

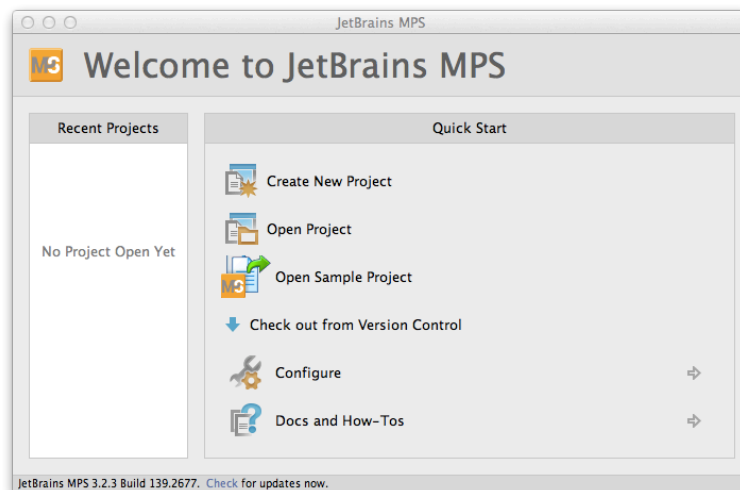
EJCP'15 – MPS Lab session

- Contact: Sébastien Mosser (mosser@i3s.unice.fr)

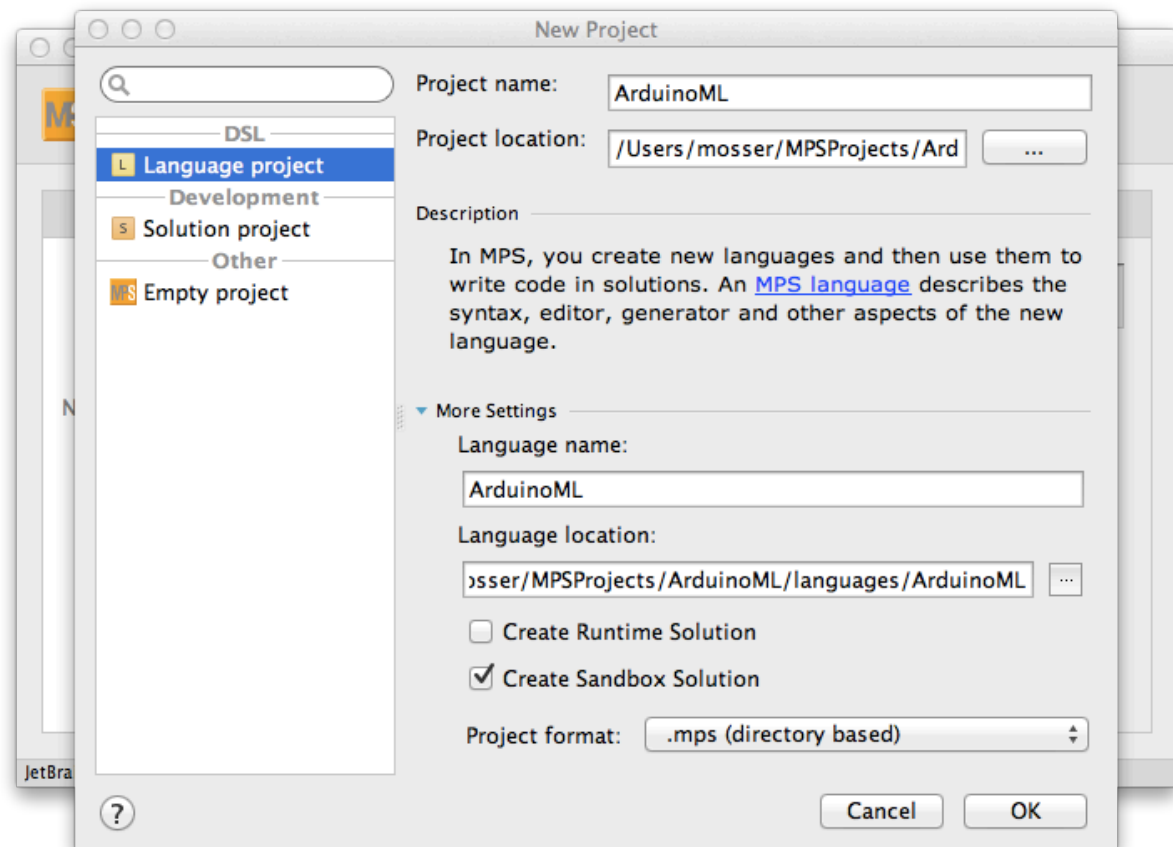
The objective of this lab session is to provide an overview of MPS in the context of ArduinoML. We first create an empty project, and then design the meta-model of structural elements involved in Arduino apps. We then change the concrete syntax associated to these elements, and finally implement a code generator to fill the “setup” function expected by the Arduino board. We will focus on the behavioral part of ArduinoML this afternoon.

Step #0: Setting up the environment

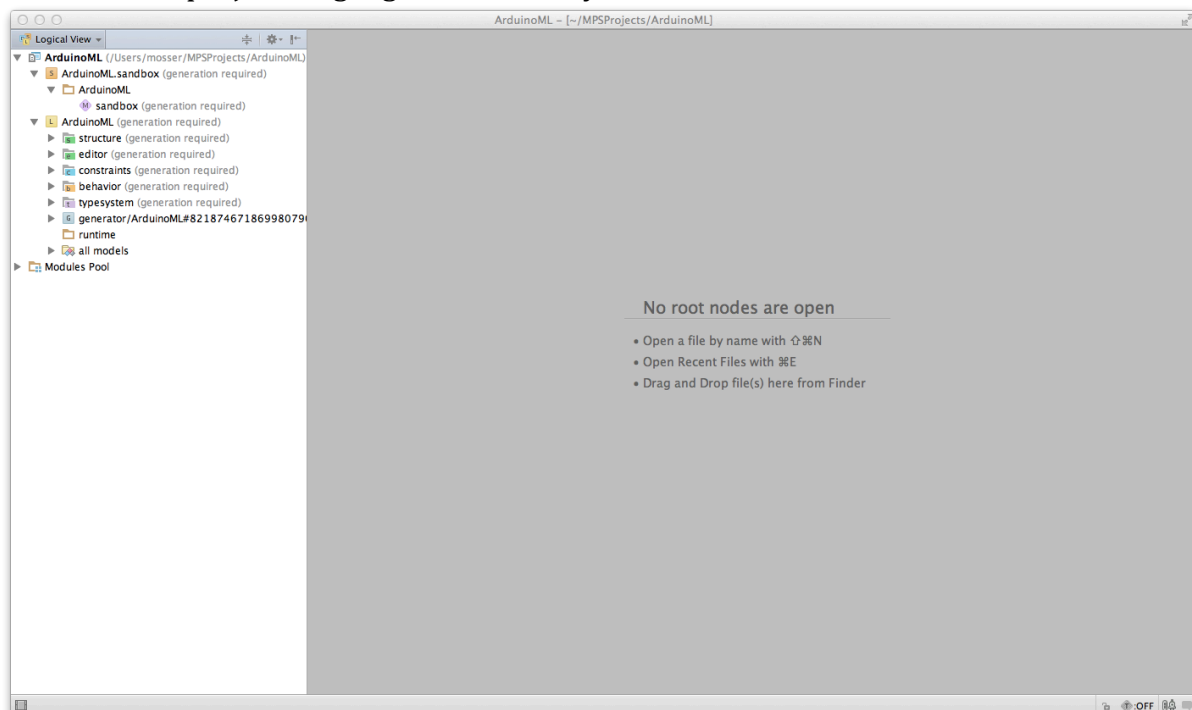
Start by creating a new project (“Create New Project”) to store our language.



The language will be named “ArduinoML”, so it is a good name for the project and the language. Tick the “Create Sandbox solution” to automatically instantiate a project dedicated for playing with your language.

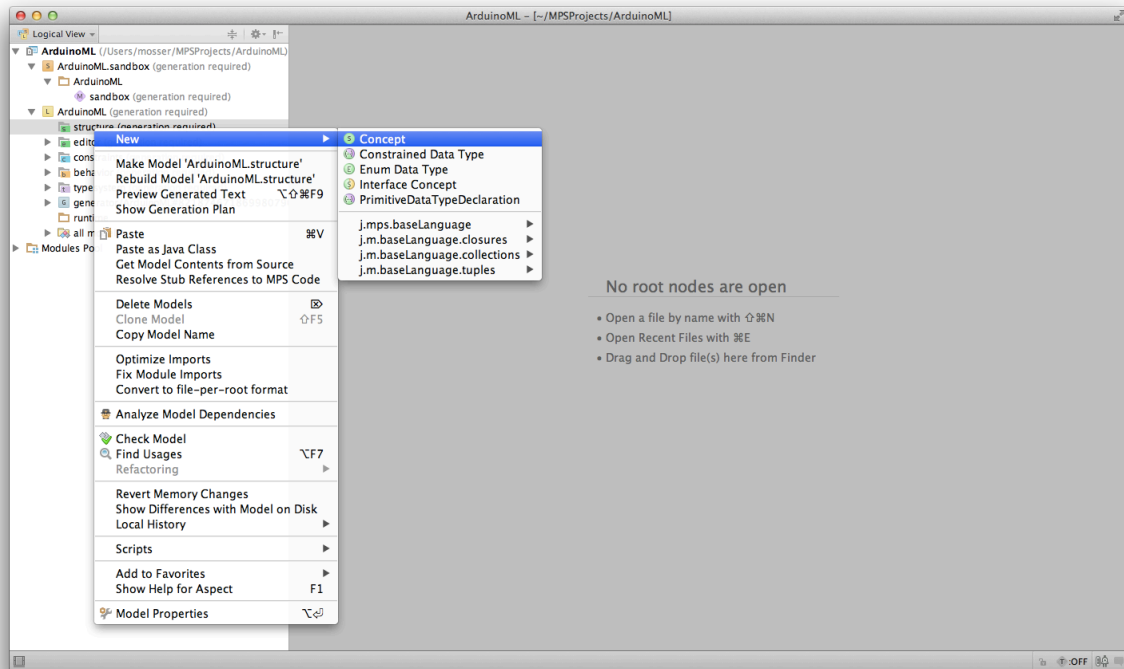


After indexing, the window will contain two projects. An “S”, meaning that this project is a “Solution” project, identifies the “ArduinoML.sandbox” element. It is linked to the “ArduinoML” project language, identified by an “L”.

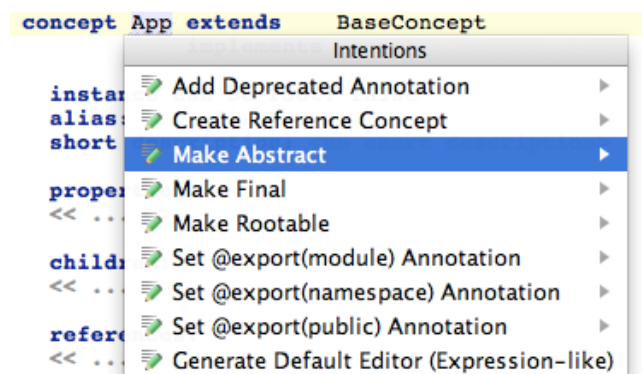


Step #1: Creating Structural concepts for ArduinoML

The meta-model contains, for the structural description of ArduinoML applications, several concepts: Actuators, Sensors (abstracted as Bricks). These bricks are contained by an App, acting as the root of our system. Create these concepts in the “Structure” part of the language.



The Brick concept



Create a new concept, and make it abstract (Alt-enter on the name to summon the contextual menu). All bricks will have a name property, declared as a string. As meta-classes often define a “name” property, we reuse the INamedConcept interface provided by MPS (Ctrl-space for code completion). We add an integer property to model the pin where the brick is plugged into the Arduino board.

```

abstract concept Brick extends BaseConcept
                        implements INamedConcept

instance can be root: false
alias: <no alias>
short description: <no short description>

properties:
  pin : integer

children:
  << ... >>

references:
  << ... >>

```

The Sensor and Actuator concepts

These two concepts simply extend the Brick concept created in the previous paragraph. Use Ctrl-Enter for code completion.

```

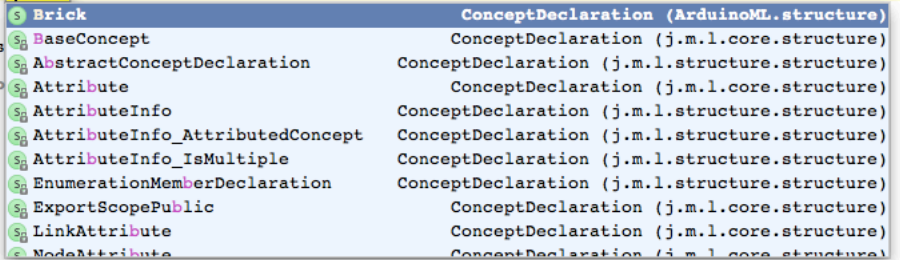
concept Actuator extends
implements
instance can be root: false
alias: <no alias>
short description: <no short description>

properties:
  << ... >>

children:
  << ... >>

references:
  << ... >>

```



The App concept

The App concept will contain the bricks used in the application, as children. The cardinality of this containment relationship is 1..n. An instance of App will be the root of our modeling environment. As a consequence, its property “instance can be root” is set to true.

```

concept App extends BaseConcept
implements <none>

instance can be root: true
alias: <no alias>
short description: <no short description>

properties:
  << ... >>

children:
  bricks : Brick[1..n]

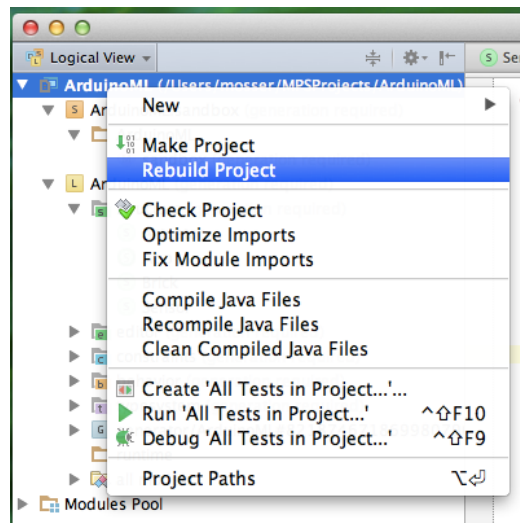
references:
  << ... >>

```

Step #2: Editing Models

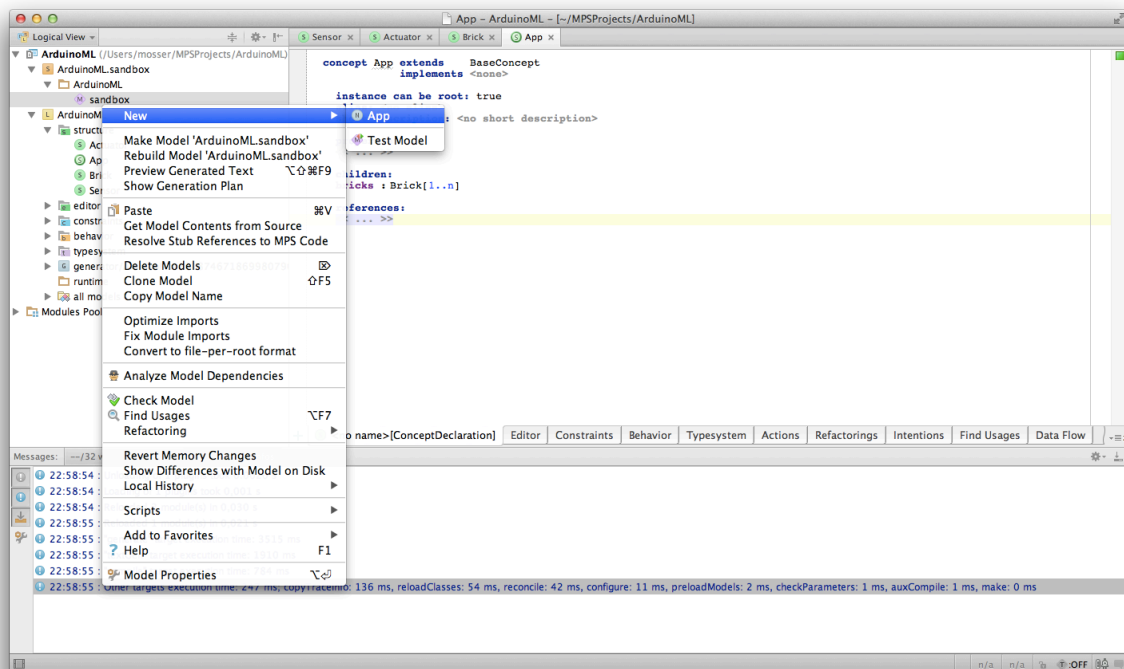
Building the system

MPS is all about “Projectional Edition”. It is time to define “projections” for our concepts. First, we will use the default editor. The preliminary step is to rebuild the project, to compile the concept modeled in the previous part, and make the concepts available in the sandbox.



Creating an App for the “Big Red Button” application

After completion of the “Rebuild” process, you can instantiate an App in the sandbox.



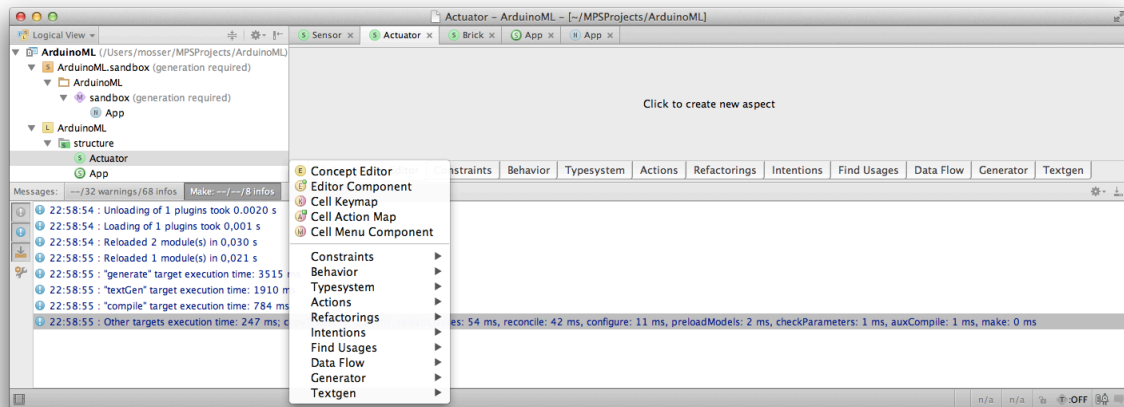
The model can be edited using code completion, using Ctrl-Enter to summon the contextual menu. Model a button (sensor) bound to pin 8, and a red led (actuator) bound to pin 12.

```
app {
  bricks :
    actuator red_led {
      pin : 12
    }
    sensor button {
      pin : 8
    }
}
```

Providing a new syntax (a new projection)

Actually, we simply filled an abstract syntax tree, through the default projection automatically provided by MPS. If you want to change the concrete syntax of your DSL, the key point is to provide another projection. The models will be automatically updated to reflect this new projection (as you are not manipulating text but projection of the AST).

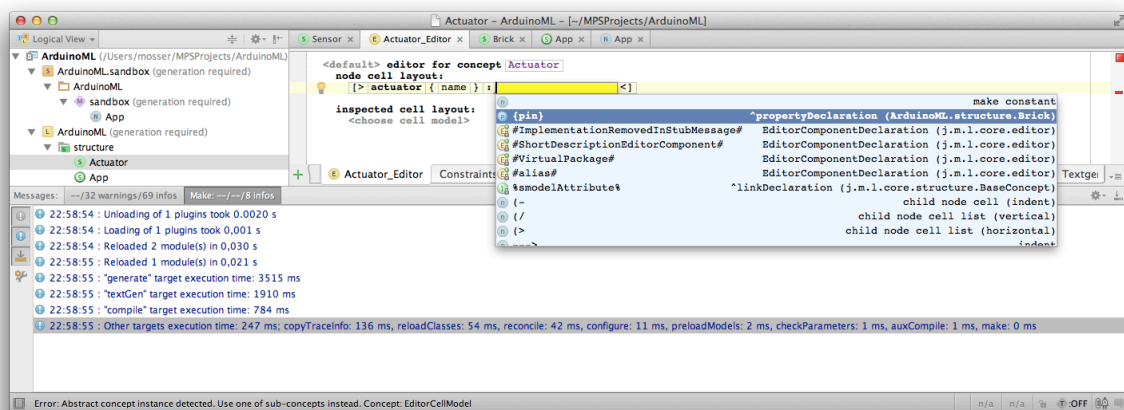
Click on the “Actuator” concept; select the “Editor” tab (bottom-left of the edition panel); click on the “+” button and select “Concept editor”.



We want to write things like “actuator red_led: 12”. This sentence is a horizontal collection of 4 elements:

“actuator” %NAME% “:” %PIN%

The syntax defined by MPS to model such a thing is to use [> ... <] to define a horizontal collection. Constants like “actuator” and “:” are defined by typing directly the expected token, and variable contents are available through code completion (Ctrl-space).



Do something similar for the Sensor concept. Rebuild the project, and open the previously created App in the Sandbox.

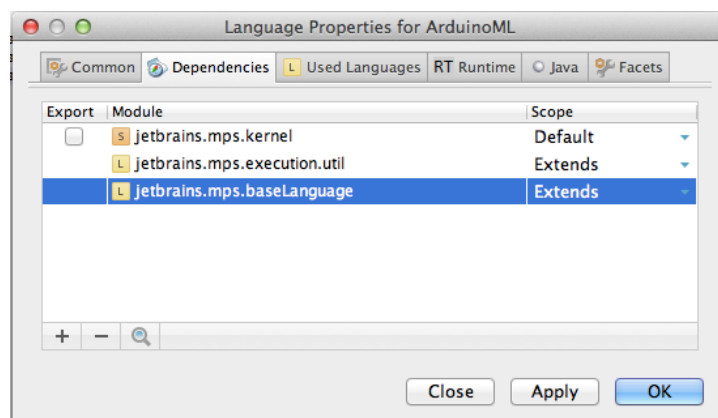
```
app {  
  
  bricks :  
    actuator red_led : 12  
    sensor button : 8  
}
```

Step #3: Generating code

MPS brings a template processor engine to generate code. Its main drawback is that this engine only works between MPS-defined languages. In this lab session, we will use a “quick and dirty” trick, by generating Java that will print the expected Arduino code. The “right” thing to do is (i) to model the Wiring language used as target or (ii) model a “text” language and uses it as a target. The process follows a map/reduce mechanism (if you are familiar with it).

Language composition: defining dependencies

Right click on the “ArduinoML” language, select “Module Properties” and the “Dependencies”. Our language needs the MPS kernel for code generation purpose. We also extend the “execution.utils” and “baseLanguage” languages for expressiveness purpose.



Making the App concept executable

Edit the “App” concept, in order to implement the IMainClass interface (provided by the newly declared dependencies).

```
concept App extends BaseConcept
            implements IMainClass

instance can be root: true
alias: <no alias>
short description: <no short description>

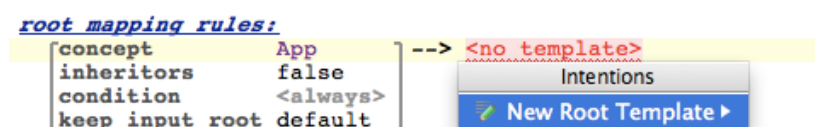
properties:
<< ... >>

children:
bricks : Brick[1..n]

references:
<< ... >>
```

Create the Root Mapping rule

Press “Enter” under the “Root mapping rule” element in the main@generator template definition. Use Ctrl-space to select the “App” concept as input of the rule. We will map an App to a Class. On the left part of the rule, use Alt-enter to summon the contextual menu and select “New root Template”.



You can now edit the target template through a projectionnal editor dedicated to Java.

```

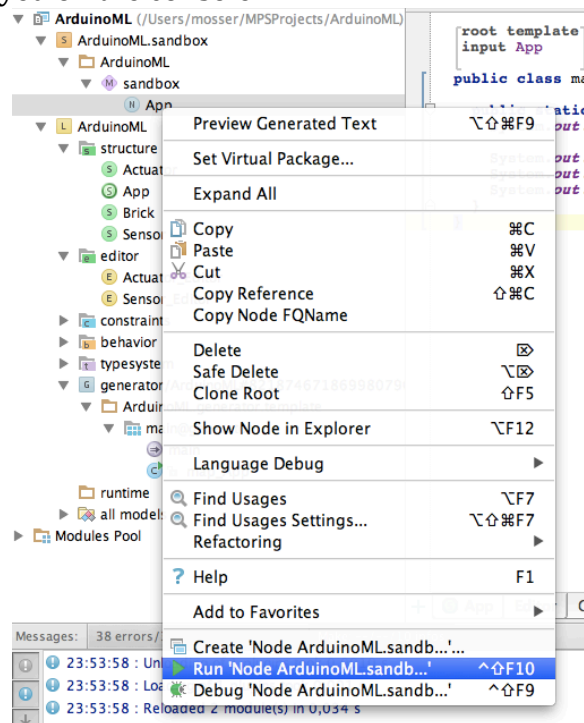
[ root template
  input App
]
public class map_App {

  public static void main(string[] args) {
    System.out.println("// Code Generated by ArduinoML");

    System.out.println("void setup() {");
    System.out.println("  // Here comes brick declaration");
    System.out.println("}");
  }
}

```

Rebuild the projects, and then right-click on the previously modeled App. The result of the execution is displayed on the console.



Concretely generating pin declaration instructions with Reduction rules

Sensor and Actuators will use different generators. Such elements are called “reduction rules”. We start by defining a reduction rule for Sensor.

Create a new reduction rule following the same steps than the root mapping one, excepting that you’ll define it under “Reduction rules”.

```

reduction rules:
[ concept Sensor
  inheritors false
  condition <always> ] --> reduce_Sensor

```

The reduction rule contains a BlockStatement (Alt-enter for code completion)

```

template reduce_Sensor
input Sensor

parameters
<< ... >>

content node:
[ blockS
  Replace with instance of BlockStatement concept
]

```


The code to be generated is the following: “pinMode(“ + pinNumber + “,INPUT);”. Thus, we put such a statement inside the code block. As the template engine will process this code, we wrap it inside a “template fragment” (alt-enter at the end of the statement). The template fragment models the text that will be returned from the reduction rule to the caller one. Rebuild the project of necessary, to remove an error if MPS is not able to recognize the template fragment.

```
template reduce_Sensor
input Sensor

parameters
<< ... >>

content node:
{
  <TF> [System.out.println(" pinMode(" + pinNumber + ", INPUT);"); ] TF>
}
```

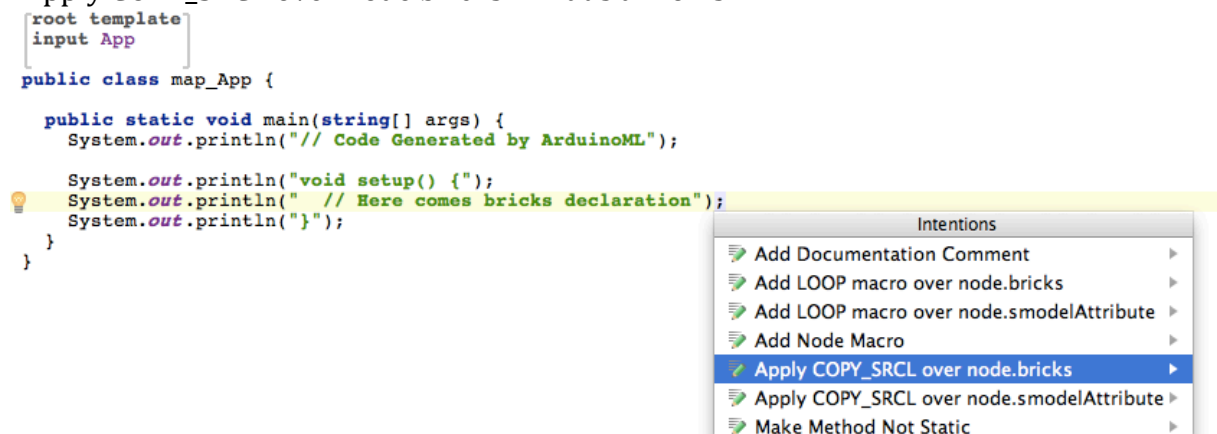
We must now replace “pinNumber” by the real one. Replace “pinNumber” by an integer (e.g., 0), and on the element, use Alt-enter to summon the contextual menu. Select “Add property macro: node.pin”



Do something very similar for Actuators, replacing INPUT by OUTPUT.

Calling the reduction rules from the main mapping one

Actually, you just want to replace the static declaration stating that “Here comes the bricks definition” by the real ones. Use alt-Enter at the end of this statement, and select “Apply COPY_SRCL over node.bricks”. That’s all folks



Rebuild the project, and re-run the App model. Congratulations!