5-2010

Semantic Annotation for Java

Douglas A. Lyon
Fairfield University, dlyon@fairfield.edu

Published Citation

Douglas Lyon, "Semantic Annotation for Java", Journal of Object Technology, Volume 9, no. 3 (May 2010), pp. 19-29

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©JOT, 2010

Vol. 9, No. 3, May-June 2010

Semantic Annotation for Java

By Douglas Lyon

ABSTRACT

This paper describes how to use annotation to provide semantic information. The applications include the automatic construction of a GUI so that the user input is constrained to a correct range (guarding the input). Elementary GUI examples include elements for a variety of data types. Given sufficient effort, we envision the annotation technique as a replacement for XML and BeanInfo-based systems.

We show that semantic annotation has several intrinsic benefits, geographically co-locating constraints with their parameters eases maintenance, enables compile-time checking, and establishes consistent constraint declarations. Semantic annotation is shown to be GUI agnostic (devoid of Swing, SWT, AWT or HTML considerations). We also show how GUI annotation can be considered dangerous. The automatically constructed GUI's are simple, and of no intellectual merit, other than their construction technique.

Categories and Subject Descriptors

D.2.3 [**Design Tools and Techniques**]: Semantics and Features – *assertion checkers, computer-human interface, programming by contract, specification.*

General Terms

Documentation, Human Factors, Languages, Reliability Verification.

Keywords

Java, assertions, semantics, GUI construction, automatic code synthesis, reflection.

1 INTRODUCTION

Annotations enable the encoding of metadata in Java [1]. They are syntactically checked by the compiler, at runtime and may be applied to classes, methods, variables, parameters and packages. This paper shows how to use annotations to provide semantic information about the use of parameters and methods. The semantic information is used to automatically construct a GUI for a series of operations. Several examples are shown for an on-going development project involving image processing [2].

2 THE PROBLEM

Typically, programmers create code that lacks any semantic guidance about its use. Consider the following code fragment:

```
public static double f(double x)…
```

What is the range on *x* or the range on the return? Without any guidance about how to properly use the function there is no way to tell. A good programmer might document the function, perhaps giving explicit instruction on use, however, these instructions are not machine-readable and are typically given in an ad-hoc manner. The GUI programmer incorporates important business logic about the semantics into the GUI, by guarding the user input. For example, if the input must be a number that lies between zero and one, a slider may be used that is constrained to that range. This brings to mind several problems:

1. The GUI now contains business logic.
2. The documentation has no compile-time check.
3. Maintenance of the function is complex.
4. GUI construction is tedious and error-prone.
5. There is no semantic guidance.

3 SEMANTIC ANNOTATION IN ACTION

This section provides several examples of semantic annotation used to describe the use of inputs and the display of outputs. The primary focus is on creating a useful set of annotations for an image processing application, currently under development. However, the examples can be applied in almost any area.

Boolean Data

A Boolean variable can take on the values of *false* and *true*. A Boolean variable also has a default value and a display name. We define an annotation interface for Boolean variables using:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface BooleanRange {
    public boolean getValue();
    public String getName();
}
```

The retention policy indicates that the annotation will be retained in the byte codes output by the Java compiler. The *getValue* method returns the default value for the Boolean variable, and the *getName* returns the display name. An example of code that makes use of the *BooleanRange* annotation follows:

```
@BooleanRange(
            getValue = true,
            getName = "isVisible"
    )
    public void setVisible(boolean visible) {
        this.visible = visible;
    }
```

Figure 1. The Checkbox

The check box of Figure 1 was generated automatically in response to the Boolean range annotation.

Numeric Data

A variable typically has a range that can be represented as a minimum, a maximum, a default value and an increment. For example:

```
public @interface FloatRange {
    public abstract float getValue();
    public abstract float getMin();
    public abstract float getMax();
    public abstract float getIncrement();
    public abstract String getName();
}
```

There are two kinds of numeric primitive data types in Java, fixed-point and floating-point. There are two kinds of floating-point data types in Java, *double* and *float*. There are four kinds of fixed-point data types, *byte, short, int* and *long*. Thus, there are six numeric primitive data types in Java, and six numeric range annotations.

Figure 2. Float Range

Figure 2 shows the automatically constructed GUI from the annotation given by

```
@FloatRange(
            getValue = 10,
            getMin = 1,
            getMax = 100,
            getName = "x",
            getIncrement = 0.01f
    )
    public void setX(float x) {
        this.x = x;
    }
```

4 GUI ANNOTATION

Sometimes the intention for the use of a variable is to enable it to be controlled at a variety of precisions. In such a case, the slider may be a good course-grained

controller, but a spinner is better at fine control. In such a case, the semantic annotation is supplemented with GUI annotation. Consider:

```
@DoubleRange(
            getValue = 0,
            getMin = -Math.E,
            getMax = Math.E,
            getName = "E",
            getIncrement = 0.01
    )
    @SpinnerProperties(
            isVisible = true,
            isCompact = false,
            isIncrementHidden = false
    )
    public void setE(double e) {
        this.e = e;
    }
```

The *SpinnerProperties* GUI annotation provides guidance on how to build a spinner. The specification for the *SpinnerProperties* annotation follows:

```
public @interface SpinnerProperties {
    public boolean isVisible();
    public boolean isCompact();
    public boolean isIncrementHidden();
}
```

Figure 3. Hybrid Spinner Slider

Figure 3. shows a hybrid spinner slider constructed from the *DoubleRange* and *SpinnerProperties* annotation. The top-most spinner shows the value and the bottom-most spinner shows the increment. Using the up and down arrows, the spinner is incremented or decremented. These are linked to the up and down arrow keys. The *home* and *end* keys set the value to the minimum and maximum.

5 COMPONENT COLOR

Annotation is unable to represent general reference data types. For example, the *Color* class is typically used to define color. However, the limits of annotation prevent the use of *color* reference data types, as a result, we use a string representation of color. This enables the use of color names, as well as hexadecimal presentations:

```
public @interface Colors {
    String getForeground();
    String getBackground();
}
```

The color is specified using a 24 bit hexadecimal string, in RGB order:

```
@DoubleRange(
            getValue = 0,
            getMin = -1.414,
            getMax = 1.414,
            getName = "Math.Sqrt",
            getIncrement = 0.1

    )
     @Colors(
            getForeground = "0xFF0000",
            getBackground = "0xFF0000"
    )
    @SpinnerProperties(
            isVisible = true,
            isCompact = false,
            isIncrementHidden = false
    )
    public void setZ(double z) {
        this.z = z;
    }
```

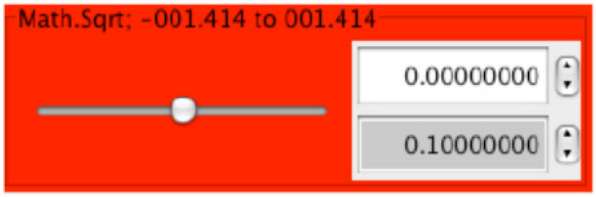


Figure 4. Color

Figure 4 shows a red hybrid slider spinner, automatically constructed with the annotation. Given the lack of reference data types, the use of GUI annotation appears impoverished as a means of representing GUIs.

6 SEMANTICS FOR REFERENCE DATA TYPES

There are several basic reference data types that have been implemented with our semantic annotation system. The effort has been directed toward those most useful for our image processing application (Strings, Images, Rectangles and arrays of primitive data types). While not a complete list of data types (by any means), we have been able to achieve a proof-of-concept prototype that demonstrates the generality of the semantic annotation.

Range Array

The range array consists of a range of numbers, each of which is to be manipulated by the user using a consistent range but a different label. For example:

```
@SpinnerProperties(
            isVisible = true,
            isCompact = false,
            isIncrementHidden = true
```

```
)
  @Colors(
        getForeground = "0x00FF00",
        getBackground = "0x8a2be2"
)
@RangeArray(
        getMin = 0,
        getMax = 1,
        getValue = 0,
        getNames = {
                "redMin", "redMax",
                "blueMin", "blueMax",
                "greenMin", "greenMax"
                },
        getIncrement = 0.01
)
public void setCornerPoints(double[] cornerPoints) {
    this.cornerPoints = cornerPoints;
}
```

Figure 5. Arrays of Numbers

Figure 5 shows an array of numbers. Arrays of numbers are grouped with other arrays of numbers in a tabbed pane, for the purpose of organizing the layout.

String properties

There is no annotation needed for string property setters:

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

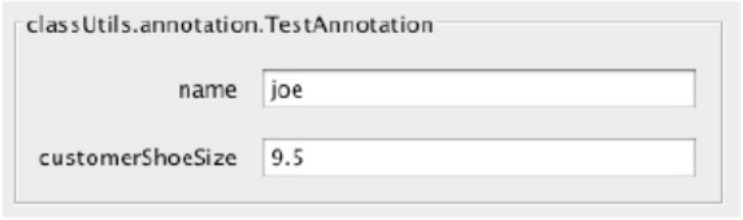


Figure 6. String Properties

Figure 6 shows the strings for the name and the customer shoe size. In this particular case, if the shoe size is a number, then the string should be checked (or a numeric data type should be used). However, shoe sizes can have both letters and numbers (10W, for size 10, wide).

7 IMAGE PROCESSING APPLICATIONS

Figure 7 shows a texture control panel, automatically created, using semantic annotation. The panel was automatically constructed using a tabbed pane, in order to organize the parameters.

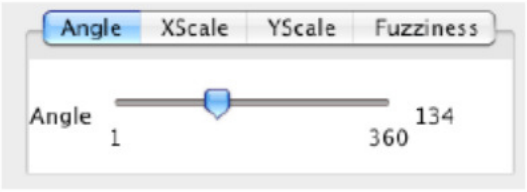


Figure 7. Texture Control Panel

Figure 8 shows the texture output by the control panel.

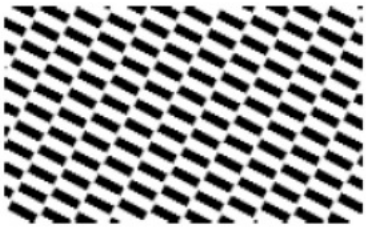


Figure 8. A Texture

Figure 9 shows the output of an iterative affine transform as described in [2].

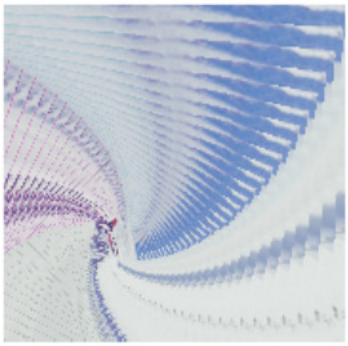


Figure 9. Affine Transform

The processing time can cause latency in the controls. As a result, the control panel has an option for continuous update, during slider manipulation.

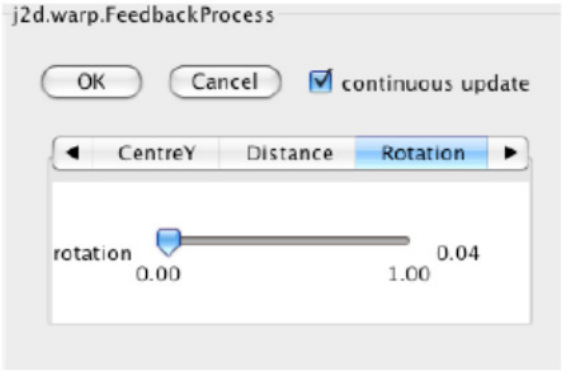
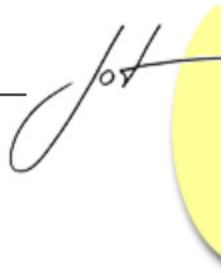


Figure 10. Affine Control Panel

This enables setting of before they take effect. To simplify the design of the image processing framework, I require that all operators implement the *ImageProcessorInterface*:

```
public interface ImageProcessorInterface extends Serializable  {
    /**
     * a basic garbage in, garbage out processor.
     * @param image input image
     * @return output image
     */
    public Image process(Image image);
}
```

My rule on interface creation is that interfaces should be as simple as possible (if not, simpler!). There are over 235 uses of the *ImageProcessorInterface* and this is an excellent example of a decision that can shape a program for years to come. We shape our tools and thereafter, our tools shape us! The polymorphic nature of the interface enables automatic invocation of operations, and semantic annotation enables automatic construction of their GUIs. Thus freeing the image processing programmer to worry only about how to implement the interface and supply proper semantic annotations.



8 ALTERNATIVES TO SEMANTIC ANNOTATION

There are several other techniques available that enable the automatic construction of GUI elements. For example, xml can be used to enable plug-in frameworks that are popular with image processing programs. The following xml comes from a blur transition filter [private communication with Jerry Huxtable].

```
<filters>
    <filter class='com.jhlabs.image.BlurTransition'
group='Stylize'>
        <param property='transition' name='Transition'
type='percentage'/>
        <param property='HRadius' name='H Radius' type='int'
min='0' max='300'/>
        <param property='VRadius' name='V Radius' type='int'
min='0' max='300'/>
        <param property='destination' name='Destination'
type='image'/>
    </filter>
</filters>
```

This approach requires that the xml document be maintained and validated at run time. Refactoring tools may not be able to update the fully-qualified class name embedded in the class property of the xml document. Worse, a typo in the class name will cause a class not found exception to be thrown, at run time.

The excellent image-processing program from the NIH, called *ImageJ* makes use of a plug-in architecture [Private communication with Wayne Rasband]. The plug-in author must implement a *PlugIn* (as in our system). Additionally, the plug-in author provides arguments in a file called *IJ_Props.txt* and a *plugins.config* file. Thus, non-Java files must be maintained and can be a fruitful source of run-time errors.

Another plug-in architecture is used by *gimp* [3]. In this one, functions are used (it is written in C) to return information about where, in the menu system, the menu item for the plug in will be placed. This type of guidance is more about GUI annotation than semantic annotation.

The Component Software Development (CSD) system typically used by Java is called the Java bean. Bean frameworks require a lot of maintenance and support. Property change listeners, property change events, custom editors, etc., become large and cumbersome frameworks, creating a code-base that dwarfs the business logic. Distracting programmers with a series of non-business logic development activities is a liability that the busy development house can ill-afford. Worse, there is no attempt to move the constraints on parameters into the business logic in the Java bean (these things are located in different, *info* classes) [4].

There is an XML wrapper for Swing available, called *SwiXML* [5]. To create a panel, with a border layout, use:

```
<panel id="borderPanel10" layout="BorderLayout">
 <button constraints="BorderLayout.NORTH">1</button>
      <button constraints="BorderLayout.EAST">2</button>
      <button constraints="BorderLayout.SOUTH">3</button>
      <button constraints="BorderLayout.WEST">4</button>
      <button constraints="CENTER">5</button>
```

```
</panel>
```

The resultant xml looks as complex as any Java code and suffers from an inability to generate a compile-time error.

The design by contract work, as implemented in JML, is inspiring [6]. It comes closest to a semantic annotation, but makes use of JavaDoc comments. The drawback is that JavaDoc comments are ignored by the compiler, whereas annotations are not. Further, annotations are available, at run-time, and do not require semi-automatic static parsing of source code for the generation of compile-time checks. The JML project was started before the annotation extension to the Java language was available.

9 CONCLUSION

Using annotation for semantics enables business logic to be geographically co-located with the constraints on use, obviating the need to propagate constraint information into the GUI. Semantic annotation also enables compile-time checking, simplifies maintenance and helps to automate the construction of the GUI. Finally, the use of annotation to implement semantics gives a consistent means for establishing the use of programs, enabling the guarding of the input and thus helping to prevent incorrect usage.

Semantic annotation is GUI agnostic. The annotation can be used to create GUI's based in AWT, Swing, SWT, HTML, etc. In fact, the automation of any or all of these GUI constructions is not only possible, but also desirable, since maintenance of manually constructed GUIs is both tedious and error-prone. For example, the HTML synthesizer code will not only look different from Swing, but it may be written by a different programmer on the team than either the progenitor of the business logic or of the HTML GUI.

Semantic annotation can have a small footprint. Compilers can turn off the generation of the semantic annotation incorporation into the byte code generation, as an option. Thus, no extra space need be taken, if code space-optimization is important.

Semantic annotation reduces the amount of code that needs to be written, since extra GUI code is synthesized automatically. This facilitates the creation of visual programming languages and keeps property editors from having to propagate at a rate equal to the number of operators in the system.

Semantic annotation enables consistency in GUI construction. Consistency can have both positive and negative side effects. On the positive side, consistency enables a style rule to be implemented and followed in an automatic fashion. On the negative side, all the GUI elements tend to look alike. As more and more powerful annotations migrate toward the business logic, the temptation is to make the annotation resemble a GUI programming language written as a data structure. Once this occurs, the annotation becomes more like GUI annotation, rather than semantic annotation. The question of how GUI annotation can (or even if) it should be used as a means of enhancing GUI construction, remains open.

REFERENCES

[1] "JDK 5.0 Developer's Guide: Annotations". Sun Microsystems. 2007-12-18. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html. Last accessed 2009-04-15.

[2] Lyon, D. 1999. *Image Processing in Java,* Prentice Hall, Upper Saddle River, NJ.

[3] http://developer.gimp.org/writing-a-plug-in/1/index.html. Last accessed 2009-04-21.

[4] Lyon, D. 2003. *Java for Programmers,* Prentice Hall, Upper Saddle River, NJ.

[5] http://www.swixml.org/. Last accessed 2009-04-22.

[6] http://www.eecs.ucf.edu/~leavens/JML/documentation.shtml. Last accessed 2009-04-23.

About the author

Douglas A. Lyon is the co-director of the Electrical and Computer Engineering program at Fairfield University, in Fairfield CT, a senior member of the IEEE, President of DocJava, Inc. and CTO of Lyon-Ratafia.

He received the Ph.D., M.S. and B.S.degrees in computer and systems engineering from Rensselaer Polytechnic Institute (1991, 1985 and 1983). Dr. Lyon has worked at AT&T Bell Laboratories and the Jet Propulsion Laboratory.

Dr. Lyon has authored or co-authored three books (Java, Digital Signal Processing, Image Processing in Java and Java for Programmers). He has authored over 45 journal publications. Email: lyon@docjava.com. Web: http://www.DocJava.com.