SYS-CON

THE PARADOX OF WRITING PERFECT CODE PAGE 20

No. 1 *i*-Technology Magazine in the World

JDJ

JDJ.SYS-CON.COM VOL.12 ISSUE:1

Mediation and the Man in the Middle

Patterns for designing scalable and robust user-interfaces

RETAILERS PLEASE DISPLAY
UNTIL MARCH 31, 2007
$5.99US $6.99CAN
02>
0 09281 01751 6

PLUS...

- EJB 3 Transactions
- Configuring WebLogic Server 9.x JDBC
- What Is SCA?
- Performance Management 101 for WebLogic Portal

Mediation and the Man in the Middle

Patterns for designing scalable and robust user-interfaces

by Michael Birken

Even for many seasoned developers, Swing code can be notoriously difficult to organize. Where is the right place to put parsing and validation logic? How do you prevent those threading issues that cause lockups or repainting glitches? Is it possible to unit test GUI logic? Can the code somehow be shared with other user-interfaces, like a web front-end? If these questions sound familiar, the solutions presented here may revolutionize the way you code with Swing.

Two-Layer Separation

Suppose you want to offer your end-users a Swing-based product built on top of a homegrown API, such as a mathematics package, that they can actually license for their own development purposes or perhaps even use to extend your product through a plugin mechanism. To achieve that goal, the API must be fully decoupled from all Swing code. Let's take a look at a simple example of such an API.

In the class Divider, I defined the following method:

```
public DivisionResult divide(
      int dividend, int divisor, IDivisionListener divisionListener)
      throws ArithmeticException { ... }
```

Given a dividend and a divisor it returns a DivisionResult, a simple bean containing quotient and remainder. If divisor is 0, it throws an ArithmeticException. I'll discuss the divisionListener parameter below.

Now, let's slap on a JFrame to exploit this API. DivisionFrame (see Figure 1) contains 2 JTextFields that enable the user to enter a dividend and a divisor. When the user presses the Divide button, the resultant quotient and remainder are displayed in a JLabel.

The simplest attempt at a separation of concerns is a GUI layer built directly on top of the API layer. The GUI layer consists of JFrames, JDialogs, Swing components and their data models. The API layer contains the business logic and as mentioned above, it should be possible to use the API layer without a GUI.

Since there are only 2 layers in this approach, user inputs must be prepared for the API layer in the GUI layer. In this case, Divider.divide() accepts integers, not strings. The ActionListener bound to the Divide button parses the values provided by the JTextFields and it either calls divide() or it changes the text color red to indicate invalid input.

Keep in mind that Divider actually represents an advanced mathematics package. Its methods may take several seconds or minutes to complete. I decided to simulate that effect by sleeping for 5 seconds inside of divide(). If divide() is called directly by the ActionListener, the GUI will appear frozen and possibly grayed-out for that time because of Swing's one-threaded nature (see the sidebar, The Event Queue).

The general solution is a request-response model. The GUI layer makes a request into the API layer on a new thread. That thread takes as long as is needed to complete the operation, freeing the event-dispatching thread to continue servicing the event queue. When the result of the operation is ready, the thread uses one of the static invokeXX() methods of the EventQueue class to safely update the GUI by requesting that the event-dispatching thread handle the update on its behalf.

In this model, a thread boundary exists between the layers: the event-dispatch thread is the only thread that runs within the GUI layer, worker threads run within the API layer, and neither type is allowed to cross the boundary. Unfortunately, there is no mechanism to automatically prevent a rogue thread from slipping through. Also, the code required to keep switching threads tends to be voluminous and ugly even with the aid of SwingWorker (see Resources).

Three-Layer Separation

Consider introducing a layer between the GUI and API layers to mediate data between them. This mediation layer is responsible for converting user input into a format acceptable by the API layer and for converting resultant values and other data from the API layer back into a format acceptable by the GUI layer. Ideally, the boundaries between the 3 layers should be formalized with interfaces; though, that may not be entirely possible if the API was provided by a third-party vendor. Finally, the event-dispatching thread is not permitted to cross into the mediation layer. Worker threads can pass back-and-forth between the mediation and API layers; however, they cannot enter the GUI layer.

Michael Birken is actively involved in the design and research of emerging trading technologies at a Manhattan-based financial software company. He's a Sun Certified Java programmer and developer. Michael holds a BS in computer engineering from Columbia University.

o__1@hotmail.com

For our simple example, I created a single class, aptly named Mediator, to serve as the entire mediation layer. To formalize the layers, Mediator implements IDivisionFrameMediator, which allows DivisionFrame to pass the values in the JTextFields directly as strings:

```
public interface IDivisionFrameMediator {
  public void divide(String dividendStr, String divisorStr);
}
```

For the reverse direction, DivisionFrame implements IDivisionFrame. It enables Mediator to push a DivisionResult back to DivisionFrame in the form of a string and to mark the input fields as valid or invalid:

```
public interface IDivisionFrame {
  public void showDivisionResult(String divisionResult);
  public void showValid(boolean dividend, boolean divisor);
}
```

Mediator is loosely-coupled to DivisionFrame; they each hold a handle to each other as an interface type. However, the handle cannot be a direct reference to the object in the opposing layer because that would break the thread-layer separation rules discussed above.

We could solve the problem by creating 2 proxy classes (see the sidebar, Proxies). Let's call these hypothetical classes MediatorProxy and DivisionFrameProxy. MediatorProxy implements IDivisionFrameMediator and contains a reference to Mediator. Whenever you invoke a method of MediatorProxy, it spawns off a new thread and uses it to call the corresponding method in Mediator. Similarly, DivisionFrameProxy implements IDivisionFrame and contains a reference to DivisionFrame. Calling a method of DivisionFrameProxy delegates the invocation to DivisionFrame using EventQueue.invokeLater().

With such proxies, the code in DivisionFrame and Mediator appears immaculate. DivisionFrame calls directly into the methods of its IDivisionFrameMediator and Mediator does the same with its IDivisionFrame. It's still a request-response model, but the proxies hide the thread switching details and there's no chance of a thread inadvertently slipping by. However, creating the proxies themselves is a tedious and repetitive task you shouldn't have to do yourself.

SwingProxy

To obviate the need to code the proxies by hand, I created a utility class called SwingProxy that dynamically generates them for you. SwingProxy provides a static method, newSwingProxy(), that accepts the target object and returns a new proxy that can be cast to any of the interface types that the target implements. For example, for an instance of DivisionFrame, which implements IDivisionFrame, you can create a proxy for it as follows:

```
IDivisionFrame divisionFrameProxy =
    (IDivisionFrame)SwingProxy.newSwingProxy(divisionFrame);
```

The proxy automatically takes care of the thread switching. In this case, if a method of divisionFrameProxy is invoked by a worker thread, it will call the corresponding method of divisionFrame with the event-dispatching thread. Similarly, a call by the event-dispatching thread will turn into a call by a worker thread.

The Event Queue

Swing provides a single thread, the *event-dispatching thread*, for servicing all user-interface requests, including repainting components. When you press a button, its associated ActionListener is not executed immediately. Instead, the request to execute it is placed onto the *event queue*. The event-dispatching thread pulls requests off the queue and executes them one-by-one, eventually handling the button press. If you implement a component listener that performs a time-consuming task or that makes a blocking call, the event-dispatching thread will sit there completely dedicated to it, neglecting the other requests on the event queue. The result is a non-responsive GUI that typically appears grayed-out because invalidated components make requests for repaint that also must pass through the event queue.

The event-dispatching thread is considered the only the thread that can safely access Swing components and their data models. A worker thread can request that the event-dispatching thread execute code on its behalf using invokeLater() or invokeAndWait(), static methods of the EventQueue class. Both methods accept a Runnable implementation where run() contains the code to execute. invokeLater() inserts the Runnable into the event queue and returns immediately. On the other hand, invokeAndWait(), enqueues the Runnable and waits until it is serviced by the event-dispatching thread before returning.

SwingProxy (see Listing 1) is built around java.lang.reflect.Proxy, a class that is capable of producing a proxy of a specified type at runtime. The Proxy utility produces proxies that resemble funnels; to create the proxy, you supply an implementation of the InvocationHandler interface and when you invoke any method of the proxy, it automatically gets funneled down to InvocationHandler's single method, invoke():

```
public interface InvocationHandler {
  public Object invoke(Object proxy, Method method,
      Object[] args) throws Throwable;
}
```

The first parameter is a reference to the proxy. The second is the method that was called in the form of a reflected method type. The third is an object array containing the arguments passed to the method.

SwingProxy contains 2 private inner classes. The first, CallHandler, implements InvocationHandler. Its invoke() method inspects the calling thread and switches accordingly. When it needs to delegate a call from a worker thread to the event-dispatching thread, it examines the return type of the method. If it returns void, invokeLater() is used. Otherwise, invokeAndWait() is called and the return value is passed back to the worker thread that called the proxy. When the proxy is called by the event-dispatching thread, the call is delegated to a worker thread and nothing is ever returned. The interface methods specifying calls from the GUI layer to the mediation layer should always return void. The worker threads that CallHandler use originate from a thread-pool supplied by java.util.concurrent.Executors.

Figure 1 DivisionFrame allows the user to enter a dividend and a divisor and get back a quotient and a remainder

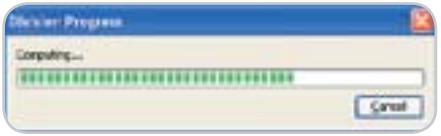

Figure 2 ProgressDialog displays the percent completed and allows the user to cancel the computation

Proxies

A proxy is an object that serves as a middleman for communication between a *source object* and a *target object*. Typically, the proxy and the target objects share the same interface. The source object holds a reference to target as the interface type, which makes the source oblivious to whether it is communicating directly with the target or communicating through the proxy.

Proxies are used throughout the Java APIs. Remote Method Invocation (RMI), for instance, allows a program to communicate with a target object on a remote machine and treat it as if it were a local object. The program actually holds a reference to a proxy that hides the networking details. The target object and the proxy share a common interface and the program only refers to the target as that type. The Java API for XML Web Services (JAX-WS), which supplants the Java API for XML-based Remote Procedure Call (JAX-RPC), performs a similar service over a different protocol.

The second inner class, TargetInvoker, implements Runnable, which allows it to be executed on a different thread. It performs the actual method invocation using reflection in its run() method.

Pushing Data to the GUI

The three-layer design described above is not limited a request-response model. The API can stream data to the GUI by pushing it through the mediation layer. To demonstrate this, I created a dialog to display the (artificially prolonged) progress of the division computation (see Figure 2).

ProgressDialog contains a JProgressBar to show completion percentage, a JLabel for status messages, and a button that allows the user to abort the computation. ProgressDialog lives within the GUI layer and to formally separate it from the other layers, it implements IProgressDialog:

```
public interface IProgressDialog {
  public void start();
  public void setProgress(String message, int progress);
  public void end();
}
```

Mediator holds a proxy to ProgressDialog as that type. start() makes the dialog appear, setProgress() updates the status label and the progress bar, and end() hides the dialog. In turn, ProgressDialog holds a proxy to Mediator in the form of an IProgressDialogMediator, which contains a method that is called when the Cancel button is pressed:

```
public interface IProgressDialogMediator {
  public void requestCancel();
}
```

Divider.divide() was written to abort if the executing thread is interrupted. Mediator obtains a reference to that thread before invoking Divider and requestCancel() simply interrupts it.

The third parameter of Divider.divide() is an IDivisionListener, which is repeatedly called back during the 5 second pause to simulate progress notifications:

```
public interface IDivisionListener {
  public void computationPerformed(int percentage);
}
```

Mediator implements IDivisionListener and it delegates the call to IProgressDialog.setProgress(). For time consuming methods that don't provide such a listener, JProgressBar can be put into indeterminate mode. That mode shows an animation conceptually similar to the moving logo in the corner of a web browser.

Mediator as the Controller

It is important to recognize that the mediation layer consists of more than just bidirectional adapters that convert data between formats. It contains the control logic that governs when and how the parts from the other layers are used. The control logic should not be intermingled with the adaptation logic that performs the parsing and the static validation.

In this simple example, Mediator.divide() contains most of the control logic (refer to Listing 2). As talked about above, divide() is invoked by the Divide button and it passes the user input fields as strings. Instead of attempting to parse the strings directly within divide(), they are used to create instances of a class called DivisionFrameParser. The constructor of DivisionFrameParser accepts a string field and parses it. The class provides methods to check if the parsing was successful and to retrieve the field as an integer. In this way, Mediator is focused on the interaction of classes across the layers and less on processing the shuttled data.

Alternate UIs

Mediator, DivisionFrame and ProgressDialog expose setters to enable their dependencies to be injected prior to use. The Main class, which serves as the entry point of the application, wires everything up. However, with Mediator loosely-coupled to the GUI layer, it's possible to completely replace the user-interface. To prove it, if you launch the application with a "-t" command-line argument, it will run in text-mode. Text-mode prompts the user for a dividend and divisor and it prints results back to the console. It was made possible by providing Mediator with alternate implementations of IDivisionFrame and IProgressDialog.

What about a web front-end? Web applications serve data on demand; data is not easily pushed to the browser. This makes showing progress updates, for instance, fairly tricky. As with text-mode, implementing a web front-end entails creating a new GUI layer, but it will also require additions to the mediation layer. The control logic discussed above was designed with pushing data in mind. New controller classes will be required for the request-response nature of the web; however, since we separated the parsing and validation logic from the control logic, it can be reused in a web application.

Testing

The GUI classes do not need to be connected to Mediator to launch them. To demonstrate this, I inserted a main() method into DivisionFrame that uses a mock implementation of IDivisionFrameMediator. In this case, the division request is simply printed out to the console.

Using mock implementations to represent the rest of the system is an especially useful technique if you are developing

without the aid of a GUI builder because it will enable you to quickly view your efforts without launching the complete application. Mock implementations can allow you to fully exercise the features provided by the GUI with much less effort. For example, you do not need to induce a problem in the real application just to make sure that error messages get displayed correctly.

Mock implementations are also the hallmark of automated unit testing. The online code that supplements this article includes JUnit tests for Divider, Mediator and DivisionFrameParser. To make Mediator easier to test, I made Divider implement IDivider, an interface that contains its single method. The complete execution cycle for Mediator is tested with mock implementations of IDivider, IDivisionFrame and IProgressDialog.

Real-Time Interaction

Suppose you want to alter DivisionFrame such that as you key in dividend and divisor, the text immediately turns red if it doesn't represent valid numerical data as opposed to after pressing the Divide button. You can bind a KeyListener to the JTextFields and receive an event for every keystroke, but where should you do the validation? One option is to delegate the key events to Mediator and let it validate and callback DivisionFrame; however, that cycle of execution is significantly different from the ones discussed above because the API layer is never invoked.

The mediation layer provides adaptation for the API layer and it shouldn't be used where the API is not needed. In this case, the behavior is entirely specific to the user-interface and nothing is gained by thread switching. DivisionFrameParser, which encapsulates the static validation logic, should be used directly inside of a KeyListener. The Mediator class remains the same and it double checks that the fields are valid also via DivisionFrameParser.

Sharing Objects

Consider a Swing application that contains a JTable with thousands of rows. Each row is a view into a simple bean where the columns are mapped to the bean properties. The table effectively enables the user to view and edit beans. The API layer requires access to a collection of those beans to function. It occasionally modifies bean properties and those changes should be reflected on the front-end.

In this example, which layer owns the beans? The beans are objects that can be manipulated by both the event-dispatching thread and worker threads apparently violating the layer-separation rules. Breaching the layers can be dangerous. For instance, if a worker thread were to remove a bean from the TableModel in the middle of a repaint, the event-dispatching thread could loop past the end of the collection and throw an exception.

Since a bean is simply a container whose methods execute in a timely manner, they can be shared by all layers with a little help. First, all methods of the bean should be synchronized to enable threads to access them atomically. For example, a worker thread should be able to obtain a lock on a bean and read several properties needed for a computation, knowing that all reads represent a consistent view. Though, it must be recognized that until that lock is released, the event-dispatching thread is potentially held up.

Second, when the API modifies a bean property, the JTable needs to be notified so it can repaint the associated cell. The bean requires a listener, such as java.beans.PropertyChangeListener, which is invoked by the setters. A mediator class would implement the listener and forward the event to the TableModel via SwingProxy to allow it to resolve the coordinates of the cell that requires repainting. When a cell is edited by the user, TableModel.setValueAt() is called. In that method, the event-dispatching thread would grab a lock on the bean, remove the listener to prevent the TableModel from being called back, update the property and restore the listener before releasing the lock.

Finally, the beans should originate from a factory class that provides a create() method and a destroy() method, and it would notify a mediator when either of those methods are called. In turn, the mediator would update the interested API collections and the TableModel in a thread-safe manner.

Listing 1

```
package example;

import java.lang.reflect.*;
import java.util.concurrent.*;
import java.awt.*;

public final class SwingProxy {

  private static class TargetInvoker implements Runnable {

    private Object target;
    private Object returnValue;
    private Throwable exception;
    private Method method;
    private Object[] arguments;

    public TargetInvoker(Object target, Method method,
       Object[] arguments) {
      this.target = target;
      this.method = method;
      this.arguments = arguments;
    }

    public boolean threwException() {
      return exception != null;
    }

    public Throwable getException() {
      return exception;
    }

    public Object getReturnValue() {
      return returnValue;
    }

    public void run() {
      try {
        returnValue = method.invoke(target, arguments);
      } catch(Throwable t) {
        exception = t;
      }
    }
  }

  private static class CallHandler implements InvocationHandler {

    private static final ExecutorService threadPool
        = Executors.newCachedThreadPool();
    private Class targetClass;
    private Object target;

    public CallHandler(Object target) {
      this.target = target;
      this.targetClass = target.getClass();
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {

      Method targetMethod = targetClass.getMethod(
            method.getName(), method.getParameterTypes());
```

We can get away with using DivisionFrameParser directly inside of a KeyListener because the parsing and validation occur almost instantly. If we required something more advanced, such as real-time spelling and grammar checking, then a call into the mediation layer is justified because that kind of validation will require a specialized API and it's unlikely to execute as timely. However, we must consider that every key press gets delegated to an independent thread. If you were to type too quickly, several threads will needlessly be processing the same input in parallel. To resolve this issue, we need to setup the KeyListener to take the request-response nature of the mediation layer into account. The KeyListener shouldn't call into mediation layer if it's already validating an input field in response to a prior key press; rather, the KeyListener should simply mark the field as changed. When the validation is complete and the GUI layer is called back, that invocation can check if the field has changed and make a successive call into the mediation layeraccordingly.

Conclusion

I hope you find the patterns discussed here useful in your own development efforts. The full source code including unit tests for the division application can be found at the online version of this article at http://jdj.sys-con.com. DivisionFrame and ProgressDialog were created using NetBeans 5.0. Refer to the NetBeans documentation for using the Swing Layout Extension library outside of NetBeans.

Resources

- Mediator pattern: http://en.wikipedia.org/wiki/Mediator_pattern
- Event-dispatching: http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html
- SwingWorker: https://swingworker.dev.java.net/
- Dynamic proxies: http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html
- Reflection: http://java.sun.com/docs/books/tutorial/reflect/index.html
- JUnit: http://www.junit.org/index.htm
- NetBeans: http://www.netbeans.org/

```
      TargetInvoker targetInvoker = new TargetInvoker(
          target, targetMethod, args);
      if (EventQueue.isDispatchThread()) {
        threadPool.execute(targetInvoker);
      } else if (method.getReturnType() == void.class) {
        EventQueue.invokeLater(targetInvoker);
      } else {
        EventQueue.invokeAndWait(targetInvoker);
        if (targetInvoker.threwException()) {
          throw targetInvoker.getException();
        } else {
          return targetInvoker.getReturnValue();
        }
      }

      return null;
    }
  }

  private SwingProxy() {
  }

  public static Object newSwingProxy(Object target) {
    return Proxy.newProxyInstance(
        SwingProxy.class.getClassLoader(),
        target.getClass().getInterfaces(),
        new CallHandler(target));
  }
}
```

Listing 2

```
package example;

public class Mediator implements IProgressDialogMediator,
    IDivisionFrameMediator, IDivisionListener {

  private IDivider divider;
  private IDivisionFrame divisionFrame;
  private IProgressDialog progressDialog;
  private Thread divideThread;

  public void setDivider(IDivider divider) {
    this.divider = divider;
  }

  public void setDivisionFrame(IDivisionFrame divisionFrame) {
    this.divisionFrame = divisionFrame;
  }

  public void setProgressDialog(IProgressDialog progressMonitor) {
    this.progressDialog = progressMonitor;
  }

  public void divide(String dividendStr, String divisorStr) {
    DivisionFrameParser dividendParser
        = new DivisionFrameParser(dividendStr);
    DivisionFrameParser divisorParser
        = new DivisionFrameParser(divisorStr);
    divisionFrame.showDivisionResult("");
    divisionFrame.showValid(dividendParser.isFieldValid(),
        divisorParser.isFieldValid());

    if (dividendParser.isFieldValid()
        && divisorParser.isFieldValid()) {
      try {
        progressDialog.start();
        synchronized(this) {
          divideThread = Thread.currentThread();
        }
        DivisionResult divisionResult = divider.divide(
            dividendParser.getField(),
            divisorParser.getField(), this);
        synchronized(this) {
          divideThread = null;
        }
        if (divisionResult == null) {
          divisionFrame.showDivisionResult("[Cancelled]");
        } else {
          divisionFrame.showDivisionResult(
            divisionResult.getQuotient()
              + " R " + divisionResult.getRemainder());
        }
      } catch(ArithmeticException e) {
        divisionFrame.showDivisionResult("NaN");
      } finally {
        progressDialog.end();
      }
    }
  }

  public synchronized void requestCancel() {
    if (divideThread != null) {
      divideThread.interrupt();
    }
  }

  public void computationPerformed(int percentage) {
    progressDialog.setProgress("Computing...", percentage);
  }
}
```