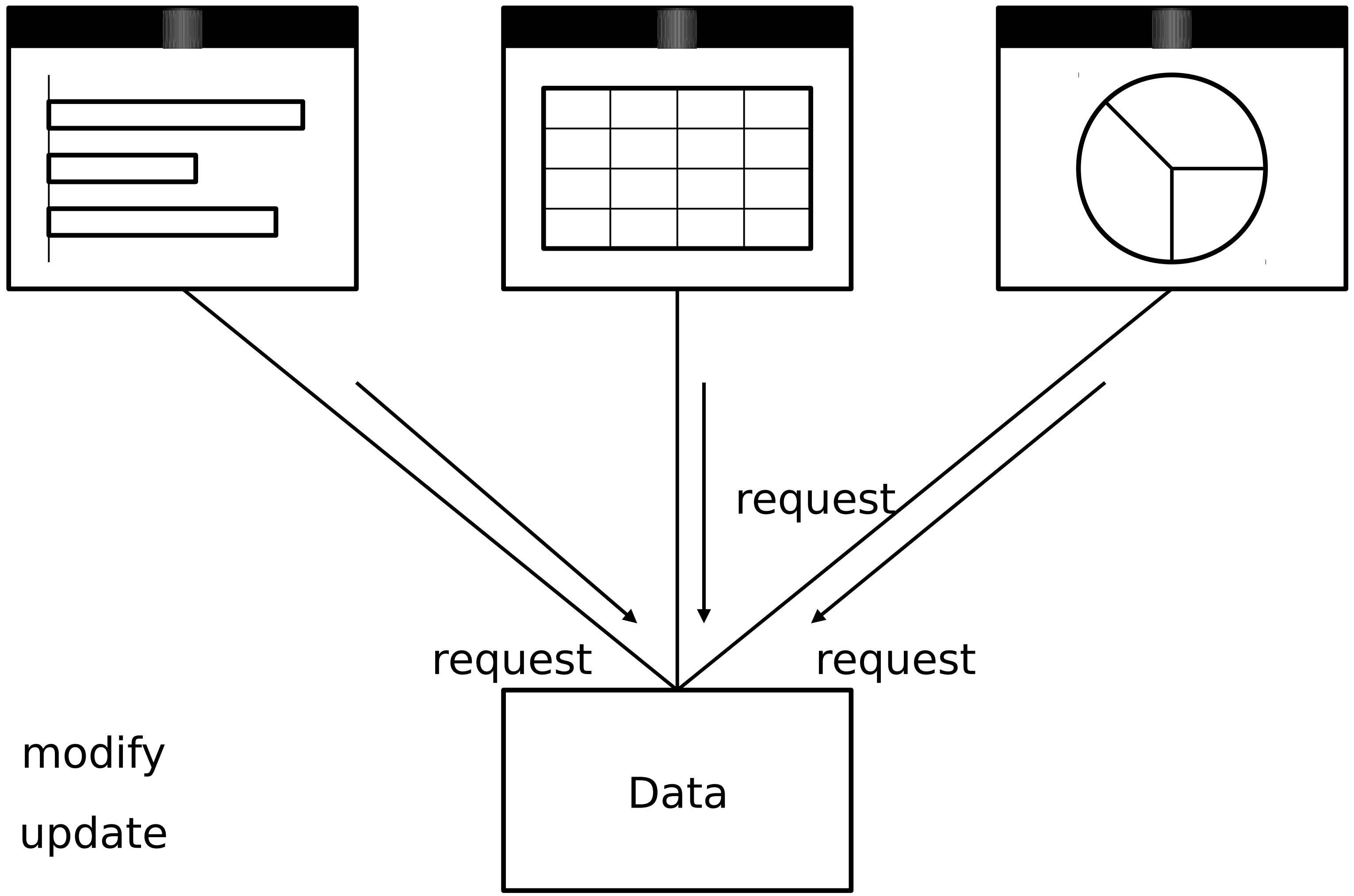
views (i.e., observers, clients)



model (i.e., subject, server)


views (i.e., observers, clients)
update
update
update
modify
update
Data
model (i.e., subject, server)

views (i.e., observers, clients)



modify

update

request

model (i.e., subject, server)

Model/View/Controller Roles

- Model:
  - entity layer
    - complete, self-contained representation of the data managed by the application
    - provides services to manipulate this data
    - “the back end”
  - main responsibilities
    - representation and computation issues
    - sometimes persistence

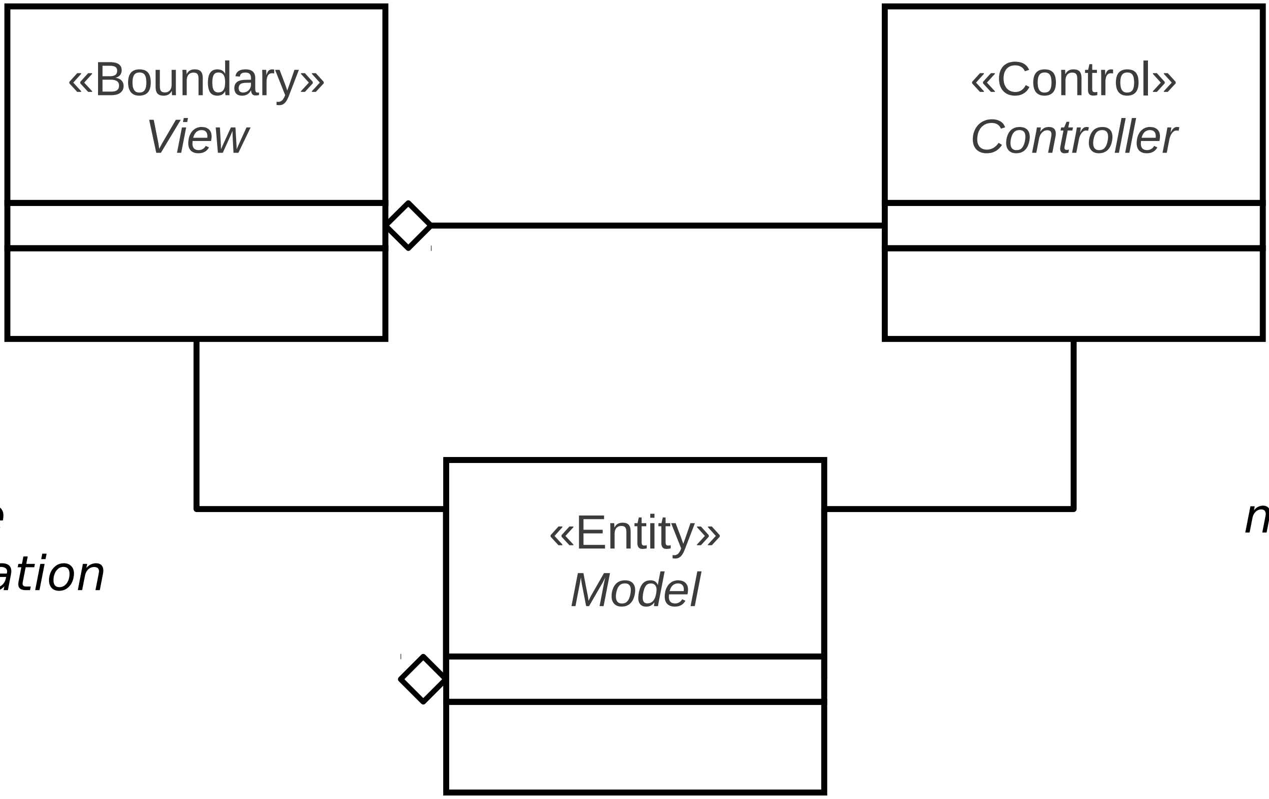
MVC Roles

- View:
  - boundary layer
    - set of user interface components
    - determines what is needed for a particular perspective of the data
    - “the front end”
  - main responsibility
    - presentation issues

MVC Roles

- Controller:
  - control layer
    - handles events and uses appropriate information from user interface components to modify the model
  - main responsibility
    - interaction issues

can create new, specific types of views without changing the model



update notification

modify

the model should not need to know the particulars of a specific view

the model should not need to know about any controllers

MVC Design Issues

- Swing dependent part:
  - views contain Swing components
  - controllers are Swing listeners
- Swing independent part:
  - the model should be as Swing free as possible
    - e.g., not using Swing types in entity classes

“MV” Design

- Generalization:
  - use “model” superclass and “view” interface
    - all models keep track of their views
    - when changed, all models notify their views to update
    - all views update themselves when notified
  - have application-specific model and view classes

Java Observer

- java.util.Observable superclass

-
  ```
public class Observable {
    …
    public Observable() { … }

    // “all models keep track of their views”
    public void addObserver( Observer o ) { … }
    public void deleteObserver( Observer o ) { … }

    // “all models notify their views to update”
    public void notifyObservers() { … }
    public void notifyObservers( Object arg ) { … }

    // note whether the model has changed
    public boolean hasChanged() { … }
    protected void clearChanged() { … }
    protected void setChanged() { … }
    …
}
```

Java Observer

- java.util.Observer interface

-
```
public interface Observer {
    public void update( Observable s, Object arg );
}
```

Java Observer

-
  ```
  // MyModel.java
  import java.util.*;

  public class MyModel extends Observable {
      private String message;

      public MyModel() {
          message = “”;
      }
      public String getMessage() {
          return message;
      }
      public void setMessage( String message ) {
          this.message = message;
          setChanged();
          notifyObservers(); // clears changed flag
      }
  }
  ```

Java Observer

-
  ```
  // MyView.java
  import java.util.*;

  public class MyView implements Observer {
      public void update( Observable s, Object arg ) {
          System.out.println(
              ((MyModel) s).getMessage()
          );
      }
  }
  ```

Java Observer

-
  ```
  // MyApp.java

  public class MyApp {

      public static void main( String args[] ) {

          MyModel theModel = new MyModel();
          MyView aView = new MyView();
          MyView anotherView = new MyView();

          theModel.addObserver( aView );
          theModel.addObserver( anotherView );

          theModel.setMessage( "hello" );
      }
  }
  ```

Observer using Java Generics

-
  ```
  // TView.java
  public interface TView<M> {
      public void update( M model );
  }
  ```

Observer using Java Generics

-
  ```
  // TModel.java
  import java.util.*;

  public class TModel<V extends TView> {
      private ArrayList<V> views;

      public TModel() {
          views = new ArrayList<V>();
      }

      public void addView( V view ) {
          if (! views.contains( view )) {
              views.add( view );
          }
      }

      …
  ```

Observer using Java Generics

-

```
        public void deleteView( V view ) {
            views.remove( view );
        }

        public void notifyViews() {
            for (V view : views) {
                view.update( this );
            }
        }

        …
    }
```

Observer using Java Generics

-
  ```
  // MyView.java
  import java.util.*;

  public class MyView implements TView<MyModel> {
      public void update( MyModel model ) {
          System.out.println( model.getMessage() );
      }
  }
  ```

Observer using Java Generics

-
  ```
  // MyModel.java
  public class MyModel extends TModel<TView> {
      private String message;

      public MyModel() {
          message = ””;
      }
      public String getMessage() {
          return message;
      }
      public void setMessage( String message ) {
          this.message = message;
          notifyViews();
      }
  }
  ```

Observer using Java Generics

-
  ```
  // MyApp.java

  public class MyApp {

      public static void main( String args[] ) {

          MyModel theModel = new MyModel();
          MyView aView = new MyView();
          MyView anotherView = new MyView();

          theModel.addView( aView );
          theModel.addView( anotherView );

          theModel.setMessage( "hello" );
      }
  }
  ```

MVC Design

- Approach:
  - use a framework that supports MVC to help structure an interactive application
  - framework is a set of cooperating classes that forms a reusable design in a particular domain
  - reusable design *and* code

MVC Framework

Who is in Control?

- Class library reuse
  - application developers:
    - write the main body of the application
    - reuse library code by calling it
- Framework reuse
  - application developers:
    - reuse the main body of the application
    - write code that the framework calls
    - reuse library code by calling it

Framework

- Separation of concerns:
  - framework
    - skeletal application code
    - general superclasses and interfaces
  - your “customizations”
    - specific subclasses and implementations

Exercise

- Design an MVC framework for building interactive applications.

Generic View

-
  ```
  // TView.java
  public interface TView<M> {
      public void update( M model );
  }
  ```

Generic Model

-
  ```
  // TModel.java
  …

  public abstract class TModel<V extends TView> {
      private ArrayList<V> views;

      protected TModel() {
          views = new ArrayList<V>();
      }

      public void addView( V view ) {
          if (! views.contains( view )) {
              views.add( view );
          }
      }
  ```

Generic Model

-

```
        public void deleteView( V view ) {
            views.remove( view );
        }

        public void notifyViews() {
            for (V view : views) {
                view.update( this );
            }
        }
    }
```

General Command

```
// TCommand.java
…

public class TCommand {
    public void execute( ActionEvent event ) {
    }
    public void execute( ItemEvent event ) {

    }
}
```

“Code Reuse”

- http://www.dilbert.com/strips/comic/1996-01-31/

General Controller

-
  ```
  // TController.java
  …

  public abstract class TController implements
      ActionListener, ItemListener {

      private JComponent component;
      private TCommand command;

      protected TController(
          JComponent component, TCommand command ) {

          this.component = component;
          this.command = command;
      }
  ```

General Controller

```
    public JComponent getComponent() {
        return component;
    }
    public TCommand getCommand() {
        return command;
    }

    public void actionPerformed(
        ActionEvent event ) {

        TCommand command = getCommand();
        if (command != null) {
            command.execute( event );
        }
    }
    …
}
```

General Button Controller

-
  ```
  // TButtonController.java
  …

  public class TButtonController extends TController {

      public TButtonController(
          JButton button, TCommand command ) {

          super( button, command );
          button.addActionListener( this );
      }
  }
  ```

General Menu Item Controller

-
  ```
  // TMenuItemController.java
  …

  public class TMenuItemController extends TController
  {

      public TMenuItemController(
          JMenuItem menuItem, TCommand command ) {

          super( menuItem, command );
          menuItem.addActionListener( this );
      }
  }
  ```

Generic Application

-
  ```
  // TApp.java
  …

  public abstract class TApp<M> {

      private static TApp theApp = null;

      public static TApp getApp() {
          return theApp;
      }

      private M model;

      public M getModel() {
          return model;
      }
  ```

Generic Application

-

```
private JFrame frame;
private JPanel content;

public JFrame getFrame() {
    return frame;
}
public JPanel getContent() {
    return content;
}
```

Generic Application

-

```
protected TApp( String title, M model ) {
    if (theApp != null) {
        return;
    }
    theApp = this;

    this.model = model;

    makeWindow( title );
}
```

Generic Application

-

```
private void makeWindow( String title ) {

    frame = new JFrame( title );

    content = new JPanel();
    frame.setContentPane( content );
}

public void show() {
    frame.pack();
    frame.setVisible( true );
}

public void addToContent(
    JComponent component ) {

    content.add( component );
}
```

Generic Application

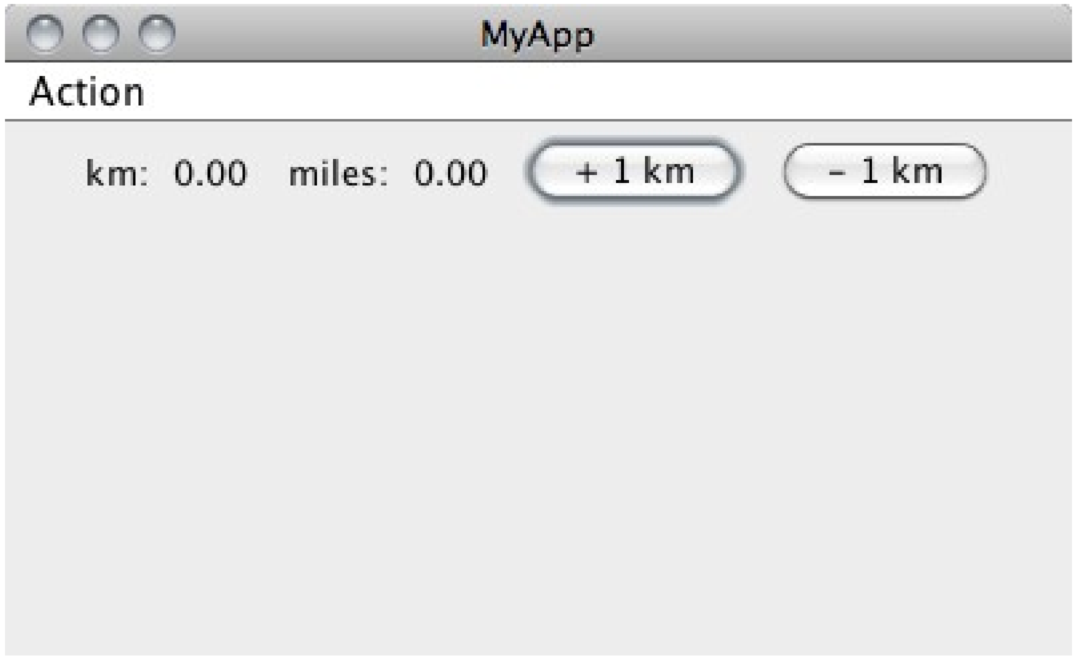
-

```
        private JMenuBar menubar = null;

        public void makeMenuBar() {
            menubar = new JMenuBar();
            frame.setJMenuBar( menubar );
        }

        public void addToMenuBar( JMenu menu ) {
            if (menubar == null) {
                return;
            }
            menubar.add( menu );
        }
    }
```

Example Custom Application



Custom View

-
  ```
  // MyLabelView.java
  …

  public class MyLabelView implements TView<MyModel>
  {

      private static DecimalFormat twoPlaces =
          new DecimalFormat( "0.00" );

      private JPanel panel;
      private JLabel labelLabel;
      private JLabel valueLabel;
      private double multiplier;
  ```

Custom View

-
```
public MyLabelView(
    String labelText, double multiplier ) {

    panel = new JPanel();
    labelLabel = new JLabel( labelText );
    panel.add( labelLabel );
    valueLabel = new JLabel( " " );
    panel.add( valueLabel );
    this.multiplier = multiplier;
}

public JComponent getComponent() {
    return panel;
}
```

Custom View

-
  ```
      public void update( MyModel model ) {
          double value =
              model.getValue() * multiplier;

          valueLabel.setText(
              twoPlaces.format( value )
          );
      }
  }
  ```

Custom Model

- `// MyModel.java`

```
public class MyModel extends TModel<TView> {
    private int value;

    public MyModel() {
        value = 0;
    }
    public int getValue() {
        return value;
    }
    public void setValue( int value ) {
        if (value < 0) {
            value = 0;
        }
        this.value = value;
        notifyViews();
    }
}
```

Custom Application

-
  ```
  // MyApp.java
  …

  public class MyApp extends TApp<MyModel> {

      public MyApp(
          String title, MyModel model ) {

          super( title, model );

          // create the UI
          MyMainView myMainView =
              new MyMainView( this, model );
          model.addView( myMainView );
      }
  ```

Custom Application

-

```
    public static void main( String args[] ) {
        MyModel model = new MyModel();
        MyApp app = new MyApp( "MyApp", model );

        model.notifyViews();

        app.getContent().setPreferredSize(
            new Dimension( 400, 200 )
        );
        app.show();
    }
}
```

Custom User Interface

-
  ```
  // MyMainView.java
  …

  public class MyMainView implements TView<MyModel> {

      private MyLabelView kmView;
      private MyLabelView milesView;

      private TCommand incrCommand;
      private TCommand decrCommand;

      private JMenu menu;

      private JMenuItem incrMenuItem;
      private JMenuItem decrMenuItem;

      private JButton incrButton;
      private JButton decrButton;
  ```

Custom User Interface

-

```
    public MyMainView(
        MyApp app, final MyModel model ) {

        // create views
        kmView = new MyLabelView(
            "km: ", 1.0
        );
        milesView = new MyLabelView(
            "miles: ", 0.621371192
        );

        // register views with model
        model.addView( kmView );
        model.addView( milesView );
```

Custom User Interface

-
  ```
  // create commands that modify the model
  incrCommand = new TCommand() {
      public void execute(
          ActionEvent event ) {

          model.setValue(
              model.getValue() + 1
          );
      }
  };
  decrCommand = new TCommand() {
      public void execute(
          ActionEvent event ) {

          model.setValue(
              model.getValue() – 1
          );
      }
  };
  ```

Custom User Interface

```
•          // views
           app.addToContent( kmView.getComponent() );
           app.addToContent( milesView.getComponent() );

           // controls
           incrButton = new JButton( "+ 1 km" );
           app.addToContent( incrButton );

           decrButton = new JButton( "- 1 km" );
           app.addToContent( decrButton );

           // associate components to commands
           new TMenuItemController(
               incrMenuItem, incrCommand );
           new TMenuItemController(
               decrMenuItem, decrCommand );
           new TButtonController(
               incrButton, incrCommand );
           new TButtonController(
               decrButton, decrCommand );
       }
```

Custom User Interface

-
  ```
      public void update( MyModel model ) {
          // nothing to do
      }
  }
  ```

Exercise

- Draw a UML sequence diagram for the behavior when a button is clicked in the example application.