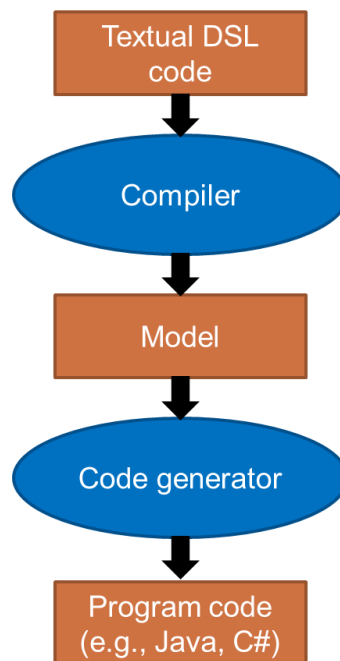# Practice 3 – Xtext and Xtend

Dr. Balázs Simon, 2024

1   Introduction

This guideline shows how to create a compiler, a code generator and IDE support for a textual DSL.

The processing of a textual DSL consist of the following steps:


The compiler builds a model from the textual DSL code. In order to be able to do this, the elements of the model and their relationships must be specified. This is the purpose of the meta-model. When defining the compiler, the mapping of the DSL source code elements to the meta-model elements must also be specified. If this mapping is accurate enough, the model can be built automatically by the compiler. By traversing the model, a code generator can produce some useful output code.

Before starting this tutorial, prepare the projects according to the guidelines of **Practice 3 – Preparing the projects**.

2 Programing language

The goal is to create a compiler and tools support for a simple state machine description language.

Take the following UML state machine as an example, which describes the behavior of a simple lamp with a switch:
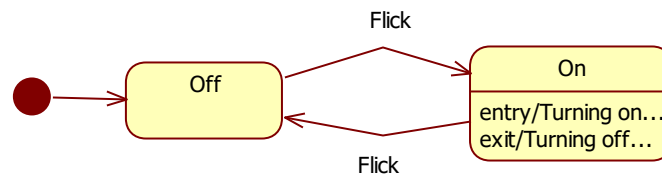
The description of this state machine is the following in our textual DSL:

```
machine Lamp
{
  initial state Off
  {
    event Flick
    {
      jump On;
    }
  }
  state On
  {
    entry
    {
      print "Turning on...";
    }
    event Flick
    {
      jump Off;
    }
    exit
    {
      print "Turning off...";
    }
  }
}
```

A state machine is defined by the **machine** keyword. The state machine has states (**state** keyword), the starting state is specified by the **initial state** keywords. There must be exactly one initial state in the state machine.

Inside a state, events can occur, and the body of the event handler specifies what should happen when the specified **event** arrives. A state may also have an optional **entry** and an optional **exit** event handler. The entry event handler is executed when its containing state is activated, the exit event handler is executed when its containing state is left.

Inside an event handler's body two statements can appear. The **jump** statement changes the state of the state machine, but only after the whole body of the event handler is finished executing. The print statement writes a character string to the output. An event handler can contain at most one jump statement.
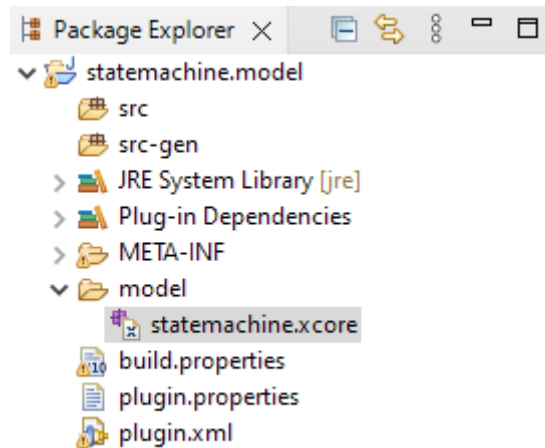
3   Meta-model

The following meta-model describes our models of state machines:
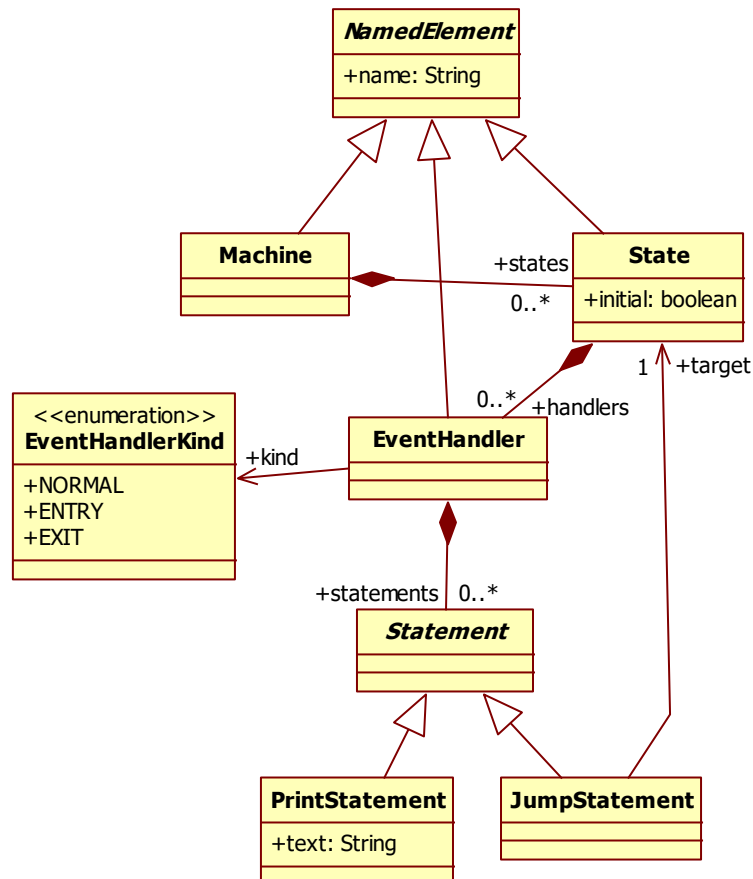
4   Defining the meta-model: Xcore

Inside the **statemachine.model** project under the **model** folder, there should be a file called **statemachine.xcore**:

Inside this file, we have to define the textual description of our meta-model in Xcore syntax. The contents of this file should be according to chapter 3:

```
package statemachine.model

abstract class NamedElement
{
    String name
}

class Machine extends NamedElement
{
    contains State[] states
}

class State extends NamedElement
{
    boolean initial
    contains EventHandler[] handlers
}

enum EventHandlerKind
{
    NORMAL,
    ENTRY,
    EXIT
}

class EventHandler extends NamedElement
{
    EventHandlerKind kind
    contains Statement[] statements
}

abstract class Statement
{
}
```

```
class PrintStatement extends Statement
{
    String text
}

class JumpStatement extends Statement
{
    refers State target
}
```

This is exactly the UML class diagram shown before: the diagram's meaning is easily translated into Xcore.

From the **statemachine.xcore** file, the Xcore tool generates Java code that contains the implementation of the meta-model using the EMF (Eclipse Modeling Framework) framework. This meta-model can be instantiated automatically as a model by the compiler, which is specified in the next chapter.

5   Defining the compiler: Xtext

The compiler consists of two parts: a lexer and a parser. The lexer produces the tokens, the parser produces the syntax tree. In the case of Xtext, the two can be described together in the same file. Keywords can be build inline into the grammar, and Xtext will produce the appropriate lexer for them. Also, Xtext comes with some commonly used built-in tokens (ID, INT, STRING) and C-style comments. The ID token must be used to define a name which is resolvable during name analysis. Xtext is therefore a bit more clever than just a simple parser generator (e.g., ANTLR4), since it can automatically perform name analysis: whenever we need to refer to a name, we simply include the referenced model element in square brackets in the language description.

Based on these considerations, the Xtext description of the grammar of our state machine language is the following (**MachineDsl.xtext** file inside the **statemachine.dsl** project):

```
grammar statemachine.dsl.MachineDsl with org.eclipse.xtext.common.Terminals

import "statemachine.model"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Machine: 'machine' name=ID '{' states+=State* '}';

State: initial?='initial'? 'state' name=ID '{' handlers+=EventHandler* '}';

enum NormalEventHandlerKind returns EventHandlerKind: NORMAL='event';
enum EntryEventHandlerKind returns EventHandlerKind: ENTRY='entry';
enum ExitEventHandlerKind returns EventHandlerKind: EXIT='exit';

EventHandler:
    kind=(NormalEventHandlerKind|EntryEventHandlerKind|ExitEventHandlerKind)
    name=ID? '{' statements+=Statement* '}'
;

Statement: PrintStatement | JumpStatement;

PrintStatement: 'print' text=STRING ';';

JumpStatement returns JumpStatement: 'jump' target=[State] ';';
```
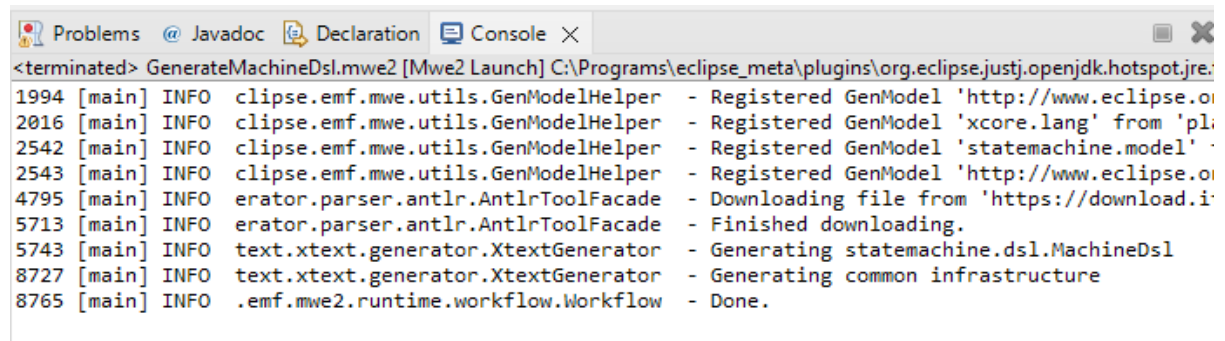
It can be seen that if the meta-model is well designed, the grammar rules can be easily specified. It is important to note that the names of the grammar rules in Xtext must be exactly the same as the names of the classes defined in the meta-model. Also, the names of the properties on the right side of the rules must match the names of the properties defined in the meta-model.

When you are ready with the grammar above, click on **GenerateMachineDsl.mwe2** with the right mouse button, and select **Run As > MWE2 Workflow**. In this step Xtext generates the Java classes that implement the compiler. This step must always be re-executed whenever you modify the Xtext file.

If everything was done correctly, the process should end without any errors:


6 Checking out the IDE plugin

In the following sections we will examine the generated files in detail, but before we do that, let's check out the IDE plugin for our state machine language generated by Xtext.

Click on the project **statemachine.dsl** with the right mouse button and select a **Run As > Eclipse Application**! This starts another instance of Eclipse called the **Runtime Eclipse**, which contains all the plugins generated by Xtext pre-installed.

In this new Eclipse, select **File > New > Project…** and create a new general project:
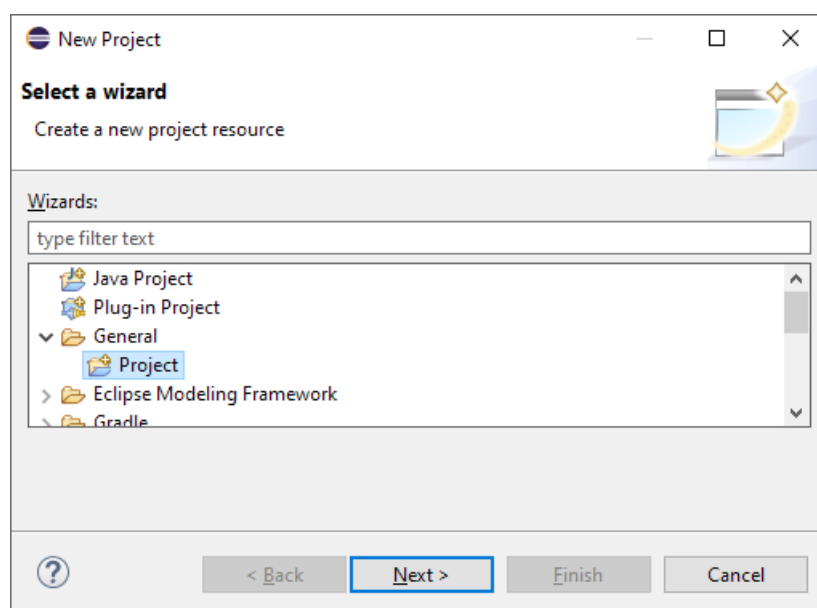
Click **Next**, and give a name to the project, e.g., **LampTest**:
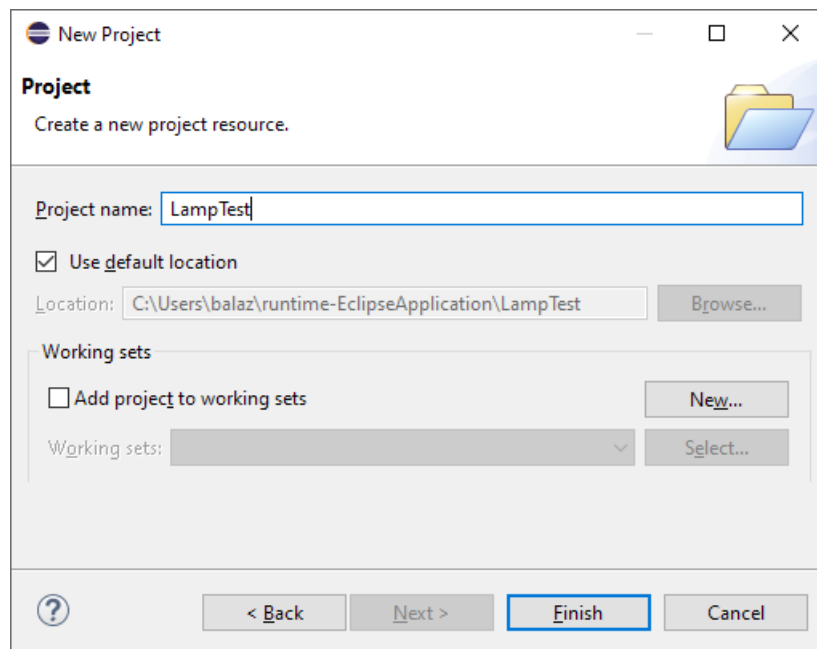
Finally, click on **Finish**.

Add a new file called **Lamp.sm** to the project! It is important to use the **sm** extension, because this is how Eclipse will recognize our state machine language. This was the extension we have specified when we created the Xtext project for our compiler. In case Eclipse asks whether you would like to convert the project to an Xtext project, answer with **Yes**:
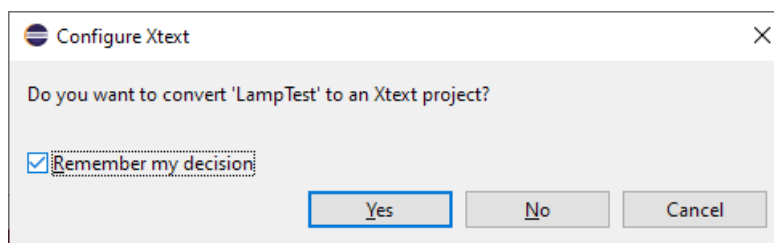
After adding the file, the project looks like:
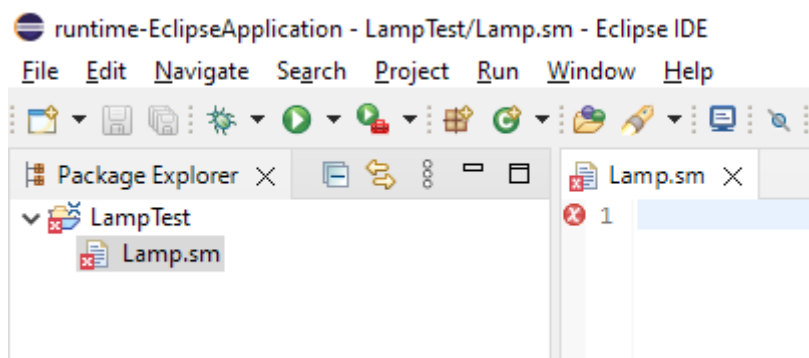
We already have an error signaled by Eclipse: the file is empty, but it should start with the keyword **machine**:
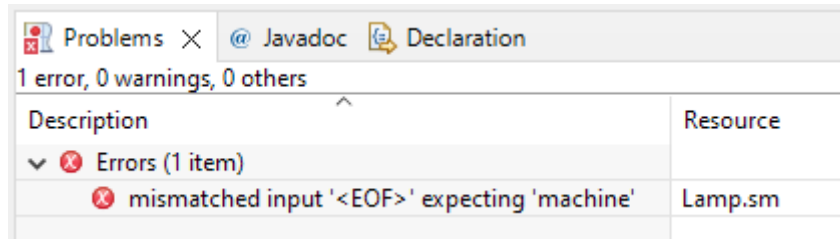
Try to press **Ctrl + Space** in the editor! Eclipse automatically adds the **machine** keyword, since at this position of the file, this is the only possible option:

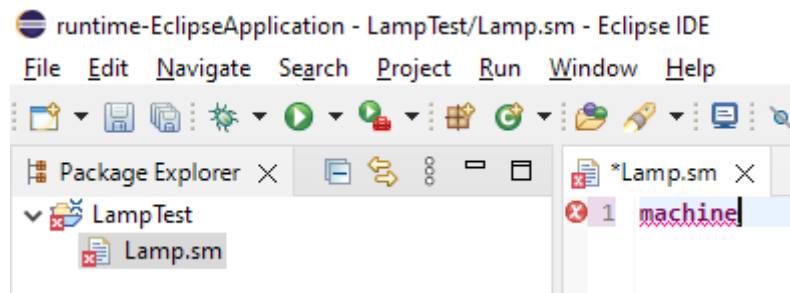Type in the example shown in chapter **Error! Reference source not found.**, and feel free to use the **Ctrl + Space** in the editor frequently to speed up the process! It will open the content assist function (called intellisense in Visual Studio). In the meantime, Eclipse provides nice syntax coloring for your code.

Try the following feature, too: while holding down the **Ctrl** key on your keyboard, click with the mouse on a name following a **jump** keyword (e.g., **On** or **Off** name)! The cursor will jump to the corresponding definition! Hence, Xtext can really do name analysis automatically!

Also check the code structure visualized by the **Outline** window:
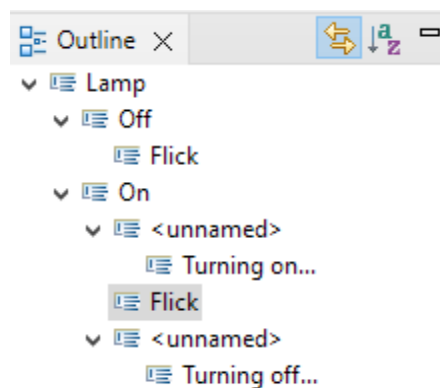

This view is also provided automatically by Xtext.

It can be seen that based on just the grammar, how many helpful IDE features Xtext provides by default. Of course, all these features can be customized. In more complex languages, it may even be necessary to do this customization, since Xtext cannot solve everything automatically.

Close the Runtime Eclipse and return to the original Eclipse.

7 Examining the generated files

Let's examine the files generated by Xtext!

Let's start with the **statemachine.dsl** project:
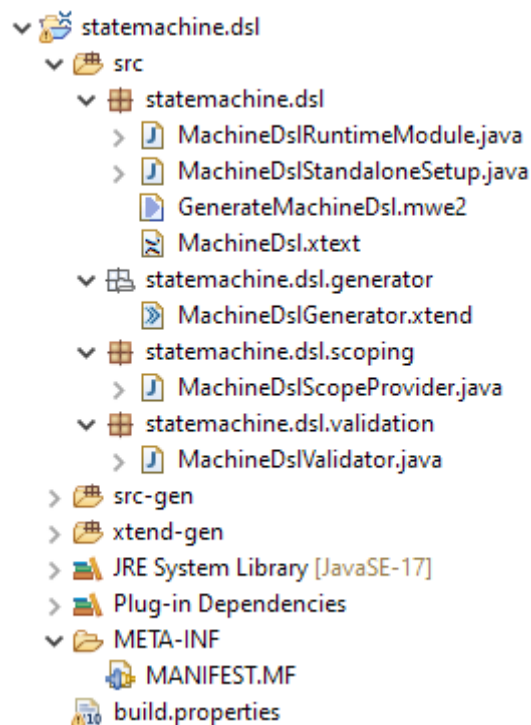
The files have the following roles:

- **MachineDslRuntimeModule.java**: you can register your own components here, with which you would like to customize the default behavior of Xtext
- **MachineDslStandaloneSetup.java**: helps to run the Xtext compiler independently of Eclipse, for example, within a simple console application
- **GenerateMachineDsl.mwe2**: generates all the Xtext files based on the grammar
- **MachineDsl.xtext**: contains the Xtext grammar of the state machine language
- **MachineDslGenerator.xtend**: using this generator you can generate your own custom code from the state machine language
- **MachineDslScopeProvider.java**: provides name analysis. This must be extended if you need more complex name analysis than is provided by Xtext automatically.
- **MachineDslValidator.java**: you can define your own semantic rules here which cannot be inferred from the grammar automatically by Xtext

Let's look at the **statemachine.dsl.tests** project:
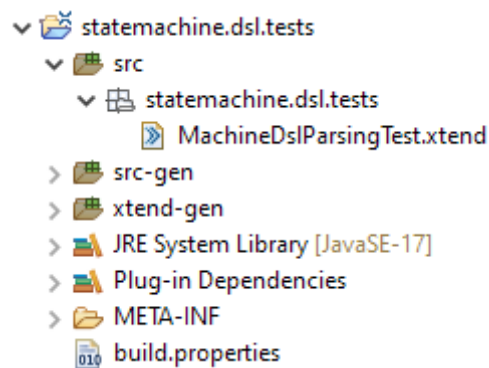
In the **MachineDslParsingTest.xtend** file you can define unit tests to check the correctness of the grammar by providing various state machine codes.

Let's look at the **statemachine.dsl.ide** project:
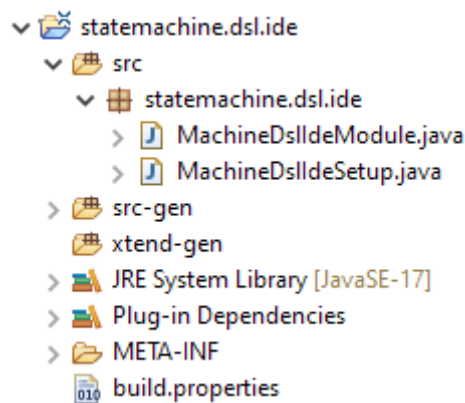
You can register your own IDE-specific components in **MachineDslIdeModule.java**, which can customize the default behavior of Xtext.

The **MachineDslIdeSetup.java** class can help to create your own Language Server. (See: Language Server Protocol, https://microsoft.github.io/language-server-protocol/) This way other IDE-s (e.g., Visual Studio Code) can also take advantage of the IDE features of Xtext.

Let's look at the **statemachine.dsl.ui** project:
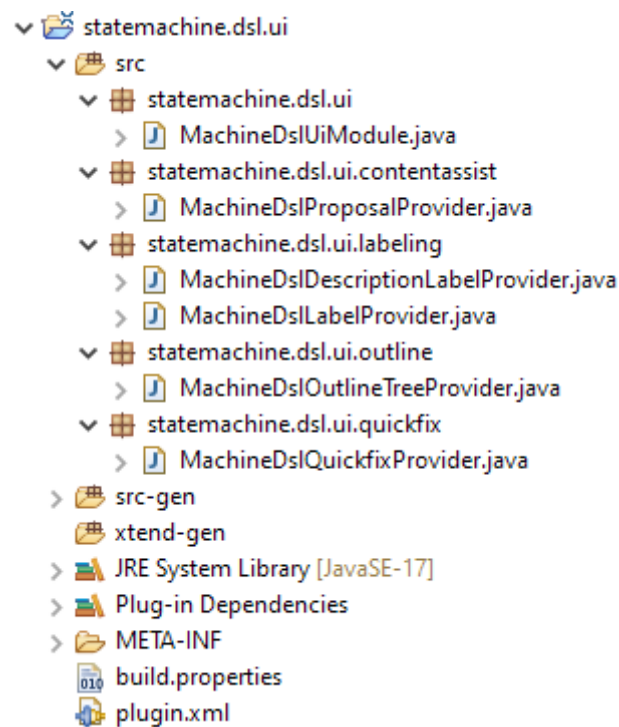
The roles of the various files are:

- **MachineDslUiModule.java**: you can register your own components to customize the default behavior of the state machine plugin inside the Eclipse IDE
- **MachineDslProposalProvider.java**: you can customize the behavior of content assist appearing for **Ctrl+Space**
- **MachineDslDescriptionLabelProvider.java**: you can customize the view of the various model elements
- **MachineDslLabelProvider.java**: you can customize the labels of the various model elements
- **MachineDslOutlineTreeProvider.java**: you can customize the **Outline** tree structure
- **MachineDslQuickfixProvider.java**: you can define your own quick fixes that help to fix mistakes recognized by the validators

In the **statemachine.dsl.ui.tests** project you can define test cases for the graphical Eclipse IDE plugins corresponding to the state machine language.

8   Defining a custom validation

Currently, the grammar does not enforce that an even handler must contain at most one **jump** statement. Let's write a validator that enforces this!

Inside the **statemachine.dsl** project modify **MachineDslValidator.java** as:

```java
package statemachine.dsl.validation;

import org.eclipse.xtext.validation.Check;
import org.eclipse.xtext.validation.CheckType;

import statemachine.model.EventHandler;
import statemachine.model.JumpStatement;
import statemachine.model.ModelPackage;

/**
 * This class contains custom validation rules.
 *
 * See https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#validation
 */
public class MachineDslValidator extends AbstractMachineDslValidator {

    @Check(CheckType.NORMAL)
    public void checkAtMostOneJump(EventHandler handler) {
        var jumpCounter = 0;
        for (var stmt: handler.getStatements()) {
            if (stmt instanceof JumpStatement) {
                ++jumpCounter;
            }
        }
        if (jumpCounter > 1) {
            error("An event handler cannot contain multiple jump statements.",
                ModelPackage.Literals.EVENT_HANDLER__KIND);
        }
    }
}
```

You can define as many validator functions as you like. The functions can have arbitrary names, but each of them must have exactly one parameter. The type of this parameter must be a class from the meta-model. Xtext will pass all possible model elements of the appropriate type to all validator functions. The **@Check** annotation can specify when the validation should happen:

- **CheckType.FAST**: the validation takes a very short time, it can be executed during typing
- **CheckType.NORMAL**: the validation takes more time, it should not be executed during typing, but can be executed after saving the file or during build
- **CheckType.EXPENSIVE**: the validation takes a lot of time, it should be executed only on-demand (right-click and selecting **Validate**)

Let's try our validator in the Runtime Eclipse:
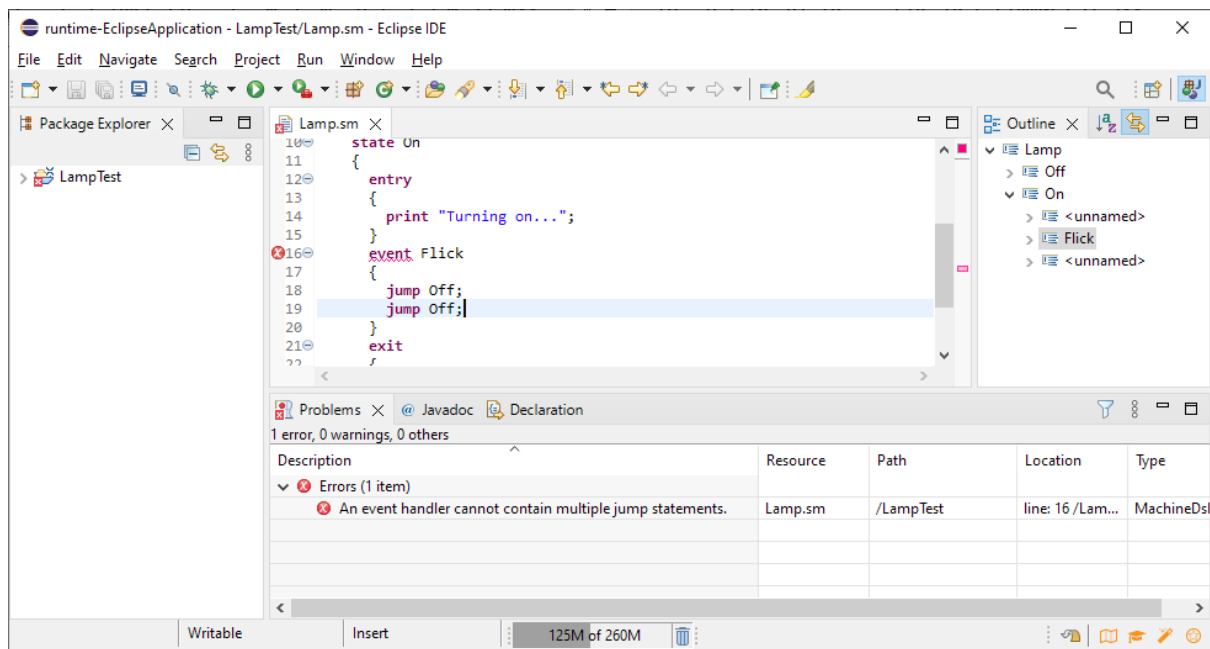
Close the Runtime Eclipse and return to the original Eclipse.

9 Defining the code generator: Xtend

We will need two helper functions in the meta-model. Inside the **statemachine.model** project in the **statemachine.xcore** file modify the **State** class as follows:

```
class State extends NamedElement
{
    boolean initial
    contains EventHandler[] handlers

    op EventHandler getEntry() {
        return handlers.filter[it.kind == EventHandlerKind.ENTRY].head
    }

    op EventHandler getExit() {
        return handlers.filter[it.kind == EventHandlerKind.EXIT].head
    }
}
```

Go back to the **statemachine.generator** project! Under the **src** inside the **statemachine. generator** package create a **MachineToJavaGenerator.xtend** file with the following content (the special chevron signs can be recalled using **Ctrl+Space**):

```
package statemachine.generator

import statemachine.model.EventHandler
import statemachine.model.JumpStatement
import statemachine.model.Machine
import statemachine.model.PrintStatement

class MachineToJavaGenerator {
    def generate(Machine machine) {
        var eventNames = machine.states.map[
                it.handlers.filter[it.name !== null].map[it.name]
            ].flatten.toSet
        var initialState = machine.states.filter[it.initial].head
        if (initialState === null) initialState = machine.states.head
        '''
        package statemachines;

        public class «machine.name» {
            private State state«IF initialState !== null» = State.«initialState.name»«ENDIF»;

            «FOR eventName: eventNames»
            public void «eventName»() {
                var nextState = state;
                switch (state) {
                «FOR state: machine.states.filter[it.handlers.exists[it.name == eventName]]»
                case «state.name»:
                    «var handler = state.handlers.filter[it.name == eventName].head»
                    «generateStatements(handler)»
                    break;
                «ENDFOR»
                default:
                    throw new IllegalStateException(state.toString());
                }
                if (nextState != state) {
                switch (state) {
                «FOR state: machine.states.filter[it.exit !== null]»
                case «state.name»:
                    exit«state.name»();
                    break;
                «ENDFOR»
```

```
                }
                state = nextState;
                switch (nextState) {
                «FOR state: machine.states.filter[it.entry !== null]»
                case «state.name»:
                    enter«state.name»();
                    break;
                «ENDFOR»
                }
            }
        }
    «ENDFOR»

    «FOR state: machine.states»
        «val entry = state.entry»
        «IF entry !== null»
        private void enter«state.name»() {
            «generateStatements(entry)»
        }
        «ENDIF»
        «val exit = state.exit»
        «IF exit !== null»
        private void exit«state.name»() {
            «generateStatements(exit)»
        }
        «ENDIF»
    «ENDFOR»

    private enum State {
        «FOR state: machine.states SEPARATOR ", "»«state.name»«ENDFOR»
    }
}
'''
}

def generateStatements(EventHandler handler) {
    '''
    «FOR stmt: handler.statements»
    «generateStatement(stmt)»
    «ENDFOR»
    '''
}

def dispatch generateStatement(PrintStatement stmt) {
    '''
    System.out.println("«stmt.text»");
    '''
}

def dispatch generateStatement(JumpStatement stmt) {
    '''
    nextState = State.«stmt.target.name»;
    '''
}
}
```

This generator translates our state machine description into executable Java code.

Inside the **statemachine.dsl** project modify the **statemachine.dsl.generator/ MachineDslGenerator.xtend** file as follows:

```
package statemachine.dsl.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.AbstractGenerator
import org.eclipse.xtext.generator.IFileSystemAccess2
import org.eclipse.xtext.generator.IGeneratorContext
import statemachine.generator.MachineToJavaGenerator
import statemachine.model.Machine

/**
 * Generates code from your model files on save.
 *
 * See https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#code-generation
 */
class MachineDslGenerator extends AbstractGenerator {

    override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
        if (resource?.contents.size != 0 &&
            resource?.contents?.get(0) instanceof Machine) {
            val machine = resource.contents.get(0) as Machine
            val gen = new MachineToJavaGenerator()
            val path = "statemachines/"+machine.name+".java";
            val code = gen.generate(machine)
            fsa.generateFile(path, code)
        }
    }
}
```
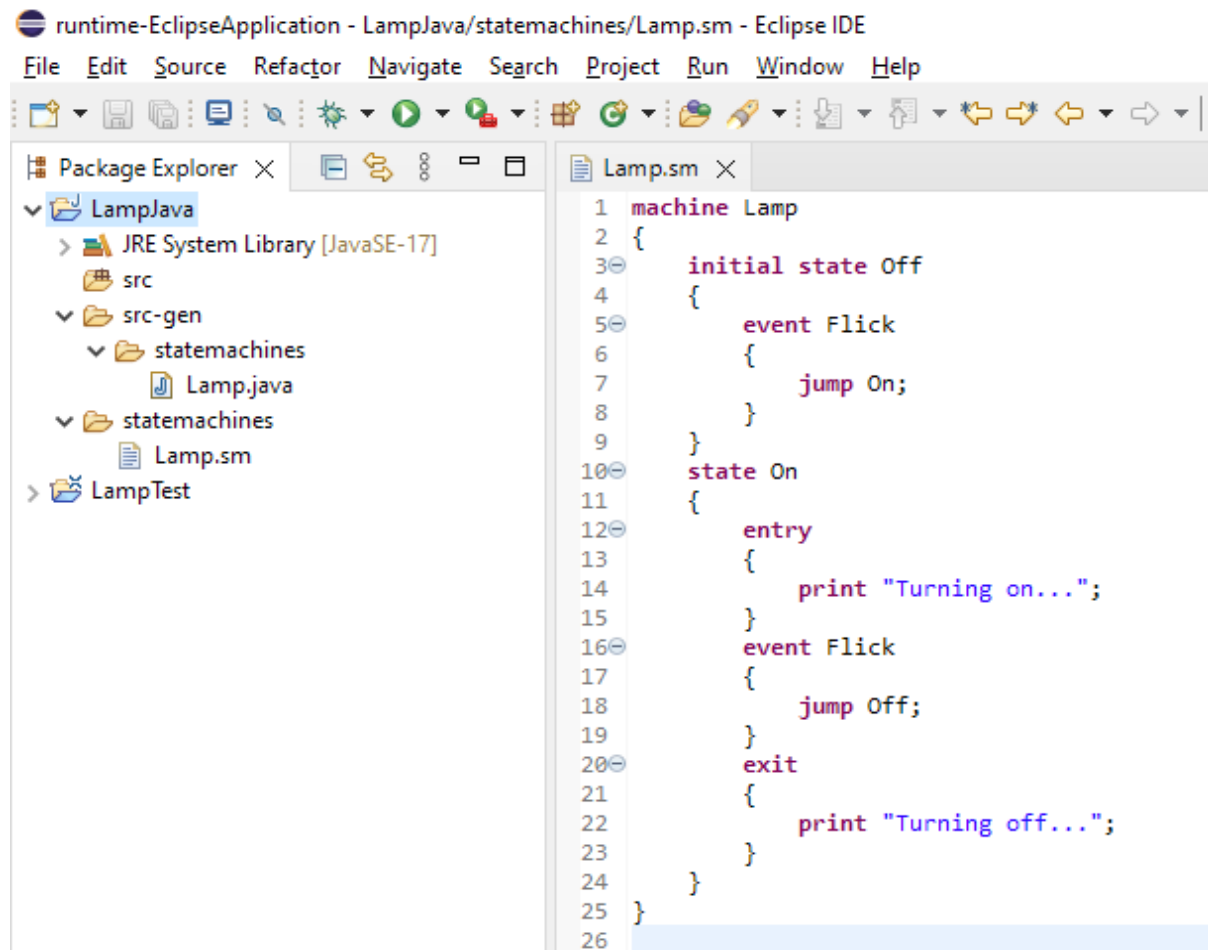
10 Checking out the generator

Right click on the **statemachine.dsl** project, and select **a Run As > Eclipse Application** to start the Runtime Eclipse!

Inside the Runtime Eclipse create a new Java project called **LampJava** (don't create any **module-info.java** files). Inside the project create a folder called **statemachines**, and create a new file called **Lamp.sm** with the content shown in chapter **Error! Reference source not found.**. When you save the file, a new **src-gen/statemachines** folder is created with a **Lamp.java** file:

Examine the generated code! You will also see that if you use Xtend correctly, the generated output is nicely formatted.

In order to be able to use this **Lamp.java** file, you need to make the **src-gen** folder to a source folder. Right click on the project and select **Properties**. Find the **Java Build Path** page, and the **Source** tab:
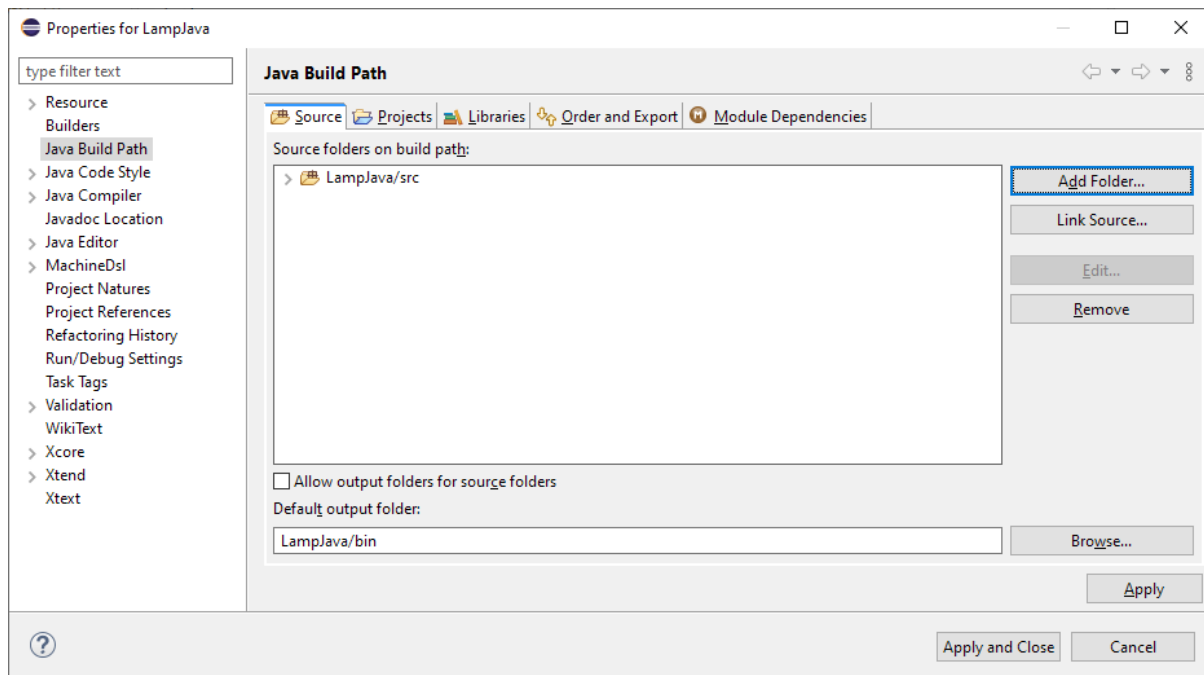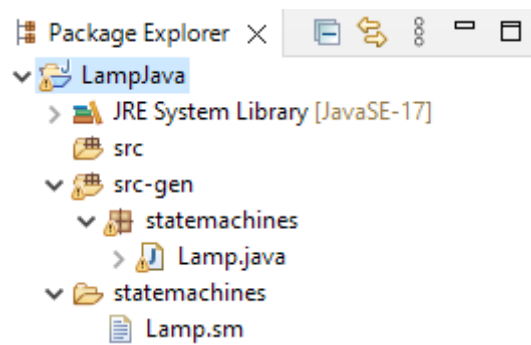

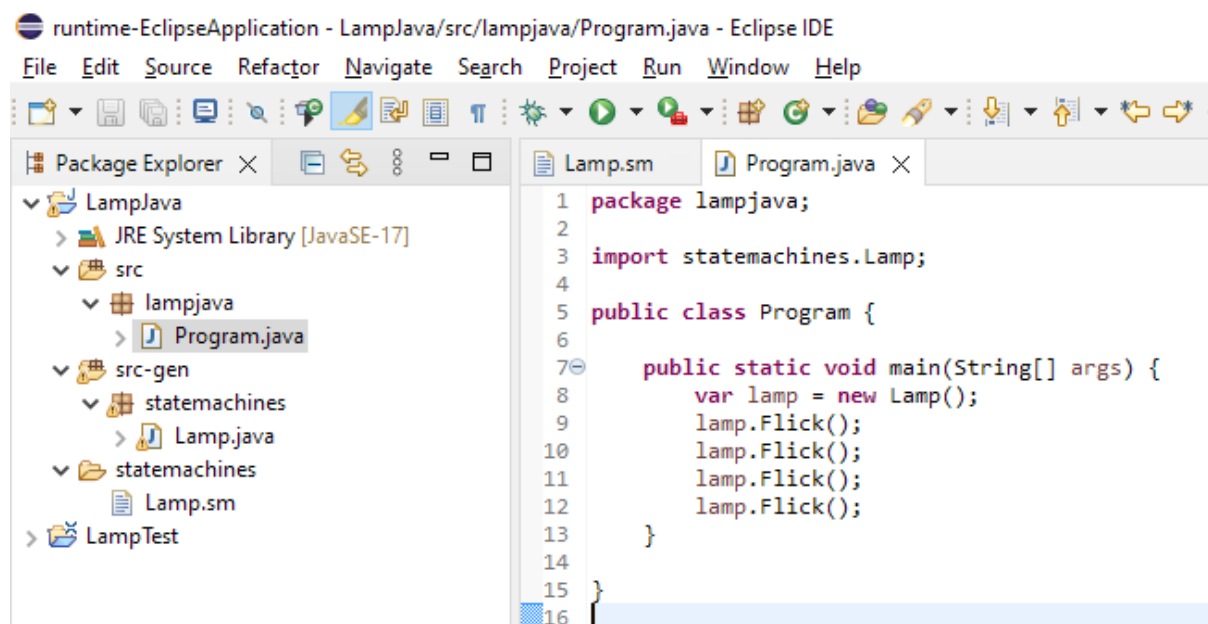Click **Add Folder…** and put a checkmark before the **src-gen** folder:
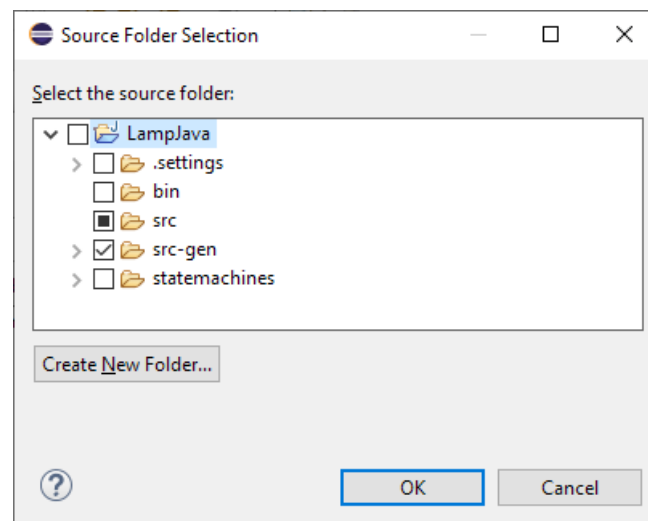
Click **OK** and close the **Properties** window using the **Apply and Close** button!
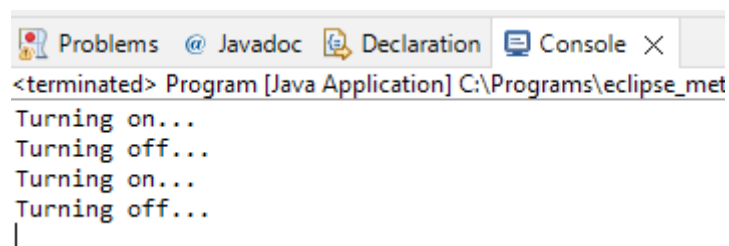
Now the project looks like this:

Under the **src** folder inside the **lampjava** package create a class called **Program** with a **main** function:

Right click on **Program.java**, and select **Run As > Java Application**! If you have done everything correctly, you should see the following output in the **Console** window:

Make some changes in the **Lamp.sm** file, save it, and run the application again! You will see that all changes are automatically mapped to the generated **Lamp.java** file.

11 Summary

This practice provided a quick glimpse into what Xcore, Xtext and Xtend are capable of, and how to define a custom textual DSL with their help. There are numerous other features that we have not touched yet, but some of those will be covered in the laboratory version of the subject.

12 Optional extensions for further practice

Here are some extra ideas to implement if you want to dig deeper into this topic.

12.1 Further validations (easy)

There are some language rules for which we have not defined validators. For example, there must be exactly one initial state, or that a state can have at most one entry and at most one exit event handler. Write the corresponding validators!

12.2 Boolean fields and if-else statements (medium)

Extend the language so that like a class in Java, the state machine could have boolean fields. Add assignment statements, too, to be able to assign values (true or false constants) to these boolean fields.

Add if-else statements into the language which can provide conditional jumps to the next states.

To solve this task, you may need to modify the Xcore meta-model, the Xtext compiler and the Xtend generator.

Also, you should remove the validator that allows only one jump statement per event hander.

12.3 Fields, statements and expressions (hard)

Extend the language so that like a class in Java, the state machine could have integer fields. Add assignment statements and arithmetic expressions (addition, subtraction, multiplication, etc.) to the language.

To solve this task, you may need to modify the Xcore meta-model, the Xtext compiler and the Xtend generator.

12.4 Functions (hard)

Extend the language so that like a class in Java, the state machine could have member functions. The functions should be able to access fields, however, they should not contain jump statements. Functions should be able to call each other.

To solve this task, you may need to modify the Xcore meta-model, the Xtext compiler and the Xtend generator.

12.5 Parallel regions (hard)

Extend the language so that it supports other capabilities of the standard UML statecharts, e.g., parallel regions.