Tutorial: Getting started with EMF

Abstract

The Eclipse Modeling Framework (EMF) allows generating the entity classes of an application based on a model. Such entity-centric applications often follow a common template. They provide a tree-based view for navigation and for browsing all entities, a detailed editor for one entity of the model and a number of additional customized views. Additionally the entities have to be persisted and often shared among multiple clients. In this tutorial we show step by step how to build this kind of application. As an example we build an application to manage a conference, including authors and submissions. We start with creating the underlying Ecore model and generating the entity classes using EMF. In a second step we show how to build a custom UI using the EMF Client platform. In a third step we connect our application to the EMFStore repository allowing to share model instances among distributed clients.

1. Introduction

This tutorial consists of 3 parts:

- EMF Modeling (1 hour)
- Customizing the UI (1 hour)
- Distributing the Model (1 hour)

Additionally we plan 1 hour for preparation and as buffer. All parts are organized in steps. Later steps can be skipped to stay in time. For each part, we provide a sample solution. The following part builds upon this solution. Therefore you can step into coding at the beginning of each part. We kindly ask you to prepare your laptop, if you plan to join coding to keep the set-up time as short as possible. Please have a look at the following website for the latest updates:

http://code.google.com/p/unicase/wiki/EMFGettingStarted

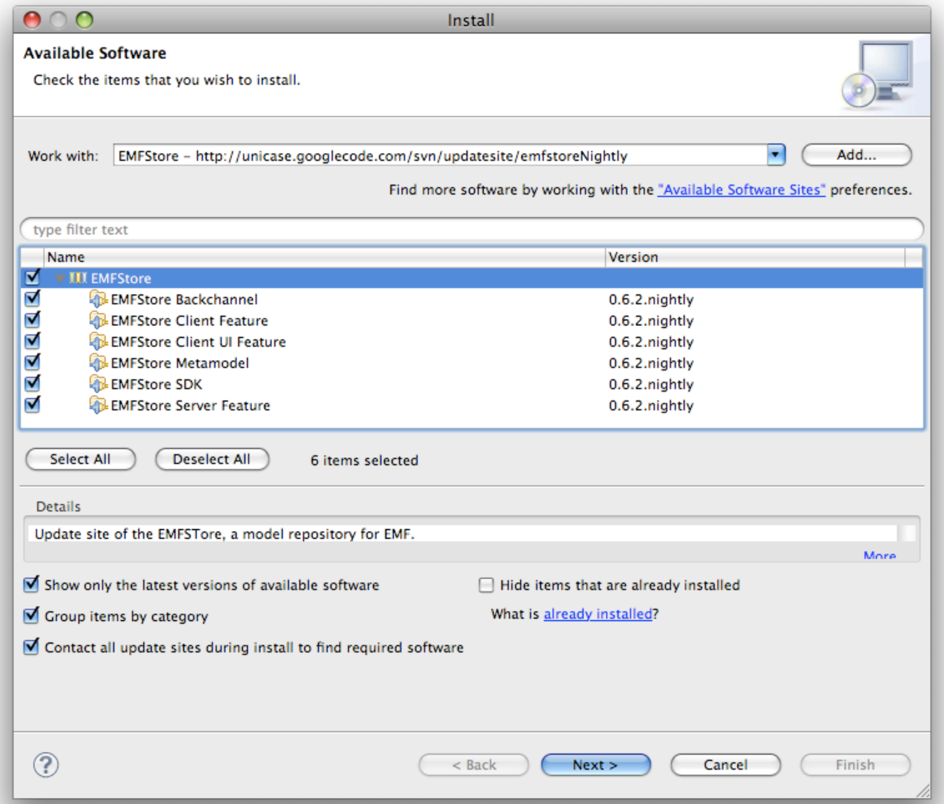
2. Preparation

Download Eclipse Helios Modeling Edition:

http://www.eclipse.org/downloads/packages/eclipse-modeling-tools-includes-incubating-components/heliosr

Install all components from the latest EMFStore Release (including EMF Client Platform):

http://unicase.googlecode.com/svn/updatesite/emfstoreNightly



Install

Available Software

Check the items that you wish to install.

Work with: EMFStore - http://unicase.googlecode.com/svn/updatesite/emfstoreNightly

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☑ EMFStore	
☑ EMFStore Backchannel	0.6.2.nightly
☑ EMFStore Client Feature	0.6.2.nightly
☑ EMFStore Client UI Feature	0.6.2.nightly
☑ EMFStore Metamodel	0.6.2.nightly
☑ EMFStore SDK	0.6.2.nightly
☑ EMFStore Server Feature	0.6.2.nightly

6 items selected

Details

Update site of the EMFSTore, a model repository for EMF.

☑ Show only the latest versions of available software
☐ Hide items that are already installed
☑ Group items by category
What is already installed?
☑ Contact all update sites during install to find required software

3. EMF Modeling

In the first step of the tutorial we will create the underlying model of our example application. The model consists of four entities as shown in Figure 1. An author list contains several authors. An author can have a name, a mail address and can either be a member or not. A conference consists of submissions. The have a title, a description, a length and a predefined type. A submission has one first author and can have several co-authors.

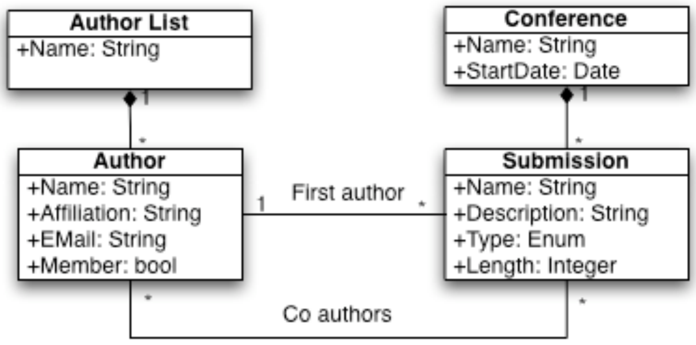
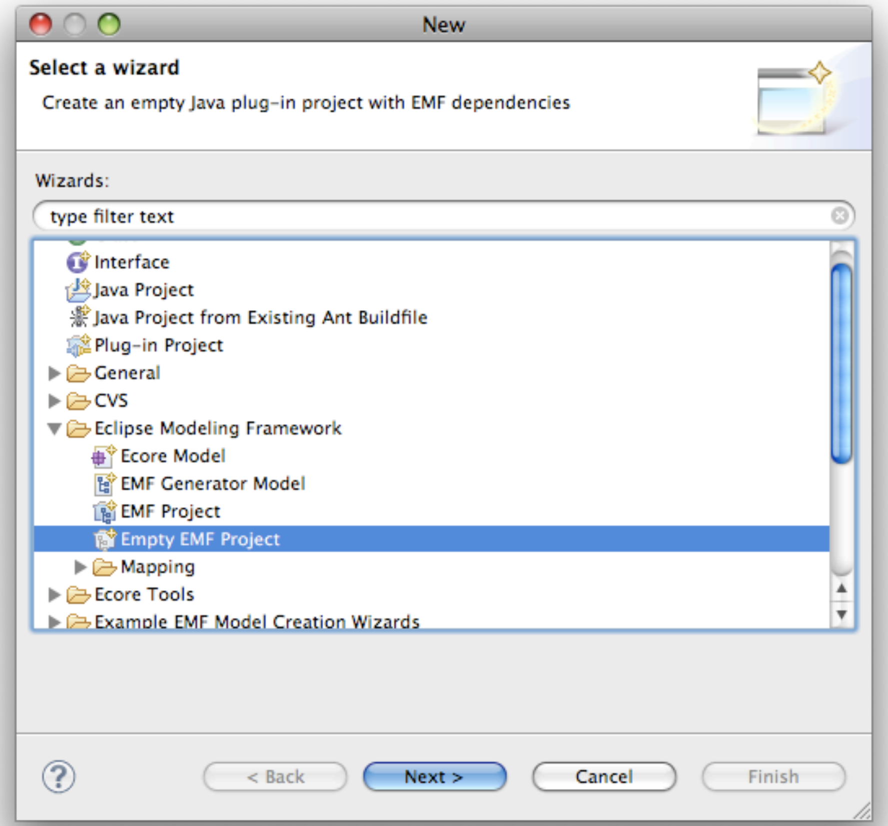


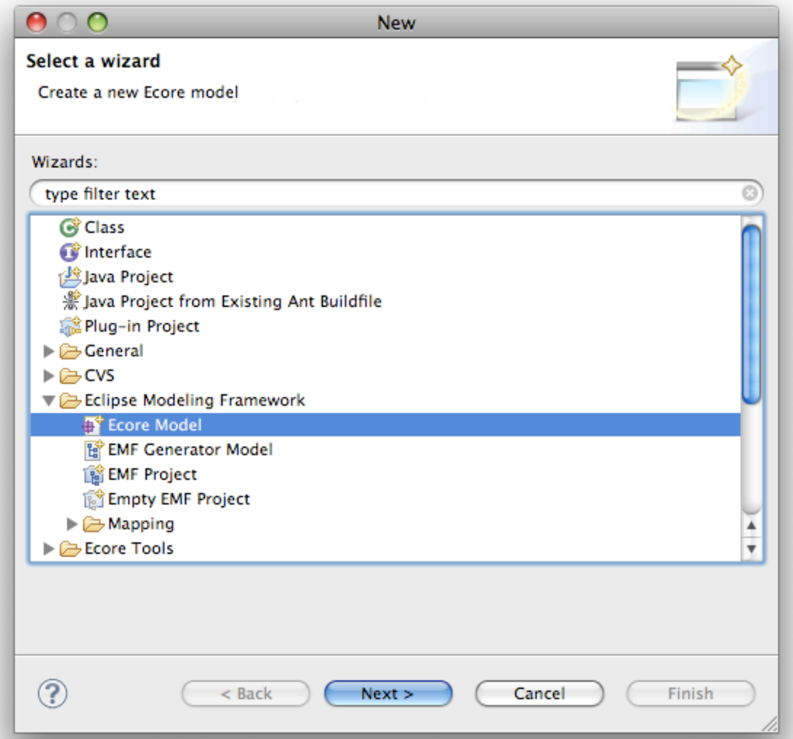
Figure 1: Model of the example application

Creating the Model

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:



The essential part of the modeling project is the model itself, defined in the format “Ecore”. Please create a new Ecore file in the model folder in your new modeling project. We will name the model “submissionsystem.ecore”:


New

Select a wizard

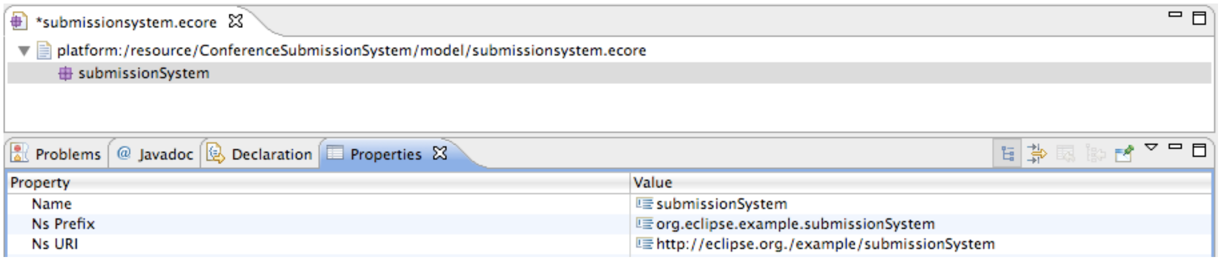
Create a new Ecore model

Wizards:

type filter text

- Class
- Interface
- Java Project
- Java Project from Existing Ant Buildfile
- Plug-in Project
- ▶ General
- ▶ CVS
- ▼ Eclipse Modeling Framework
  - Ecore Model
  - EMF Generator Model
  - EMF Project
  - Empty EMF Project
  - ▶ Mapping
- ▶ Ecore Tools

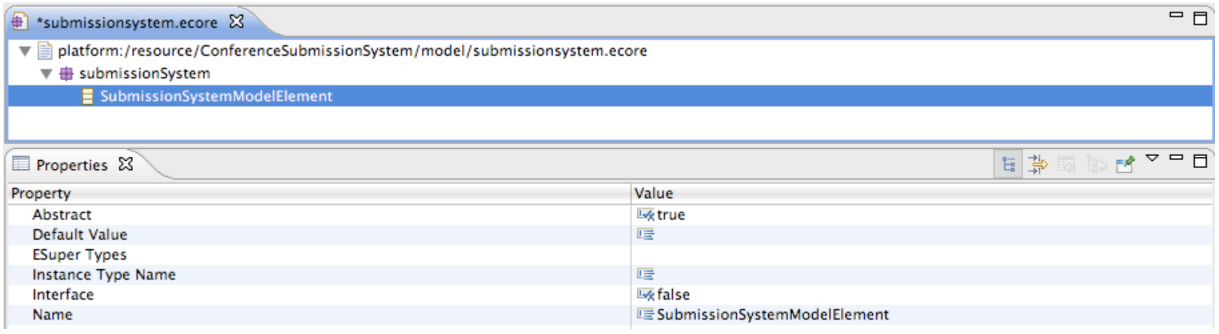
First, give the root package of your new model a name and an URI. This is used to identify the model later on:


*submissionsystem.ecore

platform:/resource/ConferenceSubmissionSystem/model/submissionsystem.ecore
submissionSystem

Property	Value
Name	submissionSystem
Ns Prefix	org.eclipse.example.submissionSystem
Ns URI	http://eclipse.org./example/submissionSystem

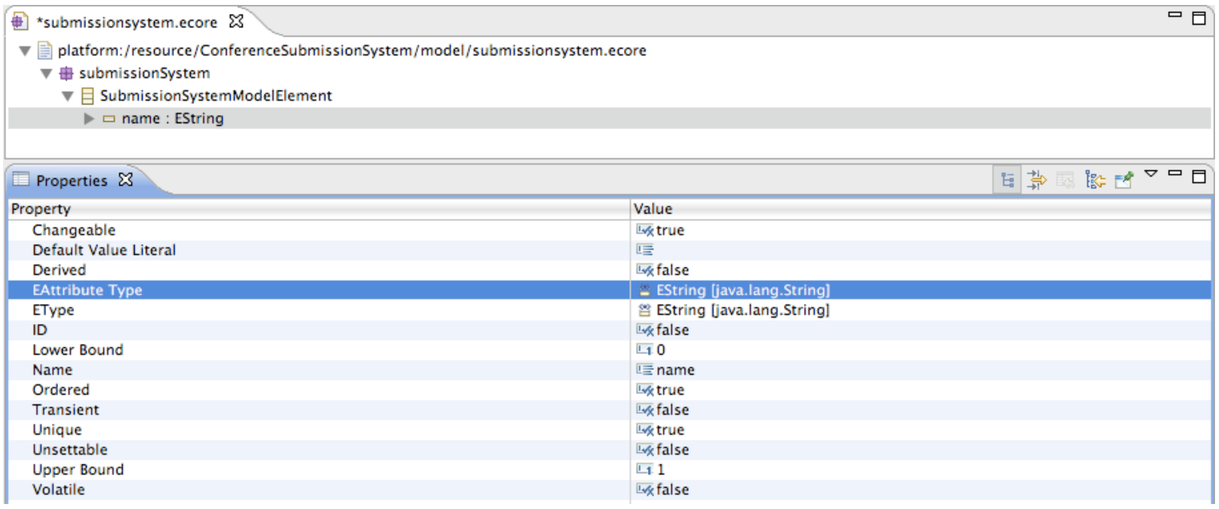
Now, we can define our model elements as children of the root package. As a first step, we will introduce a common base class for all model elements. This is not a mandatory step, but it is useful as you might define a common behavior for all entities in your application. In our example all model elements have an attribute "name". Please right click on the root package and create a new EClass. We will call the common base class "SubmissionSystemModelElement". Further, we define this class to be abstract, as only the subtypes can be instantiated:


*submissionsystem.ecore

platform:/resource/ConferenceSubmissionSystem/model/submissionsystem.ecore
submissionSystem
SubmissionSystemModelElement

Property	Value
Abstract	true
Default Value	
ESuper Types	
Instance Type Name	
Interface	false
Name	SubmissionSystemModelElement

Now, we can add features to our new model element. In EMF, feature is the super type of attributes and references. Attributes are simple values such as Integers or Strings. References are relationships to other model elements. We will add an EAttribute "name" to our base class by right clicking on it. Set the EAttribute type to EString:



```
*submissionsystem.ecore
▼ platform:/resource/ConferenceSubmissionSystem/model/submissionsystem.ecore
  ▼ submissionSystem
    ▼ SubmissionSystemModelElement
      ▶ name : EString
```

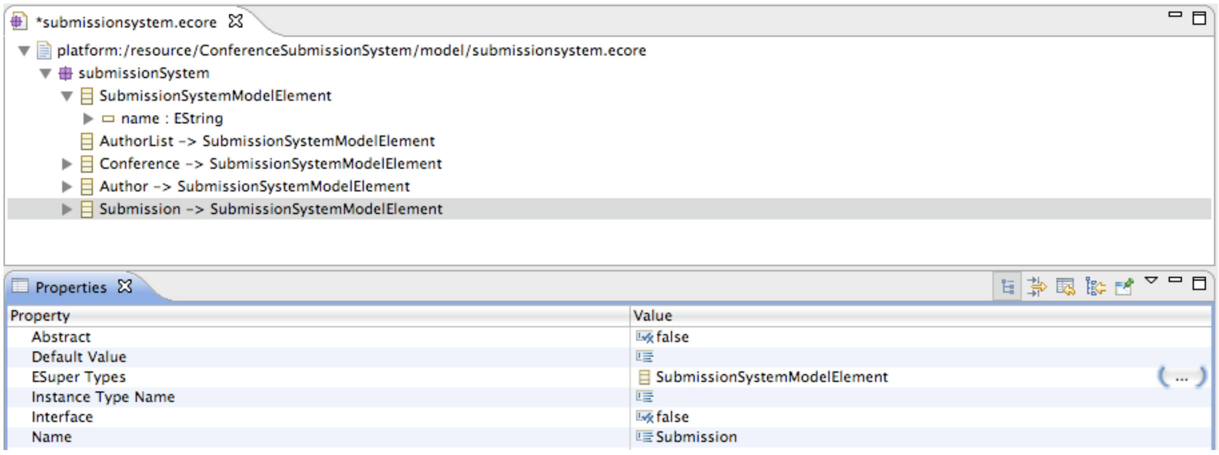
Properties

Property	Value
Changeable	true
Default Value Literal	
Derived	false
EAttribute Type	EString [java.lang.String]
EType	EString [java.lang.String]
ID	false
Lower Bound	0
Name	name
Ordered	true
Transient	false
Unique	true
Unsettable	false
Upper Bound	1
Volatile	false

Let`s have a short look at the other possible options for an EAttribute:

- Transient: Whether the attribute is serialized (e.g. into XML Files).
- ID: Whether the attribute is a globally unique identifier for its model element
- Lower/Upper Bound: How many values can be stored in the attribute. For a String attribute an upper bound greater than 1 will create a List of Strings.

In the next step, we create all entities of our submission system model. As these entities are instantiated in the application, we will not make them abstract. As we use "SubmissionSystemModelElement" as a common base class, all entities will inherit from this class. Please configure this by setting the "ESuper Types" property:

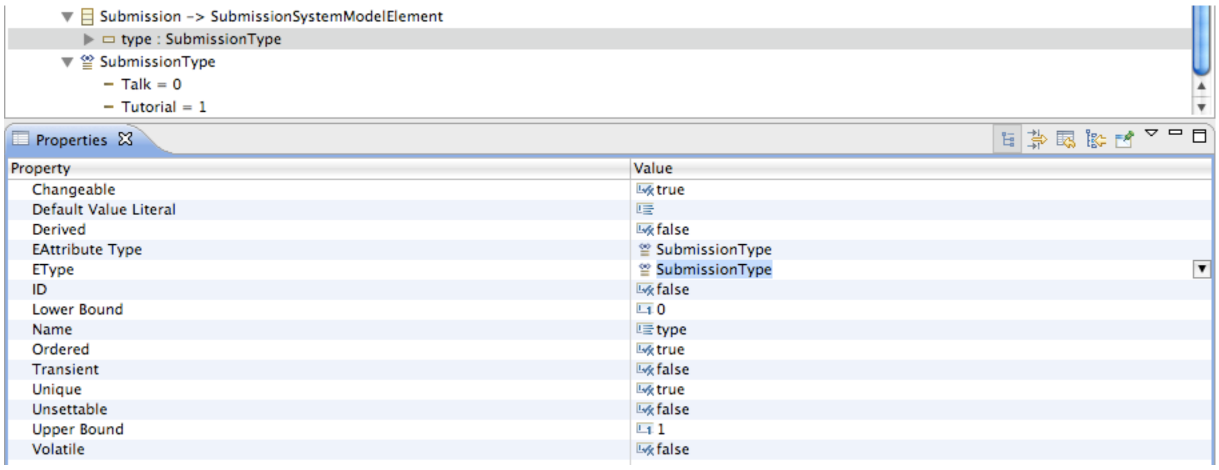


```
*submissionsystem.ecore
▼ platform:/resource/ConferenceSubmissionSystem/model/submissionsystem.ecore
  ▼ submissionSystem
    ▼ SubmissionSystemModelElement
      ▶ name : EString
    AuthorList -> SubmissionSystemModelElement
    ▶ Conference -> SubmissionSystemModelElement
    ▶ Author -> SubmissionSystemModelElement
    ▶ Submission -> SubmissionSystemModelElement
```

Properties

Property	Value
Abstract	false
Default Value	
ESuper Types	SubmissionSystemModelElement
Instance Type Name	
Interface	false
Name	Submission

Please add all simple attributes defined in Figure 1. You can set the type of an EAttribute with the "EType" property. Please note that the name attribute has already be defined in the common super class. For the enumeration attribute of the submission entity, you have to define the enumeration itself as an entity. Please add two Enum Literals to define the type of a submission: "Talk" (value 0) and "Tutorial" (value 1). Then add an EAttribute to the submission entity and set the EType to the created enumeration:



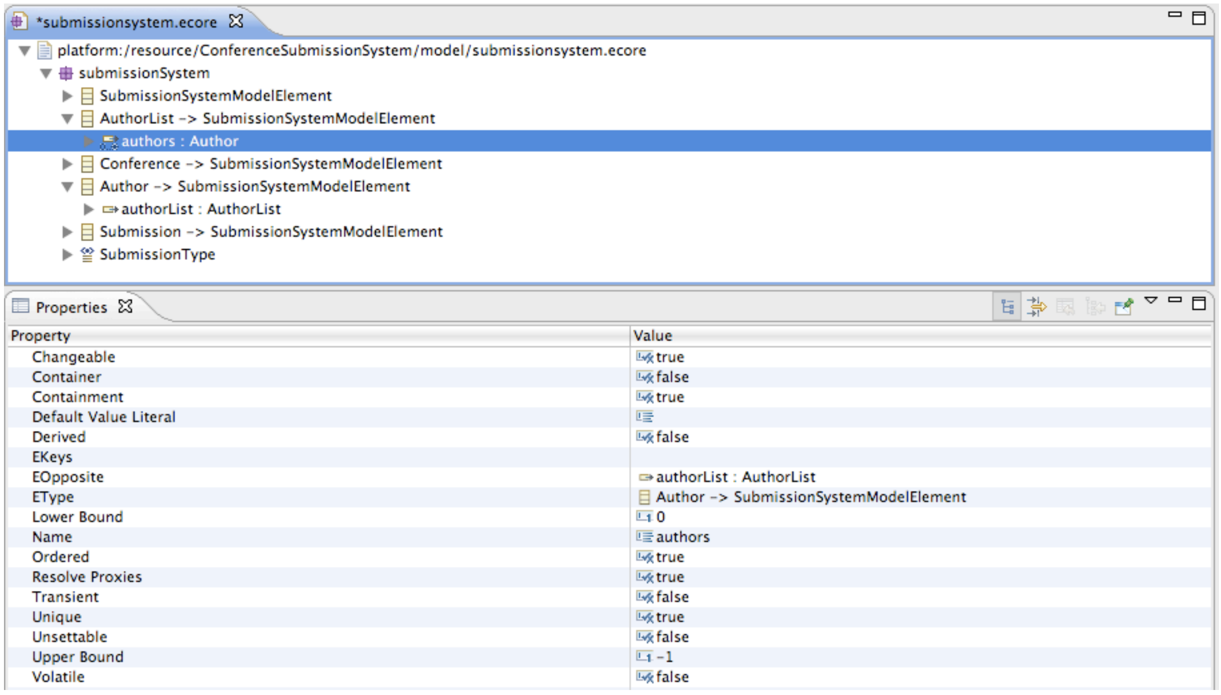
```
▼ Submission -> SubmissionSystemModelElement
  ▶ type : SubmissionType
▼ SubmissionType
    – Talk = 0
    – Tutorial = 1
```

Properties

Property	Value
Changeable	true
Default Value Literal	
Derived	false
EAttribute Type	SubmissionType
EType	SubmissionType
ID	false
Lower Bound	0
Name	type
Ordered	true
Transient	false
Unique	true
Unsettable	false
Upper Bound	1
Volatile	false

As the last step of the model creation we will add the references. References can be unidirectional or bidirectional. To define a unidirectional reference, for example from "author list" to "author", add an EReference to a model element ("author list"). Give it a name ("authors") and select the referenced model element as EType ("author"). To make a reference bidirectional, add a reference to the referenced model element ("author") and select the previously created reference ("authors") as EOpposite. In our example, we will make all references bidirectional.

To configure the multiplicity of references, define the lower and upper bound. An upper bound of "-1" represents a many relationship. Finally, references in EMF can be containment references. This expresses, that one element is contained in another. Every model element can only be contained in one other model element. In our example, the references from author list to author and from conference to submission are defined as containments. To configure this, set the "containment" property of these references to true. Please note that the opposite property "container" at the other end of the reference will automatically be set to true. Please define all references defined in Figure 1.



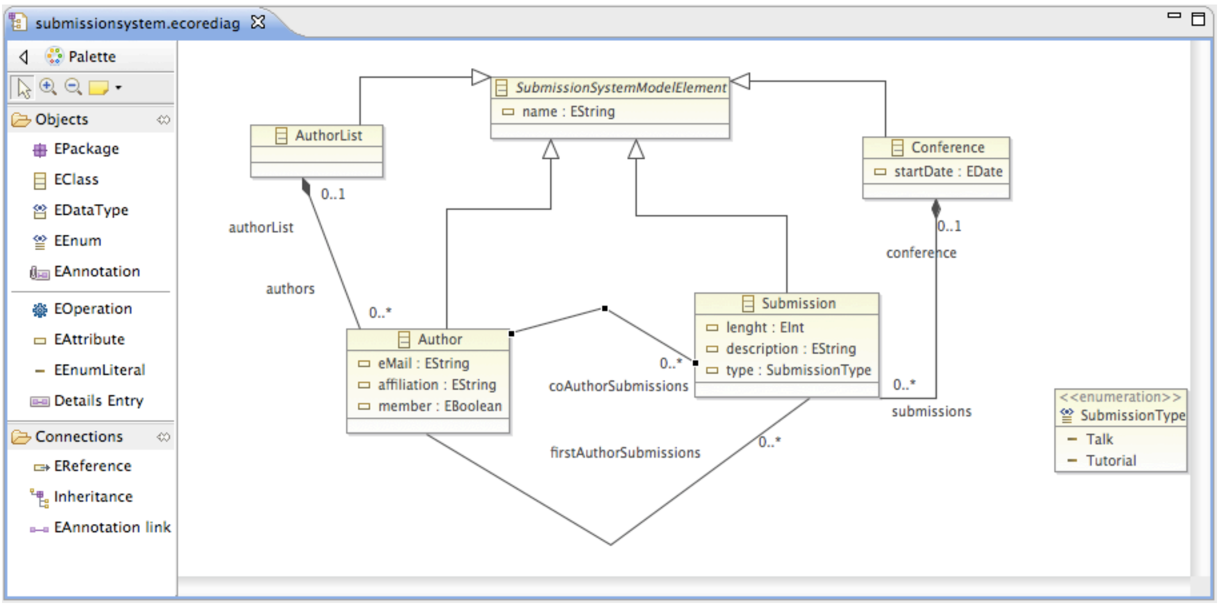
*submissionsystem.ecore

```
▼ platform:/resource/ConferenceSubmissionSystem/model/submissionsystem.ecore
  ▼ submissionSystem
    ▶ SubmissionSystemModelElement
    ▼ AuthorList -> SubmissionSystemModelElement
        authors : Author
    ▶ Conference -> SubmissionSystemModelElement
    ▼ Author -> SubmissionSystemModelElement
      ▶ authorList : AuthorList
    ▶ Submission -> SubmissionSystemModelElement
    ▶ SubmissionType
```

Properties

Property	Value
Changeable	true
Container	false
Containment	true
Default Value Literal	
Derived	false
EKeys	
EOpposite	authorList : AuthorList
EType	Author -> SubmissionSystemModelElement
Lower Bound	0
Name	authors
Ordered	true
Resolve Proxies	true
Transient	false
Unique	true
Unsettable	false
Upper Bound	-1
Volatile	false

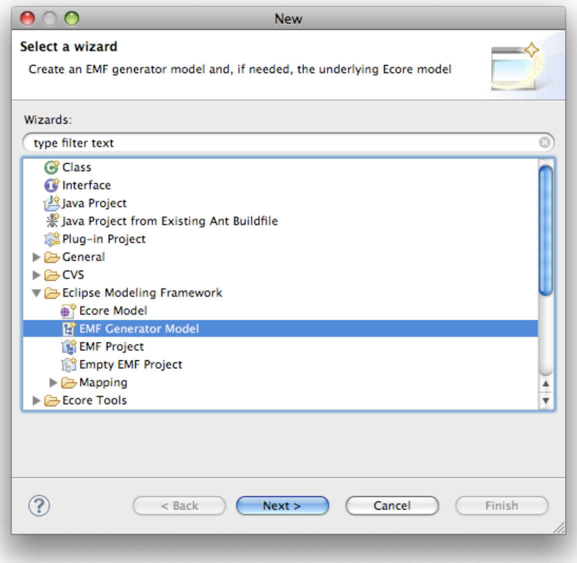
EMF models can also be defined using a graphical representation as a UML class diagram. We will use this to validate the created model. Please right click on the created ecore file and select “Initialize Ecore Diagram File”. The result should look like this, although the layout might be slightly different ;):



Generation of the Entities

In this step, we will generate the entities from the ecore file we have created. Please note that you can regenerate the entities again, if you need to change you model. EMF can deal with simple changes like adding a new attribute. If you have complex changes, like moving an attribute to another class, you would have to migrate existing instances of the model. This can be done using the EDAPT framework, which is built in the EMFStore.

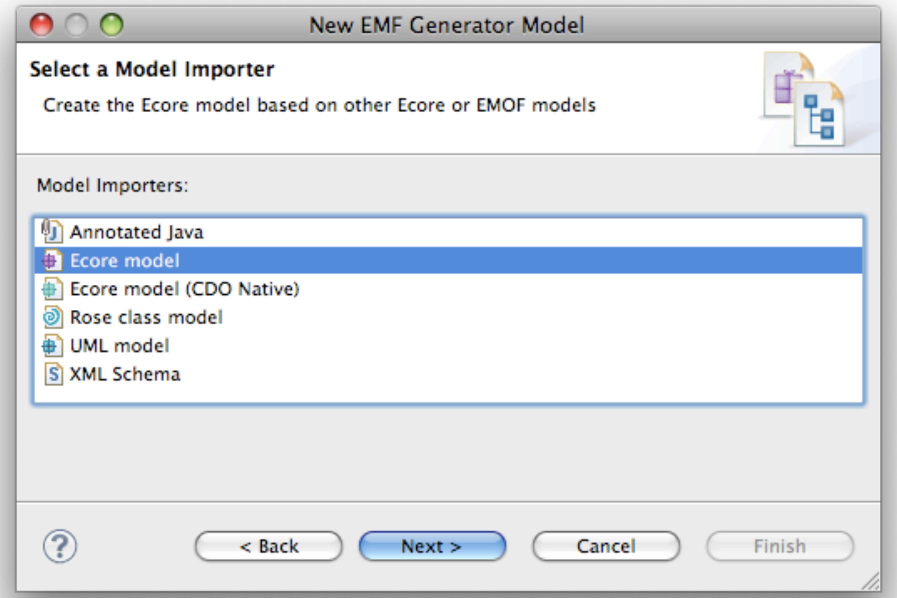
To generate entities, we have to create a generator model file first. This file allows configuring properties for the generation, which are not part of the model itself, for example the plugin and subfolder, the source code is generated to. Please create a new generator file in the model folder and select our previously created Ecore as a source model.


New

Select a wizard

Create an EMF generator model and, if needed, the underlying Ecore model

Wizards:


New EMF Generator Model

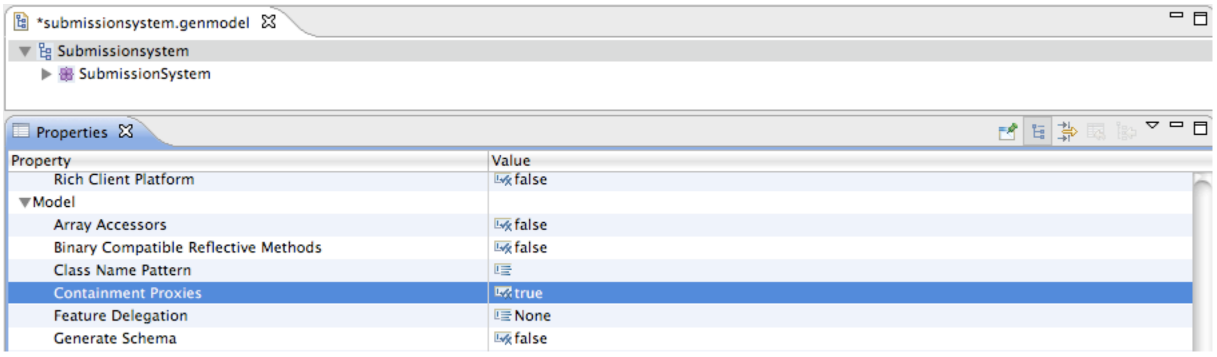
Select a Model Importer

Create the Ecore model based on other Ecore or EMOF models

Model Importers:

- Annotated Java
- Ecore model
- Ecore model (CDO Native)
- Rose class model
- UML model
- XML Schema

In the root node of the generator model, you can set the properties for the code generation. A complete description of these properties goes beyond the scope of this tutorial. We will use the standard values, expect we will set the property "Containment proxies" to true. Containment proxies allow splitting a model into several resources (cross-resource containment). If no containment proxies are generated then all contained children have to be in the same resource as their parents. The containment option is very commonly used, it is even enabled for the Ecore´s Ecore model (Ecore.ecore).


*submissionsystem.genmodel

- Submissionsystem
  - SubmissionSystem

Properties

Property	Value
Rich Client Platform	false
Model	
Array Accessors	false
Binary Compatible Reflective Methods	false
Class Name Pattern	
Containment Proxies	true
Feature Delegation	None
Generate Schema	false

In the tree of the generator model, we can set properties for every generated entity. Please set the property "Property Type" to "editable" for all references in our model. This will affect that our references are visible and editable in the UI later on. Further, set the "Property Mulit-line" to true for the attributes "affiliation" and "description". This will affect the UI to show multi-line fields for these attributes later on.

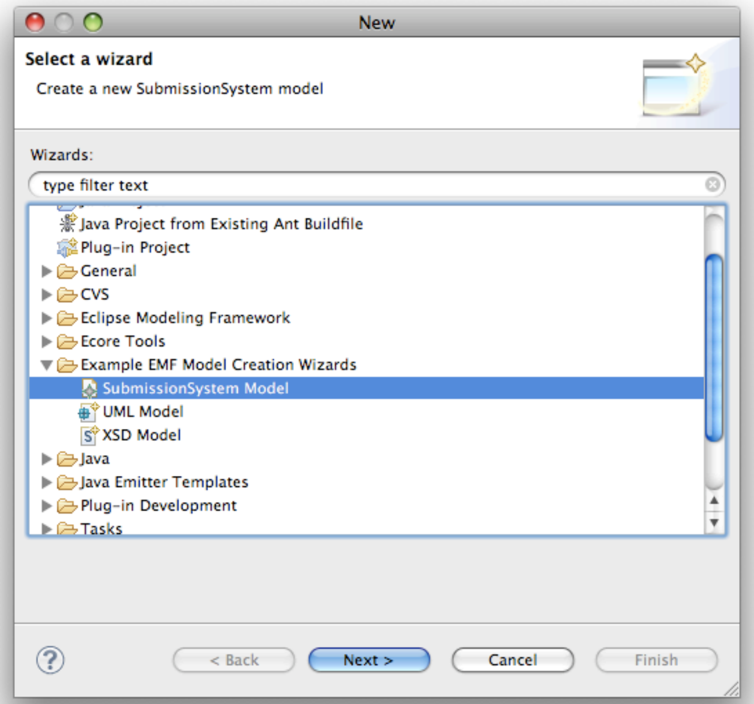
Based on the generator model, we can now generate the source code. EMF allows generating four different plugins for a defined model:

- Model: The model contains all entities, packages and factories to create instances of the model.

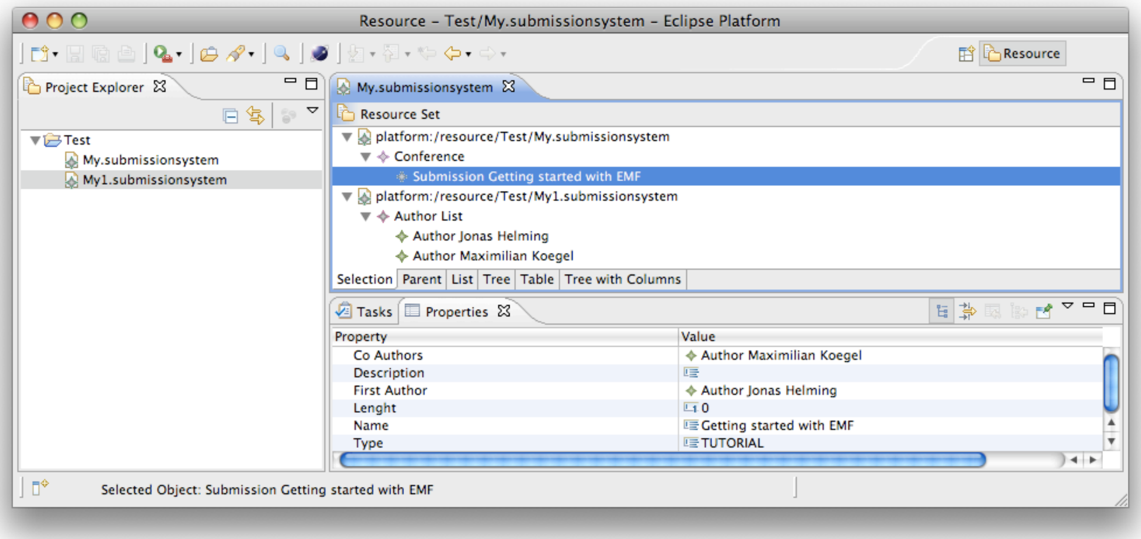
- Edit: The edit plugin contains providers to display a model in an UI. For example the providers offer a label for every model element, which can be used to display an entity in a list and showing an icon and a name.
- Editor: The editor plugin is a generated example editor to create and modify instances 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 first start the application. Please right click on the model plugin and select “Debug as => Eclipse Application”.

Please create a new project and add two new resources, one for the conference, one for the author list.



The generated example editor allows you to create new instances, edit attributes and create references.



```
▼ Test
    My.submissionsystem
    My1.submissionsystem
```

```
Resource Set
▼ platform:/resource/Test/My.submissionsystem
  ▼ Conference
      Submission Getting started with EMF
▼ platform:/resource/Test/My1.submissionsystem
  ▼ Author List
      Author Jonas Helming
      Author Maximilian Koegel
```

Property	Value
Co Authors	Author Maximilian Koegel
Description	
First Author	Author Jonas Helming
Lenght	0
Name	Getting started with EMF
Type	TUTORIAL

Selected Object: Submission Getting started with EMF

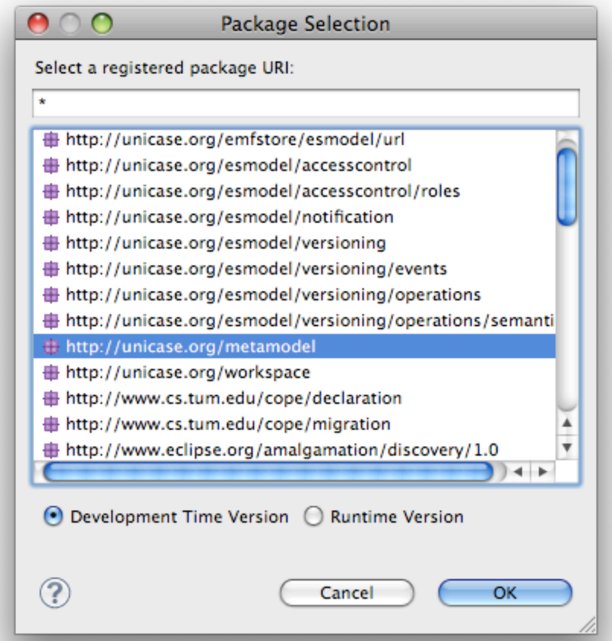
Model instances can also be created and modified using the API. We will show how to do this in the later parts of this tutorial.

4. Customizing the UI

In the second part of the tutorial, we will demonstrate, how to create and customize a User Interface for our example application. Although the generated editor of EMF provides the possibility to initially create model instances, it is meant to be an example and not an very extensible solution. Instead, we will use the EMF Client platform (EMFCP), which provides a reflective, adaptable and extensible UI for EMF applications. In contrast to the generated editor, the UI components of EMFCP have not to be generated. In a first step, we will prepare our created example model to be used with the EMF Client platform and the EMFStore later on. Second, we will adapt the appearance of model elements in the UI. Finally, we will customize the UI components of our application, the editor and the navigator.

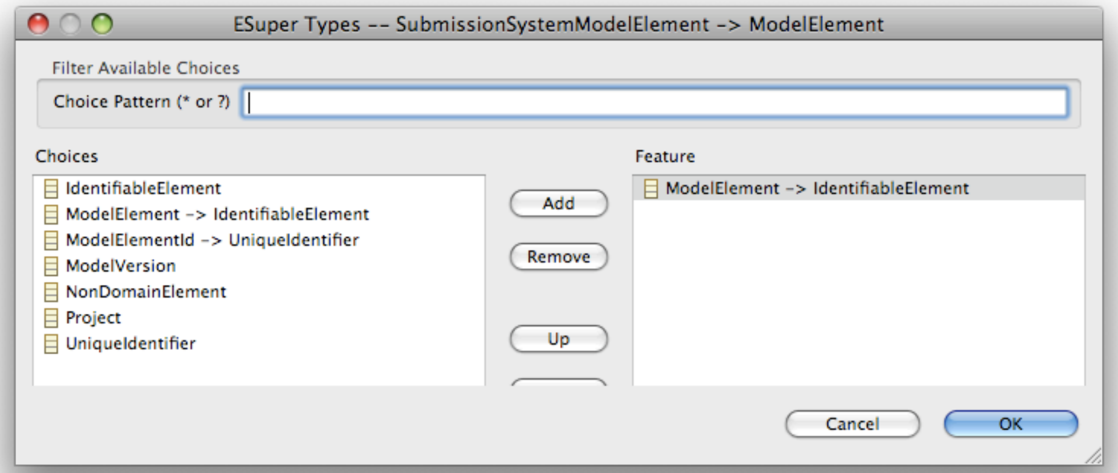
Preparing the model

In order to use the EMF Client platform and the EMFStore later on, we have to adapt our model to inherit a common meta-model. Please note, that this adaptation will not be necessary in upcoming releases. Please right click on the root node of your model and select "Load Resource" to load the meta-model. This make the meta-model available in the ecore editor. Set the super class of "SubmissionSystemModelElement" to "ModelElement" to let all elements of your model inherit from this type. Finally create a dependency of the model plugin to the meta-model.



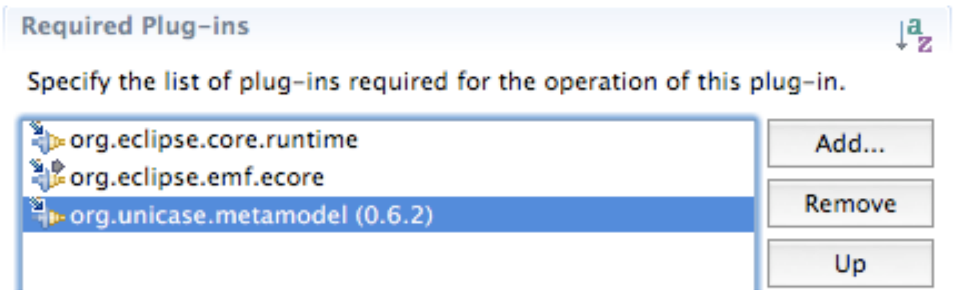
http://unicase.org/emfstore/esmodel/url
http://unicase.org/esmodel/accesscontrol
http://unicase.org/esmodel/accesscontrol/roles
http://unicase.org/esmodel/notification
http://unicase.org/esmodel/versioning
http://unicase.org/esmodel/versioning/events
http://unicase.org/esmodel/versioning/operations
http://unicase.org/esmodel/versioning/operations/semanti
http://unicase.org/metamodel
http://unicase.org/workspace
http://www.cs.tum.edu/cope/declaration
http://www.cs.tum.edu/cope/migration
http://www.eclipse.org/amalgamation/discovery/1.0

[✓ Development Time Version] Runtime Version



Choices:
IdentifiableElement
ModelElement -> IdentifiableElement
ModelElementId -> UniqueIdentifier
ModelVersion
NonDomainElement
Project
UniqueIdentifier

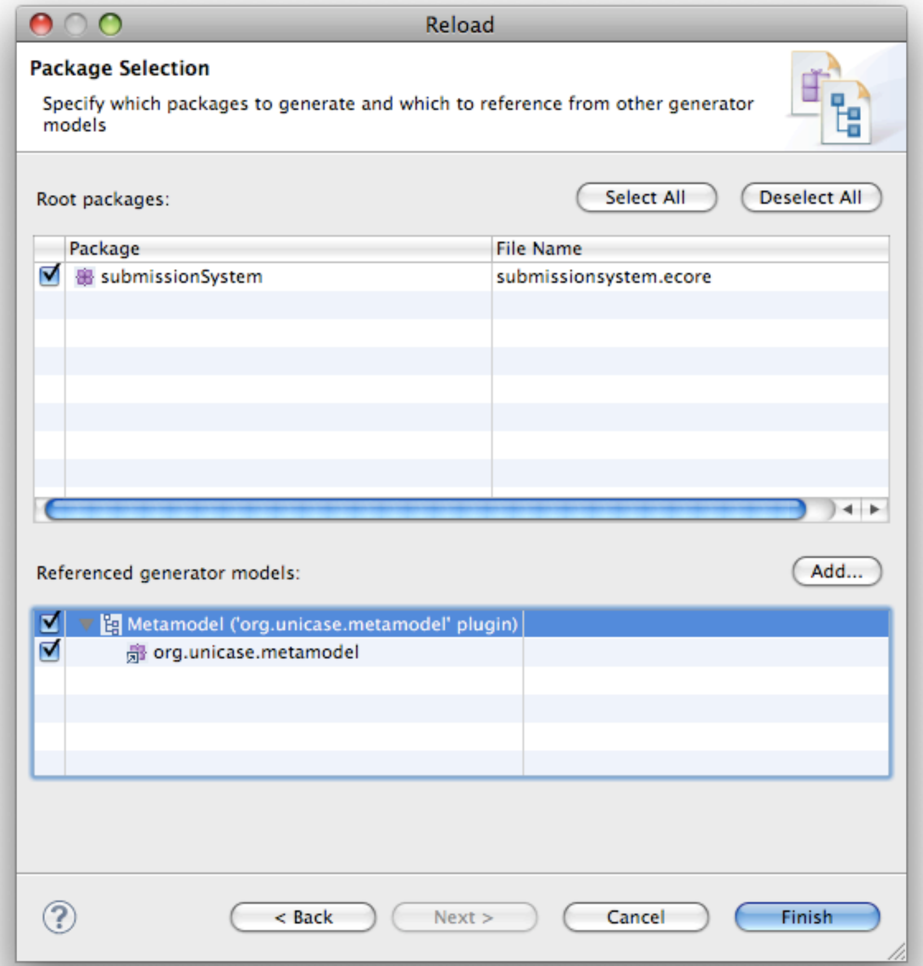
Feature:
ModelElement -> IdentifiableElement



Specify the list of plug-ins required for the operation of this plug-in.

org.eclipse.core.runtime
org.eclipse.emf.ecore
org.unicase.metamodel (0.6.2)

The next step is to regenerate our model. Right click on the generator model and select "reload". As we do not want to generate the meta-model of EMFCP, we will mark it as a referenced model. Then, generate the model as done before and restart the application.


Reload

Package Selection

Specify which packages to generate and which to reference from other generator models

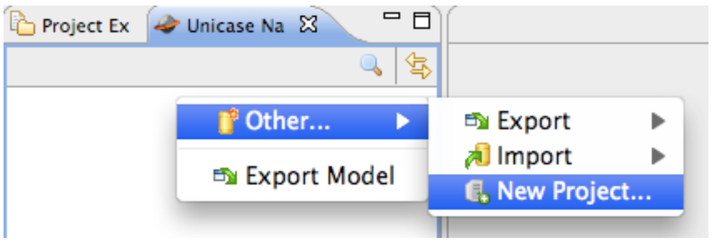
Root packages:

	Package	File Name
✓	submissionSystem	submissionsystem.ecore

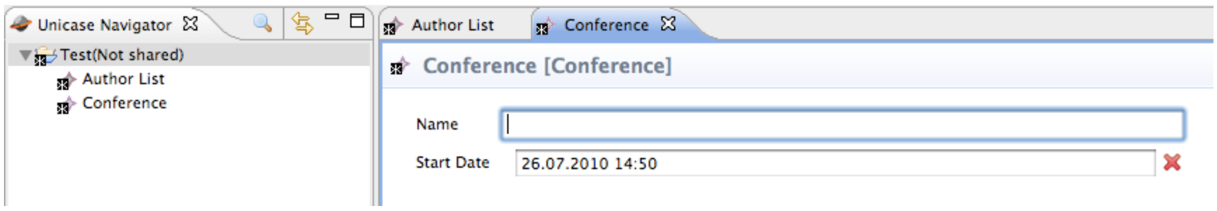
Referenced generator models:

✓ Metamodel ('org.unicase.metamodel' plugin)
✓ org.unicase.metamodel

Open the EMFCP navigator view. You will find it in the category "UNICASE", which is the umbrella project for EMFCP and EMFStore. The navigator allows you to create projects, which are containers for your models.



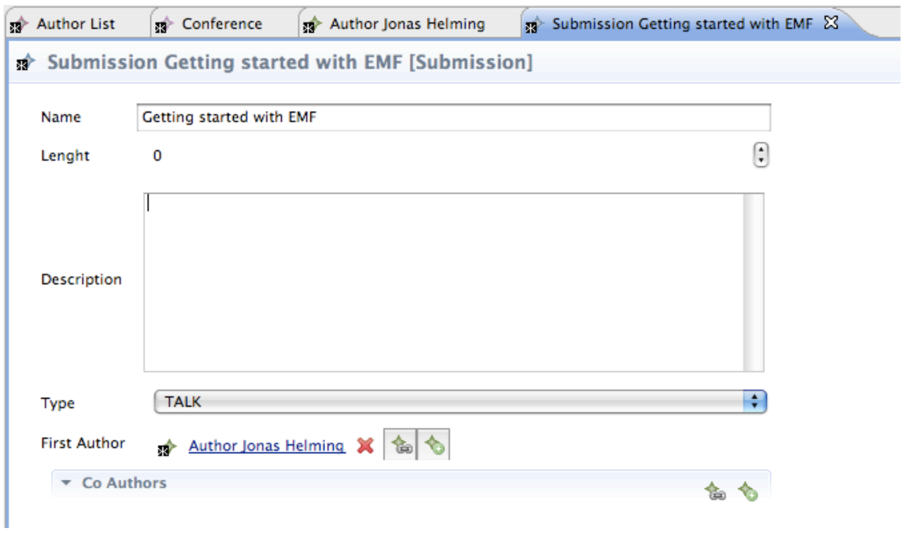
Please right click the new project and create model elements. EMFCP offers a tree-based and filterable wizard for the creation of entities. Please create an author list and a conference. As you can see, EMFCP renders model elements in a form-based editor. This editor is not generated, but reflective. It can be adapted, as we will show later on.



Conference [Conference]

Name	
Start Date	26.07.2010 14:50

To create a small example model, right click on the conference and the author list to create a submission and an author. Link both entities by clicking on the button in the editor.



Submission Getting started with EMF [Submission]

Name: Getting started with EMF
Lenght: 0
Description:
Type: TALK
First Author: Author Jonas Helming

Customizing Labels

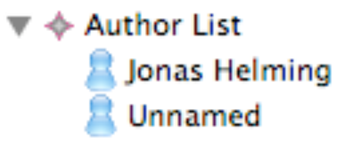
In this step, we want to customize the general appearance of model elements in our application. We want to add custom icons and remove the type information from the displayed text. This means instead of "Submission {Name of Submission}" we want to show only the name behind the icon. Therefore we change the LabelProvider offered by EMF. EMF generates a ItemProvider for every model element. I can be found in the *.edit plugin. ItemProvider implement a bunch of interfaces used by UI frameworks, to display model elements. The interface responsible for creating a label is IItemLabelProvider and consists of the methods getImage() and getText().

```java
public Object getImage(Object object) {
    return overlayImage(object, getResourceLocator().getImage("full/obj16/Author"));
}

public String getText(Object object) {
    String label = ((Author)object).getName();
    return label == null || label.length() == 0 ?
        getString("_UI_Author_type") :
        getString("_UI_Author_type") + " " + label;
}
```

To change the displayed icon, we can simply replace the respective image file. For the adaptation of the displayed text behind the icon, we change the method getText() as follows. Please adapt this method only for the AuthorItemProvider to see the differences to other elements in the application.

```
/**
 * This returns the label text for the adapted class.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated NOT
 */
@Override
public String getText(Object object) {
    String label = ((Author)object).getName();
    return label == null || label.length() == 0 ?
        "Unnamed" : label;
}
```

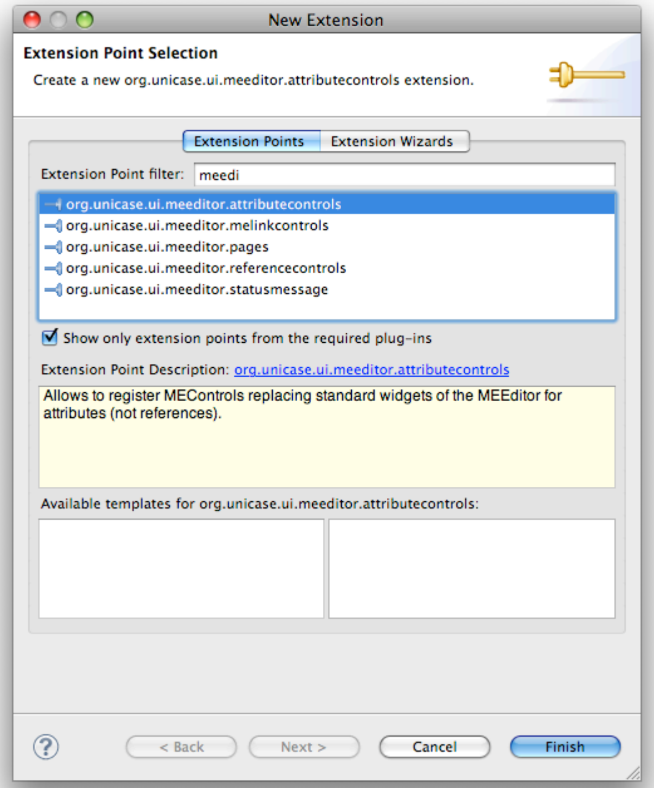
The new method will return the name of the author, if there is a name, or "Unnamed", if there is no name. Please also note that we have marked the method as "generated NOT". This will prevent the code generator from regenerating this method and preserve our change, if we regenerate the model.



Customizing the Editor

EMFCP allows rearranging the displayed attributes and references in the editor by annotating the model. Further, you can replace widgets for specific attributes and add new tabs. In the tutorial, we will adapt editor, by adding a new custom widget. The widget allows sending a Mail to an author. Therefore, we will adapt the way, the attribute "Email" of the entity author is displayed from a text field to a text field with a button. The new widget is added via an extension point.

Please add a new plugin in your workspace for the customized UI components and add a dependency to the example model as well as to the EMFCP editor: "org.unicase.ui.meeditor". Please add an extension to this plugin, using the extension point "org.unicase.meeditor.attributecontrols". This extension point allows registering custom widgets for specific attributs in your model. They will be used by the editor to render this attribute.


New Extension

Extension Point Selection

Create a new org.unicase.ui.meeditor.attributecontrols extension.

Extension Point filter: meedi

- org.unicase.ui.meeditor.attributecontrols
- org.unicase.ui.meeditor.melinkcontrols
- org.unicase.ui.meeditor.pages
- org.unicase.ui.meeditor.referencecontrols
- org.unicase.ui.meeditor.statusmessage

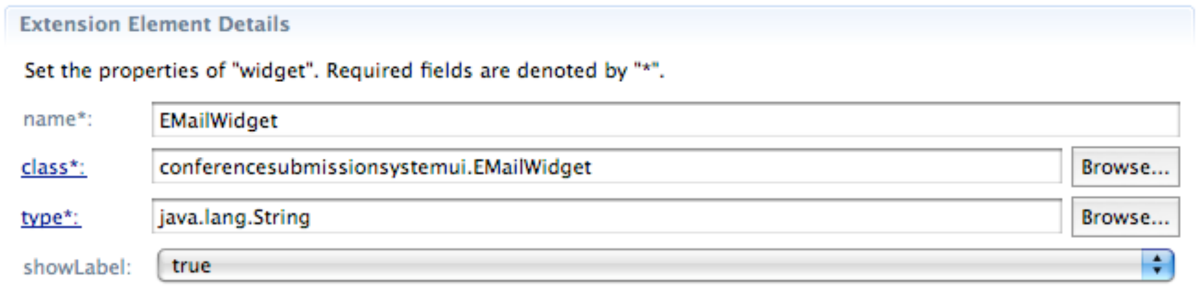
☑ Show only extension points from the required plug-ins

Extension Point Description: org.unicase.ui.meeditor.attributecontrols

Allows to register MEControls replacing standard widgets of the MEEditor for attributes (not references).

Available templates for org.unicase.ui.meeditor.attributecontrols:

Please define a name and a class for your widget. The type must match the type of the attribute, you want to display, and for the Email Attribute it is “String”. The Boolean “Show Label” determines whether the editor displays a label with the name of the attribute beside the widget. Click on the “class” link to create the new widget and let it inherit from the existing “org.unicase.ui.meeditor.mecontrols.METextControl”.


Extension Element Details

Set the properties of "widget". Required fields are denoted by "*".

name*:	EMailWidget
class*:	conferencesubmissionsystemui.EMailWidget
type*:	java.lang.String
showLabel:	true

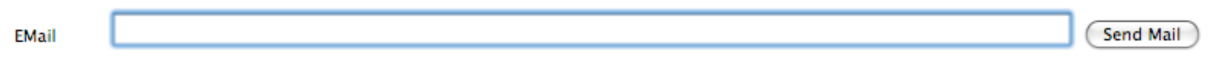
Please override the two methods: canRender() and createControl() to adapt the existing control used to render Sting attributes. The canRender() method is used by the editor to determine, which widget shall be used to display an attribute. The editor will use the widget with the highest priority. As we want to display the attribute “Email” with our custom widget, we change the method as follows:

```
public int canRender(IItemPropertyDescriptor itemPropertyDescriptor, ModelElement modelElement) {
    EStructuralFeature structuralFeature = (EStructuralFeature) itemPropertyDescriptor.getFeature(modelElement);
    if (structuralFeature.getName().equalsIgnoreCase("email")) {
        return 2;
    }
    return AbstractMEControl.DO_NOT_RENDER;
}
```

Further, we adapt the createControl() method, which is called to render the widget itself. We will used the text field provided by the existing METextControl and add a Button to send a Mail to the address specified in the text field:

```
public Control createControl(Composite parent, int style) {
    //Create Layout
    Composite composite = getToolkit().createComposite(parent, style);
    GridLayout gridLayout = new GridLayout(2, false);
    composite.setLayout(gridLayout);
    GridDataFactory.fillDefaults().align(SWT.FILL, SWT.CENTER).grab(true, true).applyTo(composite);

    //Create Text Field
    meAreaControl = new METextControl();
    final Text txtEmail = (Text) meAreaControl.createControl(composite, style, getItemPropertyDescriptor(),
        getModelElement(), UnicaseActionHelper.getContext(getModelElement()), getToolkit());
    GridDataFactory.fillDefaults().align(SWT.FILL, SWT.CENTER).grab(true, true).applyTo(txtEmail);

    //Create Mail Button
    final Action mail = new Action("Send email", SWT.PUSH) {

        @Override
        public void run() {
            String email = txtEmail.getText();
            Program.launch("mailto:" + email);
        }

    };
    Button button = new Button(composite, SWT.PUSH);
    button.addSelectionListener(new SelectionAdapter() {
        @Override
        public void widgetSelected(SelectionEvent e) {
            mail.run();
        }

    });
    button.setLayoutData(new GridData(SWT.LEFT, SWT.CENTER, false, false));
    button.setImage(Activator.getImageDescriptor("icons/mail.png").createImage());

    return parent;
}
```

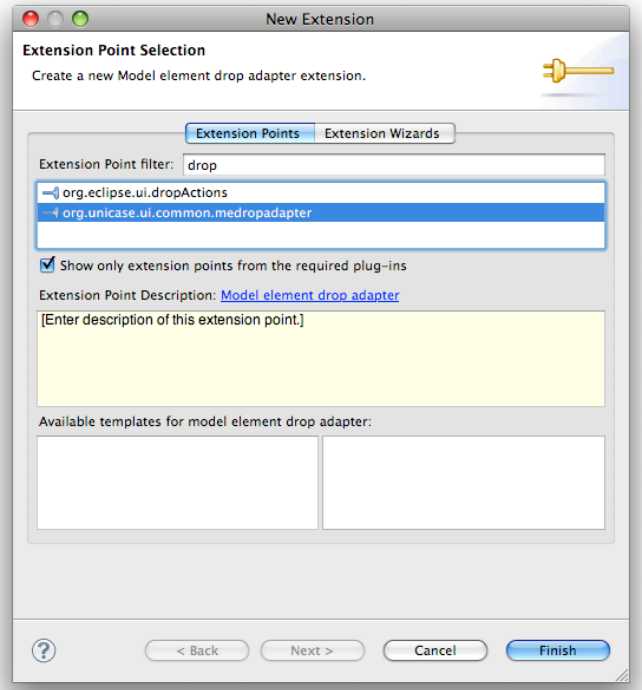
Please restart the application and open an entity of type author, to see the custom widget displayed by the editor:



Customizing Drag and Drop

In this step, we will adapt the drag and drop behavior in the EMFCP navigator. The goal is to allow a drop of a author entity on an submission entity affecting the following behavior: If the submission has no first author, the dropped author will become first author. If the submission has already an first author, the dropped author will become a co-author of the submission.

To change the drop behavior in EMFCP we will again use an extension point. Please create an extension in your UI pluging using the extension point "org.unicase.ui.common.medropadapter":


New Extension

Extension Point Selection

Create a new Model element drop adapter extension.

Extension Point filter: drop

org.eclipse.ui.dropActions
org.unicase.ui.common.medropadapter

[✓] Show only extension points from the required plug-ins

Extension Point Description: Model element drop adapter

[Enter description of this extension point.]

Available templates for model element drop adapter:

Please specify a class name for your drop adapter and create it. The drop adapter contains of two methods: isDropAdapterFor() and drop(). The method isDropAdapterFor() specifies for which type of entity, this drop adapter is used. As we want to implement a drop adapter for submissions, we change it as follows:

```
public EClass isDropAdapterfor() {
    return SubmissionSystemPackage.eINSTANCE.getSubmission();
}
```

The second method drop() defines the actual behavior, when a drop occurs. If the source is of type author, we will execute a custom drop command:

```
public void drop(DropTargetEvent event, ModelElement target,
        List<ModelElement> source) {
    if(target instanceof Submission){
        for(ModelElement modelElement: source){
            if(modelElement instanceof Author){
                new DropCommand((Submission)target, (Author)modelElement).doExecute();
            }
        }
    }
}
```

Commands are used to create read and write transactions on a model in EMF. Although you could also directly access the API of EMF entities, commands provide advantages, for example a possible undo. In our case, we use the EMF transaction framework, providing transactional safety of executed commands and therefore enables parallel access to the model. Our custom command inherits from the EMF transaction class RecordingCommand. In the doExecute() method, we actually modify the model.

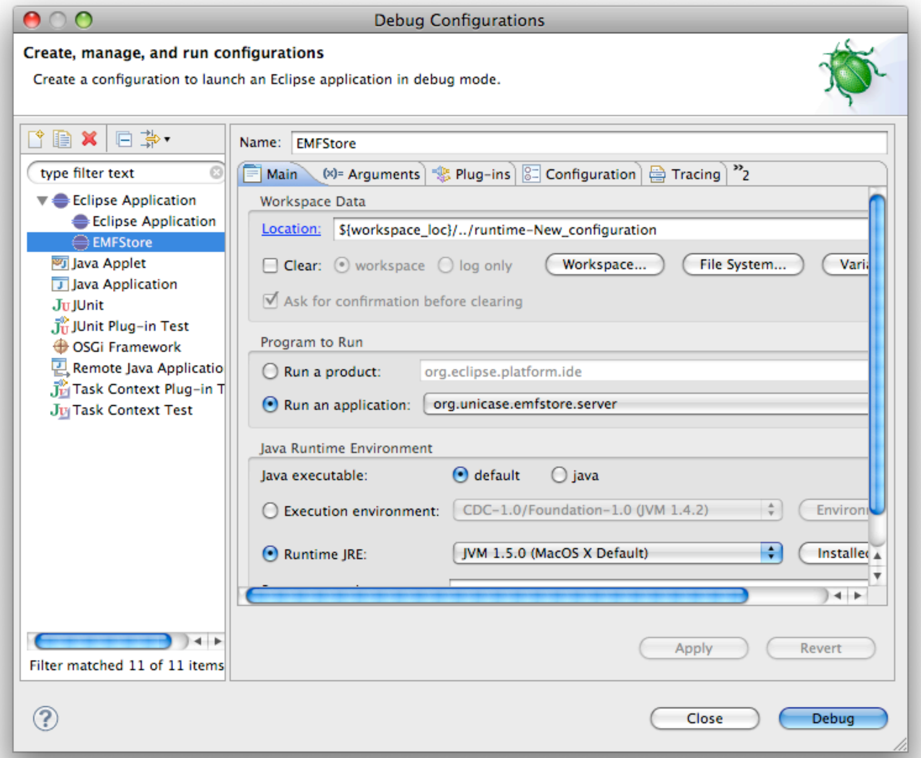
```
public class DropCommand extends RecordingCommand{

    private final Submission target;
    private final Author source;

    public DropCommand(Submission target, Author source) {
        super(getEditingDomain());
        this.target = target;
        this.source = source;
    }

    @Override
    protected void doExecute() {
        if(target.getFirstAuthor()==null){
            target.setFirstAuthor(source);
        }
        else{
            if(!target.getCoAuthors().contains(source)){
                target.getCoAuthors().add(source);
            }
        }

    }

}
```

Please restart the example application and drop several authors on a submission in the navigator to see the implemented behavior.

5. Distributing the model

In the last part of our tutorial, we will use the EMFStore to share the model instances of our example application among several clients. In the first part, we will demonstrate the features of the EMFStore available without any adaptation. In the second part, we will create a custom client for the EMFStore, which will send an email, whenever a new submission for the conference is added.

To share our model, we will start an EMFStore server locally. Please create a new debug configuration and select to run the EMFStore product:



Create, manage, and run configurations

Create a configuration to launch an Eclipse application in debug mode.

type filter text

Filter matched 11 of 11 items

Name: EMFStore

Workspace Data

Location: ${workspace_loc}/../runtime-New_configuration

Clear: workspace log only

[✓] Ask for confirmation before clearing

Program to Run

Run a product: org.eclipse.platform.ide

[✓ Run an application:] org.unicase.emfstore.server

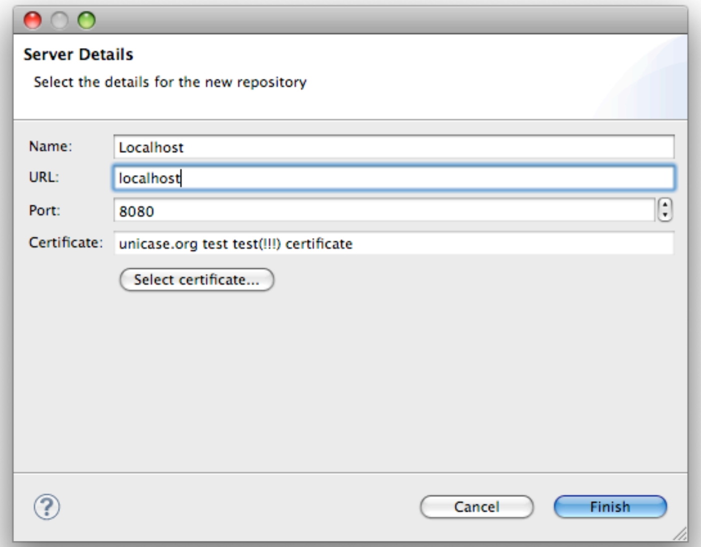
Java Runtime Environment

Java executable: [✓ default] java

Execution environment: CDC-1.0/Foundation-1.0 (JVM 1.4.2)

[✓ Runtime JRE:] JVM 1.5.0 (MacOS X Default)

After the server has successfully started, you can connect to it via the EMFStore Browser view in your example application. Therefore you have to create a new server configuration pointing to your localhost:



Server Details

Select the details for the new repository

Name: Localhost

URL: localhost

Port: 8080

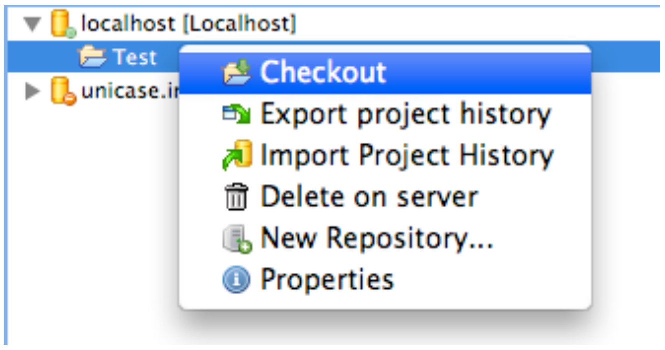
Certificate: unicase.org test test(!!!) certificate

Please login at the local server using the default credentials:

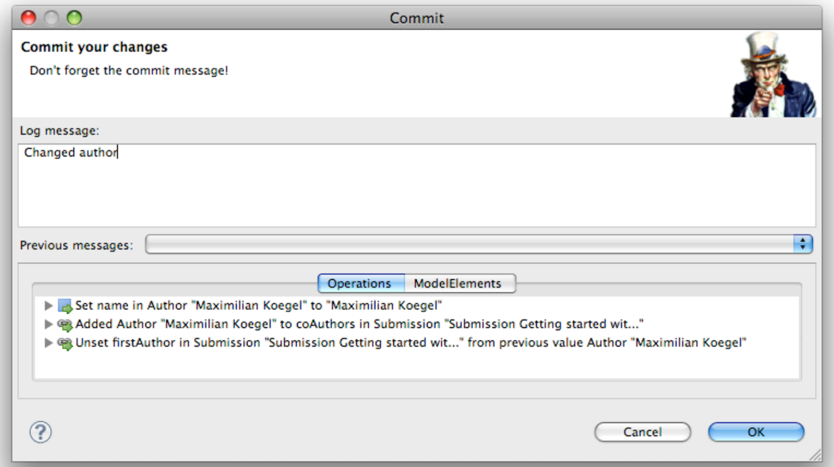
User: **super**

Password: **super**

After you are logged in, you can share the existing example project instance via a right click on the project. Please checkout a second copy of the project via the EMFStore Browser to test the collaboration features. This copy would be on another client in a realistic setting.



After the checkout, you will have to instances of the example project in your navigator. Please apply some changes on the second instance, right click the project and commit the changes. The commit dialog provides details about the changes and allows entering a commit message:


Commit

Commit your changes

Don't forget the commit message!

Log message:

Changed author

Previous messages:

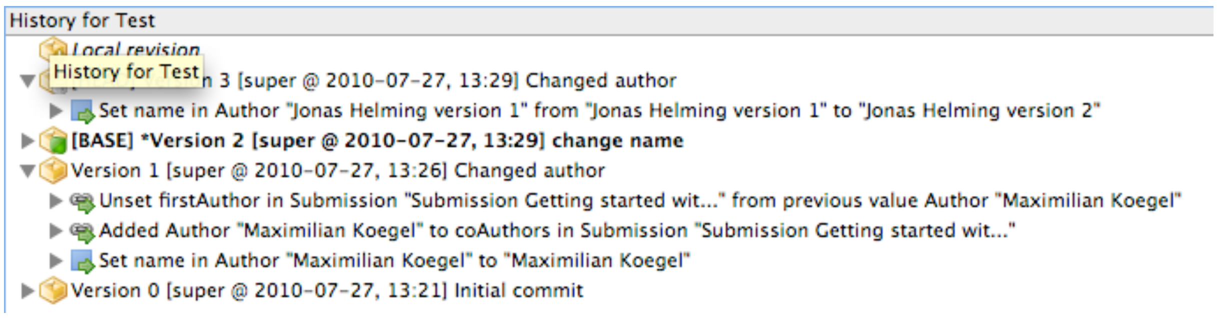
Operations | ModelElements

Set name in Author "Maximilian Koegel" to "Maximilian Koegel"

Added Author "Maximilian Koegel" to coAuthors in Submission "Submission Getting started wit..."

Unset firstAuthor in Submission "Submission Getting started wit..." from previous value Author "Maximilian Koegel"

After the commit the version number of the second instance will be increased to "1", while the first instance will remain in version "0". Now, please update the first project to obtain the changes made in the second instance. As a result, both instances are in version"1". You can obtain a detailed history of the project by right clicking on it an selecting "Show history". The history is also available for single model elements. (such as a submission of the conference for example)


History for Test

Local revision

History for Test n 3 [super @ 2010-07-27, 13:29] Changed author

Set name in Author "Jonas Helming version 1" from "Jonas Helming version 1" to "Jonas Helming version 2"

[BASE] *Version 2 [super @ 2010-07-27, 13:29] change name

Version 1 [super @ 2010-07-27, 13:26] Changed author

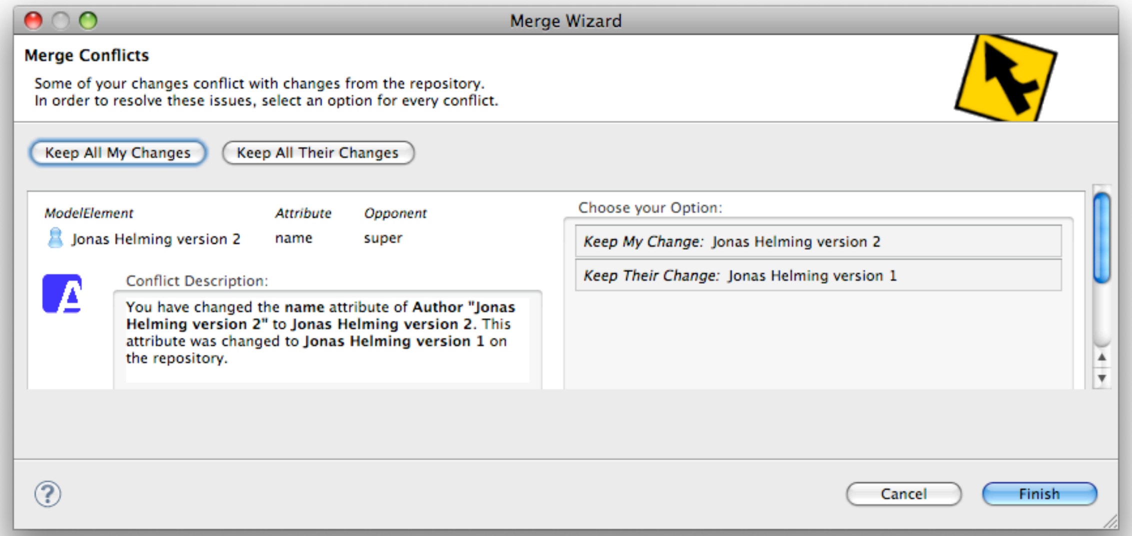
Unset firstAuthor in Submission "Submission Getting started wit..." from previous value Author "Maximilian Koegel"

Added Author "Maximilian Koegel" to coAuthors in Submission "Submission Getting started wit..."

Set name in Author "Maximilian Koegel" to "Maximilian Koegel"

Version 0 [super @ 2010-07-27, 13:21] Initial commit

The EMFStore allows working offline continuously. This might cause potential conflicts when the user commits his/her changes. Therefore, the EMFStore provides interactive merge support. Please change the name of the same author in both projects and commit the first. On a commit of the second instance, the EMFStore will show an interactive merge dialog, allowing the user to select how to resolve conflicting changes,



Some of your changes conflict with changes from the repository.
In order to resolve these issues, select an option for every conflict.

ModelElement	Attribute	Opponent
Jonas Helming version 2	name	super

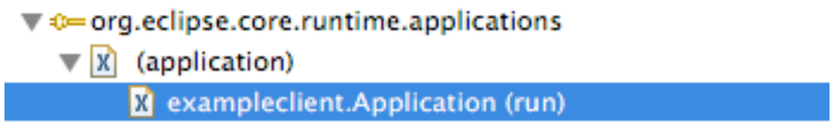
Keep My Change: Jonas Helming version 2

Keep Their Change: Jonas Helming version 1

You have changed the **name** attribute of **Author "Jonas Helming version 2"** to **Jonas Helming version 2**. This attribute was changed to **Jonas Helming version 1** on the repository.

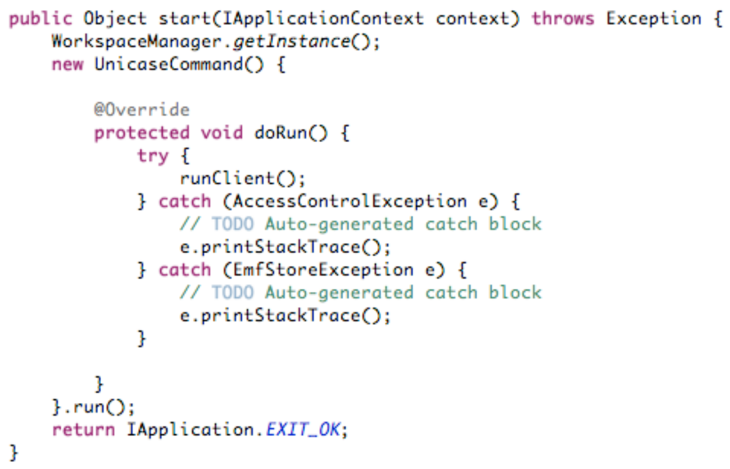
Create a custom client

In the final step of the tutorial, we will access the EMFStore via the API to create a headless client. In the example, we will create a client, which adds an example conference and submission to the project. To create the example client please create a new plugin "SampleClient" with dependencies to our example model as well as to the EMFStore workspace "org.unicase.workspace". Create a class "Application" implementing the IApplication interface. Register this class using the extension point "org.eclipse.core.runtime.application":



```
org.eclipse.core.runtime.applications
  (application)
    exampleclient.Application (run)
```

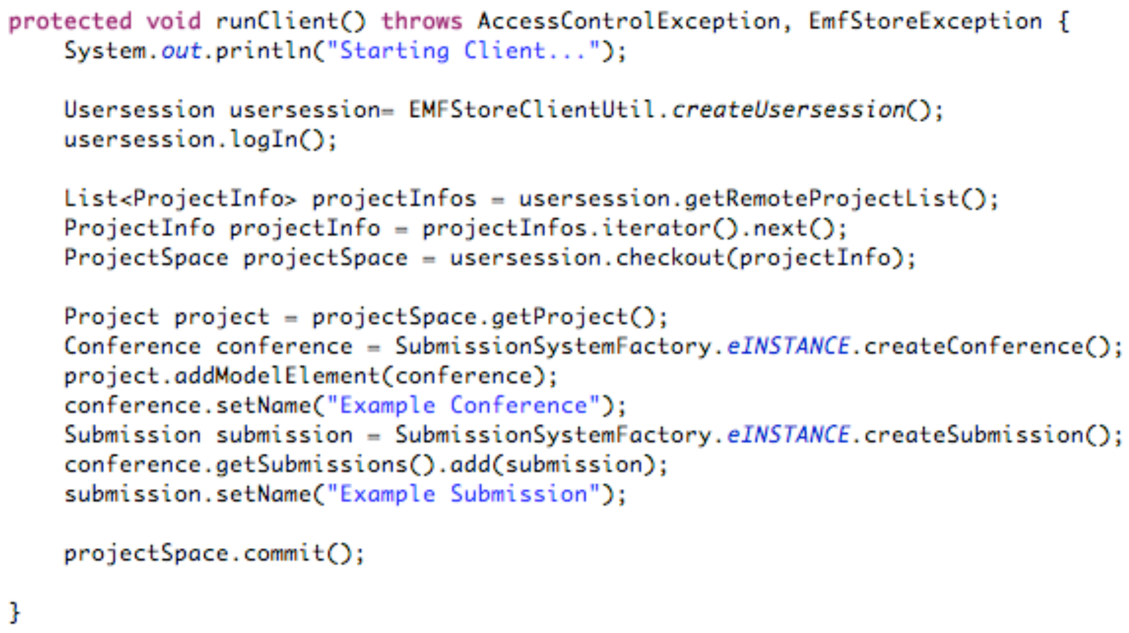
If you start this plugin, the start method of the class Application will be executed. Please modify the start() method to execute a custom command. The class UnicaseCommand is a helper class of the EMFStore workspace. In the first line, the workspace is initialized. The actual behavior is implemented in the runClient() method.



```
public Object start(IApplicationContext context) throws Exception {
    WorkspaceManager.getInstance();
    new UnicaseCommand() {

        @Override
        protected void doRun() {
            try {
                runClient();
            } catch (AccessControlException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (EmfStoreException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        }
    }.run();
    return IApplication.EXIT_OK;
}
```

Please implement the runClient() method as follows, it will execute the following steps:

- Login to the server
- Checkout the first project (Your example instance)
- Create and link a conference and submission entity
- Commit the changes to the server



```
protected void runClient() throws AccessControlException, EmfStoreException {
    System.out.println("Starting Client...");

    Usersession usersession= EMFStoreClientUtil.createUsersession();
    usersession.logIn();

    List<ProjectInfo> projectInfos = usersession.getRemoteProjectList();
    ProjectInfo projectInfo = projectInfos.iterator().next();
    ProjectSpace projectSpace = usersession.checkout(projectInfo);

    Project project = projectSpace.getProject();
    Conference conference = SubmissionSystemFactory.eINSTANCE.createConference();
    project.addModelElement(conference);
    conference.setName("Example Conference");
    Submission submission = SubmissionSystemFactory.eINSTANCE.createSubmission();
    conference.getSubmissions().add(submission);
    submission.setName("Example Submission");

    projectSpace.commit();

}
```

Please execute the application in the debug mode to follow the steps. After you have executed the application, please switch to the example client from the previous part and update the project. You will retrieve the changes done by the automated client.