

Patron: Modèle-Vue-Contrôleur (MVC)

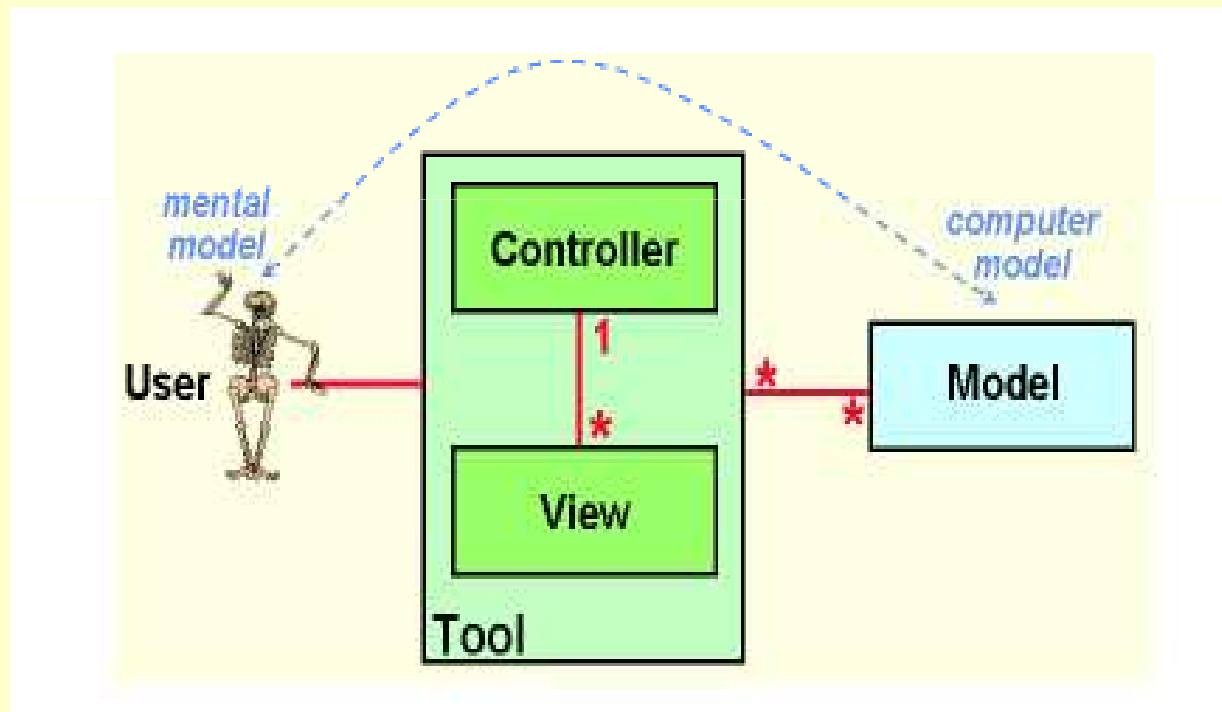
<http://www-public.it-sudparis.eu/~gibson/Teaching/DesignPatterns/DesignPatterns-MVC.pdf>

<http://www-public.it-sudparis.eu/~gibson/Teaching/DesignPatterns/DesignPatterns-MVC.ppt>

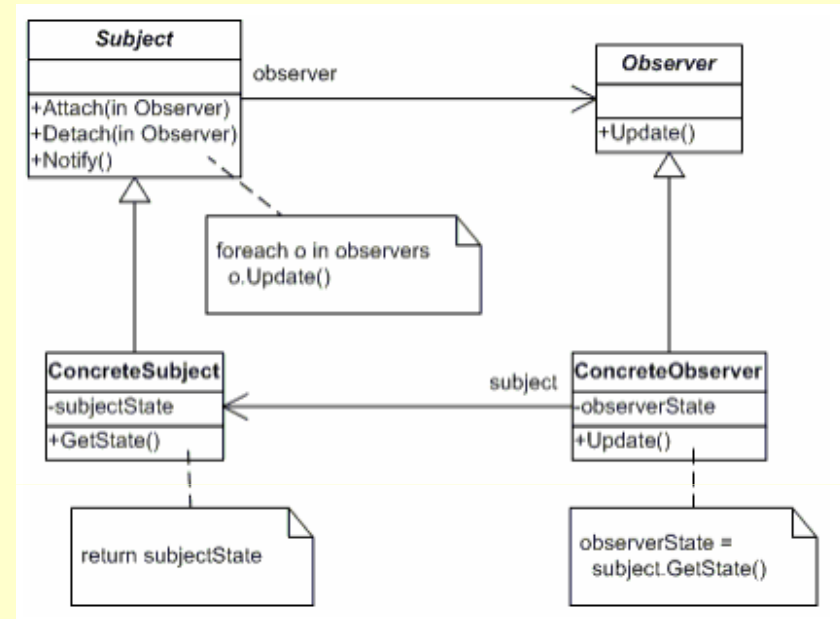
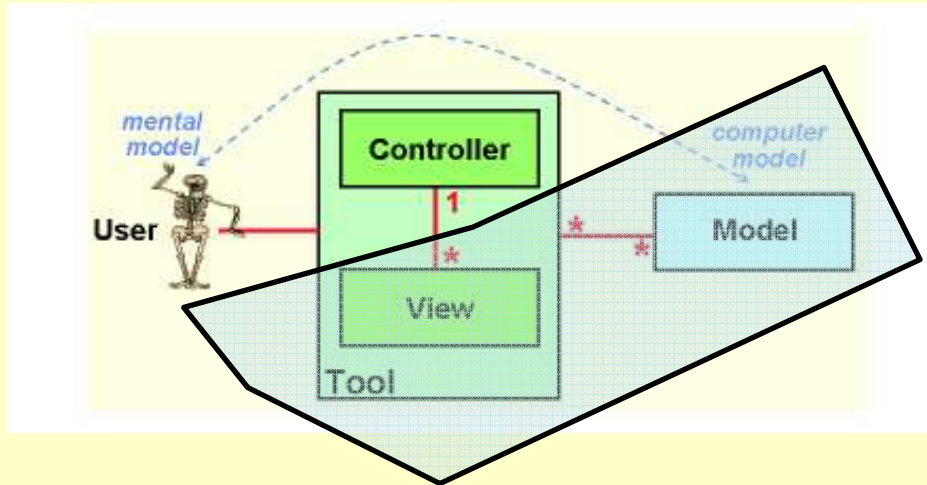
Patron: Modèle-Vue-Contrôleur

Modèle-Vue-Contrôleur (MVC) est un pattern architectural qui sépare les données (le modèle), l'interface homme-machine (la vue) et la logique de contrôle (le contrôleur).

Trygve Reenskaug - **MVC XEROX PARC 1978-79:**

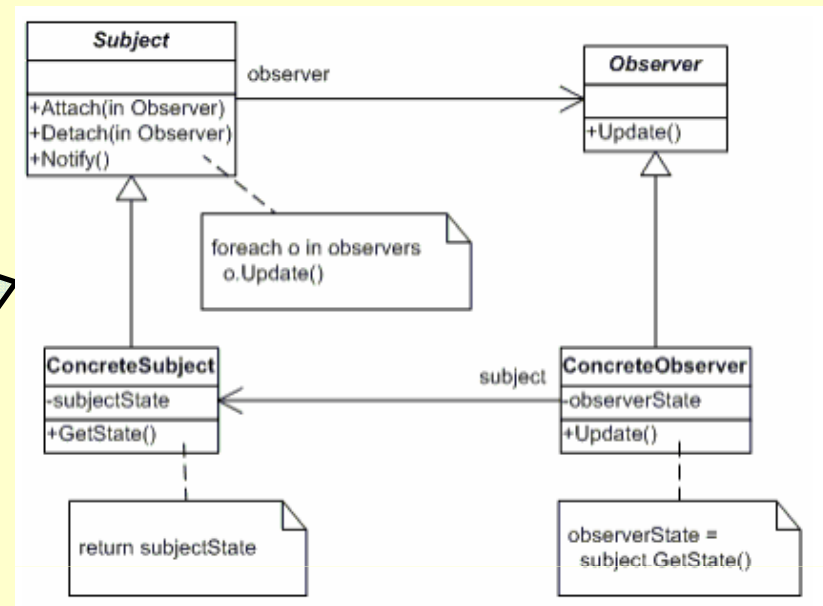
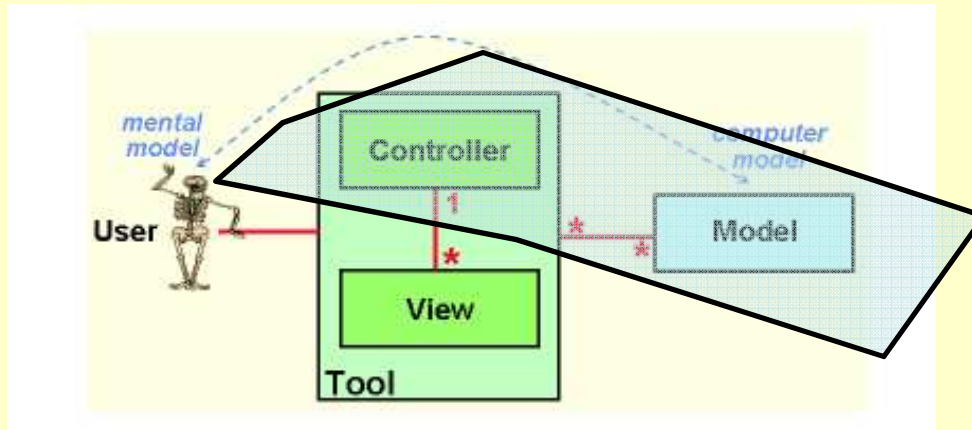


Patron: **observer** (vue-modèle)



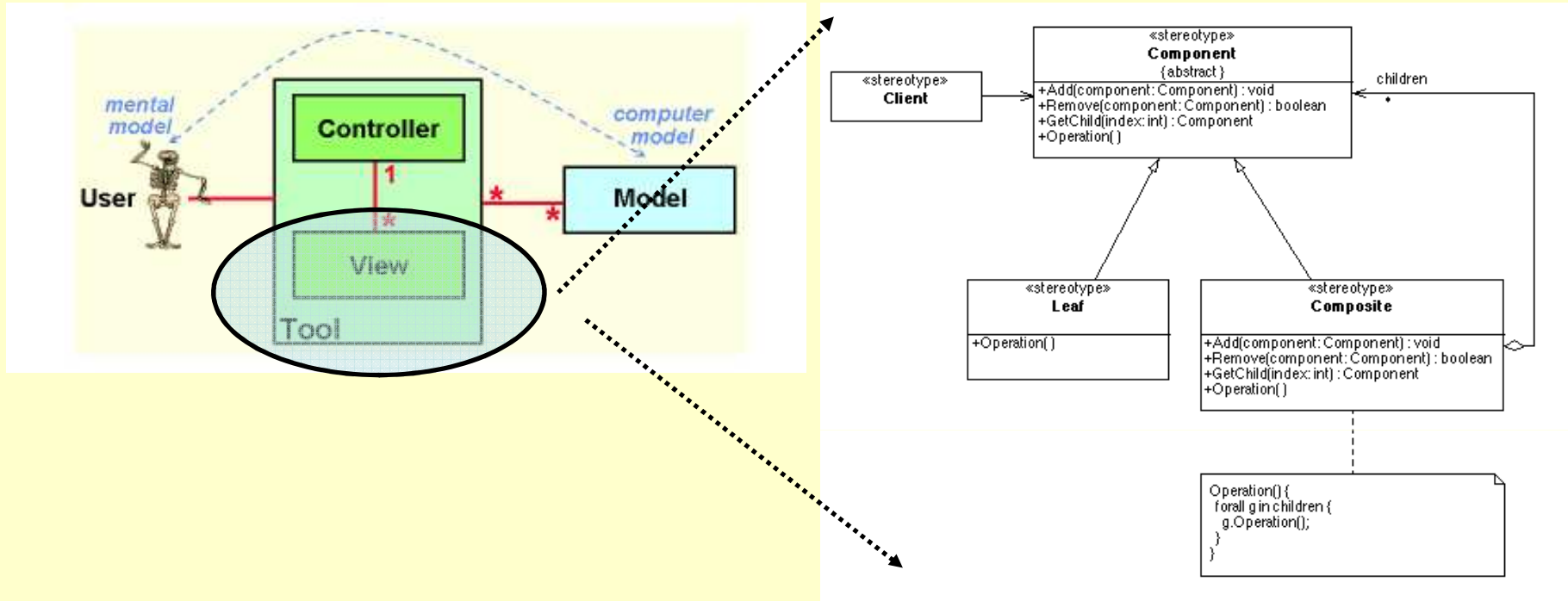
L'application (que nous voulons développer) doit pouvoir gérer plusieurs vues simultanément. Dès que le modèle est modifié, on veut que les vues se redessinent de manière adéquate.

Patron: **observer** (contrôleur-modèle)



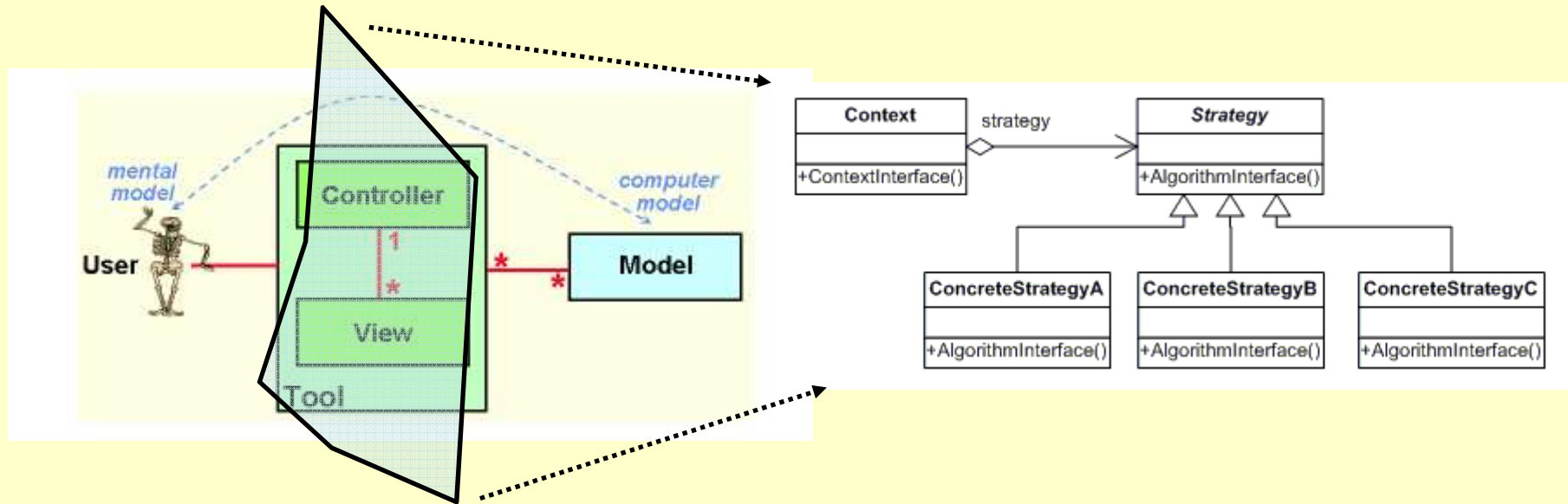
Il peut arriver que l'état du modèle ait un impact sur les contrôleurs. Par exemple, il se peut qu'en fonction du modèle, certaines options ne soient pas disponibles (menus, cases à cocher...). Les contrôleurs doivent donc être avertis de toutes les modifications du modèle. Utilisez le patron **observer** pour modéliser ce lien entre modèles et contrôleurs.

Patron: composite (vue-vue)



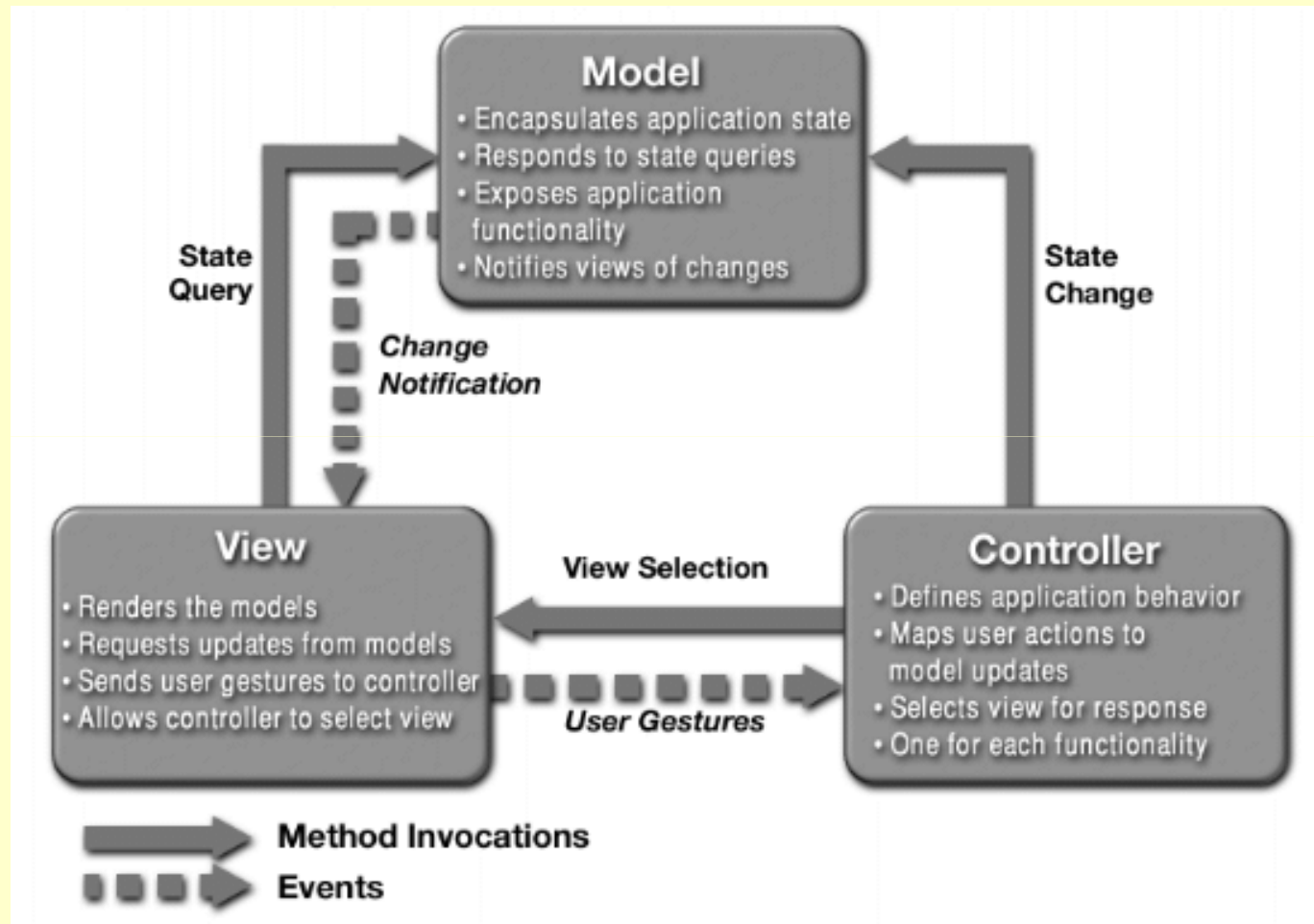
On considère que chaque composant peut l'objet d'une vue et que la vue en question peut contenir des sous-vues représentant les sous-composants

Patron: **strategy** (vue-contrôleur)

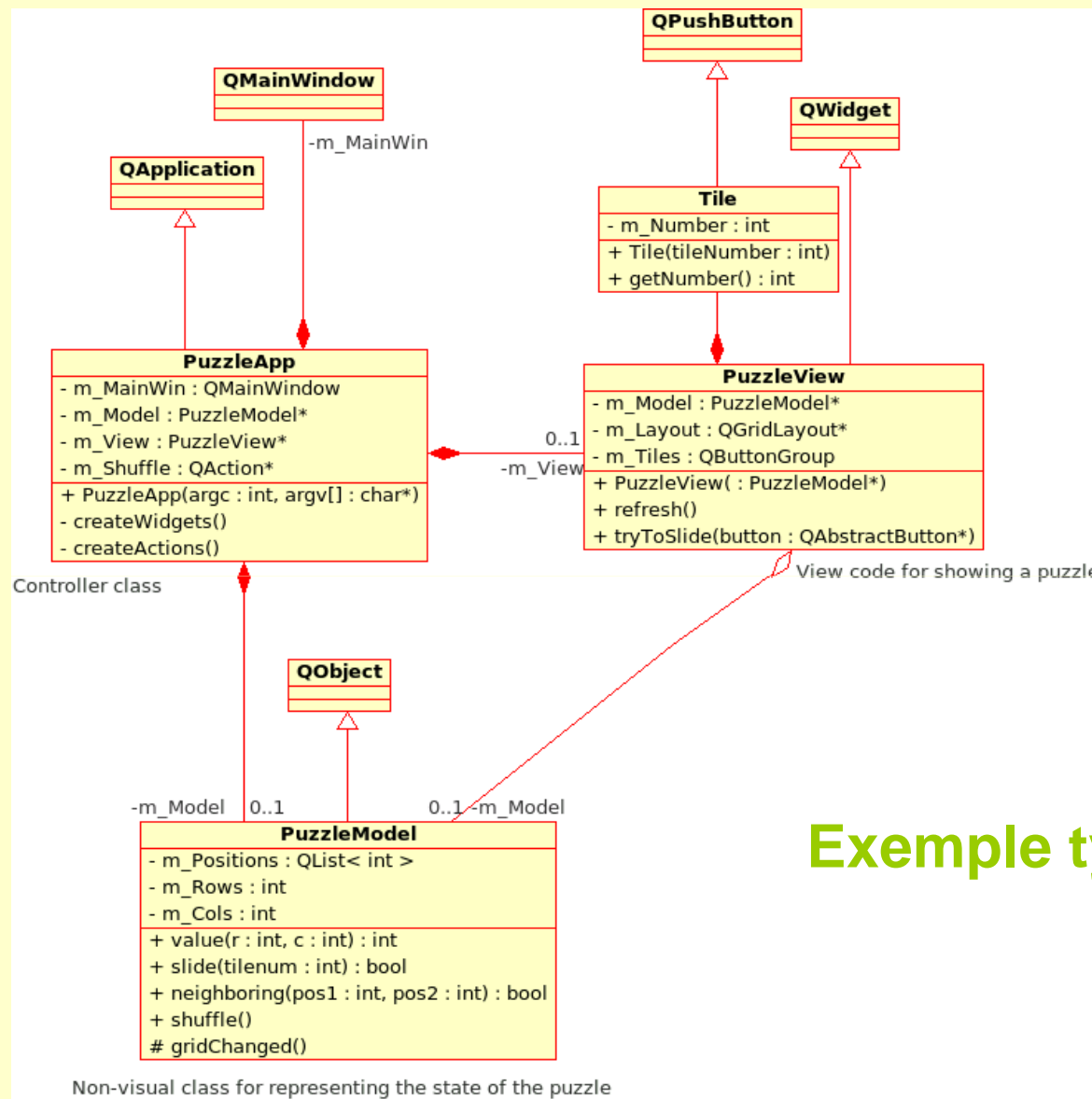


On souhaite disposer de moyens d'interaction avec le système pour modifier le modèle. Pour ce faire, on se dote de contrôleurs associés aux vues. Les contrôleurs réagissent aux événements extérieurs (clics sur des boutons, frappes clavier...) et modifient le modèle en réponse aux événements en question. La manière dont un contrôleur agit sur le modèle dépend de la vue à laquelle il est associé.

Patron: Modèle-Vue-Contrôleur



Patron: Modèle-Vue-Contrôleur (en UML)



Exemple typique: puzzle

Travaux Pratiques - MVC for temperature sensor

Work (in pairs, if you wish) using Eclipse

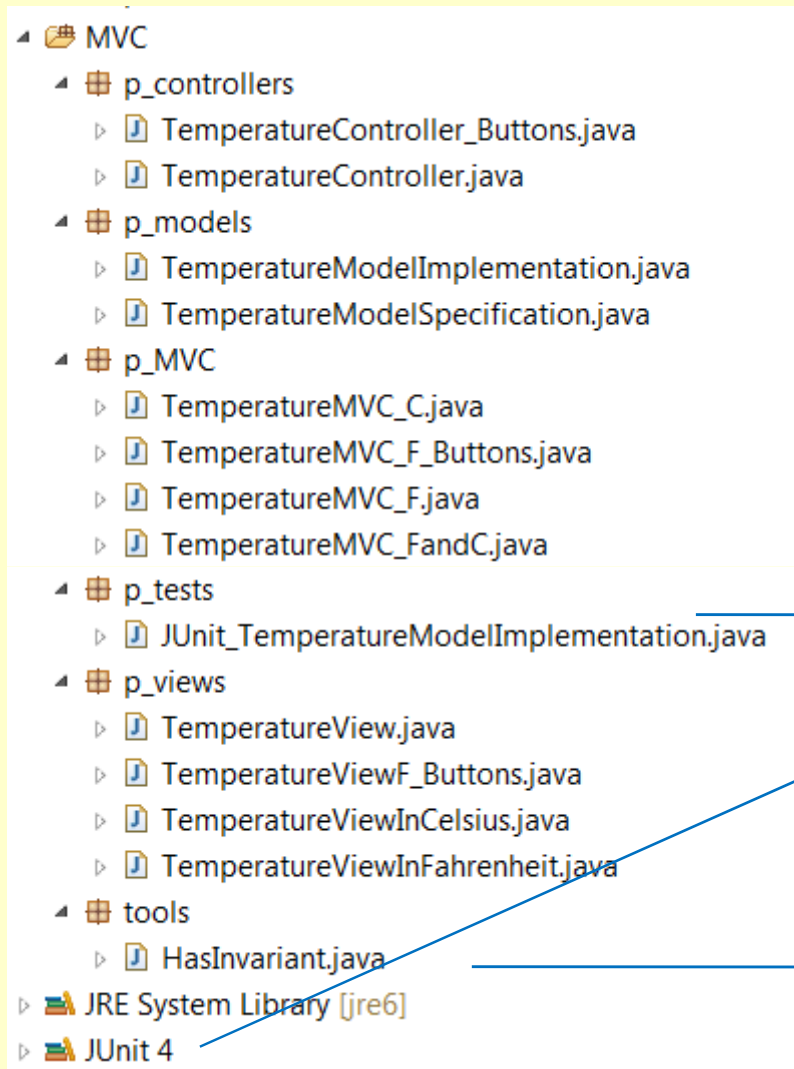
Version 1 : Simple Single Text I/O Window

- Interface TemperatureModelSpecification,
- Class TemperatureModelImplementation
- Class TemperatureViewInFahrenheit
- Class TemperatureController
- Class TemperatureMVC_F

Import these into a new project. Compile and run. Test the behaviour and try to understand how it works

Files can be found in [SourceMVC.zip](#)

Note: the package structure



Good habit:
test models
before
building MVC

Good habit:
rigorous development

Travaux Pratiques - MVC for temperature sensor

```
/**
 * @author J Paul Gibson
 * @version 1
 * MVC design pattern example for a temperature gauge with a single Fahrenheit view.<br>
 * To set up the pattern we:
 * <ul> <li> Construct the model</li>
 *     <li> Construct the view and connect to model</li>
 *     <li> Construct the controller and connect to model and view</li>
 *     <li> Connect the view and the controller </li> </ul>
 * @see TemperatureModelImplementation
 * @see TemperatureViewInFahrenheit
 * @see TemperatureController
 */
public class TemperatureMVC_F{
```

p_MVC.TemperatureMVC_F

Version:

- 1 MVC design pattern example for a temperature gauge with a single Fahrenheit view.
To set up the pattern we:
- Construct the model
 - Construct the view and connect to model
 - Construct the controller and connect to model and view
 - Connect the view and the controller

Author:

J Paul Gibson

See Also:

[TemperatureModelImplementation](#)
[TemperatureViewInFahrenheit](#)
[TemperatureController](#)

Travaux Pratiques - MVC for temperature sensor

```
public static void main(String args[]) {  
    /**  
     * The model  
     */  
    TemperatureModelImplementation model = new TemperatureModelImplementation();  
  
    /**  
     * The view  
     */  
    TemperatureViewInFahrenheit view = new TemperatureViewInFahrenheit(model, 100, 100);  
  
    /**  
     * The controller reacts to changes in the text (in the view) and to closing the window  
     */  
    TemperatureController controller1 = new TemperatureController(model, view);  
    view.addWindowListenerToFrame(controller1);  
}  
}
```

Travaux Pratiques - MVC for temperature sensor



Initial Value



Minimum Enforced



Maximum Enforced



Default Value



2 Decimal Places

TP - MVC for temperature sensor: the model

```
import java.util.Observer;
import tools.HasInvariant;
* @author J Paul Gibson
* @version 1
* Specification of different implementations of observable temperature
* behaviour
*/
public interface TemperatureModelSpecification extends HasInvariant { ... }
```

p_models.TemperatureModelSpecification

Specification of a temperature model for use in teaching MVC design pattern

Version:

1

Author:

J Paul Gibson

TP - MVC for temperature sensor: the model

```
/**
 * The minimum temperature value permitted (in Fahrenheit)
 */
public final static double MIN_TEMP_FAHRENHEIT = -100;

/**
 * The maximum temperature value permitted (in Fahrenheit)
 */
public final static double MAX_TEMP_FAHRENHEIT = 300;

/**
 * The default temperature value (in Fahrenheit)
 */
public final static double DEFAULT_TEMP_FAHRENHEIT = 32.0;
```

TP - MVC for temperature sensor: the model

```
/**
 * @return the current temperature (in Fahrenheit)
 */
public double getF();

/**
 * @return the current temperature (in Celsius)
 */
public double getC();

/**
 * @param tempF is the new temperature that would like to be set (in Fahrenheit).
 */
public void setF(double tempF);

/**
 *
 * @param tempC is the new temperature that would like to be set (in Celsius).
 */
public void setC(double tempC);
```

TP - MVC for temperature sensor: **the model**

```
/**
 * @return maximum temperature reached
 */
public double getMaxReachedF();

/**
 * @return minimum temperature reached
 */
public double getMinReachedF();

/**
 * @param o is an observer of the model
 */
public void addObserver(Observer o);
}
```

TP - MVC for temperature sensor: **the model**

p_models.TemperatureModelImplementation

A simple model for setting, storing and converting temperatures
Minimum and maximum values are also specified and enforced
The model can be observed, and notifies observers of state changes.

Version:

1

Author:

J Paul Gibson

See Also:

[TemperatureView](#)

[TemperatureController](#)

TP - MVC for temperature sensor: the model

```
public class TemperatureModelImplementation extends
Observable implements TemperatureModelSpecification{

/**
 * The temperature (initialised to {@link
TemperatureModelSpecification#DEFAULT_TEMP_FAHRENHEIT})
 */
private double temperatureF = DEFAULT_TEMP_FAHRENHEIT;

/**
 * The minimum value witnessed since the model has been
created
 */
private double minReachedF = temperatureF;

/**
 * The maximum value witnessed since the model has been
created
 */
private double maxReachedF = temperatureF;
```

TP - MVC for temperature sensor: the model

```
/**
 * @param f is the input temperature in Fahrenheit
 * @return the temperature in Celsius
 */
public static double fahrenheitToCelsius(double f){
return (f - 32.0) * 5.0 / 9.0;}

/**
 * @param c is the input temperature in Celsius
 * @return the temperature in Fahrenheit
 */
public static double celsiusToFahrenheit(double c){
return (c*9.0/5.0)+32;}
}
```

TP - MVC for temperature sensor: the model

```
/**
 * @param tempF is the new temperature that would like to be set (in Fahrenheit).<br>
 * If this value is greater than the maximum then we set the maximum value<br>
 * If this value is less than the minimum then we set the minimum value<br>
 * We also update the max/min values reached (if necessary)<br>
 * Finally, we notify the object's observers that its state has updated.
 */
public void setF(double tempF){
    temperatureF = tempF;

    if (temperatureF > maxTempF) temperatureF = maxTempF;
        else if (temperatureF < minTempF) temperatureF = minTempF;

    if (temperatureF > maxReachedF) maxReachedF = temperatureF;
        else if (temperatureF < minReachedF) minReachedF = temperatureF;

    setChanged(); notifyObservers();
}
```

TP - MVC for temperature sensor: **the model**

```
/**
 *
 * @param tempC is the new temperature that would like to be
 set (in Celsius).<br>
 * If this value is greater than the maximum then we set the
 maximum value<br>
 * If this value is less than the minimum then we set the
 minimum value<br>
 * We also update the max/min values reached (if
 necessary)<br>
 * Finally, we notify the object's observers that its state
 has updated.
 */
public void setC(double tempC){setF (tempC*9.0/5.0 + 32.0);}
```

TP - MVC for temperature sensor: **the model**

```
/**
 * @return maximum temperature reached
 */
public double getMaxReachedF() { return maxReachedF; }

/**
 * @return minimum temperature reached
 */
public double getMinReachedF() { return minReachedF; }
}
```

TP - MVC for temperature sensor: **the view**

© p_views.TemperatureViewInFahrenheit

Version:

1 A Simple Graphical View (in Fahrenheit) for the [TemperatureModelImplementation](#)
Generates a frame in which a temperature text field can be edited by the user.

Author:

J Paul Gibson

See Also:

[TemperatureModelImplementation](#)

[TemperatureController](#)

TP - MVC for temperature sensor: the **view**

```
public class TemperatureViewInFahrenheit extends TemperatureView{

/**
 * Display format temperature to 2 decimal places
 */
    private static final DecimalFormat df = new DecimalFormat("#.##");

/**
 * We choose to represent the temperature (in Fahrenheit) to 2
decimal places.
 */
    public TemperatureViewInFahrenheit(TemperatureModelImplementation
model, int h, int v){
        super("Temperature (Fahrenheit)", model, h, v);
        updateText();
    }
}
```

TP - MVC for temperature sensor: the **view**

```
public abstract class TemperatureView{

    /**
     * The temperature model for viewing
     */
    protected TemperatureModelSpecification model;

    /**
     * The graphics frame
     */
    protected Frame temperatureFrame;

    /**
     * For reading and editing the textual representation
     of the temperature
     */
    protected TextField display;
```

TP - MVC for temperature sensor: the **view**

```
/**
 * We choose to represent the temperature to 2 decimal
places.
 * The frame size is 350*100 pixels
 * @param label is the string label of the frame
 * @param model is the temperature model for viewing
 * @param h is the horizontal position of frame in screen
 * @param v is the vertical position of frame in screen
 */
TemperatureView(String label, TemperatureModelSpecification
model, int h, int v){
    this.model = model;
        display = new TextField();
        temperatureFrame = new Frame(label);
temperatureFrame.add("Center", display);
temperatureFrame.setSize(350,100);
temperatureFrame.setLocation(h, v);
temperatureFrame.setVisible(true);
}
```

TP - MVC for temperature sensor: the view

```
/**
 * @param tc will listen to the view frame and react to the events generated
 */
public void addWindowListenerToFrame(TemperatureController tc){
    temperatureFrame.addWindowListener(tc);
}

/**
 * @return the text display
 */
public TextField getDisplay(){return display;}

/**
 *
 * @return the value represented by the string in the text field display
 */
abstract public double getDisplayValueinF();

/**
 * This abstract method for representing the temperature on the screen is to be
 * defined in each view subclass
 */
abstract public void updateText();
}
```

TP - MVC for temperature sensor: the **controller**

© **p_controllers.TemperatureController**

Version:

1 Listen for window closing events of the [TemperatureView](#) and observe and react to changes to the [TemperatureModelSpecification](#) state

Author:

J Paul Gibson

TP - MVC for temperature sensor: the **controller**

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.text.DecimalFormat;
import java.util.Observable;
import java.util.Observer;

/**
 * Listen for window closing events of the {@link TemperatureView1}
 * and observe and react to changes to the {@link
TemperatureModelImplementation} state
 */

public class TemperatureController extends WindowAdapter implements
Observer, ActionListener{

    TemperatureView view;

    TemperatureModelImplementation model;
```

TP - MVC for temperature sensor: the **controller**

```
/**
 * @param model to be updated by controller in response to events
 * @param view which may generate events for the controller to respond to
 */
public TemperatureController(TemperatureModelSpecification model,
TemperatureView view) {
    this.model = model;
    this.view = view;
    // Connect the Controller as observer of the Model
    model.addObserver(this);

    // Connect the Controller as listener of the View display
    view.getDisplay().addActionListener(this);
}
```

TP - MVC for temperature sensor: the **controller**

```
public void update(Observable t, Object o) {  
  
    view.updateText();  
}  
  
public void actionPerformed(ActionEvent e) {  
    double value = view.getDisplayValue();  
    model.setF(value);  
}  
  
public void windowClosing(WindowEvent e) {  
    e.getWindow().setVisible(false);  
    System.exit(0);  
}
```

TP - MVC for temperature sensor: a second view

TO DO: create a second view class to show temperature in **Celsius**, and connect the 2 different views to the same model

HINT: the model provides some useful methods -

- `void p_MVC.TemperatureModel1.setF(double tempF)`

Specified by: [setF\(...\)](#) in [TemperatureModel](#)

Parameters:

tempF is the new temperature that would like to be set (in Fahrenheit).
If this value is greater than the maximum then we set the maximum value
If this value is less than the minimum then we set the minimum value
We also update the max/min values reached (if necessary)
Finally, we notify the object's observers that its state has updated.

- `void p_MVC.TemperatureModel1.setC(double tempC)`

Specified by: [setC\(...\)](#) in [TemperatureModel](#)

Parameters:

tempC is the new temperature that would like to be set (in Celsius).
If this value is greater than the maximum then we set the maximum value
If this value is less than the minimum then we set the minimum value
We also update the max/min values reached (if necessary)
Finally, we notify the object's observers that its state has updated.

- ^S `double p_MVC.TemperatureModel1.CelsiusToFahrenheit(double c)`

Parameters:

c is the input temperature in Celsius

Returns:

the temperature in Fahrenheit

- `double p_MVC.TemperatureModel1.getF()`

Specified by: [getF\(\)](#) in [TemperatureModel](#)

Returns:

the current temperature (in Fahrenheit)

- `double p_MVC.TemperatureModel1.getC()`

Specified by: [getC\(\)](#) in [TemperatureModel](#)

Returns:

the current temperature (in Celsius)

- ^S `double p_MVC.TemperatureModel1.FahrenheitToCelsius(double f)`

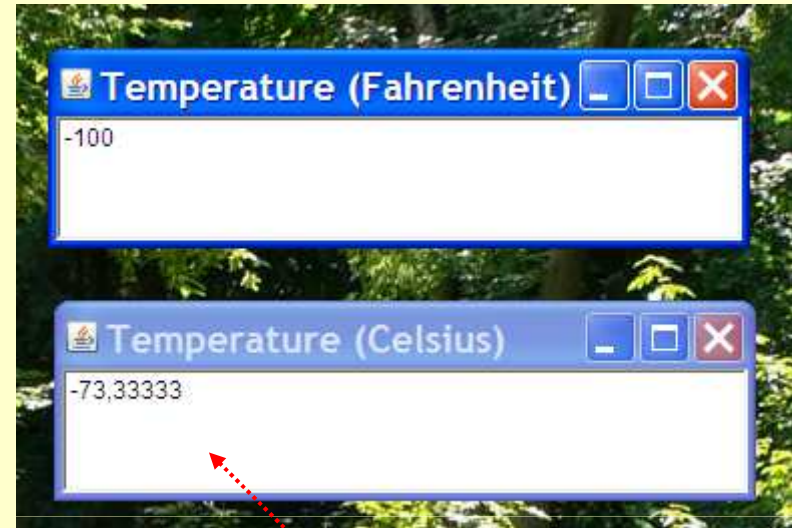
Parameters:

f is the input temperature in Fahrenheit

Returns:

the temperature in Celsius

TP - MVC for temperature sensor: two views of same model



Additional requirement for Celsius View: represent to **5 decimal places**



TP - MVC for temperature sensor: two views of same model

```
/**
 * @author J Paul Gibson
 * @version 1
 * MVC design pattern example for a temperature gauge with 2 views: Fahrenheit and Celsius
 * To set up the pattern we:
 * <ul> <li> Construct the model</li>
 *     <li> Construct the Fahrenheit view and connect to model</li>
 *     <li> Construct the Celsius view and connect to model</li>
 *     <li> Construct the Fahrenheit controller and connect to model and view</li>
 *     <li> Construct the Celsius controller and connect to model and view</li>
 *     <li> Connect the views and the controllers </li> </ul>
 * @see TemperatureModelImplementation
 * @see TemperatureViewInFahrenheit
 * @see TemperatureController
 */
public class TemperatureMVC_FandC{
public static void main(String args[]) {
TemperatureModelImplementation model = new TemperatureModelImplementation();
TemperatureViewInFahrenheit view1= new TemperatureViewInFahrenheit(model, 100,
100);
TemperatureViewInCelsius view2= new TemperatureViewInCelsius(model, 100, 300);
TemperatureController controller1 = new TemperatureController(model, view1);
TemperatureController controller2 = new TemperatureController(model, view2);
view1.addWindowListenerToFrame(controller1);
view2.addWindowListenerToFrame(controller2);
}
}
```

TP - MVC for temperature sensor: *extend the views with buttons*



p_MVC.TemperatureMVC_F_Buttons

Version:

1 Extends MVC behaviour of [TemperatureMVC_F](#) with buttons.

The controller reacts to:

- changes in the text (in the view)
- closing the window
- pressing either of the 2 buttons

Author:

J Paul Gibson

See Also:

[TemperatureViewF_Buttons](#)

[TemperatureController_Buttons](#)

TP - MVC for temperature sensor: *extend the views with buttons*

```
public class TemperatureMVC_F_Buttons extends TemperatureMVC_F {

public static void main(String args[]) {
/**
 * The model
 */
TemperatureModelImplementation model = new TemperatureModelImplementation();

/**
 * The view - with buttons for incrementing and decrementing the
 * temperature value
 */
TemperatureViewF_Buttons view = new TemperatureViewF_Buttons(model, 100, 100);

TemperatureController controller =
    new TemperatureController_Buttons(model, view);
view.addWindowListenerToFrame(controller);
}
```

TP - MVC for temperature sensor: *extend the views with buttons*

```
/**
 * @author J Paul Gibson
 * @version 1
 * extends {@link TemperatureViewInFahrenheit} with
 * 2 buttons for incrementing and decrementing the temperature
 */
class TemperatureViewF_Buttons extends TemperatureViewInFahrenheit{

/**
 * For incrementing the temperature value
 */
    private Button upButton;
/**
 * For decrementing the temperature value
 */
    private Button downButton;

/**
 * @return the upButton
 */
    public Button getUpButton () { return upButton; }

/**
 * @return the downButton
 */
    public Button getDownButton () { return downButton; }
```

TP - MVC for temperature sensor: *extend the views with buttons*

```
/**
 * Add up and down buttons to a panel in the bottom of the
 * view frame.<br>
 * Constructs the rest of the view as specified in
 * {@link TemperatureViewInFahrenheit}
 */
public TemperatureViewF_Buttons(TemperatureModelImplementation model,
int h, int v){
    super(model,h,v);
    upButton = new Button(" > ");
    downButton = new Button(" < ");
    Panel buttons = new Panel();
    buttons.add(upButton);
    buttons.add(downButton);
    temperatureFrame.add("South", buttons);
}
}
```

TP - MVC for temperature sensor: *extend the views with buttons*

```
/**
 * @author J Paul Gibson
 * @version 1
 * Extends the {@link TemperatureController} with (re)actions
 * linked to Button presses
 */
public class TemperatureController_Buttons extends TemperatureController {

    public TemperatureController_Buttons(TemperatureModel model,
    TemperatureViewInFahrenheit view) {
        super(model,view);}

    /**
     * Extends the {@link TemperatureController} with 2 Buttons
     * @param model is the temperature model to be represented in the view
     * @param view is the text frame extended with 2 buttons
     */
    public TemperatureController_Buttons(TemperatureModel model,
    TemperatureViewF_Buttons view) {
        super(model,view);
        view.getUpButton().addActionListener(this);
        view.getDownButton().addActionListener(this);
    }
}
```

TP - MVC for temperature sensor: *extend the views with buttons*

```
/**
 * Distinguishes between the 3 actions that can be generated at the view:
 * <ul> <li> Pressing the up button - increment the temperature in Fahrenheit
 * </li><li> Pressing the down button - decrement the temperature in Fahrenheit
 * </li><li> Changing the text - as in superclass
 * </li></ul>
 * @param e is the action event generated at the view
 */
public void actionPerformed(ActionEvent e){
    double value = view.getDisplayValueinF();
    if (e.getActionCommand().equals(" > ")) model.setF(model.getF()+1);
    else if (e.getActionCommand().equals(" < ")) model.setF(model.getF()-1);
    else model.setF(value);
    }
}
```

TP - MVC for temperature sensor: *extend the views with buttons*

TO DO: Extend the Celsius View with buttons in the same way and test both views together

