**Caption Motion and the Observer Patterns**

The caption in our GUI is a JLabel that listens for text changes from the JTextField labeled “Caption:”. Since we display the text using an object that extends JComponent, we utilize the Observer pattern by attaching listeners to it.

There are two objects that must listen with caption motion. The first is obviously the caption itself; the second more subtly is the object that stores the caption location to the slideshow file. Both will need to be in sync with each other; to achieve this, we created a nested, abstract class “CaptionLabelMouseInputAdapter” that extends the Java abstract class MouseInputAdapter (see Appendix A or the file “SlideShowImagePanel.java” for the source code). This abstract class encapsulates all necessary mouse motion calculations into a single location, which the two objects that must listen for mouse motion then extend to include the additional functionality they individually need.

The caption extends CaptionLabelMouseInputAdapter through an anonymous class in function “addMouseInputListenerToCaption”, which is in the file “SlideShowImagePanel.java” and as a courtesy included in Appendix B. The only interface method the caption JLabel needs to be extend is “mouseDragged”; in this function, if the caption has moved (as calculated by the abstract class functions), the caption location is updated. After the anonymous class object is created, it is added to the caption JLabel through functions “addMouseListener” and “addMotionMouseListener”.

The file contents tracker similarly extends CaptionLabelMouseInputAdapter through an anonymous class in function “addMouseInputListenerToCaptionForFileContents”, which is in the file “SlideShowContentsPanel.java” and included as a courtesy in Appendix C. The only interface method that needed to be extended was “mouseReleased” since the slideshow file contents are only concerned with the final location of the caption after dragging and not any intermediary locations. This function is called in the GUI’s main function after creating the JPanel, which contains the displayed image and caption.

**Summary:** All mouse inputs and tracking is done using listeners; this approach utilizes the Observer pattern.

**The Undo Buffer and the Command and Observer Patterns**

The Command pattern encapsulates actions (e.g. user interaction) into an object. In alignment with this approach, we defined an interface type “SlideShowGUICommand” to define any undo-able actions (see Appendix D or file “SlideShowGUICommand.java” for the source code); this interface has two methods “execute” and “undo”. By using an interface type, we could have different undoable operations handled using a single architecture.

Since the undo stack has a maximum depth (e.g. 10) and the oldest object (i.e. bottom of the stack) is deleted whenever a new object is added to a full stack, we implemented our undo stack using a circular buffer; at its lowest level, the buffer is an ArrayList that stores objects of the interface type “SlideShowGUICommand”. This functionality is encapsulated with the Undo button in the SlideShowEditJMenu class.

Saving changes to a slide is one of the undoable commands. To encapsulate this functionality, we created the SaveGUICommand class, which implements the SlideShowGUICommand interface (see Appendix E or file “SlideShowContentsPanel.java” for the source code). In order to be able to undo a “Save Image” operation, the SlideShowGUICommand object stores the previous slideshow image information. In order to undo the save image operation, the object only needs to call the undo method.

The “Undo” menu selection is a JMenuItem that is part of the overall “Edit” menu. We created a custom class “SlideShowEditJMenu” to implement the undo functionality (see the source code in file “SlideShowEditJMenu.java”). There is an ActionListener attached to the Undo JMenuItem; it is created in the method “createUndoActionListener” (see Appendix F or file for the source code). Whenever the Undo button is pressed, it removes one SlideShowGUICommand object from the undo buffer (see class UndoBuffer in “SlideShowEditJMenu.java”), and calls the undo method on the object. The ActionListener then checks if the undo buffer is empty; if it is, the listener disables the “Undo” button.

The Undo menu also has a listener attached to the “New” menu option. The listener is created in the method “generateClearUndoBufferActionListener”, which is in file “SlideShowEditJMenu.java”. Similarly, the Undo menu has ActionListeners attached to the JFileChooser objects for the Open and “Save” JMenuItems; these ActionListener objects are created in the method “generateFileChooserClearUndoBufferActionListener”. The three previously described ActionListeners are added to the JMenuItem objects in the constructor for the SlideShowJMenuBar; the specific function we created to add these ActionListeners is named “addActionListener” and is in file “SlideShowJMenuBar.java”.

Appendix – CaptionLabelMouseInputAdapter Source Code

**public** **abstract** **class** CaptionLabelMouseInputAdapter **extends** MouseInputAdapter{

//---- Store initial information about the caption

**private** **int** initialCaptionX;

**private** **int** initialCaptionY;

//---- Store the last mouse position

**private** **int** lastMouseX;

**private** **int** lastMouseY;

//---- Store Caption information

**private** **int** latestCaptionX;

**private** **int** latestCaptionY;

**private** **boolean** captionMoved;

//----

**boolean** mouseOutsideValidArea;

/\*\*

\* Stores the initial X and Y location when the mouse is pressed.

\*/

@Override

**public** **void** mousePressed(MouseEvent e){

//---- Initially latest and initial position are the same as the the caption's current position

latestCaptionX = initialCaptionX = captionLabel.getX();

latestCaptionY = initialCaptionY = captionLabel.getY();

//----- Get the mouse location information

lastMouseX = e.getXOnScreen();

lastMouseY = e.getYOnScreen();

//---- By default caption not moved

captionMoved = **false**;

//---- Mouse still in valid area

mouseOutsideValidArea = **false**;

}

@Override

**public** **void** mouseDragged(MouseEvent e){

//---- Check if the mouse position is invalid

**if**(mouseOutsideValidArea && getMousePosition(**true**) == **null**) **return**;

//---- Get the caption location

Point captionLocation = captionLabel.getLocation();

**int** captionXLoc = (**int**)captionLocation.getX();

**int** captionYLoc = (**int**)captionLocation.getY();

//---- Get the newX and newY locations for the caption label

**int** newX = captionXLoc + (e.getXOnScreen() - lastMouseX);

**int** newY = captionYLoc + (e.getYOnScreen() - lastMouseY);

//---- Handle the case where the cursor just reentered the valid space

**if**(mouseOutsideValidArea){

//--- Handle default case where the mouse did not re-enter the valid space.

**if**(newX < captionLabel.getMinimumXLocation() || newX > captionLabel.getMaximumXLocation())

newX = captionXLoc + (e.getX());

**if**(newY < captionLabel.getMinimumYLocation() || newY > captionLabel.getMaximumYLocation())

newY = captionYLoc + (e.getY());

}

//----- Ensure the mouse is not too far away from the caption

Point mousePositionInPanel = getMousePosition(**true**);

**if**(mousePositionInPanel != **null**){

//----- Ensure the mouse and CaptionLabel are not separated too much in the X direction.

**int** mouseComponentXDistance = (**int**)mousePositionInPanel.getX() - newX;

**if**( mouseComponentXDistance < 0 || mouseComponentXDistance > captionLabel.getWidth()){

newX += mouseComponentXDistance;

}

//----- Ensure the mouse and CaptionLabel are not separated too much in the Y direction.

**int** mouseComponentYDistance = (**int**)mousePositionInPanel.getY() - newY;

**if**( mouseComponentYDistance < 0 || mouseComponentYDistance > captionLabel.getHeight()){

newY += mouseComponentYDistance;

}

}

//---- Update X location

**if**(newX < captionLabel.getMinimumXLocation()){

newX = captionLabel.getMinimumXLocation();

}

**else** **if**(newX > captionLabel.getMaximumXLocation()){

newX = captionLabel.getMaximumXLocation();

}

lastMouseX = e.getXOnScreen();

latestCaptionX = newX;

//---- Update Y location

**if**(newY < captionLabel.getMinimumYLocation()){

newY = captionLabel.getMinimumYLocation();

}

**else** **if**(newY > captionLabel.getMaximumYLocation()){

newY = captionLabel.getMaximumYLocation();

}

lastMouseY = e.getYOnScreen();

latestCaptionY = newY;

//----- Check if the caption moved. May not move if you are at the boundary.

**if**(captionXLoc != newX || captionYLoc != newY) captionMoved = **true**; //---- Mark caption moved.

//---- Check if the mouse left the valid area

**if**(getMousePosition(**true**) == **null**)

mouseOutsideValidArea = **true**;

**else**{

mouseOutsideValidArea = **false**;

}

}

/\*\*

\* Gets the initial X and Y location of the Caption.

\* **@return** Point containing initial X and Y location of the captionLabel

\*/

**public** Point getInitialCaptionLocation(){ **return** **new** Point(initialCaptionX, initialCaptionY); }

/\*\*

\* Gets the final X and Y location of the Caption.

\* **@return** Point containing final X and Y location of the captionLabel

\*/

**public** Point getFinalCaptionLocation(){ **return** **new** Point(latestCaptionX, latestCaptionY); }

/\*\*

\* After the mouse has been released, this function is used to determine whether the caption moved.

\* **@return** Boolean value of whether the caption moved during the mouse dragging

\*/

**public** **boolean** getDidCaptionMove(){ **return** captionMoved; }

}

Appendix – addMouseInputListenerToCaption Source Code

/\*\*

\* Creates a MouseListener and MouseMotionListener and adds it to CaptionLabel.

\*/

**private** **void** addMouseInputListenerToCaption(){

//-- Create an anonymous object to listen for mouse motions.

MouseInputAdapter captionListener = **new** CaptionLabelMouseInputAdapter(){

@Override

**public** **void** mouseDragged(MouseEvent e){

//---- Calculate the new caption locations

**super**.mouseDragged(e);

//----- If the caption moved, update its location

**if**(getDidCaptionMove()){

captionLabel.setLocation(**this**.getFinalCaptionLocation());

}

}

};

//----- Create the mouse listener

captionLabel.addMouseListener(captionListener);

captionLabel.addMouseMotionListener(captionListener);

}

Appendix – addMouseInputListenerToCaptionForFileContents Source Code

/\*\*

\* Creates a mouse motion listener to extract the caption location for saving to the file.

\*/

**public** **void** addMouseInputListenerToCaptionForFileContents(SlideShowImagePanel imagePanel){

//-- Create an anonymous object to listen for mouse motions.

MouseInputAdapter captionListener = imagePanel.**new** CaptionLabelMouseInputAdapter(){

@Override

**public** **void** mouseReleased(MouseEvent e){

**super**.mouseReleased(e);

**if**(**this**.getDidCaptionMove()){

captionLocation = **this**.getFinalCaptionLocation();

}

}

};

imagePanel.addCaptionMouseInputListener(captionListener);

}

Appendix – SlideShowGUICommand Source Code

/\*\*

\* Interface type used to encapsulate undo-able actions.

\*

\* **@author** Zayd

\*

\*/

**public** **interface** SlideShowGUICommand {

/\*\*

\* Needs to be extended to perform the action.

\*/

**public** **void** execute();

/\*\*

\* Needs to be extended to undo the action performed by the "execute" method.

\*/

**public** **void** undo();

}

Appendix – SaveGUICommand Source Code

/\*\*

\*

\* This class is used to Save and Unsave Image Instance Saves.

\*

\* **@author** Zayd

\*

\*/

**private** **class** SaveGUICommand **implements** SlideShowGUICommand{

**private** SlideShowImageInstance previousImageInstance;

**private** SlideShowImageInstance newImageInstance;

/\*\*

\*

\* **@param** newImageInstance New Image Image Instance to be set.

\* **@param** previousImageInstance Previous Image Image Instance to be set.

\*/

**public** SaveGUICommand(SlideShowImageInstance newImageInstance,

SlideShowImageInstance previousImageInstance){

//---- Copy make clones of the input parameters.

**this**.newImageInstance = (SlideShowImageInstance)(newImageInstance.clone());

**this**.previousImageInstance = (SlideShowImageInstance)(previousImageInstance.clone());

}

@Override

**public** **void** execute() {

*slideShowFileContents*.setImageInstance(newImageInstance);

*slideShowListModel*.setElementAt(*slideShowFileContents*.getImageInstance(newImageInstance.getImageID()-1),

newImageInstance.getImageID()-1);

}

@Override

**public** **void** undo() {

*slideShowFileContents*.setImageInstance(previousImageInstance);

*slideShowListModel*.setElementAt(*slideShowFileContents*.getImageInstance(previousImageInstance.getImageID()-1),

previousImageInstance.getImageID()-1);

//---- Deselect then reselect the list item to get GUI to refresh

**if**(previousImageInstance.getImageID() - 1 == *slideShowList*.getSelectedIndex()){

*slideShowList*.clearSelection();

*slideShowList*.setSelectedIndex(previousImageInstance.getImageID() - 1);

}

}

}

Appendix – createUndoActionListener Source Code

/\*\*

\* Creates an ActionListener for the Undo Menu Item and then adds it to the menu item.

\*/

**private** **void** createUndoActionListener(){

ActionListener undoActionListener = **new** ActionListener(){

/\*\*

\* Gets the last action performed and then undoes it.

\*/

@Override

**public** **void** actionPerformed(ActionEvent e){

SlideShowGUICommand lastCommand = *undoBuffer*.remove();

lastCommand.undo();

//---- If the buffer is empty, disable the menu item.

**if**(*undoBuffer*.isEmpty()){

*undoMenuItem*.setEnabled(**false**);

}

}

};

//--- Add the ActionListener

*undoMenuItem*.addActionListener(undoActionListener);

}