Model-View-Controller Design Pattern by Example

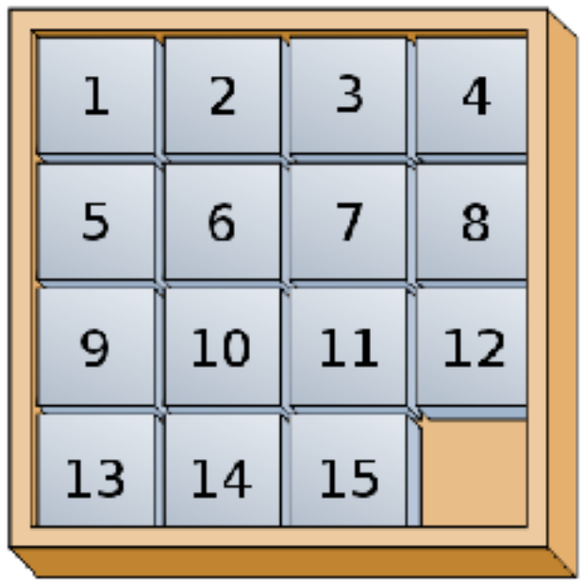
Slide Puzzle

Samir Genaim

see: examples.swing.mvc.slidepuzzle

Slide Puzzle Game

- Implement a Slide Puzzle game
- It should have a console interface where the user can make moves by typing commands, etc.
- It should have a swing interface
- In both interfaces, the user should have the possibility to log all moves/board/etc into a file



The Model - Board

```
public class RectBoard implements Board {
    int[][] board;
    int rows;
    int cols;
    ...
    public Board(int rows, int cols) {
        this.rows = rows;
        this.cols = cols;
        board = new int[rows][cols];
    }

    public boolean setPostion(int row, int col, int v) { ... }
    public int getPostion(int row, int col) { ... }
    public int getWidth() { ... }
    public int getHeight() { ... }
    public String toString() { ... }
}
```

- An interface that represents a game board and a possible implementation
- Allows setting cells to integer values, asking what's the value of a cell, etc.
- `toString` returns a string that corresponds to the board, we let 0 represent an empty cell

```
public interface Board {
    public int getPosition(int row, int col);
    public boolean setPosition(int row, int col, int num);
    public int getWidth();
    public int getHeight();
}
```

The Model - SlidePuzzle

```
public class SlidePuzzle {

    private Board board;
    private boolean finished;
    private int rows;
    private int cols;

    public SlidePuzzle() { this(4,4); }
    public SlidePuzzle(int rows, int cols) {
        reset(rows,cols);
    }

    // move the number at cell (row,col) to the empty one
    public void move(int row, int col) { ... }

    // ask if game is finished
    public boolean isFinished() { return finished; }

    // (re)start the  game
    public void reset(int rows, int cols) { ... }

}
```

- A class to represent the Slide Puzzle game
- It allows making a move, reseting the board, etc.
- move throws an exception if the move is invalid

The Console Mode - First Try

```
public class SlidePuzzle {

    private Board board;
    private boolean finished;

    ...
    public void runConsole() {
      ...
      while (!finished) {
        System.out.println( board );
        // read a command
        // execute command
        //  1. if "move x y" then call move
        //  2.if "reset rows cols" then call
        //     reset(rows,cols)
        //   ...
    }
}
```

- You'll probably get fired the moment you submit this code!!
- What if we want to add a new interface? Should we add another method to SlidePuzzle?
- Violates an important OOP principle: **code should be open for extensions but not for modifications**

The Console Mode - Second Try

```
public class ConsoleController {

    private SlidePuzzle game;

    ...
    public void run() {
      ...
      while (!game.isFinished()) {
        System.out.println( game );
        // read a command
        // execute command
        //  1. if "move x y"
        //       call game.move
        //  2.if "reset rows cols"
        //     call game.reset(rows,cols)
        //    ...
    }
}
```

- Separate interaction with the user from the model: looks promising!
- How we'll add the possibility to log the information now?

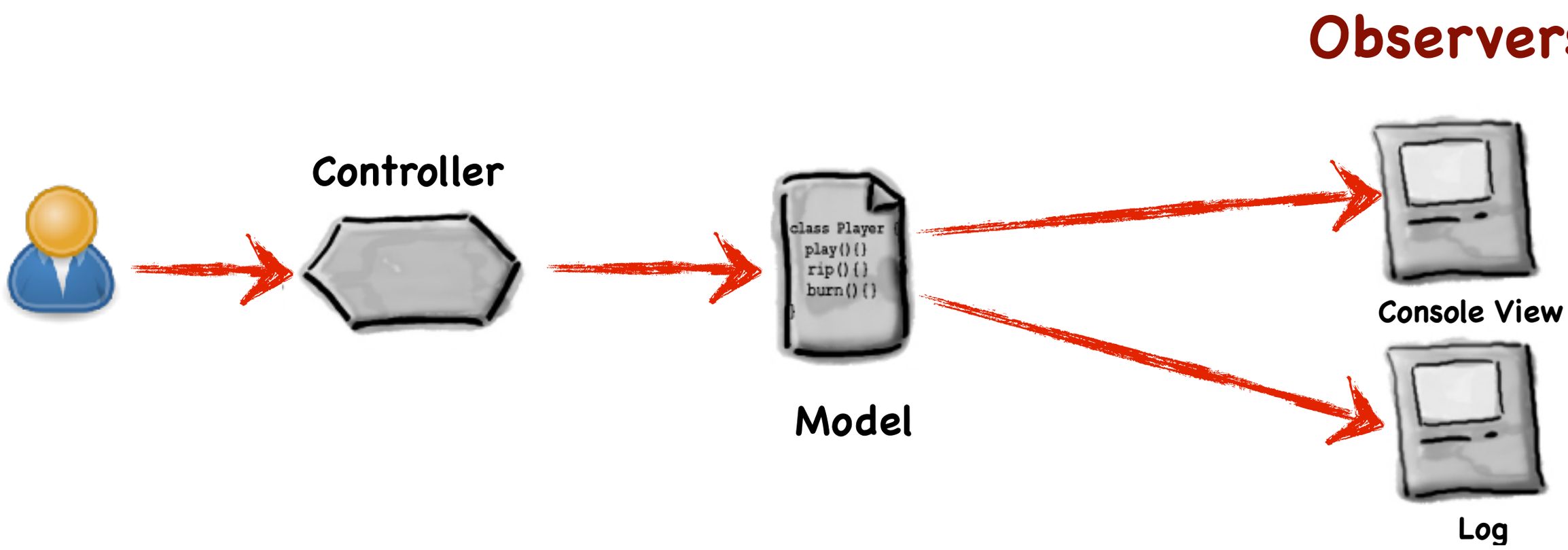
The Console Mode - Second Try

```
public class ConsoleController {

    private SlidePuzzle game;

    ...
    public void run() {
      ...
      while (!game.isFinished()) {
        System.out.println( game );
        // read a command
        // execute command
        //  1. if "move x y"
        //       call game.move
        //  2.if "reset rows cols"
        //     call game.reset(rows,cols)
        //    ...
    }
}
```

- Separate interaction with the user from the model: looks promising!
- How we'll add the possibility to log the information now?

- Since we're not allowed to touch the model, we will have to touch this loop to add some code that writes a log to a file, etc.
- still violates the principle: **code should be open for extensions but not for modifications**
- Good design should allow adding the log without modifying any existing code!

The MVC Approach



- Users tell the Controller what they want to do
- The Controller knows how to pass these requests to the model, e.g., via method calls
- The model notify the views that something has changed
- The views decide how to show the current state
- Implement using the Observable design pattern

What/How to Observe

```
public interface SlidePuzzleObserver {
    public void onGameStart(Board board);
    public void onMove(Board board, int sRow, int sCol, int tRow, int tCol);
    public void onGameOver(Board board);
    public void onError(String msg);
}
```

```
public interface Observable<T> {
    public void addObserver(T o);
    public void removeObserver(T o);
}
```

- SlidePuzzleObserver describes which notifications the model will send to observers (e.g., views) when the state changes
- Anyone who wants to observe changes in the model must implement SlidePuzzleObserver
- The model must allow registering for receiving notifications, i.e., it must implement Observable<SlidePuzzleObserver>

What/How to Observe

```
public interface SlidePuzzleObserver {
    public void notify(GameEvent e);
}
```

```
public class GameEvent {

    public enum EventType {
        Start, Change, Info, Error, Stop, ...
    }

    private EventType type;
    private Board board;
    ...
    public GameEvent(EventType type, Board board, ...) {
    }

    public EventType getType() {
        return type;
    }

    public Board getdBoard() {
        return board;
    }
        ...
}
```

- Another very common way to define the Observer interface is by using an Event class.
- This is useful because in order to add move events we don't have to change the interface ..
- Events should provide information about their, about the state, etc.

The Model - (un)register

```
public class SlidePuzzle implements Observable<SlidePuzzleObserver> {

    private Board board;
    private boolean finished;
    ...

    private List<SlidePuzzleObserver> obs;

    ...
    @Override
    public void addObserver(SlidePuzzleObserver o) {
        obs.add(o);
    }

    @Override
    public void removeObserver(SlidePuzzleObserver o) {
        obs.remove(o);
    }
}
```

List of observers

Others should use this method to register for receiving notifications

They can also stop receiving notifications

The Model - send notifications

```
public class SlidePuzzle implements Observable<SlidePuzzleObserver> {
    ...
    private Collection<SlidePuzzleObserver> obs;
    ...
    public void reset(int rows, int cols) {
      ...
      // notify observers that the game has started
      notifyGameStart();
    }

    public void move(int row, int col) {
        ...
        // notify observers that something went wrong
        if ( there is an error ) {
            notifyError("...");
        }
        ...
        // notify observers that a move has been executed
        notifyMove(...);
        ...
        // check if game is over
         if ( checkIfFinished() ) {
            finished = true;
            notifyGameOver();
         }
    }
    ...
}
```

When "things of interest" happen, traverse the list of observers and send the corresponding notifications by calling corresponding methods

The Model - send notifications

```
private void notifyStartGame() {
    for (SlidePuzzleObserver o : obs) {
        o.onGameStart(roBoard);
    }
}

private void notifyGameOver() {
    for (SlidePuzzleObserver o : obs) {
        o.onGameOver(roBoard);
    }
}
```

traverse the list of observers and send the corresponding notifications by calling the relevant methods

```
private void notifyMove(int row, int col, int i, int j) {
    for (SlidePuzzleObserver o : obs) {
        o.onMove(roBoard, row, col, i, j);
    }
}

private void notifyError(String msg) {
    for (SlidePuzzleObserver o : obs) {
        o.onError(msg);
    }
    throw new GameError(msg);
}
```

The Model - send notifications

```
private void notifyStartGame() {
    for (SlidePuzzleObserver o : obs) {
        o.onGameStart(roBoard);
    }
}
```

```
private void notifyGameOver() {
    for (SlidePuzzleObserver o : obs) {
        o.onGameOver(roBoard);
    }
}
```

traverse the list of observers and send the corresponding notifications by calling the relevant methods

To avoid breaking the encapsulation principle, we can use a readonly version of the board!

```
private void notifyMove(int row, int col, int i, int j) {
    for (SlidePuzzleObserver o : obs) {
        o.onMove(roBoard, row, col, i, j);
    }
}
```

```
private void notifyError(String msg) {
    for (SlidePuzzleObserver o : obs) {
        o.onError(msg);
    }
    throw new GameError(msg);
}
```

ReadOnly Board

```
public class ReadOnlyBoard implements Board {

    private Board board;

    public ReadOnlyBoard(Board board) {
        this.board = board;
    }

    public boolean setPosition(int row, int col, int num) {
        throw new UnsupportedOperationException("...");
    }

    public int getPosition(int row, int col) {
        return board.getPosition(row, col);
    }

    public int getWidth() {
        return board.getWidth();
    }

    public int getHeight() {
        return board.getHeight();
    }

    public String toString() {
        return board.toString();
    }
}
```

A board that encapsulates another board

Modification requests cause errors!

Read requested are delegated to the encapsulated board

ReadOnly Board

```
public class ReadOnlyBoard implements Board {

    private Board board;

    public ReadOnlyBoard(Board board) {
        this.board = board;
    }

    public boolean setPosition(int row, int col, int num) {
        throw new UnsupportedOperationException("...");
    }

    public int getPosition(int row, int col) {
        return board.getPosition(r
    }

    public int getWidth() {
        return board.getWidth();
    }

    public int getHeight() {
        return board.getHeight();
    }

    public String toString() {
        return board.toString();
    }
}
```

A board that encapsulates another board

Modification requests

```
public class SlidePuzzle {

    private Board board;
    private Board roBoard;
    ...
    public void reset() {
        ...
        board = new RectBoard(rows, cols);
        roBoard = new ReadOnlyBoard(board);
        ...
    }
}
```

The Controller - Revisited

```
abstract class Controller {
    protected SlidePuzzle game;

    public Controller(SlidePuzzle game) {
        this.game = game;
    }

    public void move(int row, int col) {
        game.move(row, col);
    }

    public void reset(int rows, int cols) {
        game.reset(rows, cols);
    }

    public void requestExit() {
        ...
    }

    public abstract  void run();
}
```

- it does not print anything that is directly related to the model
- It might print some messages that are related to the interaction with the user, e.g., "I don't understand what you want!".

```
class ConsoleController extends Controller {

    public ConsoleController(SlidePuzzle game) {
        super(game);
    }

    @Override
    public run() {
      ...
      while (!game.isFinished()) {
        // read a command
        // execute command
        //  1. if "move x y", call move(...)
        //  2.if "reset rows cols",call reset(...)
    }
}
```

The Console View

```
public class ConsoleView implements SlidePuzzleObserver {

    public ConsoleView(Observable<SlidePuzzleObserver> game) {
        game.addObserver(this);
    }

    @Override
    public void onGameStart(Board board) {
        System.out.println(board);
    }

    @Override
    public void onMove(Board board, int sRow, int sCol, int tRow, int tCol) {
        System.out.println(board);
    }

    @Override
    public void onError(String msg) {
        System.err.println(msg);
    }

    @Override
    public void onGameOver(Board board) {
        System.out.println("Game over!");
    }

}
```

First register as an observer in the model

The Console View

```
public class ConsoleView implements SlidePuzzleObserver {

    public ConsoleView(Observable<SlidePuzzleObserver> game) {
        game.addObserver(this);
    }

    @Override
    public void onGameStart(Board board) {
        System.out.println(board);
    }

    @Override
    public void onMove(Board board, int sRow, int sCol, int tRow, int tCol) {
        System.out.println(board);
    }

    @Override
    public void onError(String msg) {
        System.err.println(msg);
    }

    @Override
    public void onGameOver(Board board) {
        System.out.println("Game over!");
    }

}
```

First register as an observer in the model

These methods will be called by the model to send the corresponding notifications. The view reacts accordingly

The Log "View"

```
public class LogView implements SlidePuzzleObserver {
    private String fname;
    public LogView(Observable<SlidePuzzleObserver> game, String fname) {
        this.fname = fname;
        game.addObserver(this);
    }
    private void log(String msg) {
        // append to the file fname
    }
    @Override
    public void onGameStart(Board board) {
        log(board.toString());
    }
    @Override
    public void onMove(Board board, int sRow, int sCol, int tRow, int tCol) {
        log(board.toString());
        log("Moved ("+sRow+","+sCol+") to ("+tRow+","+tCol+")");
    }
    @Override
    public void onError(String msg) {}
    @Override
    public void onGameOver(Board board) {
        log("Game over!");
    }
}
```

Register in the model

React when receiving a notification

How to Connect all Components

```
public class Main {

    public static void main(String[] args) {
        ....
        SlidePuzzle game = new SlidePuzzle();
        Controller ctrl = new ConsoleControler(game);

        // view 1
        new ConsoleView(game);

        // view 2
        new LogView(game, "/tmp/log.txt");

        ctrl.run();
    }

}
```

Adding/Changing a view is as simple as creating an object and passing to it the object it should observe

This design respects the principle: **code should be open for extensions but not for modifications**

Window View - Ver I

View the board in a text area. Interaction with the user will still be in the console as before

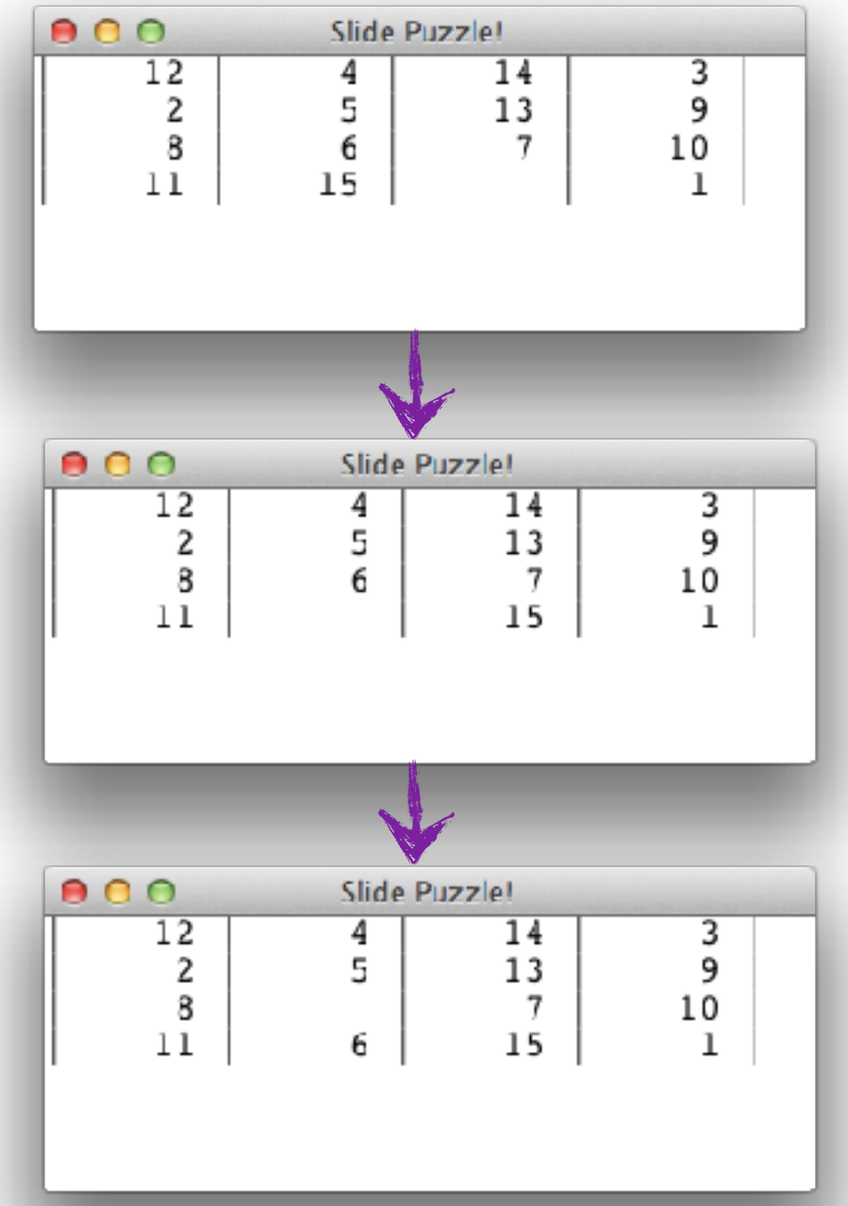
console

Please enter a command: **move 4 2**

Please enter a command: **move 3 2**

Please enter a command:

…



MainWindow1

```
class MainWindow1 extends JFrame implements SlidePuzzleObserver {

    private JTextArea txtArea;

    public MainWindow1(Observable<SlidePuzzleObserver> game) {
        super("Slide Puzzle!");
        initGUI();
        game.addObserver(this);
    }

    private void initGUI() {
        txtArea = new JTextArea(10,10);
        this.setContentPane(txtArea);
        txtArea.setEditable(false);
        txtArea.setFont(new Font("Courier", Font.PLAIN, 16));
          ...
        this.pack();
        this.setVisible(true);

    }
     ...
}
```

In the constructor we create the GUI and then register in the model. The order is sometimes important!!

MainWindow1

```
class MainWindow1 extends JFrame implements SlidePuzzleObserver {

    private JTextArea txtArea;
    ...

    @Override
    public void onGameStart(Board board) {
        txtArea.setText(board.toString());
    }

    @Override
    public void onMove(Board board, int sRow, int sCol, int tRow, int tCol) {
        txtArea.setText(board.toString());
    }

    @Override
    public void onError(String msg) {
    }

    @Override
    public void onGameOver(Board board) {
    }

}
```

When the model notifies, we change the content of the text area to include the new board.

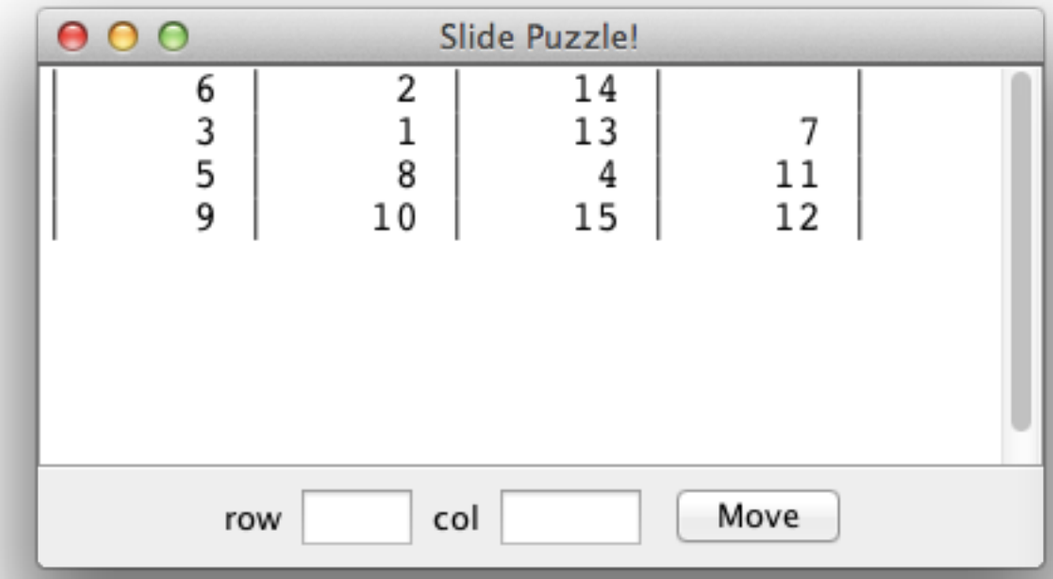
errors and game over notifications are simply ignored

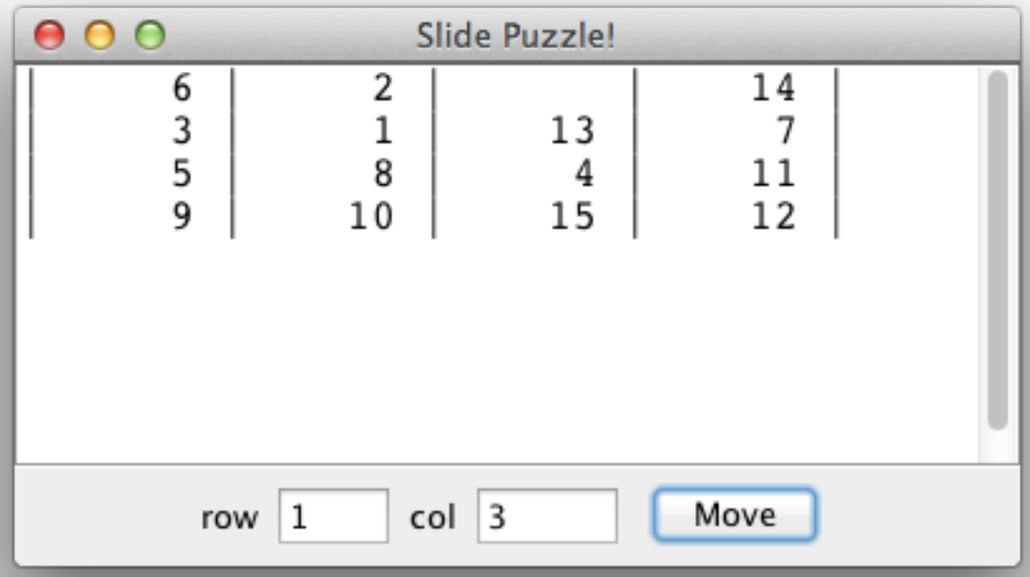
Connect the Pieces

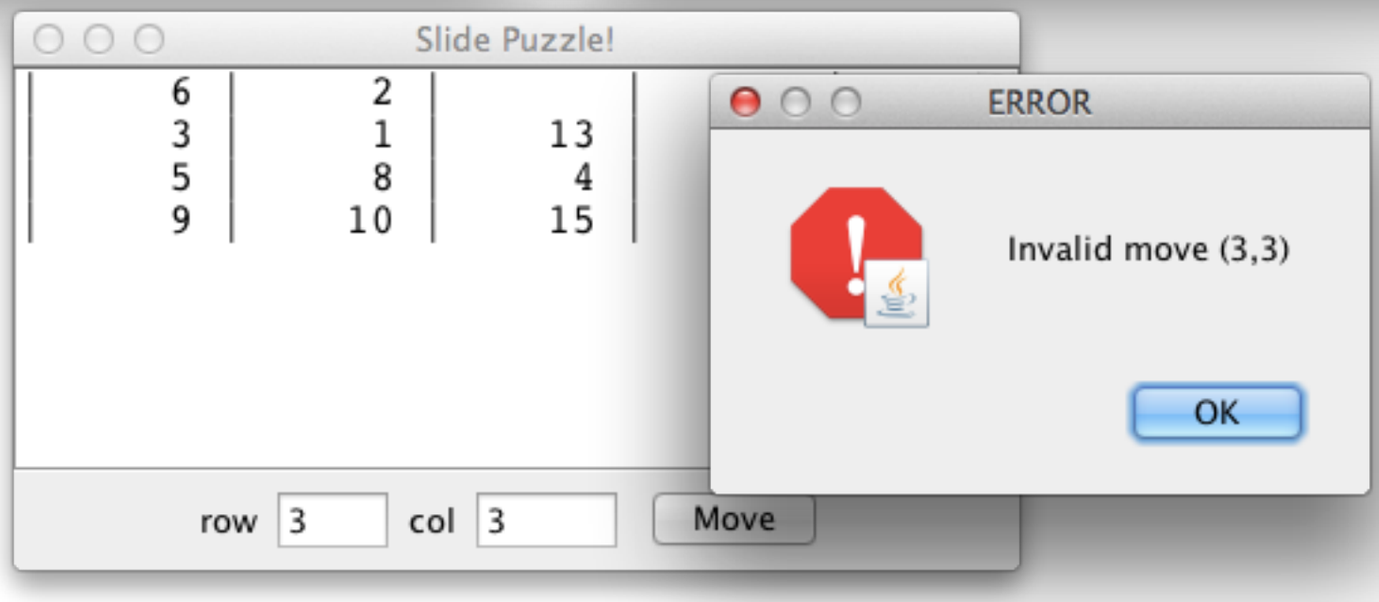
```
public class Main {
   public static void main(String[] args) {
      SlidePuzzle game = new SlidePuzzle();
      Controller ctrl = new ConsoleControler(game);
      //new ConsoleView(game);
      //new LogView(game, "/tmp/log.txt");
      new MainWindow1(game);
      ctrl.run();
   }
}
```

Window View - Ver II

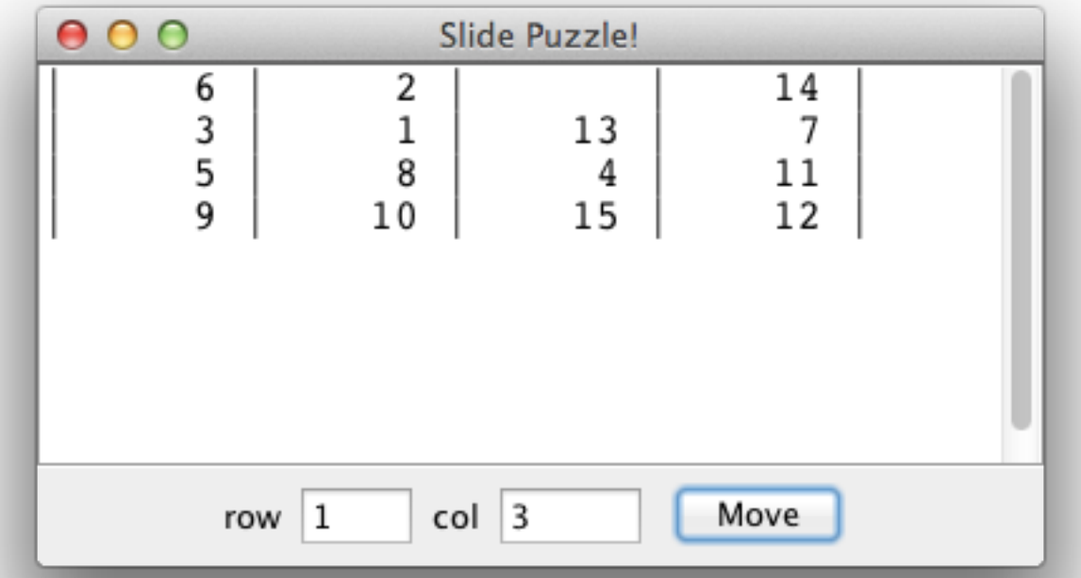
Let us improve the window view so it allows making a move, show error messages, etc. -- not to reset a game for now.



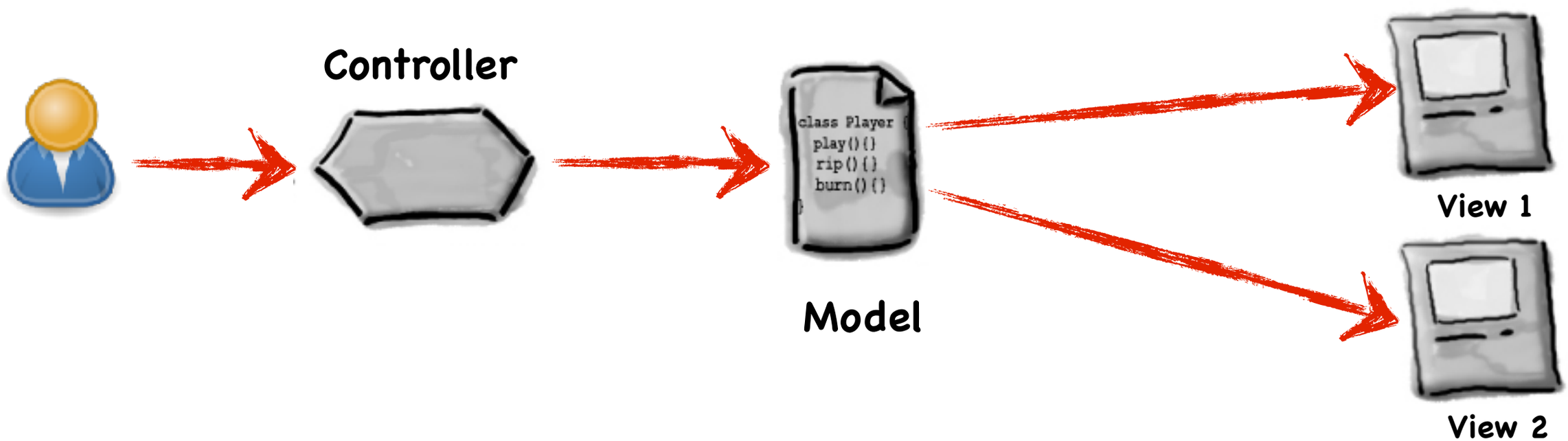




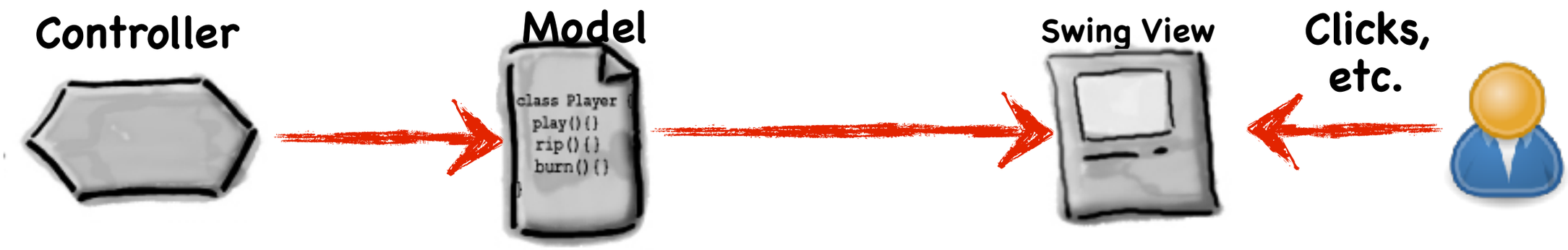
Controller and View: What is What!?



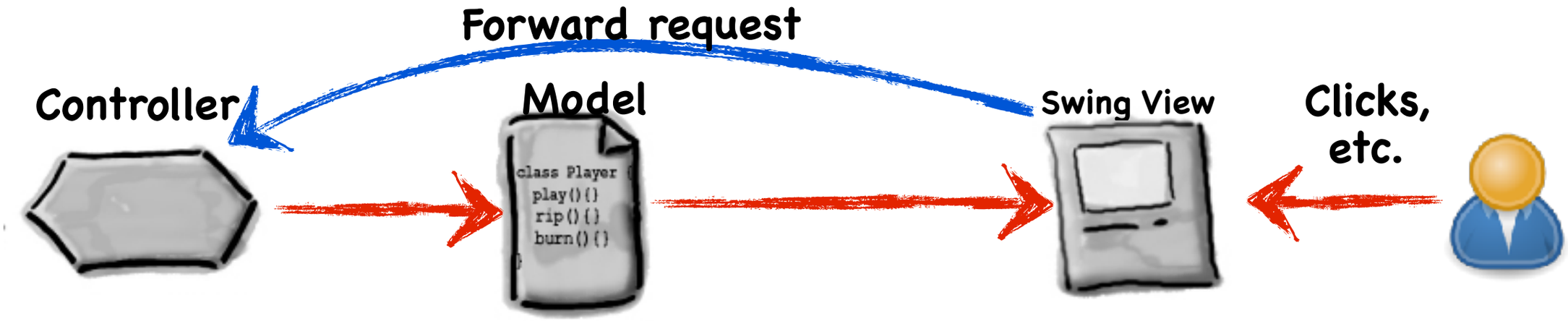
- When building views using Swing, one can be confused when deciding what belongs to the view and what belongs to controller
- We said users interact with the controller, but if we develop a view in swing, a click on the button "Move" is captured in an action listener, which can be seen as part of the view classes
- How we solve this confusion?



Controller and View: What is What!?

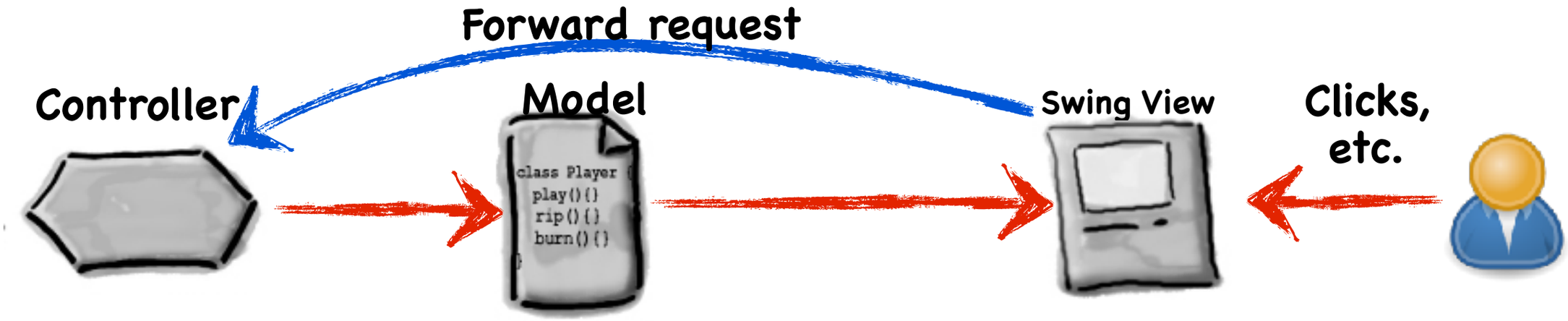


Controller and View: What is What!?



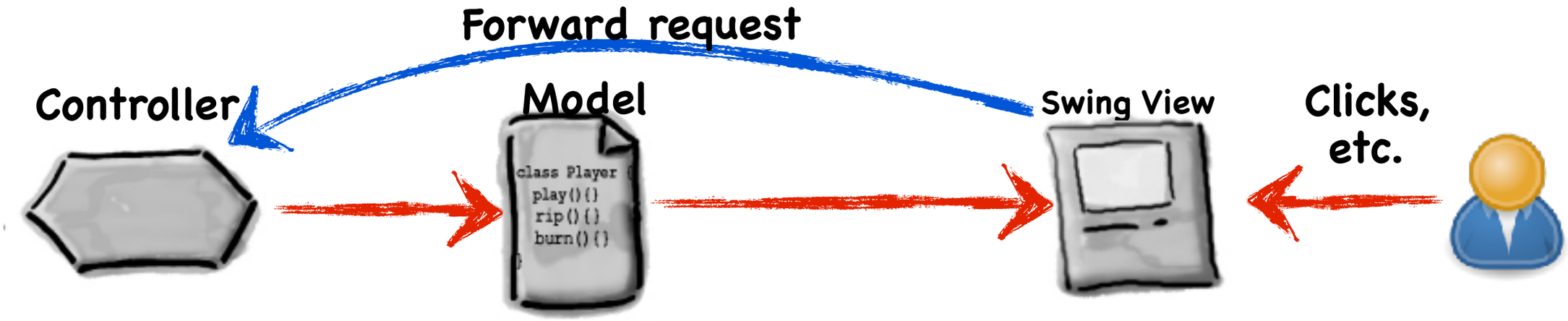
- Imagine a logical division of the "view methods", where action listeners that capture user interactions considered as part of the controller -- they will forward corresponding requests to the controller, and the controller will perform corresponding operations on the model

Controller and View: What is What!?



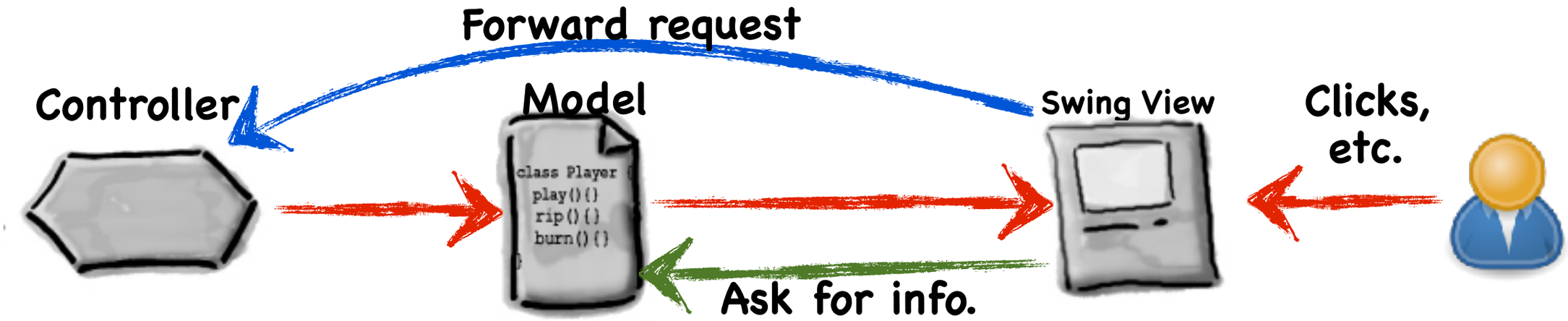
- Imagine a logical division of the "view methods", where action listeners that capture user interactions considered as part of the controller -- they will forward corresponding requests to the controller, and the controller will perform corresponding operations on the model
- Sometimes such action listeners will check for errors before sending requests to the controller, this way the controller does not have to care about opening dialog windows, etc.

Controller and View: What is What!?



- Imagine a logical division of the "view methods", where action listeners that capture user interactions considered as part of the controller -- they will forward corresponding requests to the controller, and the controller will perform corresponding operations on the model
- Sometimes such action listeners will check for errors before sending requests to the controller, this way the controller does not have to care about opening dialog windows, etc.
- So why to use a controller?!? Because it is still a useful abstraction layer for operations that can be performed on the model and for how we communicate with the model ...

Controller and View: What is What!?



- Imagine a logical division of the "view methods", where action listeners that capture user interactions considered as part of the controller -- they will forward corresponding requests to the controller, and the controller will perform corresponding operations on the model
- Sometimes such action listeners will check for errors before sending requests to the controller, this way the controller does not have to care about opening dialog windows, etc.
- So why to use a controller?!? Because it is still a useful abstraction layer for operations that can be performed on the model and for how we communicate with the model ...
- Sometime views are allowed to communicate with the model, but only asking for information not for performing operations

How the Controller Looks Like?

```
public class WindowController extends Controller {
    private SlidePuzzle game;


    public Controller(SlidePuzzle game) {
        super(game);
    }


    public void run() {
    }
}
```

It receives a game and passes it to super class

run does nothing now, interaction with the user will happen in the thread of Swing

MainWindow2

```
public class MainWindow2 extends JFrame implements SlidePuzzleObserver {

    private Observable<SlidePuzzleObserver> game;
    private Controller ctrl;

    public MainWindow2(Controller ctrl, Observable<SlidePuzzleObserver> game) {
        super("Slide Puzzle!");
        this.game = game;
        this.ctrl = ctrl;
        initGUI();
        game.addObserver(this);
    }
```

As before, but now the view gets also the controller object

```
    private void initGUI() {

        JPanel mainPanel = new JPanel( new BorderLayout() );
        this.setContentPane(mainPanel);

        JPanel boardPanel = new BoardPanel(ctrl, game);
        mainPanel.add(boardPanel, BorderLayout.CENTER);

        ...
        this.pack();
        this.setVisible(true);
    }
    ...
}
```

The part that is responsible on the board and the move widgets, etc., is defined in a separated class that extends JPanel. It will be a (sub)view by itself!

What else MainWindow2 Will Do?

```
public class MainWindow2 extends JFrame implements SlidePuzzleObserver {

    private Observable<SlidePuzzleObserver> game;
    private Controller ctrl;

    @Override
    public void onGameStart(Board board) {
    }

    @Override
    public void onMove(Board board, int sRow, int sCol, int tRow, int tCol) {
    }

    @Override
    public void onError(String msg) {
        JOptionPane.showMessageDialog(new JFrame(), msg, "ERROR",
                JOptionPane.ERROR_MESSAGE);
    }

    @Override
    public void onGameOver(Board board) {
    }
}
```

MainWindow only reacts on error notifications

BoardPanel

```
public class BoardPanel extends JPanel implements SlidePuzzleObserver {

    private JTextArea txtArea;
    private JTextField rowTxt;
    private JTextField colTxt;
    private JButton moveButton;
    private Controller ctrl;

    public BoardPanel(Controller ctrl, Observable<SlidePuzzleObserver> game) {
        this.ctrl = ctrl;
        initGUI();
        game.addObserver(this);
    }

    private void initGUI() { ... }

    @Override
    public void onGameStart(Board board) { ... }

    @Override
    public void onMove( ... ) { ... }

    @Override
    public void onError(String msg) { ... }

    @Override
    public void onGameOver(Board board) { .. }
}
```

All widgets are fields so they can be used on notifications

A view by itself, it receives the game and the controller

Initialise the GUI

react on notifications

BoardPanel - initGUI

```
public class BoardPanel extends JPanel implements SlidePuzzleObserver {
    ...
    private void initGUI() {
        this.setLayout( new BorderLayout() );
        txtArea = new JTextArea(10,20);
        txtArea.setEditable(false);
        txtArea.setFont(new Font("Courier", Font.PLAIN, 16));
        JScrollPane area = new JScrollPane(txtArea,...);
        rowTxt = new JTextField(3);
        colTxt = new JTextField(3);
        moveButton = new JButton("Move");
        moveButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    int row = Integer.parseInt(rowTxt.getText());
                    int col = Integer.parseInt(colTxt.getText());
                    ctrl.move(row, col);
                } catch ( NumberFormatException _e ) {}
            }
        });
        JPanel playPanel = new JPanel();
        playPanel.add( new JLabel( "row") );
        playPanel.add( rowTxt );
        playPanel.add( new JLabel( "col") );
        playPanel.add( colTxt );
        playPanel.add( moveButton);
        this.add(area, BorderLayout.CENTER);
        this.add(playPanel, BorderLayout.PAGE_END);
        ....
    }
}
```

When "move" is clicked, we send a request to the controller if the user provided integers in the text area, otherwise we do nothing (we could open an error dialog)

BoardPanel - handle notif.

```
public class BoardPanel extends JPanel implements SlidePuzzleObserver {
    ...
    @Override
    public void onGameStart(Board board) {
        txtArea.setText(board.toString());
        moveButton.setEnabled(true);
        rowTxt.setEnabled(true);
        colTxt.setEnabled(true);
    }

    @Override
    public void onMove(Board board, int sRow, int sCol, int ttRow, int tCol) {
        txtArea.setText(board.toString());
    }

    @Override
    public void onError(String msg) {
    }

    @Override
    public void onGameOver(Board board) {
        moveButton.setEnabled(false);
        rowTxt.setEnabled(false);
        colTxt.setEnabled(false);
    }
}
```

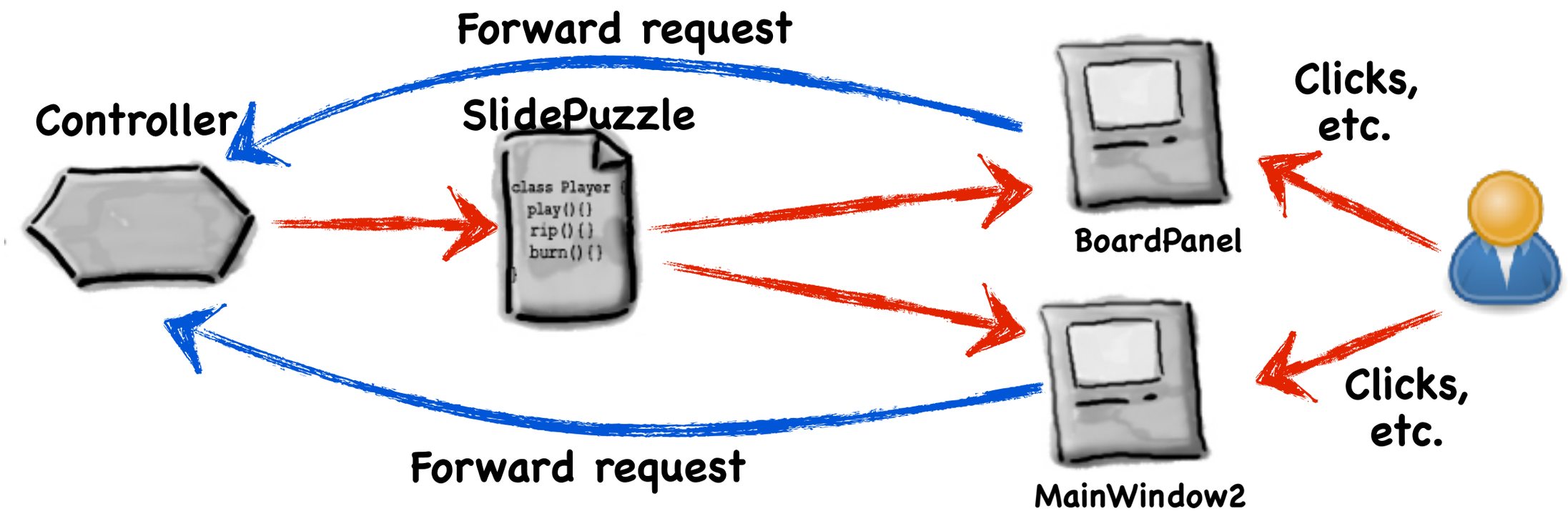
update the board, and enable the move widgets

update the board

do nothing for errors, MainWindow is already taking care of this

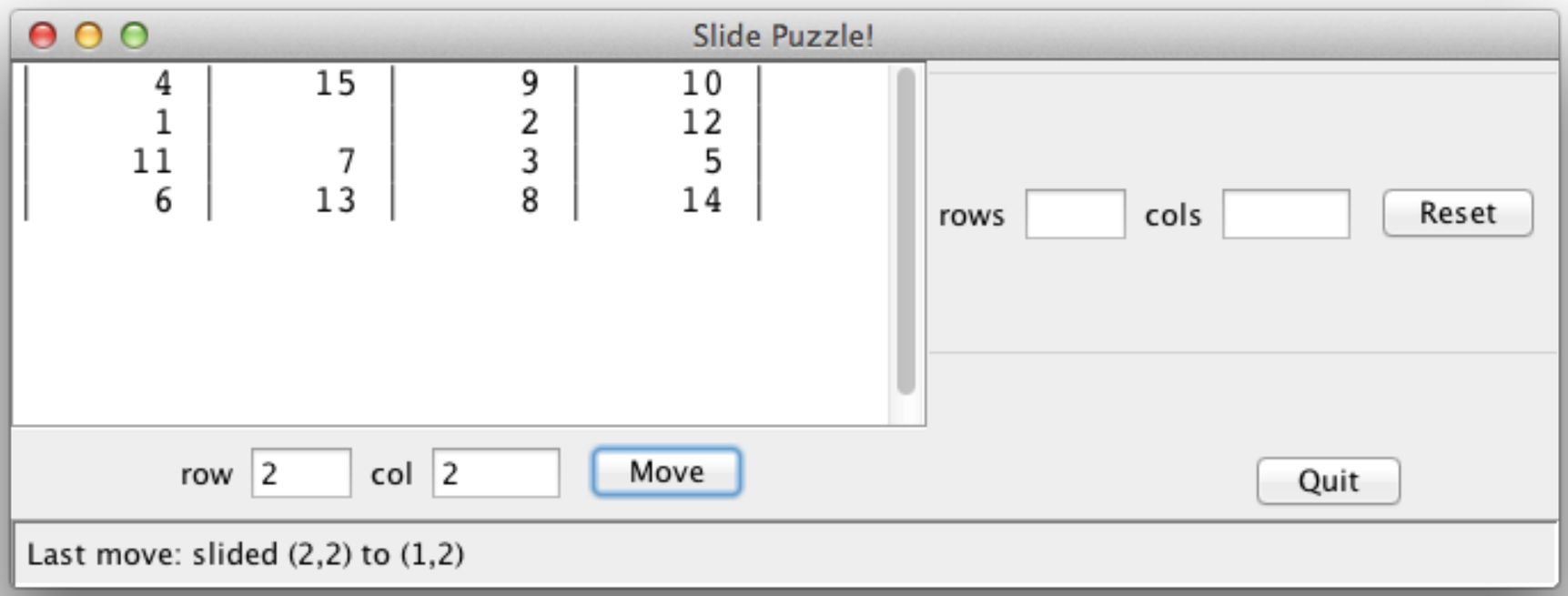
disable the move widgets when game is over

An OverView of the MVC of Ver II



Window View - Ver III

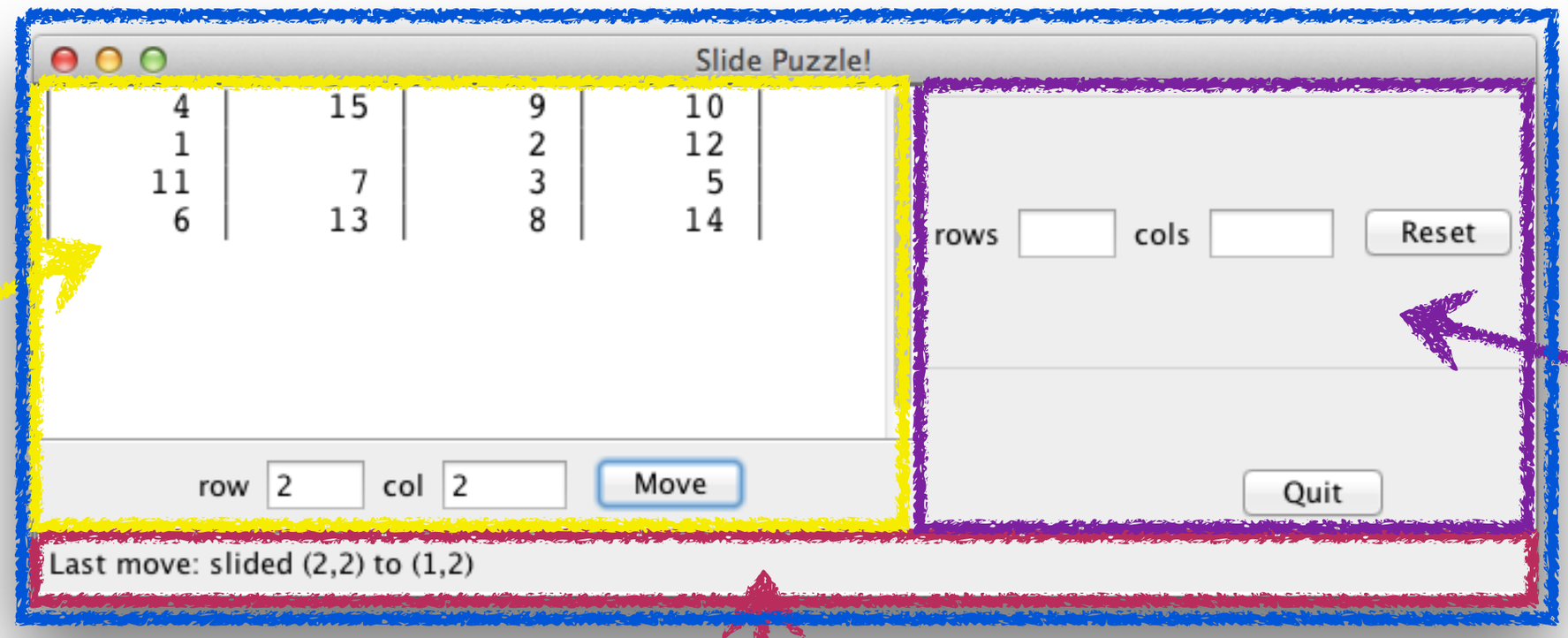
Let us add a status bar to display some information, and corresponding control panel for reseting the game, etc.



Window View - Ver III

Let us add a status bar to display some information, and corresponding control panel for reseting the game, etc.

MainWindow view



BoardPanel view

CtrlPanel view

StatusBarsPanel view

MainWindow3

```
public class MainWindow3 extends JFrame implements SlidePuzzleObserver {

    private Observable<SlidePuzzleObserver> game;
    private Controller ctrl;

    public MainWindow3(Controller ctrl, Observable<SlidePuzzleObserver> game) {
        super("Slide Puzzle!");
        this.game = game;
        this.ctrl = ctrl;
        initGUI();
        game.addObserver(this);
    }

    private void initGUI() {
        JPanel mainPanel = new JPanel( new BorderLayout() );
        this.setContentPane(mainPanel);

        JPanel boardPanel = new BoardPanel(ctrl,game);
        JPanel ctrlPanel  = new CtrlPanel(ctrl,game);
        JPanel statusBarPanel   = new StatusBarPanel(ctrl,game);

        mainPanel.add(boardPanel, BorderLayout.CENTER);
        mainPanel.add(ctrlPanel, BorderLayout.LINE_END);
        mainPanel.add(statusBarPanel, BorderLayout.PAGE_END);
        ...
    }
    ...
}
```

Create the (sub)views

Place them on the main window

CtrlPanel

```
public class CtrlPanel extends JPanel implements SlidePuzzleObserver {

    private Controller ctrl;
    private JTextField rowsTxt;
    private JTextField colsTxt;
    private JButton resetButton;
    private JButton quitButton;

    public CtrlPanel(Controller ctrl, Observable<SlidePuzzleObserver> game) {
        this.ctrl = ctrl;
        initGUI();
        game.addObserver(this);
    }

    private void initGUI() { ... }
    ...
    @Override
    public void onGameStart(Board board) { }

    @Override
    public void onMove(Board board, int sRow, int sCol, int tRow, int tCol) {}

    @Override
    public void onError(String msg) { }

    @Override
    public void onGameOver(Board board) { }
}
```

All widgets are fields so they can be used on notifications

A view by itself, it receives the game and the controller

Initialize the GUI

does not react on notifications, so in principle we could also not register CtrlPanel as a view

CtrlPanel

```
public class CtrlPanel extends JPanel implements SlidePuzzleObserver {
    ...
    private void initGUI() {
        ...
        resetButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    int rows = Integer.parseInt(rowsTxt.getText());
                    int cols = Integer.parseInt(colsTxt.getText());
                    ctrl.reset(rows, cols);
                } catch ( NumberFormatException e1 ) { }
            }
        });
        ...
        quitButton = new JButton("Quit");
        quitButton.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e) { quit(); }
        });
        ...
    }
    private void quit() {
        int n = JOptionPane.showOptionDialog(new JFrame(),
            "Are sure you want to quit?", "Quit",
            JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE, null,
            null, null);
        if (n == 0) { ctrl.requestExit(); System.exit(0); }
    }
    ...
}
```

When "reset" is clicked, a request is sent to the controller if the user provided integers in the text area

When "quit" is clicked, we ask the user to confirm, etc.

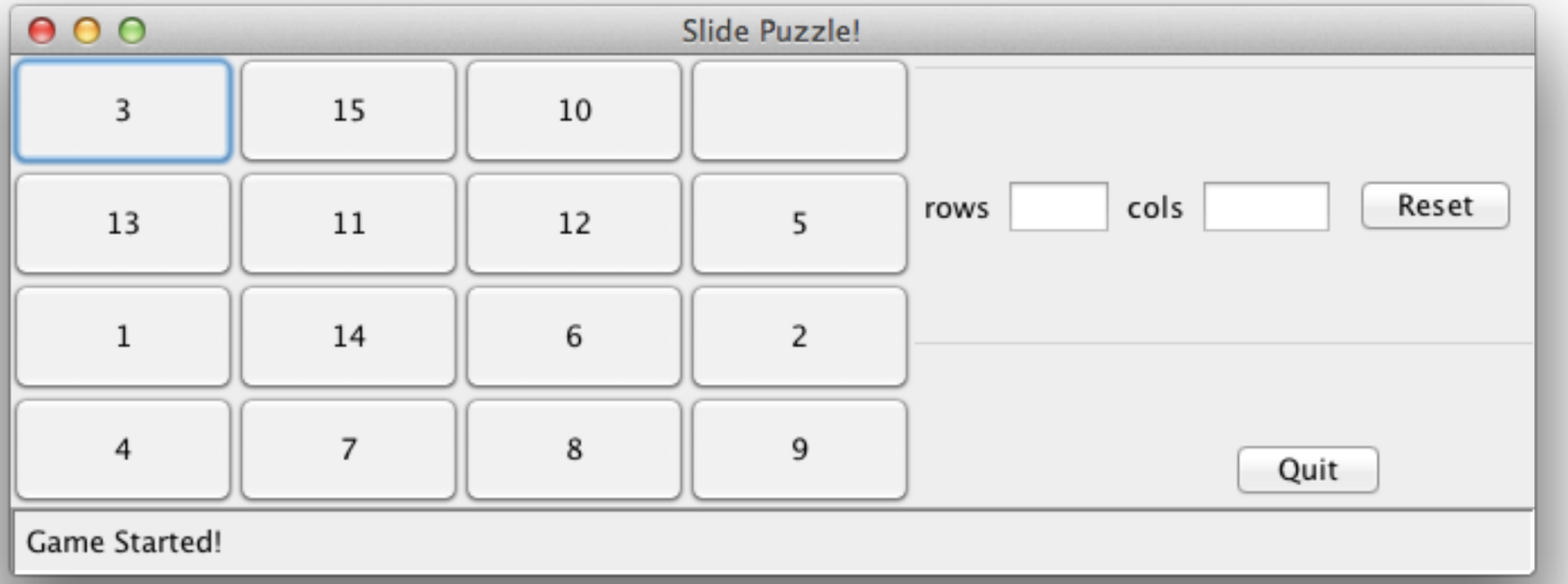
StatusBar

```
public class StatusBarPanel extends JPanel implements SlidePuzzleObserver {
    private JLabel txt;
    public StatusBarPanel(Controller ctrl, Observable<SlidePuzzleObserver> game) {
        initGUI();
        game.addObserver(this);
    }
    private void initGUI() {
        this.setLayout( new FlowLayout(FlowLayout.LEFT) );
        txt = new JLabel();
        this.add( txt );
        this.setBorder( BorderFactory.createBevelBorder(1) );
    }
    private void setMsg(String msg) {
        txt.setText(msg);
    }
    @Override
    public void onGameStart(Board board) {
        setMsg("Game Started!");
    }
    @Override
    public void onMove(Board board, int sRow, int sCol, int tRow, int tCol) {
        setMsg("Last move: slided ("+sRow+","+sCol+") to ("+tRow+","+tCol+")");
    }
    @Override
    public void onError(String msg) {}
    @Override
    public void onGameOver(Board board) {
        setMsg("Game Over!");
    }
}
```

Create a label for the msg

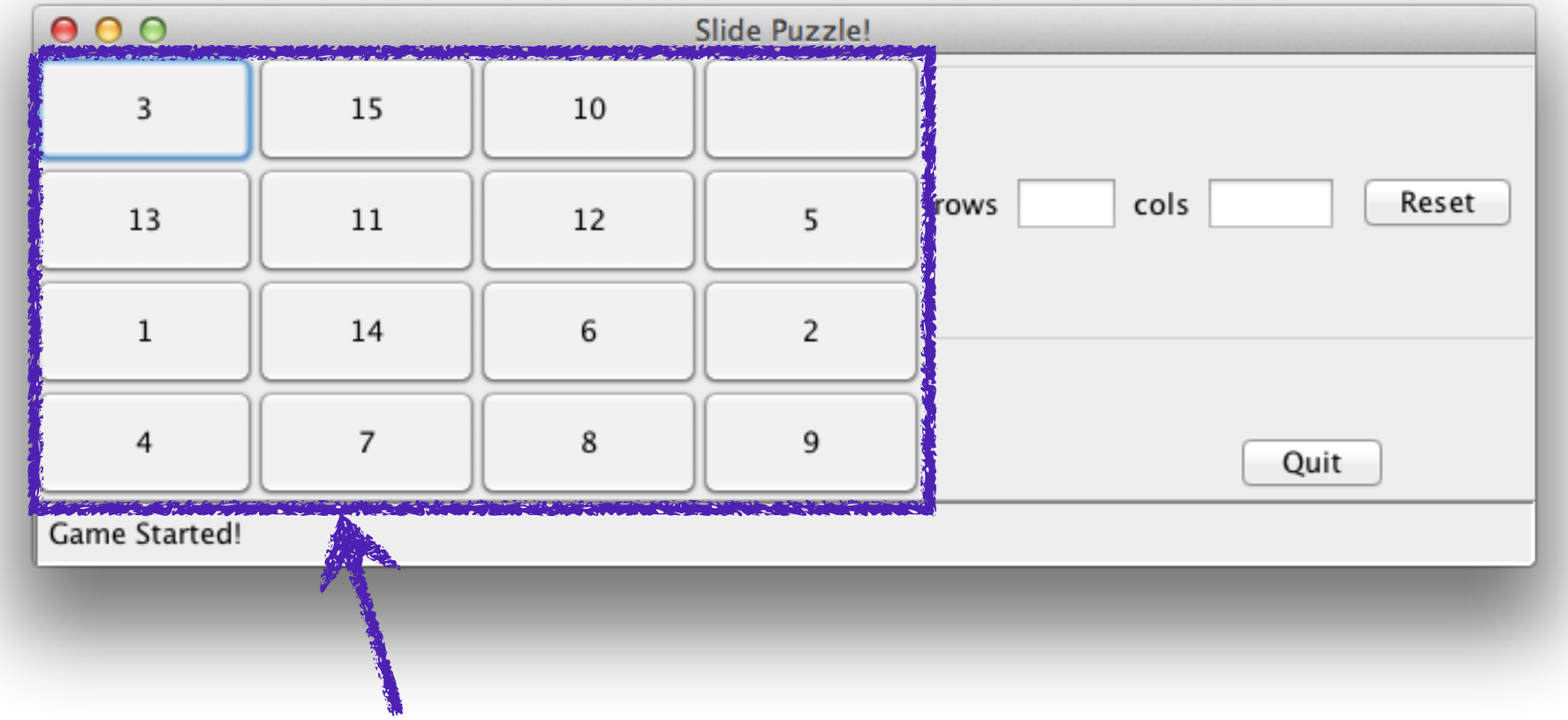
change the label text depending on the type of notification

Let us improve the BoardPanel



We will use GridBagLayout and JButtons to implement this view

Let us improve the BoardPanel



We will use GridBagLayout and JButtons to implement this view

BoardPanel2

```
public class BoardPanel2 extends JPanel implements SlidePuzzleObserver {

    private JButton[][] buttons;
    private GridBagConstraints c;
    private boolean active;
    private Controller ctrl;

    public BoardPanel2(Controller ctrl, Observable<SlidePuzzleObserver> game) {
        this.ctrl = ctrl;
        initGUI();
        game.addObserver(this);
    }

    private void initGUI() {
        this.setLayout(new GridBagLayout());
        c = new GridBagConstraints();
        c.fill = GridBagConstraints.BOTH;
        c.weightx = 1;
        c.weighty = 1;
        this.setPreferredSize(new Dimension(400, 200));
    }
    ...
}
```

1. buttons is used to store the JButtons
2. c will be used when placing the buttons
3. active will indicate if the game is over

All buttons will fill in both direction

extra space is devided between the buttons equally

BoardPanel2

```
public class BoardPanel2 extends JPanel implements SlidePuzzleObserver {

    private JButton[][] buttons;
    private GridBagConstraints c;
    private boolean active;
    private Controller ctrl;
    ...
    public void onGameStart(Board board) {
        int rows = board.getWidth();
        int cols = board.getHeight();
        buttons = new JButton[rows][cols];
        this.removeAll();

        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++) {
                int v = board.getPostion(i + 1, j + 1);
                buttons[i][j] = createButton(i, j, v);
                if ( v == 0 ) buttons[i][j].setEnabled(false);
                c.gridy = i;
                c.gridx = j;
                this.add(buttons[i][j], c);
            }
        this.revalidate();
        active = true;
    }
    ...
}
```

When reseting a game we initialize the JButtons and install them in the JPanel, according to the size of the new board

Remove the current buttons

Create and Install the new buttons

declare the game active

```
private JButton createButton(final int i, final int j, int v) {
    JButton x = new JButton(v == 0 ? "" : v + "");
    x.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            if (active) ctrl.move(i + 1, j + 1);
        }
    });
    return x;
}
```

if clicked, send a corresponding request to the controller if the game is active

BoardPanel2

```
public class BoardPanel2 extends JPanel implements SlidePuzzleObserver {
    private JButton[][] buttons;
    private GridBagConstraints c;
    private boolean active;
    private Controller ctrl;
    ...
    @Override
    public void onMove(Board board, int sRow, int sCol, int tRow, int tCol) {
        String x1 = buttons[sRow - 1][sCol - 1].getText();
        String x2 = buttons[tRow - 1][tCol - 1].getText();
        buttons[sRow - 1][sCol - 1].setText(x2);
        buttons[tRow - 1][tCol - 1].setText(x1);
        buttons[srcRow - 1][srcCol - 1].setEnabled(false);
        buttons[trgtRow - 1][trgtCol - 1].setEnabled(true);
        ...
    }

    @Override
    public void onError(String msg) {
    }

    @Override
    public void onGameOver(Board board) {
        active = false;
    }
}
```

When a move is made, exchange the labels of the corresponding buttons

When game is over, declare the game inactive, so clicks on the buttons won't have any effect

MainWindow4

```
public class MainWindow4 extends JFrame implements SlidePuzzleObserver {

    private Observable<SlidePuzzleObserver> game;
    private Controller ctrl;

    public MainWindow4(Controller ctrl, Observable<SlidePuzzleObserver> game) {
        super("Slide Puzzle!");
        this.game = game;
        this.ctrl = ctrl;
        initGUI();
        game.addObserver(this);
    }

    private void initGUI() {
        JPanel mainPanel = new JPanel( new BorderLayout() );
        this.setContentPane(mainPanel);

        JPanel boardPanel = new BoardPanel2(ctrl,game);
        JPanel ctrlPanel  = new CtrlPanel(ctrl,game);
        JPanel statusBarPanel   = new StatusBarPanel(ctrl,game);

        mainPanel.add(boardPanel, BorderLayout.CENTER);
        mainPanel.add(ctrlPanel, BorderLayout.LINE_END);
        mainPanel.add(statusBarPanel, BorderLayout.PAGE_END);
        ...
    }
    ...
}
```

Just change BoardPanel by BoardPanel2 in MainWindow3