Jazz - Java Desktop Application Development Made Easy

Introduction
Building a Graphical User Interface (GUI) with the standard Java Swing API can be a daunting task for the novice and experienced Java Swing developer alike. Using this API requires a steep learning curve due to its size and complexity. It also requires quite a bit of blood-sweat-and-tears to write your application, because Swing doesn't offer you all the bits-'n-pieces you initially may have expected straight out-of-the-box.

The Jazz toolkit was developed to ease this pain and make any Swing developer more productive. Jazz provides you with a set of high-level, enterprise-ready constructs that intuitively match with your own perceptions of what a GUI should actually encompass. This article will explain some of the main concepts behind Jazz and demonstrate a few of the benefits over plain Swing.

To give a clear picture of Jazz, let's first explain what the Jazz toolkit is **not**. It is **not** an alternative to the Swing GUI library like for instance SWT (Standard Widget Toolkit). Quite to the contrary. Jazz builds on the strengths of Swing but tries to resolve its weaknesses in elegant ways. That said, Jazz is **not** intended as a solution to improve *existing* Swing applications, save from rebuilding the whole application again from scratch. It merely provides a fast-track solution to create new, enterprise-ready applications.
The Jazz toolkit is also **not** a visual GUI designer like for instance JFormDesigner. Jazz is focused on the presentation logic that drives your GUI, not on how it looks. It simply supports the LayoutManagers that are available in Swing (FlowLayout, BorderLayout, GridbagLayout, etc.). Or any other custom LayoutManager that adheres to the Swing conventions for that matter.

Concepts
The most pervasive design pattern when building a GUI is the Model View Controller (MVC) pattern. Since the MVC pattern is by now considered a classic, we assume that you have a basic understanding of this design pattern. If you have never heard of the MVC pattern before, it is strongly recommended that you consult a design patterns reference book before you continue reading this article.

The other pillar on which Jazz leans heavily is Event-Driven Architecture (EDA). According to wikipedia.org, EDA is "*a software architecture pattern promoting the production, detection, consumption of and reaction to events*". When you develop with Swing (or most other GUI frameworks for that matter) you already should be familiar with the concept of events.

Jazz conforms to and realizes an event-driven architectural pattern and is designed around the MVC design pattern. For this purpose the main package of Jazz contains the classes: Model, View and Controller (how obvious). Besides these three classes, the main package contains three additional classes: Application, Event and Task. (There exists another class in the main package called ViewManager, but this is just an utility class. As you don't interact with this class directly, you can just take a mental note and safely ignore it.) This set of six classes and how to use them is essentially all you need to know to build a full-fledged GUI. When you previously have tried to work out how to use the standard Swing API I hope I already made the point that Jazz is far easier to learn.

HelloWorld Application
Let's start of with a well-known tradition; building a HelloWorld Application. The common way to do this in standard Swing looks something like this (taken nearly straight from the Swing Tutorial):

```
import java.awt.*;

import javax.swing.JFrame;
import javax.swing.JLabel;

public class StandardSwingApplication {

	/**
	 * Main method.
	 *
	 * @param args Command line argument values.
	 */
	public static void main(String[] args) {

		JFrame frame = new JFrame("Standard Swing Application");
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		JLabel label = new JLabel("Hello world!");

		Container contentPane = frame.getContentPane();
		contentPane.setLayout(new BorderLayout());
		contentPane.add(label, BorderLayout.CENTER);

		frame.setBounds(new Rectangle(40,50,250,200));
		frame.setVisible(true);
	}
}
```

We can build the same application with the help of Jazz and it will look something like this:

```
import nl.coderight.jazz.Application;
import nl.coderight.jazz.form.FormView;

public class MyApplication extends Application {

	private FormView view;

	public MyApplication() {
		this.view = new HelloWorldView();
	}

	@Override
	public void execute() {
		setView(view);
		showView();
	}

	/**
	 * Main method.
	 *
	 * @param args Command line argument values.
	 */
	public static void main(String[] args) {
```

```
            MyApplication app = new MyApplication();
            app.execute();
      }
}

/* The view */

import java.awt.*;

import nl.coderight.jazz.form.FormView;
import nl.coderight.jazz.form.field.LabelField;

public class HelloWorldView extends FormView {

      @Override
      public void init() {
            setTitle("My Application");
            setLayout(new BorderLayout());
            setSize(new Dimension(250,200));
            center();

            addField(new LabelField("Hello World!"), BorderLayout.CENTER);
      }

}
```

The code doesn't look very different from the plain Swing code sample. This is a Good Thing, because it demonstrates that Jazz remains close to the standard Swing coding conventions. We're trying to make your life easy, remember?

You may also have noticed that Jazz requires you to create **two** classes compared to only one for plain Swing. You may wonder; what is the benefit of that? Well, even this simple example demonstrates that Jazz gently guides you towards clean MVC code in order to separate the concerns. In the plain Swing example we also (implicitly) used a view, but it doesn't show from the code itself. This isn't really an issue in this simple case, but it rapidly can become a major issue when your application's complexity increases.

Also note the difference in the code that is required. In the plain Swing example you require various method calls on two class instances **other** than your own class instance, namely one of type JFrame and one of type Container. Experienced Swing developers are familiar with this code, but to a novice Swing developer this isn't very intuitive and this adds up to the steep learning curve. The code in the Jazz application **only** performs method calls on itself and this is far more self-explanatory. And again, this gently encourages you to write clean code and prevents you from ending up with spaghetti code.

Simple Interaction

The purpose of a GUI is to allow the user to interact with an application. For this purpose, your application needs to know what events the user is performing. The standard way to handle this in Swing is by way of Actions. The following code sample shows a simple implementation of an Action that listens for action events fired by a button:

```
import java.awt.*;
import java.awt.event.ActionEvent;

import javax.swing.*;
```

```
public class StandardSwingApplication {

	private JLabel label;
	private JTextField textField;

	/**
	 * Main method.
	 *
	 * @param args Command line argument values.
	 */
	public static void main(String[] args) {

		StandardSwingApplication app = new StandardSwingApplication();

		app.execute();
	}

	private void execute() {

		JFrame frame = new JFrame("Standard Swing Application");
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		this.label = new JLabel("What's your name?");

		this.textField = new JTextField();

		JButton button = new JButton("Enter");
		button.addActionListener(new SimpleAction());

		JPanel entryPanel = new JPanel(new BorderLayout());
		entryPanel.add(textField, BorderLayout.CENTER);
		entryPanel.add(button, BorderLayout.EAST);

		Container contentPane = frame.getContentPane();
		contentPane.setLayout(new BorderLayout());
		contentPane.add(label, BorderLayout.NORTH);
		contentPane.add(entryPanel, BorderLayout.SOUTH);

		frame.setBounds(new Rectangle(40,50,250,200));
		frame.setVisible(true);
	}

	@SuppressWarnings("serial")
	class SimpleAction extends AbstractAction {

		public void actionPerformed(ActionEvent e) {
			if (e.getID() == ActionEvent.ACTION_PERFORMED) {
				System.out.println("action! "
					+ textField.getText());

				label.setText("Hello "
					+ textField.getText() + "!");
			}
		}
	}
}
```

As you can see from this simple Swing example, Action classes are tempting to use as inner classes in order to easily access the data in your fields. This isn't a problem when you have only one or two Action classes, but your code can quickly become unwieldy when this

number increases (and it will). This again proofs the point that it's easy with Swing to create bloated objects that are hard to understand and even harder to maintain.

Now let's modify the previously created Jazz application to implement the same functionality. First change the view instantiated in the constructor of the application:

```
public class MyApplication extends Application {

	(…)

	public MyApplication() {
		this.view = new PersonView();
	}

	(…)
}
```

Next, create the new view:

```
import java.awt.*;

import nl.coderight.jazz.form.FormView;
import nl.coderight.jazz.form.control.GroupControl;
import nl.coderight.jazz.form.field.InputField;
import nl.coderight.jazz.form.field.LabelField;
import nl.coderight.jazz.form.field.button.PushButton;
import nl.coderight.jazz.form.field.event.OnClickEvent;

public class PersonView extends FormView {

	private InputField<String> textField;
	private LabelField labelField;

	@Override
	public void init() {
		setTitle("My second app");
		setLayout(new BorderLayout());
		setSize(new Dimension(250,200));
		center();

		this.textField = new InputField<String>("");

		this.labelField = new LabelField("What is your name?");

		addField(labelField, BorderLayout.NORTH);
		addField(createEntryPanel(), BorderLayout.SOUTH);
	}

	private GroupControl createEntryPanel() {
		PushButton button = new PushButton("Enter");
		button.setOnClickEvent(new UpdateEvent(UpdateEventId.UPDATE));

		GroupControl entryControl = new GroupControl();
		entryControl.setLayout(new BorderLayout());
		entryControl.addField(this.textField, BorderLayout.CENTER);
		entryControl.addField(button, BorderLayout.EAST);

		return entryControl;
	}
```

```
	/**
	 * Handle the UpdateEvents fired by the button.
	 * @param evt The fired event.
	 */
	public void handleEvent(UpdateEvent evt) {
		String value = (String) this.textField.getValue();

		switch ((UpdateEventId)evt.getId()) {
			case UPDATE:
				System.out.println("update!" + value);
				this.labelField.setValue("Hello " + value + "!");
				break;
			default:
				// unsupported type, propagate event
				propagateEvent(evt);
				break;
		}
	}
}
```

Finally, create a custom event:

```
import nl.coderight.jazz.Event;

public class UpdateEvent extends Event {

	public enum UpdateEventId { UPDATE };

	public UpdateEvent(UpdateEventId eventType) {
		super(eventType);
	}
}
```

You may have noticed the absence of an Action class in the Jazz implementation. Instead, we created a custom Event and implemented a method handleEvent() that has our custom Event as the only parameter. Similar to the actionPerformed() method of an Action, this method is called when an event of this particular type is cast by any field of the view. This method is also conveniently located inside the View where we have direct access to our fields (remember why we were tempted to use Actions as inner classes?).
When we need additional events we can simple expand the Event's enumeration with an additional type and introduce an additional switch case in our handleEvent() method. This way, each action inside your view can have its own event. So you don't have to figure out which component fired the event, because you already know!
And when you *really* find it necessary to differentiate between event types, you can simply create another custom Event class and implement a second handleEvent() method.
By the way, you don't *have* to handle the event in your View, you can also leave that up to its Controller, but we come back to that later on.

Another small point worth noting is the use of GroupControl in this code snippet. This class is used here as an alternative of sorts for the JPanel to resolve a very common layout requirement (grouping two or more components together in a single panel). Jazz has no notion of standard Swing JComponent objects. This (low-level) object is completely abstracted away from view of the developer. For the purpose of grouping components together GroupControl is the Jazz equivalent of the JPanel. However, the real purpose of GroupControl goes way beyond that, but this falls outside the scope of this article.

Data binding
Until now we just toyed around with the View and Controller without the notion of a Model. Let's change this and introduce a Model to tie this GUI to a domain object.

Again, let's start with plain Swing first. First create a simple domain object:

```
public class Person {

	private String name;

	public Person() {
		// hard-coded default value
		this.name = "John Doe";
	}

	public Person(String name) {
		this.name = name;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
}
```

Note: normally, you wouldn't hard-code a default attribute value, of course. This is only done to demonstrate the data binding mechanism later on in this code example.

Next we create a simple Model class that holds a reference to the domain object:

```
public class PersonModel {

	private Person person;

	public PersonModel(Person person) {
		this.person = person;
	}

	public Person getPerson() {
		return person;
	}

	public String getName() {
		return getPerson().getName();
	}

	public void setName(String name) {
		getPerson().setName(name);
	}
}
```

Finally, we modify the plain Swing application to use the Model, which will end up looking something like this:

```
public class StandardSwingApplication {

	private PersonModel model;

	(…)

	public StandardSwingApplication() {
		// hard-coded injection of domain object
		Person person = new Person();

		this.model = new PersonModel(person);
	}

	private void execute() {

		(…)

		this.textField = new JTextField(this.model.getName(), 25);

		(…)
	}

	@SuppressWarnings("serial")
	class SimpleAction extends AbstractAction {

		public void actionPerformed(ActionEvent e) {
			String text = textField.getText();
			System.out.println("action! " + text);
			model.setName(text);
			label.setText("Hello " + text + "!");
		}
	}
}
```

Now do the same thing in Jazz. First create a Model class:

```
import nl.coderight.jazz.form.FormModel;
import demo.jazz.domain.Person;

public class PersonModel extends FormModel {

	private Person person;

	public PersonModel(Person person) {
		this.person = person;
	}

	public Person getPerson() {
		return person;
	}
}
```

Again, this is fairly similar to the way you create a model in plain Swing. The only difference being that we extend from the FormModel base class instead of simply from Object. And we don't bother implementing a getter and setter for the name attribute of Person. (More about this later.)

Now modify the application class the same way we did with the plain Swing version:

```
public class MyApplication extends Application {

	@Override
	public void execute() {
		// hard-coded injection of domain object
		Person person = new Person();

		FormModel model = new PersonModel(person);

		setModel(model);

		(…)
	}
}
```

It looks nearly identical to the plain Swing implementation. There is only one difference: our Application (in the role of Controller) incorporates the notion of a model, so we used the setModel() method to make it aware of our model.

We conclude with the changes to the view. First, the init() method is modified to bind the text field to the name attribute of the domain object in the model:

```
	@Override
	public void init() {

		(…)

		// bind to name attribute of Person domain object
		this.textField = new InputField<String>("person.name");

		(…)
	}
```

The string “person.name” is the **only** requirement to bind the name attribute of the domain object Person to the text field. Jazz uses an Expression Language (EL) similar to the JSP EL, which is a powerful way to drill down an object hierarchy. This saves you from a lot of plumbing to create attribute accessors in your model (remember that we only created a getter for the main domain object Person in the model class?).

Finally, to pass any modifications made to the data back to the model, a call to submit() is added to the update case of the handleEvent() method.

```
	/**
	 * Handle the UpdateEvents fired by the button.
	 * @param evt The fired event.
	 */
	public void handleEvent(UpdateEvent evt) {
		String value = (String) this.textField.getValue();

		switch ((UpdateEventId)evt.getId()) {
			case UPDATE:
				System.out.println("update!" + value);
				this.labelField.setValue("Hello " + value + "!");
				submit();
				break;
			default:
				// unsupported type, propagate event
				propagateEvent(evt);
```

```
                    break;
            }
        }
```

That is all there is to it. This may not look like a big improvement, but please keep in mind that this is a very simple example. When you picture a form that contains not just one field, but instead five or more, than the efficiency gain of a single call to the submit() method to pass **all** modifications back to the model becomes a lot more appealing!

Of course, there remains a common question unanswered in this data binding example; how do you get hold of the domain object in the first place? This is a common misunderstanding of the term 'data binding' when used within the context of a presentation layer. Fetching your domain object falls outside the scope of the presentation layer and therefore is also out-of-scope for Jazz (or Swing, for that matter). Your domain object can live anywhere (a database, an IES, etc.) and it is up to you to fetch it before injecting it into your presentation layer.

Controller

Until now, we implicitly used the Controller that is available inside the Application class. In some cases this is all you really need. But in many cases you require a separate object to control the behaviour of a specific part of your GUI application.

Jazz is constructed to require a Controller for each View of your application. This, again, gently guides you to design and build your application in accordance to the MVC design pattern. As mentioned, the Application is in itself a Controller. More specifically, the Application is the **root** Controller, because all Controllers in your GUI application form a tree hierarchy with the Application as the root. For this purpose, each Controller has a method executeController() to start a new Controller (and its View). The benefit of this hierarchy is that it allows you to easily propagate events up the hierarchy when you don't want to or can't handle an Event in that particular Controller and/or View.

This all may sound a bit daunting, but essentially it is nothing more than a natural extension of the examples you have seen previously. The only difference is that we now repeat the same design pattern (creating a Model, a View, and a Controller) one or more times and nest each of these MVC groupings below the Application in a hierarchy.

This can be demonstrated with a simple application. Since the code is basically a repetition of the previous code samples it is not included in this article, but can be downloaded from the following location: http://www.coderight.nl/download/samples/src-20070415.zip. Only a few important points in this code example are shown here to get you to grips with its inner-workings and implementation details.

The sample application maintains a list of User objects and you can add or remove entries from this list. For this purpose, there are two buttons ("Add" and "Remove") provided in the GUI. When you click on the "Add" button an input dialog window appears. This dialog window is controlled by its own Controller.

As mentioned, you start a new Controller by calling the executeController() method. Therefore, a new handleEvent() method is created in the MyApplication class which includes a method call to executeController() in the add case:

```
public class MyApplication extends Application {
```

```
	(…)

	public void handleEvent(UserEvent evt) {
		switch ((UserEventId)evt.getId()) {
		case ADD:
			executeController(new UserController());
			break;
		case SAVE:
			User user = evt.getUser();
			this.view.append(user);
			break;
		default:
			break;
		}
	}
}
```

This method call starts a custom UserController class which in turn opens the dialog window. From this point on the UserController takes over the control until the dialog window is closed (and the controller is removed from the controller hierarchy) and/or until an event is passed back up the hierarchy.

Let's have a brief look at the handleEvent() of this UserController:

```
public class UserController extends Controller {

	(…)

	public void handleEvent(UserEvent evt) {
		switch ((UserEventId)evt.getId()) {
		case SAVE:
			User user = ((UserModel)getModel()).getUser();
			evt.setUser(user);
			closeView();
			propagateEvent(evt);
			break;
		case CANCEL:
			closeView();
			break;
		default:
			propagateEvent(evt);
			break;
		}
	}
}
```

In this code snippet we can see that a save event triggers three actions. First, the new User value is read from the model and added to the Event. Next, the dialog window is closed. Finally, the event is passed back (propagated) to the parent controller (in this case the MyApplication class). As can be seen from the previous code snippet of the MyApplication class, this User value is then read from the event in the handleEvent() method of the MyApplication class and appended to the list in the main view. This concludes a very simple example of how to interact between two views (parent-detail) where the result of the detail view is passed to the parent view.

Summary

Jazz is a framework based on Event-Driven Architecture and Hierarchical MVC Architecture. It is a much simpler and more efficient alternative over the use of plain Swing to build enterprise-ready GUI applications. Many common features are readily included like providing Data Binding to domain objects.

Simple code examples compared the differences between written in plain Swing to using Jazz. Through this comparison, this article intended to demonstrate the following benefits of Jazz over plain Swing:

- The Jazz API is far smaller than the Swing API which reduces the initial learning curve.
- Jazz offers high-level constructs straight out-of-the-box that incorporate much of the behaviour conventionally required in a GUI application and are easy to use. In comparison, Swing only provides low-level constructs that require a steep learning curve to use and offer less functionality straight out-of-the-box.
- Jazz gently guides the developer to apply best practices when coding GUI applications, like using the MVC design pattern to separate concerns, which produces clean and more maintainable application code.
- Jazz provides increasing productivity gains when a GUI application becomes more complex, making it the ideal framework to build enterprise-ready GUI applications.

This article just scratched the surface of what Jazz has to offer. There are many, more advanced features of Jazz not covered in this article that grant separate discussion, like Multi-Threading (for background processes a.k.a. Tasks), Internationalization (I18N), and (client-side) Validation.

For an in-depth explanation of what Jazz can do for your organization, please contact us at info@coderight.nl.

Keesjan van Bunningen
Consultant

More information at: http://www.coderight.nl/