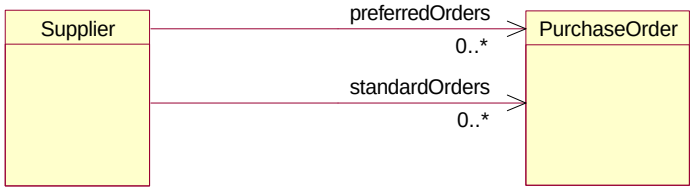
EMF FeatureMaps

June 24, 2004 (draft)

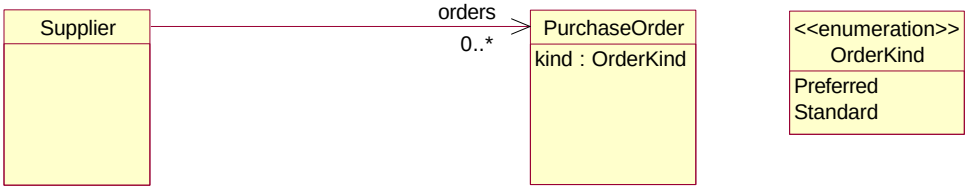
This document describes the `FeatureMap` class in EMF and shows how it can be used to automatically implement derived features in a model and how it is also used to manage repeating model groups in XML Schema based models.

Multiple features and cross-feature order

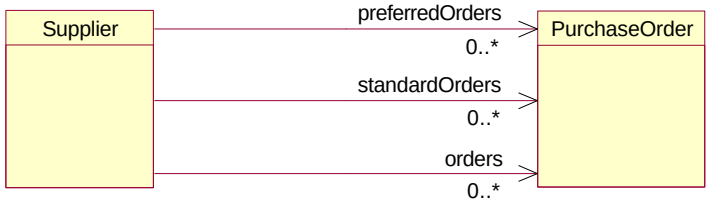
Sometimes when we design a model, we're faced with a conflict between maintaining data in a single feature versus dividing it among multiple features. Consider the following simple example of a model that manages purchase orders for some store or supplier:



In this model we use two references, **preferredOrders** and **standardOrders**, to maintain the purchase orders according to their customer's status: standard or preferred. If, however, it was important to maintain the purchase orders in, for example, the order in which they arrived, we would instead want to model this using a single reference like this:

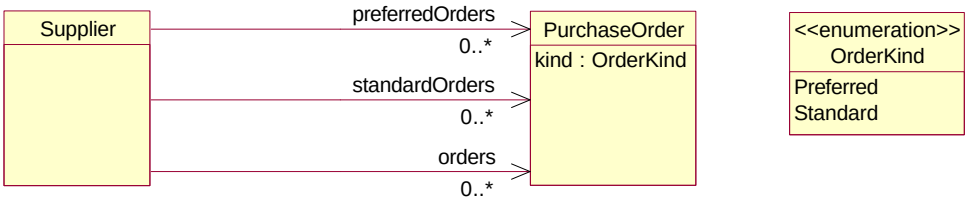


In this case, we maintain all the purchase orders in a single **orders** list, but we need to add the **kind** attribute to class **PurchaseOrder** to keep track of the preferred or standard status of each purchase order. Alternatively, we could define all three references and store each purchase order in two lists (**orders** and either **preferredOrders** or **standardOrders**, depending on its kind):



To avoid the redundant storage and having to keep multiple lists in sync, this kind of arrangement is most commonly implemented by making some of the references derive (that is, be computed) from others. For example, we could use

the **kind** attribute, from the previous diagram, to make the **preferredOrders** and **standardOrders** references derive from the **orders** reference, based on the **kind** value:

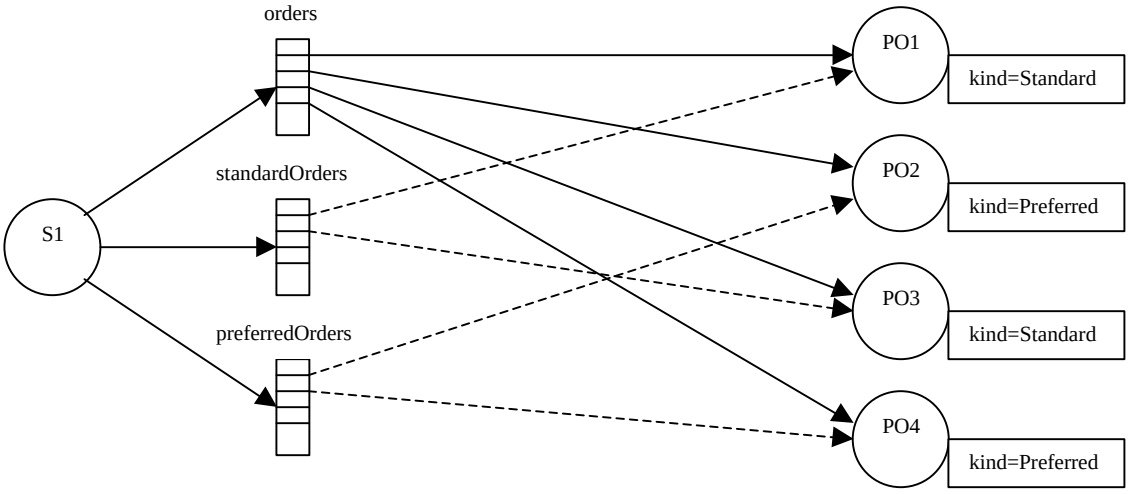


In this model, the **preferredOrders** and **standardOrders** references would be volatile, transient, and non changeable, and then implemented by iterating over and filtering the **orders** reference. For example, the `getPreferredOrders()` method would look like this:

```
public EList getPreferredOrders() {
  ArrayList preferredOrders = new ArrayList();
  for (Iterator iter = getOrders().iterator(); iter.hasNext(); ) {
    PurchaseOrder order = (PurchaseOrder)iter.next();
    if (order.getKind() == OrderKind.PREFERRED_LITERAL)
      preferredOrders.add(order);
  }
  return new EcoreEList.UnmodifiableEList(this,
                POPackage.eINSTANCE.getSupplier_PreferredOrders(),
                preferredOrders.size(), preferredOrders.toArray());
}
```

With this design, a purchase order in the **orders** list will also appear in one of the **preferredOrders** or **standardOrders** references, depending on the value of its **kind** attribute. A purchase order can only be added to or removed from the **standardOrders** or **preferredOrders** reference by adding it to or removing it from the **orders** reference, or by changing its **kind** attribute. The two derived lists are themselves not directly modifiable.

An instance of this model might look something like this:

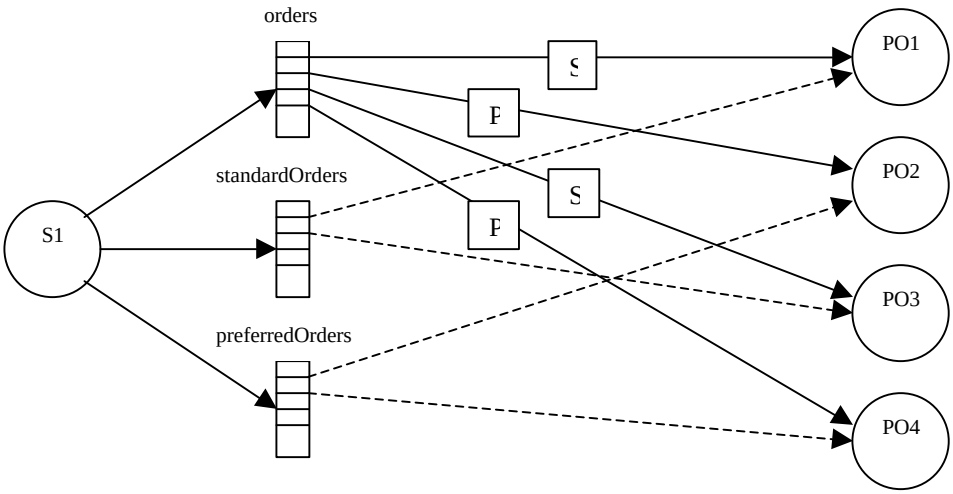


A better, but significantly more complicated, solution to this problem would not declare the **preferredOrders** and **standardOrders** references as non changeable, but instead would implement them using specialized lists that provide the entire `EList` API (including add and remove) by delegating to the **orders** list. For example, an `add()` operation on the **preferredOrders** list would delegate through to `add()` on the **orders** list, and would also set the

kind of the purchase order to `Preferred`. Because of the complexity, however, the read only approach is generally preferred.

The ExtendedPO2 model in Chapter 12 of [Ref Eclipse Modeling Framework] used this pattern to solve a similar problem. There, two references, **pendingOrders** and **shippedOrders**, were derived form the **orders** reference, based on a **status** attribute in class **PurchaseOrder**. This approach worked well in that example, but is less desirable here. The difference in this example is that, unlike the **status** attribute in the ExtendedPO2 model, the **kind** attribute is unchanging over time; a purchase order's **kind** is intended to be set only once (for example, immediately after the object is created). Changing the **kind** of a purchase order that is in the **orders** list would have the undesirable side effect of removing it from one of the derived lists and adding it to the other list.

Ideally, we would like to implement this without the **kind** attribute or any extra state information in a purchase order at all. To do that, however, we would need to somehow "tag" the entries in the **orders** list themselves with the equivalent type information:



Fortunately, EMF provides a special kind of list for doing this, `FeatureMap`, where each entry is tagged with the feature of a derived list, like **preferredOrders** or **standardOrders**, to which it belongs. Better yet, the EMF code generator understands this pattern, so the implementation can be completely generated.

FeatureMap-derived features

A `FeatureMap` is simply an `EList` subclass whose elements are feature-value pairs, defined by the interface `FeatureMap.Entry`:

```
public interface FeatureMap extends EList
{
  public interface Entry
  {
    EStructuralFeature getEStructuralFeature();
    Object getValue();
  }

  ...
}
```

In the case of a derived multiplicity-1 feature, only one entry for that feature, at most, should ever appear in the list. For a multiplicity-many feature, the backing `FeatureMap` will contain one entry for each individual value of the derived feature, as opposed to a single entry whose value is the feature's value-list itself. The `FeatureMap`

interface provides a number of convenience methods, one of which can be used to retrieve the list-view for such a feature:

```
EList list(EStructuralFeature feature);
```

Other convenience methods provide direct access to the feature or value at a specific index in the list:

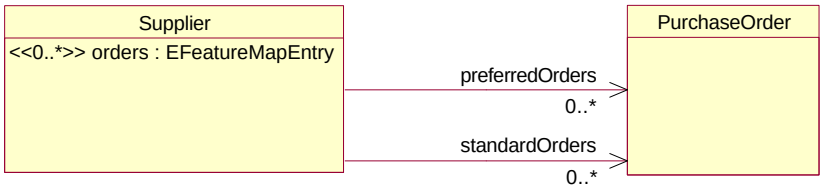
```
EStructuralFeature getEStructuralFeature(int index);
Object getValue(int index);
Object setValue(int index, Object value);
```

Methods are also provided to get or set the value of a specific single-valued feature in the list, or to add a value to a multi-valued one:

```
Object get(EStructuralFeature feature, boolean resolve);
void set(EStructuralFeature feature, Object object);
boolean add(EStructuralFeature feature, Object value);
```

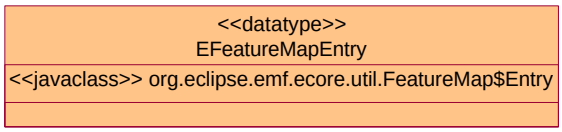
These methods, and others in the `FeatureMap` interface, provide a `Map`-like API for accessing entry values, keyed by a feature – that's why it is named `FeatureMap` instead of just `EntryList`, or something like that.

Given this powerful convenience class, you may be wondering how to use it to implement derived features, such as the **preferredOrders** and **standardOrders** references described above. Looking at the interface, you probably have some idea how you might go about programming our purchase order example, but the answer is actually simpler than you might expect. By defining a multiplicity-many `EAttribute` of type `FeatureMap.Entry` and adding a couple of `EAnnotation`s to your model, you can make the EMF code generator generate the complete implementation for you. Here's how you do it in UML:



(Note: I think that we should change the importer to alternatively allow the orders feature to be defined in UML as a third reference to PurchaseOrder as in the previous diagram. I think that providing the kind=group annotation would be sufficient information to indicate that it maps to a FeatureMap implementation)

The **orders** attribute must be a data type with instance class `FeatureMap.Entry`. The built-in Ecore data type, EFeatureMapEntry, is such a data type:

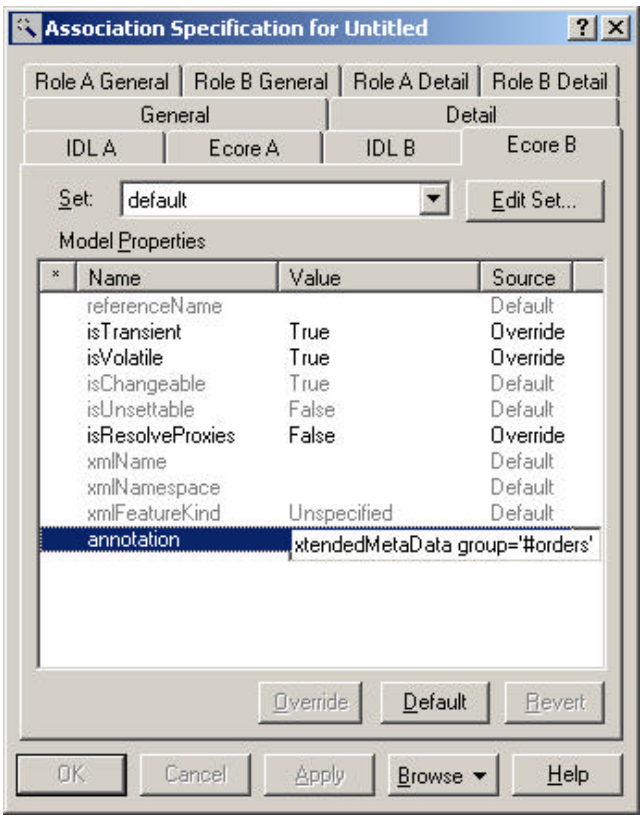


When defining these features, we need to indicate that the **preferredOrders** and **standardOrders** implementations are to be derived from the **orders** feature. To do this, we mark the two references as **transient** and **volatile**, in the usual way, but rather than hand coding the implementation methods, we annotate the Ecore model so that the derived implementations will be generated for us. `EAnnotation`s with **source** URI set to `http:///org/eclipse/emf/ecore/util/ExtendedMetaData`, are used for this purpose.

To indicate that the **orders** attribute will be used to combine (a "group" of) other features, we add to it an ExtendedMetaData-type `EAnnotation` with a single **details** entry with `key="kind"` and `value="group"`.

For the two derived references, **preferredOrders** and **standardOrders**, we also add such an `EAnnotation`, but this time with a **details** entry with `key="group"` and `value="#orders"`, to indicate that they are derived from the **orders** (group) attribute.

In Rational Rose, we can use the **Ecore** page of the **Association Specification** dialog to set these annotations. For example, we would set the annotation property of the **preferredOrders** reference like this:



The syntax of the annotation property in Rose is a **source** URI followed by one or more **details** key=value pairs, so for our example we would set it to the value: http:///org/eclipse/emf/ecore/util/ExtendedMetaData group='#orders'.

If, instead of using UML, we wanted to define this same model using XML Schema, it would be even easier:

```
<xsd:complexType name="Supplier">
  <xsd:sequence>
    <xsd:choice maxOccurs="unbounded" ecore:name="orders">
      <xsd:element name="preferredOrders" type="PurchaseOrder"/>
      <xsd:element name="standardOrders" type="PurchaseOrder"/>
    </xsd:choice>
    ...
  </xsd:sequence>
</xsd:complexType>
```

In XML Schema, a repeating choice (that is, with `maxOccurs` > 1) maps to exactly this pattern (that is, the importer would automatically add the `EAnnotation`s to the model). The choice represents a heterogeneous list of the elements defined within it; it represents the **orders** list in our example. The choice, itself, is not named in an XML Schema, so EMF supports the extended attribute `ecore:name` to name it. As you can see, we set the name to the value “orders”, highlighted in **bold** in the schema. If we had not, the `FeatureMap` feature (**orders**) would have instead been given a default name of **group** (possibly followed by a number in the case of conflict, for example **group1**).

Using either of the model definitions (UML or XML Schema), we can import the model and generate the implementation classes. As expected, the generated `Supplier` interface looks like this:

```
public interface Supplier extends EObject
{
  FeatureMap getOrders();
  EList getPreferredOrders();
  EList getStandardOrders();
}
```

The generated `getOrders()` method in class `SupplierImpl` simply creates a default generic `FeatureMap` implementation class, `BasicFeatureMap`, like this:

```
  public FeatureMap getOrders() {
    if (orders == null) {
      orders = new BasicFeatureMap(this, POPackage.SUPPLIER__ORDERS);
    }
    return orders;
  }
```

The **preferredOrders** and **standardOrders** references implementations delegate to the `list()` method of the **orders** `FeatureMap`. For example, the `getPreferredOrders()` method is implemented as follows:

```
  public EList getPreferredOrders() {
    return
     (FeatureMap)getOrders()).list(POPackage.eINSTANCE.getSupplier_PreferredOrders());
  }
```

Using these implementations, we can now add, remove or move purchase orders in any of the three lists, and the others will automatically be synchronized.