CAPS Framework

Cristian Capozucco - Davide Mariotti - Luca Grillo

Contents

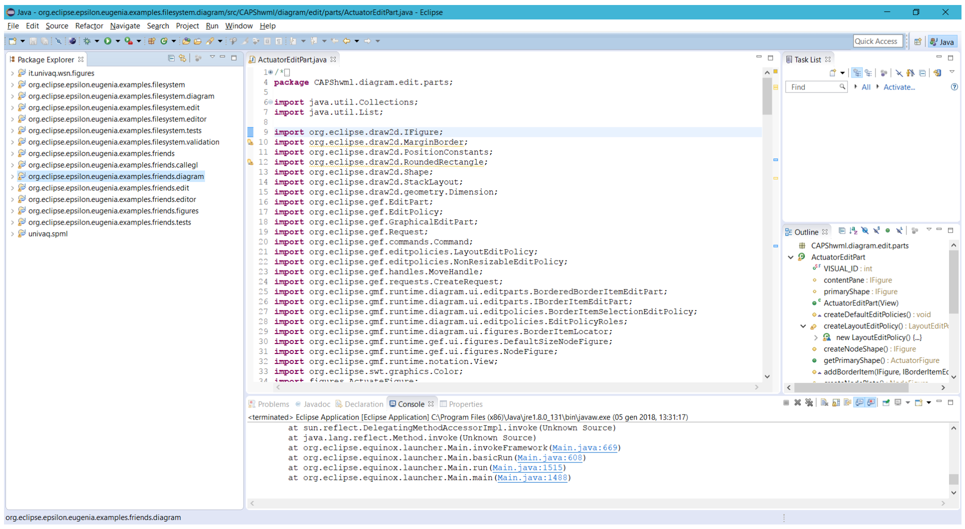
Chapter 1

Installing the framework

- Download e install **JRE - windows x86 Offline**[1]
- Download **this zip**[2]
- Unzip the **.zip** you have downloaded
- Open the **Eclipse provided by Mohammad Sharaf** (the one inside the folder **eclipse-epsilon-1.3-win32**);
- Go to **File**, **Import**, **General** and **Existing projects into Workspace**. Select **Root Directory**, click on **Browse** and select the folder **Packages** that you have unzipped. Check all the checkboxes and click on **Finish**
- Then, you will see something like the figure 1.1;
- Right click on **org.ecplise.epsilon.eugenia.examples.friends.diagram** (figure 1.2);

[1] `http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html`

[2] `https://www.dropbox.com/s/bud9ae62khtobid/CAPS.7z?dl=0`


Java - org.eclipse.epsilon.eugenia.examples.filesystem.diagram/src/CAPShwml/diagram/edit/parts/ActuatorEditPart.java - Eclipse

Package Explorer
- it.univaq.wsn.figures
- org.eclipse.epsilon.eugenia.examples.filesystem
- org.eclipse.epsilon.eugenia.examples.filesystem.diagram
- org.eclipse.epsilon.eugenia.examples.filesystem.edit
- org.eclipse.epsilon.eugenia.examples.filesystem.editor
- org.eclipse.epsilon.eugenia.examples.filesystem.tests
- org.eclipse.epsilon.eugenia.examples.filesystem.validation
- org.eclipse.epsilon.eugenia.examples.friends
- org.eclipse.epsilon.eugenia.examples.friends.callegl
- org.eclipse.epsilon.eugenia.examples.friends.diagram
- org.eclipse.epsilon.eugenia.examples.friends.edit
- org.eclipse.epsilon.eugenia.examples.friends.editor
- org.eclipse.epsilon.eugenia.examples.friends.figures
- org.eclipse.epsilon.eugenia.examples.friends.tests
- univaq.spml

ActuatorEditPart.java

```java
/*
package CAPShwml.diagram.edit.parts;

import java.util.Collections;
import java.util.List;

import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.MarginBorder;
import org.eclipse.draw2d.PositionConstants;
import org.eclipse.draw2d.RoundedRectangle;
import org.eclipse.draw2d.Shape;
import org.eclipse.draw2d.StackLayout;
import org.eclipse.draw2d.geometry.Dimension;
import org.eclipse.gef.EditPart;
import org.eclipse.gef.EditPolicy;
import org.eclipse.gef.GraphicalEditPart;
import org.eclipse.gef.Request;
import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.LayoutEditPolicy;
import org.eclipse.gef.editpolicies.NonResizableEditPolicy;
import org.eclipse.gef.handles.MoveHandle;
import org.eclipse.gef.requests.CreateRequest;
import org.eclipse.gmf.runtime.diagram.ui.editparts.BorderedBorderItemEditPart;
import org.eclipse.gmf.runtime.diagram.ui.editparts.IBorderItemEditPart;
import org.eclipse.gmf.runtime.diagram.ui.editpolicies.BorderItemSelectionEditPolicy;
import org.eclipse.gmf.runtime.diagram.ui.editpolicies.EditPolicyRoles;
import org.eclipse.gmf.runtime.diagram.ui.figures.BorderItemLocator;
import org.eclipse.gmf.runtime.gef.ui.figures.DefaultSizeNodeFigure;
import org.eclipse.gmf.runtime.gef.ui.figures.NodeFigure;
import org.eclipse.gmf.runtime.notation.View;
import org.eclipse.swt.graphics.Color;
```

Outline
- CAPShwml.diagram.edit.parts
- ActuatorEditPart
  - VISUAL_ID : int
  - contentPane : IFigure
  - primaryShape : IFigure
  - ActuatorEditPart(View)
  - createDefaultEditPolicies() : void
  - createLayoutEditPolicy() : LayoutEditP
    - new LayoutEditPolicy() {...}
  - createNodeShape() : IFigure
  - getPrimaryShape() : ActuatorFigure
  - addBorderItem(IFigure, IBorderItemEc

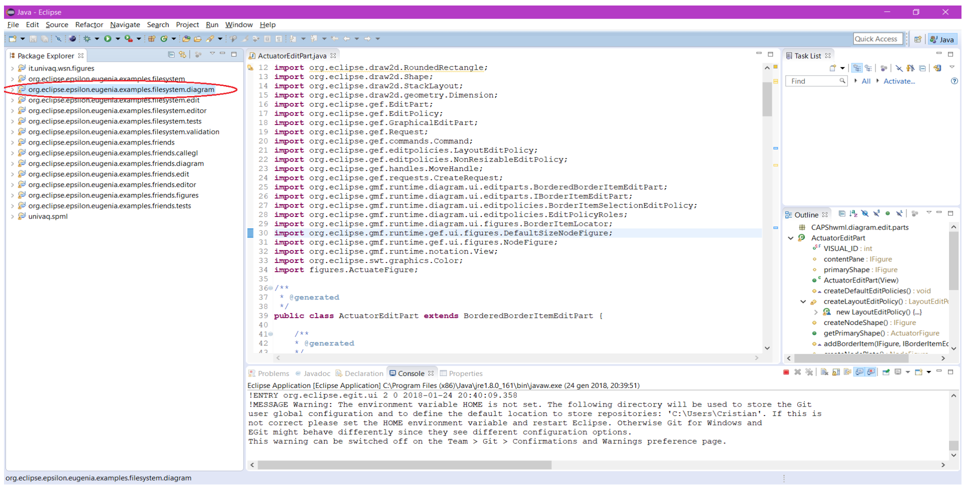
Console

```
<terminated> Eclipse Application [Eclipse Application] C:\Program Files (x86)\Java\jre1.8.0_131\bin\javaw.exe (05 gen 2018, 13:31:17)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at java.lang.reflect.Method.invoke(Unknown Source)
        at org.eclipse.equinox.launcher.Main.invokeFramework(Main.java:669)
        at org.eclipse.equinox.launcher.Main.basicRun(Main.java:608)
        at org.eclipse.equinox.launcher.Main.run(Main.java:1515)
        at org.eclipse.equinox.launcher.Main.main(Main.java:1488)
```

org.eclipse.epsilon.eugenia.examples.friends.diagram

Figure 1.1: Eclipse provided by Mohammad Sharaf

- In the appearing menu, go on **Run as** and then **Eclipse Application** (figure 1.3);
- After a while, you should see something like the screen of the figure 1.4;

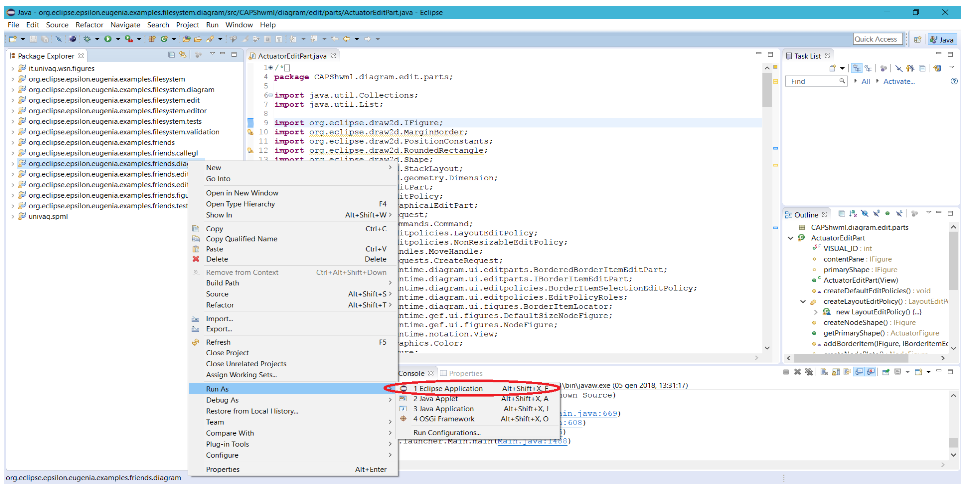


```java
import org.eclipse.draw2d.RoundedRectangle;
import org.eclipse.draw2d.Shape;
import org.eclipse.draw2d.StackLayout;
import org.eclipse.draw2d.geometry.Dimension;
import org.eclipse.gef.EditPart;
import org.eclipse.gef.EditPolicy;
import org.eclipse.gef.GraphicalEditPart;
import org.eclipse.gef.Request;
import org.eclipse.gef.commands.Command;
import org.eclipse.gef.editpolicies.LayoutEditPolicy;
import org.eclipse.gef.editpolicies.NonResizableEditPolicy;
import org.eclipse.gef.handles.MoveHandle;
import org.eclipse.gef.requests.CreateRequest;
import org.eclipse.gmf.runtime.diagram.ui.editparts.BorderedBorderItemEditPart;
import org.eclipse.gmf.runtime.diagram.ui.editparts.IBorderItemEditPart;
import org.eclipse.gmf.runtime.diagram.ui.editpolicies.BorderItemSelectionEditPolicy;
import org.eclipse.gmf.runtime.diagram.ui.editpolicies.EditPolicyRoles;
import org.eclipse.gmf.runtime.diagram.ui.figures.BorderItemLocator;
import org.eclipse.gmf.runtime.gef.ui.figures.DefaultSizeNodeFigure;
import org.eclipse.gmf.runtime.gef.ui.figures.NodeFigure;
import org.eclipse.gmf.runtime.notation.View;
import org.eclipse.swt.graphics.Color;
import figures.ActuateFigure;

/**
 * @generated
 */
public class ActuatorEditPart extends BorderedBorderItemEditPart {

    /**
     * @generated
```

```
Eclipse Application [Eclipse Application] C:\Program Files (x86)\Java\jre1.8.0_161\bin\javaw.exe (24 gen 2018, 20:39:51)
!ENTRY org.eclipse.egit.ui 2 0 2018-01-24 20:40:09.358
!MESSAGE Warning: The environment variable HOME is not set. The following directory will be used to store the Git
user global configuration and to define the default location to store repositories: 'C:\Users\Cristian'. If this is
not correct please set the HOME environment variable and restart Eclipse. Otherwise Git for Windows and
EGit might behave differently since they see different configuration options.
This warning can be switched off on the Team > Git > Confirmations and Warnings preference page.
```

Java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
it.univaq.wsn.figures
org.eclipse.epsilon.eugenia.examples.filesystem
org.eclipse.epsilon.eugenia.examples.filesystem.diagram
org.eclipse.epsilon.eugenia.examples.filesystem.edit
org.eclipse.epsilon.eugenia.examples.filesystem.editor
org.eclipse.epsilon.eugenia.examples.filesystem.tests
org.eclipse.epsilon.eugenia.examples.filesystem.validation
org.eclipse.epsilon.eugenia.examples.friends
org.eclipse.epsilon.eugenia.examples.friends.callegl
org.eclipse.epsilon.eugenia.examples.friends.diagram
org.eclipse.epsilon.eugenia.examples.friends.edit
org.eclipse.epsilon.eugenia.examples.friends.editor
org.eclipse.epsilon.eugenia.examples.friends.figures
org.eclipse.epsilon.eugenia.examples.friends.tests
univaq.spml
ActuatorEditPart.java
Problems Javadoc Declaration Console Properties
Task List
Outline
CAPShwml.diagram.edit.parts
ActuatorEditPart
VISUAL_ID : int
contentPane : IFigure
primaryShape : IFigure
ActuatorEditPart(View)
createDefaultEditPolicies() : void
createLayoutEditPolicy() : LayoutEditP
new LayoutEditPolicy() {...}
createNodeShape() : IFigure
getPrimaryShape() : ActuatorFigure
addBorderItem(IFigure, IBorderItemEd
org.eclipse.epsilon.eugenia.examples.filesystem.diagram

Figure 1.2: The package you have to right click on



```java
package CAPShwml.diagram.edit.parts;

import java.util.Collections;
import java.util.List;

import org.eclipse.draw2d.IFigure;
import org.eclipse.draw2d.MarginBorder;
import org.eclipse.draw2d.PositionConstants;
import org.eclipse.draw2d.RoundedRectangle;
```

Java - org.eclipse.epsilon.eugenia.examples.filesystem.diagram/src/CAPShwml/diagram/edit/parts/ActuatorEditPart.java - Eclipse
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Run As
Debug As
Restore from Local History...
Team
Compare With
Plug-in Tools
Configure
Properties Alt+Enter
1 Eclipse Application Alt+Shift+X, E
2 Java Applet Alt+Shift+X, A
3 Java Application Alt+Shift+X, J
4 OSGi Framework Alt+Shift+X, O
Run Configurations...
org.eclipse.epsilon.eugenia.examples.friends.diagram

Figure 1.3: The option you have to select

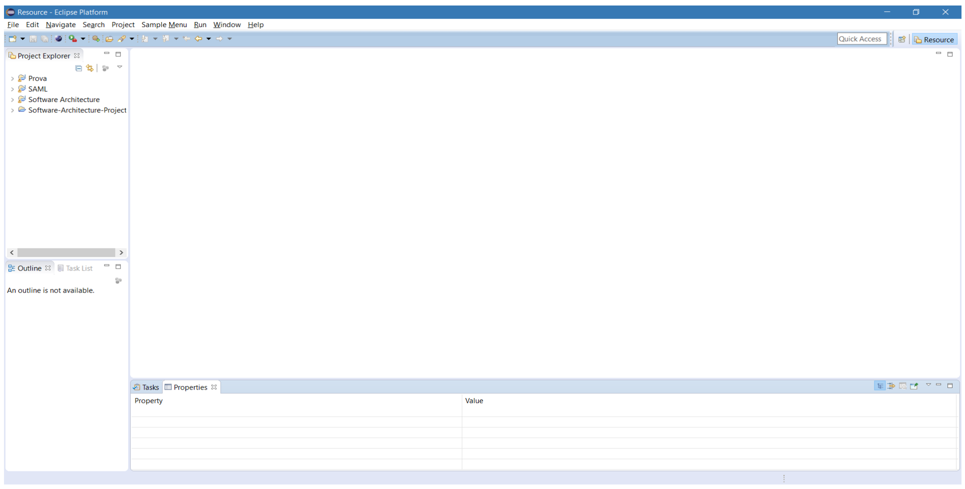

Resource - Eclipse Platform
File Edit Navigate Search Project Sample Menu Run Window Help
Quick Access
Resource
Project Explorer
Prova
SAML
Software Architecture
Software-Architecture-Project
Outline
Task List
An outline is not available.
Tasks
Properties
Property
Value

Figure 1.4: Initial screen of the framework

Chapter 2

SAML

2.1 Example

Now, we will see how to create a project to represent a simple scenario: image that there is a room with a temperature sensor with an actuator connected either to the sensor and to a window. When the temperature sensed is too high, the actuator will open the window. If the temperature is too low, then the window will be closed. Besides, the data sensed are saved in a remote server.

- Let's create the **SAML** sample project: go on **File, New, Project** (figure 2.1). Find **EMF** and select **Empty EMF Project** (figure 2.2). Then, choose a name and a location for your project (figure 2.3). By checking the **default location** checkbox, the project will be created inside the location of your workspace;

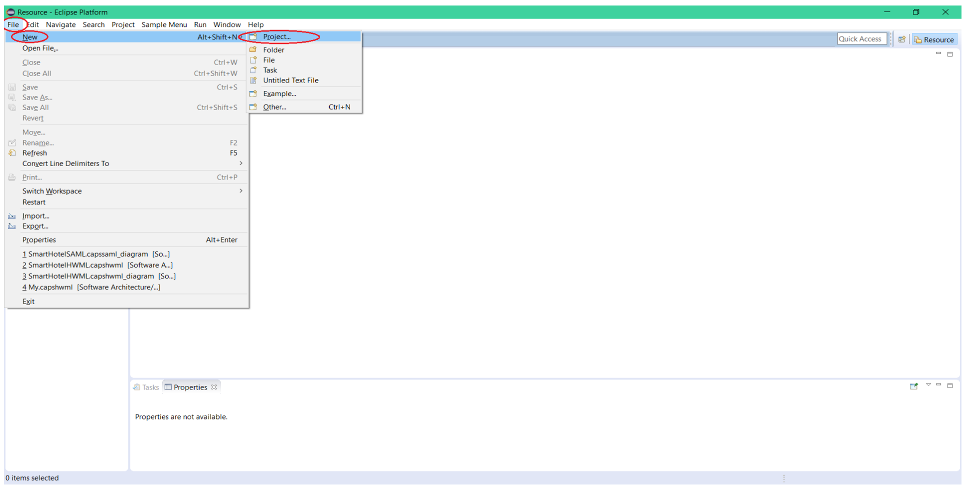


Figure 2.1: Where you can select the Project option

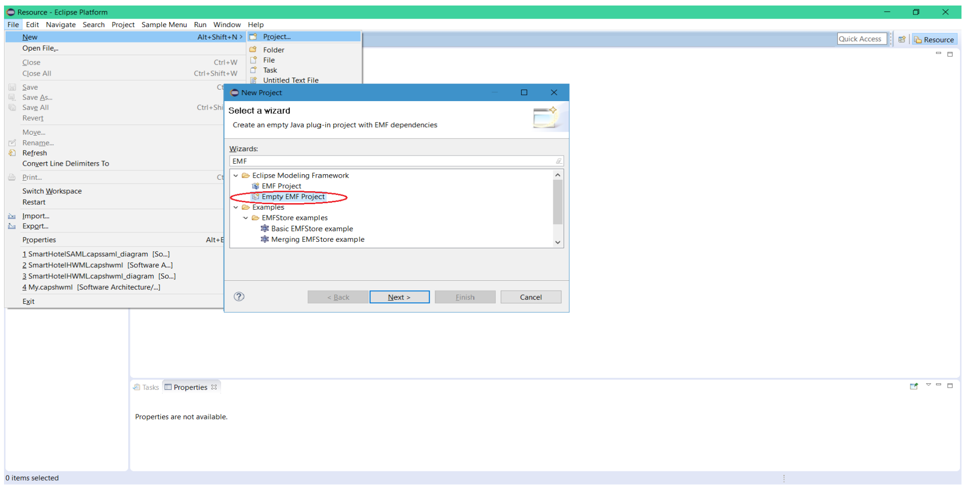
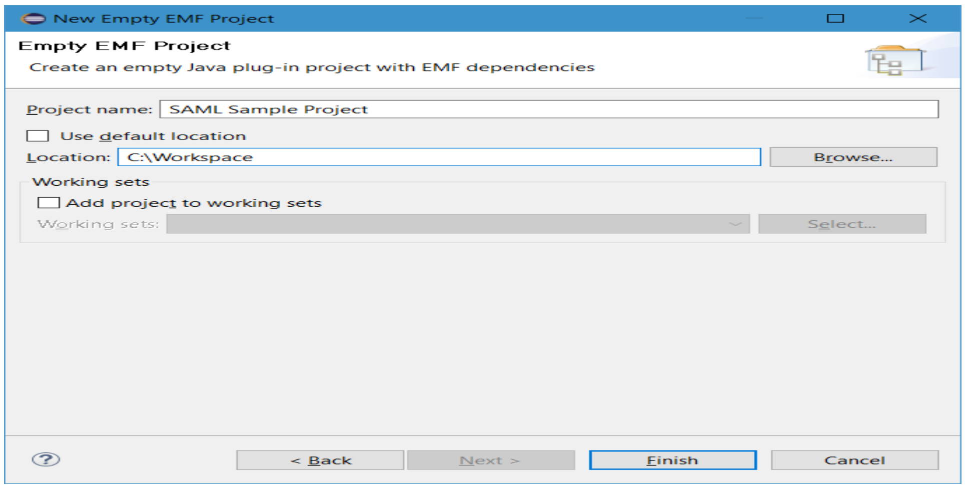

Resource - Eclipse Platform
File Edit Navigate Search Project Sample Menu Run Window Help
New Project
Select a wizard
Create an empty Java plug-in project with EMF dependencies
Wizards:
EMF
Eclipse Modeling Framework
EMF Project
Empty EMF Project
Examples
EMFStore examples
Basic EMFStore example
Merging EMFStore example
< Back
Next >
Finish
Cancel
Properties are not available.
0 items selected

Figure 2.2: The type of project you have to select



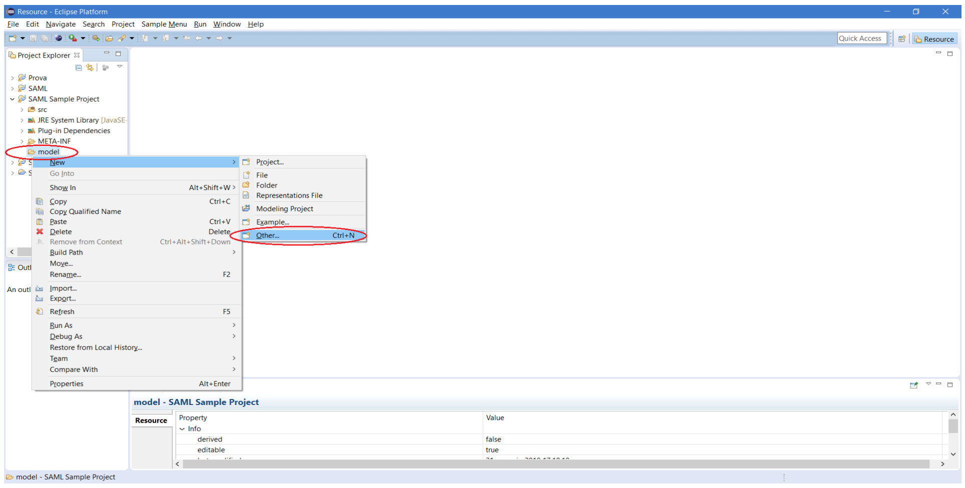
Project name: SAML Sample Project

☐ Use default location

Location: C:\Workspace

Figure 2.3: Name and location

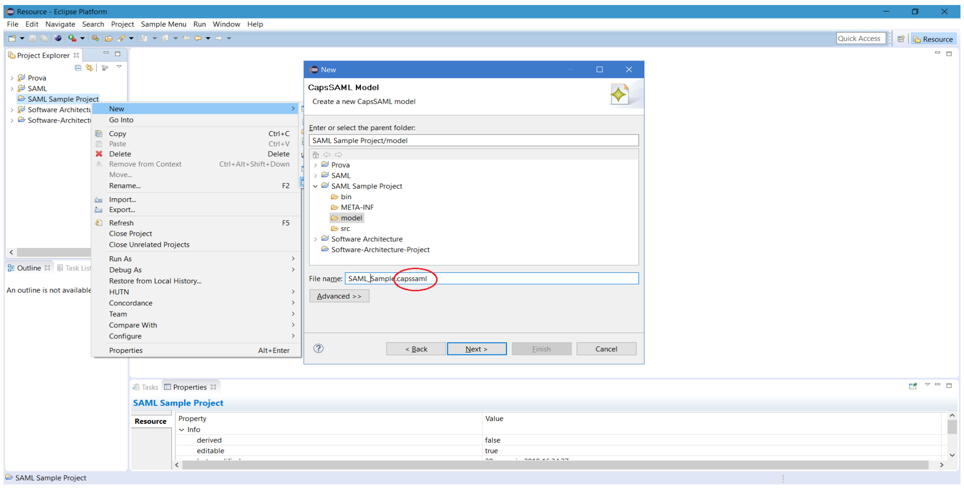
- Click on finish. Now, as you can see on the left, there is the package of your project. Expand it, right click on the **Model** folder, select new and then other (figure 2.4);



Property	Value
Info	
derived	false
editable	true

Figure 2.4: Right click on the folder and find other

- Then, find **CAPSModel** and then **CAPSSaml**. Give a name to your file paying attention to the last part of the name: it must end with **.capssaml** (concerning the folder, select **model** as well) (figure 2.5);

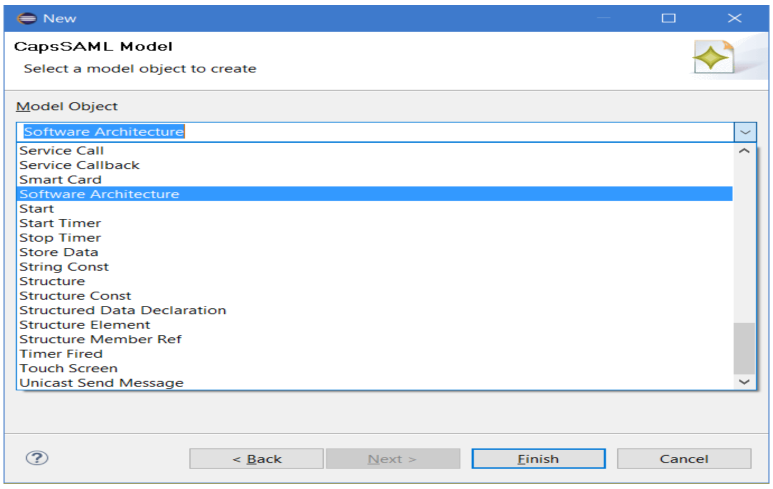


Enter or select the parent folder: SAML Sample Project/model

File name: SAML_Sample.capssaml

Figure 2.5: Naming the file

- Click on next and select **Software Architecture** from the **Model Object** items list and click finish (figure 2.6);



Model Object: Software Architecture

Figure 2.6: List of the items

- From the project explorer on the left, right click on the new file that have been created and select **Initialize friends_diagram diagram file**. This will let you to create the SAML diagrams (figure 2.7);

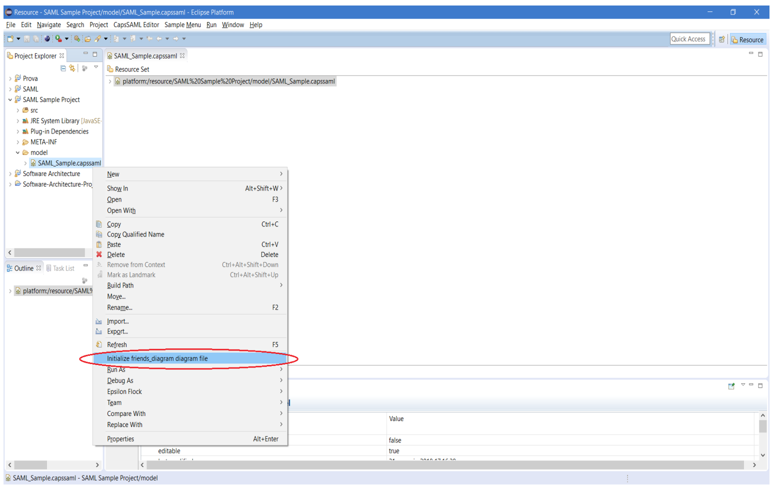


Figure 2.7: Initialize the diagram

- In the appearing window, you can select an other location and name for the file that will be created. I suggest to leave the default settings and to click on finish (figure 2.8);

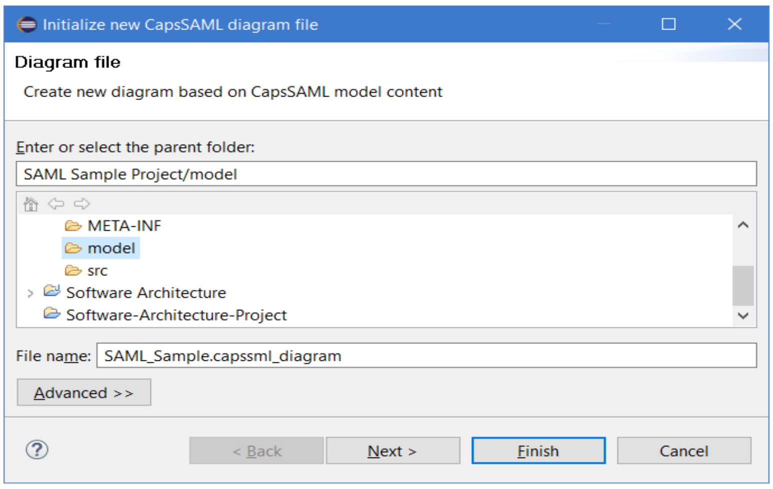

Initialize new CapsSAML diagram file

Diagram file

Create new diagram based on CapsSAML model content

Enter or select the parent folder: SAML Sample Project/model

File name: SAML_Sample.capssml_diagram

Figure 2.8: Name and location of the new file

- Now, new objects will appear on the window: as you can see, on the right, there is a **palette section**, where you can take all the items that you need for your architecture. (figure 2.9);

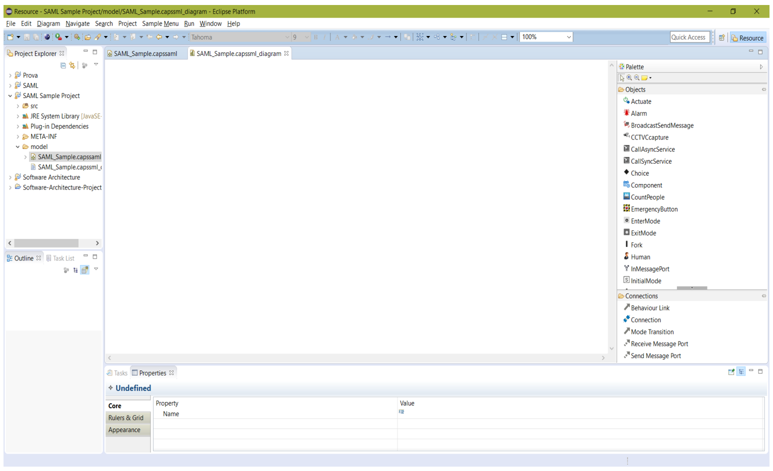


Figure 2.9: The new view

- Let's create something. Imagine that we want to model the following scenario: a temperature sensor that senses the temperature and sends it to a server. First of all, find the **component item** in the palette. Select it and create it in the center of the view (just create a rectangle like in paint: left click and enlarge it as you wish) (figure 2.10);

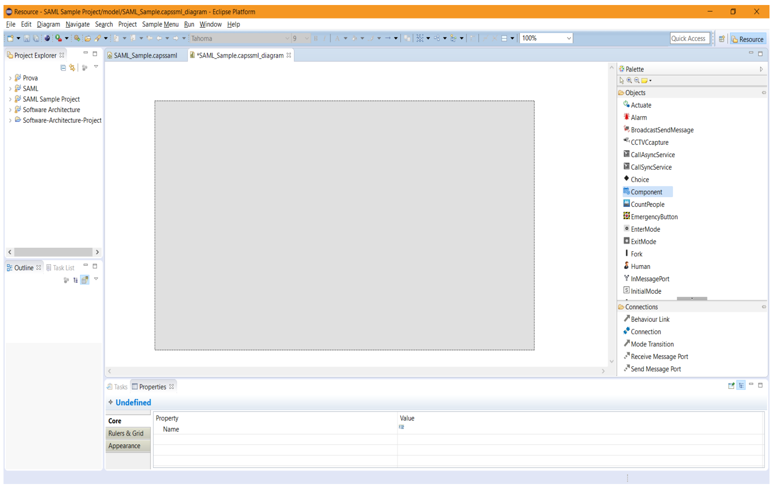
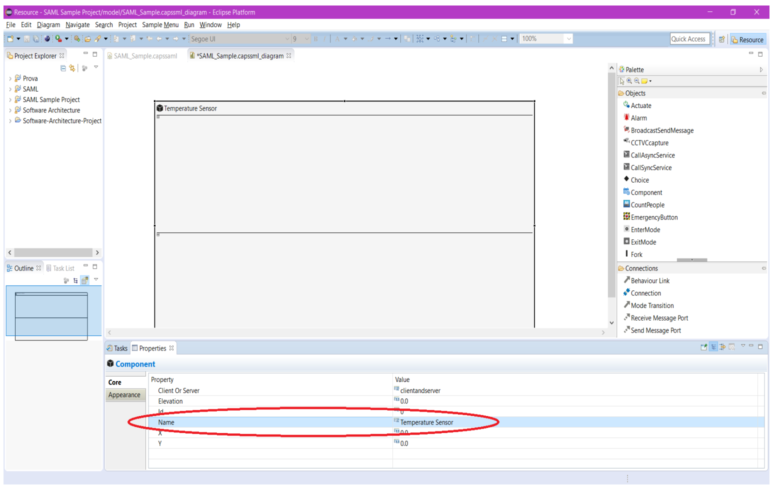


Figure 2.10: Creating the component

- Congratulations! You have just created your first SAML component. Now, give it a name: as you can see, at the bottom of the view, there is the **Properties section**. Enlarge it and search the **Name row**: select the blank part and type **Temperature Sensor** (or something else) (figure 2.11);



Property	Value
Client Or Server	clientandserver
Elevation	0.0
Id	0
Name	Temperature Sensor
X	0.0
Y	0.0

Figure 2.11: Naming the component

- Now you have a simple blank named component. Let's create some logic inside it. **The very first thing you have to do**, is to put inside the component the **Initial mode**. Search it in the **Palette view** e put it inside the component (if you try to create the mode outside the component, well, you will not able to do that) (figure 2.12). What is an **initial mode**: it's the group of logic that is active when **the component starts working**. It will be more clear in a few steps;

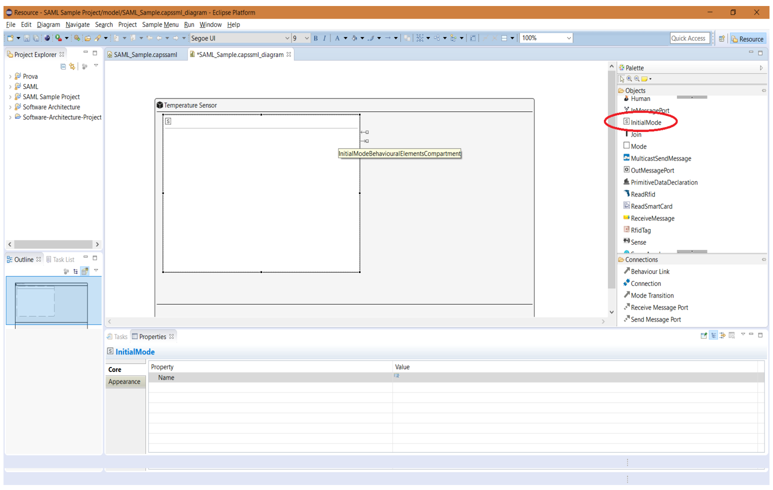


Figure 2.12: Creating the initial mode

- Now, think: what kind of actions the temperature sensors has to do when it starts? Well, it has to start sensing the temperature in the room every **X** seconds (we assume that **X=10 sec.**). So, we will need for three items from the **palette view**:
  - **StartTimer**;
  - **SenseTemperature**;
  - **TimerFired**;
  - **UnicastSendMessage**.

  In the figure 2.13 you can see the items placed.

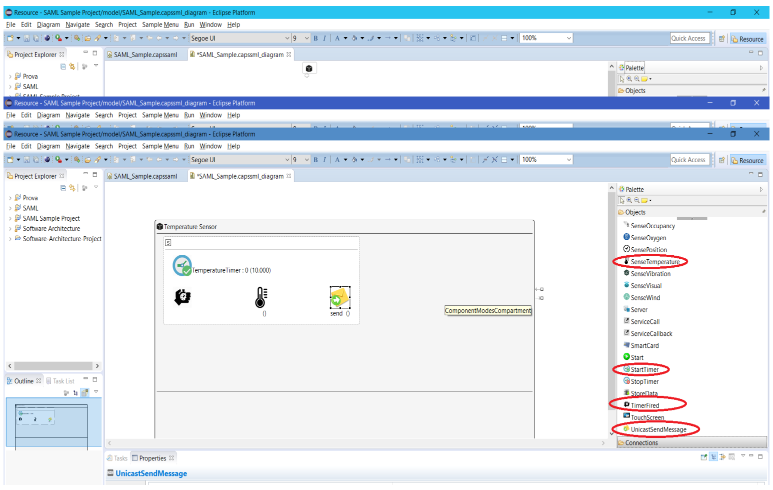


Figure 2.13: Creating the initial mode

- Give some properties to the timer: select the timer by clicking on it and, in the properties view (the section at the bottom of the screen), and set the following parameters:
  - **Cyclic** → **True**: the sensor has to give periodic values;
  - **Name** → **TemperatureTimer**;
  - **Period** → **10000** every **10 seconds** (the number expresses milliseconds);

  From the palette object section, take **PrimitiveDataDeclaration** and click on the component **TemperatureSensor**. In the **PrimitiveDataDeclaration view** (the same of the properties at the bottom of the screen) set the following parameters:
  - **Data Name** → **Temperature** the name of the data;
  - **Type** → **real** the type of the data;

- **Value** → **0.0** the default value.

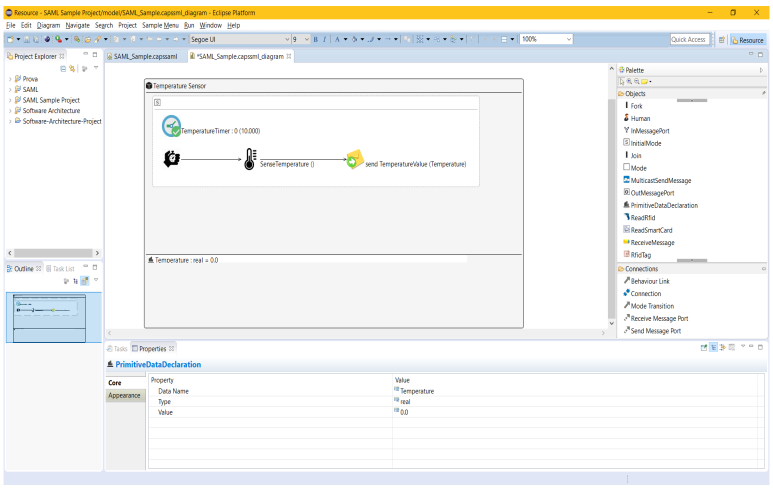
A PrimitiveDataDeclaration, let's you to specify the variables of the component (figure 2.14). Select the **TimerFired** item and in the properties section, find **Timer**, double click to the menu after it and select the name of the timer you have created. Select the **SenseTemperature** item and, in the properties view, set the following parameters.

- **Data Recipient** → **Double click and select the Temperature data** the variable where will be stored the temperature sensed;
- **Name** → **SenseTemperature**

Select the **message** item and set the following parameters:

- **Data** → **Temperature**;
- **Data Recipient** → **Select the Primitive Data Declaration temperature variable**;
- **Name** → **TemperatureValue**

• Now, in the connection subview of the palette select behaviour link and connect the **TimeFired** with the **SenseTemperature** (from the first to the second one). Do the same thing from the SenseTemperature to the message (figure 2.14);



Property	Value
Data Name	Temperature
Type	real
Value	0.0

Figure 2.14: The final result

• Let's add the server. Create an other component (name it server), add the initial mode, the **server** item, the **StoreData** item and the **ReceiveMessage** from the palette view (like you have just done for the temperature sensor component). Now, name the server item **Server** and set the following parameters in the properties view for the **ReceiveMessage** item:

- **Data Recipient** → **Primitive Data Declaration Temperature**;
- **Data Recipient Name** → **Temperature**;
- **Name** → **ReceiveTemperature**.

Use the **Behaviour link** arrow to link from the **ReceiveMessage** item to the **Server** item and from the **Server** to the **Store** item. From the palette items, click on **OutMessagePort** and click on the **Component Temperature Sensor.** This action will create a port from which the message can go to the other component. But the component needs an input port so, always from the palette view, take the **InMessagePort** item and click on the **Component Server**. Now connect them through the **Connection arrow** (**from the OutPort** to the **InPort**). Place an other **OutMessagePort** on the **TemperatureSensor Component**. You will need it later. Next step is to connect the messages to the port. So, take the **Send Message Port** link from the connections sub section of the palette and connect **from the SendMessage item to the OutMessagePort**. In the other component, connect **from the ReceiveMessage to the InputPort** with the **Receive Message Port** link. You can see the result and an highlighting of some of the items that you need in the figure 2.15.

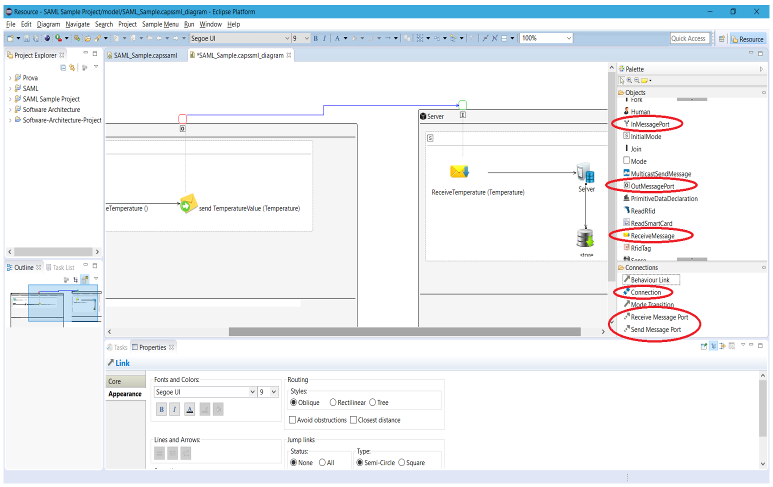


Figure 2.15: Result and items

The project is not finished. You have just modelled a scenario in which the temperature sensor saves his data into a server. The actuator of the window is missing. So, you are going to need other 2 components: the one for the **actuator** and the one for the **controller**, **something that manages the logic part of the system**.

- Create an other component, name it **Controller** and add the **Initial Mode**. Put a **ReceiveMessage** item inside the mode and set the following parameters:

  - **Name** → **ReceiveTemperature**;
  - **Data Recipient** → **Primitive Data Declaration Temperature**;
  - **Data Recipient Name** → **Temperature**.

  Place **2 Primitive Data Declaration** inside the **Controller Component** and set the following properties:

  - **Data Name** → **Close**;
  - **Type** → **boolean**;
  - **Value** → **false**.

  and:

  - **Data Name** → **Open**;
  - **Type** → **boolean**;
  - **Value** → **true**.

  Find the **Choice** item in the **Palette** and place it inside the **Initial Mode** of the **Controller** component. With a **Behaviour Link**, connect the **ReceiveTemperature** message to the **Choice** item. More over, place **2 UnicastSendMessage items** inside the mode. For one of them, set the following parameters:

  - **Data** → **Open**;
  - **Data Recipient** → **Primitive Data Declaration Open**;
  - **Name** → **sendOpen**.

  For the other one, set the following parameters:

  - **Data** → **Close**;
  - **Data Recipient** → **Primitive Data Declaration Close**;
  - **Name** → **sendClose**.

- With **2 Behaviour link**, connect from the **Choice** item to the **Send Messages** items. Set the following parameters to behaviour link connected to the **sendOpen** message:

  - **Condition** → $Temperature > 25$.

  For the other one, set the following parameters:

  - **Condition** → $Temperature < 18$.

  Then, place **2 OutMessagePort** and **1 InMessagePort**. Connect the **Receive Message (receiveTemperature)** with the **InMessagePort** through a **Receive Message Port connection** and the two **SendMessages** with the **OutPessagePort** through the **Send Message Port connection**. Connect the **InMessagePort** with the **OutMessagePort** of the **TemperatureSensor Component**.

- Now, the last component: the **Actuator**. Place a new component, name it **WindowActuator** and put the **InitialMode** inside it. Put two **ReceiveMessage** and an **Actuate** item inside the **InitialMode**. Set the following parameters for one of the **ReceiveMessage** item:

  - **Name** → **receiveOpen**;
  - **Data Recipient** → **Primitive Data Declaration Open**;
  - **Data Recipient Name** → **Open**.

  For the other one set:

  - **Name** → **receiveClose**;
  - **Data Recipient** → **Primitive Data Declaration Close**;
  - **Data Recipient Name** → **Close**.

  For the **Actuate** item set:

  - **Data** → **Actuate**;
  - **Name** → **WindowActuator**.

  Through Two **BehaviourLink** connect the **ReceiveMessages** items to the **Actuate** item. For the link coming from the **receiveOpen**, set the following parameters:

  - **Condition** → **Open**.

  For the other one set:

  - **Condition** → **Close**.

Take two **InMessagePort** item and place them on the **WindowActuator Component**. Then, connect through che **Connection**, the **OutMessagePorts** of the **Controller** component to the **InMessagePorts** of the **WindowActuator** component.

With the **ReceiveMessagePort** connection, link the **receiveOpen** with the **InMessagePort** that receives the message from the **sendOpen**. Do the same thing with the **ReceiveClose** but with the port that receives the message from the **sendClose**. The final result is shown in the figure below:

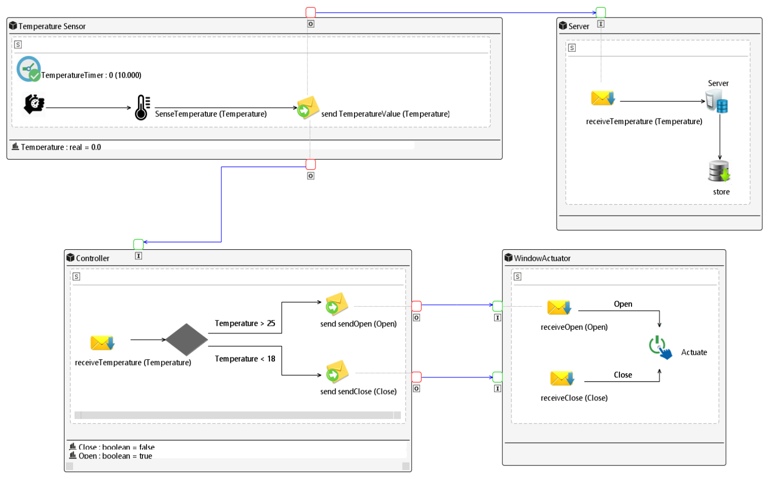


Figure 2.16: Final Result

2.2 Explanation

Let's do a brief explanation of what you have done so far.
You created **4 Components**:

1. **Temperature Sensor**;
2. **Server**;
3. **Controller**;
4. **WindowActuator**.

2.2.1 Temperature Sensor

This component has **4 items** inside it:

1. **StartTimer**;

2. **TimerFired**;
3. **SenseTemperature**;
4. **UnicastSendMessage**.

The timer is connected to the **TimerFired** (thanks to the properties set before). Every **10 seconds (10000 milliseconds)**, the component senses the temperature and saves it in a variable called **Temperature (real type)**. So, the component sends a message containing this variable (thanks again to the properties set before) to **2 output ports**. **The components exachange messages only through ports**.

These items are deployed inside the **Initial Mode** that is the **Starting Mode** of the sensor. It means that, as soon as the component starts working, it will do all the actions that have been just described.

2.2.2 Server

This component has **3 items** inside it:

1. **ReceiveMessage**;
2. **Server**;
3. **StoreData**.

The **Server** component receives messages from the **Temperature Sensor** component through a port. The received messages contain the **Temperature value sensed** and they are stored in a DB. Also in this case, the items are deployed inside the **Initial Mode**.

2.2.3 Controller

The **Controller** component represents a software that uses logic on order to process data that come from the **Temperature Sensor** component. It contains:

1. **ReceiveMessage**;
2. **Choice**;
3. **2 UnicastSendMessage**.

The **Controller** component receives the temperature value stored in the variable **Temperature**. The **Choice** let's the component to perform an action based on the **temperature value** (note that the conditions are annotated con the **Behaviour Links**). So, a message containing the **boolean variable Open** is sent if *temperature* > 25. Otherwise, the message containing the **boolean variable Close** is sent.

2.2.4 Actuator

The **Actuator** component is used in order to open/close the window. It contains, inside his **Initial Mode**, the following items:

1. **2 ReceiveMessage**;
2. **Actuate**.

The behaviour of this component is very simple: when it receives the **Close**, it closes the window. When it receives the **Open**, it opens the window.

Chapter 3

HWML

- Now we want to model some hardware. Assume the same scenario we have done so far and let's model the hardware for the **Temperature sensor** (the procedure for the other ones is the same, just change properties and so on). First of all, go to the **project explorer view** on the left (the view where you can see all your folders and files) and right click on the model folder (just like you have done for **SAML**). Click on New then Other and select **CAPShwml Model** (figure 3.1);

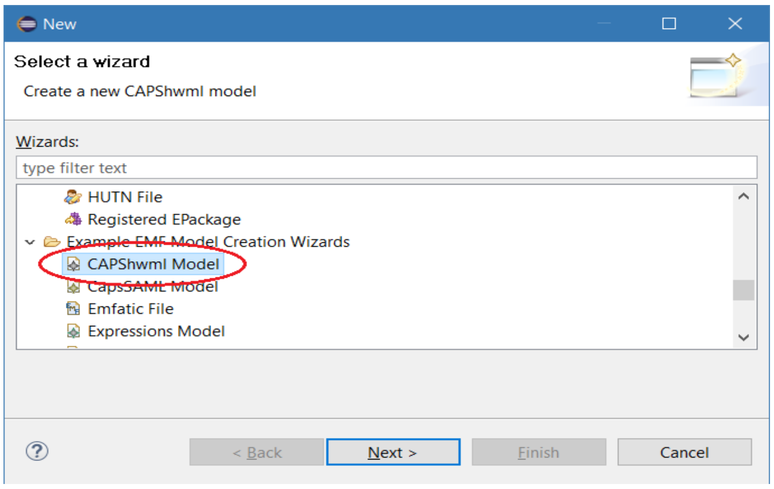
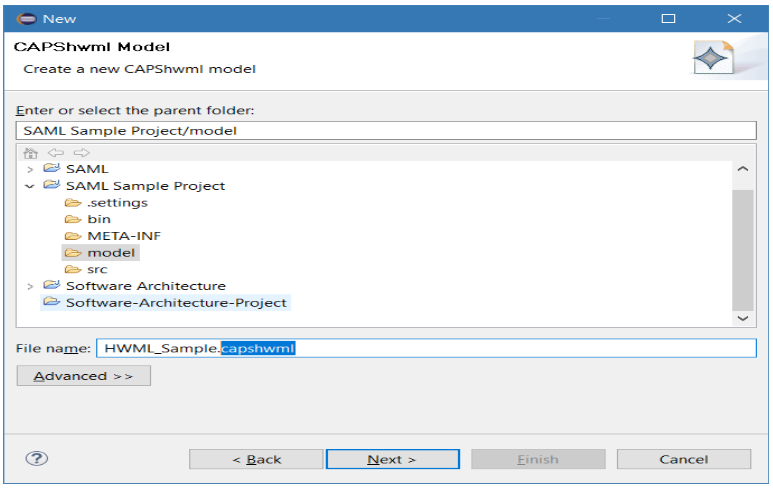


Figure 3.1: Starting HWML

- Click on next, select a name (that finishes with **.capshwml** figure 3.2), next again and choose **node specification**;
- In the explorer view, right click on the file that has been just created and select **initialize filesystem_diagram diagram file**. Choose a



New

CAPShwml Model

Create a new CAPShwml model

Enter or select the parent folder:

SAML Sample Project/model

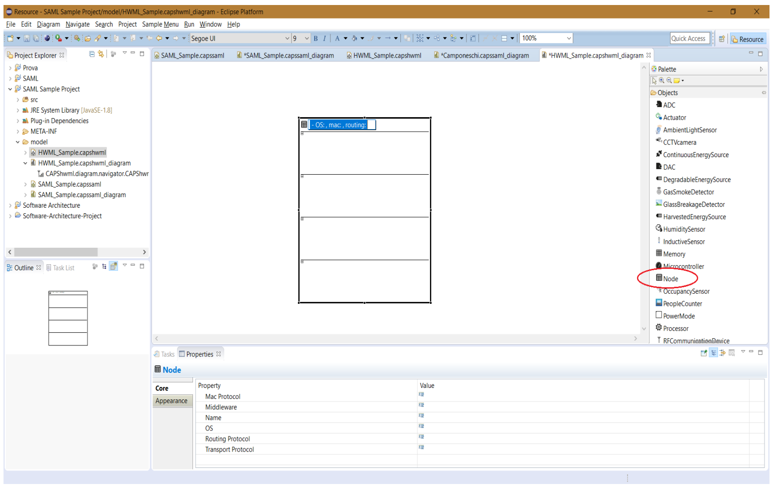
- SAML
- SAML Sample Project
  - .settings
  - bin
  - META-INF
  - model
  - src
- Software Architecture
- Software-Architecture-Project

File name: HWML_Sample.capshwml

Figure 3.2: Naming an HWML file

name and click on finish;

- Now, from the **objects** view on the right, select **node** and create it on the screen (figure 3.3);



Property	Value
Mac Protocol	
Middleware	
Name	
OS	
Routing Protocol	
Transport Protocol	

Figure 3.3: Creating a deployment node

- In the properties view, set the following parameters:
  - **Mac protocol** → **ZIGBEE**: a standard communication protocol;
  - **Name** → **Temperature Sensor**;

- **OS** → **TinyOS**: embedded, component-based operating system and platform for low-power wireless devices;
- **Routing protocol** → **GEAR**: **Geographical Energy Aware Routing** protocol for **wireless sensor network**;

- Let's add some other objects to this node: a **MicroController** to place inside the node, a **Processor** to place inside **the MicroController** and a **Volatile Memory** to place inside **the MicroController**. Set the following parameters (in the properties view) **Processor** object:

  - **Cpi** → **1.0**: **clocks per instruction**;
  - **Frequency** → **120**: frequency of the CPU expressed in MHz;
  - **Name** → **Atmel Atmega328**.

  Now, for the volatile memory:

  - **Name** → **RAM**;
  - **Size** → **2**: the size of the memory expressed in KB;

- From the object view, select **ContinousEnergySource**, place it on the node and call it **electricity**: it means that the node is powered by a continous energy source that is **electricity**. In the end, add also the TemperatureSensor object. The final result is showed in the figure 3.4;

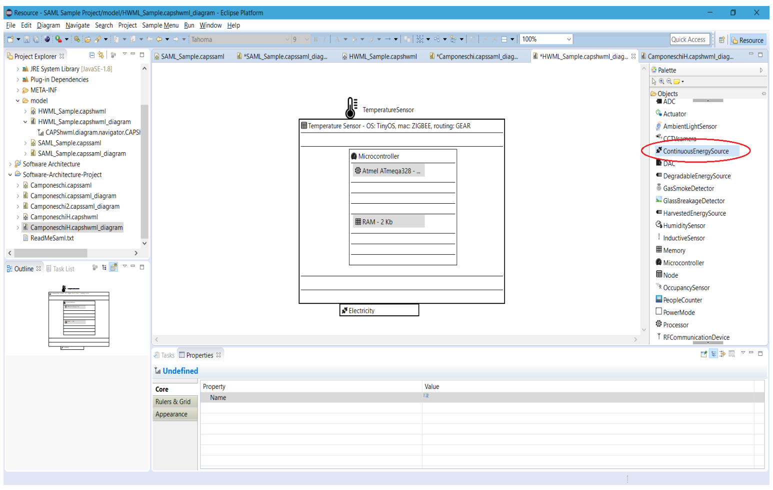


Figure 3.4: Final result

Chapter 4

SPML

- CAPS regards also the environment surrounding the sensors and so on. This part is called **SPML**. You need **Sweet Home 3D** in order to model some building and to place some sensors in it. Once you have done it, you will need a **jar file**, called **CAPSAdapter**. Open it and you will get something like the figure 4.1;

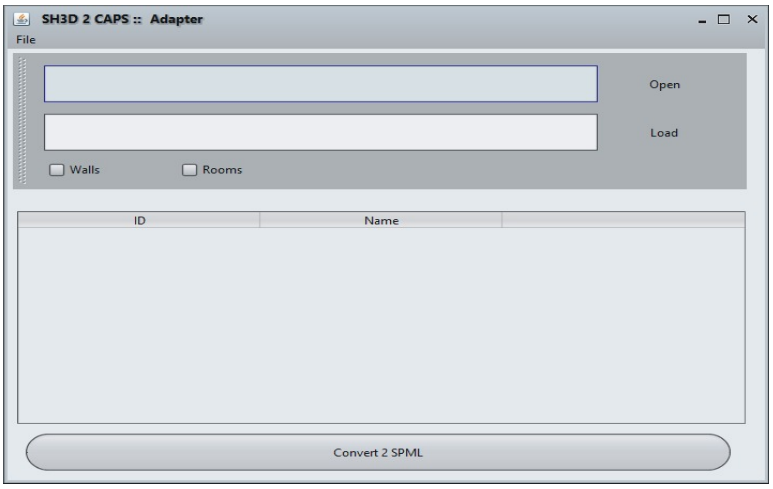
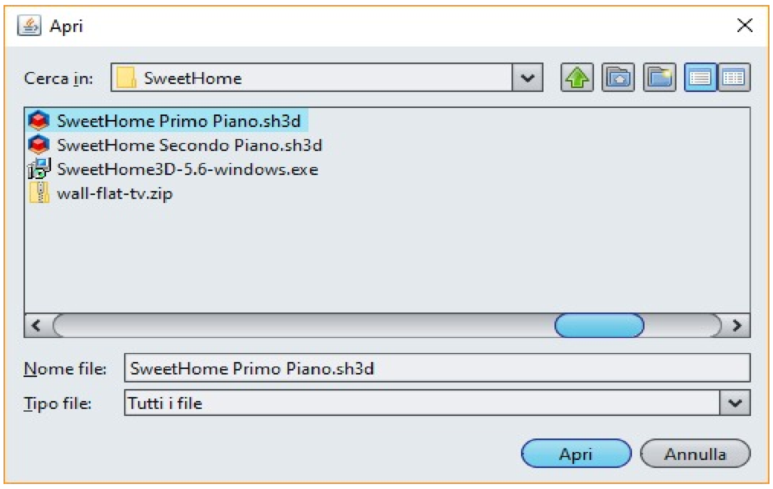


Figure 4.1: CAPS adapter

- Click on Open and select a file which extension is **.sh3d** (so, you have to select some file made with **SweetHome 3D** figure 4.2);

- Open it, tick the **checkbox wall and the checkbox rooms** and click on load (figure 4.3 and figure 4.4);

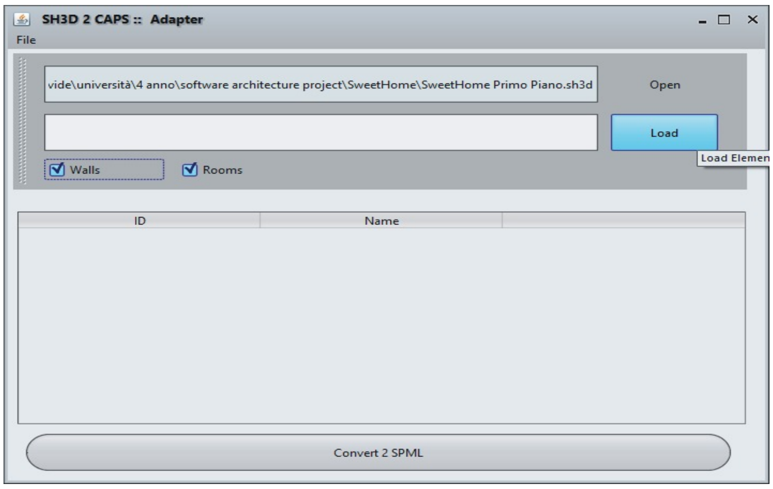


SweetHome Primo Piano.sh3d
SweetHome Secondo Piano.sh3d
SweetHome3D-5.6-windows.exe
wall-flat-tv.zip

Nome file: SweetHome Primo Piano.sh3d
Tipo file: Tutti i file

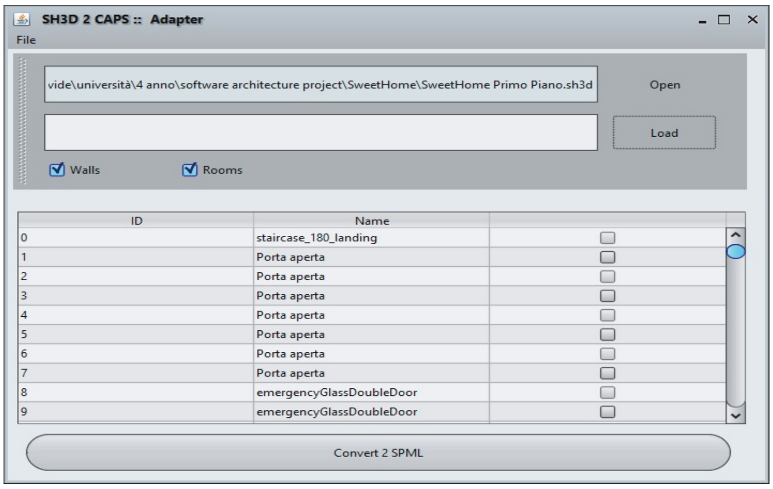
Figure 4.2: Selecting a file

- Tick all the new checkboxes (4.5) and click on **Convert 2 SPML**. When you choose the new file name, **remember to set its extension: .xmi**
- Take the file **.xmi** you have just created and place it in the package **univaq.spml/model**. That's it, you finished



vide\università\4 anno\software architecture project\SweetHome\SweetHome Primo Piano.sh3d

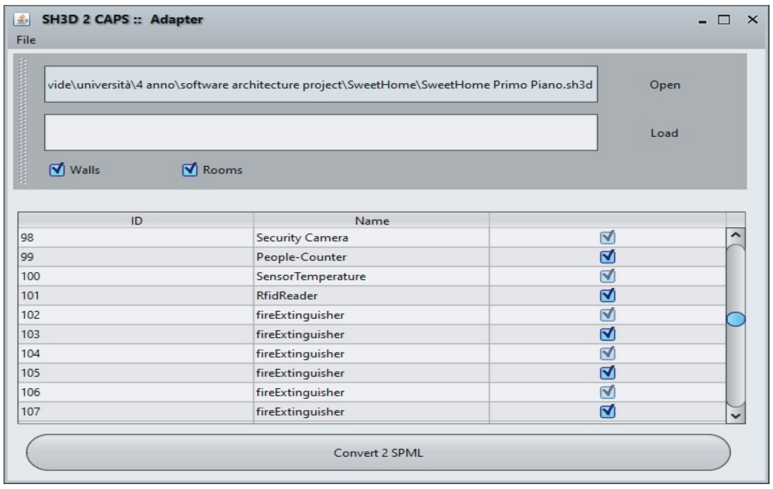
Figure 4.3: Ticking the checkboxes and clicking load



vide\università\4 anno\software architecture project\SweetHome\SweetHome Primo Piano.sh3d

ID	Name	
0	staircase_180_landing	☐
1	Porta aperta	☐
2	Porta aperta	☐
3	Porta aperta	☐
4	Porta aperta	☐
5	Porta aperta	☐
6	Porta aperta	☐
7	Porta aperta	☐
8	emergencyGlassDoubleDoor	☐
9	emergencyGlassDoubleDoor	☐

Figure 4.4: Result of the previous operations



vide\università\4 anno\software architecture project\SweetHome\SweetHome Primo Piano.sh3d

✓ Walls ✓ Rooms

ID	Name	
98	Security Camera	✓
99	People-Counter	✓
100	SensorTemperature	✓
101	RfidReader	✓
102	fireExtinguisher	✓
103	fireExtinguisher	✓
104	fireExtinguisher	✓
105	fireExtinguisher	✓
106	fireExtinguisher	✓
107	fireExtinguisher	✓

Figure 4.5: All the checkboxes ticked

Appendix A

Appendix

1. **StartTimer**: an action that starts an internal timer in the application. The expiration of a started timer is represented by a special event called TimerFired. When starting a timer, architects can set three parameters:

    (a) the delay (in milliseconds) that must occur **before the first activation of the timer**;

    (b) **the cyclic nature of the timer** (that is, whether it must be periodic or not);

    (c) the period of the timer (in milliseconds), if it is a cyclic one.

2. **StopTimer**: this action stops a previously started timer;

3. **Fork**: a specialization of **ControlAction** representing the **classical fork operation on a (control) flow graph.** Basically, this action is used to explicitly split the incoming behavioural flow into a set of parallel flows;

4. **Join**: a specialization of ControlAction representing the classical join operation on a (control) flow graph. Intuitively, it performs the inverse operation of a fork operation, i.e., it merges incoming behavioural flows and syncs them into a common outgoing one;

5. **Choice**: a specialization of ControlAction representing the classical choice operation on a (control) flow graph. This action is used to split the control flow into one or more branches. Depending on the value of the conditions in the outgoing behavioural links, one and only one control flow is executed after the choice control action;

6. **ReceiveMessage**: an event triggered when the component receives a message to one of its input message ports. The **dataRecipient** reference points to the application data declaration which will hold

the contents of the received message. TimerFired. An event triggered every time a previously started timer expires. The previously started timer is defined by the timer reference in the FiredTimer metaclass;

7. **Mode**: a mode is a specific status of the component. Examples of modes can be: sleeping mode, energy saving mode, etc. It is important to note that modes are defined at the application layer in SAML, thus they can, but are not forced to, be related to the energy-related modes of the WSN node they are running on. At any given time, one and only one mode can be active in a component. The component reacts only to those events which are defined within its currently active mode. Mode is the only metaclass that is not included in the specialization sub-tree of BehaviouralElement; we made this choice for preventing architects to define nested modes in SAML models, thus keeping the SAML language simple to some extent. A component can switch from a mode to another by means of the previously defined Link construct. Mode transitions occur by passing from a special kind of action called ExitMode to a special kind of event called EnterMode (the concepts of exit and enter mode will be described later in this section). In this way, actions and events can be linked to modes entry and exit points, creating a continuous behavioural flow among modes;

8. **EnterMode**: a specialization of the Event metaclass; it represents a special kind of event which is triggered when the behavioural flow enters a specific mode. Actions can be linked to this type of events, they will be executed immediately after the entered mode becomes active;

9. **ExitMode**: a specialization of the Action metaclass; it represents the action of exiting a specific mode, and thus entering another mode within the component containing them (see the targetMode reference of ExitMode in Figure 2. InitialMode. A special kind of mode. Intuitively, an initial mode is the first mode which is active when the component starts up. Clearly, each component can contain one and only one initial mode;

10. **Actuate**: an action that activates an actuator; optionally, an expression can be used to pass a parameter to the actuator. Also, an actuate action can optionally refer to an application data declaration to which the result of the actuate action can be stored. This action can be used, for example, to model the action of turning off a light or activating the heating system of a house;

11. **SendMessage**: An abstract metaclass representing the action of sending a message via a specific output message port (for the sake of clarity we did not show the outMessagePort reference from SendMessage to

OutMessagePort in Figure 2). Optionally, the contents of the message can be specified by passing an expression as parameter. Available types of message include: broadcast message, multicast message, and unicast message; they differ in the number of target physical nodes of the WSN actually receiving the message. By target physical node we mean all the nodes containing the components declaring an InMessagePort that is connected to the OutMessagePort referenced by the send message action;

12. **BroadcastSendMessage**: a special kind of SendMessage action in which the message is sent to every node containing the target component;

13. **MulticastSendMessage**: a special kind of SendMessage action in which the message is sent to a specific set of nodes containing the target component. Those nodes are represented by a list of receivers, identified by the name they will have in the final deployment of the WSN;

14. **UnicastSendMessage**: a special kind of SendMessage action in which the message is sent to a single node containing the target component. The receiver node is identified by its name in the final deployment of the WSN.