Proposal for Extendable Info Architecture

THIS DOCUMENT IS CURRENTLY BEING WORKED ON AND IS NOT READY FOR DISCUSSION/MODIFCATION.

Please bear with the authors until they finish up with the initial document (this note will be removed as soon as initial work is done!)

Page Description

This page will host the description and the results of the discussion on the implementation details for an extendable Info Architecture.

Problem Description

The current implementation of the ClassInfo and FieldInfo classes and its (abstract) superclass XMLClassInfo, respectively, do not provide the possibility/facilities for easy extension to add custom information provided for purposes other than XML and Java unmarshalling from a XML Schema. Hence, a refactoring of the current *Info class architecture is needed to "pave the way" for custom and/or extendable *Info classes.

The purpose of a *Info class is to store the information gathered during an unmarshalling process of a XML-Schema document containing the definition for various elements and their attributes which will eventually be mapped to Java Classes. In order to support marshalling of these generated Java classes so-called Descriptors are generated which describe the mapping of these Java classes to XML documents. To support not only the extraction of Java/XML-specific informarion from a XML-Schema, one might want to introduce a new Schema to support, for example, the mapping from Schema to SQL-based Database Engines via JDO. This developer would place her meta-information (which specifies the mapping) in the <appinfo> element of an <annotation> element within the base Schema.

The following Schema snippet shows an example of an <xs:complexType> annotated with meta-information to support the mapping between Java and JDO:

```
<xs:complexType name="employeeType">
  <xs:annotation>
    <xs:appinfo>
      <jdo:table name="employees">
        <jdo:primaryKey>
          <jdo:key>id</jdo:key>
        </jdo:primaryKey>
      </jdo:table>
    </xs:appinfo>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="id" type="xs:integer">
      <xs:annotation>
 <xs:appinfo>
   <jdo:column name="id" type="jdo:integer" />
 </xs:appinfo>
     </xs:annotation>
    </xs:element>
    ...
  </xs:sequence>
     ...
</xs:complexType>
```

Extendable ClassInfo's

"Encapsulate what varies" or "The introduction of Natures"

A Nature opens the possibility to introduce custom properties which are read from <appinfo> elements found in the schema that is handed to the Source Generator. These properties aren't stored by the Natures directly, instead they are stored in a HashMap within a ClassInfo. A Nature only specifies how these properties (contained in the ClassInfo's HashMap) are stored and assigned. In this sense a Nature represents an external accessor on a ClassInfo which is "extended" by custom properties.

In the code snippet above a schema was annotated with JDO-specific information, e.g. with a <jdo:table> tag. The information contained therein should be stored in a corresponding ClassInfo representing the "employeeType". The following code snippets show how this could be accomplished with the implementation suggested above, namely Natures.

Therefor we are in need of defining two new interfaces:

```
public interface NatureExtendable {
    /**
     * Checks if a specified nature has been added.
     *
     * @param nature the name of the nature.
     * @return true if the specified nature was added.
     */
    boolean hasNature(String nature);
    /**
     * Adds a specified nature.
     *
     * @param nature the name of the nature
     */
    void addNature(String nature);

}
public interface PropertyHolder {
    /**
     * Get a property by its name.
     *
     * @param name the name of the property to get.
     * @return the property as specified by the name.
     */
    Object getProperty(final String name);
    /**
     * Set a property specified by the name to the passed value.
     *
     * @param name the name of the property to set.
     * @param value the value to set the specified property to.
     */
    void setProperty(final String name, final Object value);
}
```

```
public class ClassInfo implements PropertyHolder, NatureExtendable {

  private Vector natures = new Vector();
  private HashMap natureProperties = new HashMap();

  //defined in NatureExtendable
  public boolean hasNature(final String natureId) {
    return natures.contains(natureId);
  }

  //defined in NatureExtendable
  public void addNature(final String natureId) {
    natures.add(natureId);
  }
  //defined in NatureExtendable
  public void removeNature(final String natureId) {
    nature.remove(natureId);

  //some more code
}
```

```
public class JDONature implements Nature {

  private static final String NATURE_ID = "jdo";

  //defined in Nature
  public Object getProperty(ClassInfo cInfo, String propertyName) {
    if(cInfo.hasNature(NATURE_ID) {
      if(propertyName.equals("tableName")) {
        return getTableName(cInfo);
      } else {
        throw new UnknownPropertyException();
    } else {
      throw new NatureNotSupportedException(cInfo.toString()); //or something else...
    }
  }

  //defined in Nature
  public void setProperty(ClassInfo cInfo, String propertyName, Object propertyValue)
{
    if(cInfo.hasNature(NATURE_ID) {
      if(propertyName.equals("tableName")) {
        assert propertyValue instanceof String;
        return setTableName(cInfo, propertyValue);
      } else {
        throw new UnknownPropertyException();
    } else {
      throw new NatureNotSupportedException(cInfo.toString()); //or something else...
    }
  }

  private String getTableName(ClassInfo cInfo) {
    return cInfo.getNatureProperties().get("tableName");
  }

  private void setTableName(ClassInfo cInfo, String tableName) {
    cInfo.getNatureProperties().put("tableName", tableName);
  }
  //some more properties & code
}
```

Along with the introduction of Natures we need to add (at least) two other Interfaces/Classes to model the flow from an annotation (representing some nature property) added to a Schema, on to the addition of a property in the ClassInfo's HashMap and finally to produce some output (JClasses, Descriptors) from these Nature-specific properties. These additional interfaces are the NatureHandler and the NatureProducer.

The NatureHandler

The NatureHandler's purpose is to parse/handle the information obtained from an <appinfo> element and extract the Nature-specific content therein. It then passes this extracted information to a corresponding ClassInfo via the appropriate Nature implementation.

The NatureProducer

The NatureProducer obtains Nature-specific information stored in the HashMap of a ClassInfo and produces the corresponding output, e.g. JClasses, Descriptors, and so on.