

Model, generate and Deploy the skeleton of your Spring/Hibernate and Struts application from standard UML 2.1 Diagrams.



Bacem Souissi
Omondo research lab

The goal of this article is to present how EclipseUML 2007 Free Edition, using an agile model driven development methodology, can generate a complete JEE application having « presentation + service + persistence » from UML 2.1 Diagrams. This article will also show how to generate code within Eclipse 3.3 Europa.

I will only explore the code generation approach based on a simple JEE UML 2.1 model and will not introduce any advanced JEE modeling methodology. Another JEE modeling methodology tutorial will be written by Karim Djaafar in September 2007. For advanced modelers, I recommend to use a traditional RUP methodology and then add PSM stereotypes.

You can use EclipseUML 2007 in order to deploy a complete JEE application by :

- adding PSM stereotype inside your Eclipse 3.3 Europa UML/XMI 2.1 model
- importing or upgrading your current model to UML 2.1 and then adding PSM stereotypes.

Please note that the UML model is a kind of PIM model for Eclipse. The use of stereotypes allows us to switch from a traditional UML Tool, only generating blue print documentation to a PSM (Platform Specific Model) Model Driven development generating over 90% of the JEE application.

I have decided to use the following PSM code generation technologies.

- Spring for services
- Hibernate for persistence
- Struts technology presentation

You first need to install:

- [Eclipse 3.3](#)
- [EclipseUML Free Edition 2007](#)
- [WTP 2.0](#)
- [JBoss 4.0.4 as application server](#)

The modeling approach described in this tutorial is provided in order to help a beginner to easily understand the logic of modeling a JEE application. If you need more advanced modeling JEE practice, we recommend going to <http://www.andromda.com> and ask for training.

The Tutorial includes 3 steps :

1. [Model your JEE application](#)
 - a. [Model presentation workflows by using a State diagrams.](#)
 - b. [Model the associated workflows by using usecase diagrams](#)
 - c. [Model service and persistence by using a class diagram](#)
 - d. [Get the back-end information to be displayed in the Struts front tier application](#)
2. [JEE Code Generation](#)
3. [Launch the JEE Server and immediately see your model running on a JBoss server](#)

1. Model your JEE application

We are going to model two usecases which represent cases of use of a JEE system. The first usecase will display the list of cars on stock. The second will allow to add new cars using a browser as an interface.

For the purpose of this article, we will use three different diagrams :

- State diagram will model the presentation tier
- Usecase diagram will model the system
- Class diagram will model persistence and service tiers

This links between State, usecase and class diagram are important because you can immediately add presentation tier logic with service and persistence tier in order get a deployed code including all these parameters.

This is why you can get over 90% of the needed code just by modeling and linking all three layers (i.e : service, persistence and presentation). Just using class diagram is not enough.

a. State Diagram modeling

The State diagram role in this tutorial is to model the presentation layer.

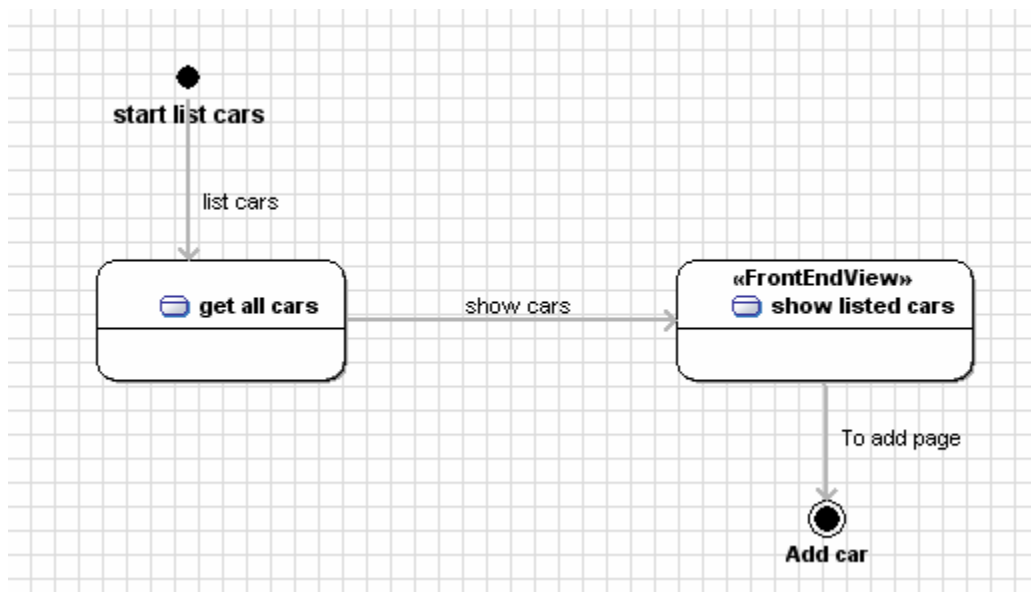
It correspond to a UML view of a dynamic Strut or JSF navigation. You can therefore open the generated code with a Strut or a JSF editor.

Each state diagram should be related to a usecase in the JEE modeling.

We need to create two state diagrams in order to model to display the list of cars and the add new car option.

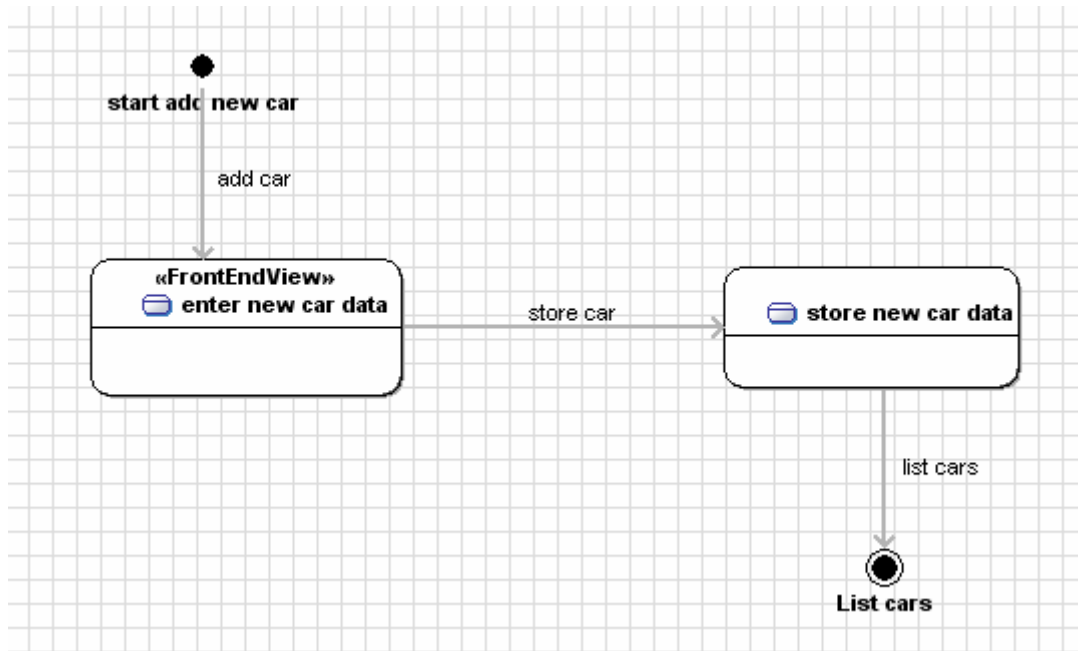
- **The « listCarActivity » State Diagram** correspond to a usecase and describe the dynamic layer presentation involved in the display of the list of available car workflow.
 - Start point (start list cars)
 - State (get all cars)
 - List cars transition
 - State (show listed cars) having « FrontEndView » stereotype
 - Show cars transistion
 - End Point (Add car) which should be the same name as the usecase in order to allow the transition to the next usecase.
 - To add page transition

You should get the following State diagram :



We now need to link the «ListCarsController» class to this State diagram in order to activate the code generation glu between Service-Persistence and presentation
The « addCarActivty » State Diagram corresponds to a usecase and describes the dynamic layer presentation involved in the display of the add car workflow.

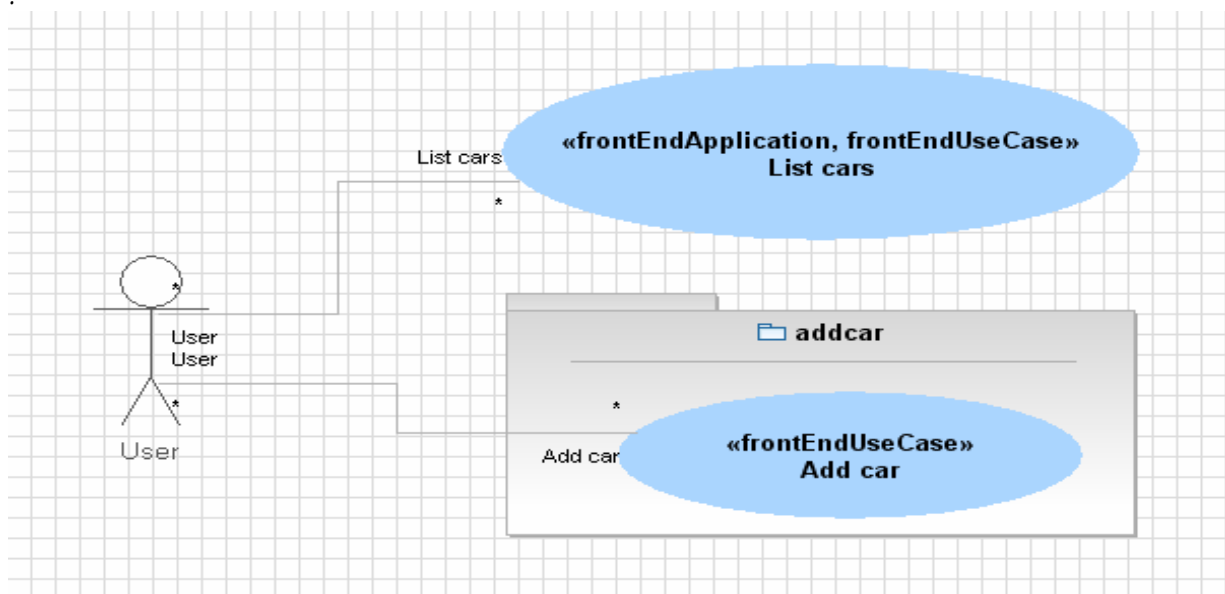
You should get the following diagram :



b. UseCase Diagram modeling

Our usecase diagram includes two usecases, because we have two State Diagrams.

- « List car » display the list of available cars. This usecase has two stereotypes
 - « Presentation ::frontEndUseCase » should be used for every usecase
 - « Presentation ::frontEndApplication » should only be used once because this is the entry point of the application
- « Add car » add new cars in the list
 - « presentation ::frontEndUseCase »



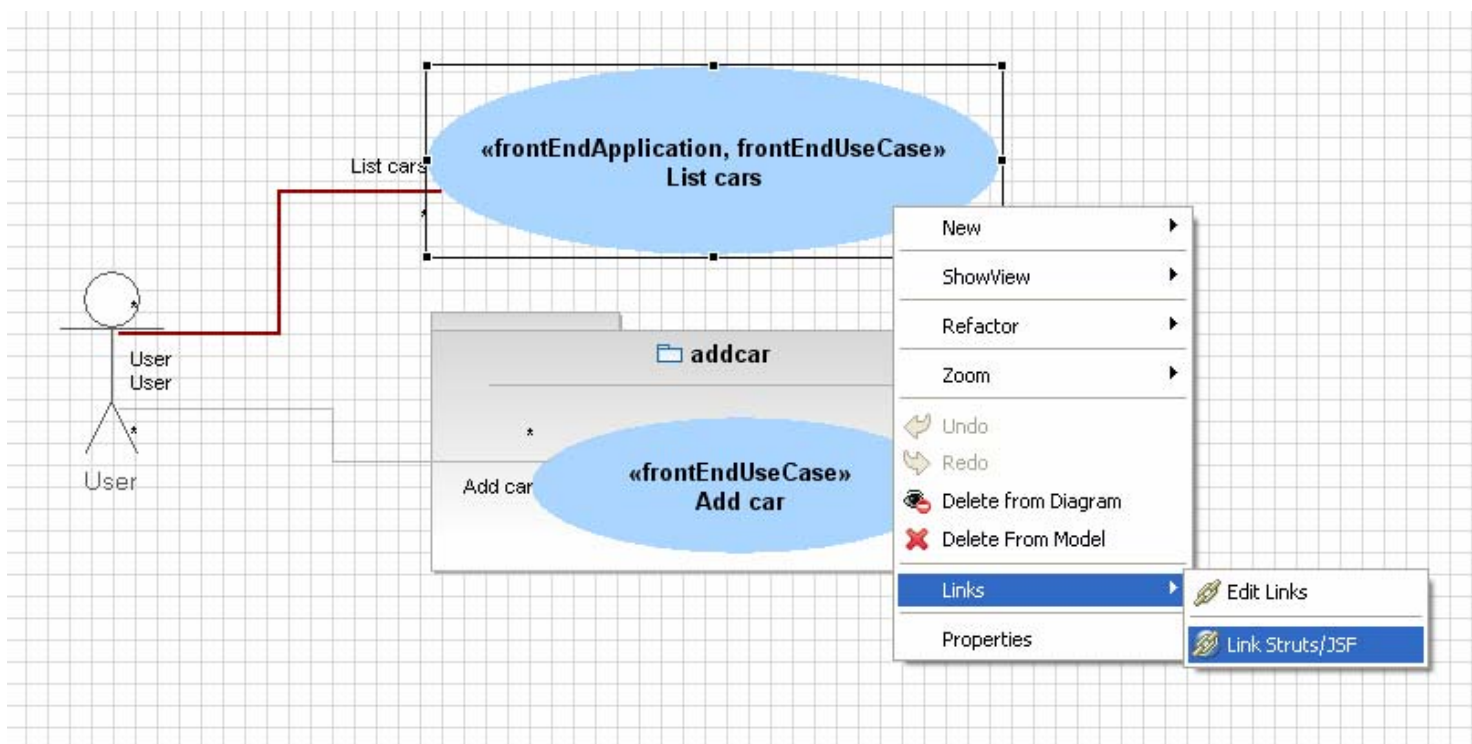
Please note that : we can't create two frontEndUseCase stereotype in the same package. This is why we have created the addcar package in the usecase diagram.

You can add stereotype on an usecase by selecting the usecase.

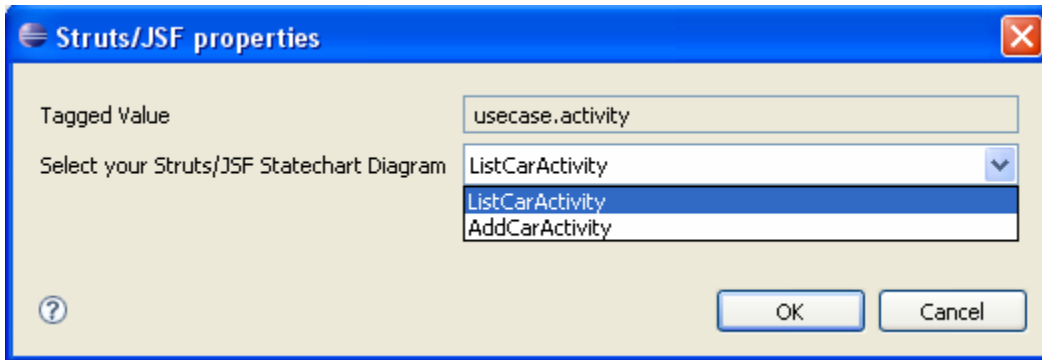
Click on **the usecase** > **select the stereotype tab** > **New Stereotype**

We now need to add the link between the usecase and the statediagram.

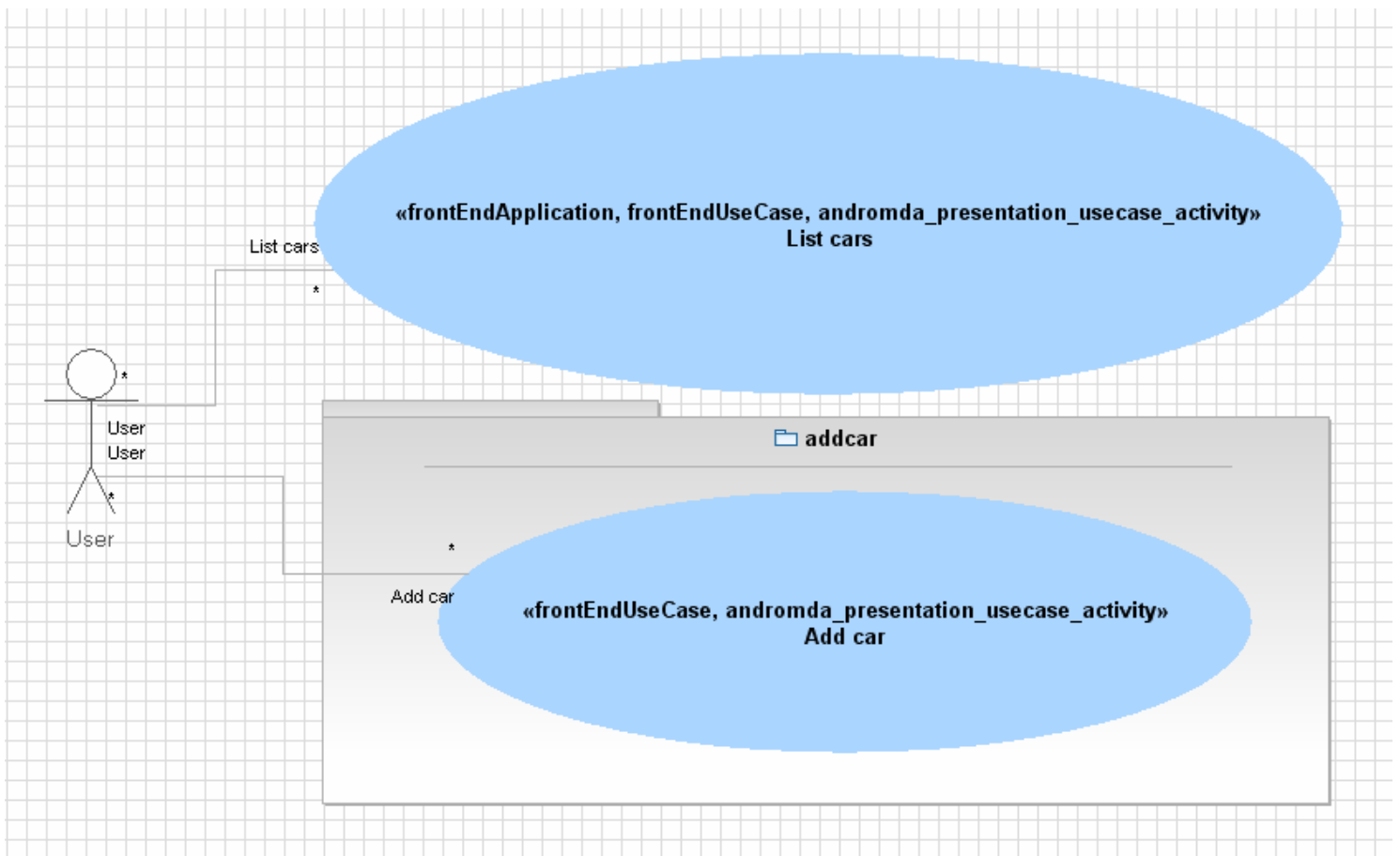
Select a usecase > **Open the contextual menu** > **Links** > **Link Struts/JSF**



Select the State Diagram to be linked to the usecase and click on the Ok Button.



You need to link both usecases to both State Diagrams.
You should get the following usecase diagram :



c. Class diagram modeling

The Class « controller » will add the glue between the service and presentation layer.

The class diagram purposes is to model the persistence and services layers and to add the needed glue between all the layers by using associations and dependencies links.

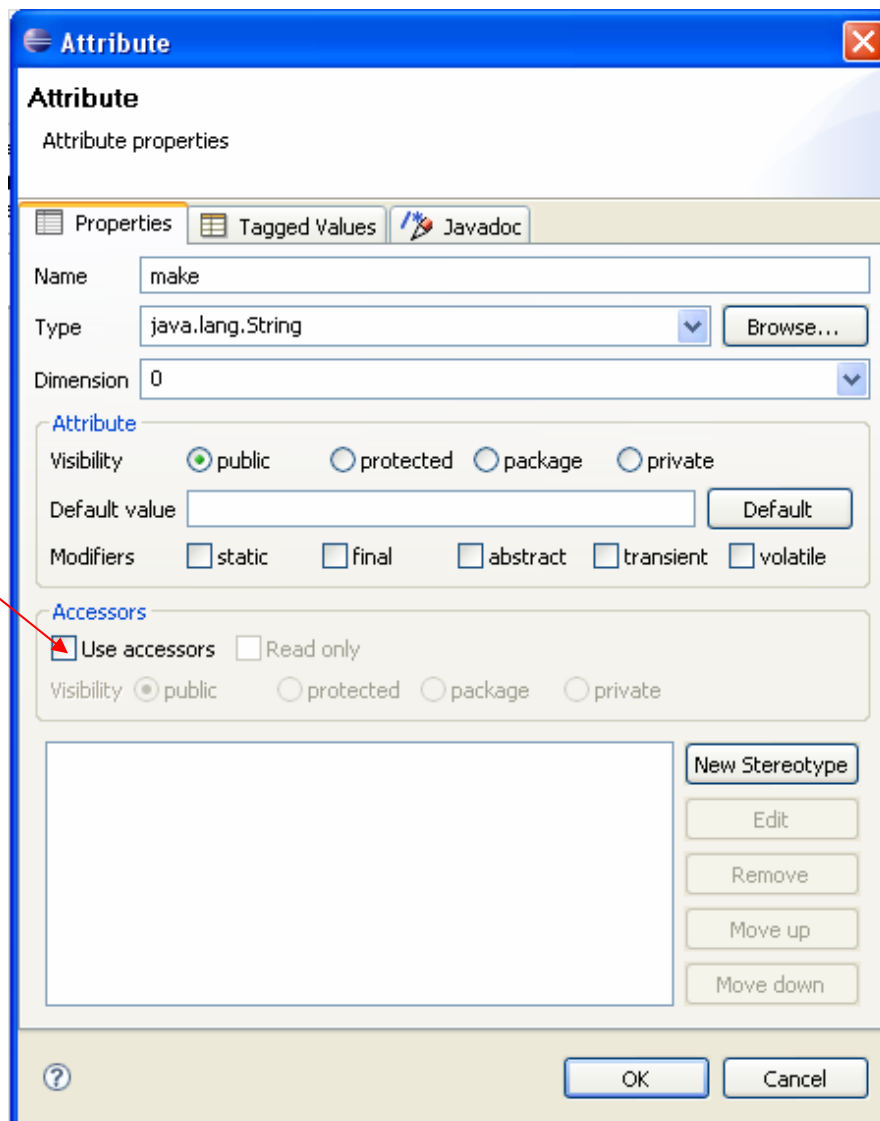
The class diagram modeling is composed by :

1. [Persistence modeling](#)
2. [Services modeling](#)
3. [The « controller » classes creation](#)
4. [The dependence links creation](#)

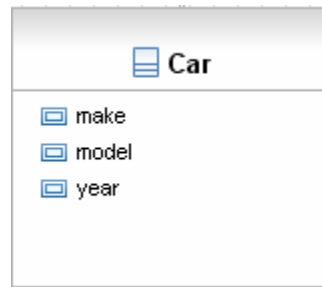
1. The persistence entity will be model by the class « car »

- The attributes are :
 - year (*type int*)
 - make (*type String*)
 - model (*type string*)

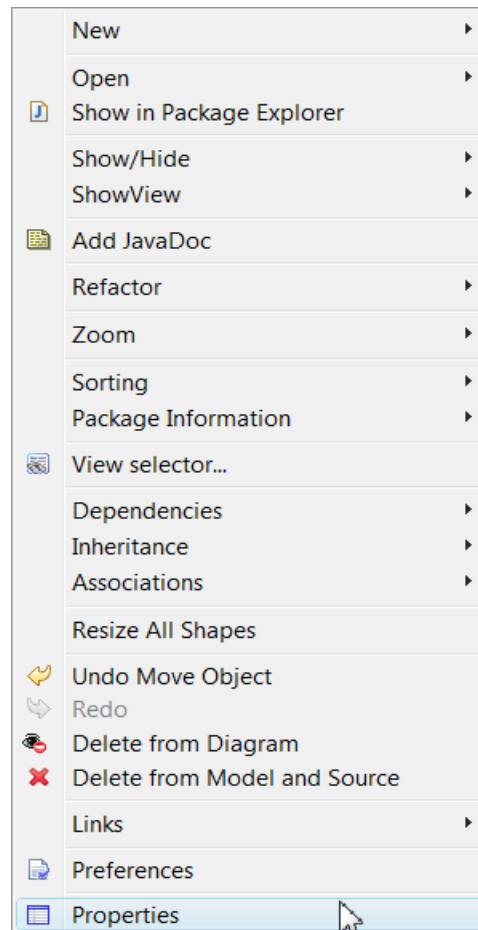
All these attributes should be public, and the « use accessor » check box should be unselected because accessors will automatically be generated in the JEE deployed code. If you select this option, then you will have double accessors in your AndromDA project.



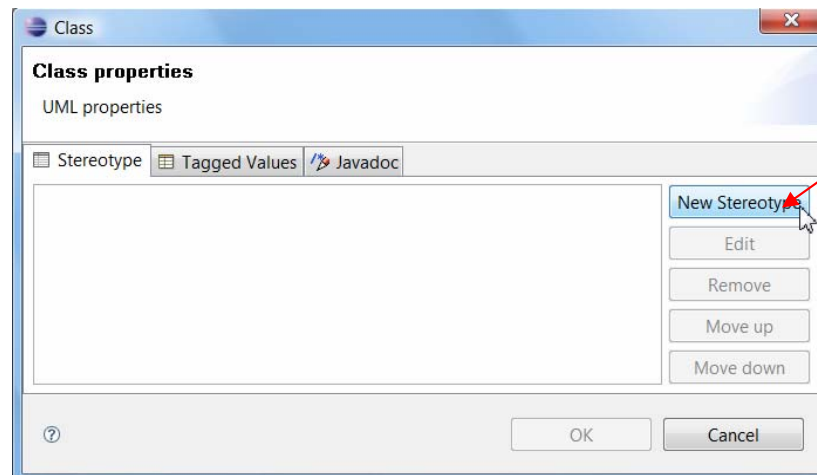
We should get the following class :



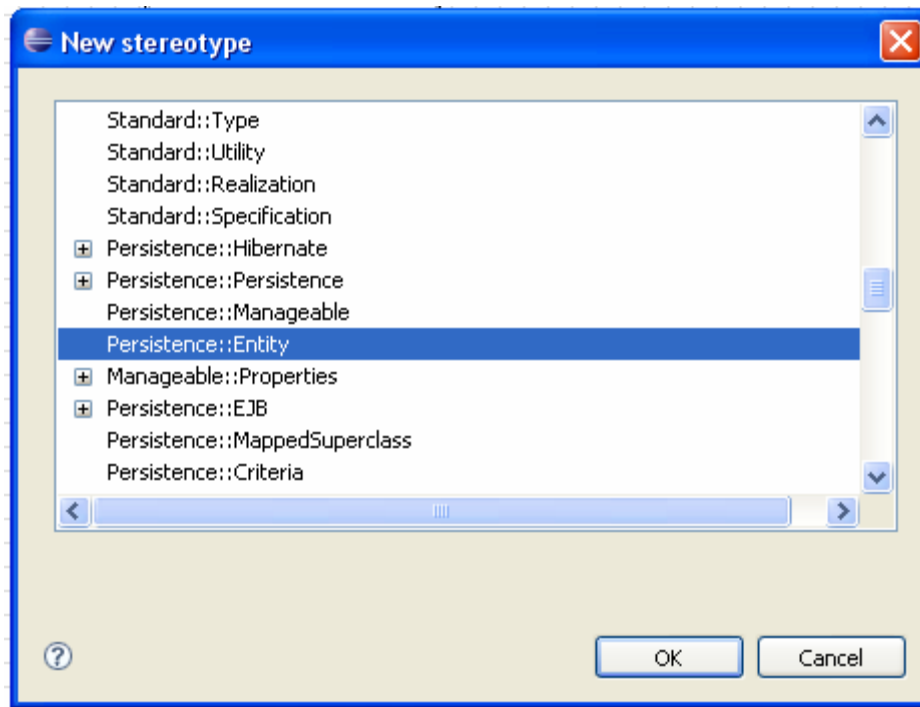
We now need to add the entity stereotype on this class.
Select the class to **open the class contextual menu > Property**



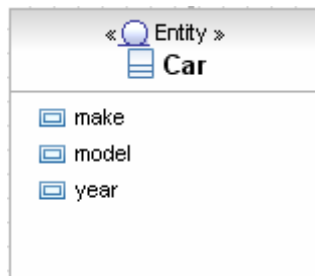
Click on New Stereotype



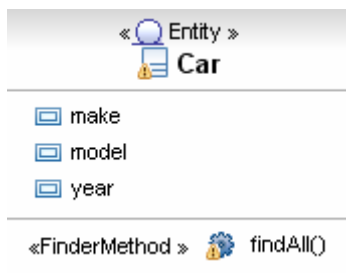
Select the entity stereotype and click on the OK Button



You should get the following class



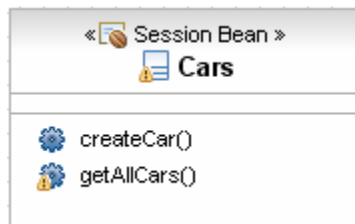
We now need to add the method « findAll » and add « Persistence :: FinderMethod » stereotype. The method return type is java.util.Collection and visibility is public



2. The service layer

It will be model by a class « cars » having the « Service » stereotype.

- The methods are :
 - « getAllCars » will call the findAll method from the Entity class named Car
 - « createCar » will call the add operation
 - This method is public, return type is void
 - Three parameters
 - Year (string)
 - Make (string)
 - Model (string)



We now need to create the « controller » classes in order to add the glue between the service and presentation layer.

Two classes are created :

- « ListCarsController »
 - Method « getAllCars » will load the list of available cars from the Class « Cars »
- « AddCarController »
 - Method « createCar » will allow adding new cars in the list

3. The « controller » classes creation :

Each « controller » class correspond to a usecase (see usecase diagram)

We therefore need to create two « controller » classes because we have two usecases.

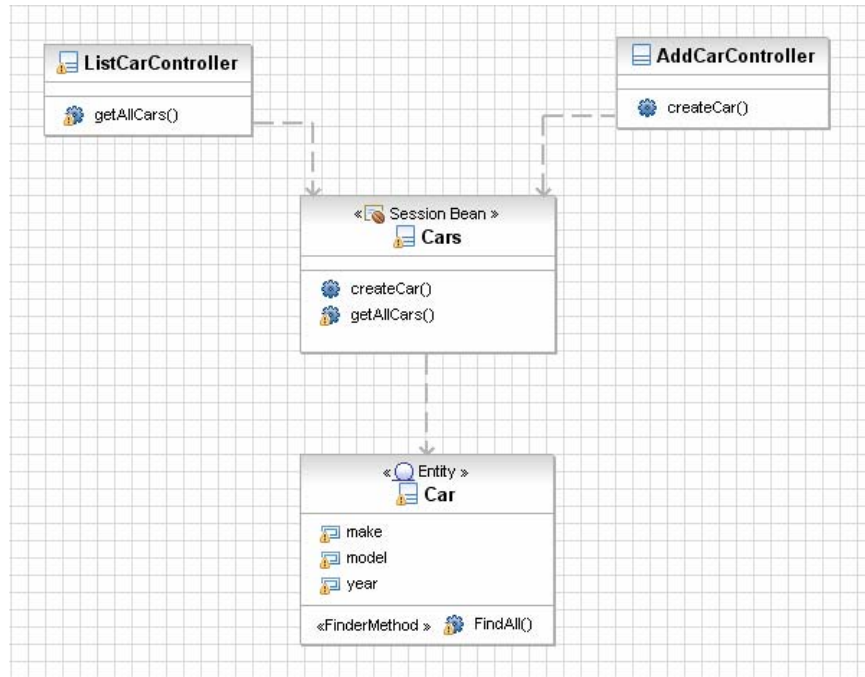
- The « AddCarController » class has one method which role is to call the services classes parameters in order to add a new car (*the session bean class Cars's parameters*).
 - Visibility is public
 - Return type : void
 - Parameters
 - Year type :int
 - Make type : string
 - Model type : string
- The «ListCarsController » class role is to get the car list and display the information by the presentation layer.
 - Visibility is public
 - Return type : void
 - Parameters type : java.util.collection

4. The dependence links creation :

Each dependence link allows to implement services from the target classe.

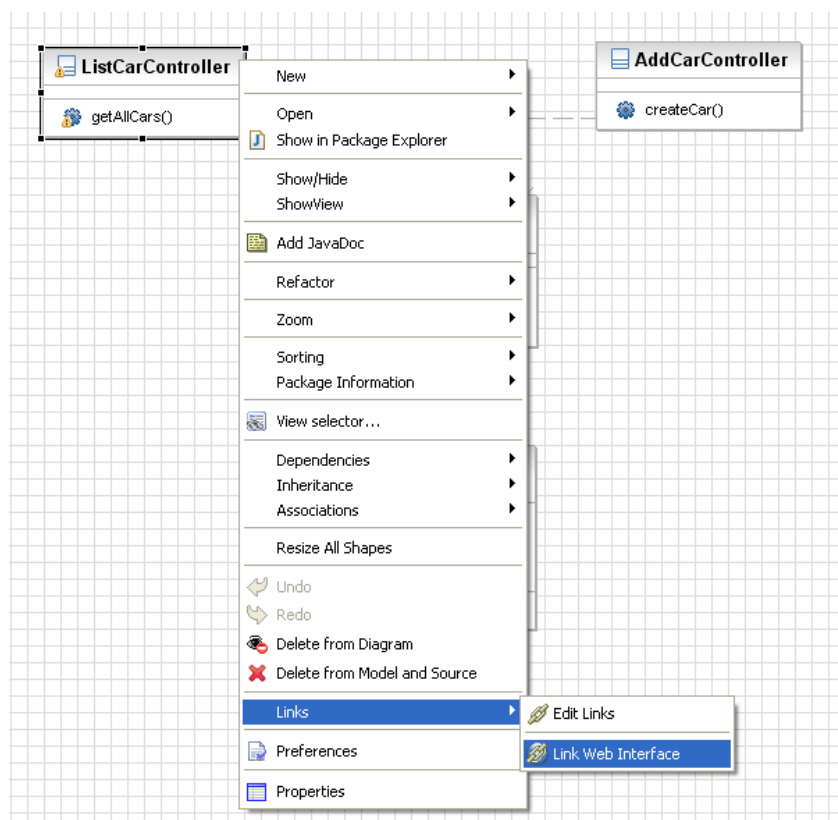
- The link between Car and Cars will add accessors to the DAO class. The DAO class (Data Access Object) is the the JEE design pater for data access operations.
- The links between the two controller classes and Cars (*session bean*) will the controller classes to get an instance and then to call methods from Cars.

You should get the following diagram :

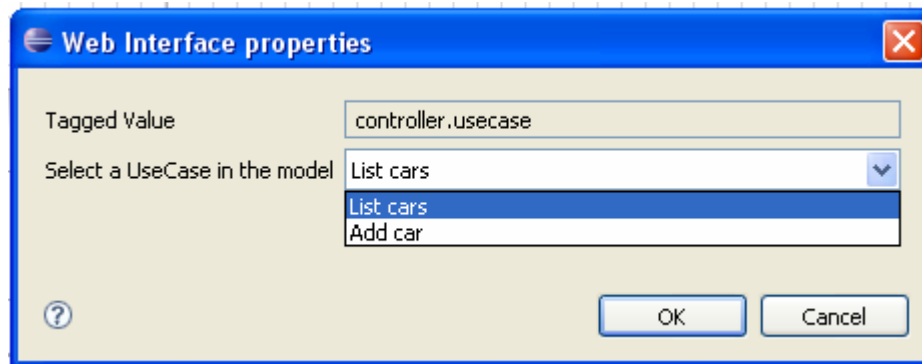


We now need to link the two controller classes to the related usecase.

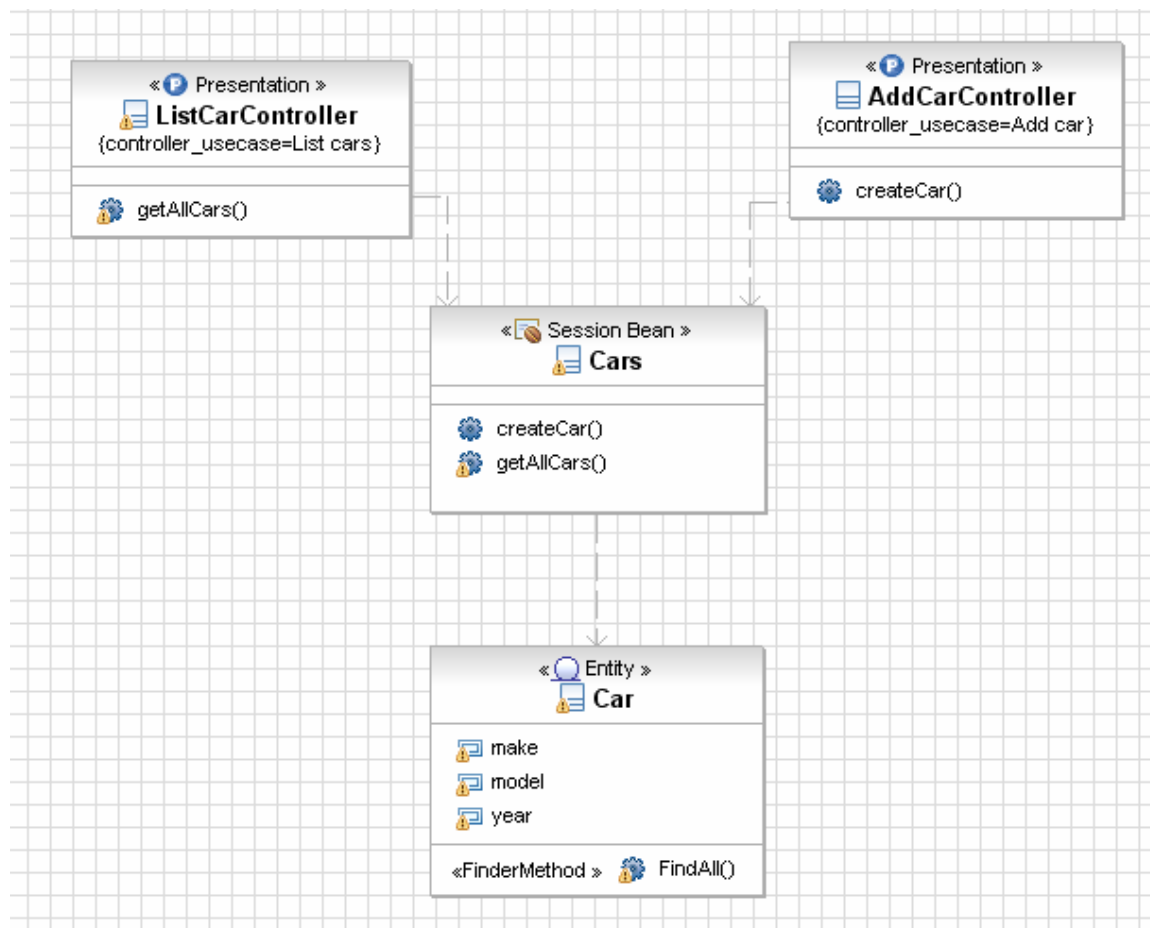
Click on one of the controller classes to **open the contextual menu > Links > Link Web Interface**



Select the usecase related to each controller class



You should get the following diagram :



d. Get the back-end information to be displayed in the Strut front tier application

We need to add the back-end information in the Strut front tier application. We will extend the state diagram in order to collect the persistence and service information from the class diagram.

The « ListCarActivity » State diagram is related to a usecase in order to model a system, and this usecase is related to a controller class in order to add service and persistence layers. We therefore need to extend the state to the controller in order to display back-end information coming from the class diagram to Strut front-tier presentation layer.

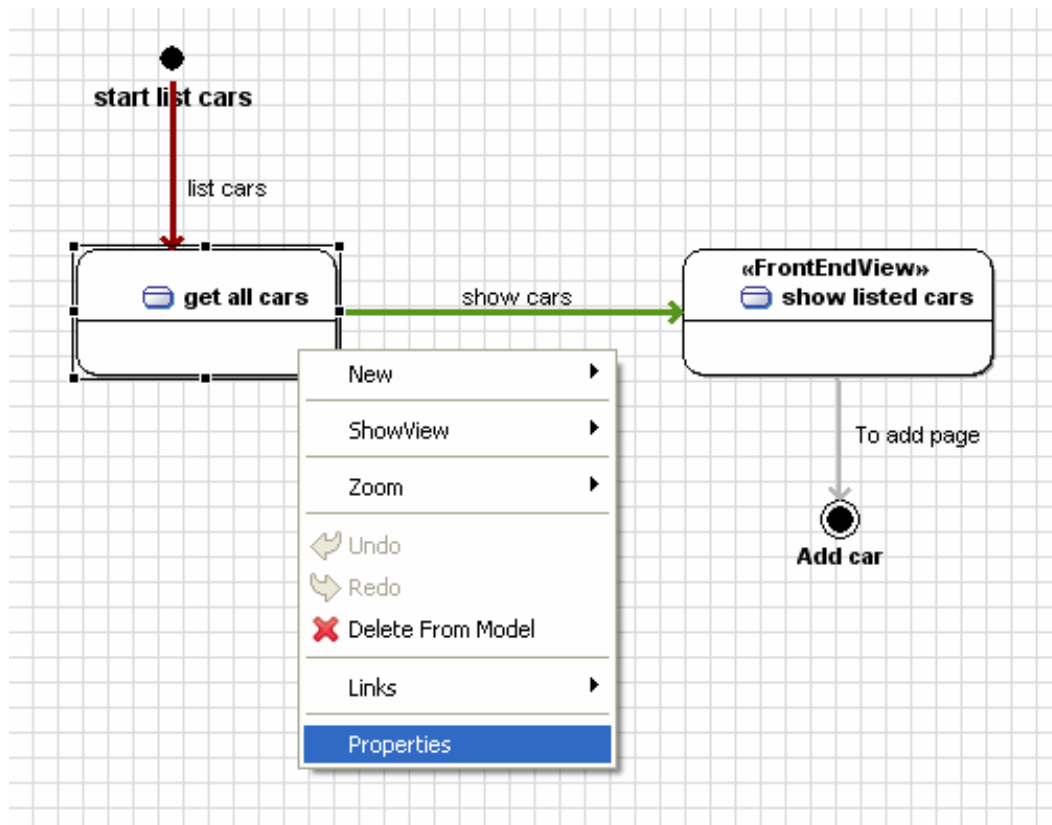
The get all cars state will call getAllCars method in order to display the list of available cars. We therefore need to extend the get all cars state to the needed method to be displayed.

This chapter is composed by the modification of :

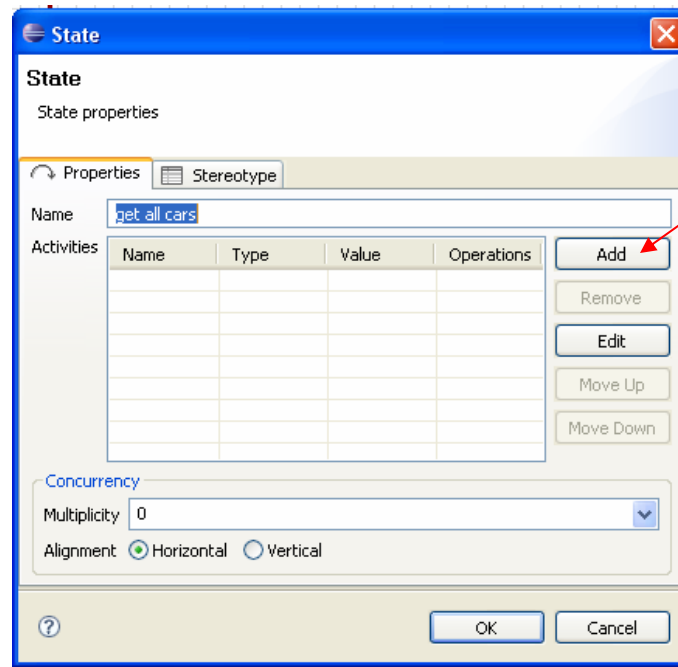
1. [The « ListCarActivity » State diagram](#)
2. [The « AddCarActivity » State Diagram](#)

1. The « ListCarActivity » State diagram

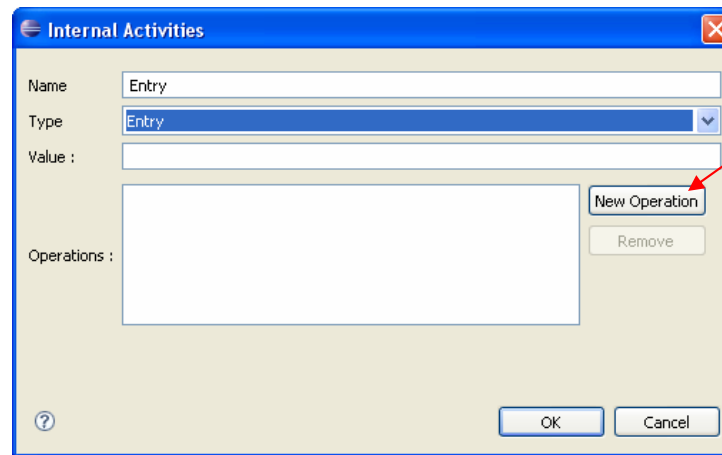
Select the get all cars state in the State Diagram > **open the contextual menu > Properties**



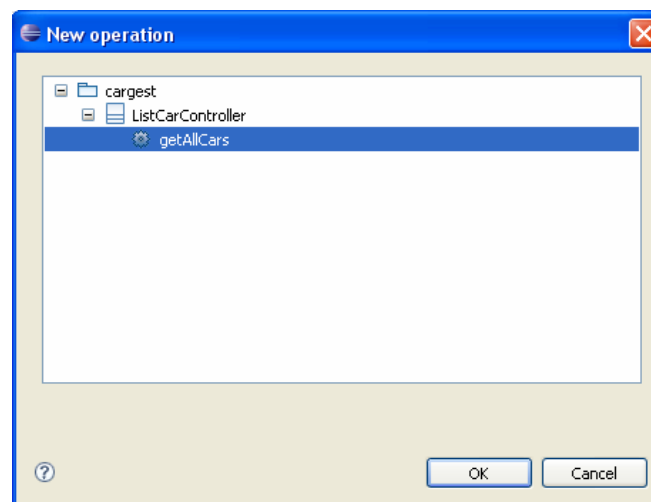
Select Add Button



We are going to add an Entry new operation to the state. **Click on the New Operation Button.** Please note that the type should be Entry in order to use the New Operation extension mechanism.

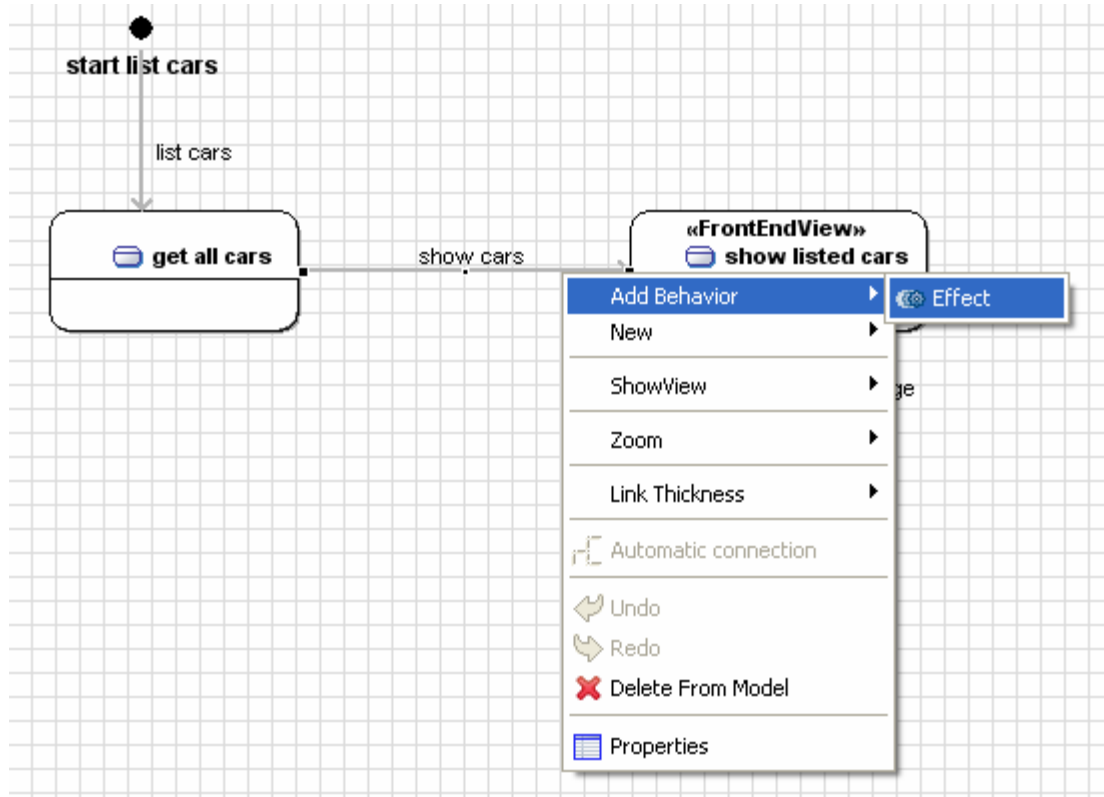


Select the getAllCars operation and click on the OK Button. Please Note that only class controllers 's methods will be displayed in the New Operation wizard selection.



The get all cars state can now list all the cars. , The next step is therefore to display this information inside the show listed cars. Please note that the stereotype «frontEndView» should be added in order to display this information.

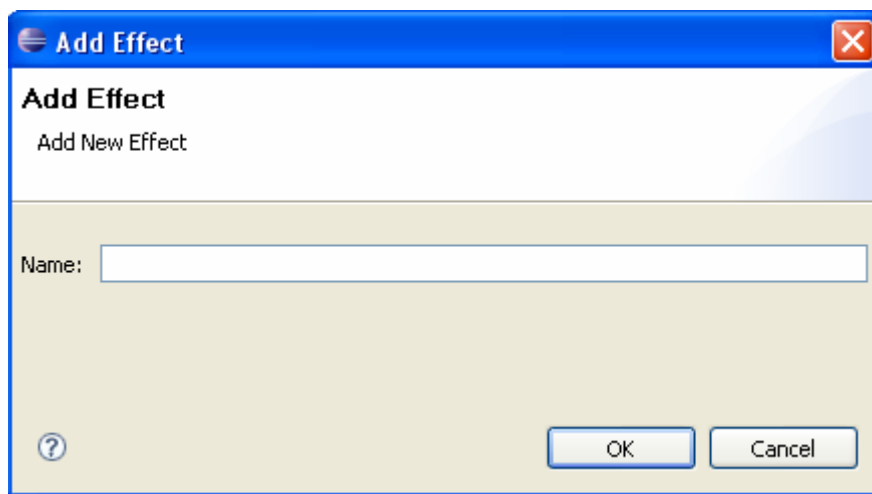
Select the transition link > **Add Behavior** > **Effect**



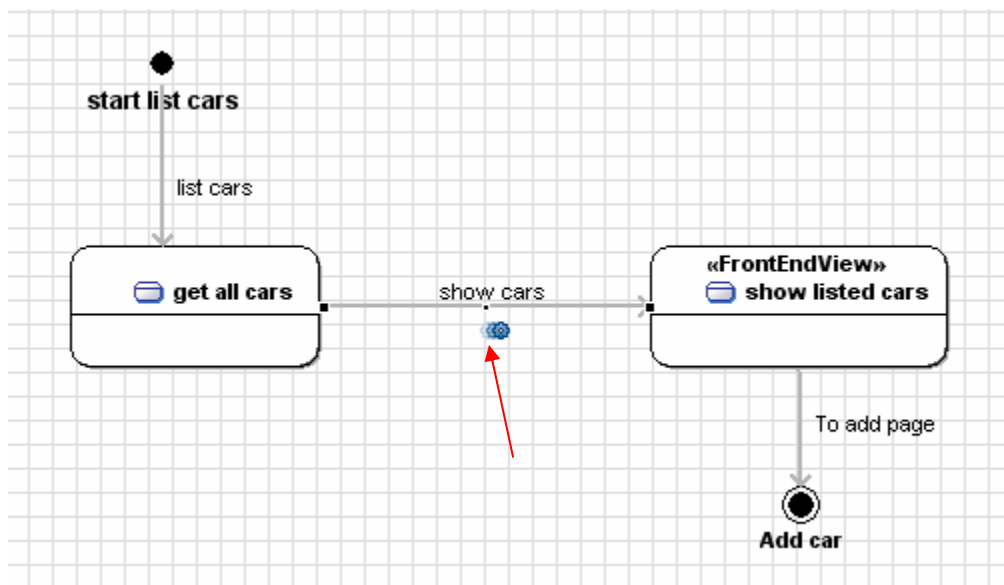
Type the name of the effect.

Don't forget that this name will be the button name of the generated Strut code.

If you leave the name empty, then no button will be created and the transition will only transmit a page of parameters to the following step.

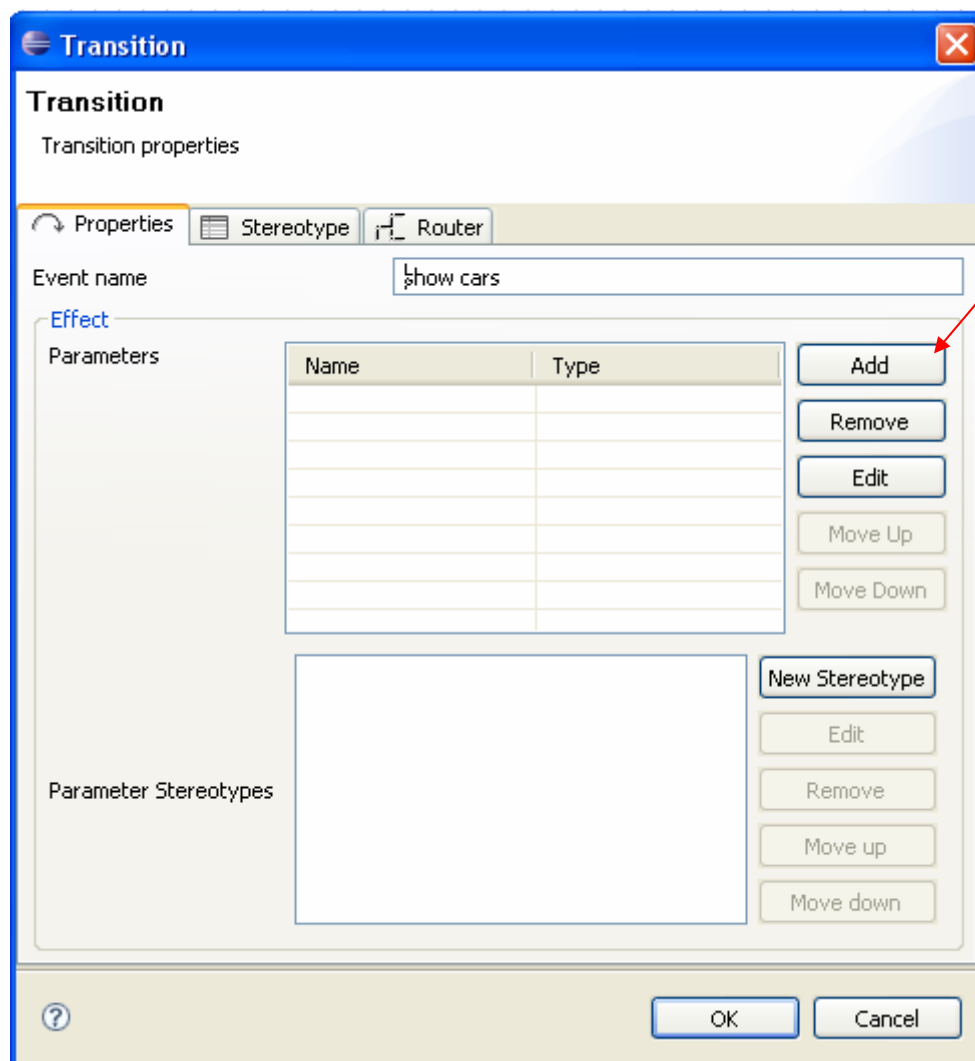


You should get the following diagram :

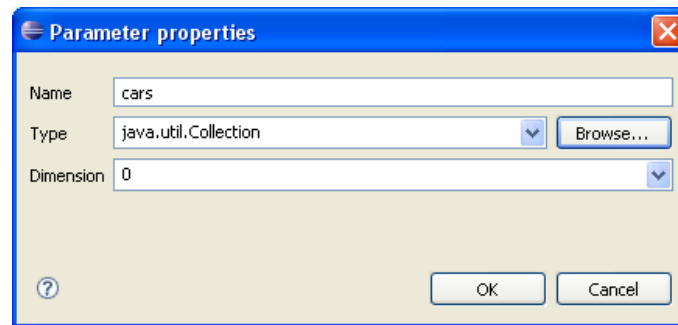


Click on the Transition > Properties > Effect Parameters > Add Button

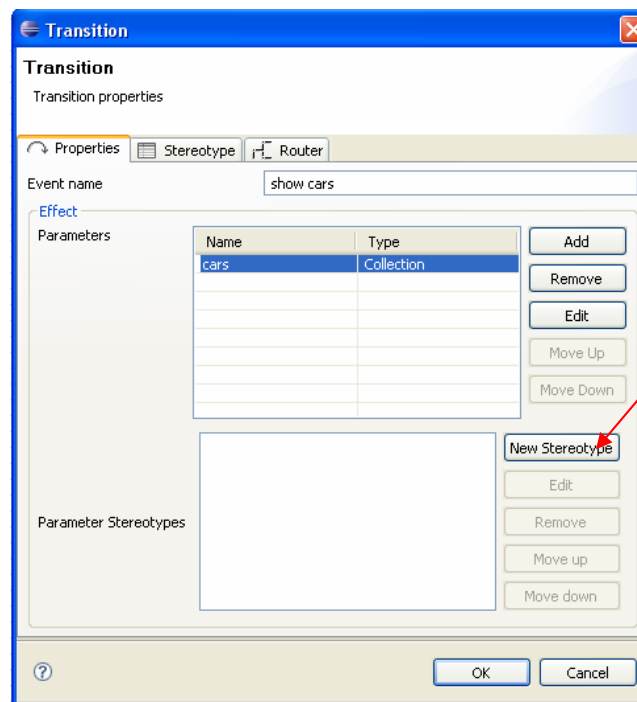
Note that you should create an effect if you want to be able to use this mechanism.



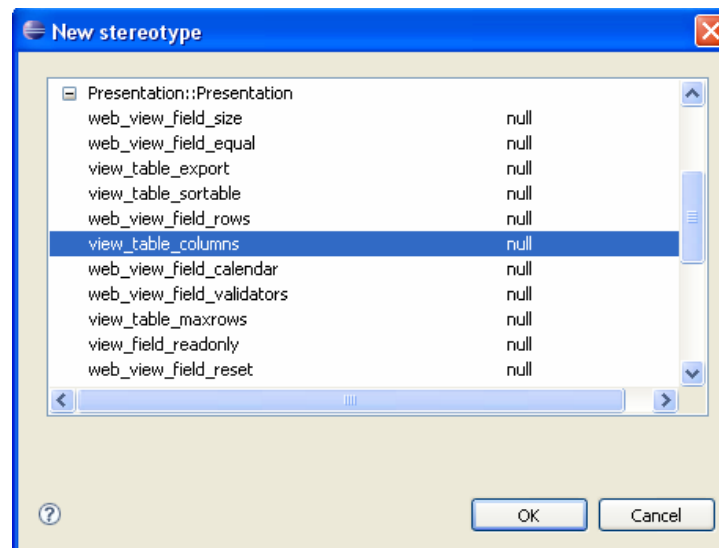
Enter the name and the type of the parameter and click on the OK Button.
Note that the name and the type should be the same as the controller class « ListCarController » method named « getAllCars ».



Click on **New Stereotype Button**. We are going to add the « Presentation :: view_table_columns » stereotype in order to set up the table property which will be displayed.



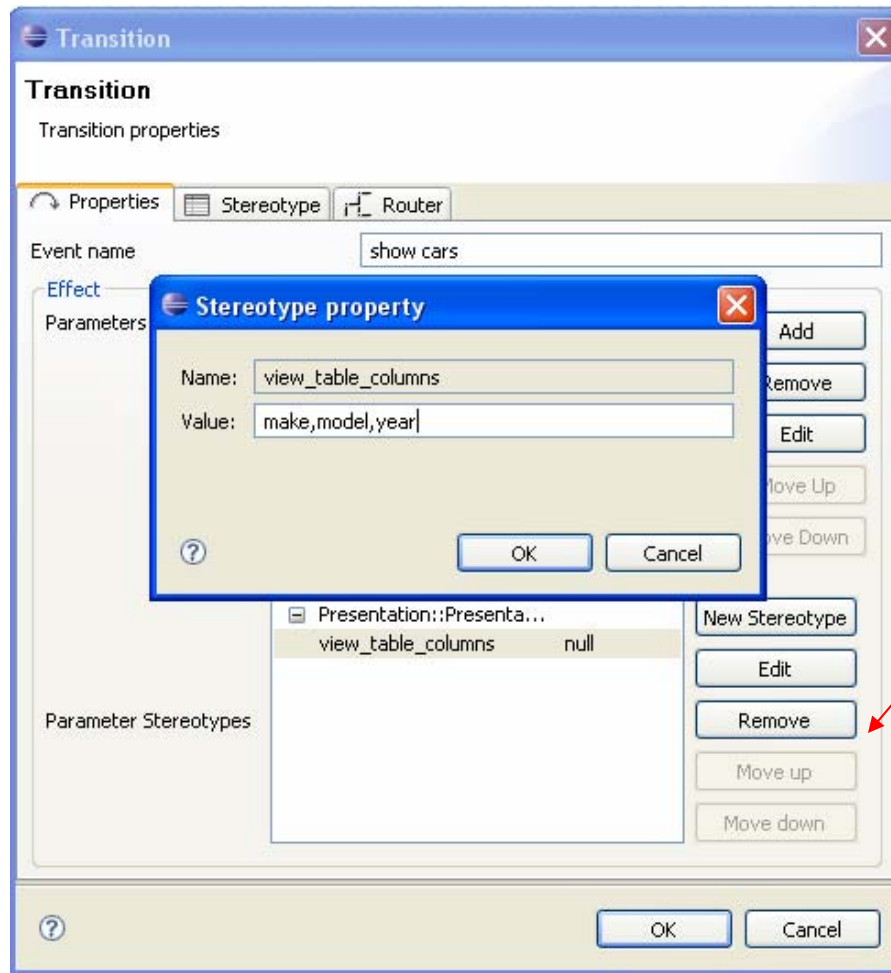
Select the « view_table_columns » stereotype and click on the OK Button.



Click on the Edit Button in order to open the Stereotype property.
Add Values which will be displayed and click on the OK Button:

- Make,model,year

Each value should be separated by a comma.



As explained before, we will first display the list of cars and now we are going to allow to add a car in the list.

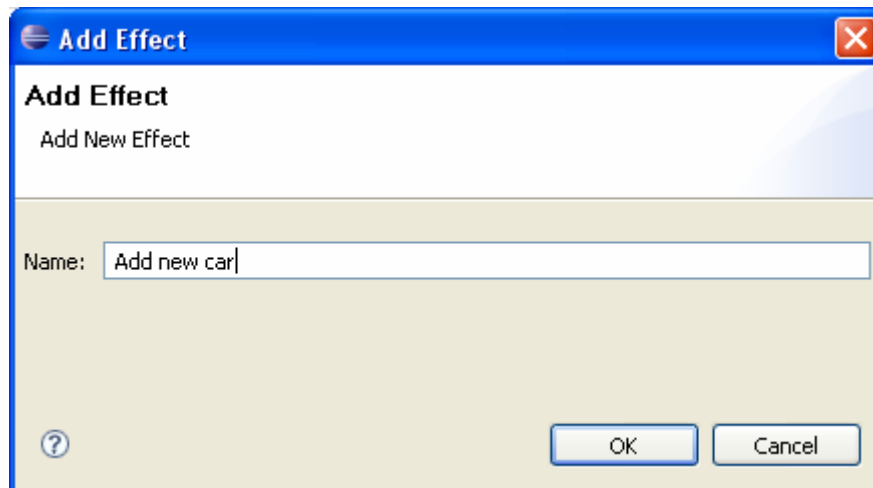
We therefore need a « Add new car » button and to allow the page transition to the other State diagram which is modeling the Add new car workflow.

Select the transition link named « To add page » **Add Behavior > Effect**

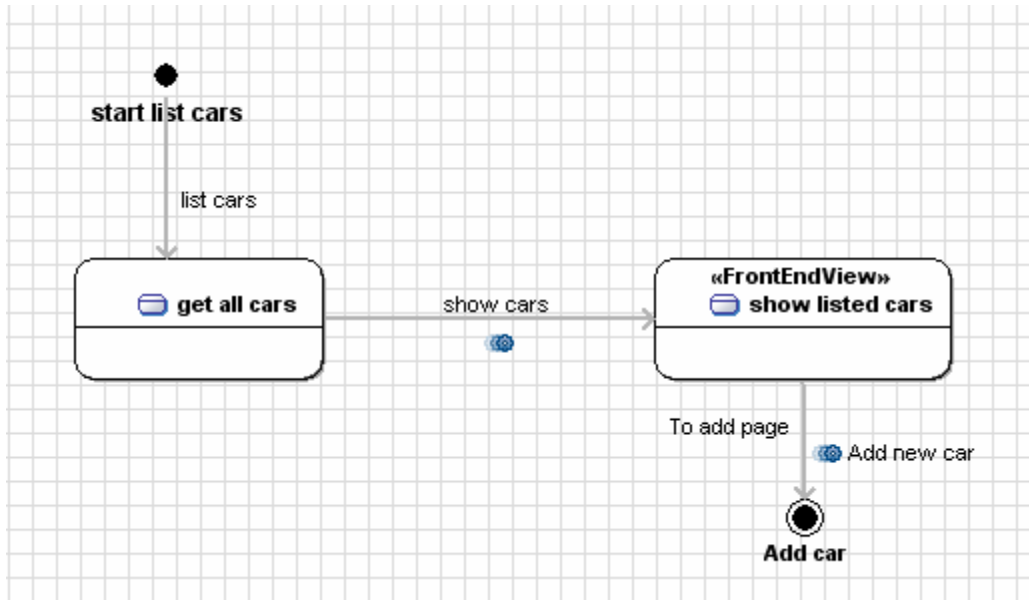
Enter the name of the button which will be created in the name field.

Enter « Add new car » in the Name field and click on the OK Button.

The End point name should be the same as the usecase in order to define the scope of this behavior.



The « ListCarActivity » State Diagram should be as below :



2. The « AddCarActivity » State Diagram

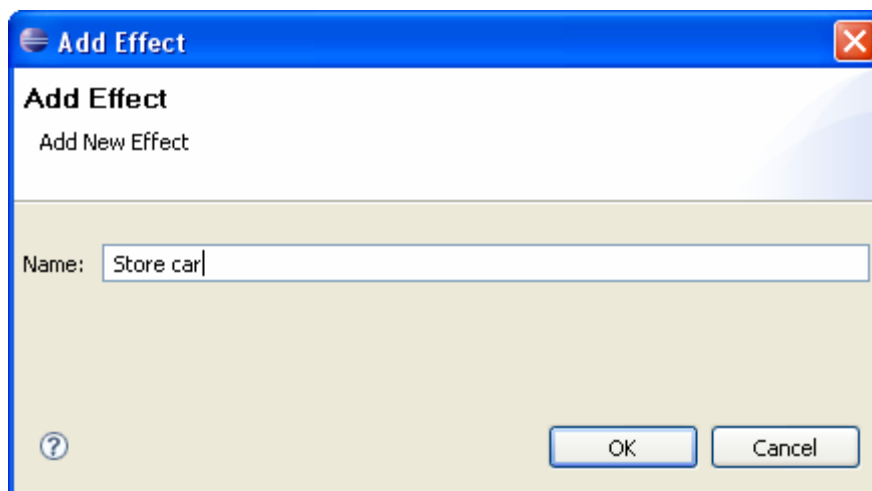
It will model the workflow of adding a new car in the list and then make a transition to the other State diagram which will display the information.

We always need to add « « Presentation :: FrontEndView » » stereotype in order to allow the creation of the customer fields which will be edited in the Strut page.

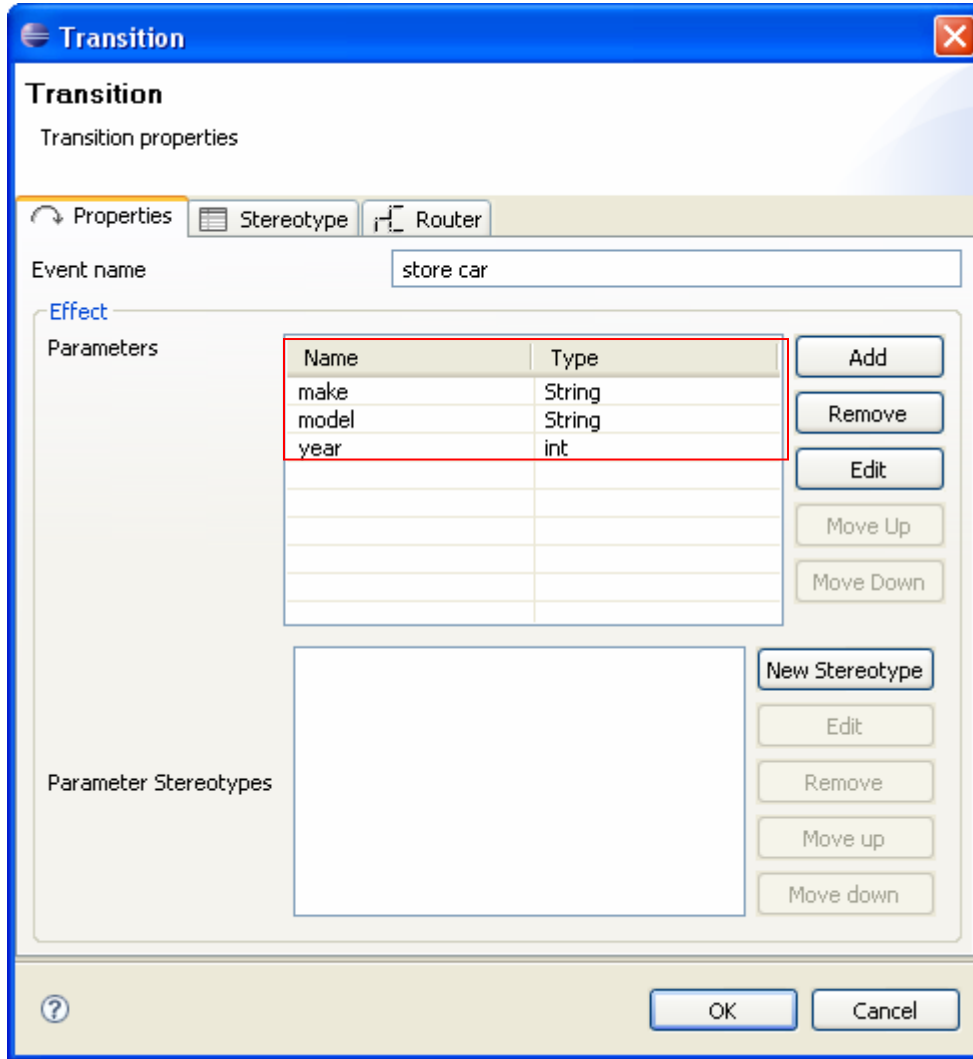
Please note that in order to push the information edited by the user to the back-end tier, we need to use the « AddCarController » class « createCar » method . This is why we need to add an « Entry » to the « get all cars » state.

Select the transition named « store car » > **Add Behavior > Effect**

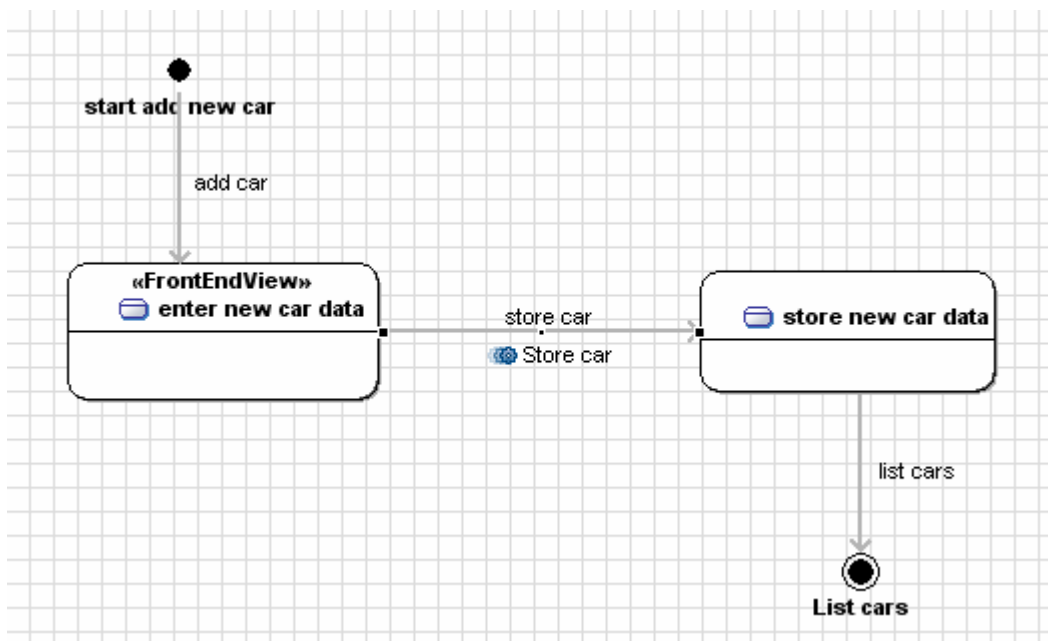
Type the name which will be the Button name in the Name field and Click on the OK Button.



If you look at the transition property, then you will notice that they will be the same as the « createCar » operation. This information is automatically added.



You should get the following diagram :



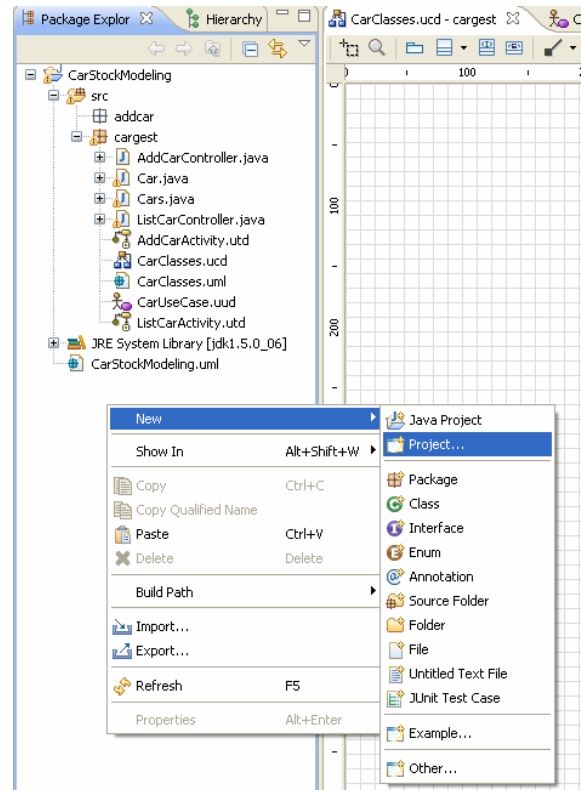
2. JEE Code Generation

This chapter is composed by :

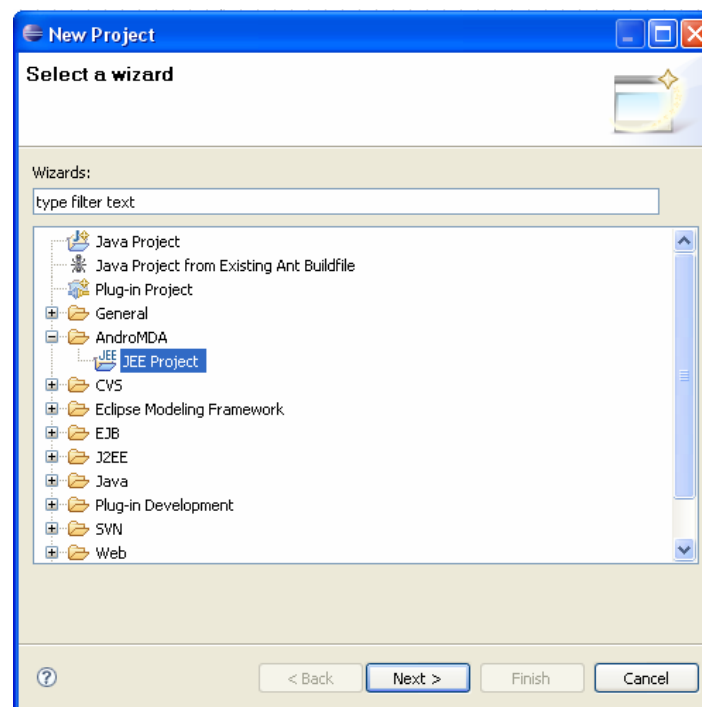
1. [Create an AndroMDA project](#)
2. [Generate the code inside the AndroMDA Project](#)
3. [Manually Implement methods and logic](#)

1. Create an AndroMDA project

Click in the package Explorer > New Project



Select AndroMDA > JEE Project and click on the Next Button



Enter the project Name and other needed information and click on the Next Button

The screenshot shows the 'New AndromDA Project' dialog box with the title 'Create an AndromDA project'. The instructions state: 'Create an AndromDA project in the workspace or in an external location.' The 'Project name (ID):' and 'Friendly project name:' fields both contain 'CarStock'. Under the 'Contents' section, the radio button for 'Create new project in workspace' is selected. The 'Directory:' field shows 'D:\eclipse33_M7\runtime-New_configuration(1)\CarStock' with a 'Browse...' button. The 'Basic Project Informations' section includes 'Root Package: CarStock', 'Version: 1.0.0', and 'Author: Omondo'. At the bottom, there are buttons for '< Back', 'Next >', 'Finish', and 'Cancel'.

Select your Model layer Component Spring 2 + Hibernate 3 & ear for your Type of Application, then click on the next Button.

The screenshot shows the 'New AndromDA Project' dialog box with the title 'Specify your project features'. Under the 'Model layer Component' section, the radio button for 'Spring' is selected, with '2' in the dropdown menu. The 'Hibernate' dropdown menu is set to '3'. Under the 'Type of Application' section, the radio button for 'ear' is selected. At the bottom, there are buttons for '< Back', 'Next >', 'Finish', and 'Cancel'.

Set up the Database information and click on the Next Button.

The screenshot shows the 'New AndroMDA Project' dialog box with the title 'Specify your Data Base informations'. It contains the following fields and options:

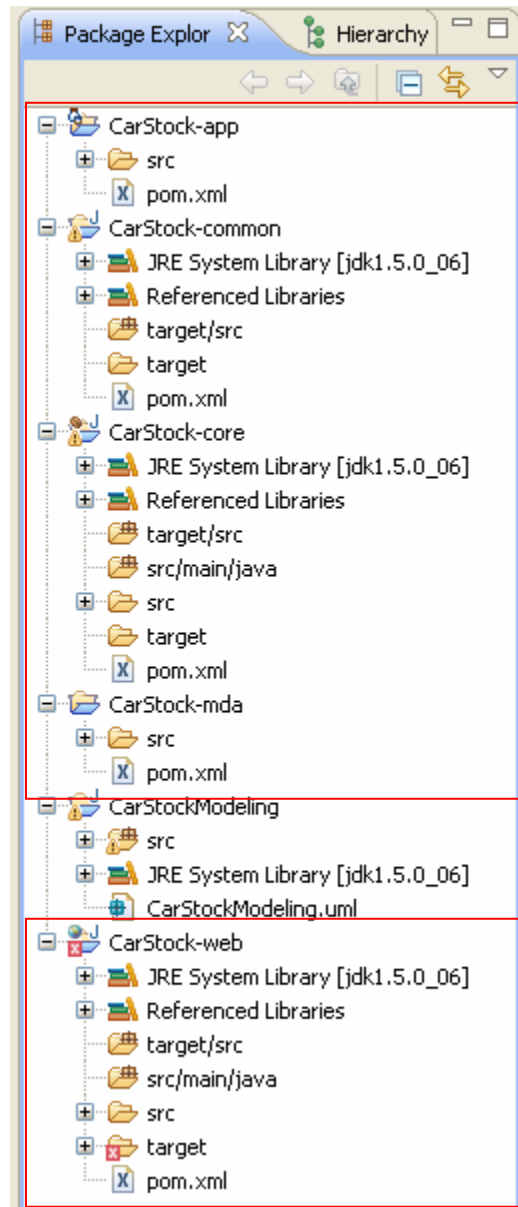
- Select A Database:** A dropdown menu with 'Hypersonic' selected.
- Data Base Informations:** A section containing four text input fields:
 - Data Base name: (empty)
 - User name: 'Omondo'
 - Password: 'Omondo'
 - SID (for oracle): (empty)
- Navigation:** A row of buttons at the bottom: '< Back', 'Next >', 'Finish', and 'Cancel'. A help icon (?) is located to the left of the 'Back' button.

Select a Web Interface and click on the Finish Button.

The screenshot shows the 'New AndroMDA Project' dialog box with the title 'Select the components of the Presentation Tier'. It contains the following options and fields:

- Web Interface Selection:** Two radio buttons:
 - I don't want to have a Web Interface
 - I want to have a Web Interface
- Web Tier Technology:** A section containing two radio buttons and a dropdown menu:
 - Struts
 - JSF
 - View type for JSF: 'jsp' (dropdown menu)
- JBoss Seam:** A section containing one checkbox:
 - Enable JBoss Seam
- Navigation:** A row of buttons at the bottom: '< Back', 'Next >', 'Finish', and 'Cancel'. A help icon (?) is located to the left of the 'Back' button.

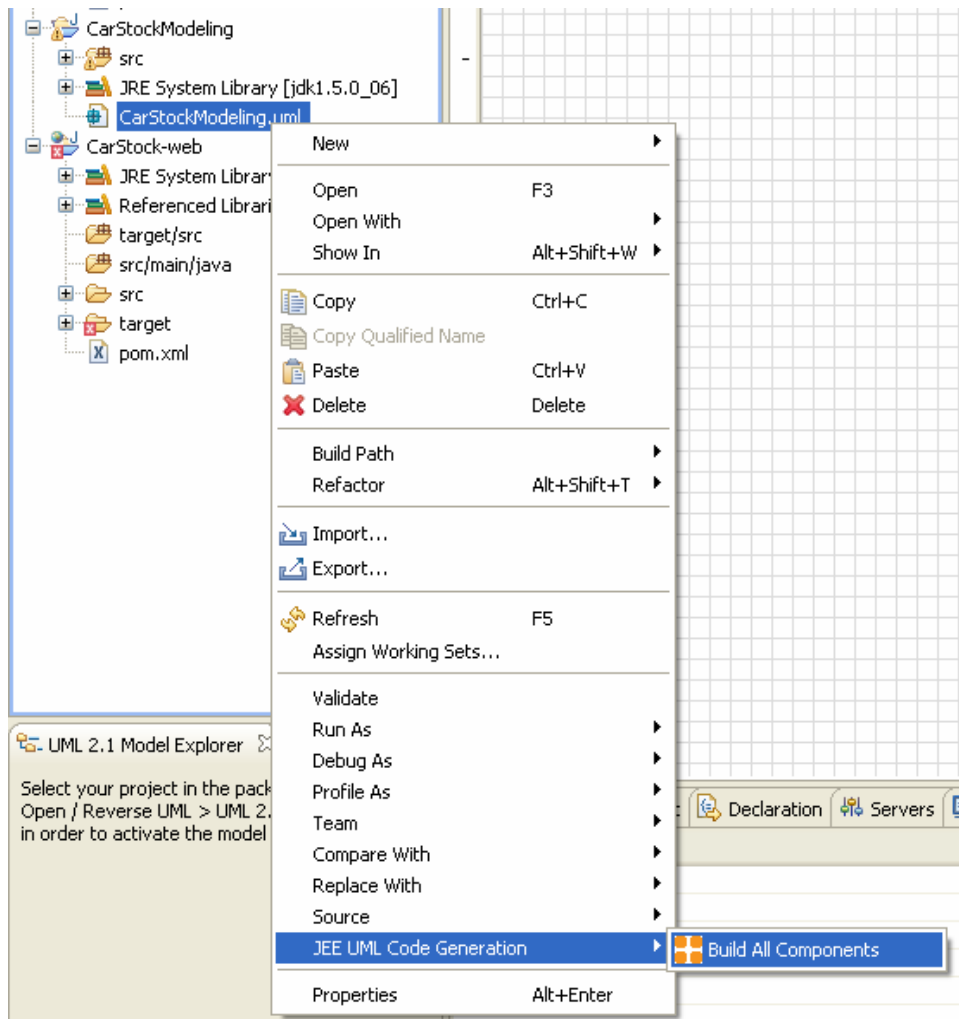
The following JEE project structure has been created in your package explorer. This project structure is still empty, waiting for the code generation mechanism from the UML 2.1 model to fulfill all the packages with spring, hibernate, struts and java code.



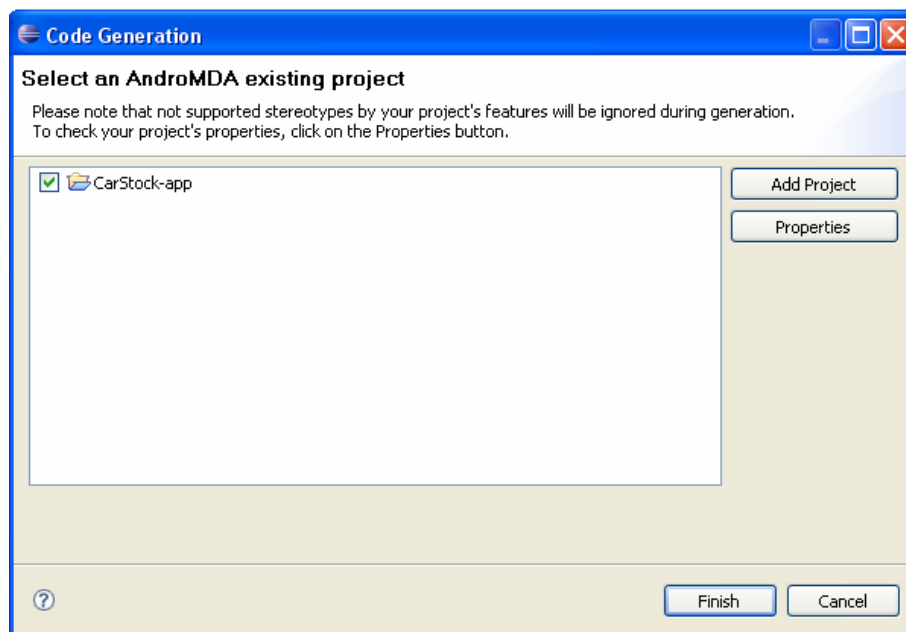
2. Generate the code inside the AndroMDA project

The code generation is coming from the project model.

Select the « carStockModeling.uml » file in your project explorer > **JEE UML Code Generation > Build All Components**



Select the project in which the code will be deployed.



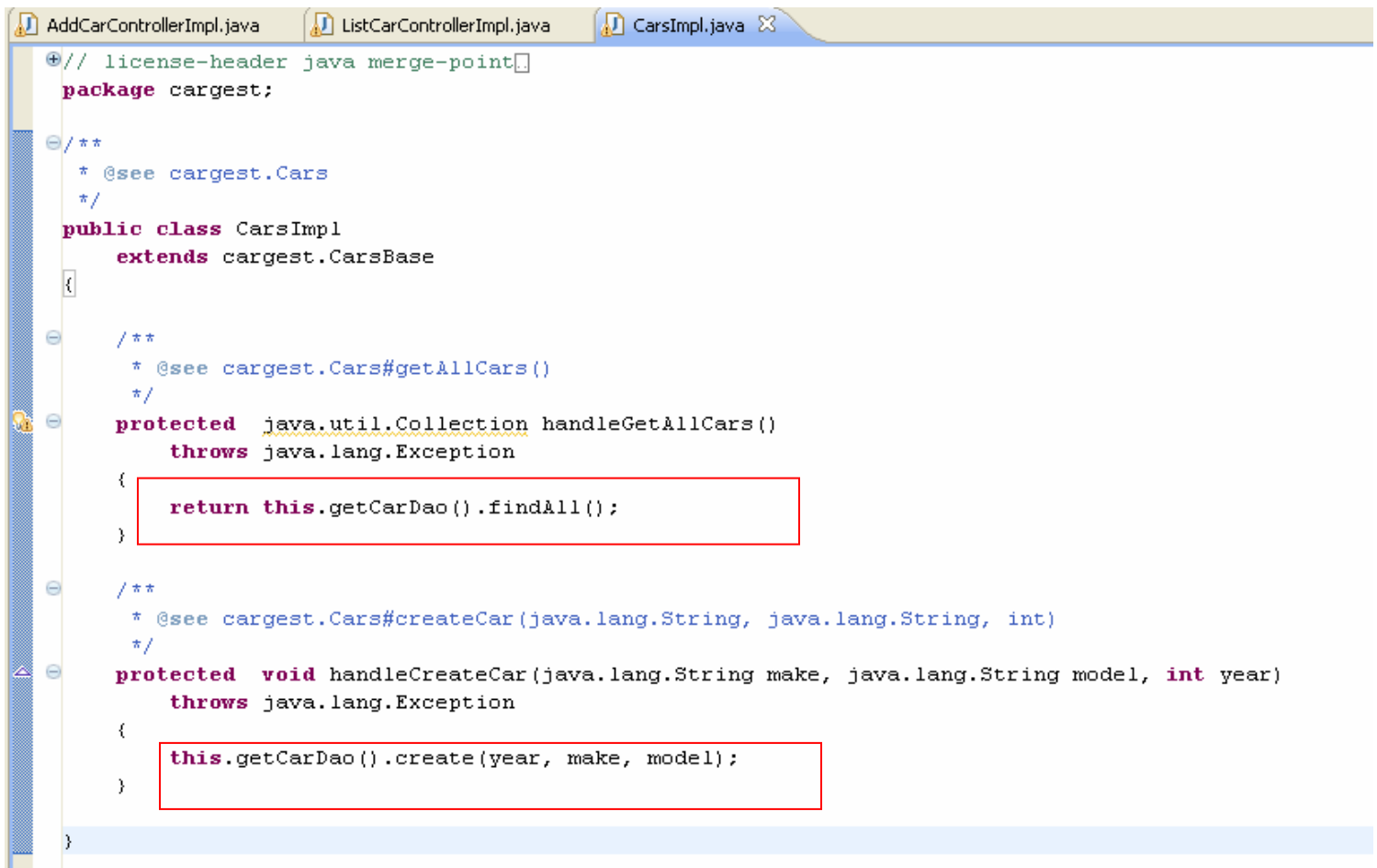
This is the maven console showing the code generation process :

```
Maven Console
[INFO] -----
[INFO] ear:generate-application-xml
[INFO] resources:resources
[INFO] ear:ear
[INFO] install:install
[INFO] andromdapp:deploy (execution: default)
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 3 second
[INFO] Finished at: Thu Aug 09 09:56:08 CEST 2007
[INFO] Memory 52M/242M
[INFO] -----
```

3. Manually Implement methods and logic

The last step is the manual implementation of some methods in order to complete the implementation logic. These methods are not automatically implemented because AndroMDA considers that the user can implement the method using different approaches. All the non implemented methods are located in the src/main/java package inside the classes having the end of the named finished by «Impl ». It represents less than one percent of all the generated code.

We need to implement two methods « handleGetAllCars » and « handleCreateCar » from the class named « CarsImpl »



```
AddCarControllerImpl.java | ListCarControllerImpl.java | CarsImpl.java X
+// license-header java merge-point
package cargest;

/**
 * @see cargest.Cars
 */
public class CarsImpl
    extends cargest.CarsBase
{

    /**
     * @see cargest.Cars#getAllCars()
     */
    protected java.util.Collection handleGetAllCars()
        throws java.lang.Exception
    {
        return this.getCarDao().findAll();
    }

    /**
     * @see cargest.Cars#createCar(java.lang.String, java.lang.String, int)
     */
    protected void handleCreateCar(java.lang.String make, java.lang.String model, int year)
        throws java.lang.Exception
    {
        this.getCarDao().create(year, make, model);
    }
}
```

We also need to implement the method «createCar» from the class named «AddCarControllerImpl»

```
AddCarControllerImpl.java | ListCarControllerImpl.java | CarsImpl.java
// license-header java merge-point
package addcar;

import org.apache.struts.action.ActionMapping;

/**
 * @see addcar.AddCarController
 */
public class AddCarControllerImpl extends AddCarController
{
    /**
     * @see addcar.AddCarController#createCar(org.apache.struts.action.ActionMapping, addcar.CreateCarForm, j
     */
    public final void createCar(ActionMapping mapping, addcar.CreateCarForm form, HttpServletRequest request,
    {
        try
        {
            this.getCars().createCar(form.getMake(), form.getModel(), form.getYear());
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }
}
```

Finally, we need to implement the method «getAllCars» from the class «ListCarController»

```
AddCarControllerImpl.java | ListCarControllerImpl.java | CarsImpl.java
// license-header java merge-point
package cargest;

import org.apache.struts.action.ActionMapping;

/**
 * @see cargest.ListCarController
 */
public class ListCarControllerImpl extends ListCarController
{
    /**
     * @see cargest.ListCarController#getAllCars(org.apache.struts.action.ActionMapping, cargest.GetAllCarsForm,
     */
    public final void getAllCars(ActionMapping mapping, cargest.GetAllCarsForm form, HttpServletRequest request,
    {
        try
        {
            form.setCars(this.getCars().getAllCars());
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
            throw new RuntimeException(ex);
        }
    }
}
```

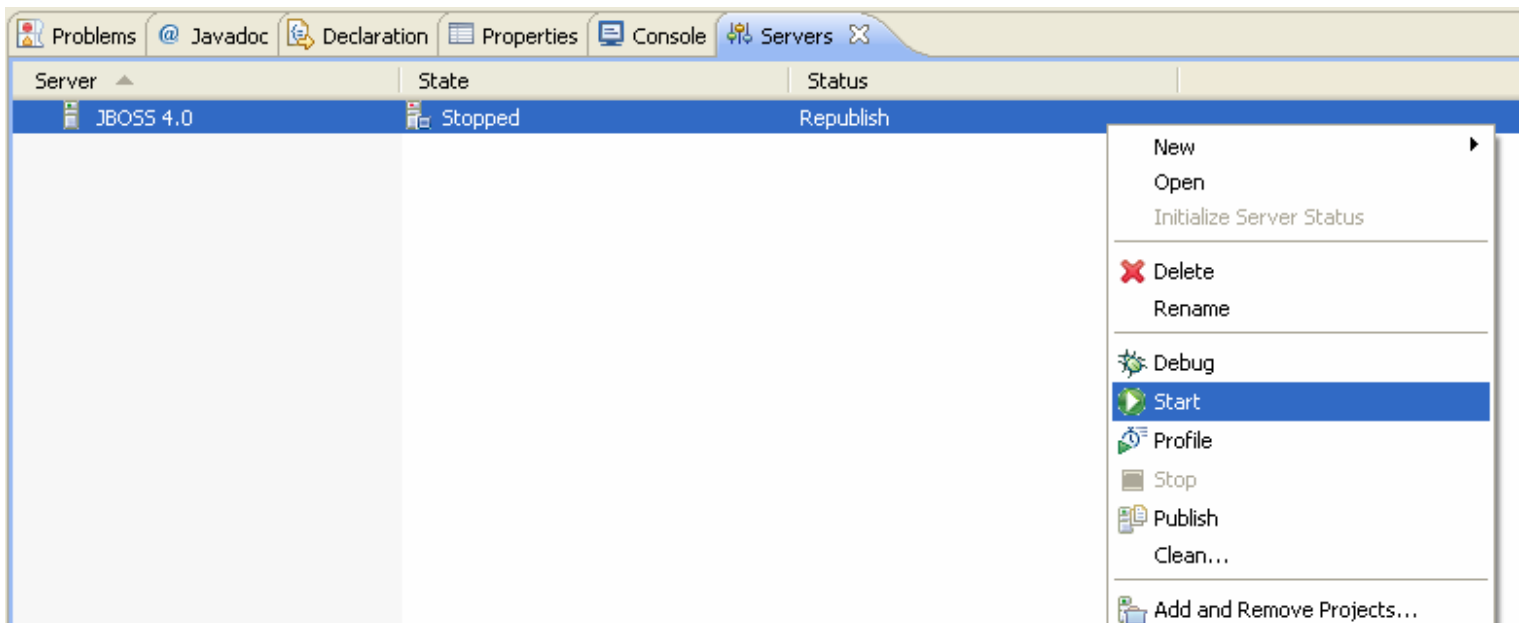
Now that the method classes has been implemented, we need to generate the project again in order to add logic between the new implemented class and the JEE code generation. Please note that the new generation will not erase your new methods or classes, but only erase the code previously created by AndroMDA code generation. You can therefore generate as many time as needed your code from your model inside the same project.

Here is the maven console showing the code generation

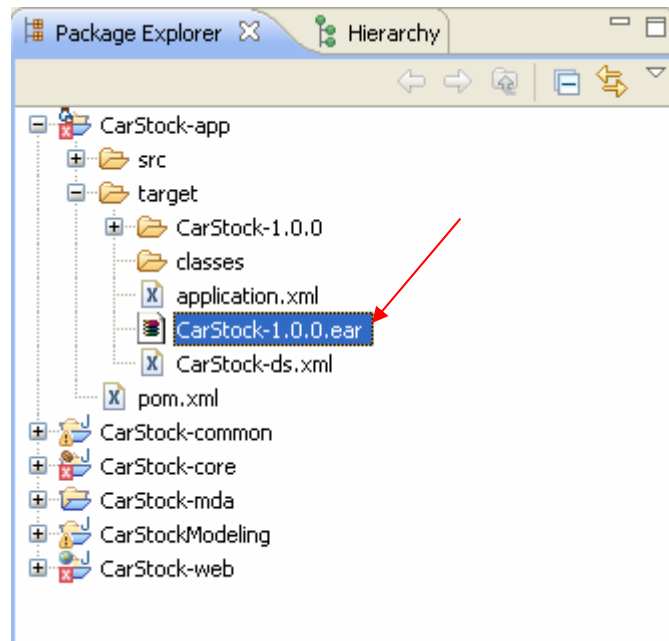
```
Maven Console
[INFO] Memory 83M/314M
[INFO] -----
[INFO] ear:generate-application-xml|
[INFO] resources:resources
[INFO] ear:ear
[INFO] install:install
[INFO] andromdapp:deploy (execution: default)
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 4 second
[INFO] Finished at: Thu Aug 09 10:35:37 CEST 2007
[INFO] Memory 83M/314M
[INFO] -----
```

3. Launch the JEE Server and immediately see your model running on a JBoss server

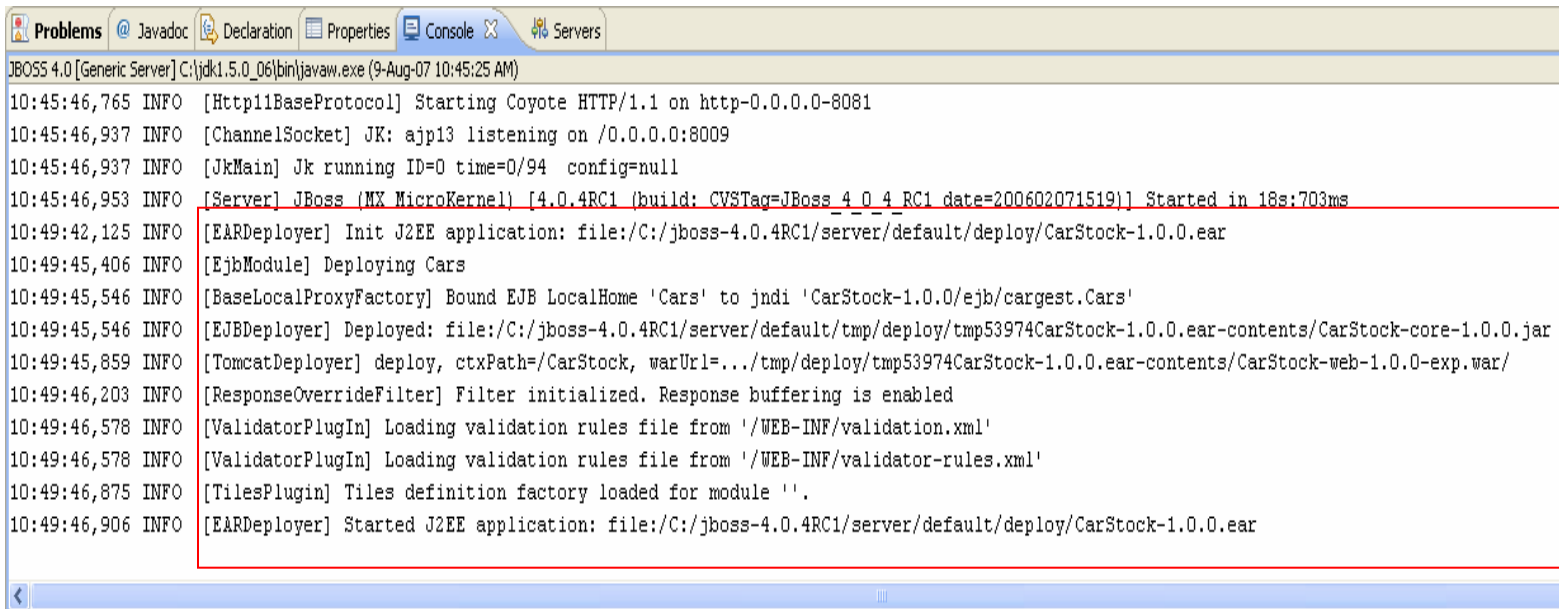
We need to launch the JBoss server.
Go to the server view, click on the IBoss 4.0 server > Start



Copy the file named « CarStock-1.0.0.ear » into the \$JBoss_HOME\$/server/default/deploy folder



The Eclipse console should show the following message



Select the « Open Web Browser » in the toolbar



Enter the following address:

<http://localhost:8080/CarStock>

The following page should be displayed :

Show listed cars

Show listed cars

Add new car

Add new car

4 items found, displaying all items.

1

Make	Model	Year
bacem	mazda	2000
c3	citroen	2000
309	peugeot	1990
e200	marcedes	2004

Export options: [CSV](#) | [Excel](#) | [XML](#) | [PDF](#)

[Help](#)

Click on the Add new car Button to go to the other web page.

Show listed cars > Enter new car data

Enter new car data

Store car

Make

Model

Year

Store car

Fields marked with an asterisk are required

[Help](#)

Enter the new car description and click on the Store Car Button.

This operation will save the new data in the Hypersonic database and will then return to the first page.

You should have the following list car page

Show listed cars > Enter new car data > Show listed cars

Show listed cars

Add new car

Add new car

5 items found, displaying all items.

1

Make	Model	Year
bacem	mazda	2000
c3	citroen	2000
309	peugeot	1990
e200	mercedes	2004
Omondo	Omondo	2007

Export options: [CSV](#) | [Excel](#) | [XML](#) | [PDF](#)

[Help](#)