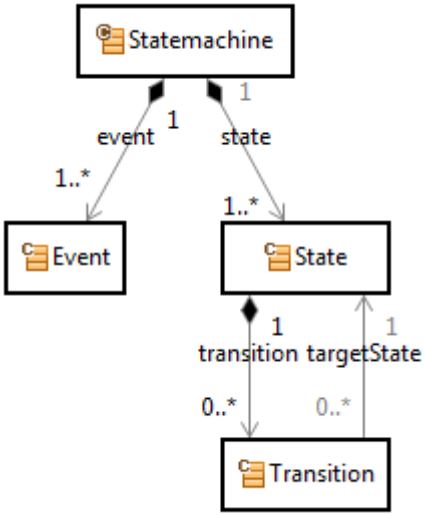
Tutorial

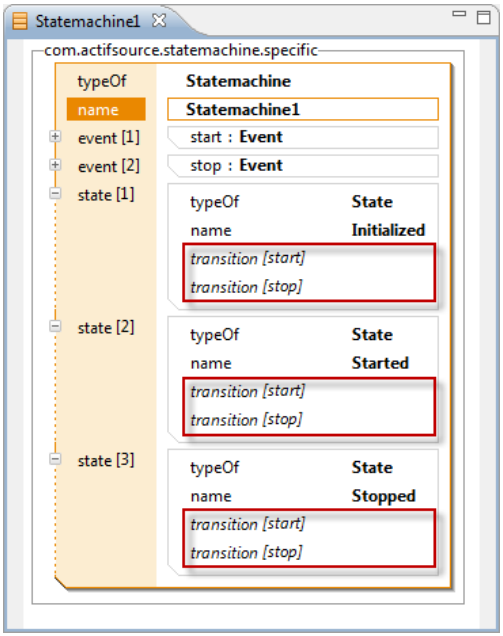
State Machine

Tutorial	Actifsource Tutorial – State Machine
Required Time	• 40 Minutes
Prerequisites	• Actifsource Tutorial – Installing Actifsource • Actifsource Tutorial – Simple Service • Actifsource Tutorial – Complex Service
Goal	• Developing an easy to use state machine model • Show possible events in every transition • Restrict transition target to state instances of the own state machine
Topics covered	• Decorating Relation Aspect • Range Restriction Aspect • Selector (forward and reverse selection)
Notation	↳ To do ⓘ Information • **Bold**: Terms from actifsource or other technologies and tools • **Bold underlined**: actifsource Resources • Underlined: User Resources • *UnderlinedItalics*: Resource Functions • `Monospaced`: User input • *Italics*: Important terms in current situation
Disclaimer	The authors do not accept any liability arising out of the application or use of any information or equipment described herein. The information contained within this document is by its very nature incomplete. Therefore the authors accept no responsibility for the precise accuracy of the documentation contained herein. It should be used rather as a guide and starting point.
Contact	**actifsource GmbH** Täfernstrasse 37 5405 Baden-Dättwil Switzerland www.actifsource.com
Trademark	**actifsource** is a registered trademark of **actifsource GmbH** in Switzerland, the EU, USA, and China. Other names appearing on the site may be trademarks of their respective owners.
Compatibility	Created with actifsource Version 5.8.5

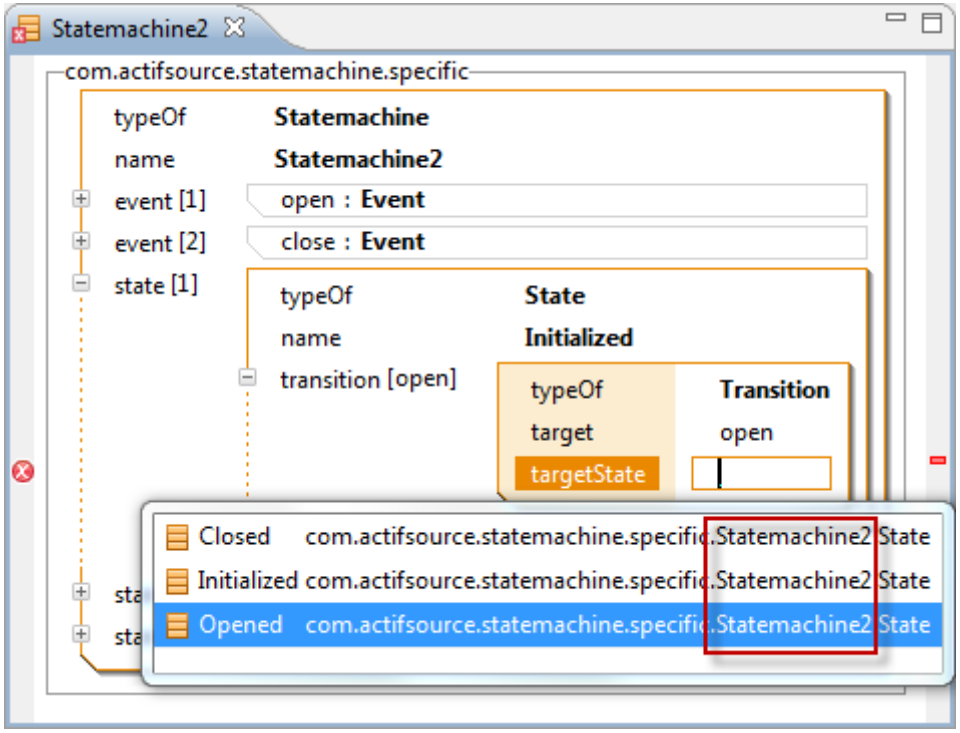
- Create a simple state machine



- Show possible events in every transition



- Restrict transition target to state instances of the own state machine



- Write a code template to generate code for a statemachine

Statemachine1Impl.hpp

```
class Statemachine1Impl
{
private:
    enum{
      Initialized,
      Started,
      Stopped
    } m_aState;

public:
  void start()
  {
    switch(m_aState)
    {
      case Initialized:
        m_aState = Started;
      break;
      case Stopped:
        m_aState = Started;
      break;
    }
  }

  void stop()
  {
    switch(m_aState)
    {
      case Started:
        m_aState = Stopped;
      break;
    }
  }

};
/* Actifsource ID=[0b6aff85-85f9-11e4-a105-d1f
```

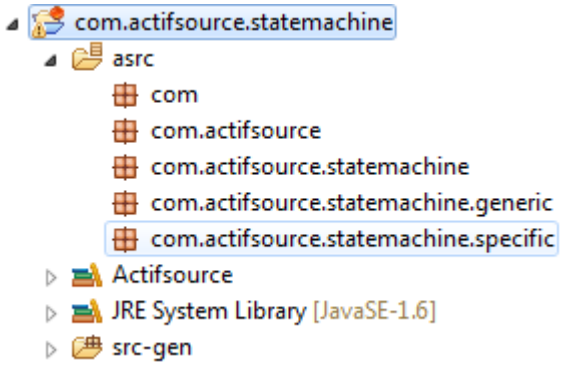
Statemachine2Impl.hpp

```
class Statemachine2Impl
{
private:
    enum{
      Initialized,
      Opened,
      Closed
    } m_aState;

public:
  void open()
  {
    switch(m_aState)
    {
      case Initialized:
        m_aState = Opened;
      break;
      case Closed:
        m_aState = Opened;
      break;
    }
  }

  void close()
  {
    switch(m_aState)
    {
      case Opened:
        m_aState = Closed;
      break;
    }
  }

};
/* Actifsource ID=[0b6aff85-85f9-11e4-a105-c
```

Part I: Preparation

6

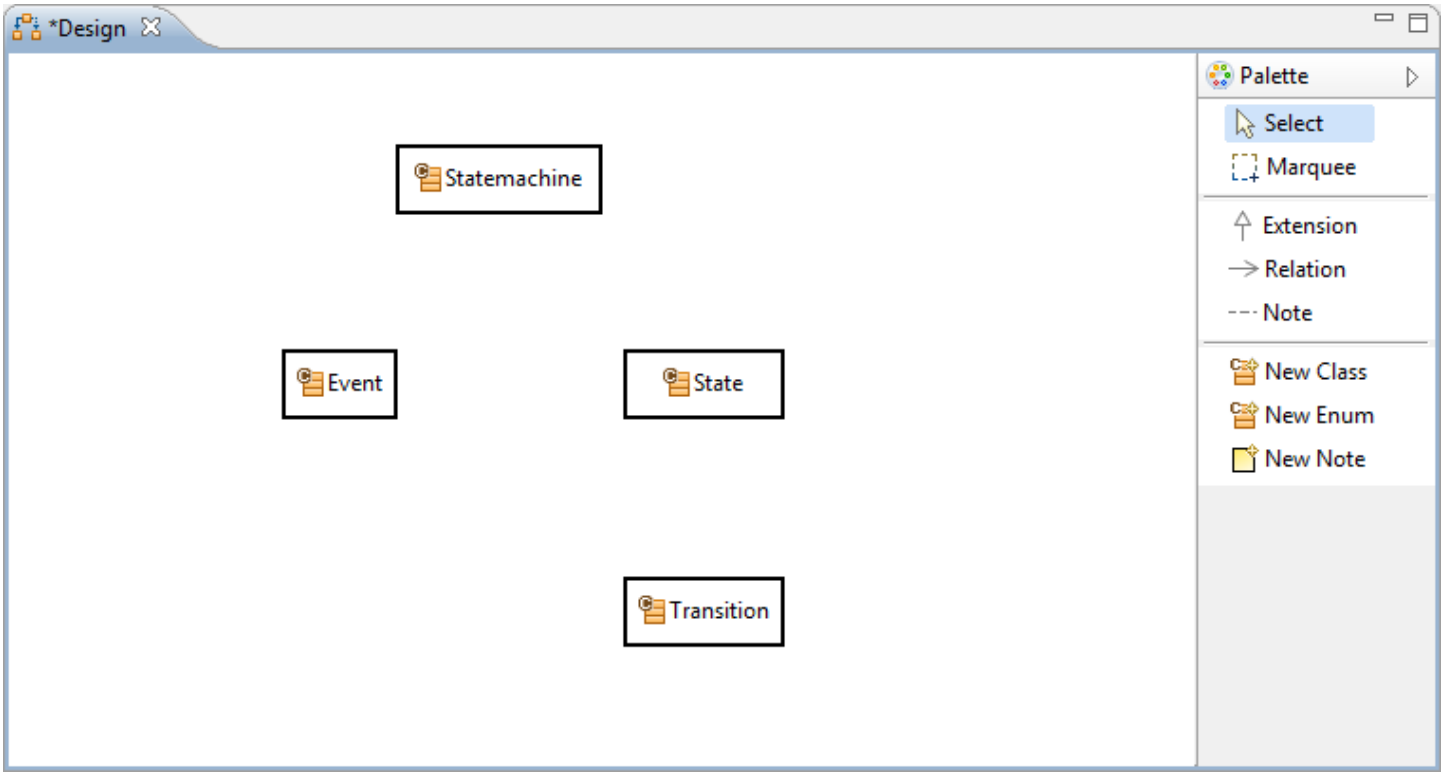
- Prepare a new **actifsource Project** named com.actifsource.statemachine as seen in the *Actifsource Tutorial – Simple Service*
- Use the following package structure



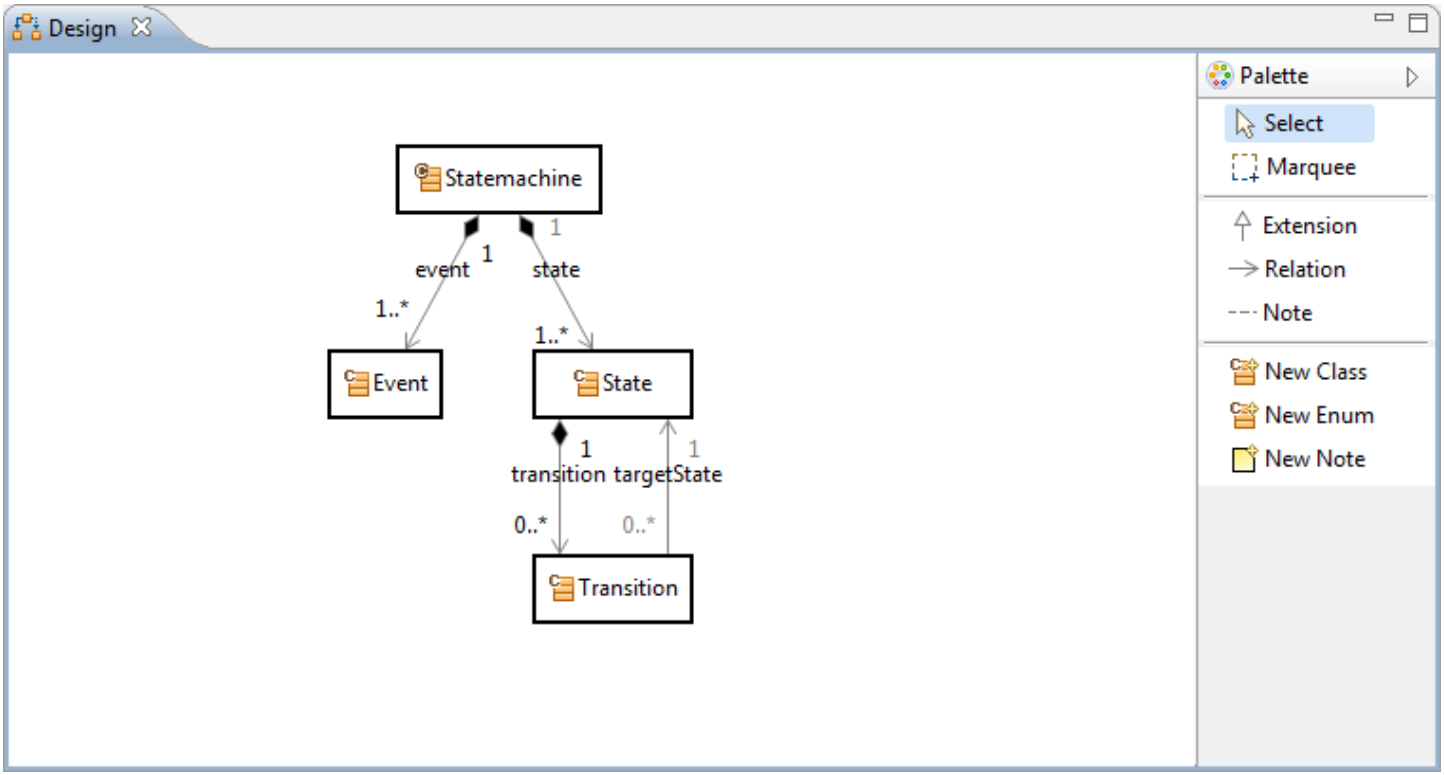
Part II: 7
Create a State Machine

- ⓘ Create a simple state machine
- ⓘ Instantiate the state machine and see its deficits

Create a Generic State Machine Model 8

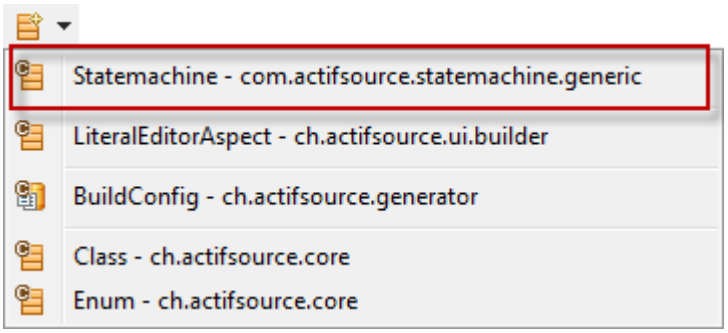


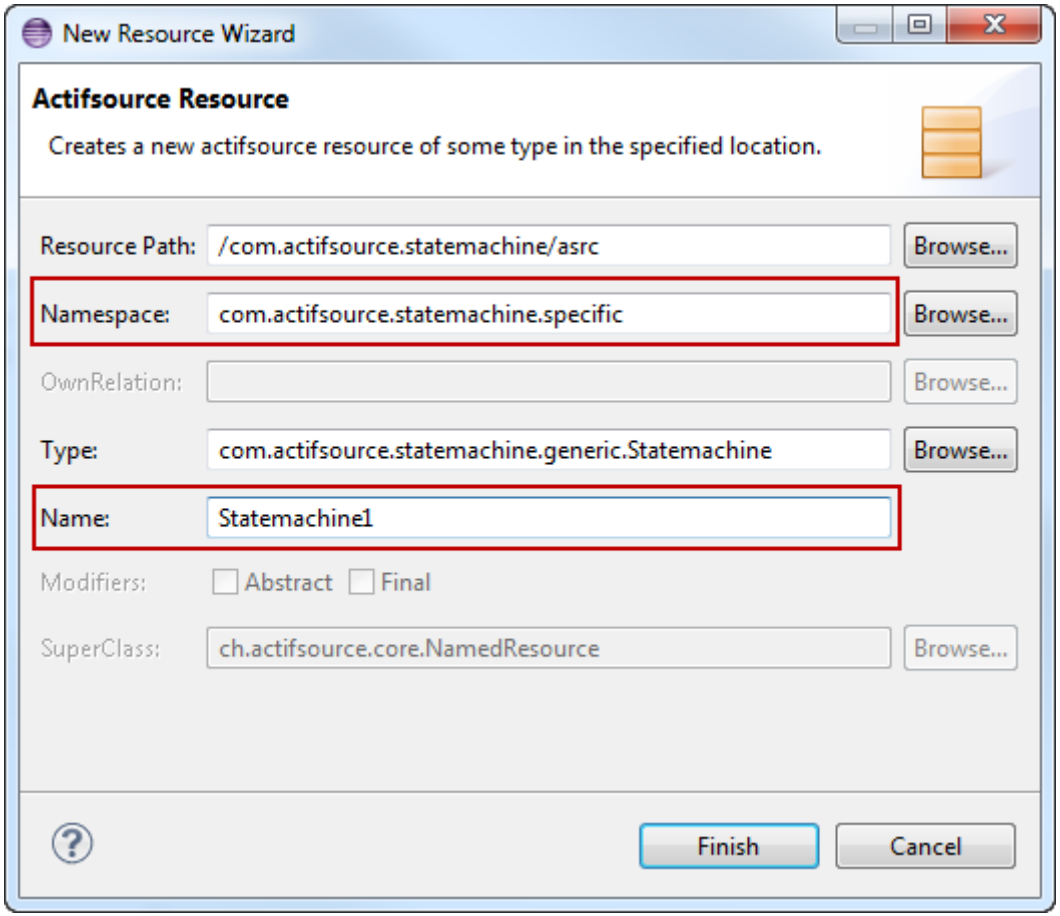
- Create a **Generic Domain Model** named *Design* in the **Package** *generic* using the **DiagramEditor**
- The Design shall contain the following **Domain Classes**
  - Statemachine, Event, State, Transition



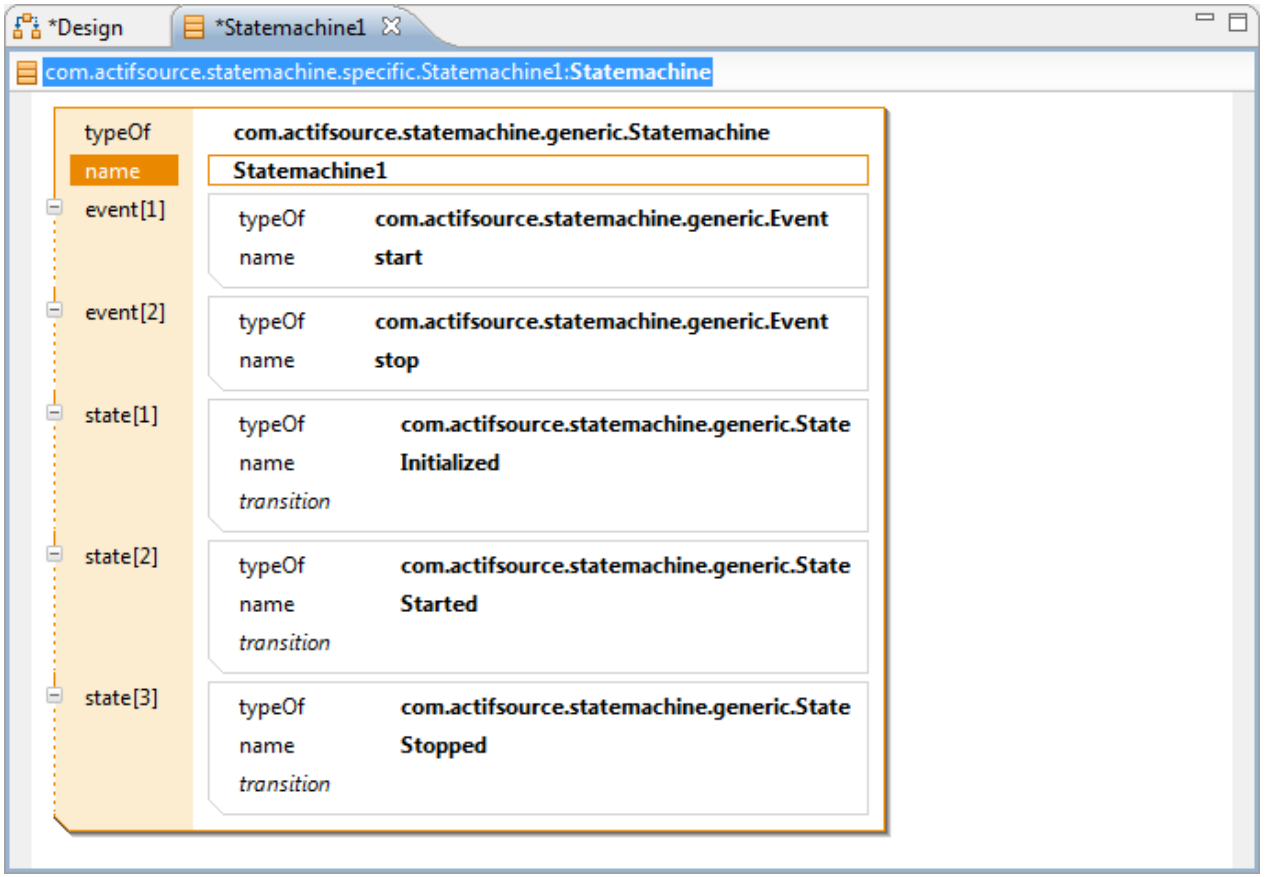
- Insert a **OwnRelation** between
  - Statemachine and Event
  - Statemachine and State
- Insert a **DecoratingRelation** between
  - State and Transition
- Insert a **UseRelation** between
  - Transition and State
- Adjust the **Cardinalities** as shown above
- ⓘ Warning: The layout for the relations transition and targetState might differ in your editor

Create a Specific State Machine





↳ Create a Statemachine named Statemachine1 in the **Package** *specific*

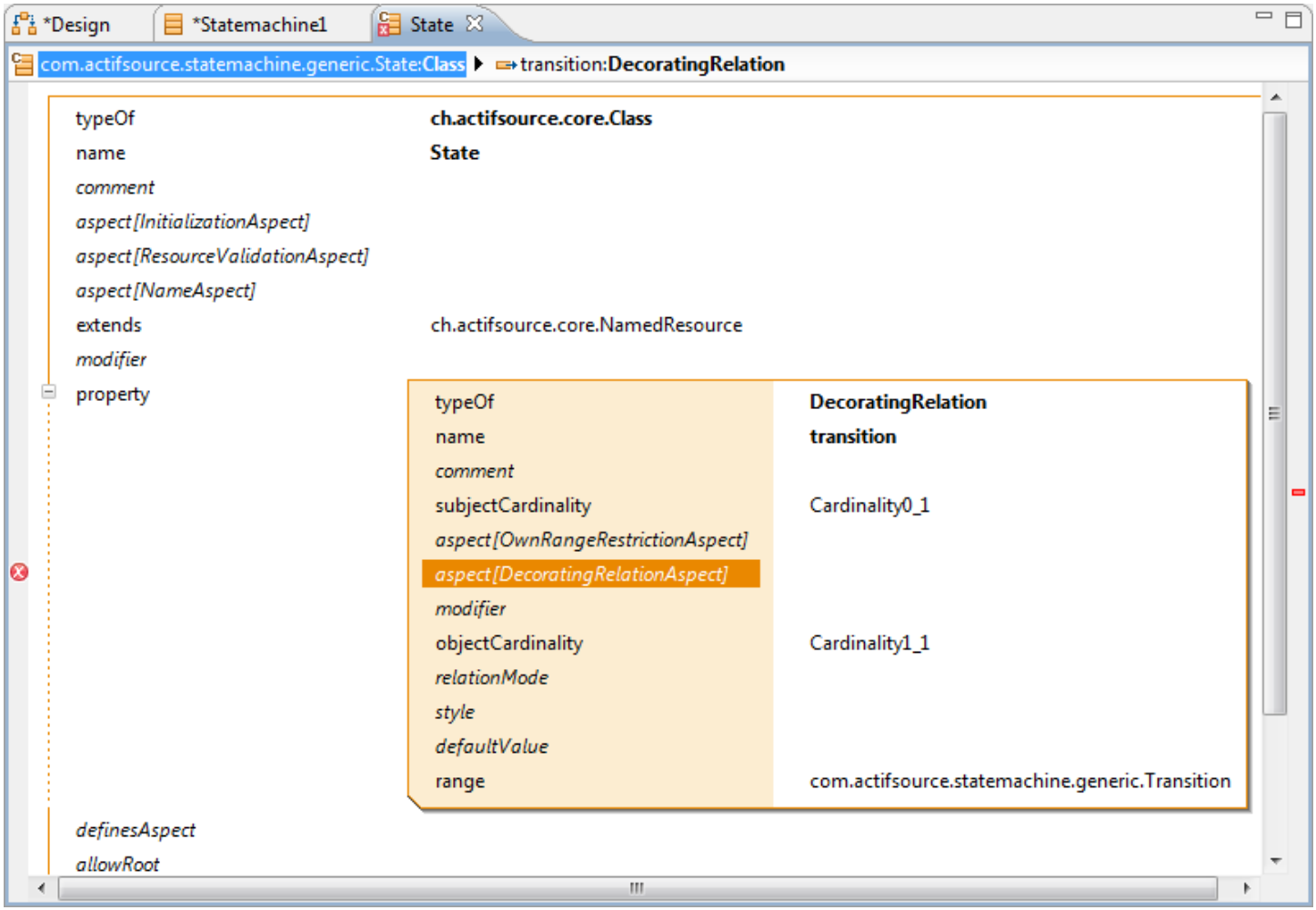


- Add the Events start and stop
- Add the States Initialized, Started and Stopped as shown above

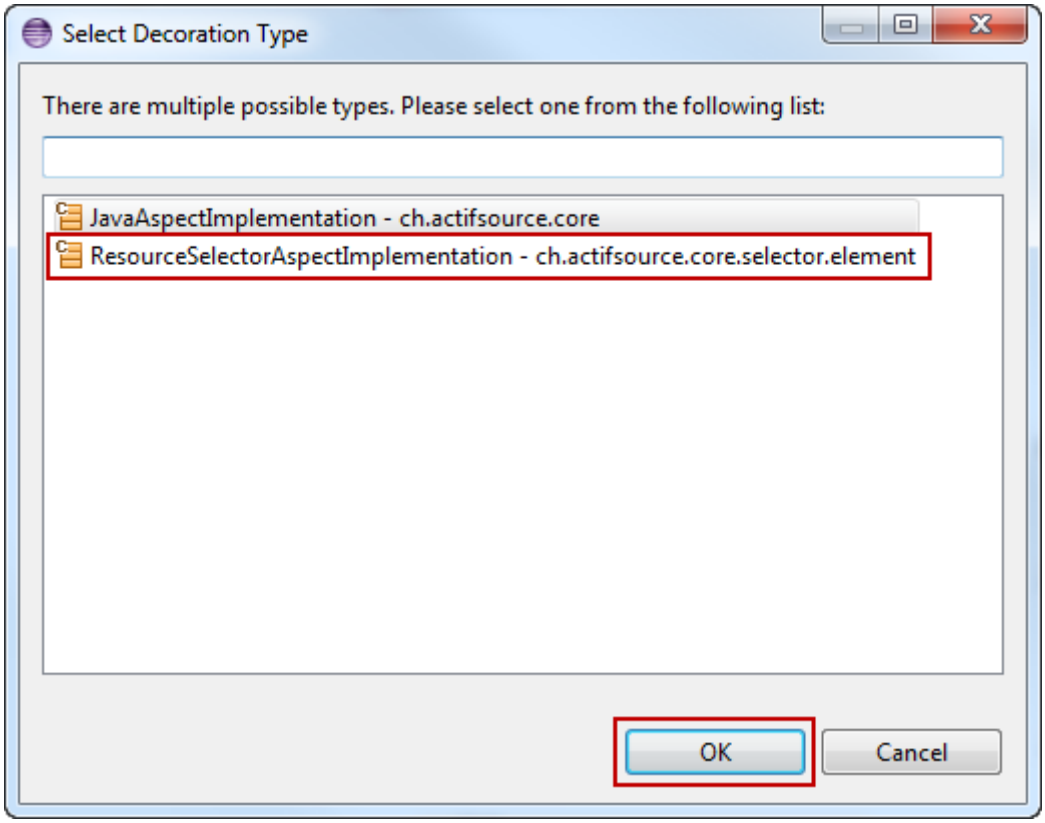
Part III: 12
Decorating Relation Aspect

ⓘ Learn how to decorate a relation with a list of resources in order to prevent the mixing of instances from different Statemachines

Add a Decorating Relation Aspect



- In State open the **DecoratingRelation** transition
- Press Enter on *aspect[DecoratingRelationAspect]*

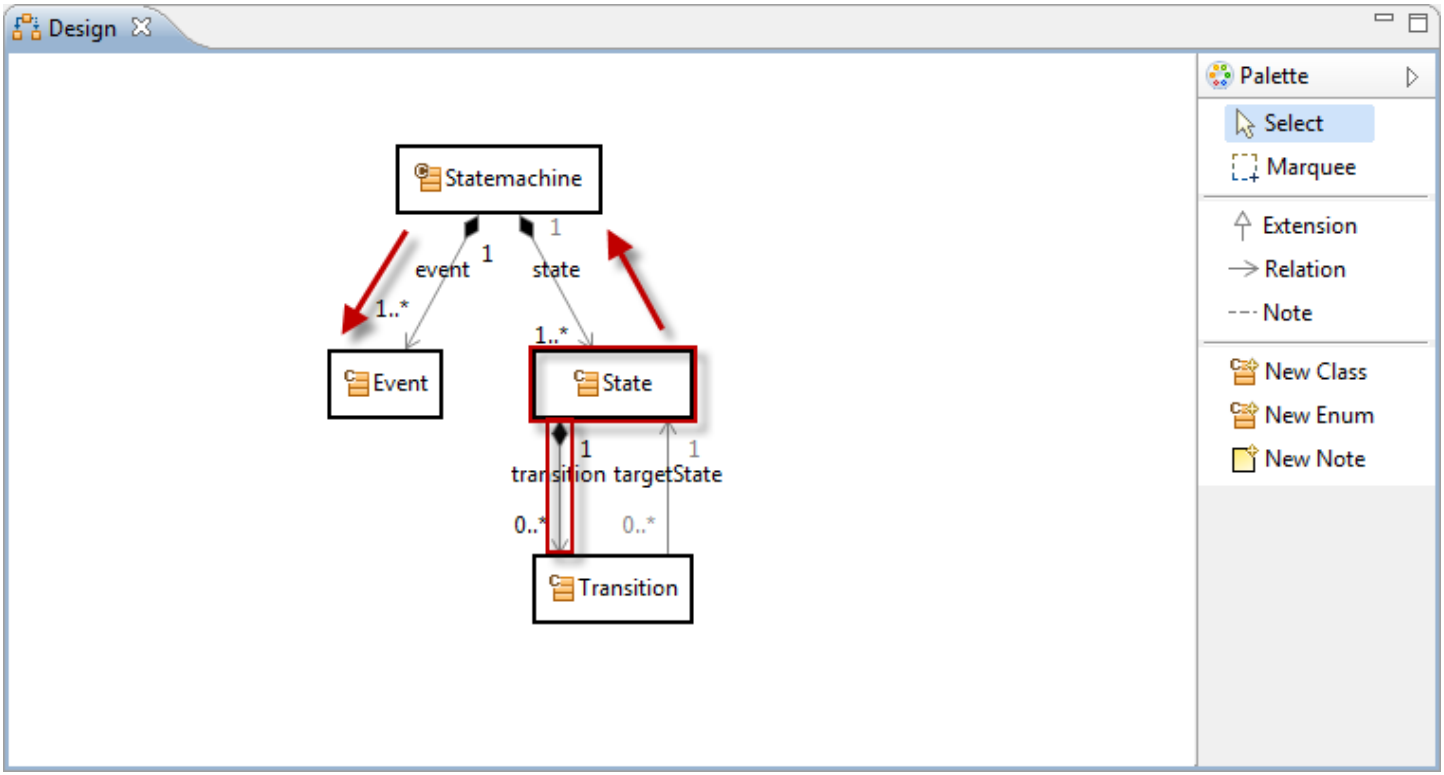


ⓘ Note that you can choose between a *JavaAspectImplementation* and a *SelectorAspectImplementation*
- Selecting the *JavaAspectImplementation* allows you to write Java Code for complex operations
- Selecting the *ResourceSelectorAspectImplementation* allows you to use the easy Selector syntax

↳ Select *ResourceSelectorAspectImplementation*

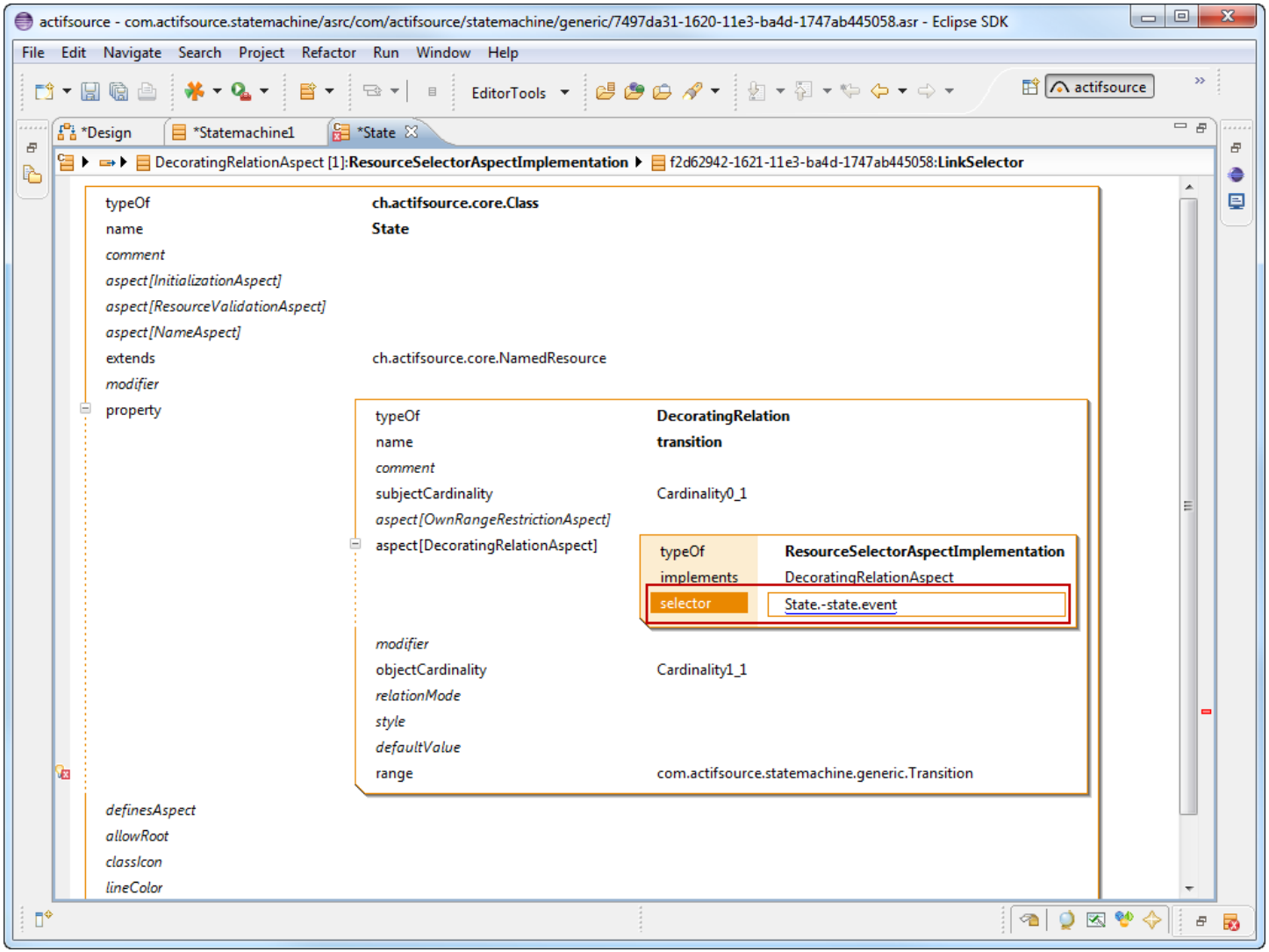
↳ Click *OK*

Add a Decorating Relation Aspect 15



- Let's look at a possible Transition for every Event
- The **DecoratingRelation** transition is found in State
- We have to navigate from State to Event
  - Navigate backwards from State via state to Statemachine
  - Navigate forward from Statemachine via event to Event

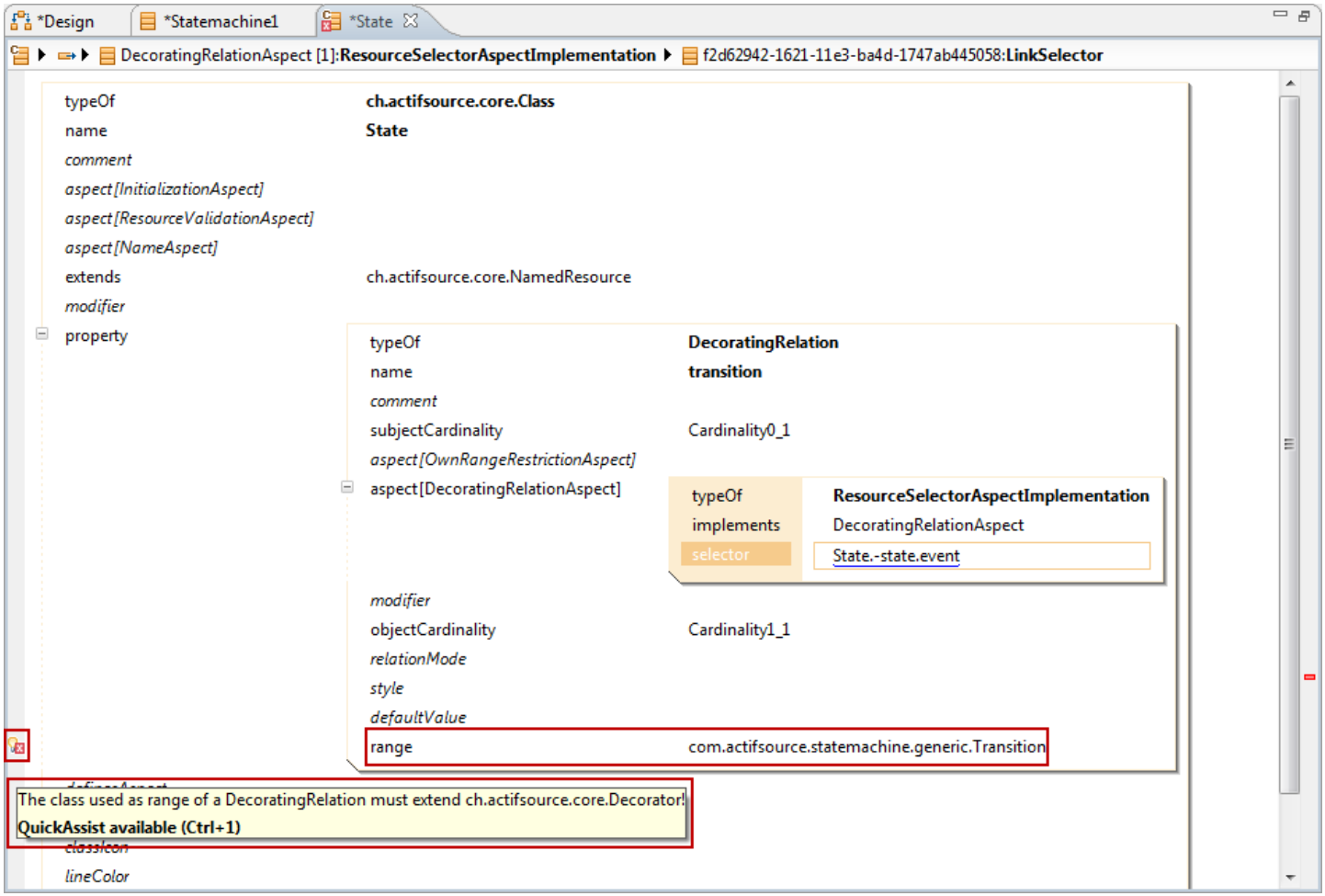
Add a Decorating Relation Aspect 16



↳ Enter the Selector `State.-state.event` using Content Assist (Ctrl+Space)

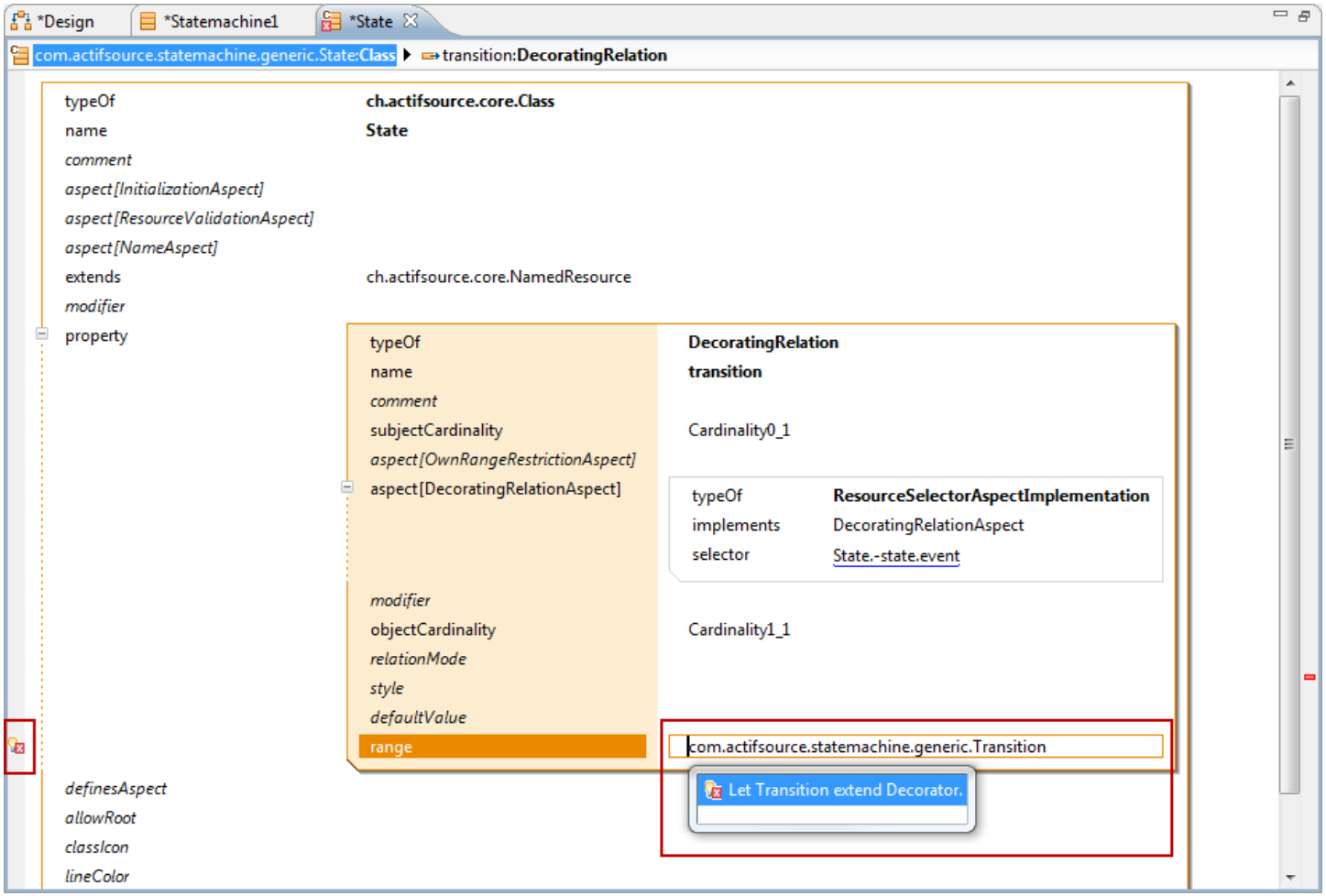
ⓘ Note that State.–state navigates backwards from State to Statemachine

Add a Decorating Relation Aspect 17

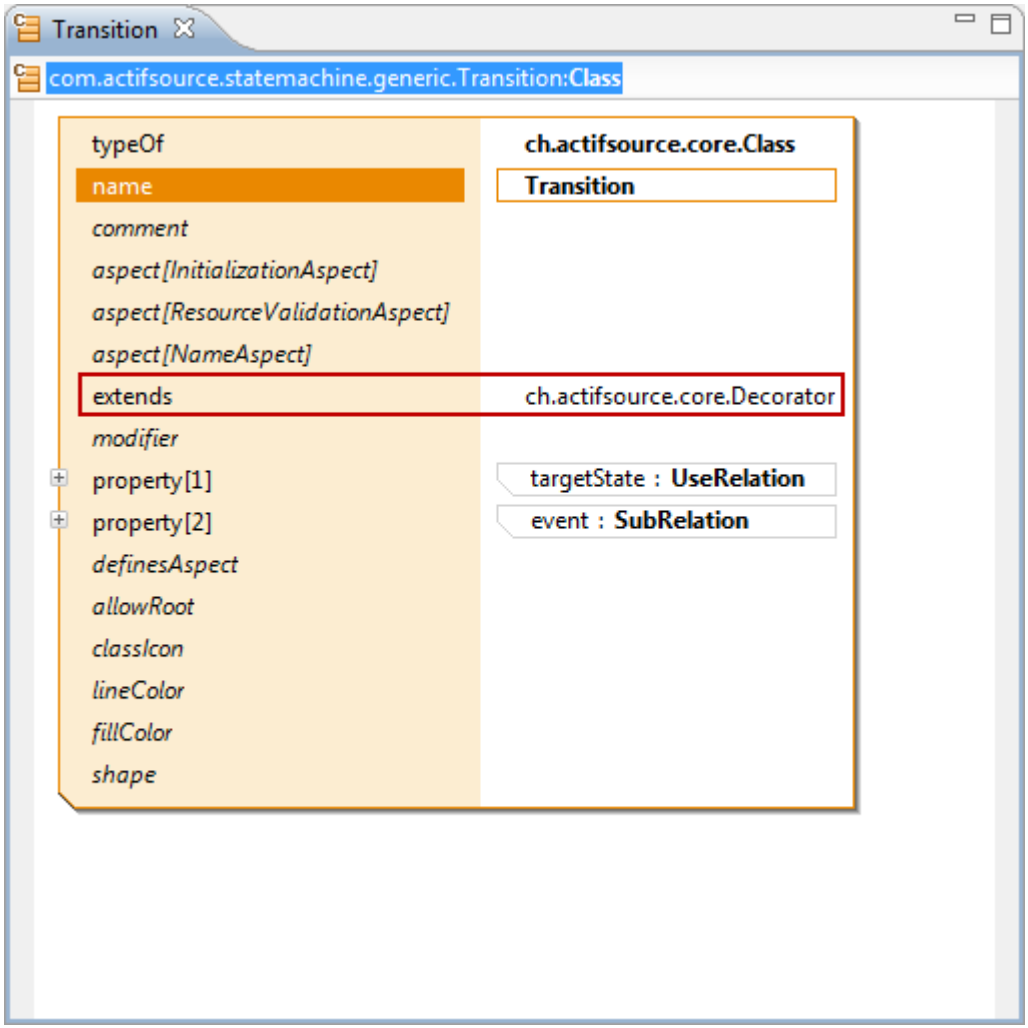


- ⓘ Implementing a *DecoratingRelationAspect* asks for a subclass of Decorator
- ⓘ Decorator has a **useRelation target** which is used to store the specific decorating Resource
  - Shown as: decoratingRelation[target]
- ↳ Open **Quick Assist** by clicking the light bulb or press Ctrl+1

Add a Decorating Relation Aspect 18

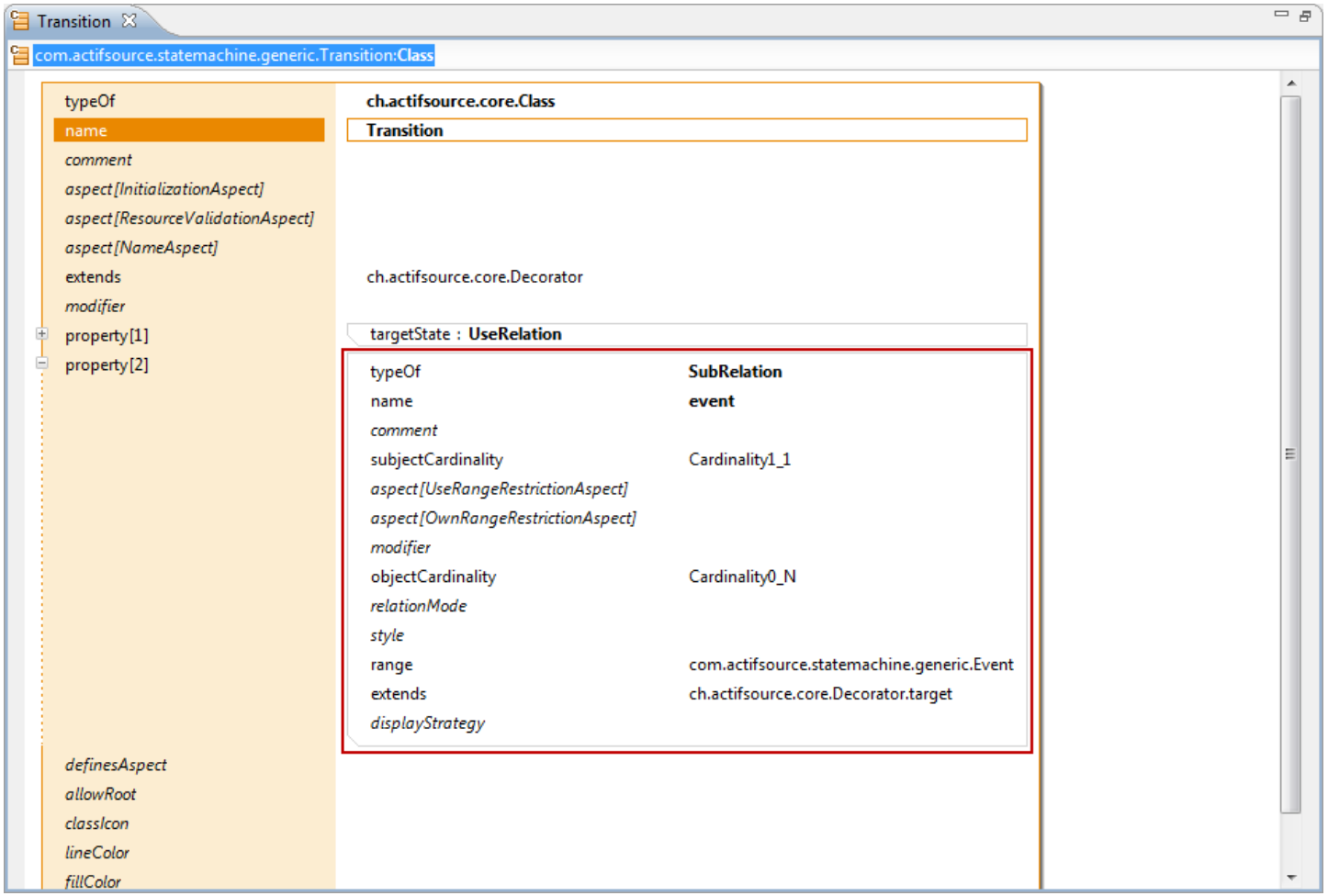


↳ Use **Quick Assist** to let Transition extend **Decorator**



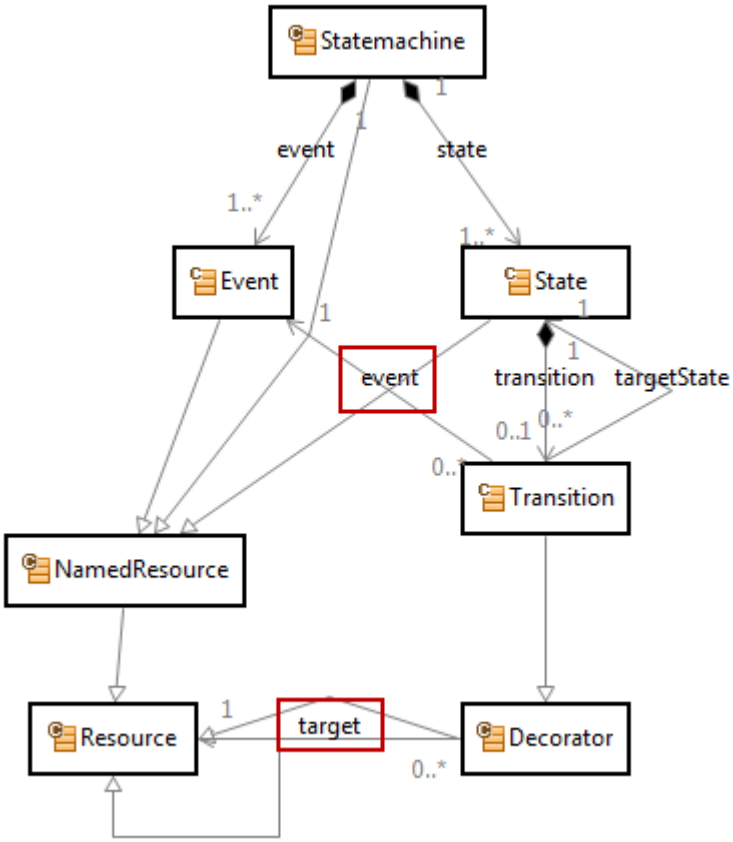
- Open Transition
- By default a **Class** extends **NamedResource**
- The **Quick Assist Action** changed the **extends** statement from **NamedResource** to **Decorator**

Add a Decorating Relation Aspect 20

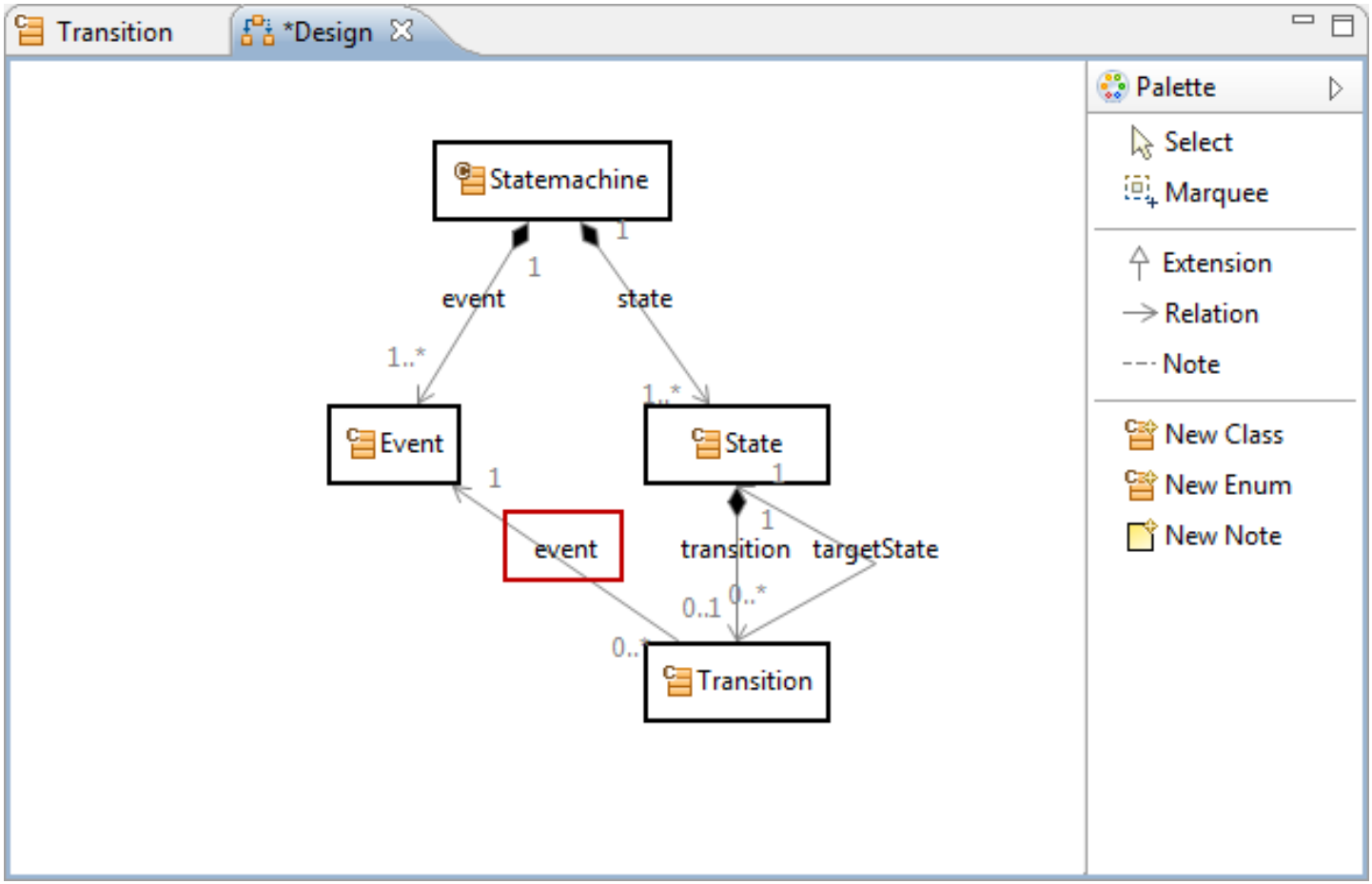


ⓘ Quick Assist has done the following

- Added **extend** **Decorator**
- Added **SubRelation** event

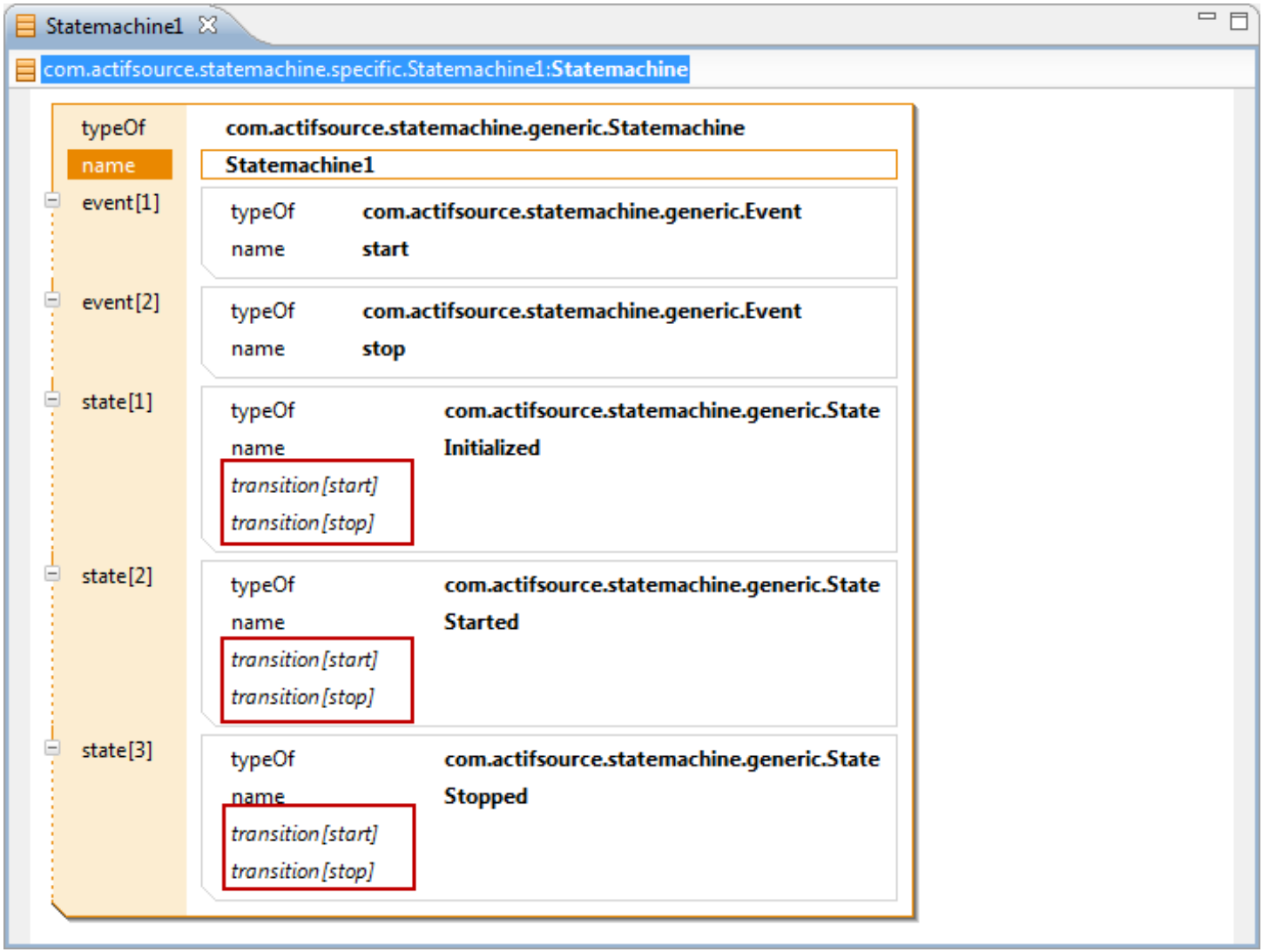


- The **range** of Decorator.target is **Resource** and therefore untyped in the context of your domain
- The new **SubRelation** target extends **Decorator.target** but with Event as its **range**
- When writing template code, you are able to access Transition.event typed as Event



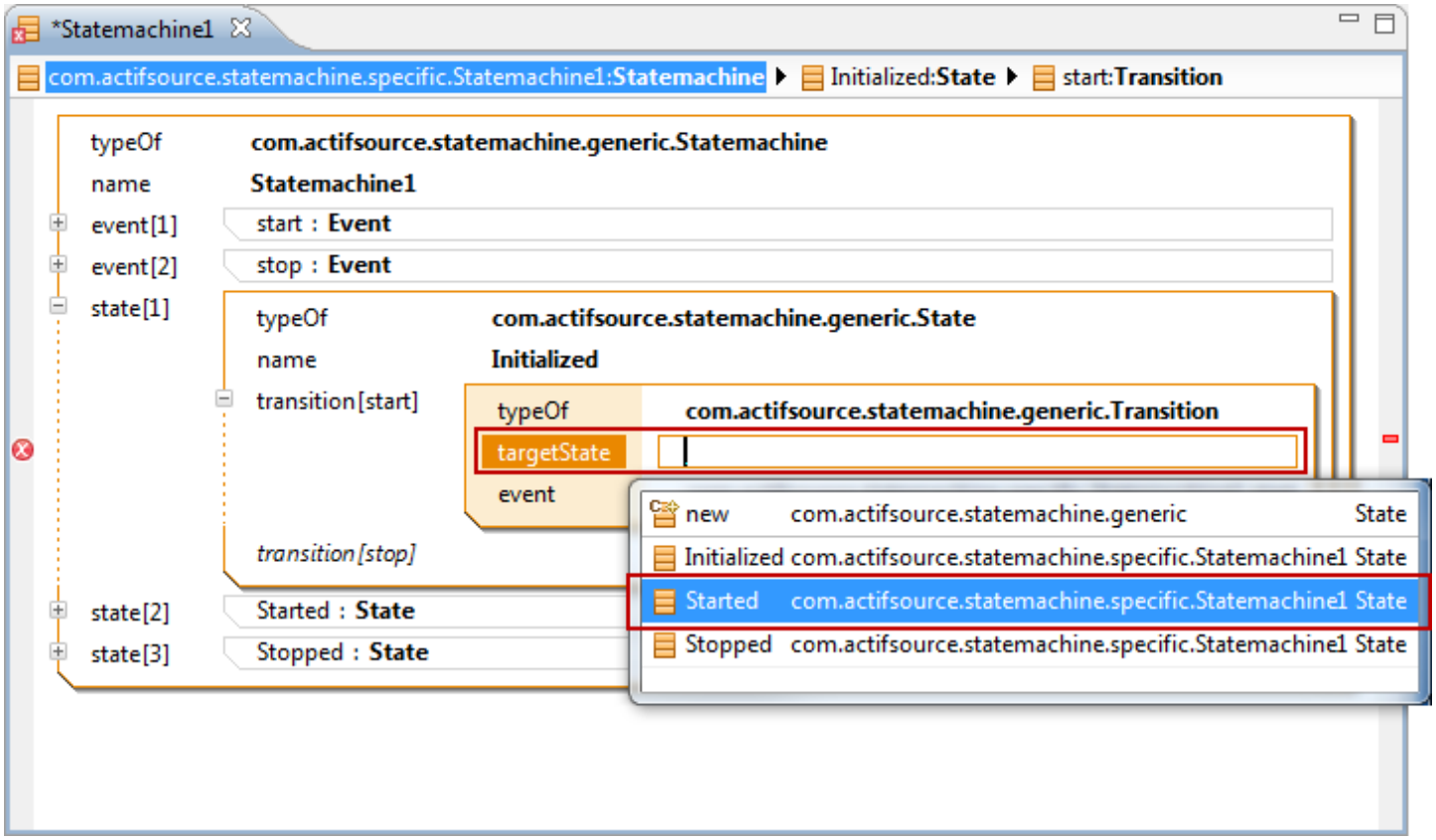
ⓘ Note that the **SubRelation** target has been added in the *Design* **Diagram** automatically

ⓘ

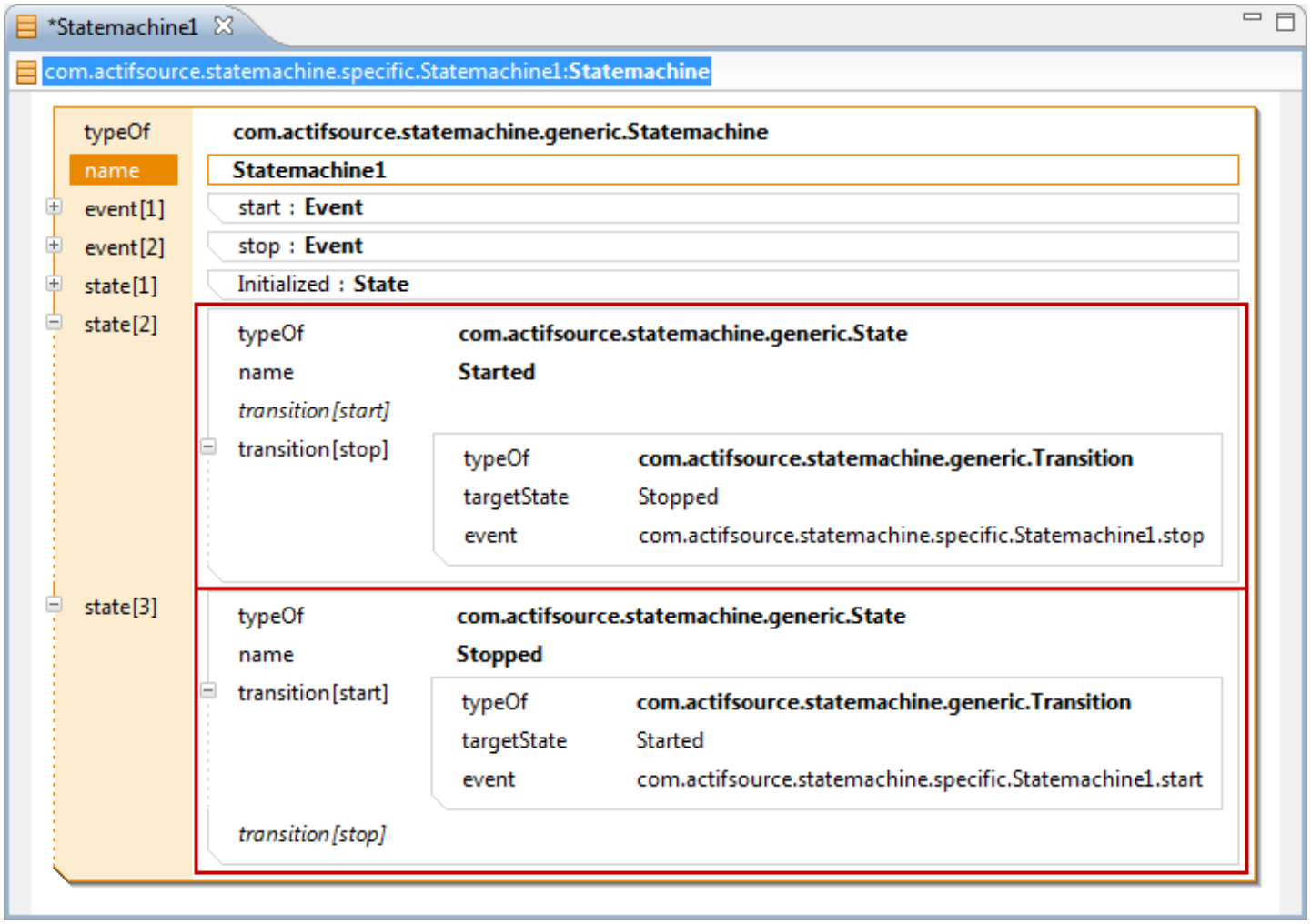


- Open the specific Statemachine Statemachine1
- ⓘ Note there is a **decoratingRelation** transition for every Event
- ⓘ Add new Events and observe the **decoratingRelation** transition

Use the Decorating Relation Aspect



- In the State Initialized create a new Transition for transition[start]
- Select Started as targetState
- Note that the relation target has been completed automatically with the specific decorating Event start

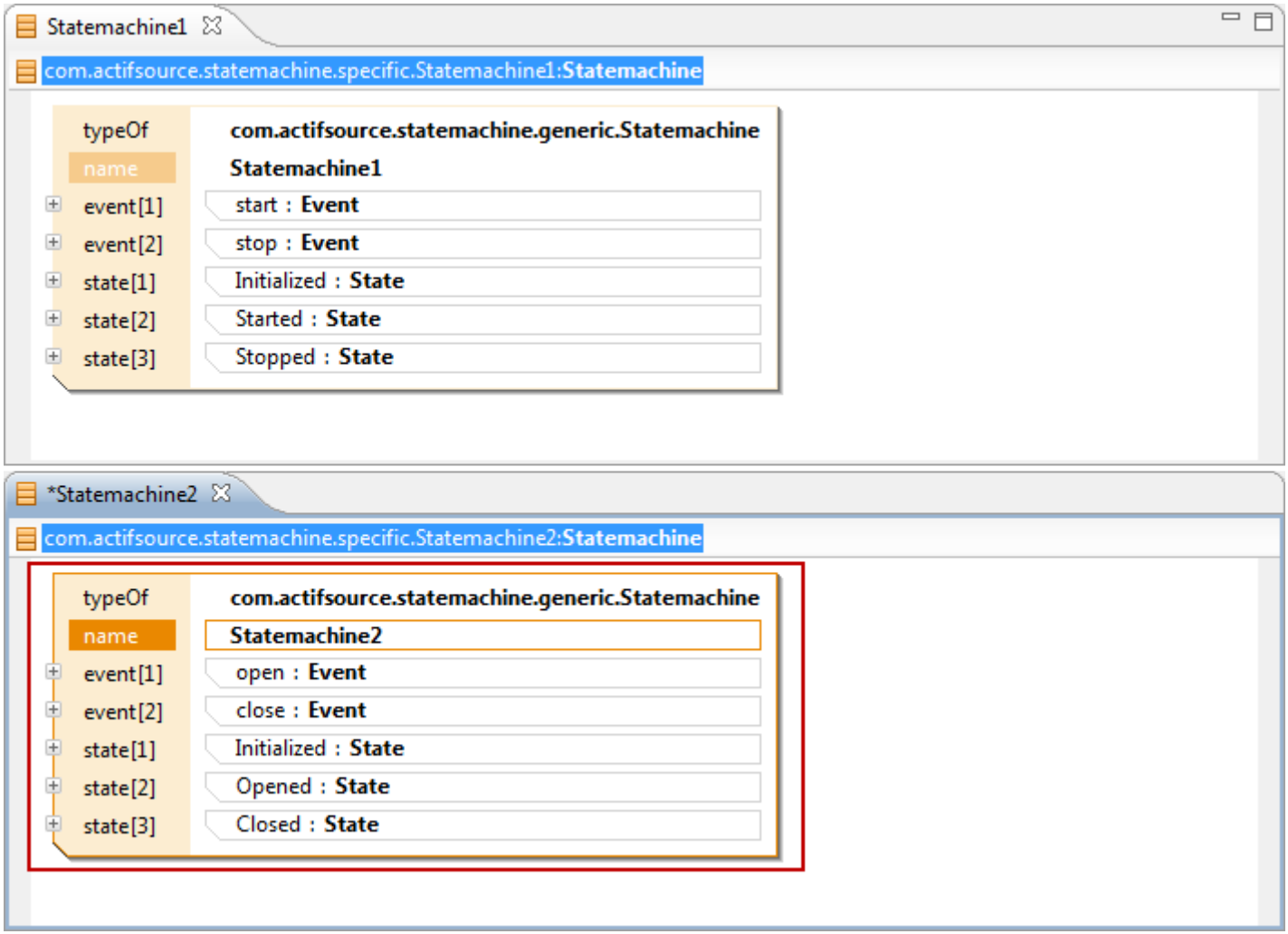


↳ Configure the State instances Started and Stopped as shown above

Part IV: 26
Range Restriction Aspect

- ⓘ Content Assist (Ctrl+Sapce) in actifsource shows all instances of a desired type; It is often useful to restrict this selection
- ⓘ Learn how to apply range restrictions to filter instances for a given type

Without Range Restriction



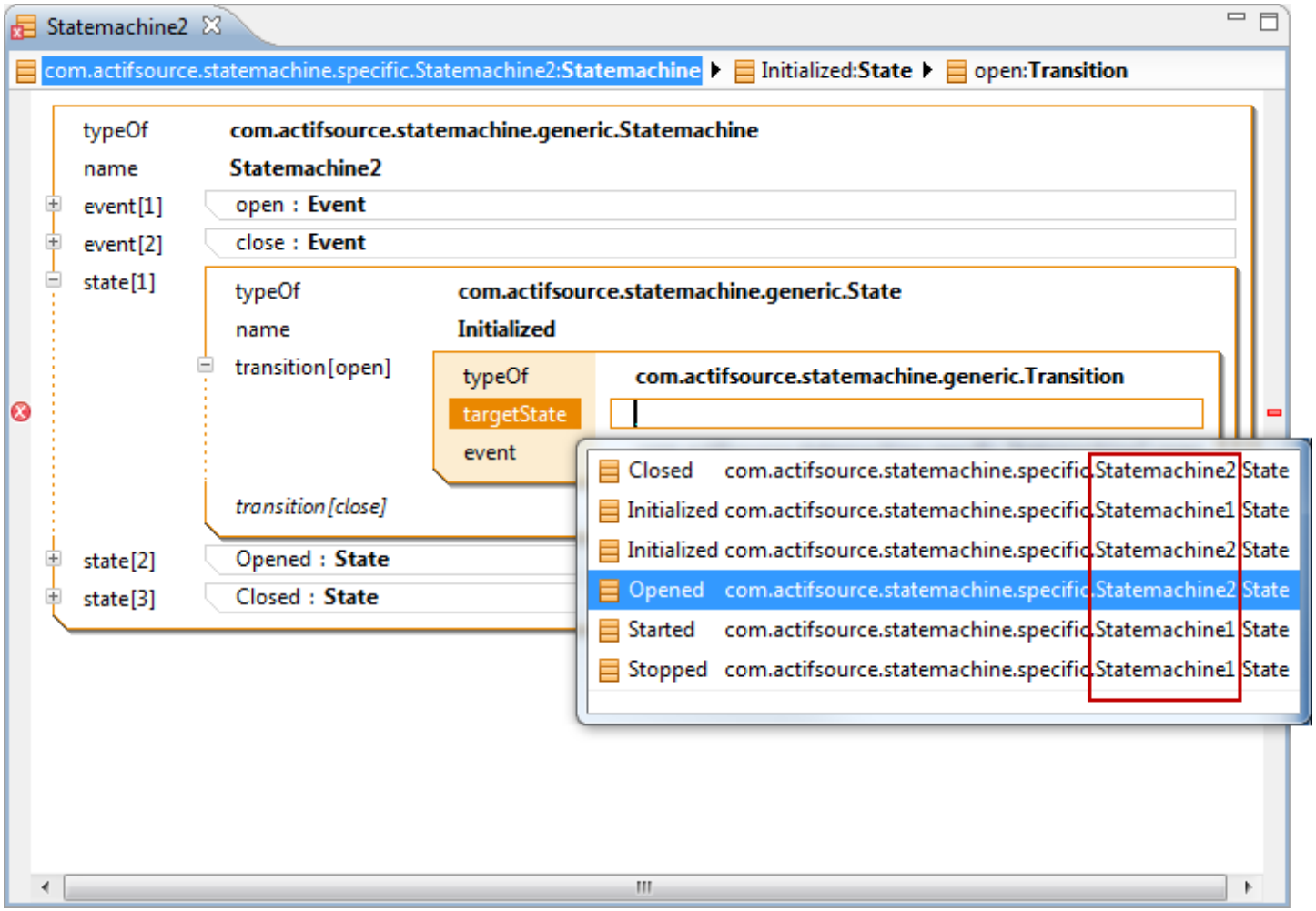
ⓘ Let's discover the needs for a range restriction aspect

↳ Create a Statemachine named Statemachine2 in the **Package** *specific*

↳ Add the Event instances open and close

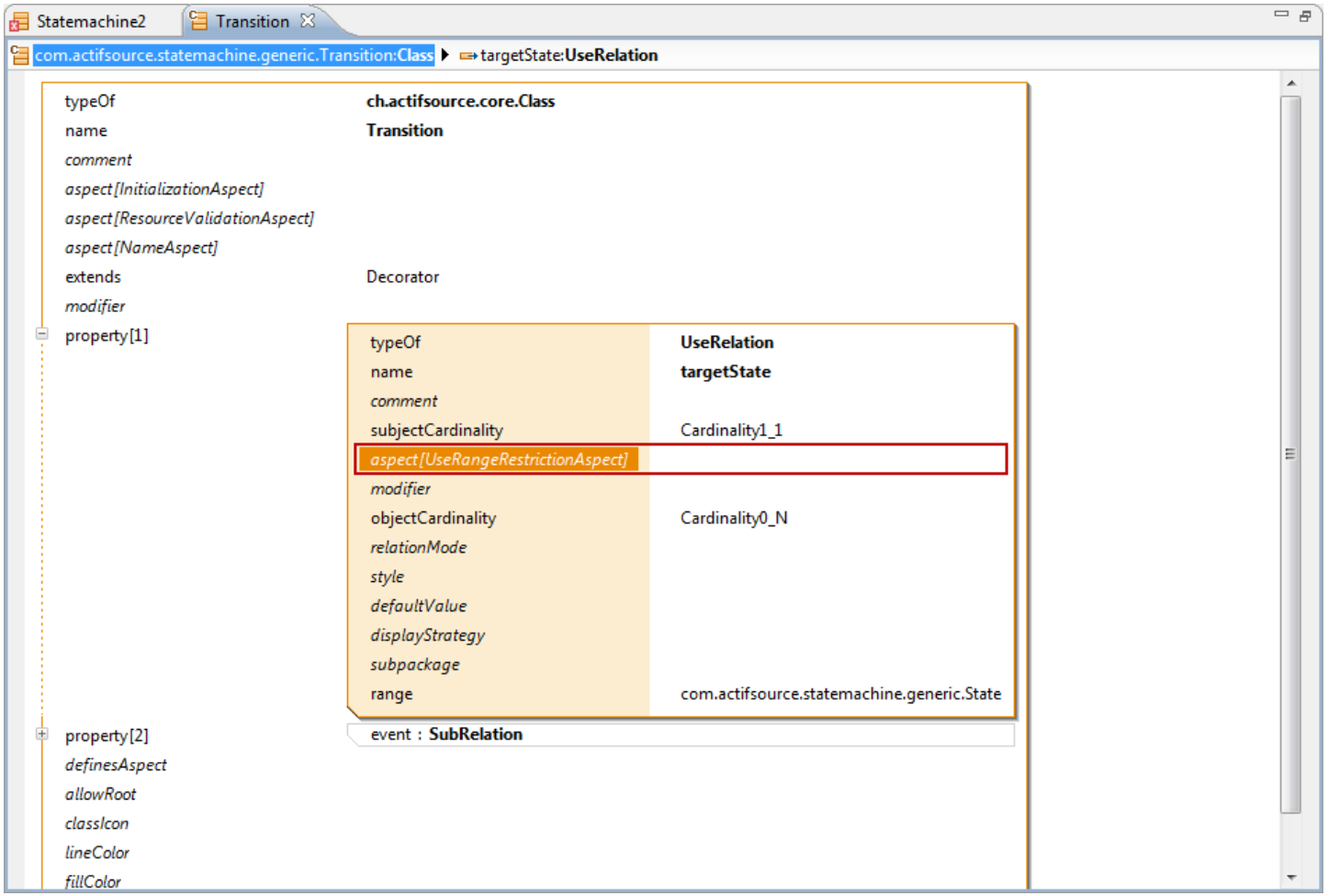
↳ Add the States instances Initialize, Opened and Closed

Without Range Restriction

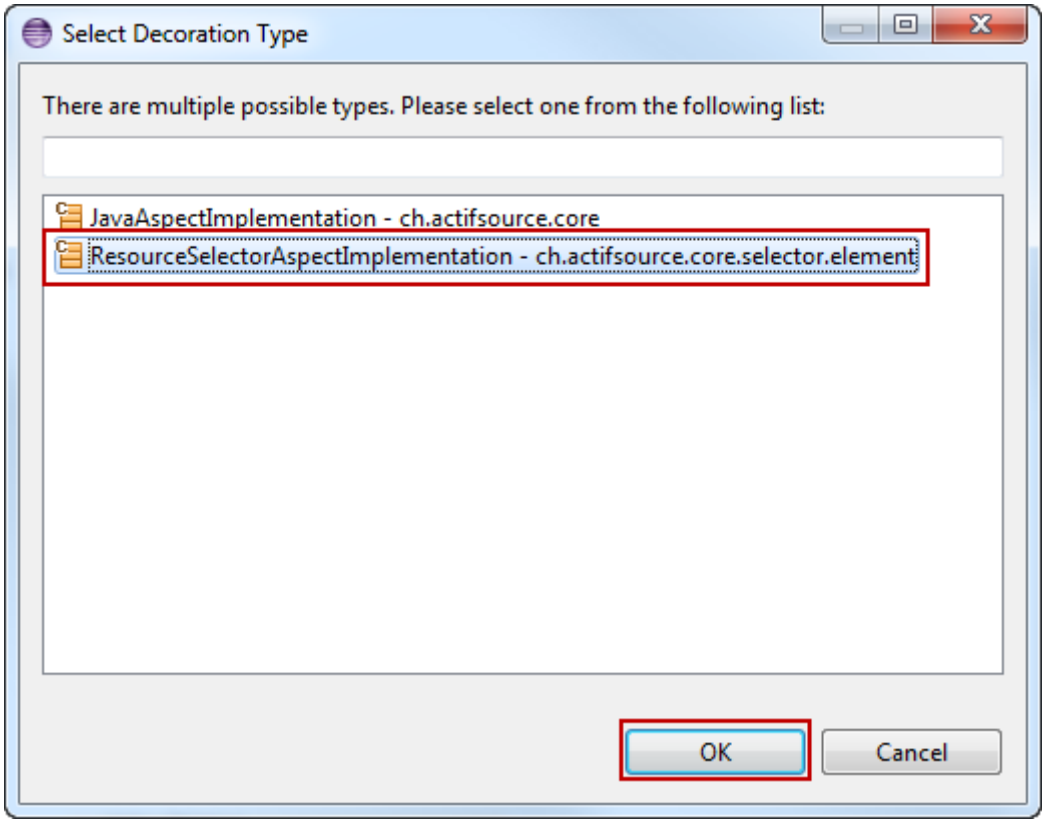


- Create any new Transition
- Use **Content Assist** (Ctrl+Space) to add a targetState of type State
- ⓘ Note that all instances of State are listened instead of just the ones from Statemachine2

Add a Range Restriction Aspect



- In Transition open the **useRelation** targetState
- Press Enter on *aspect[RangeRestrictionAspect]*

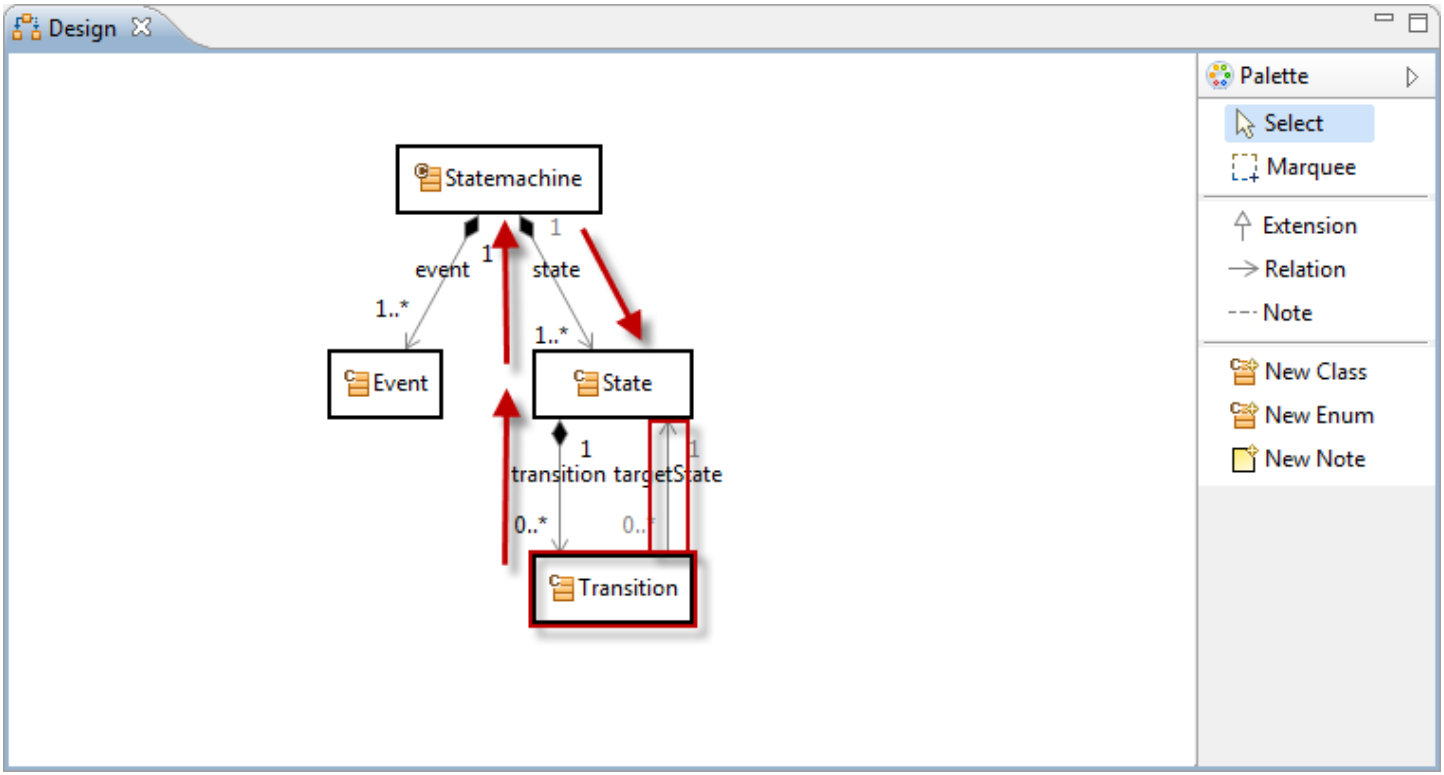


ⓘ Note that you can choose between a *JavaAspectImplementation* and a *SelectorAspectImplementation*
- Selecting the *JavaAspectImplementation* allows you to write Java Code for complex operations
- Selecting the *ResourceSelectorAspectImplementation* allows you to use the easy Selector syntax

↳ Select *ResourceSelectorAspectImplementation*

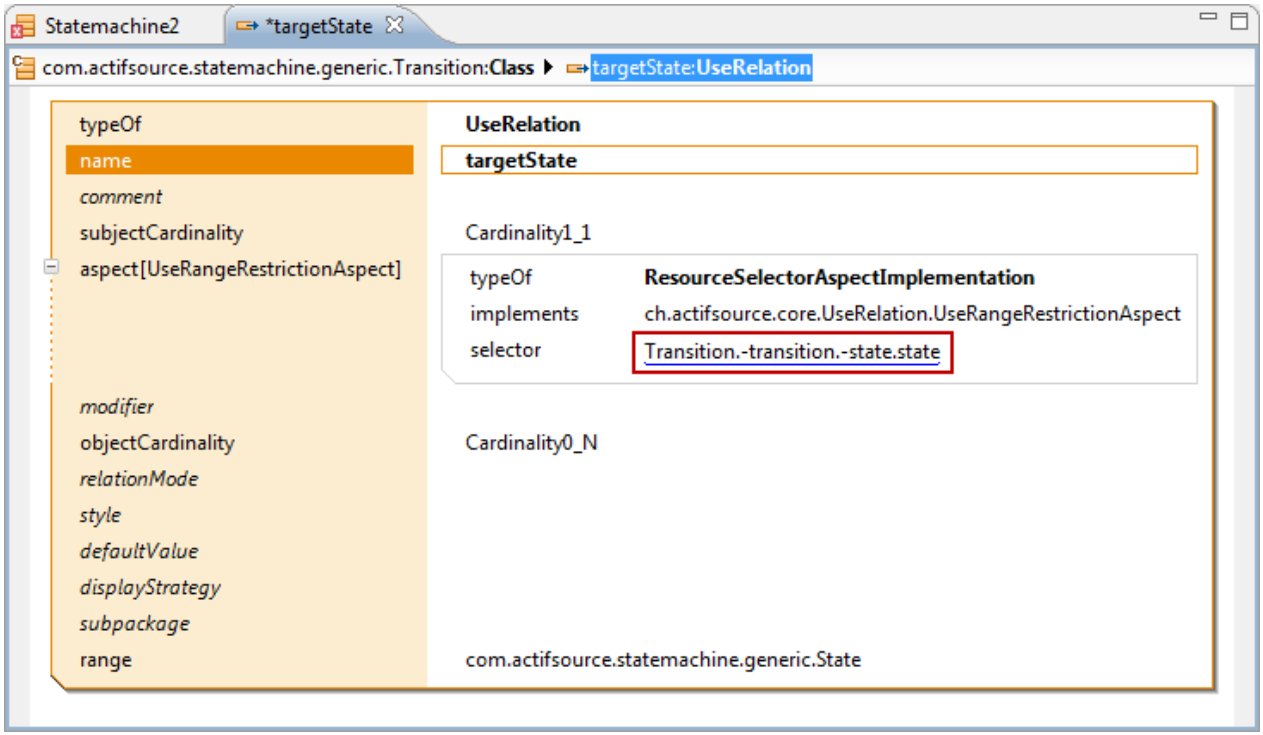
↳ Click *OK*

Add a Range Restriction Aspect

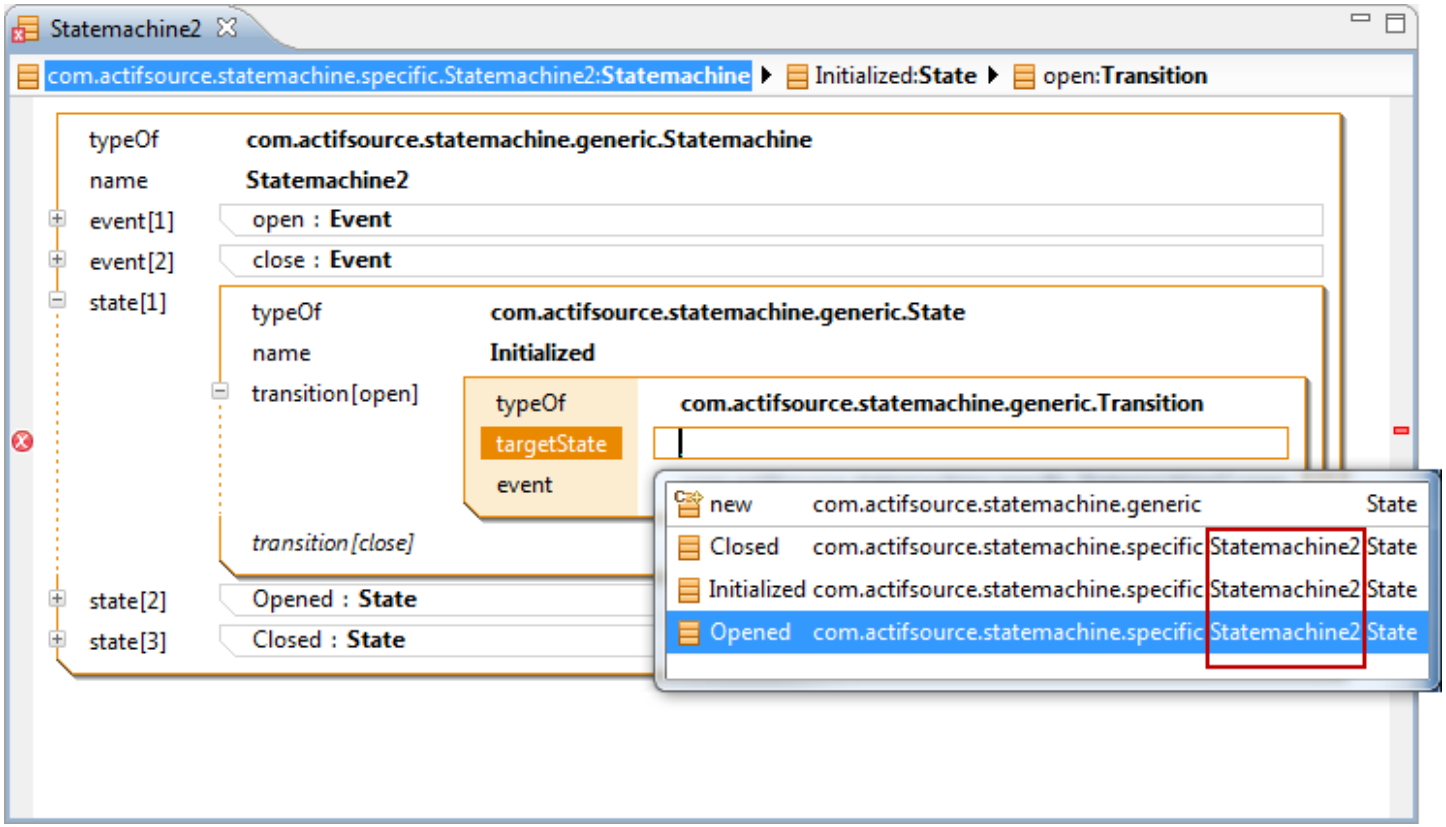


- Let's restrict the **range** of targetState to instances of States owned by the own Statemachine
- The **useRelation** targetState is found in Transition
- We have to navigate from Transition to all States of the Statemachine
  - Navigate backwards from Transition via transition to State
  - Navigate backwards from State via state to Statemachine
  - Navigate forward from Statemachine via state to State

Add a Range Restriction Aspect

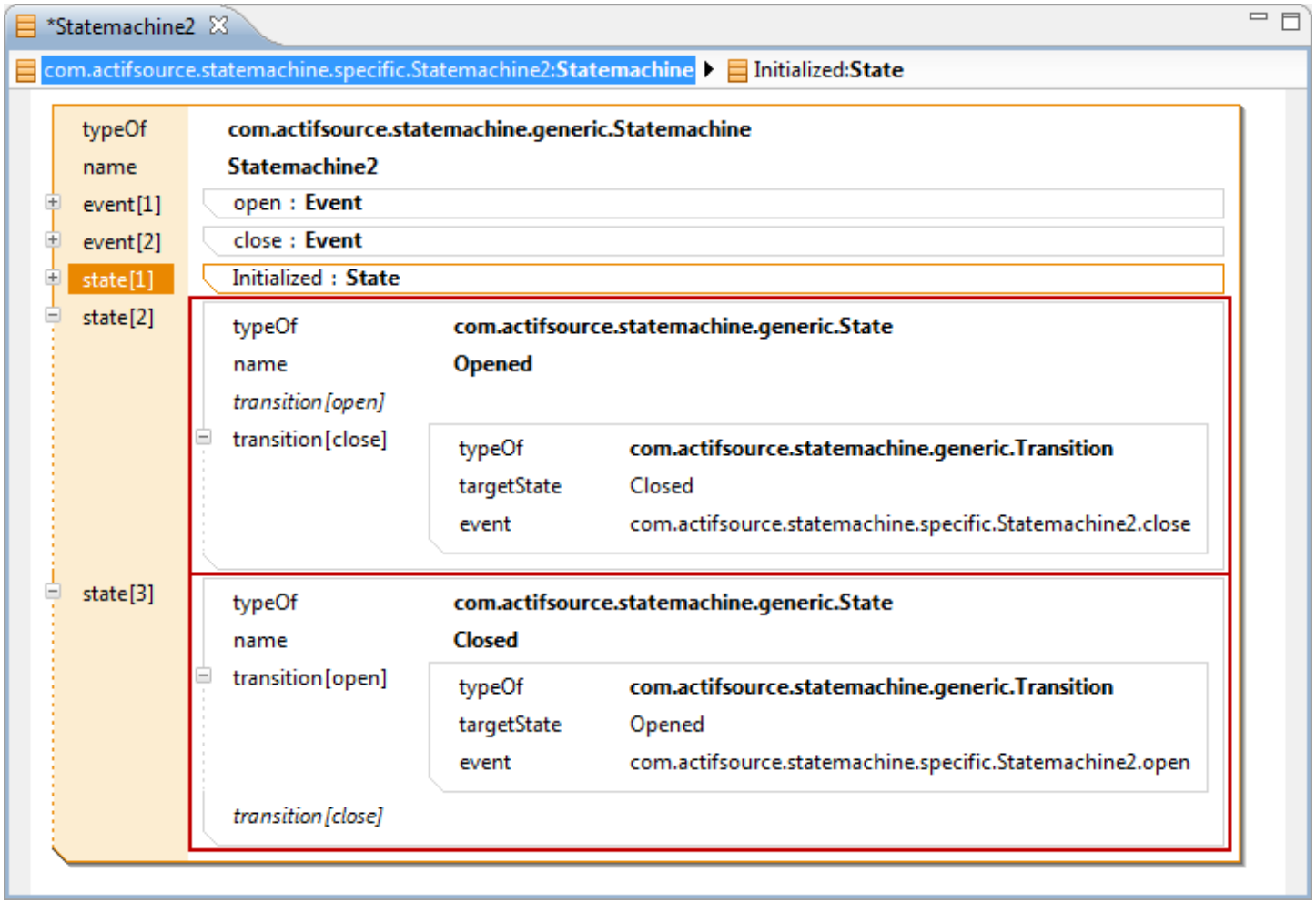


↳ Enter the Selector `Transition.-transition.-state.state` using Content Assist (Ctrl+Space)



↳ Use **Content Assist** (Ctrl+Space) again to add the targetState Opened of type State

ⓘ Note that only instances of State from Statemachine2 are listed

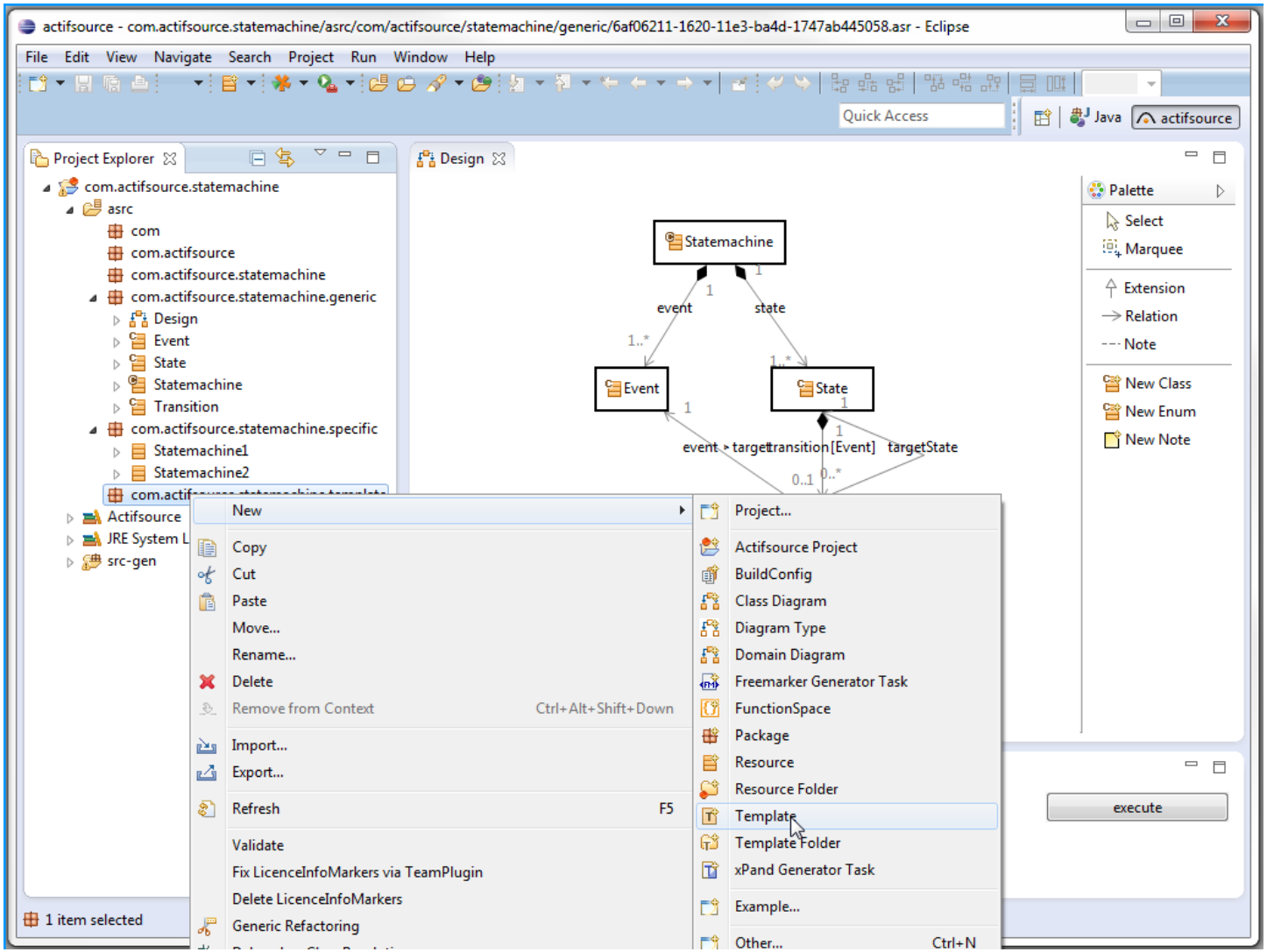


- Get familiar with **Decorating Relations** and **Range Restrictions**
- Write an **actifsource Code Template** to generate a state machine

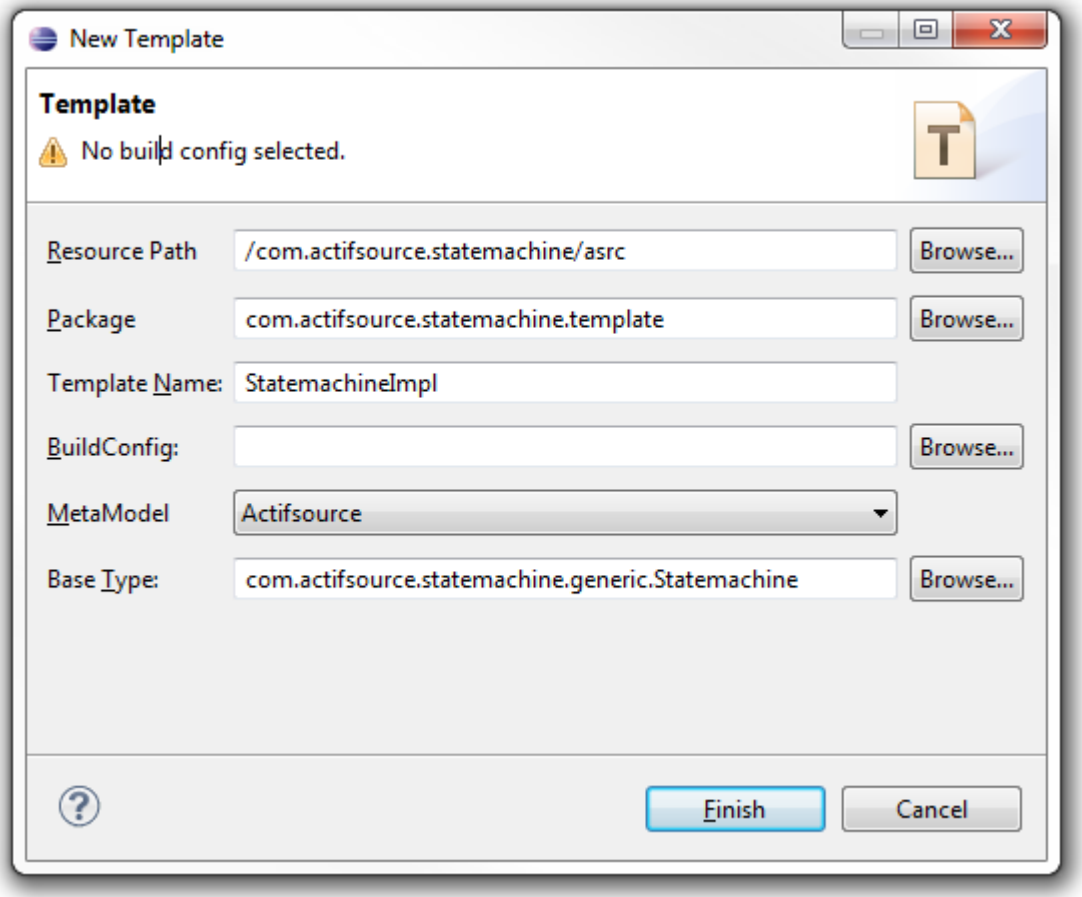
Part V: 35
Code Template for Statemachines

ⓘ Write a code template for instances of Statemachine

Write a code template for Statemachines 36

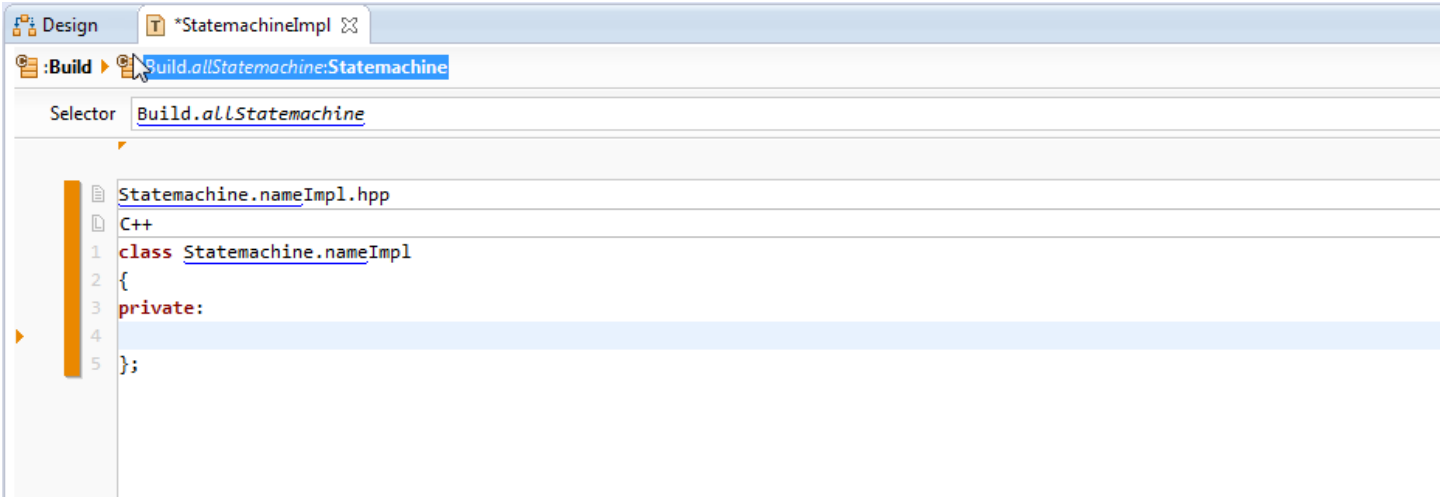


- Create a package `com.actifsource.statemachine.template`
- Select the new package and choose **New->Template** from the context menu.

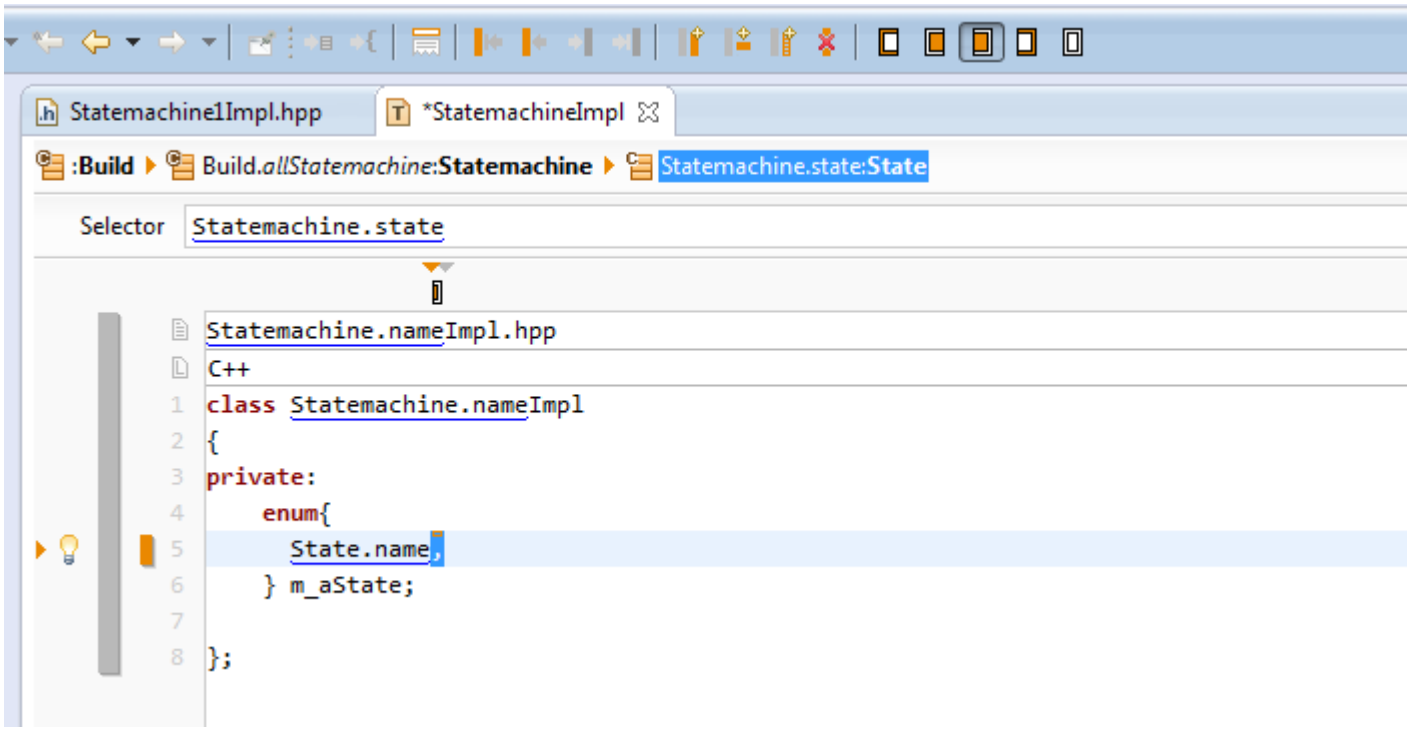


- Insert `StatemachineImpl` as Template Name
- Choose the Base Type com.actifsource.statemachine.generic.Statemachine
- Click **Finish**

Write a code template for Statemachines



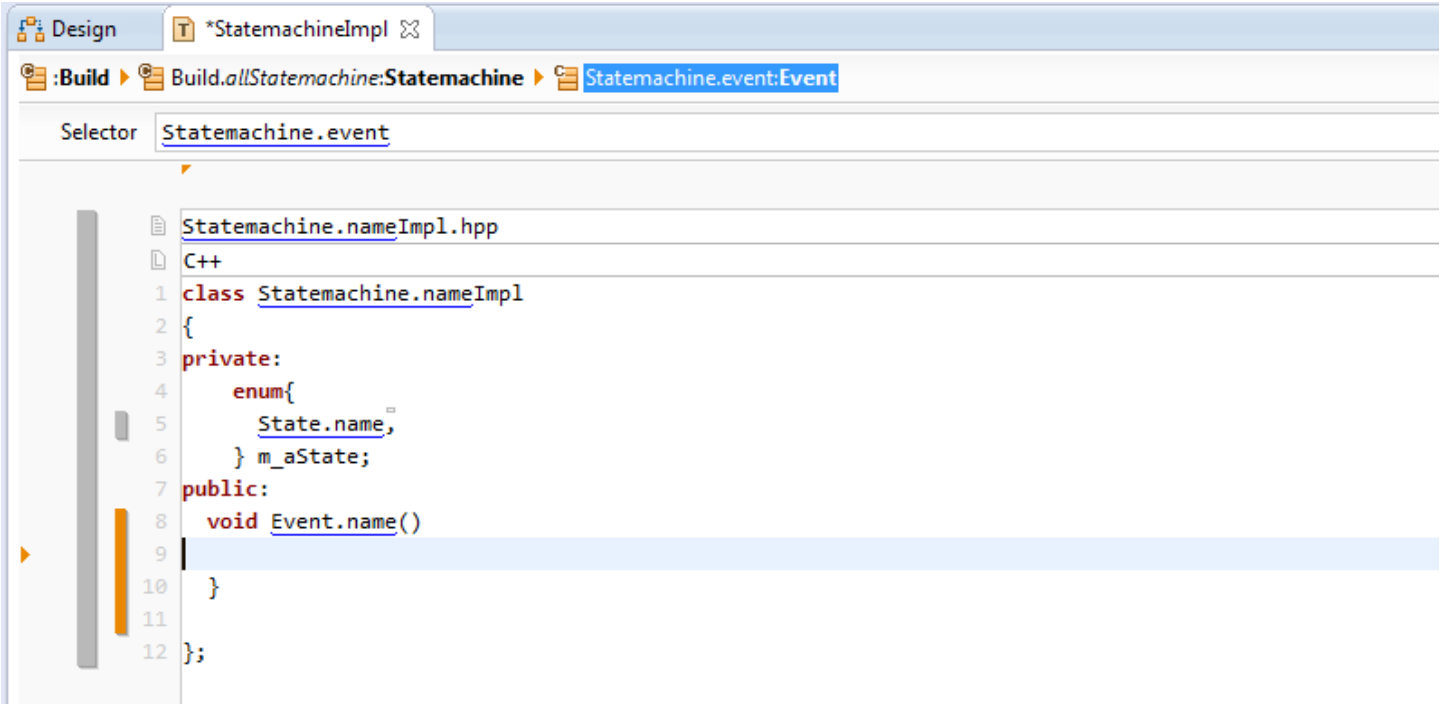
- Insert Statemachine.nameImpl.hpp on the Filename Line and make sure that the language (C++) is automatically detected.
- Write the skeleton for a class Statemachine.nameImpl



Next, we define an enumeration variable with the all the States of a Statemachine as enumerators. This variable stores the current state of a Statemachine:

- Write the declaration of enumeration variable *m_aState*
- Insert a LineContext in the enumeration list and choose the Selector Statemachine.state with the support of the Content Assist
- Insert State.name in the newly created LineContext. Append a ','. Then mark the ',' and select `NotLast` to make sure that there is no comma after the last entry in the enumeration list.

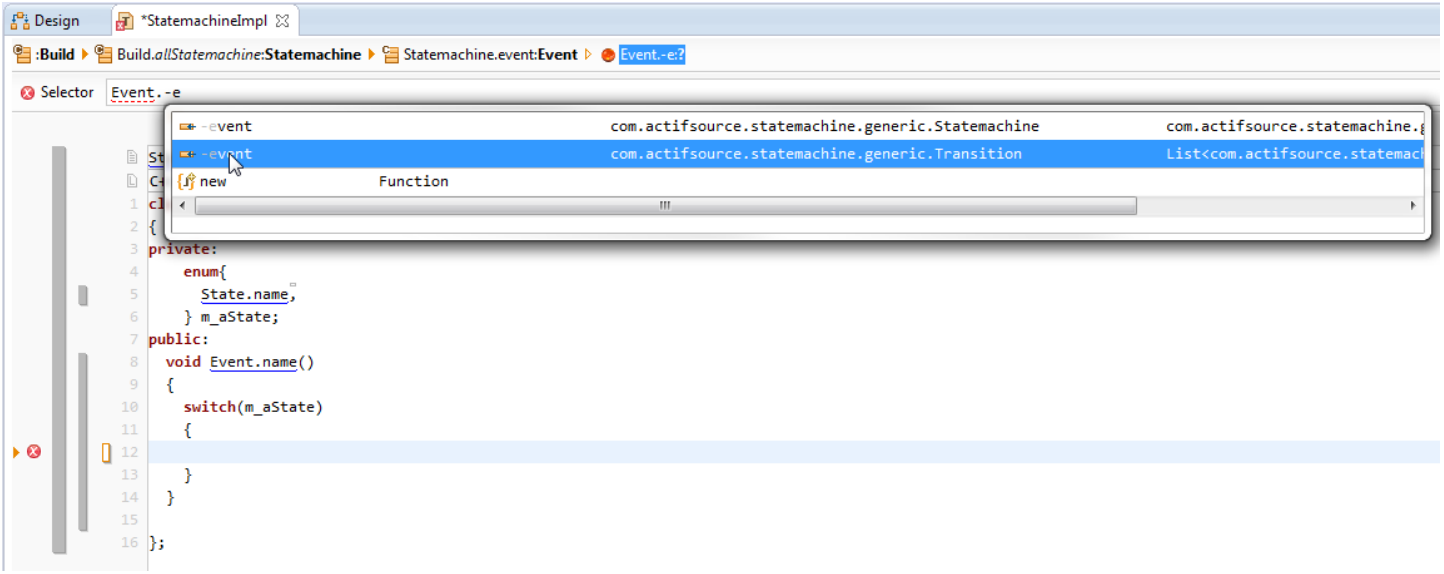
Write a code template for Statemachines 40



We define a member function for each event of our Statemachine, which will later handle all the possible transitions triggered by the event:

- Create a new LineContext and choose Statemachine.event as the selector of the line context
- Write the skeleton of a function returning void named Event.name

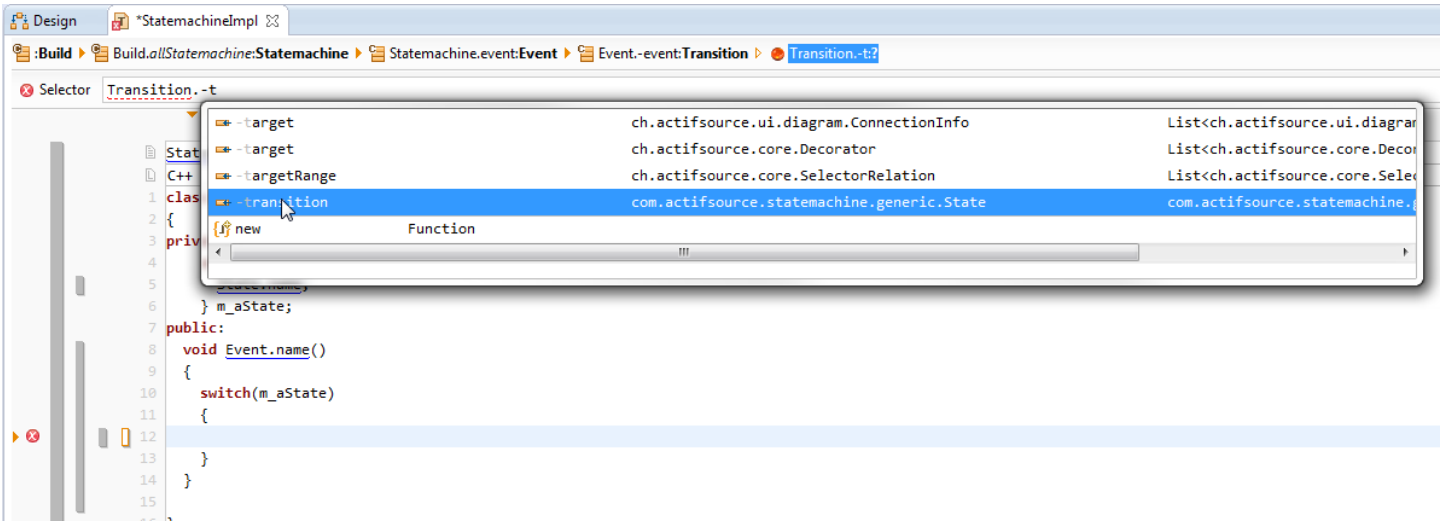
Write a code template for Statemachines



We write a switch-statement with the current state `m_aState` as control variable and define a LineContext that iterates over all Transitions referring to an Event through the relation Transition.event*:*

- Create a switch-statement with the `m_aState` as control variable
- Create a LineContext inside the switch-statement
- Choose Event.-event as the Selector of the new LineContext

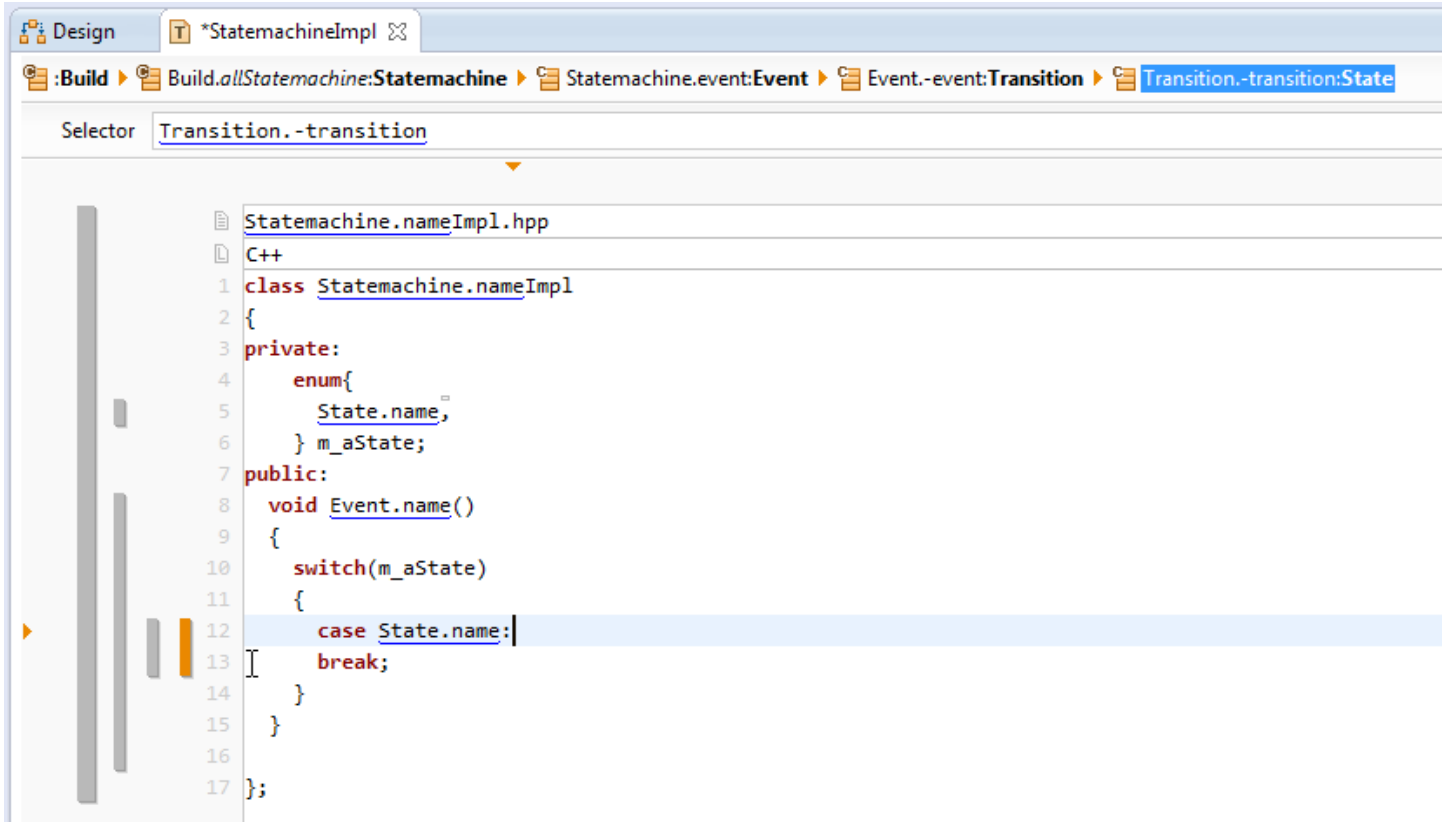
Write a code template for Statemachines 42



We create a LineContext that iterates over all States that refer to a Transition through the relation State.transition*:*

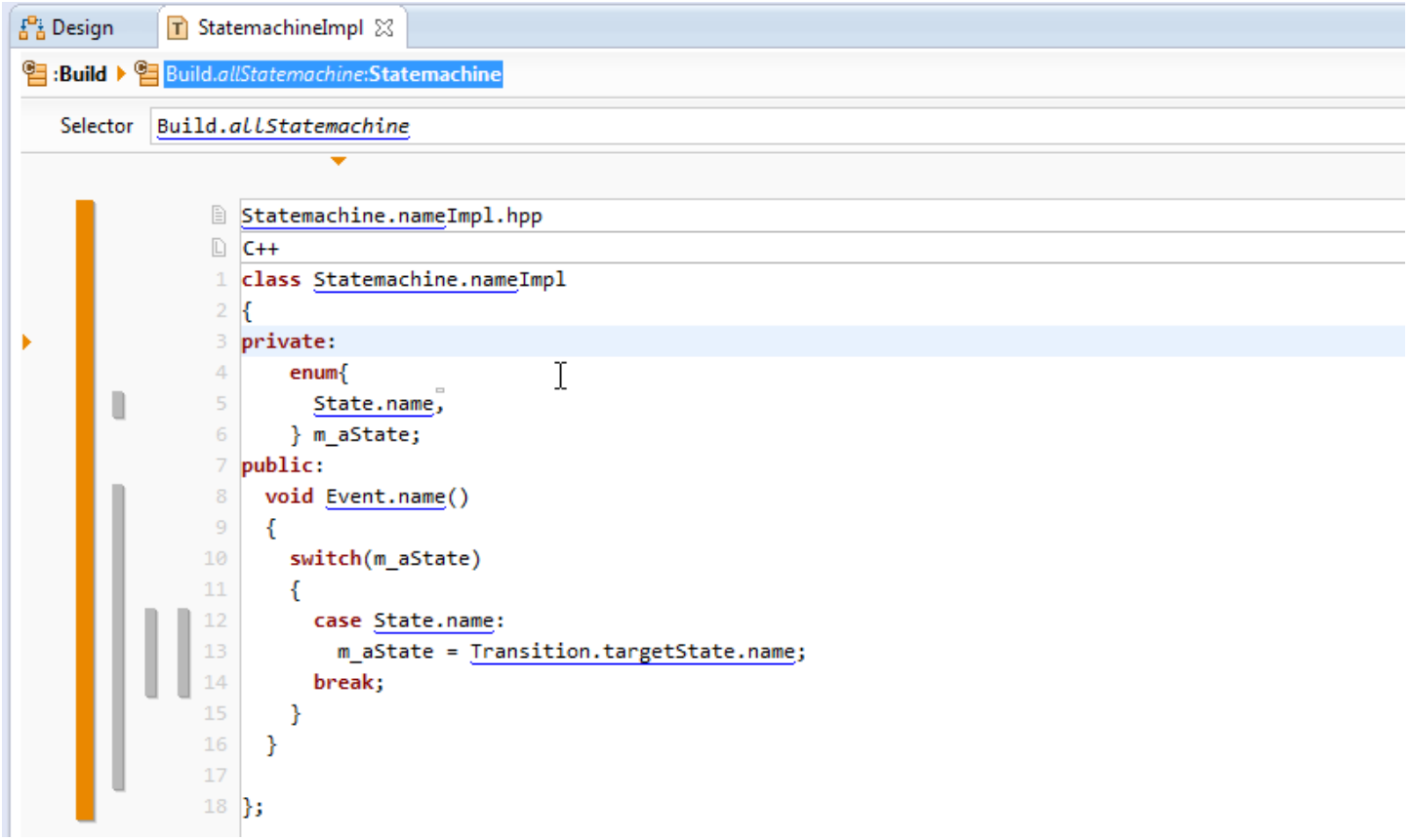
- Create a LineContext on the same line as LineContext that we have crated before
- Choose Transition.-transition as the Selector of the new LineContext

Write a code template for Statemachines



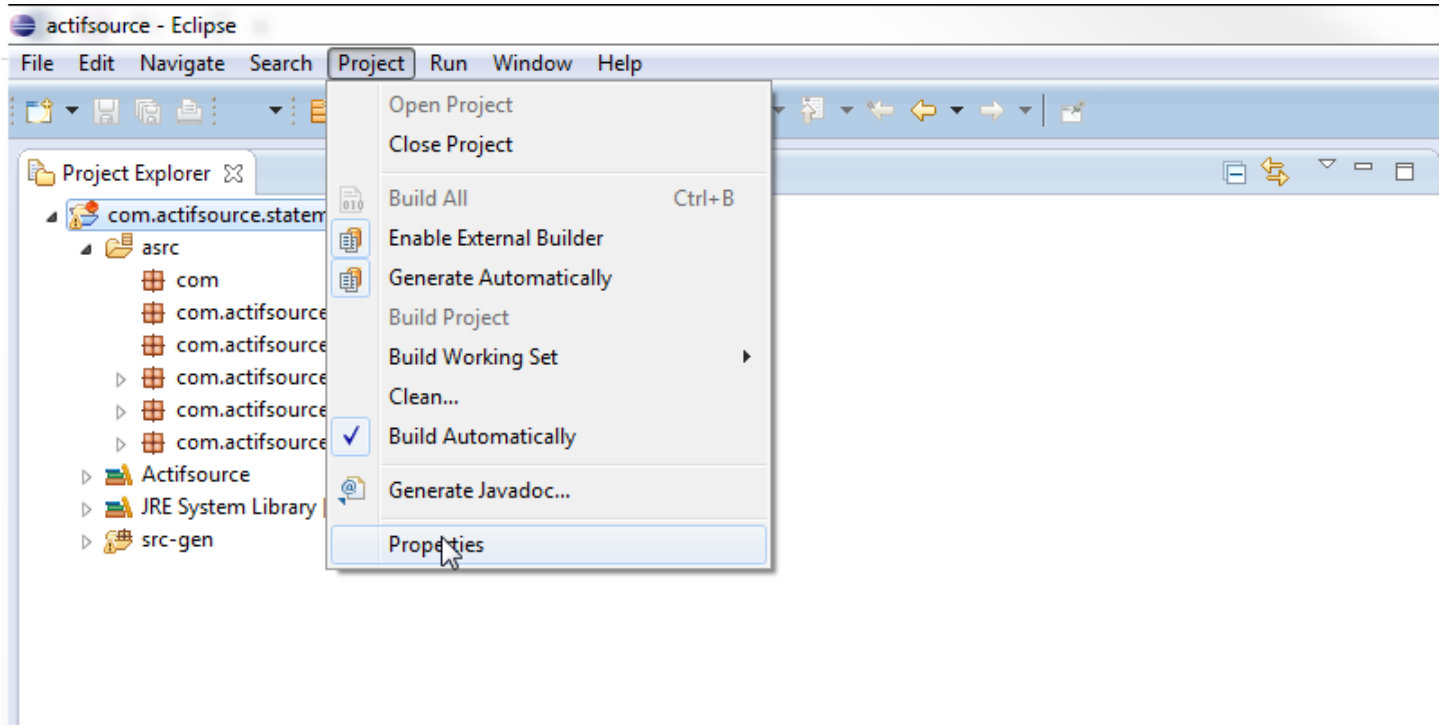
We write a case-statement for each State that is (indirectly) referring to an Event through State.transition.event:

- Insert a `case` State.name and add a `break` at the end of the case-statement



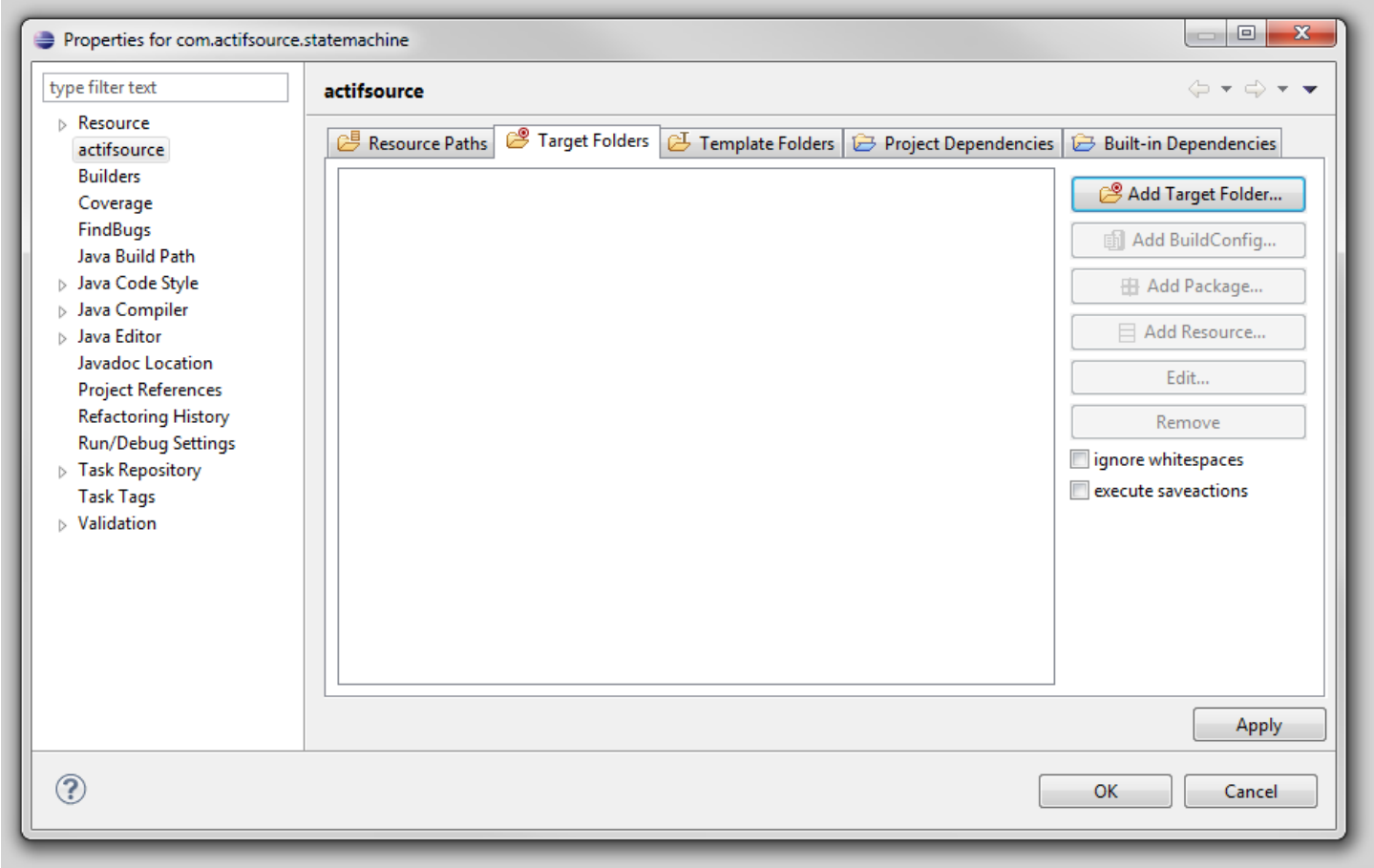
We update the current state as follows: We first select for an Event the Transitions that refer to the Event trough Transition.event. For each Transition, we select the States that are connected to Transition by State.transition. For each State, it holds that if the current state `m_aState` is equal to State, then the new State of the Statemachine is Transition.targetState:

- Write code to assign Transition.targetState.name to the variable `m_aState`

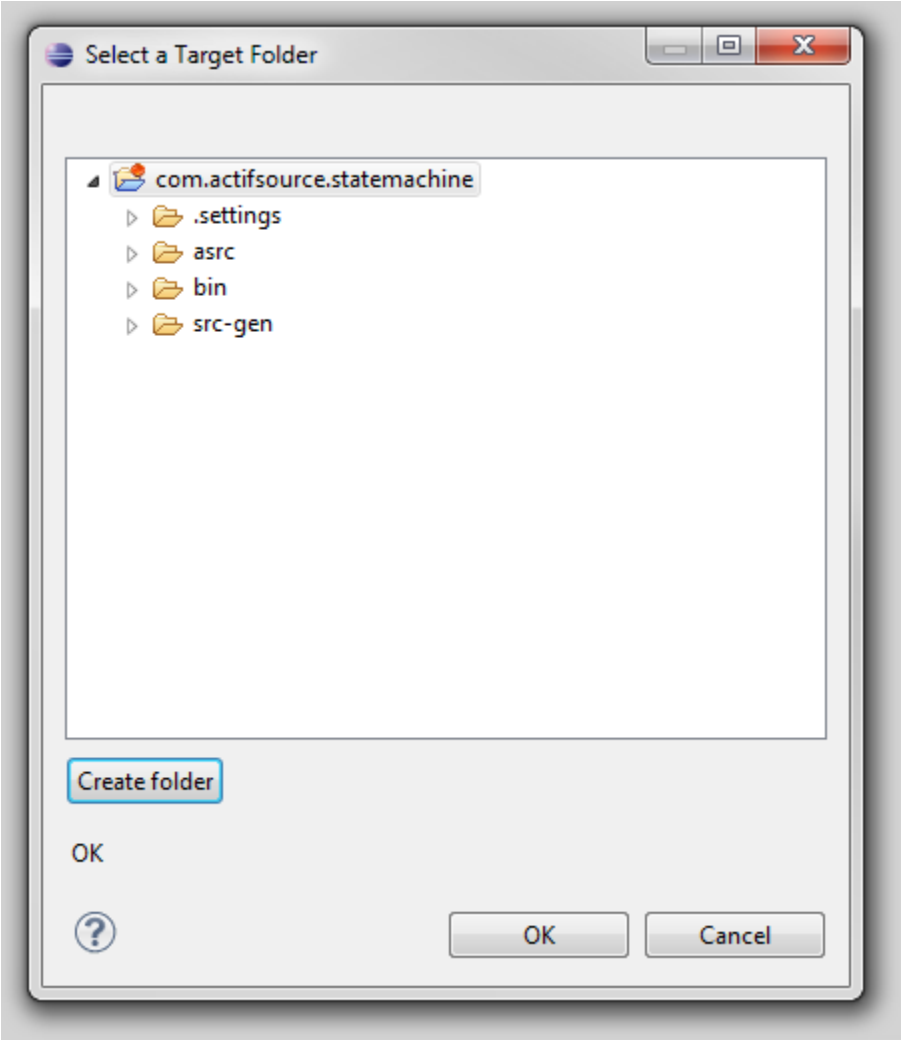


In order to generate code from the template we have implemented before, we setup the project properties for Actifsource:

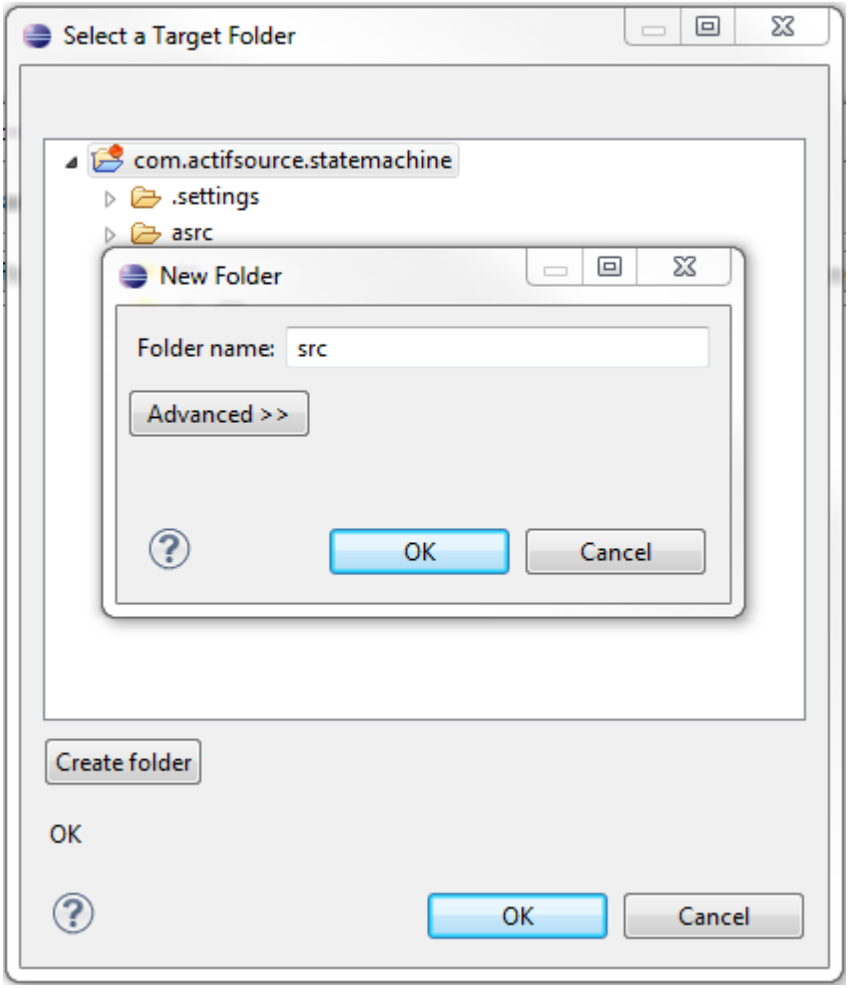
- Select the project com.actifsource.statemachine and choose **Project->Properties** from the main menu



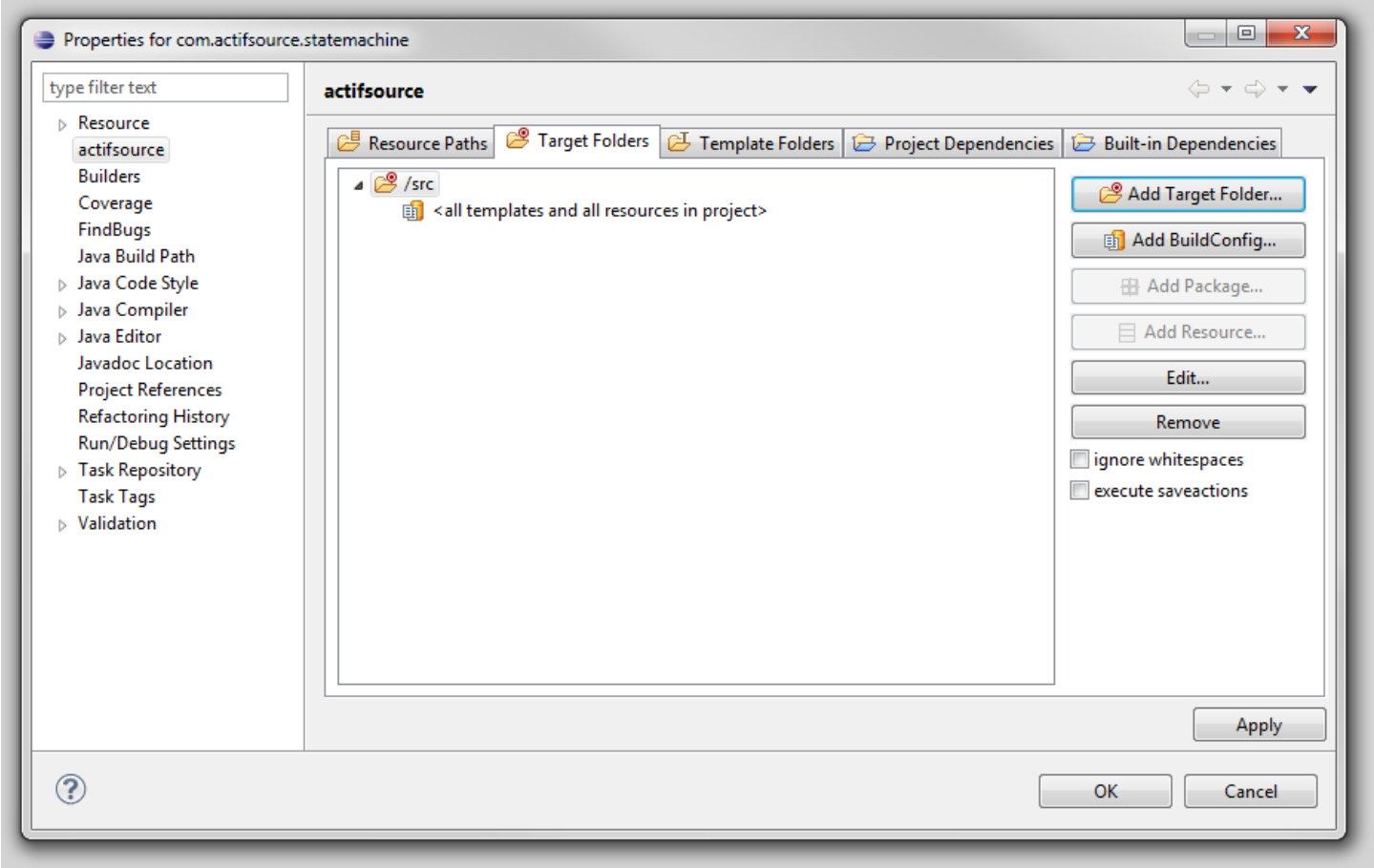
↳ In the Properties dialog choose **actifsource** and select the tab **Target Folders**



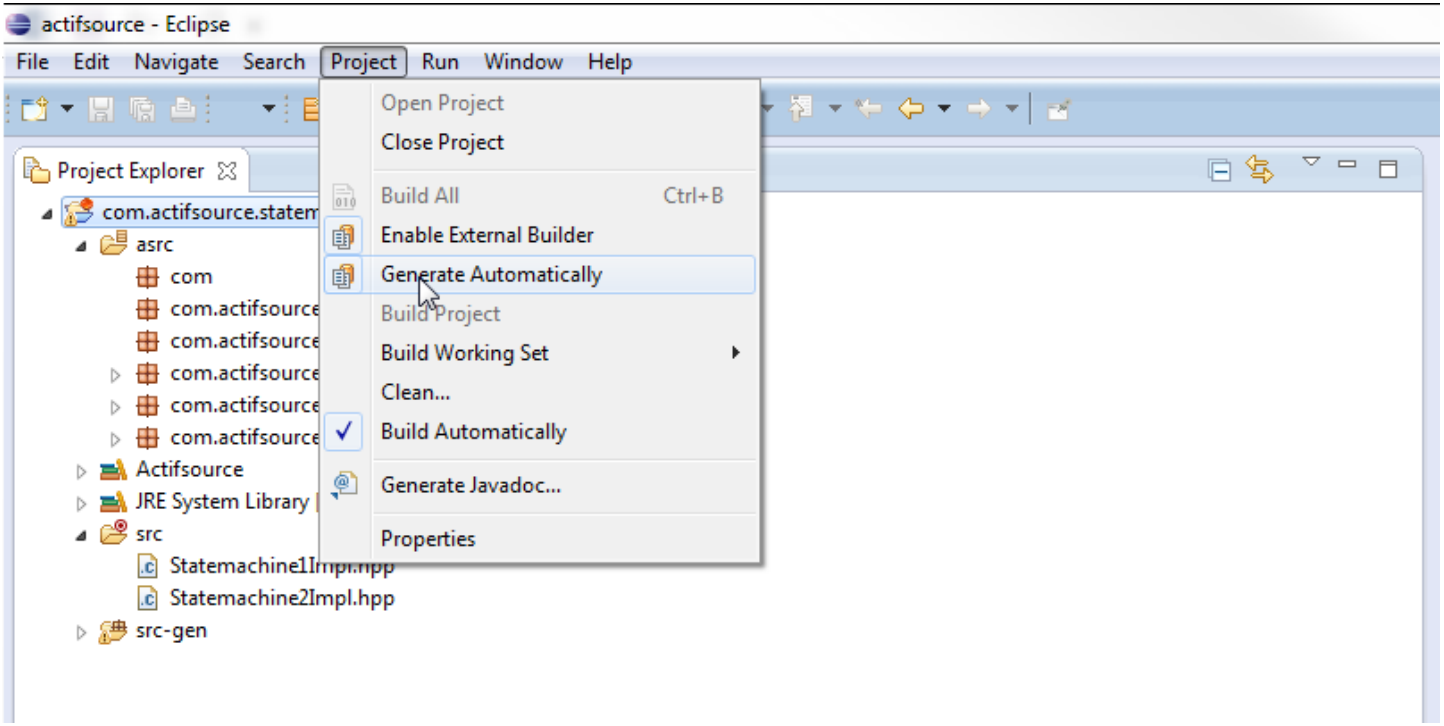
- In the dialog **Select Target Folder,** click on the button **Create folder**



- Enter `src` as Folder Name in the **New Folder** dialog
- Click on **OK** in the New Folder dialog and then in the **Select Target Folder** dialog



↳ Check the settings on the **Target Folders** tab and close the dialog by clicking on **OK**



The code generator now applies the template StatemachineImpl to the two Statemachine instances and stores the resulting files to the `src` folder:

- Open the `src` folder and check that the two files `Statemachine1Impl.hpp` and `Statemachine2Impl.hpp` have been generated
- If the files have not been generated, make sure that **Generate Automatically** is active under **Project** in the main menu

Statemachine1Impl.hpp

```
class Statemachine1Impl
{
private:
    enum{
      Initialized,
      Started,
      Stopped
    } m_aState;

public:
  void start()
  {
    switch(m_aState)
    {
      case Initialized:
        m_aState = Started;
      break;
      case Stopped:
        m_aState = Started;
      break;
    }
  }

  void stop()
  {
    switch(m_aState)
    {
      case Started:
        m_aState = Stopped;
      break;
    }
  }

};
/* Actifsource ID=[0b6aff85-85f9-11e4-a105-d1f
```

Statemachine2Impl.hpp

```
class Statemachine2Impl
{
private:
    enum{
      Initialized,
      Opened,
      Closed
    } m_aState;

public:
  void open()
  {
    switch(m_aState)
    {
      case Initialized:
        m_aState = Opened;
      break;
      case Closed:
        m_aState = Opened;
      break;
    }
  }

  void close()
  {
    switch(m_aState)
    {
      case Opened:
        m_aState = Closed;
      break;
    }
  }

};
/* Actifsource ID=[0b6aff85-85f9-11e4-a105-c
```

↳ Open the newly generated files and inspect and compare the code for the two Statemachines

ⓘ Learn how to extend the Statemachine by conditional transitions and actions executed together with a transition by working through the Actifsource Tutorial – Code Snippets

ⓘ Complete the generated classes by adding a member function initialize()