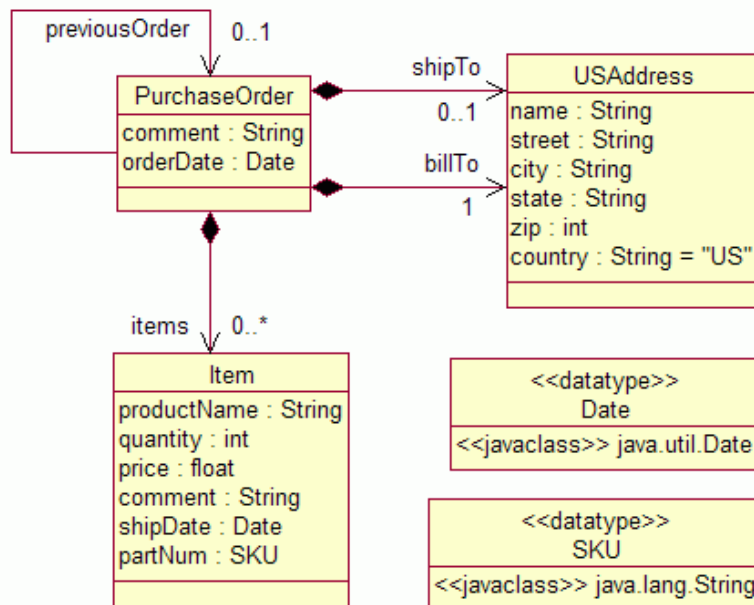


Exercise 1: Code Generation, Regeneration, and Merge

What This Exercise Is About

You will build a model and editor from an XML Schema document. Once the editor is generated, you will have an opportunity to explore the regeneration and merge capabilities of EMF by tweaking the code code for the editor and regenerating it to see the results. The schema you will use describes a fairly simple model for purchase orders. It is equivalent to the following UML class diagram, which depicts the class structure of the application.



The schema also describes the details of an XML serialization for purchase orders, which could be used for storage or transmission. The model code generated in this exercise will also be reused in exercises 2 to 4.

What You Should Be Able To Do


At the end of the lab, you should be able to:

- Import a model into Eclipse using EMF
- Generate a model editor
- Use the generated editor to create an instance of the model
- Modify the generated editor code to implement UI changes without losing those changes when regenerating

Required Materials

- Eclipse 3.2
- Eclipse Modeling Framework (EMF) 2.2

General Advice / Warnings

- In order to simplify some of the tedious tasks you would need to both setup your environment and perform the exercises, we've created a cheat sheet that should be used together with the instructions available in HTML format. Look for this icon  to identify steps you can skip using the cheat sheet.
- Eclipse conventions suggest naming plugin projects with a fully qualified plugin ID, such as com.example.po. Do not use dashes in project names - this will create invalid entries in your MANIFEST.MF file.
- Make sure you're using the IBM JDK or else have implemented the Sun JDK Crimson DOM bug workaround (<http://eclipse.org/emf/downloads-xerces.php>)
- For more on XML Schema:
 - <http://www.eclipse.org/emf/xsd/>

- o <http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/>
- o <http://www.w3.org/TR/2004/PER-xmlschema-1-20040318/#declare-type>

Exercise Instructions

This exercise is carried out entirely using the Eclipse Software Development Kit (SDK) version 3.2 with the Eclipse Modeling Framework (EMF) 2.2 installed into it. The exercise instructions refer to this product as either Eclipse or as "the workbench."

These instructions are located in your workspace, in the [EMF_Workshop/Exercisel_CodeGenRegen_Merge](#) folder that was imported by the [Setup](#) step of the cheat sheet. If you browse into this location by expanding the folders, you should find two additional files: `PurchaseOrder.xsd` and `PurchaseOrder.mdl`. These contain two different representations of the purchase order model: the XML Schema document that you will use in this exercise and the Rose model pictured above. You can open the schema in a text editor or the Sample XML Schema Editor to see how it presents the same information as the class diagram.

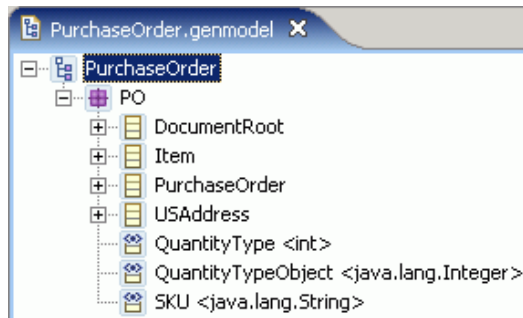
The full solution to this exercise can be installed using the cheat sheet.

Directions

Step A: Import the model

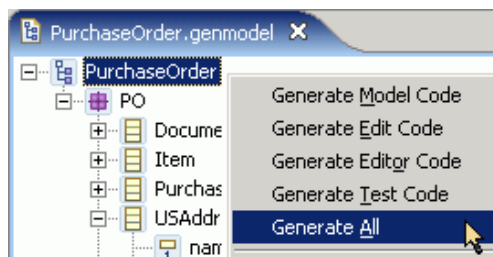
There are three supported sources for importing a model into EMF: XML Schema (XSD), annotated Java and UML (Rational Rose). You can create an Ecore model and generate code from any of these, but in this exercise you will use a schema.

1. Switch to the [Plug-in Development](#) perspective, if not already there.
 - a. Select [Window -> Open Perspective -> Other... -> Plug-in Development](#).
2. In the [Package Explorer](#) view, browse to the [Exercisel_CodeGen_Regen_Merge](#) folder and select the file `PurchaseOrder.xsd`.
3. Create a new EMF Project called `com.example.po`.
 - a. Right-click and select [New -> Project...](#) (or hit [CTRL-N](#)). You can also select [New -> Project...](#) from the [File](#) menu.
 - b. Expand the twisty next to [Eclipse Modeling Framework](#) and select [EMF Project](#). Hit [Next](#).
 - c. Name the project `com.example.po` and hit [Next](#).
 - d. Select [XML Schema](#). Hit [Next](#).
 - e. Hit [Load](#) to load the schema file that's been prefilled for you. Hit [Next](#).
 - f. Because the model is specified in one schema with a single target namespace, its Ecore representation is a single package. Notice that this package, `com.example.po`, has been selected. Hit [Finish](#).
4. Expand the generator model, `PurchaseOrder.genmodel`, in the EMF Generator.
5. Walk the tree to see what was imported. Contrast with the XML Schema source file.
6. Rename the `Po` node to `PO` so that the generated classes (for the package, factory, resource implementation, editor, etc.) will have more conventional names.
 - a. Select the `Po` node, right-click, and select [Show Properties View](#) (if this view is not already shown).
 - b. Scroll the [Properties](#) view to find the [All](#) twisty. Underneath is the [Prefix](#) property. Set that property to `PO` instead of `Po`.
 - c. Save your changes.



Step B: Generate Editor Code

1. Generate all code from the genmodel file.
 - a. Open the created genmodel file (if not already opened by Eclipse).
 - b. Select the topmost node **PurchaseOrder**. Make sure you select the genmodel root, not a package or class below in the tree, as the selected node will determine what artifacts are generated. The benefit here is that if you later make changes that affect just a single class, you can regenerate only the relevant code, which is faster than regenerating everything.
 - c. Right-click and select **Generate All**.



- d. Three new projects will be created: **com.example.po.edit** (edit support code not dependent on Eclipse), **com.example.po.editor** (editor code dependent on Eclipse), and **com.example.po.tests** (JUnit test skeletons for exercising any volatile features and operations defined in the model). Additionally, new classes and packages will be created within your **com.example.po** project.
2. The code should be compiled automatically as it is generated, and should recompile whenever it is changed. If you have disabled automatic building in the workbench preferences, you can initiate compilation manually by selecting **Build All** from the **Project** menu.
3. Observe the **Problems** view. There should be warnings about code never being used locally. These stubs will be used by tests that you could write to test your model, but that is out of the scope of this exercise. You can safely ignore these warnings.

Description	Resource	In Folder	Location
The method getFixture() from the type DocumentRootTest is ...	DocumentRootT...	com.example.po.tests/src...	line 78
The method getFixture() from the type ItemTest is never used...	ItemTest.java	com.example.po.tests/src...	line 71
The method getFixture() from the type PurchaseOrderTest is ...	PurchaseOrderT...	com.example.po.tests/src...	line 71
The method getFixture() from the type USAddressTest is neve...	USAddressTest.j...	com.example.po.tests/src...	line 71

Step C: Create a Model Instance (Run Generated Model Editor Code)

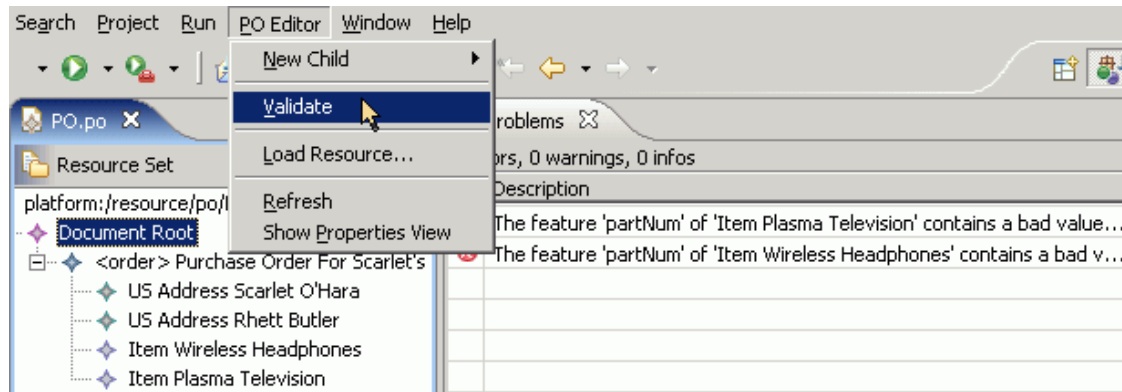
All steps below will be done in a second, testing workbench, or Eclipse Application workbench.

1. Launch a new Eclipse Application to try out your editor.
 - a. Select your plugin project **com.example.po** in the Package Explorer.

- b. Select **Run As/Eclipse Application** from the **Run** toolbar drop-down or menu. Note that the Run menu is context-sensitive, so options available under **Run As** will change depending on where your cursor is (which view/editor and what file type).
 - c. Once it opens, close or minimize the **Welcome** view.
 - d. Select **About Eclipse SDK** from the **Help** menu, click on **Plug-in Details**, and look for the contributed PurchaseOrder plugins, `com.example.po`, `com.example.po.edit`, `com.example.po.editor`, and `com.example.po.tests`.
2. The PurchaseOrder Model wizard can now be used to create a new instance of the model.
 - a. Bring up the **File/New/Project...** dialog.
 - b. Expand **General** and select **Project**. Click the **Next** button.
 - c. Give the project a name, such as `po`, and click the **Finish** button.
 - d. Right-click the project and select **New/Other...** from the pop-up menu.
 - e. Expand **Example EMF Model Creation Wizards** and select **PO Model**. Click the **Next** button.
 - f. Enter a file name for the PurchaseOrder model, such as `PO.po`. Make sure it ends with a `.po` extension. Then, click the **Next** button.
 - g. Select **Order** as the model object and click the **Finish** button.
 - h. The newly created PurchaseOrder model is opened in an editor.
3. The root object in this editor corresponds to the `PO.po` resource. Under it lies a DocumentRoot, which acts as a container for the single `order` document element corresponding to the PurchaseOrder object.
4. The steps below will show you how to use the model editor to create instance data.

Cheat If pressed for time, you can skip this step by running the cheat sheet step to copy Solution 1 to your workspace, then copying `Solution1/data/PO.po` from your first workbench to your second, and editing the resulting file (if desired).

 - a. Expand the `platform:/resource/po/PO.po` resource and the **Document Root** to see the **Purchase Order** object.
 - b. If the Properties view isn't already showing, right-click the **PurchaseOrder** object and select **Show Properties View** from the pop-up menu.
 - c. In the Properties view, click on the **Value** column of the **Comment** property, and provide a comment for the purchase order. The label in the editor will be updated when you hit Enter.
 - d. Right-click the order and select **New Child/Item** from the pop-up menu. A new Item is added to the purchase order. Enter values for the **Product Name** and **Part Num** attributes using the Properties View. Note that the correct syntax for the Part Num (SKU) is `/\d{3}-[A-Z]{2}/`, or three digits, a dash, then two uppercase letters.
 - e. **Cheat** If you run the cheat sheet step to copy Solution 1 to your first workspace, you can then copy the file `Solution1/data/PO.po` in the **EMF_Workshop** project to your second workbench to save time. Otherwise, you can manually enter a comment, price, quantity, and ship date (yyyy-mm-dd) for the item in the Properties view.
 - f. Similarly, a **Bill To US Address** and **Ship To US Address** can be added to the order.
 - g. When adding dates, use the XML Schema representation of a date, in ISO format: yyyy-mm-dd.
 - h. Select **File -> Save** to serialize the purchase order.
5. Open the model data using the text editor to see it serialized in XML format.
6. To verify that the instance data conforms to the model, you can use the EMF Validation Framework to quickly check. With your `.po` file open and the Document Root node selected, run **PO Editor -> Validate**, or right-click any node in your model and select **Validate**. Errors - if any - will appear in the **Problems** view. Note that the data in `Solution1/data/PO.po` is **intentionally** wrong. We will be fixing it programmatically in a later exercise.
7. Correct any problems with your instance data, revalidate, save, and close the Eclipse Application window (the second workbench). Leave the first workbench open for the next step. We will revisit the Validation Framework later.



Step D: [Optional] Modify Code and Regenerate

In these optional steps, you'll see how the EMF code generator deals with regenerating and merging code. To do so, you'll change a label in the generated editor. First, you will make a change in the generator model, which will affect the code that gets generated. Then, you'll edit the generated code directly, ensuring your changes are retained when the code is regenerated.

1. Close your second Eclipse workbench (if not already closed). The following coding steps will be done in your first workbench, and changes will be tested by relaunching the second workbench.
2. As described in the [EMF.Edit Framework Overview](#), EMF.Edit uses item providers to, among other things, determine what label to display for a given type of object. In particular, it is the `getText()` method that does this, and that you will need to change.
 - a. **Cheat** Open the file `com.example.po.edit/src/com.example.po.provider/USAddressItemProvider.java`.
 - b. Scroll down to the `getText()` method (or select `getText()` in the **Outline** view to jump there directly).
 - c. Notice that the generated code currently uses `getName()` (that is, the name of the person or company associated with a given address) to define the label shown for the address item when displayed in the model editor. The code looks like this:

```
/**
 * This returns the label text for the adapted class.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public String getText(Object object) {
    String label = ((USAddress)object).getName();
    return label == null || label.length() == 0 ?
        getString("_UI_USAddress_type") :
        getString("_UI_USAddress_type") + " " + label;
}
```

- d. Open the purchase order generator model (`PurchaseOrder.genmodel`) and select the `USAddress` class (third level down: Model > Package > Class).
 - e. In the **Properties** view, change the **Label Feature** to the `street : String` attribute. This determines which feature will be used in the label for `USAddress` objects, so this should set the field to display the street address instead of the name of the person or company at that address.
 - f. To have this change take effect, you don't have to regenerate all the code. You just need to regenerate the item provider class for `USAddress`. Save your changes, then right-click `USAddress` and select **Generate Edit Code** from the pop-up menu. There is also no harm in regenerating all the code -- it just takes longer.
3. If you switch over to the file `com.example.po.edit/src/com.example.po.provider/USAddressItemProvider.java`, you'll see that the generated code now shows `getStreet()` instead of `getName()`.

```
/**
 * This returns the label text for the adapted class.
```

```

* <!-- begin-user-doc -->
* <!-- end-user-doc -->
* @generated
*/
public String getText(Object object) {
    String label = ((USAddress)object).getStreet();
    return label == null || label.length() == 0 ?
        getString("_UI_USAddress_type") :
        getString("_UI_USAddress_type") + " " + label;
}

```

4. Test the change.
 - a. Launch the Eclipse Application again (**CTRL-F11**) and open the **PO.po** resource.
 - b. Expand the resource object and select the purchase order.
 - c. Notice that the street address of the USAddress is shown in the label, instead of the name.
5. As generated, the `getText()` method simply appends the value of the label feature to the type (class) of the object (in this case, "US Address"). Since a purchase order uses the same type, USAddress, to represent both the bill to and ship to addresses, it would be helpful to modify the item provider to produce a label that distinguishes between the two. It can do this by determining which feature of the PurchaseOrder contains the USAddress. To implement this, you'll need to modify the code by hand.
 - a. Open the file `com.example.po.edit/src/com.example.po.provider/USAddressItemProvider.java`. Locate the `getText()` method.
 - b. Replace the generated method with the following code:

```

/**
 * This returns the label text for the adapted class.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated NOT
 */
public String getText(Object object)
{
    String label = ((USAddress)object).getStreet();
    StringBuffer result = new StringBuffer();

    PurchaseOrder po = (PurchaseOrder)((EObject)object).eContainer();
    if (po != null && po.getBillTo() == object)
    {
        result.append(getString("_UI_PurchaseOrder_billTo_feature"));
    }
    else if (po != null && po.getShipTo() == object)
    {
        result.append(getString("_UI_PurchaseOrder_shipTo_feature"));
    }

    if (label != null && label.length() != 0)
    {
        result.append(' ');
        result.append(label);
    }
    return result.toString();
}

```

This code is relatively straightforward, but don't worry if any parts aren't clear. It uses a few things we haven't discussed yet, like the EObject API and externalized strings.

- c. The key point is to remove the `@generated` Javadoc tag, or suffix it with `NOT`. This ensures that the hand-written code will be retained when the code is regenerated.
- d. Pasting in this code introduces two errors due to missing imports on the following line:

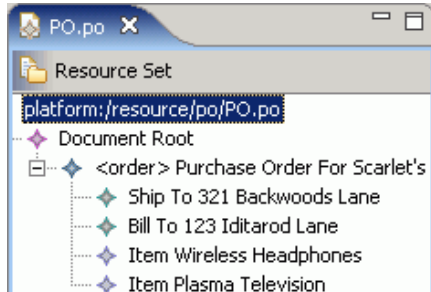
```
PurchaseOrder po = (PurchaseOrder)((EObject)object).eContainer();
```

To fix them, you can either select the above line and hit **CTRL-1** (Quick Fix context menu), or click the Quick Fix icon to the left of the line (a lightbulb with a red 'X' overlapping it). You can also use **Source -> Organize Imports (CTRL-SHIFT-O)** from either the main menu or by right-clicking. The following

imports will be added:

```
import com.example.po.PurchaseOrder;
import org.eclipse.emf.ecore.EObject;
```

- e. Save the change, launch the Eclipse Application again, and open **PO.po**. Notice that the labels for USAddresses now identify which is the Bill To and which is the Ship To address for the purchase order.



- f. Regenerate the item provider class for USAddress again, and verify that the `getText()` method is not changed.

The implementation of `getText()` has been changed from what was originally generated. The label feature property on the PurchaseOrder class in the generator model no longer has any effect on the generated code. This is because you removed the `@generated` Javadoc tag, preventing this method from being overwritten during code generation.

Step E: [Optional] Redirect Generated Code to Merge with Custom Code

It is also possible to use a hand-written method in combination with the generated implementation. In this example, you might like to have the new `getText()` method call the old generated version. This approach will still use the label feature specified in the generator model, but will append its value to the name of the containing feature, instead of just the class.

1. Provide a target for the generated method body and regenerate it.
 - a. Create a new method `getTextGen()`, with an `@generated` annotation.
 - b. To show that it will be regenerated, create an empty method body with only a single null return statement.

```
/**
 * This returns the label text for the adapted class.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public String getTextGen(Object object)
{
    return null;
}
```

- c. Save `USAddressItemProvider.java`, switch back to the generator model, select `USAddress`, and regenerate the edit code (as before).
- d. Switch back to the `getTextGen()` method in `USAddressItemProvider.java`. Notice that the generator has recognized the `Gen` suffix and redirected the original `getText()` implementation (using the street for the label feature, as specified in the generator model) into it.

```
/**
 * This returns the label text for the adapted class.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public String getTextGen(Object object)
{
```

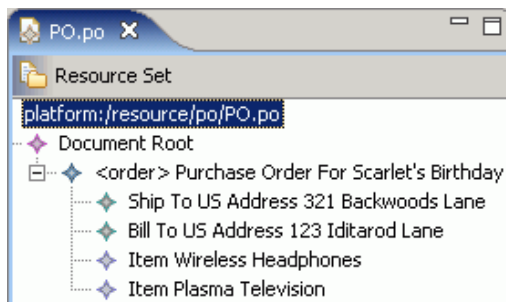


```
String label = ((USAddress)object).getStreet();
return label == null || label.length() == 0 ?
    getString("_UI_USAddress_type") :
    getString("_UI_USAddress_type") + " " + label;
}
```

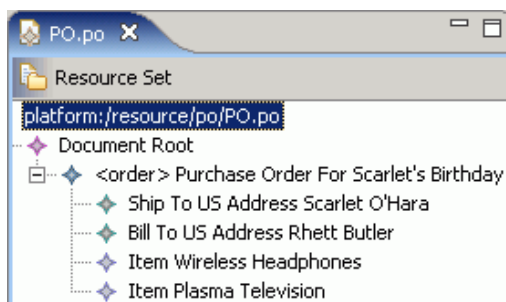
2. Modify the hand-written `getText()` implementation to call the generated version.
 - a. Change the first line of `getText()` to use `getTextGen()`, instead of calling `getStreet()` on the address directly.

```
String label = getTextGen(object);
```

- b. When you relaunch your second workbench, you'll see that the addresses are now identified as Bill To or Ship To, but also use the specified label feature from `getTextGen()`.



- c. You can go back to the generator model, change the label feature back to `name : String`, regenerate the code, and verify that this does, indeed, affect only the generated code. Relaunching your second workbench, you'll see this:



Summary

You generated a model and editor from an XML Schema document, and then used that editor to create, validate, and save an instance of that model. You modified the editor using both by using the generator model to customize the generated pattern and by hand-modifying the code, and you saw how the generator can merge its changes in with your hand-written code.