Meta-JB - Overview

Table of contents

1. What is this crazy thing?

Let's start with the basics. A Java bean is a regular Java class with a method naming convention that makes reflection easier. If your class has an attribute named "foo" that is an integer then to be a bean it will have a `getFoo()` method that returns `int`. It is then said to have a read only bean property named "foo". A writable property would also have a `setFoo(int)` method.

```
public class Bar {
    private int foo;

    public int getFoo() {
        return foo;
    }

    public void setFoo( int foo ) {
        this.foo = foo;
    }
}
```

Table 1: Example Bean:

Java provides a standard set of utility classes for dealing with these beans (`java.beans.*`). These utility classes use reflection to enable an application to access the property values by name without making direct method calls. The problem is that this was not made as straight forward as it sounds.

Java beans seem to have been intended as a way to support drag-and-drop programming. The idea was that you could run some kind of development tool that would allow you to create beans and link their event notifications together and form applications from components. In fact, Java has some classes built into it like `PropertyEditorManager` and `PropertyEditor` implementations that help facilitate this. These classes provide a tightly coupled way of finding GUI components that can be used to edit bean properties. My original framework was based on these classes.

The original framework's goal was to provide GUI generation similar to what these bean frameworks provided, but with a nicer user and programmer experience.

2. Why would I want to generate a UI?

Well, maybe you wouldn't. However, let's consider the alternatives.

Hand-coding -	a highly repetitive practice. A good developer can

	cobble together excellent GUIs this way, but ends up rewriting what is essentially the same code, over and over. And over. This code must also be maintained. Furthermore, if even just the label or type of a field changes, the code must be touched in numerous places to make the change.
GUI Builder -	this simplifies the coding at the expensive of using tool-generated code that may be inefficient. Although, the code generation downsides are minor. This also suffers from some of the same problems as hard-coding. Changes to your object schema can require extensive changes to your UI. Even adding a single new class can require alot of rework in some cases. Any kind of dynamic data types are probably out of the question.
XML Configuration -	this is getting closer to the right idea. Simple changes are simple to do. The problem is that it is still tightly coupling the concept of a UI screen to an actual data object. Dynamic data types are still hard to handle and runtime creation or tailoring of UI components can be problematic.

Table 1: Conventional UI creation methods:

A properly abstracted framework will separate the type descriptions from the means by which they will be displayed. In this way, an object type description can be used in multiple UI formats, for example: a Swing UI, an HTML page, or writing data to an XML file. Or even in multiple forms in the same output type such as a single Swing editor panel for one object or a `JTable` for a list of objects. These are all just different ways of viewing the same data, they ought to be able to rely on the same data descriptions.

3. Why didn't Java PropertyEditors work?

In fact, they did. The problem was that they were bulky, bean-specific, and very tied to the AWT. In order to simply format a property value as a String, it was necessary to instantiate a `PropertyEditor`. A `PropertyEditor` is a fairly heavy-weight object because it takes the value out of the bean and sets up its own change notification and such. Way too expensive just for formatting an object as text.

Also, at the time it seemed like the concept of property descriptions might be applicable to more than just beans. It also seemed like more description information could be included.

This is what the second version of the framework fixed, but it was a bit kludgy and didn't really go far enough in the right direction. Additionally, it ended up being tightly couple in different ways and required extensive custom components to be written for many different otherwise similar data types.

4. Which brings us to today.

We've abstracted away the bean into a generic `MetaObject`. The `MetaObject` provides a `setProperty` method for setting any property value on the object and a `getProperty` method for retrieving any value from the object. It is completely up to a `MetaObject` implementation to decide how these methods actually do their jobs.

```
public interface MetaObject {

    public Object getProperty( String name );

    public void setProperty( String name, Object value );
}
```

Table 1: MetaObject method sub-set:

The list of properties that a meta-object contains can be retreived from the object's `MetaClass`. A `MetaClass` is an implementation-independent description of the properties in an object. This means that the same `MetaClass` can be used for different `MetaObject` implementations. For example, the same class description could apply to a bean wrapper, an SQL result set entry wrapper, or a Hashtable wrapper. As long as they supported the meta-class properties.

A `MetaObject` implementation for interacting with Java beans is included in the core framework.

The main function of the framework is then to associate property types to various ways of viewing or setting those property types. This is done using registries that map property types to objects that perform implementation specific rendering or editing for those types.

The simplest display type is formatting an object value as a String. The following example illustrates string formatting:

```
public class Example {

    // Create a registry mapping property types to format
implementations.
    // A few default format implementations are built-in.
    private static FormatRegistry formatRegistry = new FormatRegistry();

    public static void printProperty( MetaObject metaObject, String
propertyName ) {
```

```
        // Get the type description for the passed property name
        PropertyType type = metaObject.getMetaClass().getPropertyType(
propertyName );

        // Get an object that will format the value for us based on the
        // type of the property
        PropertyFormat format = formatRegistry.getFormat( type );

        // Get the value of the property so that it can be formatted
        Object value = metaObject.getProperty( propertyName );

        // Format the value and print the result
        String text = format.format( value );
        System.out.println( text );
    }
}
```

Table 2: Text formatting example:

This may seem like alot of extra work to go through just to print an object's String representation, especially when all objects already have a `toString()` method. Indeed, the default `PropertyFormat` implementation just calls the `toString()` method on the specified object. The problem is that `toString()` is not always the best way to convert an object to a string. It may also be important to format the object a different way depending on context.

The format registry allows the application to define custom format implementations. For example, a custom format object may convert `Color` objects into a hexidecimal RGB value instead of the debug-friendly output normally expected. Another example might be formatting lists of values into new-line separated strings instead of the comma-separated way that `ArrayList` uses. The nice thing is that the code doing the formatting doesn't need to care about this.

This is the basic idea behind the entire framework. Decouple how the data is displayed from the code that is actually asking the data to be displayed.

5. The Swing Toolkit

One of the view toolkits available is the swing toolkit. This toolkit allows UIs to be generated at runtime for any MetaObject.

```
public class Example {

    // Create a registry containing Swing component factories
    // for creating editors or renderers for property values and
    // meta-objects.  Many default factories are built-in.
    private static FactoryRegistry swingFactories = new
```

```
FactoryRegistry();

    /**
     *  This method returns a UI component for rendering the specified
     *  meta-object property.  If the value of the property changes the
     *  UI component will automatically update its display.
     */
    public static Component getRendererComponent( MetaObject metaObject,
String propertyName ) {

        // Get the type description for the passed property name
        PropertyType type = metaObject.getMetaClass().getPropertyType(
propertyName );

        // Create an object that will keep track of the property value
        // and provide us with a UI component that displays it.
        MetaPropertyRenderer renderer =
swingFactories.createPropertyRenderer( type );

        // Retrieve the property from the meta-object as a mutable
single-valued
        // object that will automatically update the property in the
meta-object.
        PropertyMutator mutator = metaObject.getPropertyMutator(
propertyName );

        // Associate the value with the renderer
        renderer.setPropertyMutator( mutator );

        // Return the display component
        return renderer.getUIComponent();
    }

}
```

Table 1: Simple Swing toolkit example:

The above example will return a component that will display a single property value. If the property value changes then the display will also change. The calling code doesn't need to worry about how the value is displayed. The application can customize this behavior or rely on the built-in implementations. A property containing a `Color` may be rendered as a single block of color while a property containing a `Double` may render as a label displaying the double value formatted to two decimal places. Even a property containing another MetaObject can be rendered. In fact, convenience methods are provided for rendering a single meta-object this way.

```
public class Example {

    // Create a registry containing Swing component factories
    // for creating editors or renderers for property values and
    // meta-objects.  Many default factories are built-in.
    private static FactoryRegistry swingFactories = new
```

```
FactoryRegistry();

    /**
     *  This method returns a UI component for rendering the specified
     *  meta-object property.  If the value of the property changes the
     *  UI component will automatically update its display.
     */
    public static Component getMetaObjectComponent( MetaObject metaObject
) {

        // Get the appropriate meta-object renderer based on the
        // meta-class.
        MetaClass metaClass = metaObject.getMetaClass();
        MetaObjectUI renderer = swingFactories.createMetaObjectRenderer(
metaClass );

        // Tell the renderer which object it is viewing
        renderer.setMetaObject( metaObject );

        // Return the display component
        return renderer.getUIComponent();
    }

}
```

Table 2: Swing toolkit MetaObject component example:

The examples above can easily be changed to create editor components instead. Also, the above examples do things in a very straight forward way which is not necessarily the best way for every application. For example, it is not necessary to create a new MetaObject renderer/editor every time if the application is only displaying one component at a time anyway. A MetaObjectUI instance can be reused to display a different meta-object of the same meta-class in the same previously returned UI component.

6. Conclusion

Meta-JB provides a generic bean-like abstraction that can be adapted to any object implementation that has the concept of fields and values. Upon this framework, toolkits are built to generate different types of views without being coupled to the actual data implementation.

For additional working examples, download the examples distibution.