# Frame Messenger: An different way to switch JPanels

By: Josh Moore

## 1. Overview

When I first started working on the GUI for our group, I had trouble wrapping my head around something. We had learned that double binding, where two classes should not have references to each other, because that creates a circular pattern that is a bad design choice. I was trying to find a way to where the frame could be separated from all of the panels and minimize imports (but still be able to call them) and the panels no references to the overall frame itself. What I came up with is called  Frame Messenger. It can help eliminate worries about having to import panel changes from everywhere in the GUI system, and it instead uses one static call to bring up ANY panel, from ANYWHERE in the GUI system.

## 2. Design Principles

The Frame Messenger class is a hybrid of two different design patterns: Singleton and Command.

A Singleton pattern is when a class has only ONE instantiation and persists across the entire system. The best thing about a Singleton is the ability to be called statically from anywhere in the system. At the same time, variables attached to the singleton, such as Username, can be referenced along with any methods in the singleton.

A Command pattern is a system where a class can call a function in another class, where that class interprets the input and decides which methods to use based off the input. An example of this would be an Undo/Redo function. The Undo records previous actions, so when you hit undo, an Undo method figures out what the last command was, and runs another function to revert the change based off of the last command.

## 3. Implementation

The first thing we need is a Frame Messenger class. So let's set one up:

```
public class FrameMessenger{}
```

Next thing we need to do is setup the Singleton aspect of the Frame Messenger. This is done by giving the FrameMessenger a static class field for a reference to itself, a instantiation method and a private constructor.

```
public class FrameMessenger {
	/**
	 * Field ref - A reference to itself once instantiated.
	 */
	public static FrameMessenger ref;

	/**
	 *  Private Constructor for FrameMessenger.
	 */
	private FrameMessenger(){

	}

	/**
	 * Instantiation Method for the class
	 * @return FrameMessenger
	 */
	public static FrameMessenger getMessenger(){
		if (ref==null){
			ref=new FrameMessenger();
		}
		return ref;
	}
```

So what this is doing is that if I call FrameMessenger.getMessenger() , the class will try and find a reference to itself (done by the static class variable). If that reference variable has not been set up, it creates one. The private method is to ensure that another instance cannot be made outside of the class. This is the basic setup of the singleton pattern.

Now we actually need to setup the "Messenger" Portion of the class. This is done in a couple of steps. First step is changing the FrameMessenger to keep a reference to the frame you are working with and the backbone of how to send messages to that frame. This is done by adding a static reference to said frame, a way to setup that frame, and the actual messenger to the frame itself. I will try to highlight the changes I have made. My frame class is called TestFrame.

```
public class FrameMessenger {
	/**
	 * Field ref - A reference to itself once instantiated.
	 */
	public static FrameMessenger ref;

	/**
```

```
	 * Field frame.
	 */
	public static TestFrame frame;

	/**
	 *  Private Constructor for FrameMessenger.
	 */


	private FrameMessenger(){

	}

	/**
	 * Instantiation Method for the class
	 * @return FrameMessenger
	 */
	public static FrameMessenger getMessenger(){
		if (ref==null){
			ref=new FrameMessenger();
		}
		return ref;
	}

	/**
	 * Method setFrameReference.
	 * @param inFrame TestFrame
	 */
	public static void setFrameReference(TestFrame inFrame){
		frame=inFrame;
	}

	/**
	 * Method sendPanelChangeMessage.
	 * @param inMsg String
	 */
	public static void sendPanelChangeMessage(String inMsg){
		frame.recieveMessage(inMsg, frame);
	}
```

These functions allow you to set the frame you want the messenger to talk to and then send messages to it. Keep in mind that since all of the methods are STATIC, which means you can call them from anywhere. So now we have the Messenger done for now...what do we need to change about our JFrame class? Well first we need add the function that sendPanelChangeMessage() calls, which is recieveMessage(). I will highlight the important parts below.

```
public class TestFrame extends JFrame {

	public TestFrame() {
```

```
	/** All of you JFrame GUI implementation **/

	}

	public void recieveMessage(String inMsg, JFrame frame){

		if(inMsg=="LoginPanel"){
			frame.getContentPane().removeAll();
			frame.getContentPane().add(new LoginPanel());
			frame.pack();
			frame.setVisible(true);

		/** Other Panels can be added later here **/

		}

	public static void main(String[] args) {
			EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					TestFrame frame = new TestFrame();
					frame.setMinimumSize(new Dimension(800, 600));
					frame.setName("mainFrame");
				frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
					frame.setVisible(true);
					FrameMessenger.getMessenger();
					FrameMessenger.setFrameReference(frame);
				FrameMessenger.sendPanelChangeMessage("LoginPanel");
				} catch (Exception e) {

	HospitalLogger.logError(this.getClass().getPackage().getName(), e, "UI
Error");
				}
			}
		});
	}
```

So starting with the main method of the JFrame class just above, the highlighted section shows the steps to setup and call your first frame. The first line sets up the Messenger class as a singleton. The second sets the frame of reference to the one just made in that method. The next one calls the panel change, which uses a string ("LoginPanel" in this case) to call a deletion in the content pane in the referenced panel and instantiation of a new panel (in this case LoginPanel()) using the recieveMessage function. Now all that needs to be added is additional cases to the recieveMessage function in order to move to different panels. Its that simple!

Now I was thinking that this is a good system of changing panels, but then a problem came up. The Hospital Database system we were creating had multiple panels moving towards the same panel, but I had no way to record where I had been. In order to fix this problem I added three more things to the FrameMessenger class.

```
public static Stack<String> frameStack = new Stack<String>();
```

```
public static void pushLastPanel(String inPanel){
		frameStack.push(inPanel);
	}

	public static String popLastPanel(){
		return frameStack.pop();
	}
```

These functions are used whenever you move from a frame you want to go back to. Lets say that I have a UserPanel class that I wand my AddUser to go back to when I was done, when I go to change panels I put

FrameMessenger.pushLastPanel("UserPanel");

FrameMessenger.sendPanelChangeMessage("AddUserPanel")

This puts a String on the top of a stack of previous panels, the most recent one being the top. Then in the recieveMessage function we can put:

```
if(inMsg=="LastPanel"){

		FrameMessenger.sendPanelChangeMessage(FrameMessenger.popLastPanel());

}
```

This then allows us to use the "LastPanel" command  in sendPanelChangeMessage() which will then get the last panel at the top of the stack, and then call itself again with the sendPanelChangeMessage() now with the popped command.

This system allows for easy panel calls and easy to implement towards other and allows for easy additions of extra panels if the need arises. Just at the panel to the tree in recieveMessage to make the panel and call it.

Thanks to the TA's and Bob Waters for teaching me the new design patterns to come up with this system!