PersistentInterfaces

Persistent Interfaces

Persistent Interfaces is a feature of JDO 2 that allows users to define their domain object model in terms of Java interfaces instead of Java classes. For example, this defines a persistence-capable class called Company with two persistent fields:

```
class Company {
  long companyid;
  String name;
}

<class name="Company">
  <field name="companyid" primary-key="true"/>
  <field name="name" column = "NAME"/>
</class>
```

To use an interface instead, this defines a persistent interface called ICompany with two properties.

```
interface ICompany {
  long getCompanyid();
  void setCompanyid(long id);
  String getName();
  void setName(String name);
}

<interface name="ICompany">
  <property name="companyid" primary-key="true"/>
  <property name="name" column = "NAME"/>
</interface>
```

A goal is to map the interfaces in package `org.apache.jdo.tck.pc.company` (hereinafter "the company package") to exactly the same schema as is used for the classes, and to use the same `CompletenessTest` and xml data. This way, all of the standard mappings of the company model can be used as is, and any bug fixes to the handling of the schema, xml data, or comparison of data will be automatically propagated to the interface tests.

XML Test Data

To change the xml test data to use factories, the attributes `factory-method` and `factory-bean` are added to the test data `bean` elements.

```
  <bean id="company1" class="org.apache.jdo.tck.pc.company.Company">
    <constructor-arg index="0" type="long">
        <value>1</value></constructor-arg>
    <constructor-arg index="1" type="java.lang.String">
        <value>Sun Microsystems, Inc.</value></constructor-arg>
    <constructor-arg index="2" type="java.util.Date">
        <value>11/Apr/1952</value></constructor-arg>
  </bean>
```

is changed to become

```
    <bean id="company1" factory-bean="companyFactory"
            factory-method="newCompany">
    <constructor-arg index="0" type="long">
        <value>1</value></constructor-arg>
    <constructor-arg index="1" type="java.lang.String">
        <value>Sun Microsystems, Inc.</value></constructor-arg>
    <constructor-arg index="2" type="java.util.Date">
        <value>11/Apr/1952</value></constructor-arg>
    </bean>
```

This requires an instance of Company{{`Factory be registered under the name companyFactory in the Bean}}`Factory, which is subclassed by CompanyModel`Reader.

Refactoring

The company package has been refactored to have each domain class implement the corresponding interface. The algorithm of the CompletenessTest creates an in-memory domain model graph from xml data and make its elements persistent. Then, a new transient in-memory domain model graph is constructed from the same xml data and the transient graph is used as the model to verify that the graph of persistent instances as fetched from the database is isomorphic to the transient graph.

CompanyFactory Patterns

Since persistent instances that implement the persistent interfaces use a factory pattern, we introduce a CompanyFactory concept that allows a runtime switch between various factories. The transient graph that is used to compare to the persistent graph is always constructed using the factory that creates instances of the concrete classes. The persistent graph is constructed using one of these patterns:

- Company{{`Factory}}`Concrete`Class: the factory instantiates new instances of the concrete classes
- Company{{`Factory}}`PMInterface: the factory calls the Persistence`Manager newInstance method with the interfaces as parameters
- Company{{`Factory}}`PMAbstractClass: the factory calls the Persistence`Manager newInstance method with abstract classes that implement the interfaces as parameters
- Company{{`Factory}}`PMConcreteClass: the factory calls the Persistence`Manager newInstance method with the concrete classes as parameters

CompanyFactory Interface

CompanyFactory is the interface that each factory must implement. The methods in the interface are those that are required by the current xml data implementations. They include constructors for each concrete class in the model.

The strategy for implementation is for a registry class Company{{`Factory}}`Registry that instantiates the default implementation of CompanyFactory, CompanyFactoryConcrete`Class, that contains methods that instantiate a new instance of the concrete class.

An application program, e.g. `CompletenessTest` uses the Company{{`Factory}}`Registry to create and register the company factory, using the class name and PersistenceManager instance to be used by the company factory. The CompanyModel`Reader (the bean factory) obtains the company factory instance from the registry via the static method `CompanyFactoryRegistry.getInstance()` and installs it in the bean factory under the name "companyFactory". Then, when the xml file is read, the factory-bean reference "companyFactory" is resolved to the factory.

The Company{{`Factory}}`Registry class contains methods to create and register factories. It does not itself implement the CompanyFactory interface but delegates to an instance of a class that does implement the Company`Factory interface.

Abstract Implementation Class

An abstract class `CompanyFactoryAbstractImpl` contains implementations for each required method, and eight abstract methods to create a new instances with no properties set. The properties are then set using setProperty methods. This allows a subclass to implement the CompanyFactory interface simply by implementing the eight abstract methods.

```
public abstract class CompanyFactoryAbstractImpl implements CompanyFactory {

    protected PersistenceManager pm;

    /** Creates a new instance of CompanyFactoryAbstractImpl */
    public CompanyFactoryInterfaceAbstractImpl(PersistenceManager pm) {
        this.pm = pm;
    }

    abstract IAddress newAddress(); // implemented in subclass

    public IAddress newAddress(long addrid, String street, String city,
        String state, String zipcode, String country) {
        IAddress result = newAddress();
        result.setAddrid(addrid);
        result.setStreet(street);
        result.setCity(city);
        result.setState(state);
        result.setZipcode(zipcode);
        result.setCountry(country);
        return result;
    }
...
```

"All a concrete factory implementation has to do" is to subclass the abstract Company{{`Factory}}`Abstract`Impl and provide implementations for the abstract methods.

```
public class CompanyFactoryPMInterface
        extends CompanyFactoryInterfaceAbstractImpl {

    /** Creates a new instance of CompanyFactoryPersistentInterface */
    public CompanyFactoryPMInterface(PersistenceManager pm) {
        super(pm);
    }

    IAddress newAddress() {
        return (IAddress)pm.newInstance(IAddress.class);
    }
...
```

Build Issues

A new system property `jdo.tck.mapping.companyfactory` is used to pick the company factory used to create the persistent object graph. After constructing the persistent object graph, the default factory is reset so that during construction of the compared objects the standard constructor is used. The maven.xml file needs to pass the system property to the CompletenessTest.

```
  <goal name="doRuntck.jdori">
    <java fork="yes" dir="${jdo.tck.testdir}"
      <sysproperty key="jdo.tck.mapping.companyfactory"
        value="${jdo.tck.mapping.companyfactory}"/>
    </java>
  </goal>
```

Configurations can specify this property in the .conf file:

```
%cat test/conf/clr.conf
jdo.tck.description = Completeness test with factory class
#jdo.tck.mapping.companyfactory=
#  org.apache.jdo.tck.pc.company.CompanyFactoryConcreteClass
jdo.tck.mapping.companyfactory=
  org.apache.jdo.tck.pc.company.CompanyFactoryPMInterface
jdo.tck.classes = org.apache.jdo.tck.mapping.CompletenessTest
jdo.tck.testdata = org/apache/jdo/tck/pc/company/companyNoRelationships.xml
jdo.tck.mapping = 0
```

New Sub-package acompany

All the components described so far will remain in the company package. I propose to add the components to instantiate persistent abstract classes into a subpackage, `org.apache.jdo.tck.pc.company.acompany` and to implement the abstract classes that implement the interfaces as well as the company factory that instantiates the abstract classes using the persistence manager newInstance method with the abstract class as the argument.

Feeback Requested

The interface Company{{`Factory is implemented by subclasses of Company}}`FactoryAbstract`Impl. Here are proposed names for the components:

- CompanyFactory seems reasonable
- Company{{`Factory}}`Registry is the class that manages the registration of the factory.
- Company{{`Factory}}`AbstractImpl is the abstract implementation class for the Company`Factory interface.
- Company{{`Factory}}`Concrete`Class is the factory that instantiates new instances of the concrete classes
- Company{{`Factory}}`PMInterface is the factory that calls the Persistence`Manager newInstance method with the interfaces as parameters
- Company{{`Factory}}`PMAbstractClass is the factory that calls the Persistence`Manager newInstance method with abstract classes that implement the interfaces as parameters
- Company{{`Factory}}`PMConcreteClass is the factory that calls the Persistence`Manager newInstance method with the concrete classes as parameters