The Callback Design Pattern

Aaron Stevens (azs@bu.edu)

Introduction/Motivation

A single GUI window application is straightforward enough, but any “real world” application will likely involve multiple GUI windows. The challenge of a multiple-window application is to enable the windows to communicate with each other. This is done via methods, and I call this the callback pattern.

The Cast

We can think of any multiple GUI window application as a play between windows, where one is the “master” – the main window which is shown at start up time when you run the application – and all the others are “slaves” – child windows which are shown and hidden based on a command from the “master.” (Of course, any “slave” window can be a “master” to its own “slaves”, so the pattern is repeatable infinitely.)

General Approach

All windows in a GUI application are created (at design time) as classes, and thus at run time each window is an instance of its class, or an object. Now that we can think of each window as an object, we can think about how objects interact – via method calls.

The general approach to inter-window communication is to have each window (object) export (provide) public methods which can be called its partner (e.g. the “master” can call the “slave’s” public method(s), and the “slave” can call its “master’s” public methods.

Recall that to call a method, the caller needs to have an object reference to the object which provides that method. Thus, the Master will have a reference to the Slave object, and the Slave will have a reference to the Master object.

An Example

Consider two classes, Master and Slave. Here are some simple declarations for each:

```
class Master
{
  // … hold a reference to the Slave window object
  Slave slave;

  public void sendMessage(String msg)
  {
       // … Master does something when message received
  }
}

class Slave
{
  // … hold a reference to the Slave window object
  Master master;

  public void sendMessage(String msg)
  {
       // … Slave does something when message received
  }
}
```

In your application, of course, both Master and Slave will be Jigloo-created windows, extending JFrame (master) and JDialog (slave).

This example shows that the Master has a member variable which is a reference to the Slave, and vice-versa. When the Master needs to communicate to the Slave, it will make a method call like:

```
public void someActionEventhandler(ActionEvent e)
{
    String text = “some message”;
    slave.sendMessage(text);
}
```

This works similarly in the other direction too. Of course, the methods are not limited to sending messages, but can be any type of method, especially accessor and mutators.

Initializing the Callback References

The above model is helpful to understand basics of communications between the windows. However, the actual code you will write will need to account for one more thing: setting up the references between the objects.

There are several approaches to setting up the references for objects to call back to each other, but the simplest approach is to use mutator methods. Consider the example code again, with these master/slave reference mutator methods:

```
class Master
{
  // … hold a reference to the Slave window object
  Slave slave;

  public void setSlave(Slave slave)
  {
      this.slave = slave;          // store a reference to the slave object
  }

  public void sendMessage(String msg)
  {
       // … Master does something when message received
  }
}

class Slave
{
  // … hold a reference to the Slave window object
  Master master;

  public void setMaster(Master master)
  {
      this.master = master;      // store the reference to the master object
      master.setSlave(this);     // sets the Master’s slave ref. to current object
  }

  public void sendMessage(String msg)
  {
       // … Slave does something when message received
  }
}
```

Note that the Slave’s *setMaster* method will call the Master back to set its Slave reference, so that both references will be set by the act of the Master calling the slave’s *setMaster* method once.

Now all we need is a way to “kick off” the process of setting these references. The best place to do this is where the Master window creates the Slave window the first time, e.g.:

```
public void btnPopUpChildWindowActionHandler(ActionEvent e)
{
     // time to create and pop up the slave window:
     Slave slaveWindow = new Slave();

     // set the call back references:
     slaveWindow.setMaster(this);

}
```

And that’s that. Once these callback references are set up, any communication between the windows is done by calling the public methods you write for this purpose. From a design perspective, it’s important to think carefully about a given method: which window should provide the method, and which window will call the method?