EclipseSource

What every Eclipse Developer should know about EMF

by Jonas Helming and Maximilian Koegel

EclipseSource Munich

Contact Details:
Email: munich@eclipsesource.com
Website: http://eclipsesource.com/munich
Phone: +49 89 21 555 301

This tutorial is also available as a more comprehensive EMF training.

Abstract

This tutorial is an introduction to EMF and explains the basics of EMF. We start by showing you how to build a very simple data-centric application, including the UI, based on EMF. We explain how to define a model in EMF and generate code from it. We explore the API of the generated code, that is, how to create, navigate and modify model instances.

Next we demonstrate how to build a UI based on this model using databinding. For our example, we build an application to manage a bowling league, including matches and players. Later on in the tutorial, we explore the advantages of using AdapterFactories and briefly look at data management in EMF. We also include a few pointers on the most important add-on technologies for EMF. If you are interested in getting fast results building an application based on EMF, maybe EMF Client Platform is also a good starting point for you, see this tutorial.

Installation Requirements: To work through the examples, you'll need to download and install a fresh version of the **Eclipse Modeling Tools** from the Eclipse Download Page.

Introduction

To answer the question, "What is EMF?", we'll borrow the description from the EMF website:

> "The EMF project is a **modeling framework** and **code generation** facility for building tools and other applications based on a **structured data model**. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of **Java classes** for the model, along with a set of **adapter classes** that enable **viewing** and **command-based editing** of the model, and a **basic editor**."

Source: http://www.eclipse.org/emf

It is worth mentioning that in addition to being a successful modeling framework, EMF has also been a stable standard for many other modeling technologies. We recommend using EMF for any structured data model you want to create in Eclipse, especially if it is stored, displayed and modified in UIs.
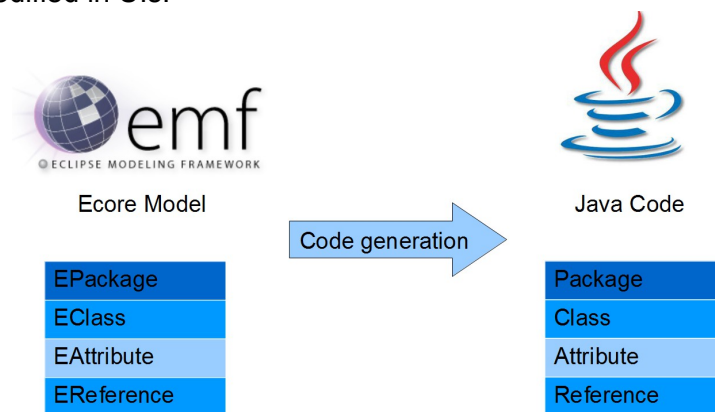

The basic EMF workflow is very pragmatic; a model is created and defined in the Ecore format, which is basically a subset of UML Class diagrams. From an Ecore model, you can generate Java code.

Later in this tutorial we will have two running instances of Eclipse. In the first instance, called the "IDE", we will define the model and generate code from it. The second instance, called the "Runtime", will be started from the IDE and will contain instances of the generated model.
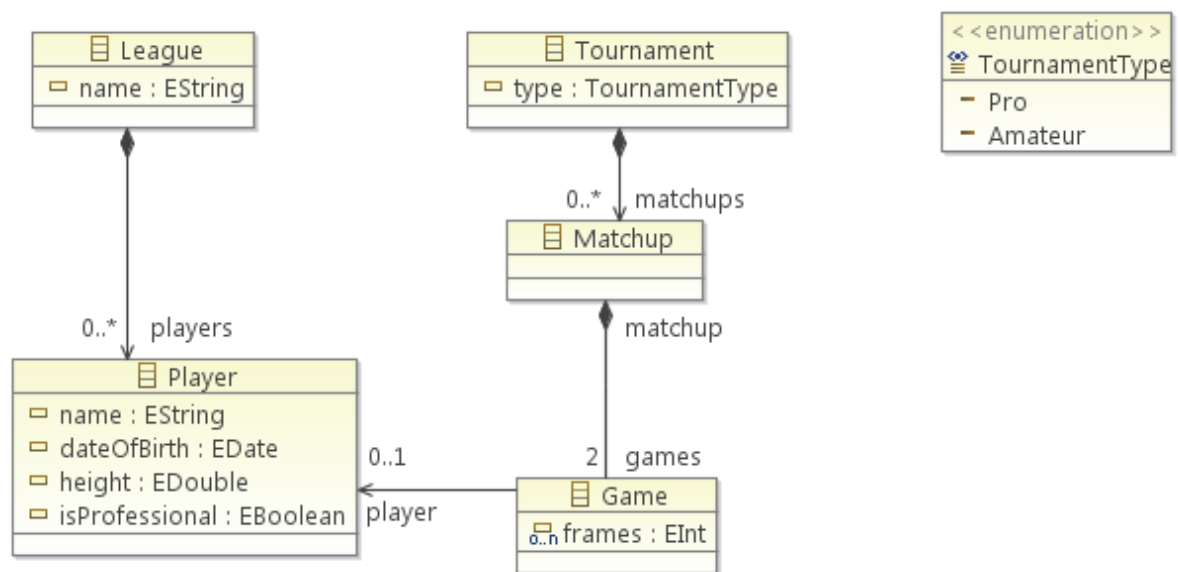
IDE

starts
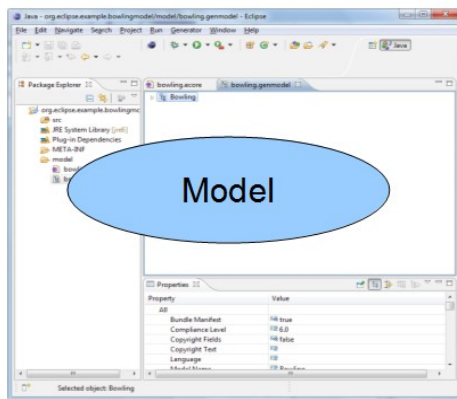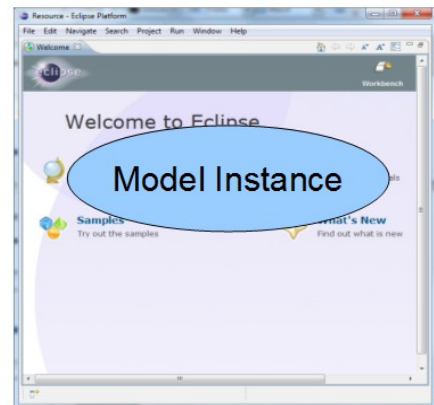
Runtime

Example Model

In this tutorial we will create an example model for managing a bowling league and its tournaments. A League contains an arbitrary number of Players. A Tournament consists of an arbitrary number of Matchups. Each Matchup always contains two Games. A Game is a list of frames (the score) and is assigned to a certain Player. Finally, a Tournament has an Enumeration that determines the type of Tournament.

```
┌─────────────────────┐        ┌──────────────────────────┐        ┌───────────────────────────┐
│ ⊟ League            │        │ ⊟ Tournament             │        │ <<enumeration>>           │
├─────────────────────┤        ├──────────────────────────┤        │ ⚇ TournamentType         │
│ ▭ name : EString    │        │ ▭ type : TournamentType  │        ├───────────────────────────┤
├─────────────────────┤        ├──────────────────────────┤        │ ▬ Pro                     │
│                     │        │                          │        │ ▬ Amateur                 │
└─────────────────────┘        └──────────────────────────┘        └───────────────────────────┘
          ◆                                  ◆
          │                        0..* │ matchups
          │                      ┌──────────────────┐
          │                      │ ⊟ Matchup        │
          │                      ├──────────────────┤
          │                      │                  │
          │                      └──────────────────┘
          │                              ◆
   0..* │ players                        │ matchup
┌──────────────────────────┐
│ ⊟ Player                 │        2 │ games
├──────────────────────────┤   0..1  ┌──────────────────────┐
│ ▭ name : EString         │         │ ⊟ Game               │
│ ▭ dateOfBirth : EDate    │ player  ├──────────────────────┤
│ ▭ height : EDouble       │◄────────│ ▤ frames : EInt      │
│ ▭ isProfessional : EBoolean │      └──────────────────────┘
└──────────────────────────┘
```

In the next section, we will show how to create and generate code from this model.
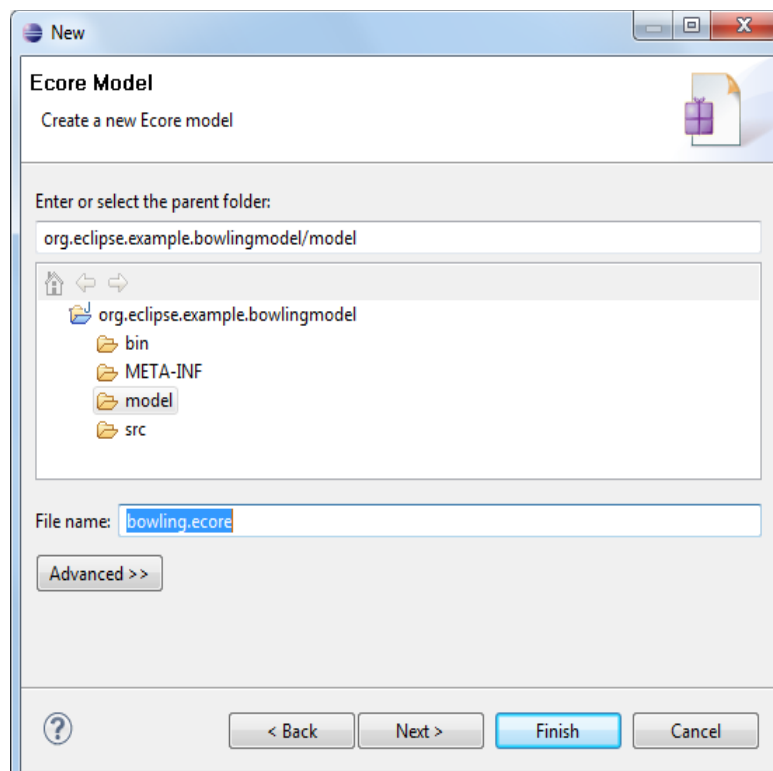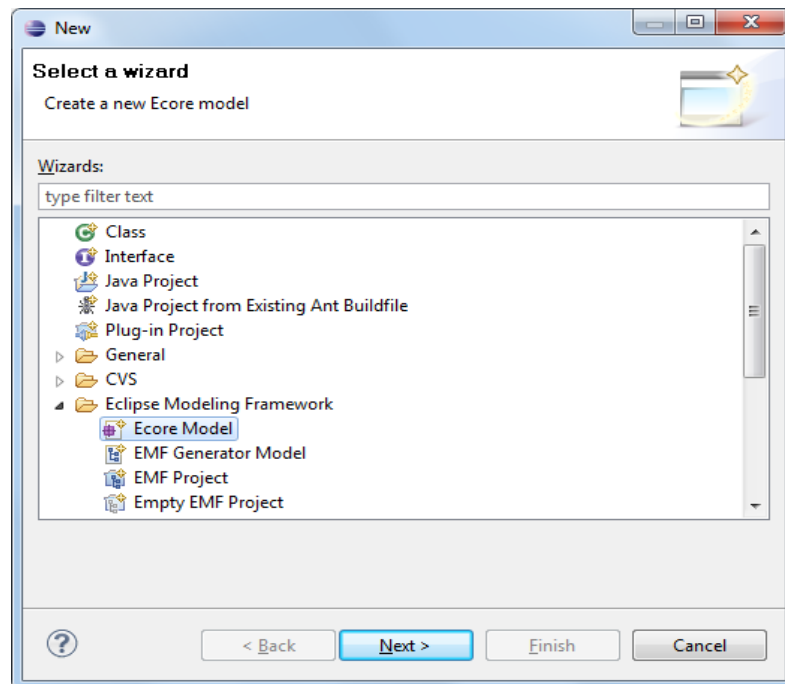
Modeling

We will create our example model in EMF to generate the entity classes for our application. The first step is to create an empty modeling project in your workspace. In your running IDE select "File" from the toolbar menu → "New" → "Other..." and choose "Empty EMF Project".
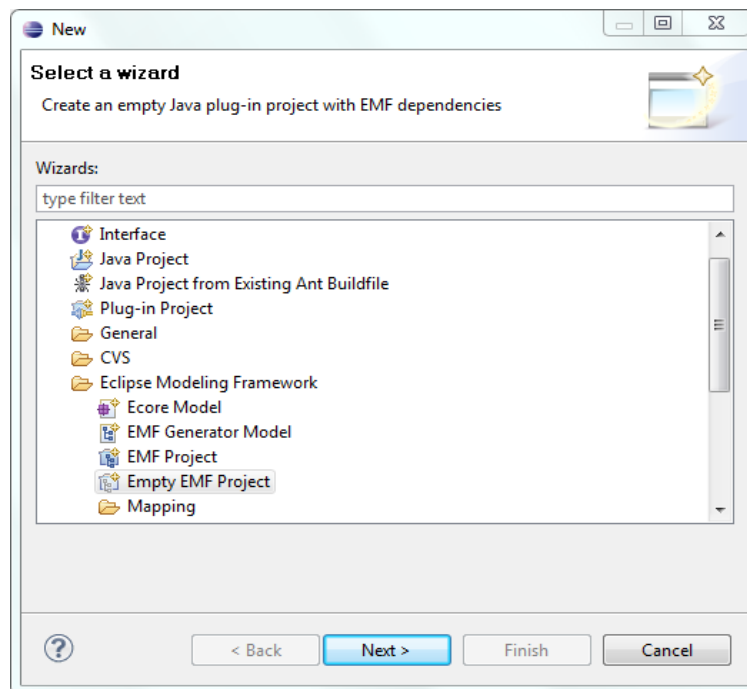
Click "Next", enter a name for the project, e.g., "org.eclipse.example.bowlingmodel", and hit "Finish":
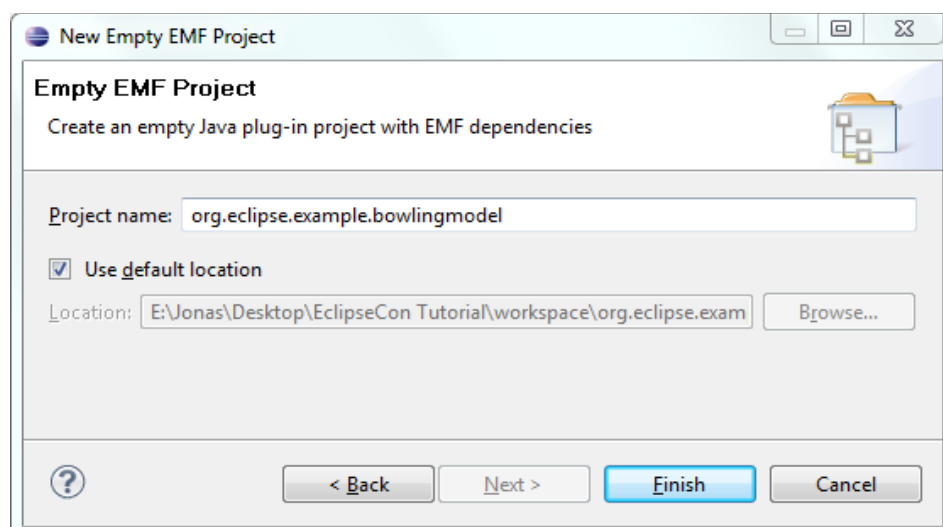
The essential part of the modeling project is the model itself, defined in the format "Ecore". Please right click the model folder in your new modeling project → "New" → "Other..." → "Ecore Model" → "Next" and give the ecore file the name "bowling.ecore".

Click "Finish" to create the model. It will open in the default Ecore editor, which allows the definition of Ecore models in a tree-based view. There are several additional options for defining Ecore models, including graphical modeling, textual modeling, Java annotations and importing from UML tools. We will stick to the default editor in this tutorial and later, briefly demonstrate the graphical editor for Ecore.

In the Ecore editor tree, you can create and delete model elements as well as modify the structure of your model via drag and drop. Properties of model elements can be modified in a second view, which opens up on double-click or right-click → "Show Properties View".
You'll need to give thepackage of your new model a name and an URI. This will be done in the properties view. The URI is used to identify the model later on. Name the package "bowling", set the Ns Prefix to "org.eclipse.example.bowling" and the Ns URI to "http://org/eclipse/example/bowling".
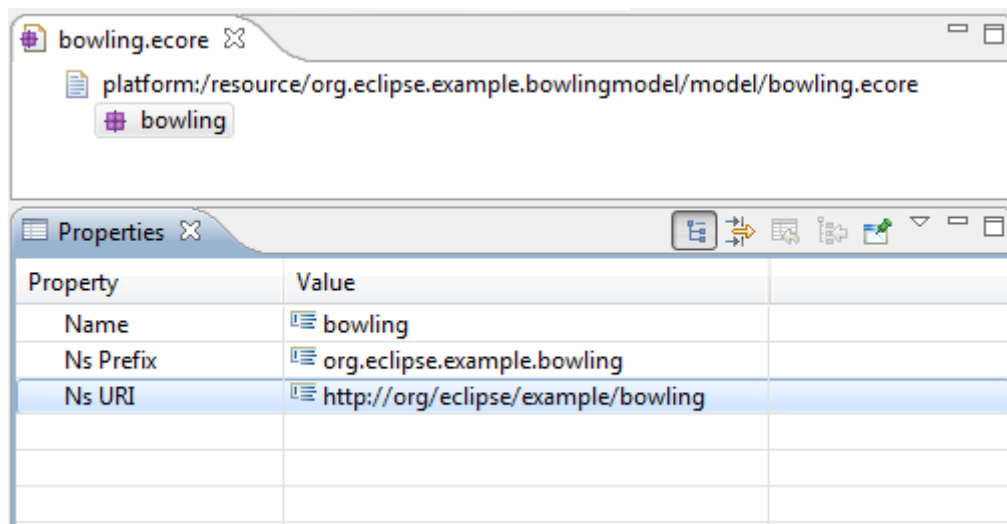

Now we can define our model elements as children of the root package. Create a new EClass by right clicking on the bowling package → "New Child" → "EClass" and set the name to Player in the properties view of the newly created EClass.
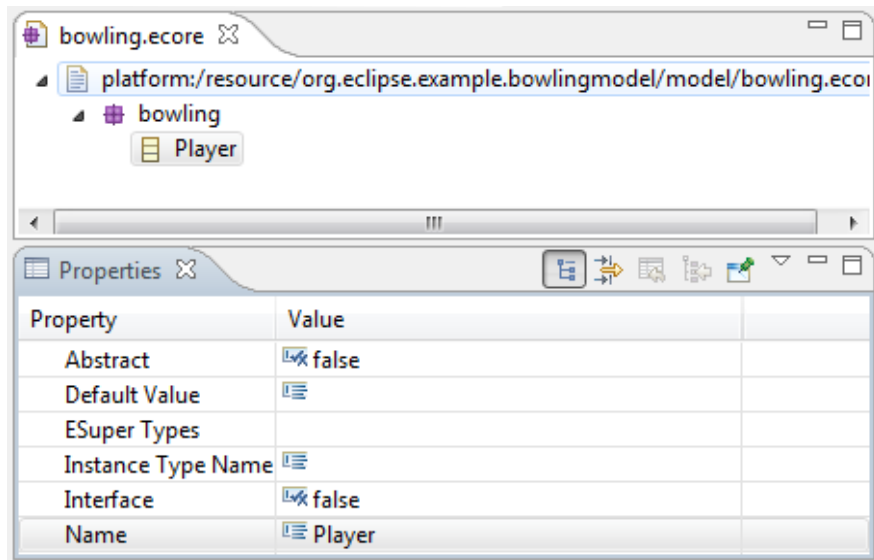
From the context menu of an EClass, you can add EAttributes and EReferences as children. Create an EAttribute in the Player EClass and open the Property view for it. The properties of an EAttribute define its name, its data type and other properties, which we will cover later in the tutorial. Set the name to "name" and assign the EType "EString" (java.lang.string). Repeat this step and add a second EAttribute named "dateOfBirth" of type "EDate". The convention we'll use here is that all class names start with an uppercase letter, attributes and references start with a lowercase letter.
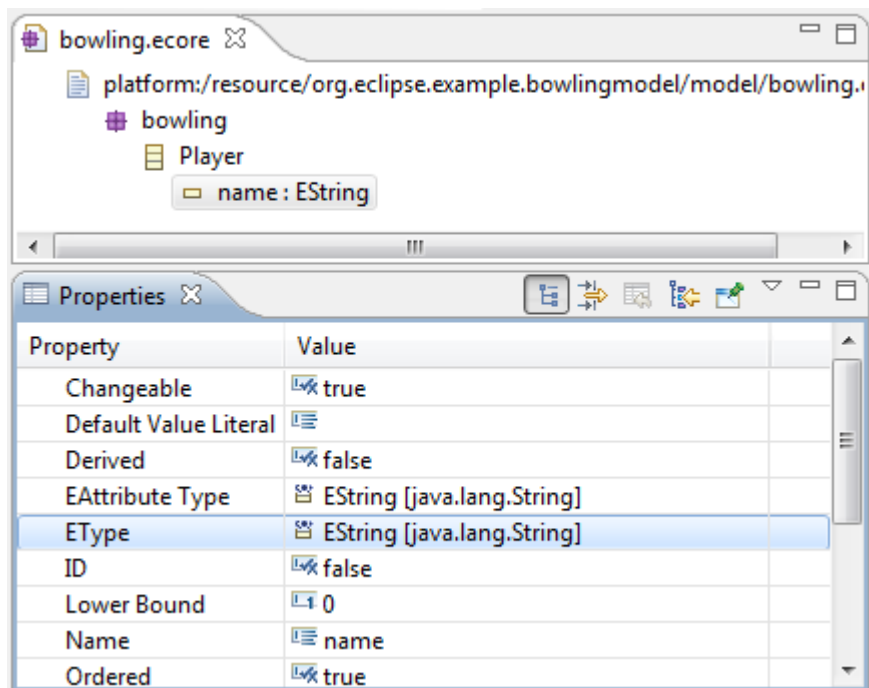
EMF models usually build up a structured hierarchy, that is, model element instances. For example, a Player is contained in a specific container object. This provides a tree structure, which is useful for navigation and serialization (e.g. XML). This tree structure is often referred to as a containment tree. In our model, Players are contained in a League. It is important to note that this also implies that every Player is referenced by exactly one League, and thus cannot be part of more than one League. EMF will automatically make sure that a player is not contained in more than one league. If you add a player to a second league, its reference to the original league vanishes.
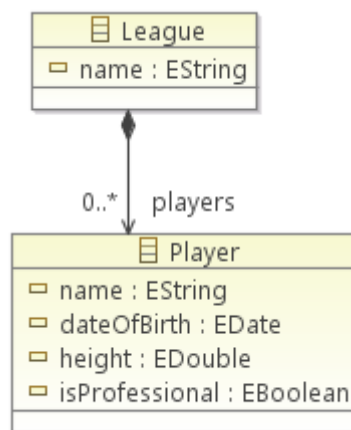
Create a second EClass and name it "League". To identify the League, also create an EString attribute called "name". The next step is to create an EReference between League and Player by right clicking on the League model element. Name the reference "players". Set the EType of the reference to "Player". Since a League can contain an arbitrary number of Players, set the upper bound to "-1", the equivalent of "many". Finally, set the property Containment to "true", defining the EReference to be a containment reference.
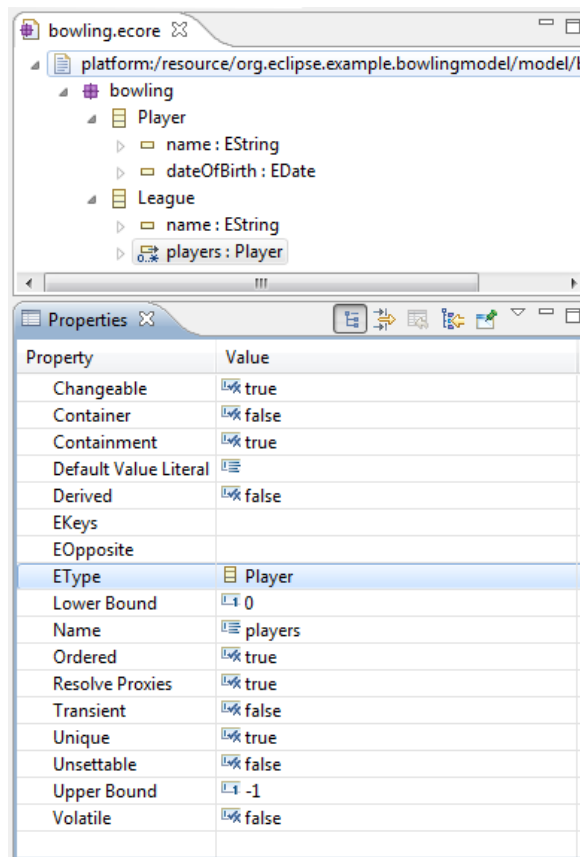
We can already generate code from this first model iteration, which will be shown in the next section. EMF can also generate an example editor. With this editor you can create instances of the generated model, in our case, instances of Leagues and Players. This allows us to do initial testing on the model by creating instances of it. Then we can further refine and add more EAttributes and EReferences in a second iteration that will complete the model.
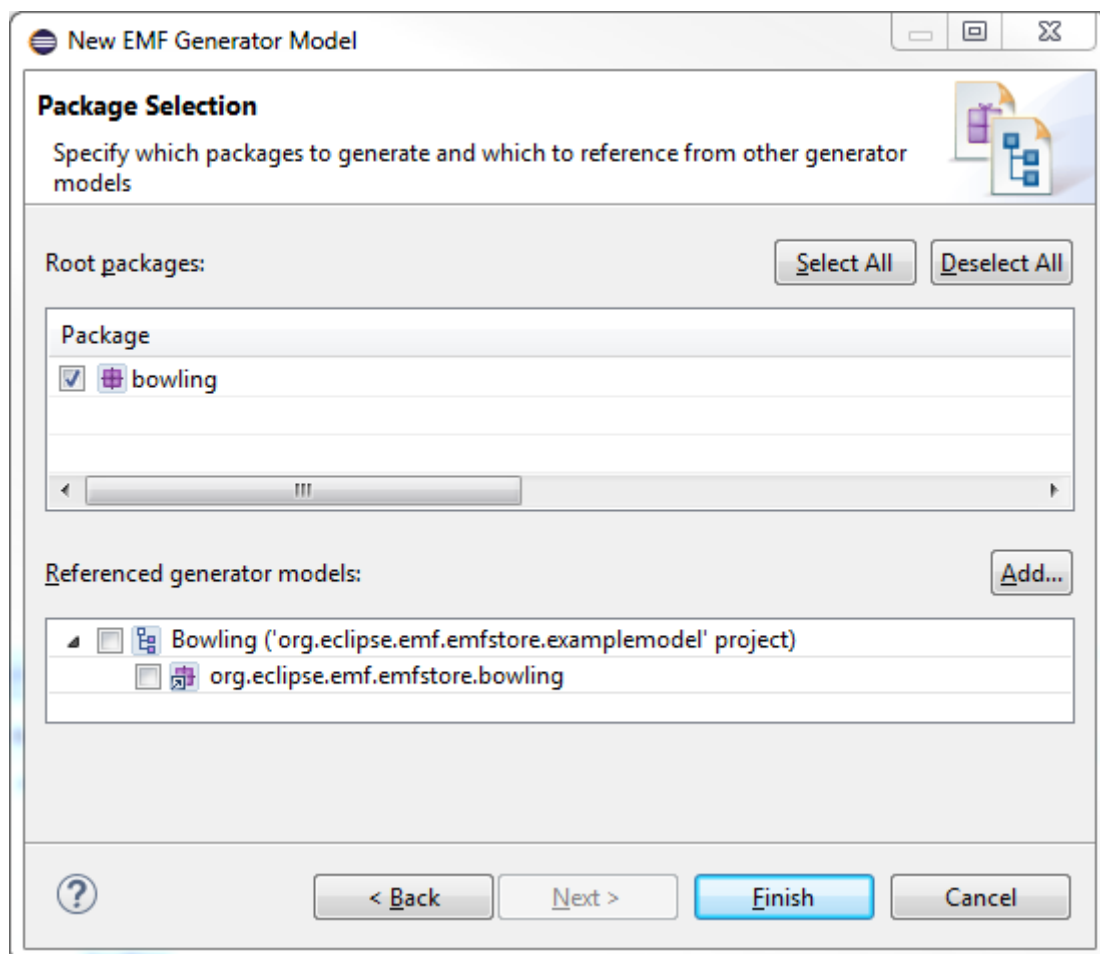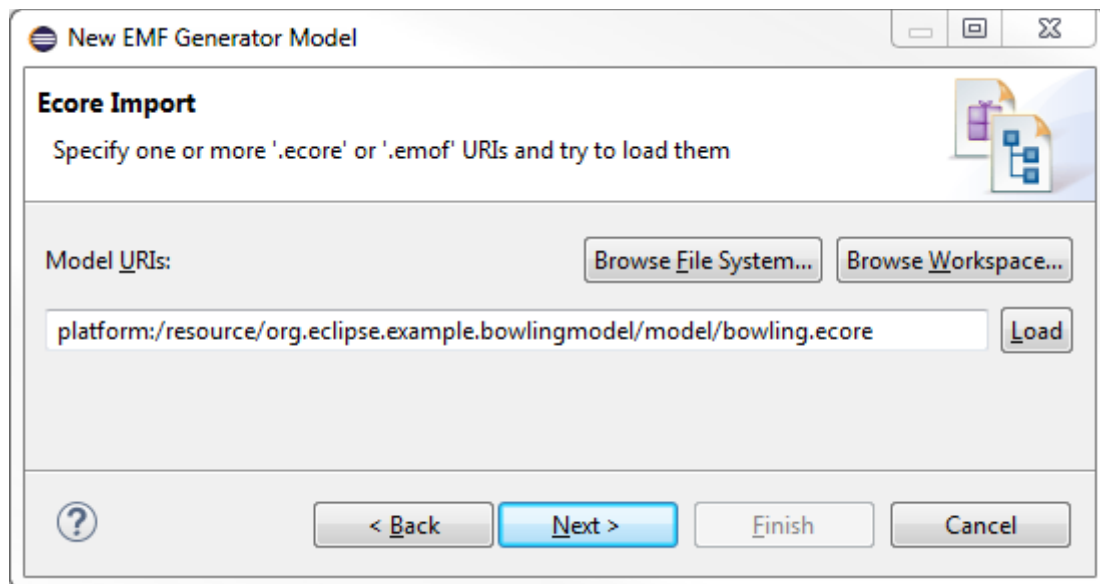
Code Generation

In this step, we will generate the entities from the Ecore file we have created. Note that if you need to change your model, you will be able to regenerate the entities again. EMF can deal with simple changes like adding model elements or EAttributes. If you have complex changes, like moving an attribute to another class, you will have to migrate existing instances of the model. This is supported by the EDAPT framework. (see http://www.eclipse.org/edapt/)

To generate entities, we first have to create a generator model. This allows you to configure properties for the code generation that are not part of the model itself. For example, source code is generated for the plugin and subfolder as well.

Right click the model folder in the project → "New" → "Other..." → "EMF Generator Model" → "Next" and enter bowling.genmodel as the file name. Proceed to the next page and select "Ecore model" as the model importer. After clicking "Next", select "Browse Workspace..." and select our previously created bowling.ecore. Go the next wizard page and select "Finish".

Copyright 2011-2014 EclipseSource
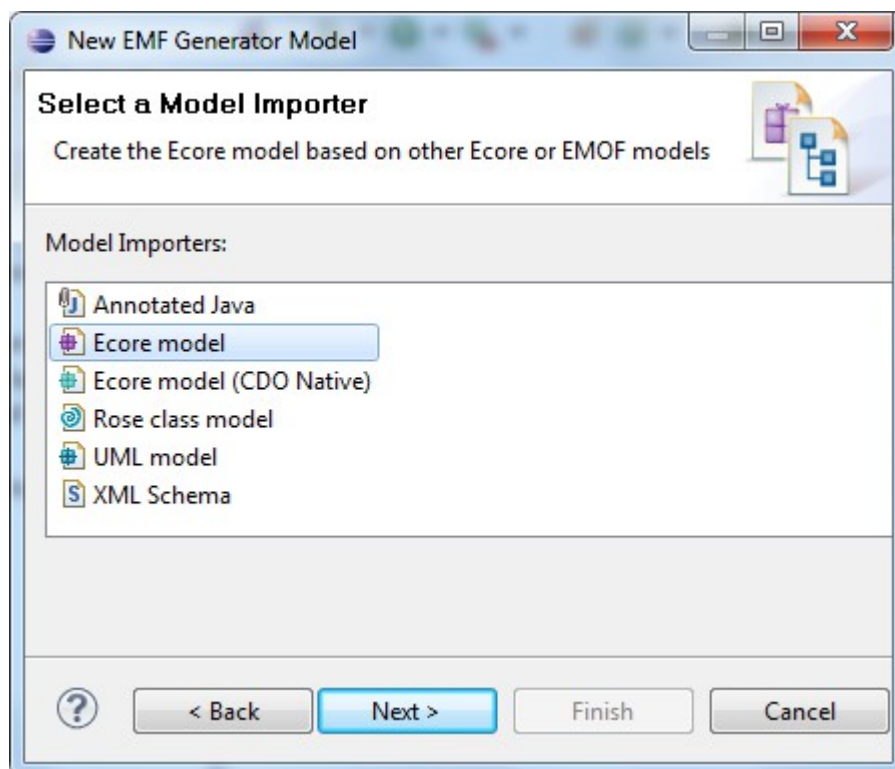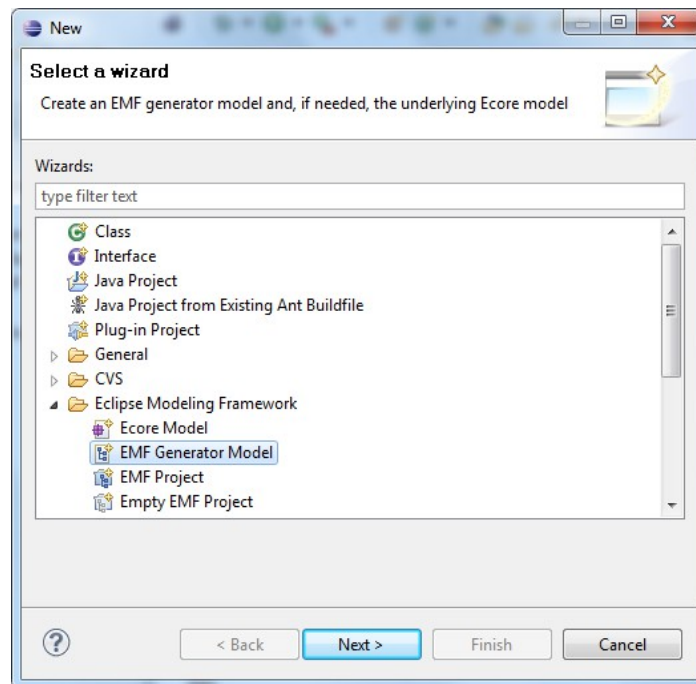
In the root node of the generator model, you can set the properties for generating code. In the tree of the generator model, we can set properties for every generated entity. For the first code generation, we'll use the default settings. Based on the generator model, we can now generate the source code. EMF allows you to generate a maximum of four different plugins for a defined model:

- **Model**: The model contains all entities, packages and factories to create instances of the model.
- **Edit**: The edit plugin contains providers to display a model in a UI. For example, the providers offer a label for every model element, which can be used to display an entity showing an icon and a name.
- **Editor**: The editor plugin is a generated example editor to create and modify instances of a model.
- **Test**: The test plugin contains templates to write tests for a model.

To generate the plugins, right-click on the root node of the generator model and select the plugin. For our tutorial, please select "generate all".

Before we look at the generated code, let's start the application and create an entity of our model. Right click on the plugin containing the ecore file and select "Debug as → Eclipse Application". This will start a new runtime Eclipse.
Then, in the runtime, create a new empty project (Toolbar menu → "File" → "New" → "Other..." → "General" → "Project") named bowlinginstance.
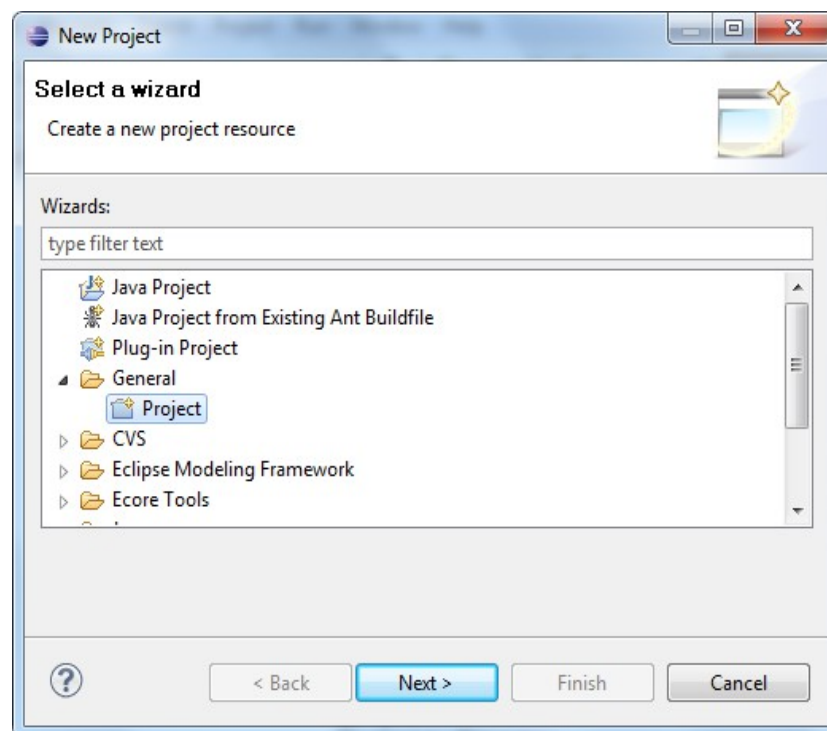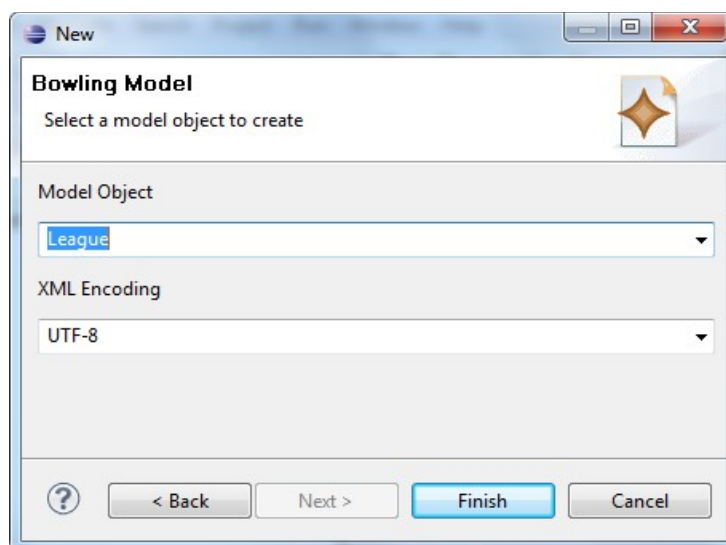
Right click the created project → "New" → "Other..." → "Example EMF Model Creation Wizards" → "Bowling Model" → "Next" and enter league.bowling as the name. This file will contain a serialized version of our model instance.
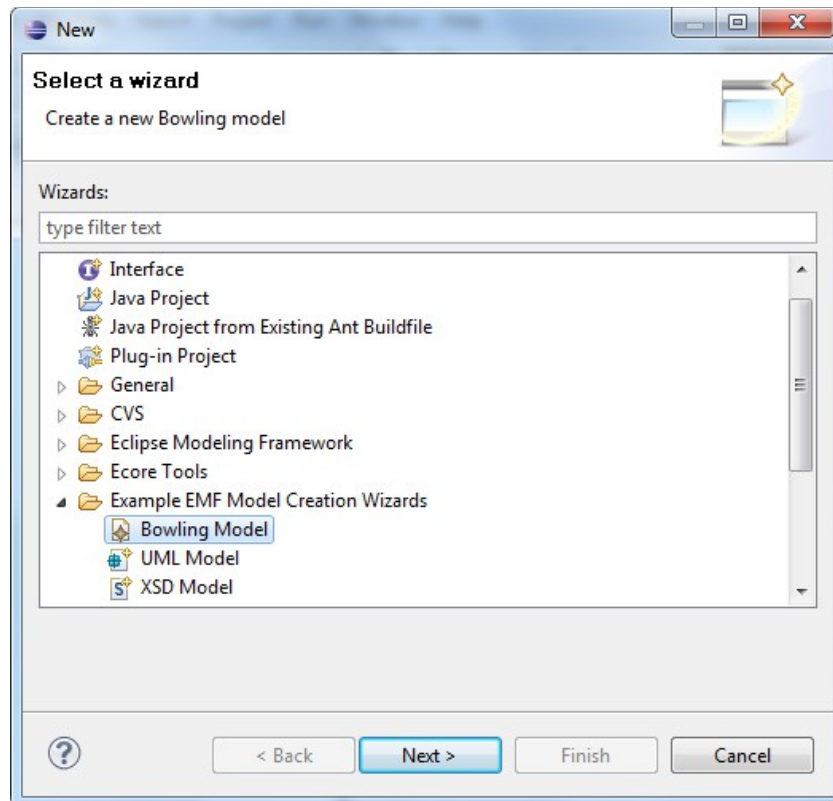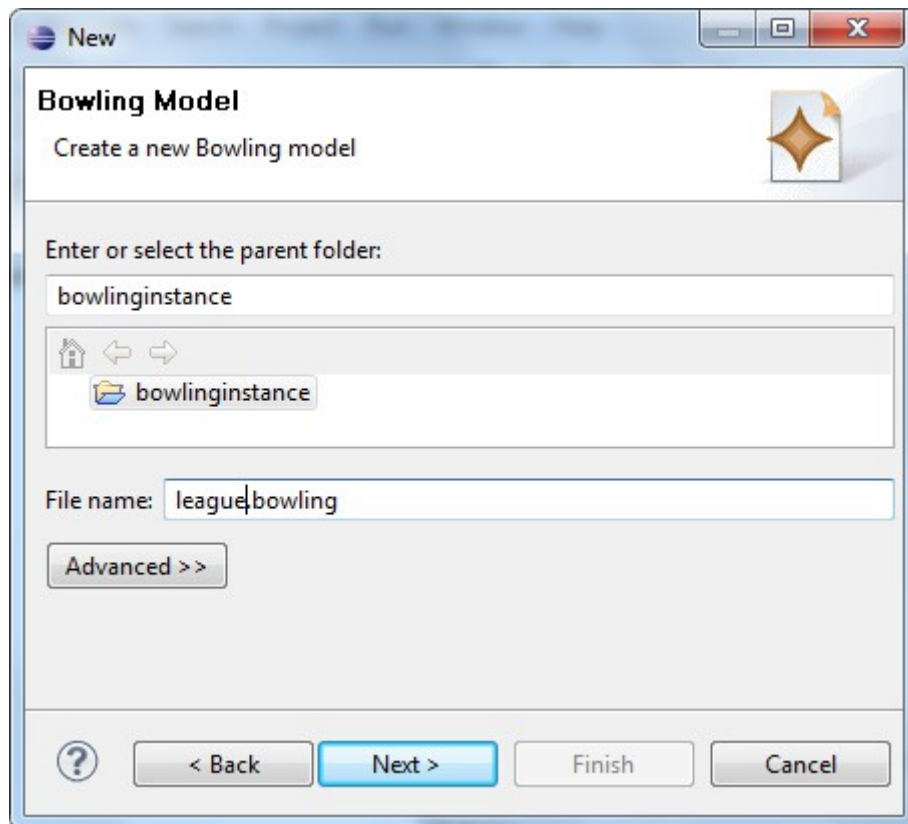
Select League as the model object. This sets the root object of the model instance we are about to create.

The generated editor for model instances works similarly to the Ecore editor. Model element instances can be created via a right-click and EAttributes can be modified in the properties view. Please give the League a name and create two Players. On save, all created instances are serialized in the XMI file "league.bowling".
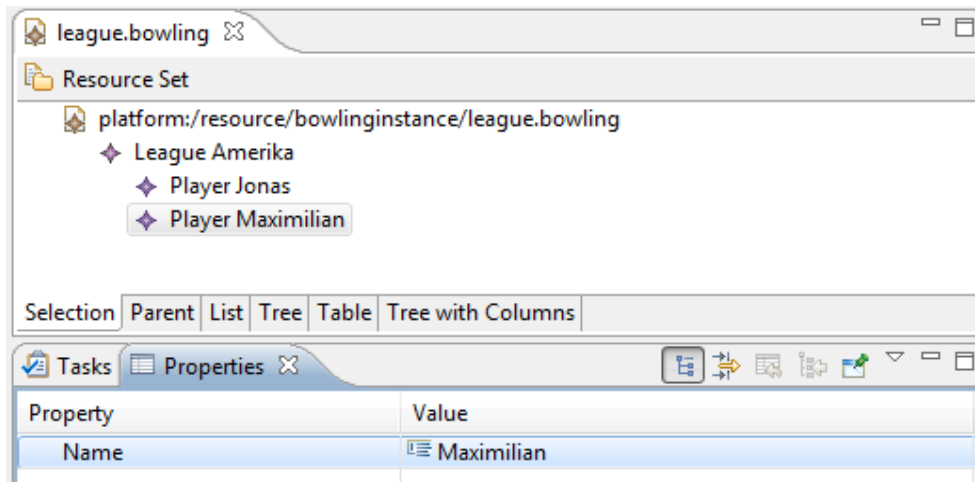

Model Refinement

Let's switch back to our IDE Eclipse environment, complete the model and regenerate the source code. In this second model iteration, we will add different type of EReferences as well as EEnums and Multi-EAttributes. First, add the following classes to the bowling model:

- Tournament
- Matchup
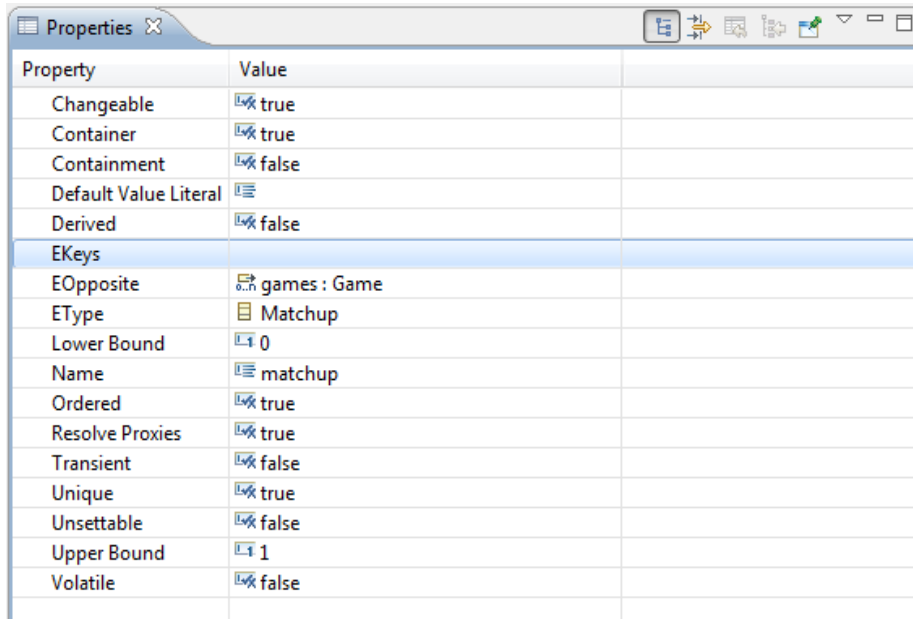- Game

These classes model the results of bowling tournaments and build up a second tree in our model. Therefore, we add containment EReferences from Tournament to Matchup and from Matchup to Game. Following the bowling rules, a Matchup consists of two Games (each from one Player). We model this by setting the upper bound and lower bound of the EReference "games" of the EClass Matchup to "2".
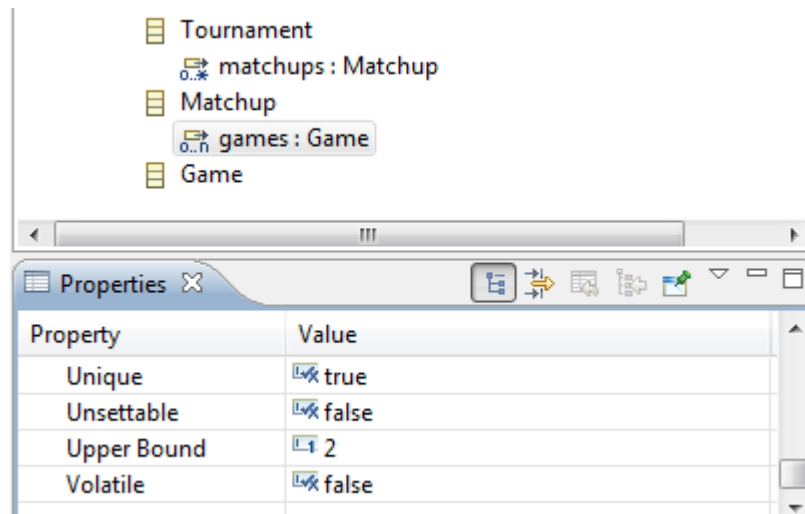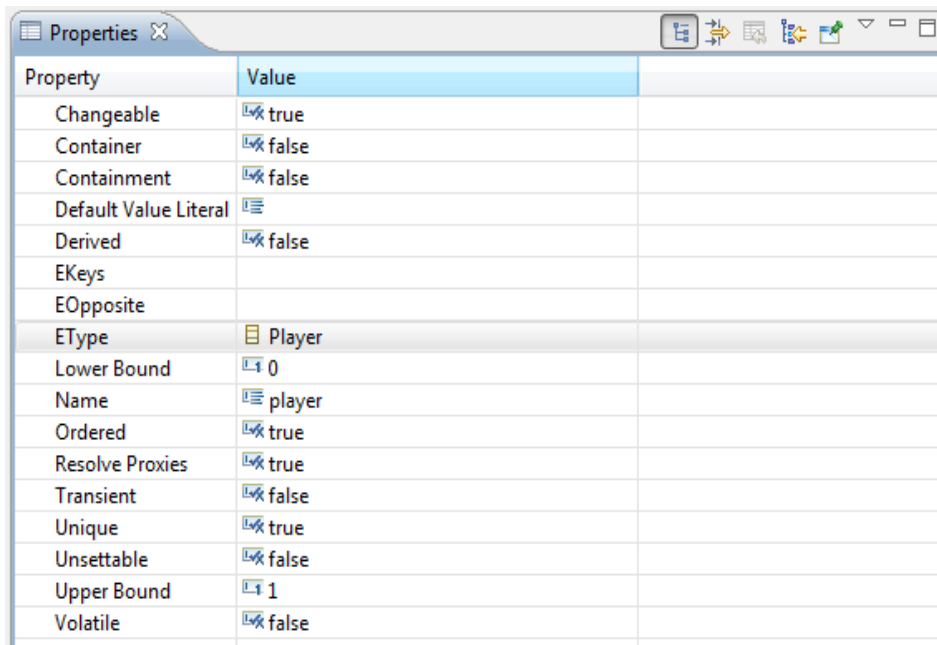
We defined the EReference between Matchup and Game as bi-directional. This means that the reference can be navigated from both ends. Therefore we have to create a second EReference from Game to Matchup and bind both EReferences. EMF will take care of the bi-directional synchronization. In other words, adding a Matchup to a Game will automatically add the Game to the Matchup.

Please add an EReference to Game called "matchup" with the EType "Matchup". By setting the EOpposite to the EReference "games", both EReferences are coupled bi-directionally. Note that the property "Container" will automatically be set to True.

| Property	Value	
Changeable	true	
Container	true	
Containment	false	
Default Value Literal		
Derived	false	
EKeys		
EOpposite	games : Game	
EType	Matchup	
Lower Bound	0	
Name	matchup	
Ordered	true	
Resolve Proxies	true	
Transient	false	
Unique	true	
Unsettable	false	
Upper Bound	1	
Volatile	false	

The next step is to add a cross-EReference. In contrast to containment EReferences, cross-referenced model elements do not contain each other. In our model, we add a cross-referencing EReference from Game to Player named "player". Set both container and containment properties to "false". An arbitrary number of games can be assigned to a Player now and the Player is still contained in a League.

Property	Value	
Changeable	true	
Container	false	
Containment	false	
Default Value Literal		
Derived	false	
EKeys		
EOpposite		
EType	Player	
Lower Bound	0	
Name	player	
Ordered	true	
Resolve Proxies	true	
Transient	false	
Unique	true	
Unsettable	false	
Upper Bound	1	
Volatile	false	

As a final mandatory step, we will create an EEnumeration for the type of Tournament. A Tournament can be of type "Pro" and "Amateur" in our model. Please create an EEnum by right-clicking on the root bowling model package, in the same way we created a class. Add two EEnum Literals to this EEnum.
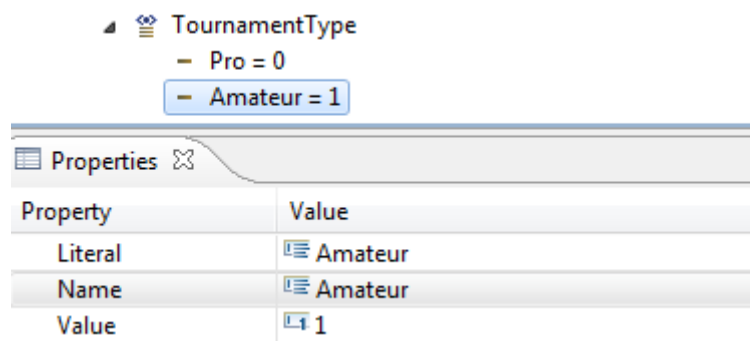
Then, add an EAttribute to the EClass Tournament, name it "type" and set the EType to "TournamentType".

The extended example model contains more EAttributes and EReferences to be added including all basic types and some special cases as the Multi-Integer EAttribute in Game. If you'd like, you can also model the following features:

Player

- height: EDouble
- isProfessional: EBoolean

Game

- frames: EInt, UpperBound = 10

After applying complex changes to the model it is always a good idea to validate it with a right-click on the model root in the Ecore editor. Let's do something wrong in the model and set the lower bound of the EAttribute "games" (in Matchup) to 3. As the upper bound is 2, this model doesn't make too much sense. This will be detected by the model validation - something that is impossible in plain Java code.
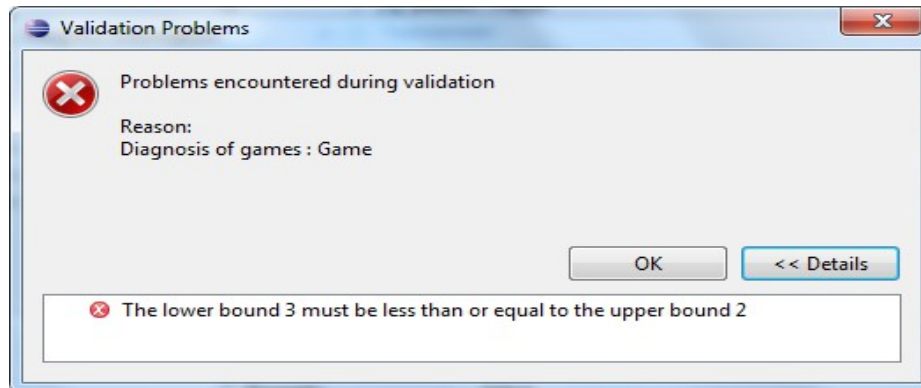
After this model refinement, we will re-generate the code to reflect our changes. Start the runtime Application again and create a second model "tournament". Add a Matchup and two Games. To assign the Games to Players you will have to load the "league" model created earlier. Select "Load Resource" from the menu "Bowling Editor" and select the first model file. Now link the Games to the Players in the properties view.

Why is This Better than Writing POJOs?

You might ask, "Why should I use EMF instead of creating the model by writing plain POJOs?" Without considering benefits like the generated editor for rapid testing and all the additional frameworks available for EMF, let's look at two very simple and exemplary benefits.

Before we look at the generated code (we will do that in a minute), let's consider the amount of code we have just produced. The Eclipse metrics plugin tells us that we have generated over 1,000 LOC, while only 150 are part of utility classes. Even very simple code is considered to be worth $1 per LOC. So, we have earned $1,000 just by clicking some buttons ;)
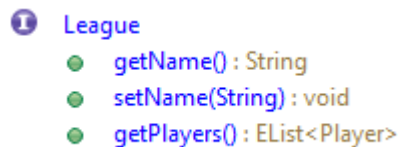

In the next sections, we'll explore the EMF API for the code we have generated.

EMF API

In this part of the tutorial, we will explore EMF's API, including the generated code, as well as EMF's utility classes. Let's have a look at the generated code first.

In the model plugin from our tutorial org.eclipse.example.bowling you will find interfaces and implementations for all of the model's entities. A look at the outline of an entity's interface reveals that it contains getters and setters for the attributes we have defined in the model as well as getters for the references. All entities of the generated EMF model are subclasses of EObject. EObject contains basic functionality – for example, a change notification mechanism.

The model plugin contains factories to create model element entities. Note that the constructor of EObjects is usually not public. Please also note that factories are used by many frameworks for their functionality, e.g., deserialization. Changing these methods successfully requires some careful planning. Let's use the factories to programmatically create some entities and use their APIs to modify them. We will use the pre-generated test plugin to run this example code. If you open the plugin, org.eclipse.example.bowlingmodel.test, you will find a generated test class for all of the entities of your model. By adding methods starting with "test" you can create single test cases. The test cases can be started with a right-click on the test class => "Debug As" => "JUnit Test". Please note that we will not really "test" our model. In this context test cases are just a very simple way of exploring and using the API of the generated classes.

In this very simple example, we'll use the BowlingFactory to create a Matchup and a Game, adding a reference to the Matchup and checking the bi-directional update on the Game.

```java
public void testMatchupGameRef() {
    Matchup matchup = BowlingFactory.eINSTANCE.createMatchup();
    Game game = BowlingFactory.eINSTANCE.createGame();
    matchup.getGames().add(game);
    assertEquals(game.getMatchup(), matchup);
}
```

The super class EObjects offers many methods to access an entity in a more generic way. For example, we will test the containment between Matchup and Game by accessing the EContainer instead of the getMatchup() method.

```java
public void testMatchupGameRef() {
        Matchup matchup = BowlingFactory.eINSTANCE.createMatchup();
        Game game = BowlingFactory.eINSTANCE.createGame();
        matchup.getGames().add(game);
        assertEquals(game.eContainer(), matchup);
}
```

EObjects offer reflective access to their attributes using the methods eSet() and eGet(). This can be useful in modifying an entity in a generic way.

```java
public void testReflection() {
        EObject eObject = BowlingFactory.eINSTANCE.createPlayer();
        eObject.eSet(BowlingPackage.eINSTANCE.getPlayer_Name(), "Jonas");
        Player player = (Player) eObject;
        assertEquals("Jonas", player.getName());
}
```

Information about the available EAttributes and EReferences, as well as all additional concepts we have modeled before, can be accessed either through the EClass or through the EPackage. The following test checks whether the EReference of League is a multiplicity greater than one.

```java
public void testReflectiveInformation() {
        League league = BowlingFactory.eINSTANCE.createLeague();
        assertTrue(league.eClass().getEAllReferences().get(0).isMany());
        assertTrue(BowlingPackage.eINSTANCE.getLeague_Players().isMany());
}
```

EMF also supports the validation of model instances. For example, we can validate the model's constraint that a matchup must always consist of two games.

```java
public void testValidation() {
        Matchup matchup = BowlingFactory.eINSTANCE.createMatchup();
        matchup.getGames().add(BowlingFactory.eINSTANCE.createGame());
        Diagnostic validate = Diagnostician.INSTANCE.validate(matchup);
        assertEquals(Diagnostic.ERROR, validate.getSeverity());
}
```

Finally, EMF provides many utility classes. A very important one is EcoreUtil. It is worthwhile browsing through the available methods of EcoreUtil. We'll use the copy method to create a copy of an EObject.

```java
public void testCopy() {
        Player player = BowlingFactory.eINSTANCE.createPlayer();
        player.setName("Jonas");
        Player copy = EcoreUtil.copy(player);
        assertNotSame(player, copy);
        assertEquals(player.getName(), copy.getName());
}
```

Import Intermediate Sample Solution

Before we continue with the tutorial, please import the intermediate sample solution, which can be downloaded here.
Switch to an empty workspace (File → Switch Workspace) and select "Import" → "General" → "Existing Projects into Workspace". Select "EMFTutorialIntermediateSolution.zip" and import all projects.
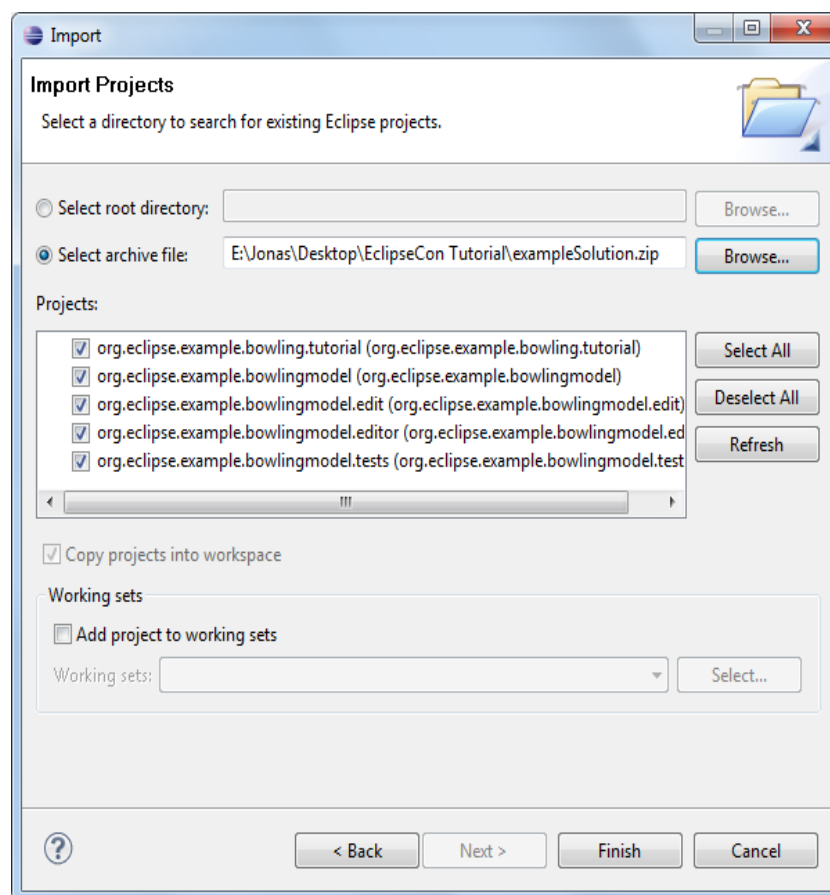
AdapterFactories

For the next sections of the tutorial, it is important to understand the concept of AdapterFactories. We will give a basic introduction. More advanced concepts are also described here.

The basic function of AdapterFactories is to provide you with the interface you need for a certain purpose such as an ILabelProvider needed in the UI. EMF generates a lot of these classes for you. To retrieve the right class, you can use an AdapterFactory implementation of the interface you need, e.g., an AdapterFactoryLabelProvider.

The AdapterFactoryLabelProvider will retrieve the generated LabelProviders for all EObjects using an AdapterFactory.
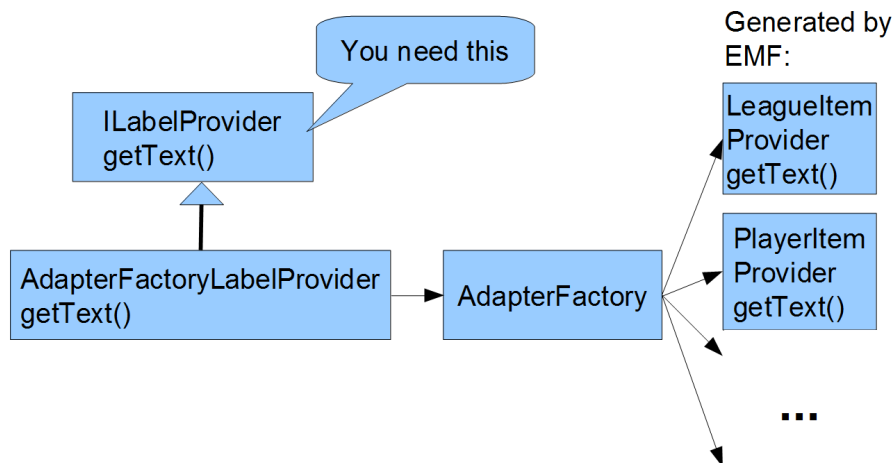
EMF Data Management

In the previous sections we have shown how to generate a structured data model with EMF. In a typical application, these data models have to be stored and likely also versioned and distributed. There are a couple of frameworks that support different use cases.

By default EMF provides the ability to serialize EObjects to XMI files. In the following example, we will load EObjects from a file and save them afterwards. EMF also offers commands to make modifications to the model. Commands can be easily undone. In the example, we will load an XMI file containing a Tournament. We can add new Matchups to that Tournament and undo these changes. When we're finished, we can save the changes back to the file.

For the tutorial, we have prepared an example dialog in the plugin org.eclipse.example.bowling.tutorial, which has been imported from the sample solution. You can open this dialog by right-clicking a file containing instances of a bowling model and select "Tutorial" → "Open Tournament Example Dialog". After implementing the next two sections of the tutorial, it will look like this:
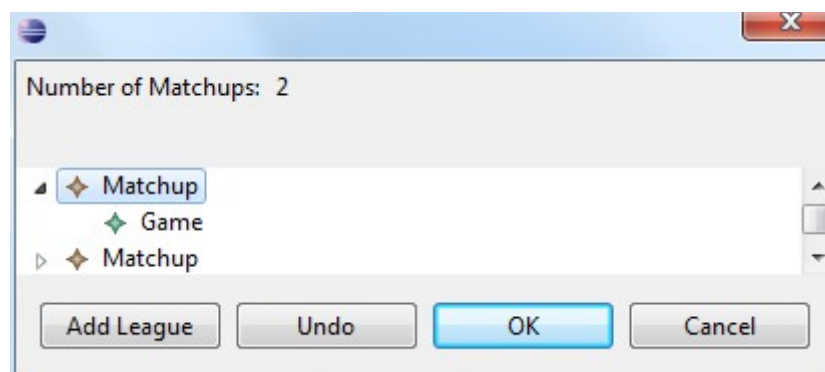
In the subclass ExampleTournamentDialog, there are empty method stubs to be implemented in this tutorial. Just a note here that for the purposes of the tutorial we have focused on simplicity over perfect design. Also, everything that is not relevant for the tutorial is implemented in an abstract base class called AbstractTournamentExampleDialog.

Now you'll need to open the class ExampleTournamentDialog. We will implement the loadContent method, which is triggered by opening the example view. The purpose of this method is to get a Tournament from the file that is then displayed in the example view. To keep it simple, we assume that the file contains a Tournament and this Tournament is the first element in the file. You can easily create a file like this with the generated example editor.

First, we create an editing domain. An editing domain manages a set of interrelated models and the commands that are run to modify them. For example, it contains the stack of all former commands. An editing domain can create a resource, which is a container for storing EObjects. Resources can be saved and loaded and contents can be added to them. In the example, we get the first EObject in the resource, assume it is a Tournament and make it a member of our superclass.

```java
@Override
protected void loadContent(IFile file) throws IOException {
    // Load Tournament from file and set it with setTournament
    AdapterFactoryEditingDomain domain = new AdapterFactoryEditingDomain(
                    getAdapterFactory(),
                    new BasicCommandStack());
    resource = domain.createResource(file.getFullPath().toString());
    resource.load(null);
    EObject eObject = resource.getContents().get(0);
    setTournament((Tournament) eObject);
}
```

After loading the content, we will implement a save. This will be triggered by pressing OK in the dialog and will serialize the model and apply all changes to the file.

```java
@Override
protected void save() throws IOException {
        // save changes in the file
        resource.save(null);
}
```

Now we want to implement the addition of a Matchup to a Tournament. We will use a command for this. First we create a Matchup using the appropriate factory. The factory, by convention, has the same name as the base package of the model. Then we create a command which adds the newly created Matchup to the Tournament that has been loaded from the resource in the previous step. Finally, we run the command on the command stack of the editing domain.

```java
@Override
protected void addMatchup() {
  // add a new Matchup using a Command
  Matchup matchup = BowlingFactory.eINSTANCE.createMatchup();
  EditingDomain editingDomain = AdapterFactoryEditingDomain
                                .getEditingDomainFor(getTournament());
  Command command = AddCommand.create(editingDomain, getTournament(),
                    BowlingPackage.eINSTANCE.getTournament_Matchups(),
                    matchup);
  editingDomain.getCommandStack().execute(command);
}
```

At this point, the changes will not be reflected in the dialog's UI, but we will implement the code in the next section of the tutorial.

The next step is to implement undo. To undo the last command, all you'll need to do is to call

undo on the command stack of the editing domain.

```java
@Override
protected void undo() {
    // Undo the last change
    AdapterFactoryEditingDomain
        .getEditingDomainFor(getTournament())
        .getCommandStack().undo();
}
```

Now, start the bowling application and create an XMI file with the example editor. It should contain a Tournament and several Matchups and Games. Right click on the file and select "Tutorial" → "Open Example Tournament View". In this view, you can add new Tournaments, undo this operation and save by clicking on "OK". You can validate the result by opening the file in the Ecore editor. Please note again that the UI of the View will not be updated yet, but we will initialize the UI in the next step of the tutorial.

Additional Persistence Frameworks

There are several frameworks for storing and versioning EMF model instances. Here are three that we can recommend:

- EMFStore (Model Repository)
- CDO (Model Repository)
- Teneo (Database Back-end)
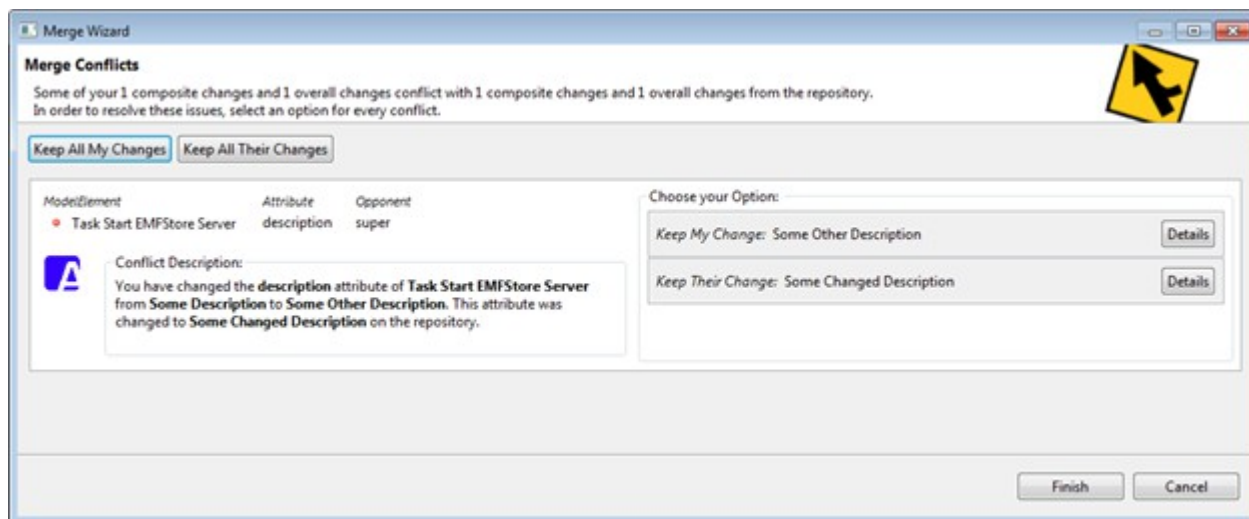
EMFStore Merge Dialog

EMF UI

In this section, we will fill the example view with two basic UI elements. First, we will bind the Label on top showing the number of Matchups in the opened Tournament to the model. We will use the notification mechanism to update the Label whenever the number of Matchups changes. Second, we will fill the TreeViewer with a list of the Matchups and display their Games as children. To update, the Label we will register a listener on the Tournament EObject, which is opened in the view. This listener will always be notified by the EMF runtime if there is a change in the Tournament EObject.

```
@Override
protected void initializeListener() {
   // initialize a listener for the Label displaying the number of Matchups
   numberOfMatchupListener = new NumberofMatchupListener();
   getTournament().eAdapters().add(numberOfMatchupListener);
}
```

In the second step, we will implement the listener itself. In the notify method, we check first whether the change was on the EReference to Matchups and consequently influenced the number of Matchups. If this is the case, we update the Label via the updateNumberOfMatchups method (implemented in the AbstractTournamentExampleView).

```
private final class NumberofMatchupListener extends AdapterImpl {
   // Implement a listener to update the Label. Call updateNumberOfMatchups
   @Override
   public void notifyChanged(Notification msg) {
     if (msg.getFeature().equals(
       BowlingPackage.eINSTANCE.getTournament_Matchups())) {
       updateNumberOfMatchups();
     }

     super.notifyChanged(msg);
   }
}
```

This is how you would manually implement listeners. To create UIs with bi-directional updates between UI elements and data-models, we recommend using data binding that is already available for EMF. In Eclipse data-binding, you can bind a certain UI element to a certain EAttribute or EReference, and it will take care of bi-directional updates.

Next, we will initialize the TreeViewer to display the Matchups of the current Tournament and their Games as children. A TreeViewer needs three things to be initialized: the ContentProvider, the LabelProvider and an input. The ContentProvider defines the structure of the Tree by providing the method getChildren(). The LabelProvider is called to get an icon and the text to be displayed for one node. The input of a TreeViewer is the invisible root element of the Tree. The elements displayed in the root of the tree are the children of that element. In our case, the input is the Tournament.
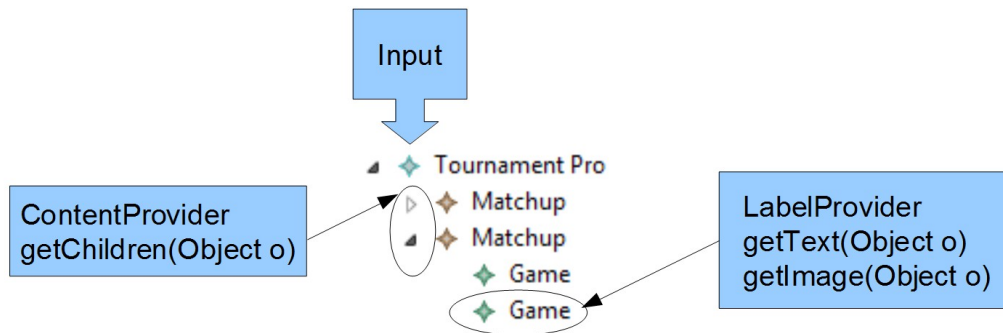
ContentProvider and especially LabelProvider usually depend on a certain EClass. EMF generates providers for several purposes including Content- and LabelProvider. We will use the AdapterFactory concept explained previously to retrieve the right provider for every element. Finally, we set the Input to the Tournament that is currently open.

```
@Override
protected void initializeTreeviewer(TreeViewer treeViewer) {
    // initialize a TreeViewer to show the Matchups
    // and Games of the opened Tournament
    AdapterFactoryLabelProvider labelProvider =
new AdapterFactoryLabelProvider(
getAdapterFactory());
    AdapterFactoryContentProvider contentProvider =
new AdapterFactoryContentProvider(
getAdapterFactory());

    treeViewer.setLabelProvider(labelProvider);
    treeViewer.setContentProvider(contentProvider);
    treeViewer.setInput(getTournament());
}
```

To test the UI features just implemented, you'll need to re-start the bowling example application. To modify the appearance of a Label you can simply modify the ItemProvider of the respective class. Let's modify the LabelProvider for Matchup. To modify the appearance of EObjects in the TreeViewer, you can adapt the generated ItemProvider MatchupItemProvider. We will show the number of Games contained in a Matchup Game in the example. Mark the method as "generated NOT" to prevent it from being overwritten during the next generation.

```
/**
 * This returns the label text for the adapted class.
 * <!-- begin-user-doc
 * --> <!-- end-user-doc -->
 * @generated NOT
 */
@Override
public String getText(Object object) {
```

```
        if (object instanceof Matchup) {

                EList<Game> games = ((Matchup) object).getGames();

                if (games != null) {
                        return "Matchup, Games: " + games.size();
                }
        }

        return getString("_UI_Matchup_type");
}
```
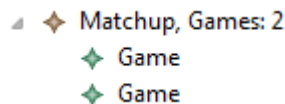
In the running application, the new LabelProvider is displayed in the Tournament Example view as well as in the Ecore Editor:


As a last step, you should remove all listeners when closing the view. Note that LabelProvider and ContentProvider are registered listeners on the model, so you should delete them as well.

If you want to have a look at the final sample solution, please import the projects from this link.

Additional UI Frameworks

There are several frameworks for displaying data from an EMF model instance. If you want to create an application similar to the screenshot below, you should definitely have a look at the EMF Client Platform (Tutorial).
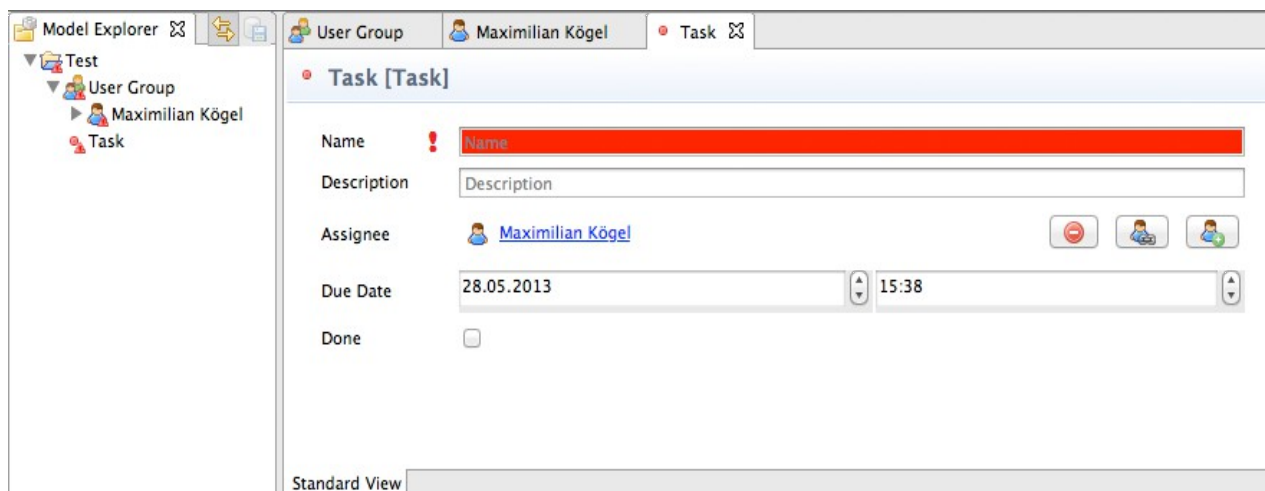
EMF Client Platform Navigator and Editor

Additional frameworks that are worthwhile having a look at for creating UIs are:
- [Graphiti](#) (Graphical Editor)
- [Graphical Modeling Framework](#) (Graphical Editor)
- [Extended Editing Framework](#) (Advanced Property View)

Additional EMF-based Technologies

In this last section of the tutorial we'd like to give you our short list of additional EMF-based technologies for you to explore:

- [EMF Compare](#) for comparing models
- [EDAPT](#) for migrating models
- [EMF IncQuery](#) for querying models
- [XText](#) for textual modeling (DSLs)

Conclusion

We hope you found this tutorial helpful. If you have feedback or questions, please feel free to contact us at modeling@eclipsesource.com. Updates of this tutorial will become available [here](#).