JHOVE2 and Java Reflection

Or, "By indirection find directions out"

"We're gonna need a bigger boat ..."

We have committed to a number of requirements that will be a challenge to meet using everyday Java coding techniques:

- We have promised a very high level of user specification and control over
  - the depth of parsing
  - the choice of enumeration or tally of features
  - the granularity and inclusiveness of conformance checking
  - the scope and granularity of assessment reporting
  - the modification, extension, or replacement of assessment rules

- We have promised modularity of design, enabling users of the various components of the JHOVE2 application to configure, compose, and use these components in ways best adapted to their own workflows. This includes
  - invocation of the format modules separately from the JHOVE2 backplane
  - “plug-ability” of rules engine

- We have committed to the use of consistent, intelligible design and coding idioms to make it easy to meet the universally expressed need for more format modules than will be delivered by this project. The choice of such idioms must enable
  - rapid development of new format modules that will provide the same level of user specification and control as listed above
  - rapid development of new format modules whose organization and structure “reads” consistently with JHOVE2-project-developed modules
  - rapid development of new format modules whose behavior is consistent with the JHOVE2 domain model of such key terms as “format”, “validity”, and “assessment”
  - rapid development of new format modules that “wrap” existing format parsers and other tools, and provide a JHOVE2 facade around those existing tools in order to perform feature extraction, and also, where possible, validation and assessment
  - ease of extensibility of existing format modules as new versions/profiles of a format family are developed

- We have committed to documentation artifacts that include
  - developer guide, including new format module development guide.
  - data dictionary for JHOVE2-specific schemas
  - development artifacts, such as our format specification template, which might well comprise a component of the developer guide and of the template for new module development
And, of course, we will be delivering what amounts to a basic workflow framework that can operate over arbitrarily deep levels of nesting of reportable objects. And all of this is promised to be more efficient than the current JHOVE.

We’re going to need some tools, and some heuristics.

First, the tools: the Java reflection API, and Java Annotations.

"Mirror, mirror…"

What is the Java reflection API? It’s a mature API, available since Java 1.1 and adapted to handle generics since Java 1.5. It is essential to the working of debuggers and class browsers. It is the basis of JavaBeans component technology (plain old JavaBeans, not EJB). It’s how Spring works its inversion-of-control, dependency-injection magic. It provides the underpinning of Maven’s plug-in architecture. And it’s how the Drools rules engine (which, like Spring, complements reflection with aspect-oriented techniques) cranks along, firing off rules to be applied to data.

What is reflection? As we were all taught, an object encapsulates data and behavior. Reflection is the ability of an object to examine its data-containing structures and its methods ("introspection"), and to modify either or both at runtime ("intercession").[1,2] Note that this means more than just access to the contents of a field (we have that already using everyday Java). It means we have access to the attributes which comprise the definition of a field or method. The package java.lang.reflect, in its own words,

> Provides classes and interfaces for obtaining reflective information about classes and objects. Reflection allows programmatic access to information about the fields, methods and constructors of loaded classes, and the use of reflected fields, methods, and constructors to operate on their underlying counterparts, within security restrictions.

The take-away here is something we sometimes forget: code doesn’t just operate on data; it is data. With the Java reflection API, not only do we have access to an object’s data, we also have access to its metadata. And reflection enables all sorts of run-time binding and proxy capabilities otherwise unavailable to Java programs.

Sun has provided a tutorial[3] on the reflection API, which you might want to skim through before proceeding through the demo code. For our purposes, the 3 reflection classes we want to focus on are java.lang.Class (which implements interface java.lang.reflect.Type), java.lang.reflect.Field, and java.lang.reflect.Method A quick look at the JavaDocs for each of the classes gives an idea of what information about a class (and an instance of a class) is available at run-time:

- With java.lang.Class, we can find out what fields and methods are members of a class, any inner classes defined by a class, what a class's superclass is, what interfaces are implemented by a class, whether or not a class is an instance of either an array or an interface, any type variables associated with a class declaration, and what annotations (more on these later) are associated with a class.

- With java.lang.reflect.Field, we can determine the type of a field, get or set its value in the underlying object containing a field, and retrieve any annotations associated with a field.
With `java.lang.reflect.Method`, we can determine the number and type of parameters used by a method when it is invoked, a method’s return type, and any annotations associated with a method. We can also invoke a method on a specified object, using specified parameters.

An `Annotation` (`java.lang.annotation.*`) is a special kind of interface that gives us the ability to add our own metadata to Java classes (including interfaces), fields, and methods. (Sun has a very brief tutorial/description [4] of annotations. O’Reilly provides somewhat longer overview [5], and IBM DeveloperWorks has yet another [6].) We can choose to make these metadata visible at source-time, compile-time or run-time. Annotations can inherit, and they can be detected by JavaDoc and other documentation tools. Annotations can have members whose names are specified in the Annotation type declaration (much like an interface declaration), and whose values can be set when the annotation is applied to a class, method, or field.

You will likely have seen some of the Java 1.5 standard compile-time annotations injected by Eclipse into your source code. For example, in the UTF8 demo class, you will see the `@Override` annotation applied to the UTF8.parse() method, indicating that this method overrides the `parse()` method in the J Parsable interface.

```
@override
public long parse(JState state, JReportable parent, JInputable input)
{
    return this.parse(state, parent, input, true, false);
}
```

Notice that it is not necessary to declare that the UTF8 class implements the `@Override` annotation type in the UTF8 class declaration. It is sufficient to decorate the `parse()` method declaration with the `@Override` annotation itself. Also, by itself, an annotation doesn’t do anything. It’s a lot like a processing instruction in an XML document. Nothing happens unless an application directs its attention to the annotation, and does something based on its presence or absence, and the values of its text “pseudo-attributes”. In the example above, the `@Override` annotation will cause Eclipse to indicate a syntax error if you attempt to modify the parse method signature to anything that is not declared in one the interfaces or superclasses of the UTF8 class.

Let’s see how the demo code puts reflection and annotation to use.

"It’s show time!"

The demo application’s “business” classes are groceries: BreadLoaf, BreadSlice, Jelly, PeanutButter, Sandwich, JunkFood, LowFatJunkFood, and PaperGoods, all of which implement `IgroceryStorePurchasable`. An instance of any one of them can end up in the `GroceryBag.contents ArrayList`. Each of them is about as plain as a POJO – or a JavaBean – can be.

Let’s look at the PeanutButter class. This is a class with just one field (`consistency`), no-arg and 1-arg constructors, and an accessor and a mutator for that single field.

---

*The demo project has been committed to the JHOVE2 SVN repository. The project name is reflection-annotations-demo. Please see the README.txt file in the base directory for directions on how to modify the demo.properties file before building and running the application.*
What if PeanutButter were an object of interest to the JHOVE2 community? Suppose we wanted to capture the technical metadata about a particular PeanutButter instance. Leaving aside the parse() (shop()? chew()??) method for now, what else would we need to create a JHOVE2 PeanutButter module?

1. Section 8 of our format specification template requires us to fill in a table containing every feature of the class: its name, its (unique) identifier, a description of the property's semantics, and, possibly, a reference to the authoritative specification(s). This information must always be consistent with what actually exists in the implementing class, or we're headed for specification-rot.

2. We have said we will make a feature's name (which is not guaranteed to be unique or immutable from user to user) both internationalizable and user-configurable. So we need a way (create a properties file that can be filled in, or create a CSV file that can be imported into a spreadsheet and edited, or use the “JavaBean-ness” of PeanutButter to create a visual "property sheet" for user editing, etc., etc.) to expose the list of PeanutButter features to user configuration of those feature names.

3. Once the user configures the feature names and specifies a locale, we need a feedback loop to associate that information with the PeanutButter class whenever we report on a PeanutButter object (round-trip between code and user configuration).

4. Some users may not be the least bit interested in PeanutButter of whatever consistency. We have to provide a way to enable them to turn off reporting of any PeanutButter objects we encounter during recursive descent. In fact, we have to be able to turn off even looking at a
5. When we do report on the PeanutButter object:

- We have to ensure consistency between the identifier in the reported object and the identifier in the specification.
- We don’t report on the object at all, if the user doesn't want to hear about it.
- If the user does want to hear about it, we have to report the name the user chose for it.
- Besides the identifier and the name, we need to report the feature's type, and the feature's "arity".
- We might need more than one way to report the object (as XML, as property=value pairs, as a hierarchy, as a flat list, etc.).
- We might need control over when we report the object (immediately upon parse, top-down, bottom-up, when we're running low on memory, when we reach certain level of recursion in our backplane, etc.)
- Conceivably, some JHOVE2 users might want to extend the list of reported feature characteristics to include the semantic description and authoritative specification reference included in the JHOVE2 PeanutButter specification document.

6. We have to populate yet another user interface with the identifiers and (user-specified) names to enable users to configure acceptability rules. These too must be capable of round-tripping between the exposure of the class features, the rules engine, user configuration information, and, depending on our implementation choice, reporting of validity results (which also must be consistent in its use of the identifiers).

7. We might want to use the same engine for validation that we do for acceptance checking, so that's another configuration round-trip.

8. A user may take us up on our plug-and-play modularity commitment, and decide to swap in a different assessment engine (Drools for Schematron, or Schematron for Drools, or whatever the user chooses for whatever we actually provide out of the box). So our solution has to be flexible enough to make the generation or adaptation of assessment configuration also at least reasonably plug-and-playable.

9. If we have ISandwichFilling profiles (Vegetarian, Hypoallergenic, FiftiesLunchBoxClassic) we wish to test for and report, this is yet another configuration capability to expose and to round-trip.

10. If we ever modify the PeanutButter class definition and add, say, a totalGramsOfFatPerOunce field, then we have to (consistently!! round-trip-ably!! infinitive-split-ably!!) implement items 1 through 9 for this feature.

11. If the user does want to hear about consistency, but doesn’t want to hear about totalGramsOfFatPerOunce, then we have to make it possible for the user to indicate that choice, and then, again, round-trip that information to the run-time code, rules engine, etc.
All this for one class, with one feature. We – and anyone writing or extending a JHOVE2 format module down the road – will have to do this for all reportable features, for all reportable classes.

“This is going to take a lot of work.”

Or not.

Notice the `@ClassReportable` annotation at the beginning of the `PeanutButter` class definition. This annotation references the `org.jhove2.reflection_demo.annotations.ClassReportable` annotation type. Let's take a look at that type declaration:

```java
package org.jhove2.reflection_demo.annotations;

import java.lang.annotation.*;
/**
 * Annotation to indicate all members of a class are reportable.
 * By default, all fields of a @ClassReportable class are reportable
 * To switch off reportability of a field, @see
 * org.jhove2.reflection_demo.annotations.@FieldNotReportable
 */
@Documented
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface ClassReportable {}
```

This declaration is, if anything, more stripped down than one of the demo POJO business classes. Its body is completely empty. All that really elaborates it are the Java-standard annotations which decorate the declaration itself:

- `@Documented` means that this annotation is exposed to documentation tools such as JavaDoc
- `@Target(ElementType.TYPE)` means that this annotation will decorate classes (rather than fields or methods)
- `@Retention(RetentionPolicy.RUNTIME)` means that this annotation will be available (via the Java reflection API) at runtime

As the JavaDoc comment suggests, the intention of applying `@ClassReportable` to a class is to indicate that objects of that class are considered “reportable” (whatever that means), and that any field (public, protected, private) of that class is also considered "reportable". Note that, if we prefer, we could choose a different level of granularity for this annotation. We could declare its target to be a field (`@Target(ElementType.FIELD)`), and then attach the annotation to any field in any class. In this demo, we'll assume that the most frequent case is the need to report all fields of a class of interest. We'll “switch off” any exceptional, non-reportable field using the `@FieldNotReportable` annotation.

Note also that by simply declaring an annotation type, and annotating a class with it, we haven't really made anything happen yet – for good or for ill. An annotation is very non-intrusive. We haven't altered or extended the behaviors of our POJO class. By itself, the annotation doesn't add any functionality the way, for example, a concrete method in an abstract class might do. Nor does it encumber the class with the requirement to implement any method signatures. It's just what we
ordinarily mean by the word “annotation” – a note, an indication, a piece of information about which we can choose to do something, or nothing.

But of course we do want to do things – a lot of things – a list of at least 11 things – in both the classes we write, and the classes we and others will be wrapping in some way to extend existing tools into JHOVE2 format modules. We just don't want to have to write code (or the configuration files, or the user-input spreadsheets, or...) to do those same 11 things again and again for every class and every feature of every class.

When we think about how the JHOVE2 application will be used, we can distribute most of the activities involved in meeting the needs above into 3 different phases:

1. User configuration template creation: This refers to the generation of templates (text files, xml files, input screens, forms, whatever we choose) for users to enter configuration information (what objects to report, what fields in those objects to report, what name in what language to use when reporting, what rules to apply to which fields of which objects, whether or not to even parse objects of a certain type, and, if the user is a developer, information to populate documentation)

2. Run-time application configuration: We have to apply the information that users place in the templates we create

3. Run-time processing and reporting: We have to report on run-time instances of reportable classes, consistent with user specification, in different formats

And we want to “Write Once, Reuse Everywhere”. Any module from PDF to PeanutButter should be able use the same code base to do the same work, regardless of the differences in content models.

Let's take this a phase at a time, and see how reflection can make this all this possible. The demo project has a class -- org.jhove2.reflection_demo.features.FeatureUtilities -- which consists of static utility methods. The main method of the Demo class exercises these static methods in turn.

1. User configuration template creation

First, let's determine how many of the classes in the reflections-annotation-demo project are “reportable” – i.e., have been decorated with the @ClassReportable annotation. The FeatureUtilities.getAllClassNamesInProject() method returns a list all 34 classes (including enumerations) declared in the project, by the simple mechanism of counting the number of files with a .class extension in the target subdirectories (no reflection here). However, when we run FeatureUtilities.getReportableClassNamesInProject(), we get this list of 10 classes:

org.jhove2.reflection_demo.groceries.GroceryBag
org.jhove2.reflection_demo.groceries.IGroceryStorePurchasable
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf
org.jhove2.reflection_demo.groceries.food.bread.BreadSlice
org.jhove2.reflection_demo.groceries.food.sandwichfillings.ISandwichFilling
org.jhove2.reflection_demo.groceries.food.sandwichfillings.Jelly
org.jhove2.reflection_demo.groceries.food.sandwichfillings.PeanutButter
org.jhove2.reflection_demo.groceries.food.sandwichfillings.Sandwich
org.jhove2.reflection_demo.groceries.sundries.JunkFood
Note that these are classes that are defined as being reportable at compile time (or, more accurately – at programmer-coding time). This is before we have received any user configuration information. Another way to look at this is as the list of all possibly reportable classes. Notice also that this list does not include one of the classes that implements IgroceryStorePurchasable. The org.jhove2.reflection_demo.groceries.sundries.PaperGoods class was not annotated with the @ClassReportable annotation, and so is not included as a “reportable” project class (This is a completely arbitrary decision, intended to show how the annotation works in conjunction with the class declaration and the run-time use of reflection).

Let’s look at the code for this method:

```java
public static ArrayList<String> getReportableClassNamesInProject() throws Exception {
    ArrayList<String> classNames = getAllClassNamesInProject();
    ArrayList<String> reportableClassNames = new ArrayList<String>();

    Class<ClassReportable> ClassReportableClass = (Class<ClassReportable>) Class.forName("org.jhove2.reflection_demo.annotations.ClassReportable");

    for(String projectClassName:classNames){
        Class projectClass = Class.forName(projectClassName);
        if (projectClass.getAnnotation(ClassReportableClass)!=null){
            reportableClassNames.add(projectClassName);
        }
    }
    return reportableClassNames;
}
```

The method uses the static `java.lang.Class.forName(String className)` reflection method to create the `Class` object that defines the `ClassReportable` annotation we used to mark reportable classes. Then, for each .class file in the target subdirectories, the method

- creates the Class object which defines the particular demo class, again using `Class.forName`
- uses the `java.lang.Class.getAnnotation(Class<A> annotationClass)` reflection method to see if the project `Class` contains a `@ClassReportable` annotation in its metadata (we use the `Class` object for the annotation which we created in the first step as the parameter to this method)
- if the project `Class` does have this annotation, adds the class name to the list of reportable classes in the project

Now that we know we can obtain a list of reportable classes, let’s see how to obtain the list of all possibly reportable features of these classes. We have a utility method to inspect a `Class` object and determine all its declared fields and the declared fields of all its superclasses:
protected static Field[] getClassAndInheritedFields(Class objectClass) throws Exception {
    ArrayList<Field> fields = new ArrayList<Field>();
    Field[] objectFields = objectClass.getDeclaredFields();
    for (Field objectField:objectFields){
        fields.add(objectField);
    }
    Class superClass = objectClass.getSuperclass();
    if (superClass != null){
        objectFields = getClassAndInheritedFields(superClass);
        for (Field objectField:objectFields){
            fields.add(objectField);
        }
    }
    Field[] fieldArray = new Field[fields.size()];
    fieldArray = (Field[])fields.toArray(fieldArray);
    return fieldArray;
}

We use the `java.lang.Class.getDeclaredFields()` method to return an array containing the `Field` objects of a class. We use the `java.lang.Class.getSuperClass()` method recursively to ascend the inheritance hierarchy of the class and get all inherited fields as well.

Once we have the array of all the fields of a reportable class, we can add them to our list of reportable features. But first, we want to make sure the programmer has not declared a field of a reportable class to be non-reportable. Look at the class declaration for `Jelly`:

```java
@ClassReportable
public class Jelly implements IGroceryStorePurchasable, ISandwichFilling {
    public enum Consistency {
        preserves,
        clear
    }
    protected String flavor;
    protected Consistency consistency;
    @FieldNotReportable
    protected int calories;
    ...(constructors, accessors and mutators here)
}
```

Since the class declaration has been decorated with the `@ClassReportable` annotation, by default all three fields (`flavor`, `consistency`, `calories`) would ordinarily be automatically considered reportable. However, in flight from reality, we have decided not to report the calorie count. So we have decorated the `calories` field with the `@FieldNotReportable` annotation. Our utility class assembles the list of fields, checks each one to see if it's reportable (i.e. does NOT have a `@FieldNotReportable` annotation), invokes other reflective utility methods to determine the type and
arity and type of the feature, and checks for yet another annotation (@FeatureDescribable) to link the feature to an authoritative specification. Here’s the code:

```java
public static ArrayList<FeatureDescription> getClassReportableFeatureDescriptions
  (Class objectClass) throws Exception{
  //only report reportable classes
  Class<ClassReportable> ClassReportableClass =
    (Class<ClassReportable>)Class.forName(
      "org.jhove2.reflection_demo.annotations.ClassReportable");

  if (objectClass.getAnnotation(ClassReportableClass)==null){
    throw new NonReportableFeatureException();
  }
  String classCanonicalName = objectClass.getCanonicalName();
  Class<FieldNotReportable> ClassFieldNotReportable =
    (Class<FieldNotReportable>)
    Class.forName("org.jhove2.reflection_demo.annotations.FieldNotReportable");

  // Now get each field, and get its description UNLESS the field
  // is not reportable
  ArrayList<FeatureDescription> fDescriptions =
    new ArrayList<FeatureDescription>() ;
  Field[] objectFields = getClassAndInheritedFields(objectClass);
  for (Field field:objectFields){
    String fieldName = field.getName();
    StringBuffer sb =
      new StringBuffer(classCanonicalName).
        append(".").append(fieldName);

    // only report reportable features
    // (i.e. DOES NOT HAVE FieldNotReportable annotation)
    if (field.getAnnotation(ClassFieldNotReportable)==null){
      String fieldGenericType = field.getGenericType().toString();
      String cleanFieldGenericType = cleanUpGenericTypeName(fieldGenericType);
      String fieldDisplayName = fieldName;
      FeatureArity arity =
        FeatureUtilities.determineFeatureArity(field.getType());
      FeatureDescription fdesc =
        new FeatureDescription(arity,
          sb.toString(), cleanFieldGenericType, fieldDisplayName);

      //We have another annotation that allows the reporting of
      // a field’s semantics and an authoritative reference
      Class<FeatureDescribable> FeatureDescribable =
        (Class<FeatureDescribable>)Class.forName(
          "org.jhove2.reflection_demo.annotations.FeatureDescribable");
      FeatureDescribable annFeatureDescribable =
        field.getAnnotation(FeatureDescribable);
      if (annFeatureDescribable!=null){
        String featureDoc = AnnotationUtil.featureDescribableToString
          (annFeatureDescribable);
        fdesc.setFeatureDocumentation(featureDoc);
      }
      fDescriptions.add(fdesc);
    }// end if reportable field
  }//end for each field
}
```

* This is a feature which allows us to extend our definition of the representation information of a feature and to report a semantic description of the field, and give a reference to the authoritative specification. Have a look at the org.jhove2.reflection_demo.annotations.FeatureDescribable declaration to see how this annotation is defined, and at org.jhove2.reflection_demo.groceries.food.bread.BreadSlice to see the annotation in use. We might not want to report these metadata in the RepInfo. But it would enable us to populate Section 8 of our format specification template from information in the source code, while observing the DRY (Don’t Repeat Yourself) heuristic, and would keep our code in sync with our specification.
return fDescriptions;
}

So, what do we do with all this meta-information? Well, we have a lot of user configuration templates to create. Our code "knows" how to introspect and get a list of all reportable fields and a lot of information about each one (its identifier, its name, its type, its arity, and any semantic notation and authoritative reference information the programmer notes in the class declaration). So, let's use our utility methods to get a list of all reportable feature identifiers, along with the feature name, and spit them out in the Java properties file format:

org.jhove2.reflection_demo.groceries.GroceryBag.contents = contents
org.jhove2.reflection_demo.groceries.GroceryBag.itemCount = itemCount
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.slices = slices
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.flavor = flavor
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.declaredLoafWeightInOunces = declaredLoafWeightInOunces
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.sellByDate = sellByDate
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.inspectedBy = inspectedBy
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.arrayOfSlices = arrayOfSlices
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.sampleSlice = sampleSlice
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.sliceCount = sliceCount
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.endPieceCount = endPieceCount
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.avgSliceWeightInOunces = avgSliceWeightInOunces
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.maxSliceWeightInOunces = maxSliceWeightInOunces
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.minSliceWeightInOunces = minSliceWeightInOunces
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.actualLoafWeightInOunces = actualLoafWeightInOunces
org.jhove2.reflection_demo.groceries.food.bread.BreadSlice.weightInOunces = weightInOunces
org.jhove2.reflection_demo.groceries.food.sandwichfillings.Jelly.flavor = flavor
org.jhove2.reflection_demo.groceries.food.sandwichfillings.Jelly.consistency = consistency
org.jhove2.reflection_demo.groceries.food.sandwichfillings.PeanutButter.consistency = consistency
org.jhove2.reflection_demo.groceries.food.sandwichfillings.Sandwich.top = top
org.jhove2.reflection_demo.groceries.food.sandwichfillings.Sandwich.bottom = bottom
org.jhove2.reflection_demo.groceries.sundries.JunkFood.type = type
org.jhove2.reflection_demo.groceries.sundries.JunkFood.brand = brand
org.jhove2.reflection_demo.groceries.sundries.JunkFood.weight = weight
org.jhove2.reflection_demo.groceries.sundries.LowFatJunkFood.containsSaturatedFat = containsSaturatedFat
org.jhove2.reflection_demo.groceries.sundries.LowFatJunkFood.caloriesPerOunce = caloriesPerOunce
org.jhove2.reflection_demo.groceries.sundries.LowFatJunkFood.brand = brand
org.jhove2.reflection_demo.groceries.sundries.LowFatJunkFood.weight = weight

Now a user has an editable file with all possible reportable features listed. That user can edit any name value he or she wishes (or create i18n versions of that file), so that, at run-time, user-preferred names for features are displayed.

Or, if we just spit out a simple list of all reportable features, a user can select the features that should be suppressed. Which brings us to:
2. Run-time application configuration

With reflection and annotations, we have an automatic mechanism to create user configuration templates from our source code. Once the users fill in these templates, the files can be used to configure the application. In real JHOVE2 life, we’d probably want to generate and use Spring configuration files. In this demo, we’ll just use the properties file we generated, and the demo’s org.jhove2.reflection_demo.config.ProjectConfig utility methods to load them when the demo is executed.

In the demo, we configure two aspects of the configuration capabilities we’ve listed in our requirements: user names for features, and user suppression of reporting of features. For the first aspect, a reflection-generated properties file was edited to create resources/org/jhove2/reflection-demo/kbFeatureNames.config. This file specifies user choices for feature names (feature names were edited to begin with the prefix "KB_").

For the second aspect – feature suppression – we generate another file, which simply lists all possible reportable features. This file was edited to contain only the features we want suppressed. In the demo, we split this file into a set of configuration files, one each at the level of the Grocery class being configured, to model "localizing" this information by package (We could just as easily have created a single file, anywhere on the classpath, listing all features to be suppressed). These are the configuration files:

```
resources/org/jhove2/reflection-demo/groceries/
  GroceryBagSuppressedFeatures.config
resources/org/jhove2/reflection-demo/groceries/food
  BreadLoafSuppressedFeatures.config
  BreadSliceSuppressedFeatures.config
resources/org/jhove2/reflection-demo/groceries/sandwichfillings
  JellySuppressedFeatures.config
```

This runtime configuration via completed, programmatically-generated user-configuration templates brings us to:

3. Run-time processing and reporting

Invoking the FeatureUtilities.getUserConfiguredNamesFeatureDescriptions() utility method to give us only user-configured reportable features, with their user-configured name, we get:

```
**************************************************************************
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.flavor,KB_flavor
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.declaredLoafWeightInOunces,KB_declaredLoafWeightInOunces
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.sellByDate,KB_sellByDate
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.ingredients,KB_ingredients
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.inspectedBy,KB_inspectedBy
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.sliceCount,KB_sliceCount
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.endPieceCount,KB_endPieceCount
org.jhove2.reflection_demo.groceries.food.bread.BreadLoaf.avgSliceWeightInOunces,KB_avgSliceWeightInOunces
**************************************************************************
```
Where the user has configured a name, we see names with the “KB_” prefix. Where the user has not configured a name (see the Sandwich and LowFatJunkFood features), we use the “compile-time” default name. And we only see the features the user wants reported.

There is something important to note here. None of our “business” classes contains any code to serialize the object as a String or as XML or as a properties file or as anything else. None of the business classes contains any code that makes a determination about whether or not an object of that class should be traversed or reported. None of the business classes knows anything about a JHOVE2 identifier for its features, nor does it contain any code to hardwire that identifier name to a display name or a data type or an arity or a value. Nor does any business class contain code to structure an object instance of that class as a ReportableObject (this demo’s equivalent of a RepInfo). These POJOs really are POJOs. But with reflection, every one of them can be JHOVEnated into manifesting all these capabilities.

By way of illustration, the demo loads up a GroceryBag, creates the RepInfo tree for the contents of the bag, and then serializes all the reportable features of the each one of the reportable objects in that GroceryBag as XML, all with (sometimes reflective) static utility methods in our utility class. Embedded here is an example of the output:

"Viewer discretion is advised"

There are still no silver bullets; there is still no free lunch. What’s the downside of using reflection and annotations?

There are three main concerns: complexity, efficiency, and purity.
Unusually, almost quaintly, the Sun Java Reflection tutorial[^3] slaps a warning label on the reflection API:

> This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language.

As with pointers in C programming, a programmer can get into trouble using reflection. It won’t work very well in an Applet. It violates the principle of encapsulation, giving external access to private class members. Like the character in Moliere, who did not realize he had been speaking prose all his life, many of us developers have been using reflection all our (programming) lives – by implementing interfaces, and extending classes. But we don’t necessarily wittingly invoke it in our everyday use, or understand its inner workings. This might comprise a barrier to anyone picking up JHOVE2 with an eye to extending it.

Having duly warned us, however, the tutorial continues:

> With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible

So it does, and so it would for JHOVE2. But we would not want to employ it promiscuously. It should be “containerized” in a library transparently simple to employ (say, by slapping an annotation on a POJO) for those not interested in the finer details of the reflection API, any more than they are interested in the workings of the `java.io` library as it adapts to each operating system and platform on which it is invoked.

Efficiency is a concern. There is a run-time cost associated with indirection, whether we are talking about interfaces or inheritance or reflection. It is the (run-time) price we pay for (run-time) flexibility and (development-time) reuse. We obviously are not going to worry about the run-time costs of using reflection to generate configuration templates, as this is an “off-line” process, rarely invoked, and finished before the first time any one user actually invokes JHOVE2. Configuration via reflection is a run-time cost, but not a recurring one as we process multiple reportable units.

Our greatest concern should be with the library of reflective methods used for assembling and serializing representation information (type, name, identifier, etc.) of a format instance at run-time. If these methods do prove too expensive to use at run-time, however, we are not necessarily back to hacking out essentially the same boiler plate code, class by reportable class. Just as we generate user configuration templates via reflection, we can use template Java code and reflection to generate the concrete methods that provide this functionality. This will be extremely valuable as we and the JHOVE community look to extend the number of format modules by wrapping existing format tools.

Finally, there’s purity. As mentioned, use of reflection violates the principle of encapsulation. If we “encapsulate” our use of reflection, that may mitigate concern on this score.

Also, some feel that the use of annotations injects a framework dependency on a POJO (If someone wants to reuse our PeanutButter class, even without the static utility methods, they’re going to have to import our annotations package, because we have an annotation in the class declaration). We will

[^3]: This could be done in much the same way that the Maven XRTS plugin is used to generate concrete, specialized Java MessageBundle classes from XML message internationalization files. See [http://www.mhaller.de/archives/26-maven-xrts-plugin-Generating-Message-Bundles-from-XML.html][ext2]. See also the Sun Microsystems article on using annotations to add validity constraints to JavaBeans [10], below in references.
have to weigh this cost against the benefits of code reuse and ease of adaptation of existing tools to JHOVE2 use. If we feel we don’t want to use annotations, however, we are not precluded from using reflection. We will have to employ some other convention for indicating “reportable” fields (for example, any protected field, or any field whose name begins with a particular suffix (\'_J2\_’)), and have reflection make that comparison.

And now, the promised heuristics:

“*They’re not really rules – they’re more what you’d call guidelines*”

"I hear nothing, I see nothing, I know nothing!"

Sergeant Schultz as the original POJO – who knew? But he had the right idea: POJOs are our friends. Filter out cross-cutting concerns.

"*Work is the curse of the drinking classes*"

We all have a lot on our plates. Let the code and the user input templates and the documentation write themselves.

"*It’s turtles all the way down*"

JHOVE2 code not only has to work well; it has to read well. It has to be easy for the JHOVE2 community to pick up and understand and extend the delivered code – especially to extend it with new format modules. That means we need to filter out details -- static data. It also means trying to make this code self-similar across all scales. We should reuse the large scale coding idioms of our framework (Spring) and tools (Maven) – things like declarative programming, convention over configuration, inversion of control, loose coupling, separation of application logic from cross-cutting concerns, application assembly by dependency injection. We should use them consistently, conventionally. We don’t want just to Write Once, Use Anywhere. We want others to be able to Read Once, Understand Everywhere.

"Wait for the opportune moment"

We are still working out how our framework will iterate and recurse over large numbers of reportable objects. We may find we have yet another configuration choice to make between the “SAX-ness” and the “DOM-ness” of the representation information or object tree we are accumulating, and when we want or need to serialize it or test it for validity or acceptability. Though not demonstrated here, the reflection API also provides proxy capabilities, which, used in conjunction with aspects, will give us the flexibility we need to make optimum choices at run-time. Spring gives us the tools to do this. But this will make it even more important, as we design our format classes, to filter out cross-cutting concerns.

And, finally,

"*Longest way round is the shortest way home*"

Boiling a format specification down to its object model, and the navigation model required properly to tokenize a byte stream and populate that object model, is hard. We shouldn’t make it any harder for
someone to create a JHOVE2 format module. The maintenance of format documentation, the
generation of configuration information collectors, the incorporation of that configuration information
into the processing of a format instance, the assembly and serialization of representation information
from a format, how we assess an instance, maybe even how we validate an instance – any possible
cross-cutting concern should be factored out of a format module. If it's boilerplate code, then it
should only be written once, it should be data-driven, and the data to generate or drive it should be
automatically generated whenever possible. At the cost of some degree of indirection or abstraction,
we can make what is manual and error-prone both automatic and consistent.

Let indirection find directions out.

"Roll the credits"

[1] Forman, Ira R. and Scott H. Danforth, Putting MetaClasses to Work: A New Dimension in Object-
Oriented Programming, Addison-Wesley, 1998


[5] Jayaratchagan, Narayanan, Declarative Programming in Java, O'Reilly, April 21, 2004

S_TACT=105AGY82&S_CMP=GENSITE (accessed 02.28.2009)

02.28.2009)

[8] Sosnoski, Dennis, Java programming dynamics, Part: Introducing refection, IBM DeveloperWorks,

[9] Sun Microsystems, JSR 175: A Metadata Facility for the Java Programming Language

Developer Network, March 2005
02.28.2009)