The ObjectRelations Framework

Contents

1. Introduction

Some years ago when I started working on another object-oriented software development project I realized that something was missing in Java and other object-oriented programming languages. While object-orientation had brought many improvements in the abstraction and reuse of code there was still a lot of code that needed to be replicated and re-implemented with every new project. Not for the objects themselves, but for the connections between the objects of an application which define the processes that are implemented by the application.

Some frameworks have tried to address this problem, but mostly by extracting the wiring of objects into separate artifacts (typically XML files). This often leads to more problems than it solves due to the separation of development processes, restricted automatic error checking, lack of tool support, and very limited reuse and abstraction. But what was really missing was a way to define the relations between objects similar to how the classes of object-orientation define the objects of an application.

Originally I planned to implement a solution as an extension to the Java language. But I soon realized that it would be very difficult to introduce new language features if that requires fundamental changes to existing development processes. Examples like aspect-oriented programming showed that even with extensive tool support adoption by developers would be marginal. Fortunately the advent of Java 5 brought new language features that allowed me to implement the concept as a regular library that would be available for normal Java development without sacrificing code readability or type safety.

2. ObjectRelations

As mentioned above, the idea behind this framework is the modeling of relations between object. But what exactly are relations? Essentially, a relation is a reference from one object to another. But that's not all: a relation is also defined by the behavior that is associated with it, and that is provided by the program code that evaluates and manipulates the reference. A good example is the common event listener mechanism. An object supporting event listeners not only needs references to the listener objects but also the code to manage listeners, collect event data, and to notify the listeners of new events.

To model relations in a generic way it is therefore necessary to create an abstraction that combines object references and the associated program code. Someone familiar with object-orientation will notice the similarities with the classes of object-oriented code. Classes achieve abstraction by bundling the data of a certain domain with the code that operates on the data. And that is basically what the ObjectRelations framework does for relations.

The ObjectRelations framework is based on three fundamental elements: relation types, related objects, and relations, each represented by corresponding classes. The following sections explain these elements in detail.

Relation Types

Relation types are to relation-oriented software development what classes are to object-orientation. They represent the abstraction of a relation between objects as described above. The ab-

stract base class of all relation types is `org.obrel.core.RelationType`. To create a new kind of relation between objects a developer only needs to subclass `RelationType` and implement or override the necessary methods to control the behavior of relations with the new type.

The class `RelationType` is a generic type. It uses the generics feature of Java 5 to declare the type of the objects that will be referenced by relations of this type. By using generic types relation-based operations will be as type-safe as statically typed code, reducing the need of type casts and therefore enhancing the readability of relation-based code.

The generic declaration of the class is not so obvious at first glance: `RelationType<T,R>`. Instead of a single generic parameter defining the referenced type there are two different parameter T and R. The idea behind this is that not all relations will reference an object directly, especially when defining relations to objects in a different context (like another application, on the network, in a persistent storage etc.).

To support such scenarios the referenced type of a relation is split in two parts. First there is the relation target, defined by the generic type T. This is the type of the value that is actually stored by a relation of the respective type. Second is the resolved type R, which is the type that will be returned when the relation is queried. A typical example is a relation that accesses remote objects on a network. Instead of storing a heavyweight remote object reference, the relation target will instead contain a URL (or a similar identifier). Only if the relation is actually accessed will the target value be resolved and returned as a value of type R. All this is handled by the relation type subclass and is completely transparent to application code.

To implement a new relation type the minimum a subclass of `RelationType` must do is to implement the single abstract method `resolve()`. The purpose of this method is to take the target value of a particular relation and to convert it to the resolved value of the type. All other methods have default implementations that may also be overridden to modify the behavior of a relation type or to add additional functionality. Please see the framework documentation for details about these methods.

While the separation of target and resolved types facilitates the implementation of sophisticated relation types, many relation types will be used in a local context where the target and resolved values will be the same (i.e., T and R will be identical). To support this the framework contains the base class `PropertyType<T>`. It's `resolve()` method has been implemented to simply return the target value. The name expresses that relations of such a type are very similar to the properties of objects in object-oriented applications because such properties are nothing else than references to other objects.

Related objects

As a relation is a reference **to** a certain object there must also be an object from which the relation originates. The class `org.obrel.core.RelatedObject` serves as this starting point for relations. Code that wants to use relations should subclass it in as many classes as possible. `RelatedObject` manages the relations associated with an instance and provides methods to access and modify them. It is to the ObjectRelations framework what the class `Object` is to the standard class library of Java.

One of the few disadvantages of ObjectRelations being implemented as a library instead of a native language features is that objects of existing classes cannot contain relations because they are obviously not derived from `RelatedObject`. But the framework provides a way to manage relations for such objects too. The class `ObjectRelations` contains several static methods that support working with relations. One of these is `getRelatable(Object)`which returns a related object that can be used to manage the relations for an arbitrary Java object.

The returned object will be an instance of the `org.obrel.core.Relatable` interface which defines the methods to access and modify the relations of an object. It is implemented by the `RelatedObject` class and is not intended to be implemented by classes outside of the ObjectRelations framework. But it should be extended by interfaces that want to signal that their implementations will (and must) be instances of `RelatedObject`.

As a rule of thumb it can be said that each class in a relation-oriented project should be a subclass of `RelatedObject`, like each class in a Java project is a subclass of `Object`. The overhead compared to normal objects is very low so it should probably never harm to subclass it.

Relations

Relations are basically the instances of relation types, in the same way as objects are the instances of classes in object-oriented code. When a relation is set on a related object a new instance of the class `org.obrel.core.Relation` (or a subclass) will be created. The main difference to objects is that relations are created automatically and that they are typically accessed and managed through the methods of the `RelatedObject` class (which are defined in the `Relatable` interface). In most cases application code doesn't need to access relation instances directly.

But it is possible to query the relations from related objects and for some advanced scenarios it makes sense to access relations directly. One very interesting property of relations is that they are also related objects (like almost all classes in the ObjectRelations framework). This means that it is possible to set relations on relations (and on relation types as well). Such relations can seen as meta-relations or, as is the ObjectRelations terminology, annotations. The `Relation` class contains additional methods for accessing annotations on both the relation and it's type.

Meta-relations provide an additional dimension when working with relations. For example, a user interface library could evaluate the annotations of the relations of an object that is to be visualized and determine certain attributes from the meta-relations, like mandatory fields, default values, value ranges, data formats, hierarchy information, etc. Indirect meta-relations could even retrieve such attributes over the network or from some configuration storage. And all this in a fully generic way that will work with any related object and can be used transparently by all frameworks that is enabled to handle relations.

Working with relations

The ObjectRelations framework already contains several common relation types in the package `org.obrel.type`. Let's have a look at a declaration from the class `StandardProperties` which contains standard property types:

```
public static final PropertyType<String> NAME;
```

This declaration defines a property type constant with a string datatype that is called `NAME`. Please note that because ObjectRelations is a standard Java library and not a language extension it follows the standard naming convention for Java code. According to this convention the names of constants must be all upper case, which is why relation types constant are named that way. The following code shows the typical usage of a relation of the above type:

```
import static org.obrel.type.StandardProperties.NAME;


RelatedObject aObject = new RelatedObject();
aObject.set(NAME, "HelloWorld");
String sName = aObject.get(NAME);
aObject.deleteRelation(NAME);
```

Please note from this example that `RelatedObject` is not an abstract class and can be instantiated directly like `java.lang.Object`. But unlike `Object` it often even makes sense to use it directly because by setting relations on an instance arbitrary informations or even functionality can be added.

The example makes use of another new feature from Java 5: static imports. This allows to use relation types almost like object-oriented attributes as is shown above for the property NAME. Apart from the slightly different naming the main difference to accessing attributes in object-oriented code is that the property name is not part of the method name but rather a method argument. And this illustrates one big advantage of using relations even for simple attributes: relations are dynamic compared to the static attributes of object-oriented code. While the at-

tributes of classes must be defined at development time relations can be added (or removed) even at runtime as the example shows. This allows to create dynamic applications that are not easily possible in object-oriented code.

As explained above, properties are the most simple form of relations. And for many applications it may be sufficient to mainly declare additional property types without subclassing. But let's have a look at a slightly more complex example of subclassing a relation type for the common case of implementing a listener mechanism.

To support a certain listener interface it is typically necessary to implement the code to manage and notify the listeners again in every class that has to support them. Sometimes it may be possible to build the code into a base class, but due to the lack of multiple inheritance this often is not possible, especially if multiple listener types need to be supported. Extracting the listener management into a separate class still requires to implement delegating code. With ObjectRelations you only need to implement the code once in a relation type. And it is so simple that it can be done in an inner class declaration, as an example for the `ActionListener` interface shows:

```
public static final
   ListenerType<ActionListener,ActionEvent> ACTION_LISTENERS =
      new ListenerType<ActionListener,ActionEvent>("ACTION_LISTENERS")
      {
         protected void notifyListener(ActionListener rListener,
                                       ActionEvent    rEvent)
         {
            rListener.actionPerformed(rEvent);
         }
      };
```

Again this short examples demonstrates several aspects of the ObjectRelations framework. First it shows that relation types support inheritance like all other classes. The class `ListenerType` is provided by the framework and implements the base functionality for typical event listener relations. The action listener type only needs to subclass it, define the listener and event classes as the generic types, and implement a single method which performs the actual listener notification. Because of the generic types used the code is completely type-safe and does not need any type casting.

Furthermore, the example illustrates that relations can refer to arbitrary complex objects. The base class declaration of `ListenerType` is actually `PropertyType<List<L>>` where L stands for the type of the listener interface (`ActionListener` in the example). The result is that a relation of type `ACTION_LISTENERS` refers to a list of action listeners. By convention relation types that refer to multiple objects are named in plural.

Finally, this example indicates that all relation types have a name (the string argument to the constructor) that should be identical to the name of the corresponding constant declaration. The framework contains tools to enforce the identity of declaration and instance names.

The following code shows how the `ACTION_LISTENERS` relation type is used to set an action listener on an object as an instance of an anonymous inner class:

```
aActionSource.get(ACTION_LISTENERS).add(new ActionListener()
   {
      public void actionPerformed(ActionEvent rEvent)
      {
         // handle action event here
      }
   });
```

In this example `aActionSource` would be a related object that is capable of generating action events. How exactly this is done depends on the respective implementation and is not the concern of the `ACTION_LISTENERS` relation type. But it would also be possible to create relation types which perform the listener notification automatically. For example, another relation type referencing a button could register itself as an action listener on the button and notify action

listeners through the listener type if an event occurs. Notifying all listeners of a `ListenerType` subclass is very simple:

```
ACTION_LISTENERS.notifyListeners(this, rEvent);
```

Where `this` stands for the object that wants to notify it's listeners and `rEvent` is the action event that occurred. If another relation type wants to notify the listeners of a relation's parent object it would use that parent object as the first argument instead.

This concludes the introduction to the main elements of the ObjectRelations framework. The abstraction and reusability it provides can reduce the complexity and size of typical object-oriented code considerably. And because the framework itself is very lightweight it can be added to almost any project, even in constrained environments.

3. ObjectSpaces

The previous section has demonstrated that the ObjectRelations framework can simplify typical development tasks in single applications. But the dynamic and generic nature of relations make them also well suited to create relations that reference objects outside of a local context. Such a remote context could be another application that runs concurrently, a system service, or a server on a network. There is only one problem: how can remote objects be identified, especially if the state of an application is to be made persistent between accesses to the objects? One option would be to create different relation types for separate kinds of connections to external contexts, but then it would be necessary to duplicate code for similar relations just for different communication methods.

To provide a solution to this problem the ObjectRelations framework introduces the additional concept of object spaces. The purpose of an object space is to associate objects with an unique identifier and to provide a way to look up an object by it's identifier. This functionality is defined in the interface `org.obrel.space.ObjectSpace` which only contains a few methods for exactly this purpose. All further functionality of object spaces, especially how objects are managed, stored, or exported, is left completely to implementations of this interface. But object spaces are not only intended for remote access. They can also be used to create a meaningful abstraction of the objects that an application needs in its local context.

One important element of the object spaces concept is that they can be hierarchical, i.e. an object space can contain other object spaces recursively, each having an identifier. The hierarchy of object spaces in combination with the identifier of an object contained in this hierarchy can be expressed as a path or URL that identifies that particular object. Some object space implementations will then export their contents including the sub-spaces in different ways to allow global access to the contained objects. Such exporting can be based on different mechanisms, e.g. HTTP, RMI, or XML-RPC. If the object URLs are then combined with unique top-level identifiers like domain names or IP addresses, objects will be globally reachable through relations.

That said it has to be mentioned that the object spaces concept is the youngest part of the framework. It is still an ongoing research project, especially in the area of the implementation of remote access protocols.