```java
import java.util.HashMap;
import java.awt.Image;

public class IconCounterCounter
{
    public static int getCount(Image im)
    {
        Object o;
        if ((o = ourMap.get(im)) == null) {
            ourMap.put(im, o = new Counter());
        }
        else {
            ((Counter) o).incr();
        }
        return ((Counter) o).getValue();
    }

    private static HashMap ourMap = new HashMap();

    private static class Counter
    {
        Counter()
        {
            myCount = 0;
        }

        void incr()
        {
            myCount++;
        }

        int getValue()
        {
            return myCount;
        }

        int myCount;
    }
}
```



```java
import java.awt.Image;
import java.awt.MediaTracker;
import java.awt.Toolkit;
import java.awt.Component;
import javax.swing.ImageIcon;
import java.net.URL;

/**
 * Encapsulates image loading so clients can open an
 * image simply without resorting to MediaTrackers
 * or Icons.  In the current version of this class
 * the getImage(..) functions use the swing ImageIcon
 * class which uses a MediaTracker internally. This means
 * the MediaTracker code doesn't appear here (but does in ImageIcon).
 * <P>
 * These factory methods support getting an image by filename,
 * by url, and from an Applet/jar file using the Component/name
 * factory method.
 * <P>
 * <em>TODO: update to use jdk 1.4 imageIO classes</em>
 * <P>
 * @author Owen Astrachan
 */
public class ImageFactory
{
    /**
     * load an image from a file, return the image
     * waits for image to be loaded before returning
     * @param filename is the source of the image (e.g., foo.gif)
     * @return the image bound to filename
     */
    public static Image getImage(String filename)
    {
        return getImage(new ImageIcon(filename));
    }

    /**
     * load an image from a URL, return the image
     * waits for image to be loaded before returning
     * @param filename is the source of the image (e.g., foo.gif)
     * @return the image bound to filename
     */
    public static Image getImage(URL urlname)
    {
        return getImage(new ImageIcon(urlname));
    }

    /**
     * Load an image from a jar file, e.g., using an
     * object's getResource method. Used, for example,
     * by an Applet passing itself as the Component and
     * the name of the file in the jar file storing the
     * image.
     * @param thing is the Applet/JApplet/Component/Object
     * @param name is the jar file resource storing the image
     * @return the image
     */
    public static Image getImage(Object thing, String name)
    {
        return getImage(thing.getClass().getResource(name));
    }

    /**
     * Returns an image repesentation of the given icon.
     */
    private static Image getImage (ImageIcon icon)
    {
        return icon.getImage().getScaledInstance(PuzzleConsts.IMAGE_SIZE,
                                                 PuzzleConsts.IMAGE_SIZE,
                                                 Image.SCALE_REPLICATE);
    }
}
```

```java
import javax.swing.ImageIcon;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Component;
import java.awt.Image;

/**
 * An image icon with a scaled/placed part
 * of an image. An ImagePuzzleIcon is constructed
 * with an image and a size --- the size is the length
 * of one side of the grid/model the icon will be used with.
 * <P>
 * Each ImagePuzzleIcon associated with the same image has
 * a self-determined number so that each "knows" what part of
 * the image it represents. This knowledge is used to scale
 * the image and show one portion of it --- the portion represented
 * by a particulare ImagePuzzleIcon.
 * <P>
 * A static count/variable will not work for determining how many
 * pieces there are of a particular image. A static count would work
 * if there is only one image ever used. Suppose, however, that three
 * images are used. Each image might be divided into 25 pieces --- and
 * each ImagePuzzleIcon object must know which of the 25 pieces it is.
 * But, if numbered 0,1,...,24, there will be a total of 75 different
 * ImagePuzzleIcons if there are three such larger collections, and
 * each one of the three must track which part it is.
 *
 * @author Owen Astrachan
 */
public class ImagePuzzleIcon extends ImageIcon
{
    /**
     * construct from an image source with specified params
     */
    public ImagePuzzleIcon(Image im, String label,
                           int modelSize)
    {
        super(im);
        myLabel = label;
        init(modelSize);
    }

    private void init(int modelSize)
    {
        // determine what piece of image I am then adjust
        // my coordinates appropriately
        myCount = IconCounterCounter.getCount(getImage());
        mySize = PuzzleConsts.IMAGE_SIZE/modelSize;
        myX = myCount % modelSize * mySize;
        myY = myCount / modelSize * mySize;
    }

    /**
     * Paint me where I think I'm supposed to be
     * depending on what I am (blank or not)
     */
    public void paintIcon(Component c, Graphics g, int x, int y)
    {
        int fillSize = c.getWidth();
        if (! myLabel.equals(PuzzleConsts.BLANK)) {
            g.drawImage(getImage(),0,0,fillSize,fillSize,
                        myX,myY,myX+mySize,myY+mySize,c);
        }
        else {
            g.setColor(c.getBackground());
            g.fill3DRect(0,0,fillSize+PuzzleConsts.OFFSET,
                         fillSize+PuzzleConsts.OFFSET,true);
        }
    }

    protected int    myCount;      // what number image am I?
    protected int    myX;          // where am I (x,y) coordinates
    protected int    myY;
```



```java
    protected String myLabel;      // determines if I'm blank or not
    protected int    mySize;       // used to break up/draw scaled image
}
```

```java
import javax.swing.Icon;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Component;

/**
 * Not in image, just a numbered puzzle icon/tile.
 *
 * @author Owen Astrachan
 */

public class PlainPuzzleIcon implements Icon
{
    /**
     * Create a tile with specific label of designated
     * size.
     */
    public PlainPuzzleIcon(String label, int size)
    {
        myLabel = new String(label);
        mySize = size;
    }

    /**
     * Draws this plain icon at the proper size.
     * @param c is used to determine how big to draw this Icon
     * @param g is the graphics context in which drawing takes place
     */
    public void paintIcon(Component c, Graphics g, int x, int y)
    {
        int fillSize = c.getWidth();
        if (! myLabel.equals(PuzzleConsts.BLANK)) {
            doPaint(g,Color.blue,Color.yellow,fillSize);
        }
        else {
            doPaint(g,Color.yellow,Color.blue,fillSize);
        }
    }

    private void doPaint(Graphics g, Color background,
                         Color foreground, int fillSize)
    {
        g.setColor(background);
        g.fill3DRect(0,0,fillSize,fillSize,true);
        g.setColor(foreground);
        g.fill3DRect(PuzzleConsts.OFFSET,PuzzleConsts.OFFSET,
                     fillSize,fillSize,true);
        g.setColor(Color.black);
        g.drawString(myLabel,PuzzleConsts.OFFSET + fillSize/2,
                     PuzzleConsts.OFFSET + fillSize/2);
    }

    /**
     * Needed for interface, not used in this project.
     */
    public int getIconHeight()
    {
        return mySize;
    }

    /**
     * Needed for interface, not used in this project.
     */
    public int getIconWidth()
    {
        return mySize;
    }

    protected String myLabel;
    protected int    mySize;
}
```



```java
/**
 * Simple puzzle app which attaches three views
 * to one model and one controller.
 *
 * @author Owen Astrachan
 */
public class PuzzleApp
{
    public PuzzleApp()
    {
        PuzzleController control = new PuzzleController();
        PuzzleModel pmodel  = new PuzzleModel(control, 10);
        PuzzleGui   pgui    = new PuzzleGui(control, "ola.jpg");
        PuzzleGui   pgui2   = new PuzzleGui(control);
        PuzzleGui   pgui3   = new PuzzleGui(control, "mickey.gif");
    }

    public static void main(String args[])
    {
        new PuzzleApp();
    }
}
```

```java
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.*;
import java.awt.Image;
import javax.swing.*;



/**
 * n-puzzle with sliding images
 * @author Owen Astrachan
 */
public class PuzzleApplet extends JApplet implements PuzzleView
{
    public void init()
    {
        myControl = new PuzzleController();
        PuzzleModel pmodel = new PuzzleModel(myControl, getNumDivisions());
        doInitialize(pmodel, getImageName());
    }

    /**
     * Create a gui with a model and an image (image optional)
     * @param imageSource is null or the url/file for an image
     * @param model is the model for this view
     */
    public void doInitialize(PuzzleModel model, String imageSource)
    {
        JPanel panel = new JPanel(new BorderLayout());

        myControl.addView(this);
        myModel = model;
        myUndo = new JButton("undo");
        myUndo.addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e)
                {
                    myControl.undoMove();
                }
            });
        setEnabledUndo(false);

        myText = new JTextField(20);
        myButtonPanel = new ButtonPanel(myModel.getSize(), imageSource);
        panel.add(myButtonPanel, BorderLayout.CENTER);
        panel.add(myText,        BorderLayout.SOUTH);
        panel.add(myUndo,        BorderLayout.NORTH);

        setContentPane(panel);
        setVisible(true);
    }

    public void showText(String text)
    {
        myText.setText(text);
    }

    public void setEnabledUndo(boolean value)
    {
        myUndo.setEnabled(value);
    }

    public void showGrid(int[] list)
    {
        myButtonPanel.setGrid(list);
    }

    private String getImageName ()
    {
        // allow user to give puzzle image in HTML file
        String puzzleName = getParameter("image");
        if (puzzleName == null) {
            puzzleName = OUR_DEFAULT_IMAGE;
        }
```



```java
        return puzzleName;
    }

    private int getNumDivisions ()
    {
        // allow user to give puzzle image in HTML file
        int gridSize = 2;
        try {
            gridSize = Integer.parseInt(getParameter("divisions"));
        }
        catch (Exception e) {
            gridSize = OUR_DEFAULT_GRID;
        }
        return gridSize;
    }


    private PuzzleController myControl;
    private PuzzleModel myModel;
    private JTextField  myText;
    private ButtonPanel myButtonPanel;
    private JButton     myUndo;

    private static final int OUR_WIDTH  = 500;
    private static final int OUR_HEIGHT = 500;

    private static final String OUR_DEFAULT_IMAGE = "ola.gif";
    private static final int    OUR_DEFAULT_GRID  = 5;


    class ButtonPanel extends JPanel
    {
        private JButton myButtons[];

        /**
         * Create a button panel consisting of nButtons per side
         * of a square (total # buttons is nButtons*nButtons) using
         * the designated imageSource (if not null).
         * @param nButtons is the number of buttons PER SIDE
         * @param imageSource is the source of the image for this panel
         */

        ButtonPanel(int nButtons, String imageSource)
        {
            // construct superclass and make button array
            super(new GridLayout(nButtons,nButtons));
            myButtons = new JButton[nButtons*nButtons];

            // load the image if one is specified
            Image image = null;
            if (imageSource != null) {
                image = ImageFactory.getImage(this, imageSource);
            }

            // make listeners for buttons in panel
            // first one to show text in this GUI
            ActionListener textDisplayer = new ActionListener(){
                    public void actionPerformed(ActionEvent e)
                    {
                        showText(e.getActionCommand());
                    }
                };

            // make listener to do the move chosen by user
            ActionListener moveMaker = new ActionListener(){
                    public void actionPerformed(ActionEvent e)
                    {
                        int val = Integer.parseInt(e.getActionCommand());
                        myControl.makeMove(new PuzzleMove(val));
                    }
                };

            makeButtons(image,textDisplayer,moveMaker);
```

```java
        }

        /**
         * Makes the buttons for this GUI/view. The number
         * of buttons is determined my the size of the array
         * myButtons. This helper function takes some of
         * the busy work out of the ButtonPanel constructor.
         */
        private void makeButtons(Image image,
                                 ActionListener textDisplayer,
                                 ActionListener moveMaker)
        {
            for(int k=0; k < myButtons.length; k++) {

                String label = ""+k;
                String iLabel = label;
                if (k == myButtons.length-1) {
                    iLabel = PuzzleConsts.BLANK;
                }
                Icon icon;
                if (image == null) {
                    icon = new PlainPuzzleIcon(iLabel,
                                               myControl.getModelSize());
                }
                else {
                    icon =
                        new ImagePuzzleIcon(image,
                                            iLabel,
                                            myControl.getModelSize());
                }
                // create button with icon
                myButtons[k] = new JButton(icon);
                myButtons[k].setActionCommand(label);
                myButtons[k].addActionListener(textDisplayer);
                myButtons[k].addActionListener(moveMaker);

                add(myButtons[k]);
            }
        }

        /**
         * Set all the buttons by re-displaying them all
         * in the right order. First we remove all the components
         * in this panel (that's the buttons). Then we add the
         * buttons to this panel in the right order based on what
         * the ordering of the buttons in the model is.
         * Finally, we revalidate so the buttons are shown (GUI
         * will redraw as a result of the revalidate).
         */
        public void setGrid(int list[])
        {
            removeAll();
            for(int k=0; k < list.length; k++) {
                add(myButtons[list[k]]);
            }
            revalidate();
        }

    }

}
```



```java
public class PuzzleConsts
{
    public final static String BLANK = " ";
    public final static int IMAGE_SIZE = 240;
    public final static int OFFSET = 1;
}
```

```java
import java.util.ArrayList;

public class PuzzleController
{
    private PuzzleModel myModel;
    private ArrayList   myViews;
    private ShowBoardCommand myShowCommand;
    private ViewCommand      myUndoCommand;

    /**
     * Create Controller not bound to any model --- will
     * cause null pointer exceptions if setModel(m) is not
     * called before any Controller methods are called.
     * @see #PuzzleController
     */

    public PuzzleController()
    {
        this(null);
    }

    /**
     * Create a controller for the designated model. Multiple
     * views can be added to a controller, but a control has
     * a single model.
     * @param model is the model for this controller
     */
    public PuzzleController(PuzzleModel model)
    {
        myModel = model;
        myViews = new ArrayList();
        myShowCommand = new ShowBoardCommand();
        myUndoCommand = new UndoCommand();
    }

    /**
     * Set the model for this controller
     * @param model is the model bound to this controller
     */

    public void setModel(PuzzleModel m)
    {
        myModel = m;
    }

    /**
     * Add a view to be updated by this controller when
     * the model changes.
     * @param view is the view added to this controller
     */

    public void addView(PuzzleView view)
    {
        myViews.add(view);
    }

    /**
     * Returns the size of the model bound to this controller
     * @return the size of the model used by this controller
     */
    public int getModelSize()
    {
        return myModel.getSize();
    }

    /**
     * Make a move (presumably results in changes to views)
     * @param move is the move made
     */
    public void makeMove(PuzzleMove move)
    {
        if (myModel.makeMove(move)){
            updateAllViews(myUndoCommand, Boolean.TRUE);
```



```java
        }
    }

    /**
     * Undo the last undoable move (just made or uncovered
     * as a result of an undo).
     */

    public void undoMove()
    {
        if (!myModel.undo()){
            updateAllViews(myUndoCommand, Boolean.FALSE);
        }
    }

    /**
     * Update all views with the given command using
     * parameter o to command.execute(...)
     * @param command is the command executed to updated views
     * @param o will be passed to the command's execute function
     */
    private void updateAllViews(ViewCommand command, Object o)
    {
        for(int k=0; k < myViews.size(); k++){
            PuzzleView view = (PuzzleView) myViews.get(k);
            command.execute(view,o);
        }
    }

    /**
     * Update view with new "board", called by model
     * @param list is the board, # elements in list is
     * getModelSize()*getModelSize().
     */

    public void showBoard(int[] list)
    {
        updateAllViews(myShowCommand,list);
    }


    private class ShowBoardCommand extends ViewCommand
    {
        protected void hook(PuzzleView view, Object o)
        {
            int[] board = (int[]) o;
            view.showGrid(board);
        }
    }

    private class UndoCommand extends ViewCommand
    {
        protected void hook(PuzzleView view, Object o)
        {
            boolean value = ((Boolean) o).booleanValue();
            view.setEnabledUndo(value);
        }
    }

}
```

```java
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.Image;
import java.awt.event.*;
import javax.swing.*;

/**
 * This is a view for a sliding n-puzzle with one image
 * serving as the soruce for many different pieces each
 * of which can slide. Clients can also specify a null
 * image which results in simple, numbered, and colored squares
 * being used for each piece of the n-puzzle.
 * <P>
 * This is the view, it talks to a controller which mediates between
 * a view (or several) and the view(s) model.
 *
 * @author Owen Astrachan
 */
public class PuzzleGui extends JFrame implements PuzzleView
{
    private PuzzleController myControl;
    private JTextField  myText;
    private ButtonPanel myButtonPanel;
    private JMenuItem   myUndo;

    private static final int OUR_WIDTH  = 500;
    private static final int OUR_HEIGHT = 500;

    public PuzzleGui(PuzzleController control)
    {
        this(control,null);
    }

    /**
     * Create a gui with a model and an image (image optional)
     * @param imageSource is null or the url/file for an image
     * @param model is the model for this view
     */
    public PuzzleGui(PuzzleController control, String imageSource)
    {
        setTitle("OOGA Puzzle");
        JPanel panel = new JPanel(new BorderLayout());

        myControl = control;
        myControl.addView(this);

        myText = new JTextField(20);
        myButtonPanel = new ButtonPanel(myControl.getModelSize(),
                                        imageSource);

        panel.add(myButtonPanel, BorderLayout.CENTER);
        panel.add(myText,        BorderLayout.SOUTH);

        setContentPane(panel);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        makeMenu();
        constrainResize();

        pack();
        setSize(OUR_WIDTH, OUR_HEIGHT);
        setVisible(true);
    }

    /**
     * Required by interface, displayes text message.
     * @param text is to be displayed by view
     */

    public void showText(String text)
    {
        myText.setText(text);
    }
```



```java
    /**
     * Required by interface, enables/disables undo invoker
     * @param value sets the status of the undo (button, menu-item, etc.)
     */

    public void setEnabledUndo(boolean value)
    {
        myUndo.setEnabled(value);
    }

    /**
     * Required by interface, show the grid that's the model
     * @param list represents the model
     */
    public void showGrid(int[] list)
    {
        myButtonPanel.setGrid(list);
    }

    /**
     * Make resizing result in a square view, so "snap"
     * to squareness before resizing takes effect.
     */
    private void constrainResize()
    {
        addComponentListener(new ComponentAdapter(){
                public void componentResized(ComponentEvent e)
                {
                    int w = getWidth();
                    int h = getHeight();
                    w = Math.max(w,h);
                    setSize(w,w);
                }
            });

    }

    private void makeMenu()
    {
        JMenu menu = new JMenu("Puzzle");
        myUndo = new JMenuItem(new AbstractAction("Undo")
            {
                public void actionPerformed(ActionEvent e) {
                    myControl.undoMove();
                }
            });
        setEnabledUndo(false);
        menu.add(myUndo);

        menu.add(new AbstractAction("Quit")
            {
                public void actionPerformed(ActionEvent e) {
                    System.exit(0);
                }
            });
        JMenuBar menubar = new JMenuBar();
        menubar.add(menu);
        setJMenuBar(menubar);
    }

    public static void main(String args[])
    {
        PuzzleController control = new PuzzleController();
        PuzzleModel model = new PuzzleModel(control,10);
        PuzzleGui pg = new PuzzleGui(control);
    }


    class ButtonPanel extends JPanel
    {
        private JButton myButtons[];

        /**
```

```java
         * Create a button panel consisting of nButtons per side
         * of a square (total # buttons is nButtons*nButtons) using
         * the designated imageSource (if not null).
         * @param nButtons is the number of buttons PER SIDE
         * @param imageSource is the source of the image for this panel
         */

        ButtonPanel(int nButtons, String imageSource)
        {
            // construct superclass and make button array
            super(new GridLayout(nButtons,nButtons));
            myButtons = new JButton[nButtons*nButtons];

            // load the image if one is specified
            Image image = null;
            if (imageSource != null) {
                image = ImageFactory.getImage(this, imageSource);
            }

            // make listeners for buttons in panel
            // first one to show text in this GUI
            ActionListener textDisplayer = new ActionListener(){
                    public void actionPerformed(ActionEvent e)
                    {
                        showText(e.getActionCommand());
                    }
                };

            // make listener to do the move chosen by user
            ActionListener moveMaker = new ActionListener(){
                    public void actionPerformed(ActionEvent e)
                    {
                        int val = Integer.parseInt(e.getActionCommand());
                        myControl.makeMove(new PuzzleMove(val));
                    }
                };

            makeButtons(image,textDisplayer,moveMaker);
        }

        /**
         * Makes the buttons for this GUI/view. The number
         * of buttons is determined my the size of the array
         * myButtons. This helper function takes some of
         * the busy work out of the ButtonPanel constructor.
         */
        private void makeButtons(Image image,
                                 ActionListener textDisplayer,
                                 ActionListener moveMaker)
        {
            for(int k=0; k < myButtons.length; k++) {

                String label = ""+k;
                String iLabel = label;
                if (k == myButtons.length-1) {
                    iLabel = PuzzleConsts.BLANK;
                }
                Icon icon;
                if (image == null) {
                    icon = new PlainPuzzleIcon(iLabel,
                                               myControl.getModelSize());
                }
                else {
                    icon =
                        new ImagePuzzleIcon(image,
                                            iLabel,
                                            myControl.getModelSize());
                }
                // create button with icon
                myButtons[k] = new JButton(icon);
                myButtons[k].setActionCommand(label);
                myButtons[k].addActionListener(textDisplayer);
                myButtons[k].addActionListener(moveMaker);
```



```java

                add(myButtons[k]);
            }
        }

        /**
         * Set all the buttons by re-displaying them all
         * in the right order. First we remove all the components
         * in this panel (that's the buttons). Then we add the
         * buttons to this panel in the right order based on what
         * the ordering of the buttons in the model is.
         * Finally, we revalidate so the buttons are shown (GUI
         * will redraw as a result of the revalidate).
         */
        public void setGrid(int list[])
        {
            removeAll();
            for(int k=0; k < list.length; k++) {
                add(myButtons[list[k]]);
            }
            revalidate();
        }
    }
}
```

```java
import java.util.LinkedList;

/**
 * This is the model for a sliding puzzle game. The model
 * keeps track of the location of n^2 squares, for n specified
 * when the model is constructed.  A grid if n^2-1 visible squares
 * is represented by the model, one square is "the blank square".
 * <P>
 * Clients click on a square adjacent to the blank square and the
 * clicked on square swaps with the blank. This simulates moving
 * the clicked square into the blank spot --- and the previous
 * location of the clicked square is now blank.
 *
 * All communication between a model and its views is handled
 * by the model's Controller. The controller can have multiple
 * views, but the model has a single controller in the current
 * architecture.
 * <P>
 * <b>Notes on Implementation</b>
 * <P>
 * The model is conceptually a grid of n x n elements. However,
 * the current implementation models this as a single array
 * of n x n elements --- and the elements are int values.
 * <P>
 * The idea is that originally, array[k] == k where array is
 * the state that represents the model. When two values are swapped,
 * they change locations in the array. Thus, when a client wants
 * to move square number 7, this square must be searched for. That is
 * the value 7 is searched for in the array that represents the model.
 * This happens for the blank square too --- both the move and the
 * blank are searched for in the entire grid/array. Then the code
 * currently implemented determines if these elements (move and blank)
 * are neighbors. If so, they're swapped and the controller updates
 * the views.
 * <P>
 * <em>This means that making a move is an O(n^2)
 * operation for an n x n grid.</em>
 *
 */

public class PuzzleModel
{
    /**
     * Creates a model with the specified controller and size. The
     * size parameter is one dimension of a 2D grid represented by
     * this model.
     * @param control mediates communication between model and its views
     * @param size is the size of one square of the model
     */

    public PuzzleModel(PuzzleController control, int size)
    {
        myControl = control;
        myList = new LinkedList();
        mySize = size;
        myNumbers = new int[size*size];
        for(int k=0; k < size*size; k++) {
            myNumbers[k] = k;
        }
        myBlank = size*size - 1;
        myControl.setModel(this);
    }

    /**
     * Return the value associated with a Blank
     */
    private  int getBlankValue()
    {
        return myBlank;
    }

    /**
     * return location/index of number in myNumbers
```



```java
     * @param number between 0 and mySize^2-1 (inclusive)
     * @return the index such that myNumbers[index] == number
     */

    private int getIndex(int number)
    {
        for(int k=0; k < myNumbers.length; k++) {
            if (myNumbers[k] == number) {
                return k;
            }
        }
        return -1;
    }

    /**
     * @param a index in [0..mySize^2-1)
     * @param b index in [0..mySize^2-1)
     * @return true if a and b are neighbors in grid, else returns false
     */
    private boolean isNeighbors(int a, int b)
    {
        int aRow = a / mySize;
        int bRow = b / mySize;
        int aCol = a % mySize;
        int bCol = b % mySize;

        if (aRow == bRow && Math.abs(aCol - bCol) == 1) {
            return true;
        }
        if (aCol == bCol && Math.abs(aRow - bRow) == 1) {
            return true;
        }
        return false;
    }

    /**
     * Attempts to make a move and returns true if the move
     * was successfully made, else returns false.
     * @param move is the move being attempted in this model
     * @return true if move succeeded, else returns false
     */

    public boolean makeMove(PuzzleMove move)
    {
        return doMove(move,true);
    }

    /**
     * Make the move by swapping the tile/square associated
     * with the move with the blank. Returns true iff successful.
     * @param doStore determines if the move will be stored and thus
     * undoable
     * @param move is the move being attempted
     * @return true iff making move succeeds
     */
    private boolean doMove(PuzzleMove move, boolean doStore)
    {
        int index = getIndex(move.getValue());
        int blankIndex = getIndex(getBlankValue());

        if (isNeighbors(index,blankIndex)) {
            int temp = myNumbers[blankIndex];
            myNumbers[blankIndex] = myNumbers[index];
            myNumbers[index] = temp;

            myControl.showBoard(myNumbers);
            if (doStore){
                myList.add(move);
            }
            return true;
        }
        return false;
    }
```

```java
    /**
     * Undoes the last move (if there is one) and returns true iff
     * another undo is possible
     * @return true if another undo is possible, else returns false
     */
    public boolean undo()
    {
        if (myList.size() != 0) {
            doMove((PuzzleMove) myList.removeLast(),false);
        }
        return myList.size() > 0;
    }

    /**
     * Returns the dimension of one side of the grid represented
     * by this model.
     * @return the dimension of one side of the puzzle model
     */
    public int getSize()
    {
        return mySize;
    }

    private int mySize;
    private int myBlank;
    private int myNumbers[];
    private LinkedList myList;
    private PuzzleController myControl;
}
```



```java
/**
 * A move in the puzzle game is simply a square.
 * It must be moved to the "blank" location, so
 * the square's location is simply stored as part
 * of the move
 */

public class PuzzleMove
{
    private int mySquare;

    /**
     * Construct a move from a square (presumably in
     * the range of legal values for a puzzle game.)
     * @param square is the move
     */

    public PuzzleMove(int square)
    {
        mySquare = square;
    }

    /**
     * Returns the value of this move.
     * @return the move value
     */
    public int getValue()
    {
        return mySquare;
    }

}
```

```java
/**
 * The basic interface for a generic view
 * of a PuzzleModel. Adding other methods here
 * will force views to implement the methods
 *
 * @author Owen Astrachan
 */
public interface PuzzleView
{
    /**
     * Shows text in some view-specific way, e.g.,
     * in a textfield for information.
     * @param text is the text to display
     */

    public void showText(String text);

    /**
     * Enable/disable the undo move command.
     * The controller should enable undo when undos
     * are possible and disable undo when no moves have
     * been made that are undoable
     * @param value determines whether undo is enabled(true) or
     * disabled (false)
     */

    public void setEnabledUndo(boolean value);

    /**
     * Show a grid/board, where list is the values
     * between 0 and size*size-1 representing locations
     * of square-elements. If list[3] = 7 then square 7 is
     * in location 3 (zero/first row, column 3 if there are more than 3
     * elements per side.
     * @param list is the elements to be shown
     */
    public void showGrid(int[] list);
}
```



```java
/**
 * A command executed by a view. Subclasses implement
 * the hook method, clients call execute which will
 * call the hook method. Client code can assume
 * every command has an execute method, but the execute
 * method calls the hook method which does the real work.
 */

public abstract class ViewCommand
{
    public void execute(PuzzleView view)
    {
        hook(view,null);
    }
    public void execute(PuzzleView view, Object o)
    {
        hook(view,o);
    }

    protected abstract void hook(PuzzleView view, Object o);
}
```