

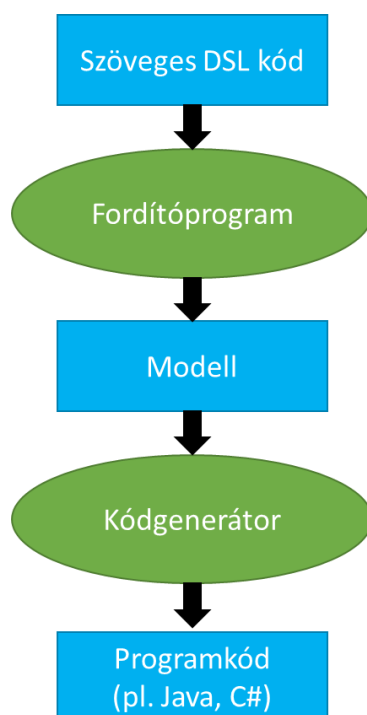
3. Gyakorlat – Xcore, Xtext és Xtend

Dr. Simon Balázs, 2024.

1 Bevezetés

Ez az útmutató arra mutat példát, hogy hogyan lehet az Xtext és Xtend keretrendszerrel egy egyszerű szöveges DSL-hez fordítót és kódgenerátort, valamint Eclipse fejlesztőkörnyezethez (IDE – Integrated Development Environment) támogatást készíteni.

Egy szöveges DSL feldolgozása az alábbi lépésekből áll:



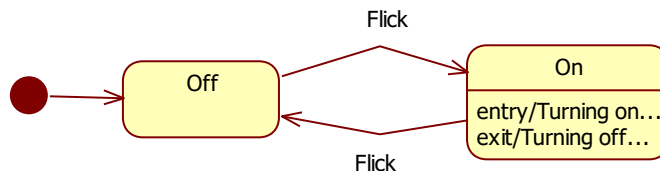
A szöveges DSL kódból a fordító egy modellt épít. Ehhez meg kell adni, hogy a modell milyen elemekből áll, és azok hogyan kapcsolódnak egymáshoz. Erre szolgál a metamodel. A fordító definiálásánál meg kell adni azt is, hogy a kód elemei hogyan képezhetők le a metamodel egyes elemeire. Ha ez a leképezés elég pontos, a modellt a fordító automatikusan fel tudja építeni. A felépített modell bejárásával egy kódgenerátor készíthet valamilyen kimeneti kódot.

A feladat megoldása előtt készítsük elő a projekteket a **3. Gyakorlat – Projektek létrehozása** c. útmutató alapján!

2 Programnyelv

A cél, hogy egy egyszerű állapotgép-leíró nyelvhez készítsünk fordítót és eszköztámogatást.

Vegyük például az alábbi UML állapotgépet, amely egy kapcsolós lámpa működését írja le:



Az általunk készített szöveges DSL-ben a fenti állapotgép leírása az alábbi:

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

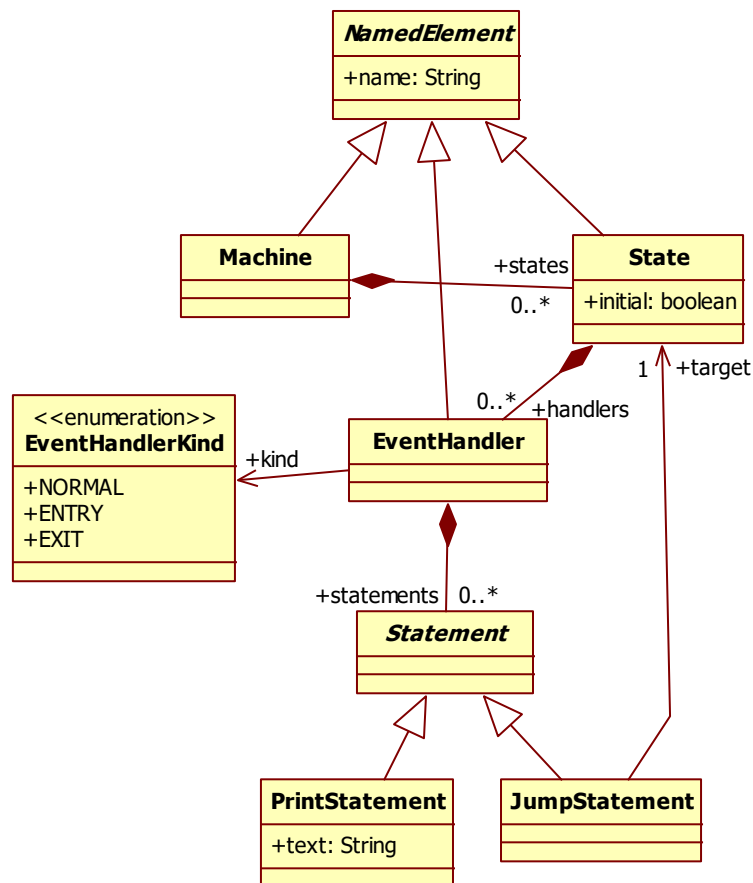
Egy állapotgépet a **machine** kulcsszó definiál. Az állapotgépnek vannak állapotai (**state** kulcsszó), a kezdő állapotot az **initial state** adja meg. Kezdőállapotból pontosan egynek kell lennie.

Egy állapoton belül érkehetnek események (**event**). Az eseménykezelő törzse adja meg azt, hogy az esemény hatására mi történjen. Egy állapotnak lehet legfeljebb egy belépési (**entry**) és legfeljebb egy kilépési (**exit**) eseménykezelője is. A belépési eseménykezelő akkor fut le, amikor belépünk az adott állapotba, a kilépési eseménykezelő pedig akkor, amikor kilépünk az adott állapotból.

Az eseménykezelőkön belül kétfajta utasítás szerepelhet. A **jump** utasítás hatására az állapotgép állapotot vált, de csak az eseménykezelő teljes lefutása után. Egy eseménykezelőben legfeljebb egy **jump** utasítás szerepelhet. A **print** utasítás egy karakterláncot ír ki a képernyőre.

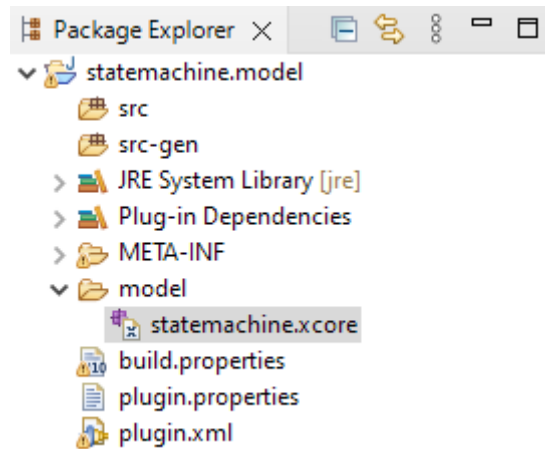
3 Metamodel

Az állapotgép leírását az alábbi metamodelnek megfelelő modellé fogjuk lefordítani:



4 Metamodell elkészítése: Xcore

A **statemachine.model** projektben a következőképpen néz ki:



A **model** könyvtár alatt a **statemachine.xcore** fájlban kell definiálnunk a metamodell szöveges leírását Xcore szintaxissal. A fájl tartalma a 3. fejezetnek megfelelően a következő legyen:

```
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
}

```

Ez pontosan a fenti UML osztálydiagram segítségével leírt metamodel: viszonylag egyszerűen áttanszformálható az osztálydiagram jelentése.

A **statemachine.xcore** fájlból a háttérben Java kód generálódik, amely a metamodel Java implementációját tartalmazza EMF (Eclipse Modeling Framework) keretrendszerben. Egy ennek a metamodelnek megfelelő modellt építhetünk fel a fordítóprogram segítségével, amelyet a következő szakaszban készítünk el.

5 Fordító elkészítése: Xtext

A fordító két részből áll: egy lexerből és egy parszerből. A lexer tartalmazza az elemi fordítási egységeket, vagyis a tokeneket. A parszer pedig a kód struktúráját leíró nyelvtani szabályokat adja meg. Xtext esetén a kettő összevonható. A kulcsszavakat inline beépíthetjük a parszer nyelvtanba, és abból az Xtext automatikusan elkészíti a lexert. Az Xtext emellett néhány szokásos tokent (ID, INT, STRING) és a C-stílusú kommenteket is automatikusan belerakja a lexerbe. Az ID tokenet kell használni egy olyan név definiálására, amelyet a névelemzés során fel kell tudni oldani. Az Xtext egy kicsivel többet tud, mint egy hagyományos parszer generátor (pl. ANTLR4), hogy a szemantikai elemzésből a névfeloldást is képes automatikusan elvégezni. Névre történő hivatkozásnál a nyelvtanban szögletes zárójelbe kell tenni a hivatkozott elem típusát.

Ennek megfelelően az állapotgépleíró nyelvünk nyelvtanának Xtext leírása a következő (**MachineDsl.xtext** fájl a **statemachine.dsl** projektben):

```

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 ';';

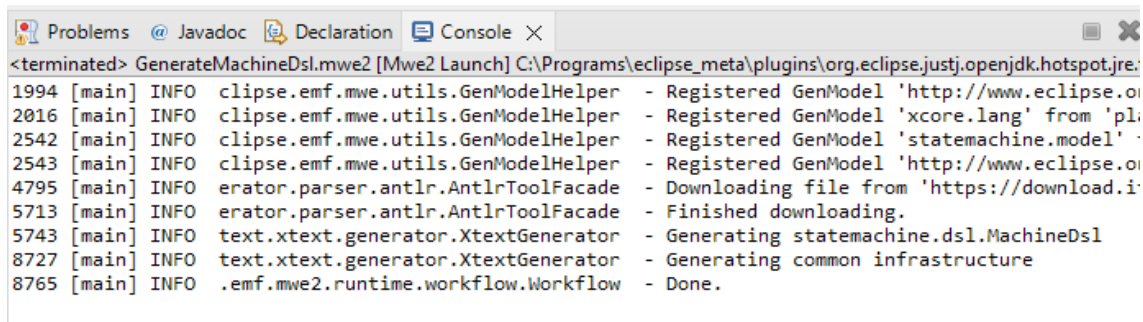
JumpStatement returns JumpStatement: 'jump' target=[State] ';';

```

Jól látható, hogy ha a metamodel jól van megtervezve, a nyelvtani szabályok viszonylag könnyen elkészíthetők. Fontos, hogy az Xtext leírásban a nyelvtani szabályok neve pontosan megegyezik a metamodelben definiált osztályok nevével, valamint a nyelvtani szabályok jobb oldalán álló property-k neve is megegyezik a metamodelben definiált property-k neveivel.

Ha a fenti nyelvtani leírással megvagyunk, kattintsunk a **GenerateMachineDsl.mwe2** fájl jobb gombbal, és válasszuk a **Run As > MWE2 Workflow** menüpontot. Ekkor az Xtext legenerálja a fordítót implementáló Java osztályokat. Ezt a lépést mindig meg kell tennünk, valahányszor módosítjuk az Xtext nyelvtant.

Ha mindent jól csináltunk, a generálás sikeresen befejeződik:



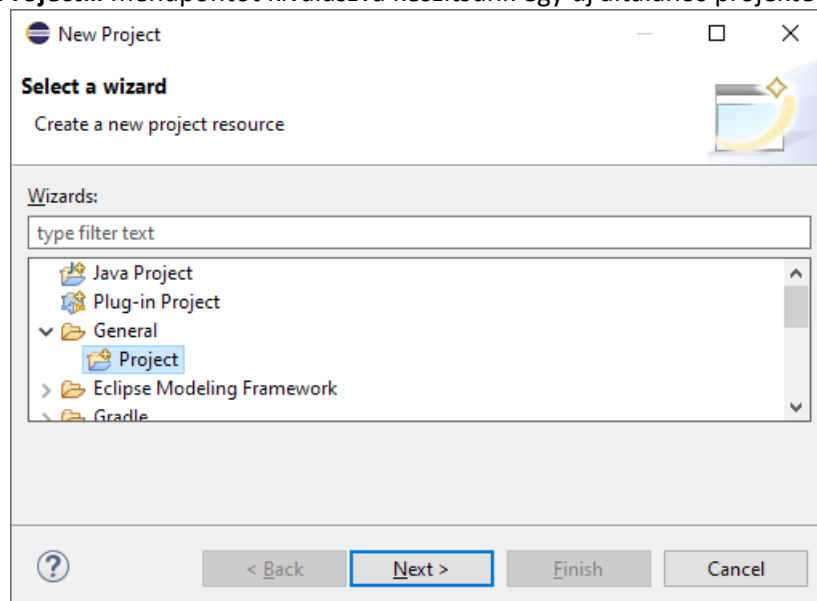
```
<terminated> GenerateMachineDsl.mwe2 [Mwe2 Launch] C:\Programs\eclipse_meta\plugins\org.eclipse.justi.openjdk.hotspot.jre:
1994 [main] INFO   eclipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'http://www.eclipse.org/xcore/2008/09/01/xcore.ecore'
2016 [main] INFO   eclipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'xcore.lang' from 'plugin:/org.eclipse.xtext.xcore.ecore'
2542 [main] INFO   eclipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'statemachine.model' from 'plugin:/org.eclipse.xtext.statemachine.model'
2543 [main] INFO   eclipse.emf.mwe.utils.GenModelHelper - Registered GenModel 'http://www.eclipse.org/xtext/2008/09/01/xtext.ecore'
4795 [main] INFO   erator.parser.antlr.AntlrToolFacade - Downloading file from 'https://download.eclipse.org/xtext/2008/09/01/xtext.ecore'
5713 [main] INFO   erator.parser.antlr.AntlrToolFacade - Finished downloading.
5743 [main] INFO   text.xtext.generator.XtextGenerator - Generating statemachine.dsl.MachineDsl
8727 [main] INFO   text.xtext.generator.XtextGenerator - Generating common infrastructure
8765 [main] INFO   .emf.mwe2.runtime.workflow.Workflow - Done.
```

6 IDE plugin kipróbálása

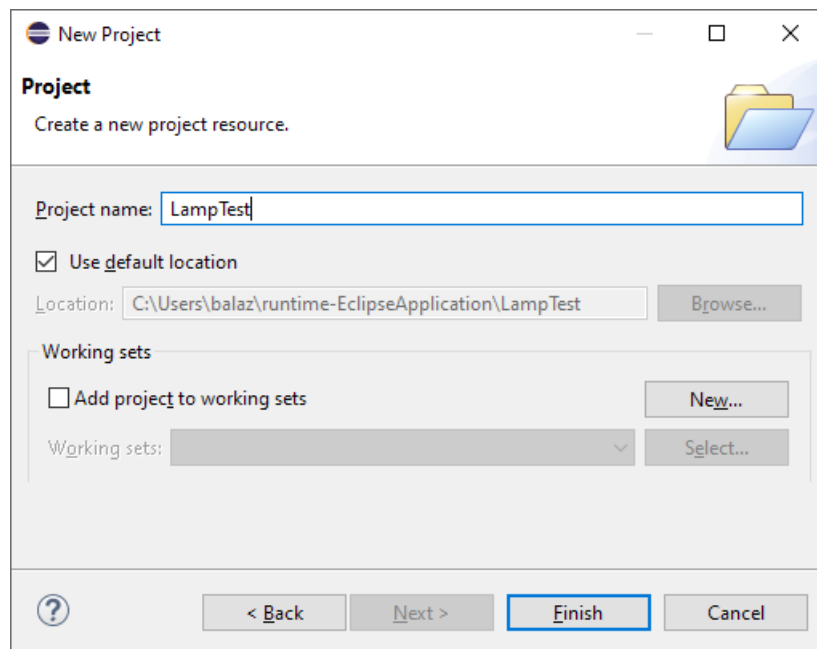
A következő szakaszokban részletesen is megvizsgáljuk majd a generált fájlokat, de előtte próbáljuk ki az Xtext által alapértelmezetten készített IDE támogatást a nyelvünkhöz!

Ehhez kattintsunk jobb gombbal a **statemachine.dsl** projekten, és válasszuk a **Run As > Eclipse Application** menüpontot! Ekkor egy másik Eclipse, az úgynevezett **Runtime Eclipse** fog elindulni, amelybe alapértelmezett módon telepítve lesznek azok a pluginok, amelyeket az Xtext generált a saját állapotgépes DSL-ünkhöz.

A **File > New > Project...** menüpontot kiválasztva készítsünk egy új általános projektet:

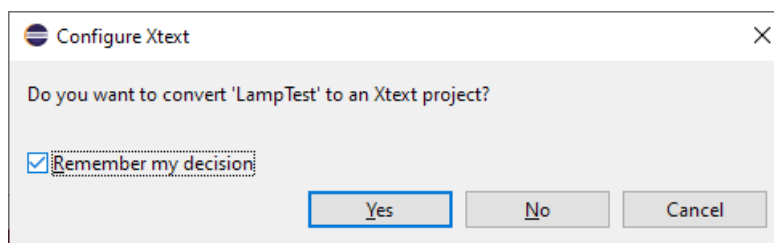


Kattintsunk a **Next**-re, és adjunk meg egy tetszőleges projekt nevet, pl. **LampTest**:

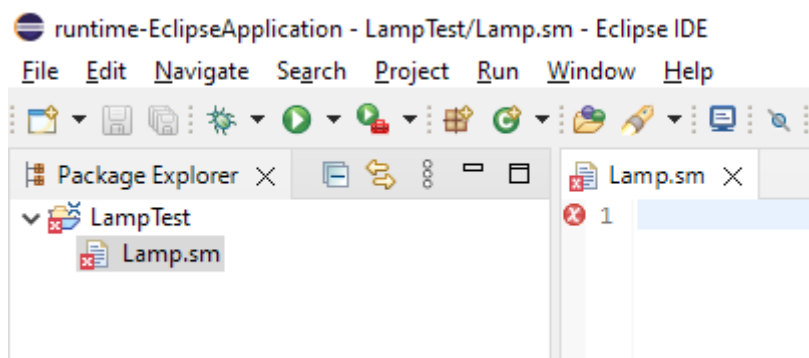


Végül kattintsunk a **Finish** gombra.

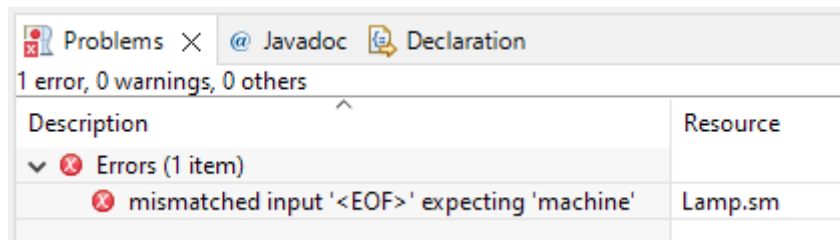
A megjelenő projekthez adjunk hozzá egy **Lamp.sm** nevű fájlt! Fontos, hogy az **sm** kiterjesztést használjuk, mert így fogja felismerni a saját nyelvünket az Eclipse, hiszen ezt a kiterjesztést adtuk meg az Xtext projektünk létrehozásánál. Ha az Eclipse a fájl hozzáadásakor rákérdez, hogy szeretnénk-e Xtext projektté konvertálni a projektünket, akkor feleljünk igennel:



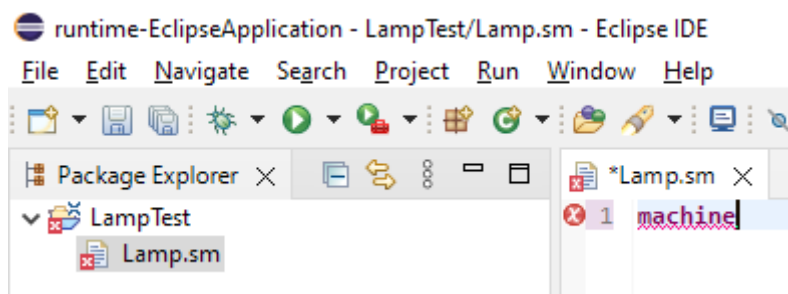
A fájl hozzáadása után a projekt így néz ki:



Már is van egy hiba, amit az Eclipse automatikusan kijelez: üres a fájl, pedig **machine** kulcsszóval kellene kezdődnie:



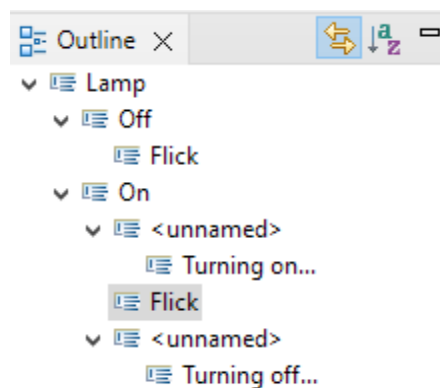
Próbáljuk ki, hogy a szövegszerkesztőben lenyomjuk a **Ctrl + Space** billentyűkombinációt! Ennek hatására az Eclipse automatikusan beilleszti a **machine** kulcsszót, mert ezen a ponton csak ez következhet:



Gépeljük be a 2. fejezetben bemutatott példát, és közben bátran használjuk a **Ctrl + Space** segítségével előhozható content assist funkciót (a Visual Studio-ban intellisense néven ismert)! Időközben az Xtext szépen kiszínezi a kódunkat.

Próbáljuk ki az alábbi funkciót is: a **Ctrl** gomb nyomvatartása mellett kattintsunk bal egérgombbal valamelyik **jump** utasítás után álló **On** vagy **Off** névre! A kurzor automatikusan átugrik a megfelelő definícióra! Működik tehát az automatikus névfeloldás is!

Vizsgáljuk meg az **Outline** ablakban a kód struktúráját:



Ezt a nézetet is automatikusan biztosítja az Xtext.

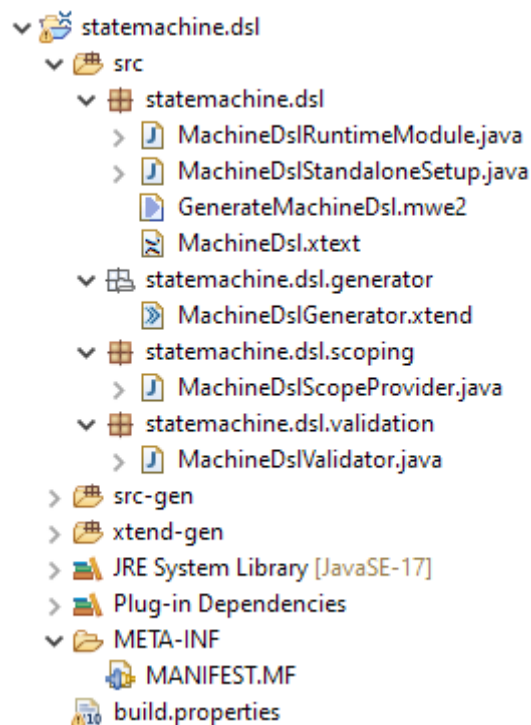
Jól látható, hogy csak a nyelvtan megadása után már mennyi sok segítséget tud adni az Xtext. Természetesen az itt bemutatott funkciók mind testreszabhatók. Sokszor szükség is van rá, ha bonyolultabb a programnyelvünk, és az Xtext nem tud mindent automatikusan megoldani.

Zárjuk be a Runtime Eclipse-et, és térjünk vissza az eredeti Eclipse-hez!

7 Generált fájlok vizsgálata

Vizsgáljuk meg, hogy milyen fájlokat állított elő az Xtext!

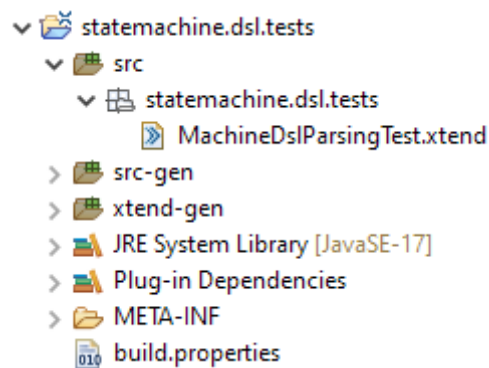
Kezdjük a **statemachine.dsl** projekttel:



Az egyes fájlok feladata a következő:

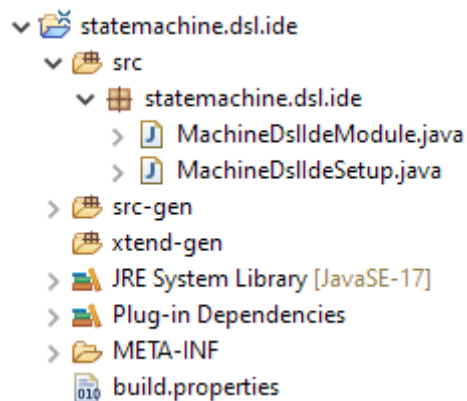
- **MachineDslRuntimeModule.java:** itt lehet beregisztrálni a saját komponenseinket, amellyel az Xtext alapértelmezett működését szeretnénk testreszabni
- **MachineDslStandaloneSetup.java:** ennek az osztálynak a segítségével lehet egy egyszerű konzol alkalmazásban, Eclipse-től függetlenül is használni az Xtext fordítót
- **GenerateMachineDsl.mwe2:** ez a workflow generálja a különböző fájlokat az Xtext nyelvtanból
- **MachineDsl.xtext:** ez a saját nyelvünk Xtext nyelvtani leírása
- **MachineDslGenerator.xtend:** ennek a segítségével lehet a saját nyelvünkből más programkódokat generálni
- **MachineDslScopeProvider.java:** ez az osztály végzi a névfeloldást a névelemzés során. Ezt kell specializálni, ha az Xtext automatikus névfeloldása nem elegendő számunkra.
- **MachineDslValidator.java:** ebben az osztályban lehet saját szemantikai ellenőrző szabályokat definiálni, amelyek a nyelvtanból nem következethetők ki automatikusan

Tekintsük a **statemachine.dsl.tests** projektet:



Itt a **MachineDslParsingTest.xtend** fájlban megadhatunk olyan unit teszteket, ahol a nyelvtanunkat ellenőrizhetjük, hogy az általunk specifikált állapotgép programkódokat képes-e megfelelően elemezni.

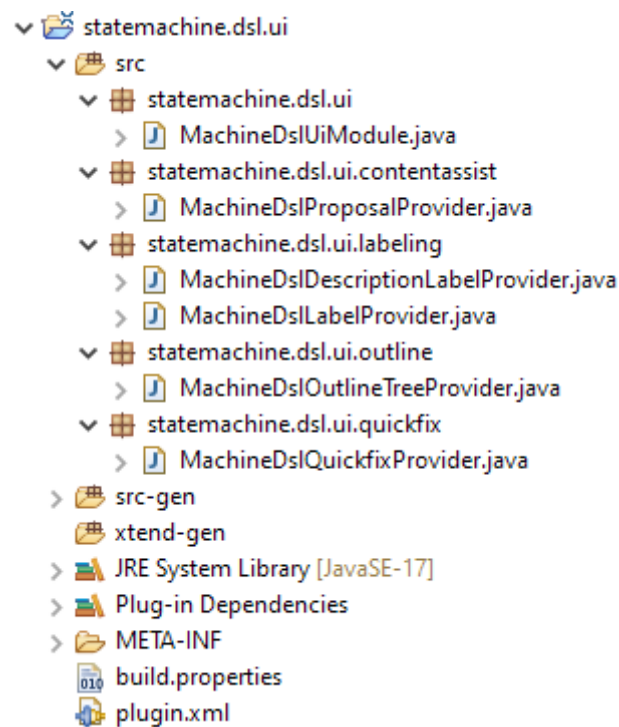
Tekintsük a **statemachine.dsl.ide** projektet:



A **MachineDslIdeModule.java** fájlban lehet beregisztrálni a saját IDE specifikus komponenseinket, amelyekkel az Xtext alapértelmezett működését szeretnénk testre szabni.

A **MachineDslIdeSetup.java** osztály segítségével lehet Xtext alapú Language Servert beüzemelni. (Lásd: Language Server Protocol, <https://microsoft.github.io/language-server-protocol/>) Így nemcsak Eclipse alatt, hanem más IDE-k alatt (pl. Visual Studio Code) is működnek az Xtext által kínált funkciók.

Tekintsük a **statemachine.dsl.ui** projektet:



Az egyes fájlok feladata a következő:

- **MachineDslUiModule.java:** itt lehet beregisztrálni a saját komponenseinket, amellyel az Xtext Eclipse IDE-re vonatkozó alapértelmezett működését szeretnénk testreszabni
- **MachineDslProposalProvider.java:** itt lehet testreszabni a **Ctrl+Space**-re felugró content assist működését
- **MachineDslDescriptionLabelProvider.java:** itt lehet testreszabni az egyes modellelemek leírásait
- **MachineDslLabelProvider.java:** itt lehet testreszabni az egyes modellelemek címkéit
- **MachineDslOutlineTreeProvider.java:** itt lehet testreszabni az alapértelmezett **Outline** fastruktúrát
- **MachineDslQuickfixProvider.java:** itt lehet felvenni saját quick-fixeket, vagyis olyan akciókat, amelyek valamilyen validációs hibát képesek javítani

A **statemachine.dsl.ui.tests** projektben a DSL-ünkhöz tartozó Eclipse IDE grafikus felületével kapcsolatos teszteket lehet definiálni.

8 Saját validáció készítése

Jelenleg a nyelvtan nem kényszeríti ki, hogy egy eseménykezelő blokkban legfeljebb egy **jump** utasítás fordulhat elő. Írjunk saját validátort, amely ezt ellenőrzi!

Ehhez a **statemachine.dsl** projektben a **MachineDslValidator.java** fájlt egészítsük ki az alábbi módon:

```
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;

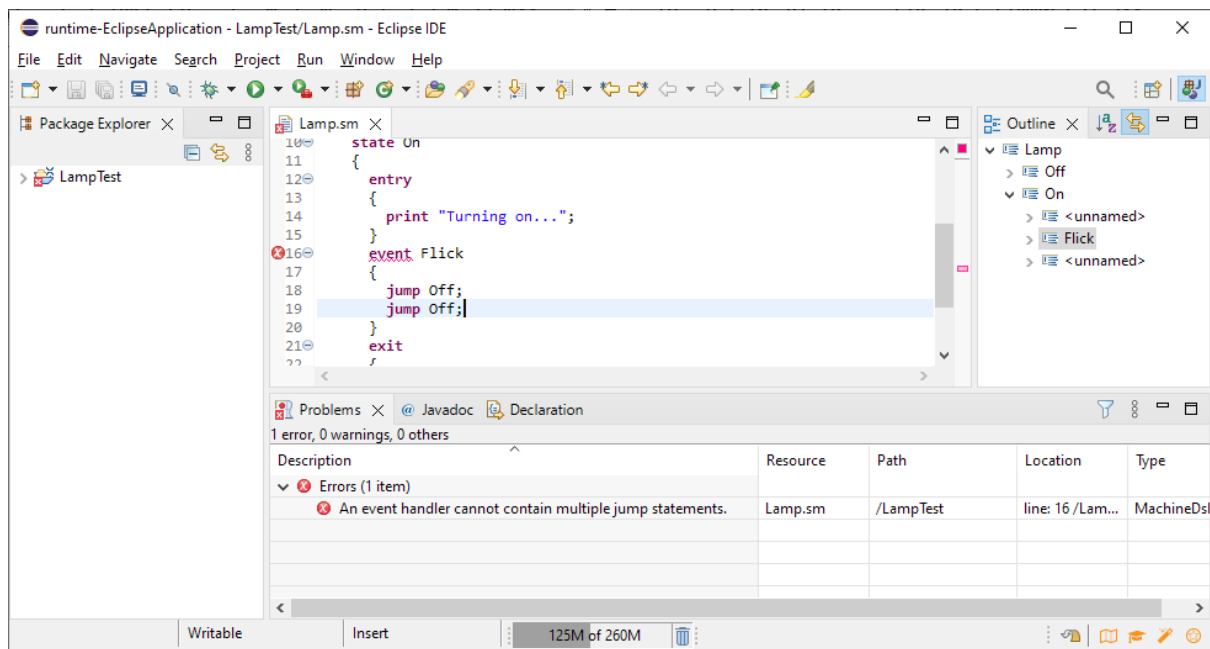
/**
 * This class contains custom validation rules.
 *
 * See https://www.eclipse.org/Xtext/documentation/303\_runtime\_concepts.html#validation
 */
public class MachineDslValidator extends AbstractMachineDslValidator {

    @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);
        }
    }
}
```

A validátoron belül tetszőlegesen sok függvény felvehető. A függvényeket tetszőleges módon elnevezhetjük. A lényeg, hogy minden függvénynek pontosan 1 paramétere legyen: valamilyen modellelem, amin egy szemantikai ellenőrzést szeretnénk végezni. Az Xtext az összes lehetséges szóba jöhető modellelemet be fogja küldeni az összes validáló függvénynek ellenőrzésre. A **@Check** annotációban megadható, hogy mikor fusson le az ellenőrzés:

- **CheckType.FAST**: rövid idő alatt kiértékelhető ellenőrzés, gépelés közben is lefutásra kerül
- **CheckType.NORMAL**: sokáig tartó ellenőrzés, amely gépelés közben nem, csak mentés és buildelés során fut le
- **CheckType.EXPENSIVE**: nagyon sokáig tartó ellenőrzés, amely csak külön kérésre fut le (jobb gomb, **Validate** menüpont választása)

Próbáljuk ki a validátorunkat a Runtime Eclipse alatt:



Zárjuk be a Runtime Eclipse-et, és térjünk vissza az eredeti Eclipse-hez!

9 Kódgenerátor elkészítése: Xtend

A generátor működéséhez szükség lesz két segédfüggvényre a metamodelben, így a **statemachine.model** projektben a **statemachine.xcore** fájlban a **State** osztályt frissítsük az alábbi módon:

```
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
    }
}
```

Térjünk vissza a **statemachine.generator** projektbe! Az **src** könyvtár alatt egy **statemachine.generator** csomagban a **MachineToJavaGenerator.xtend** nevű fájlt töltsük fel az alábbi tartalommal (a speciális dupla-kacsacsőr jeleket a **Ctrl+Space** billentyűkombinációval hozhatjuk létre):

```
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»;
    ...
}
}

```

Ez a generátor Java osztállyá fordítja az állapotgép leírását.

A **statemachine.dsl** projektben az **statemachine.dsl.generator/MachineDslGenerator.xtend** fájlt írjuk át így:

```
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

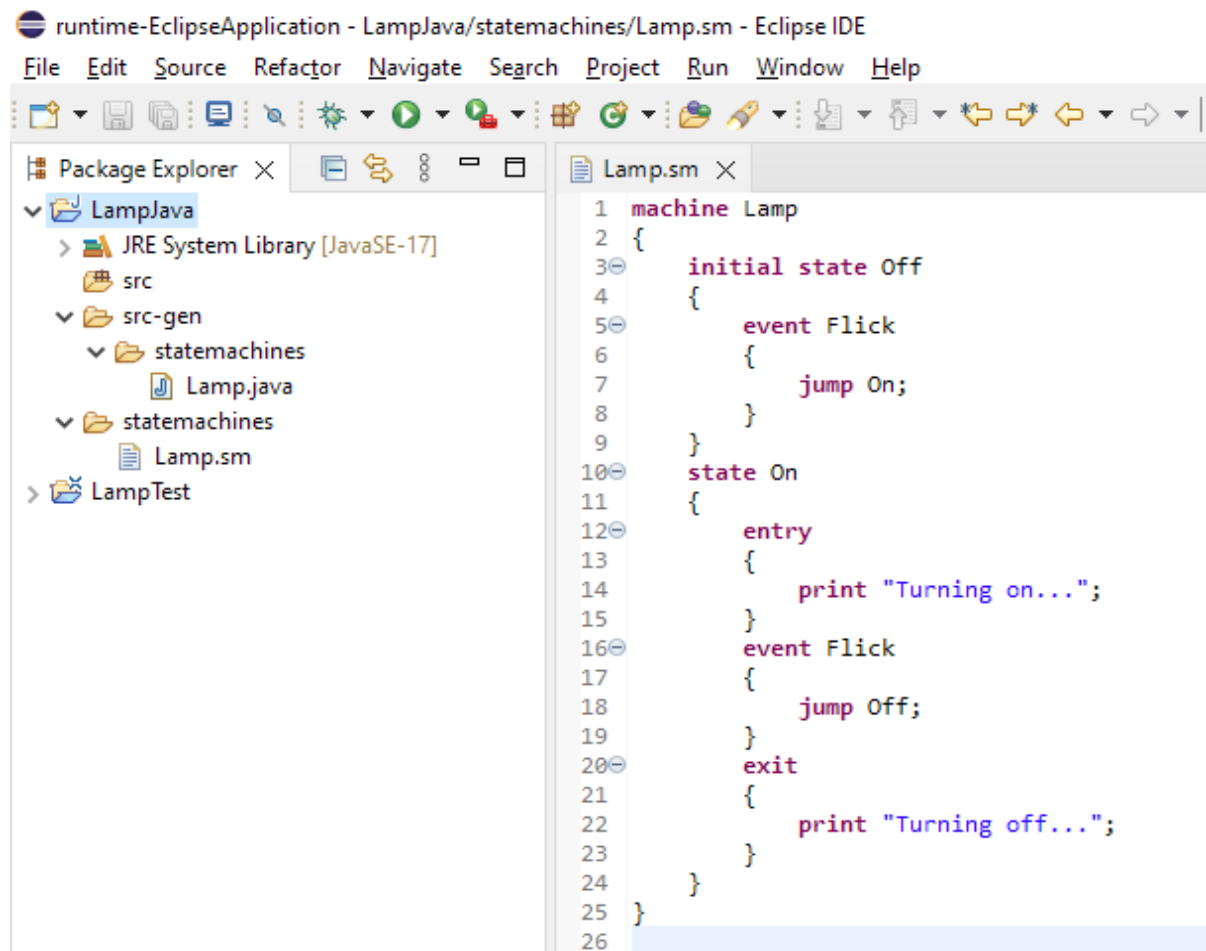
/**
 * Generates code from your model files on save.
 *
 * See https://www.eclipse.org/Xtext/documentation/303\_runtime\_concepts.html#code-generation
 */
class MachineDslGenerator extends AbstractGenerator {

    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 Generátor kipróbálása

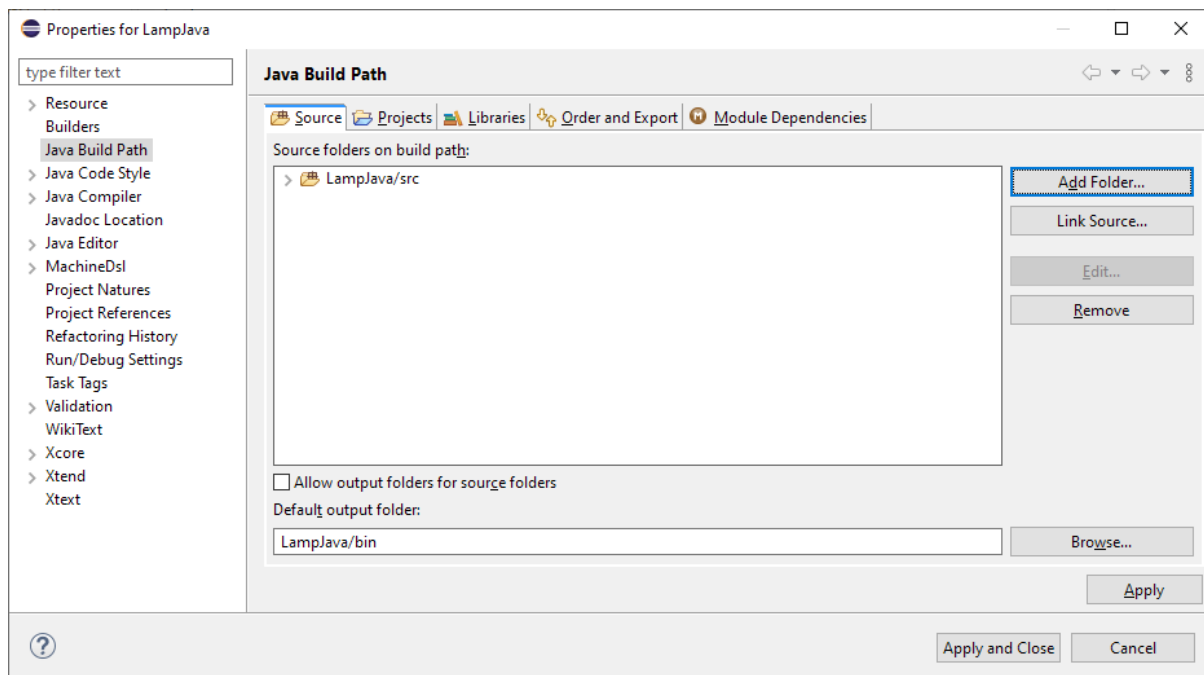
Kattintsunk jobb gombbal a **statemachine.dsl** projekten, és válasszuk a **Run As > Eclipse Application** menüpontot!

A megnyíló belső Eclipse-ben készítsünk egy új Java projektet **LampJava** néven (**module-info.java**-t ne hozzunk létre). A projekten belül készítsünk egy **statemachines** nevű könyvtárat, és ebben a könyvtárban hozzuk létre a **Lamp.sm** fájlunkat a 2. fejezetben bemutatott tartalommal! Ahogy elmentjük a fájlt, automatikusan létrejön az **src-gen/statemachines** könyvtárban a kigenerált **Lamp.java** fájl:

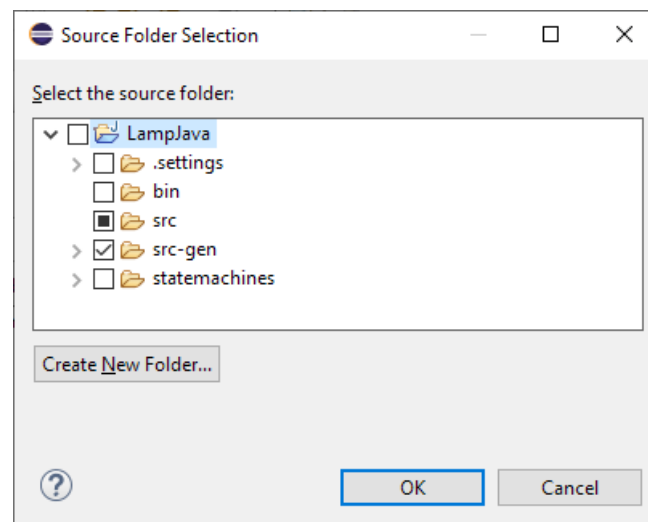


Vizsgáljuk meg és értelmezzük a generált kódot! Az is jól látható, hogy ha az Xtend-et helyesen használjuk, a kimenet szépen formázott lesz.

Ahhoz, hogy ezt a **Lamp.java** fájlt használni tudjuk, az **src-gen** könyvtárat is a Java forráskönyvtárak közé kell tenni. Ehhez kattintsunk jobb gombbal a projekten, és válasszuk a **Properties** menüpontot. A megjelenő ablakban keressük meg a **Java Build Path** oldalt, és a **Source** fület:

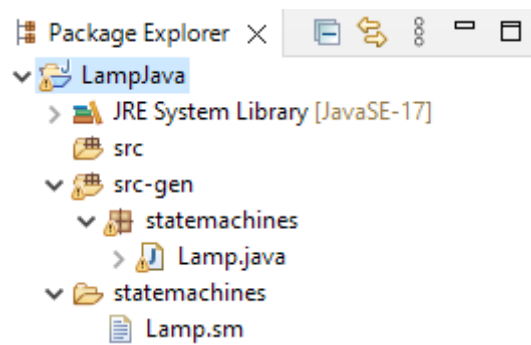


Kattintsunk az **Add Folder...** gombra, és pipáljuk ki az **src-gen** könyvtárat:

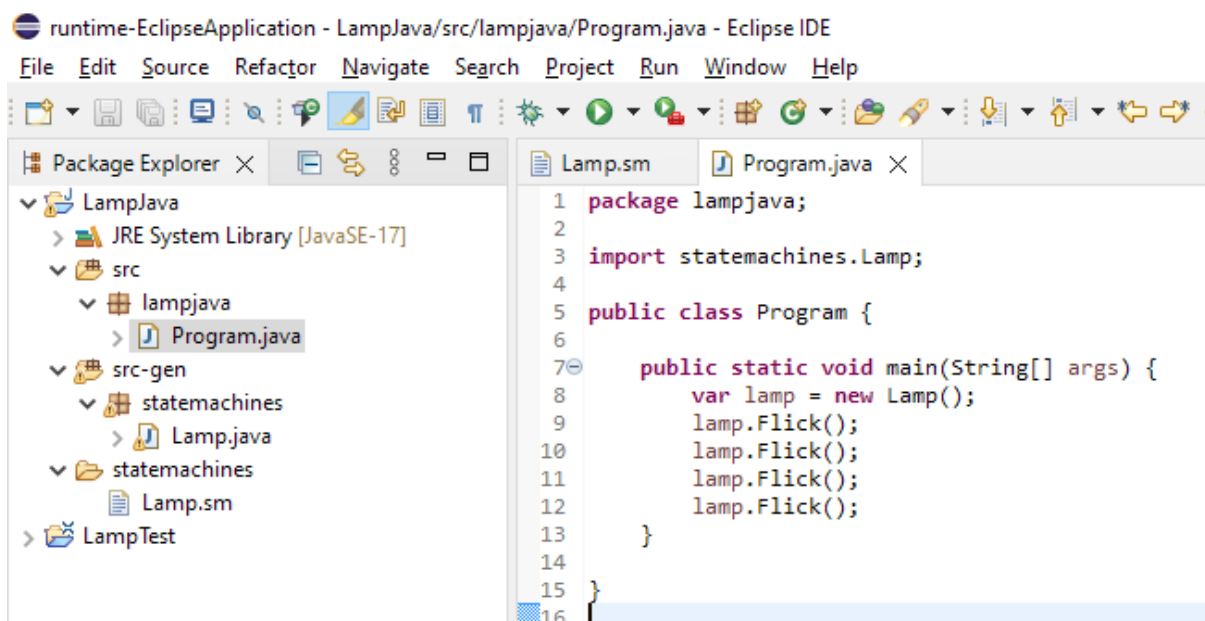


Kattintsunk az **OK** gombra, majd zárjuk be a **Properties** ablakot az **Apply and Close** gombbal!

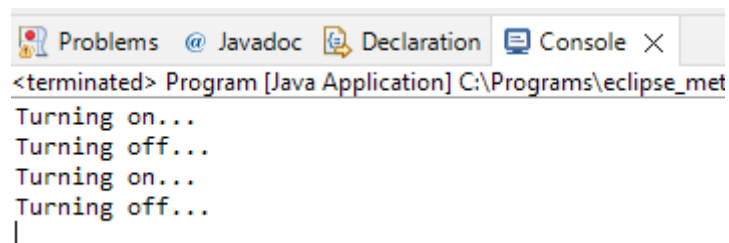
A projektünk most így fog kinézni:



Az **src** könyvtár alatt hozzunk létre egy **Program** osztályt **main** függvénnyel fájl a **lampjava** csomagban a következő tartalommal:



Kattintsunk jobb gombbal a **Program.java** fájlra, és válasszuk a **Run As > Java Application** menüpontot! Ha mindent jól csináltunk, a **Console** ablakban látszik az állapotgépünk futásának eredménye:



Változtassunk a **Lamp.sm** fájlra, mentjük el, és próbáljuk meg újra futtatni az alkalmazást! Figyeljük meg, hogy minden változtatás automatikusan átvezetésre kerül a generált **Lamp.java** fájlba.

11 Összefoglalás

Ez a gyakorlat egy rövid ízelítőt adott abból, hogy az Xcore, Xtext és Xtend használatával hogyan lehet kényelmes IDE támogatást készíteni egy saját szöveges DSL-hez. Számos olyan lehetőség van még, amit nem érintettünk, de a tárgy labor változatában még visszatérünk majd rájuk.

12 Szorgalmi feladatok

Idő hiányában sok dolog kimaradt a gyakorlatból, de aki szeretne jobban elmélyülni a témában, annak itt van néhány ötlet a továbblépéshez.

12.1 További validációk készítése (könnyű)

Vannak olyan nyelvi szabályok, amelyekre még nem készítettük el a validátort. Például, hogy pontosan egy kezdőállapot legyen, vagy hogy állapotonként maximum egy belépési és maximum egy kilépési eseménykezelő lehet. Készítsünk ezekre is validátort!

12.2 Logikai típusú mezők, és if-else utasítások hozzáadása (közepes)

Egészítsük ki a nyelvet úgy, hogy az állapotgépnek lehessenek boolean típusú mezői, akárcsak egy osztálynak. Vegyünk fel értékadás utasítást, amely a boolean típusú változóknak értéket tud adni (true vagy false konstans érték).

Egészítsük ki a nyelvet egy if-else utasítással, amely segítségével egy boolean változó értékétől függően feltételesen is lehet ugrani a következő állapotra.

A feladat megoldásához módosítani kell az Xcore metamodellt, az Xtext fordítót és az Xtend generátort is.

Célszerű továbbá eltörölni azt a nyelvi szabályt, hogy legfeljebb egy jump utasítás szerepelhet egy eseménykezelőben, így töröljük az ehhez tartozó validátort!

12.3 Állapotváltozók és egyéb utasítások, kifejezések hozzáadása (nehéz)

Egészítsük ki a nyelvet úgy, hogy az állapotgépnek lehessenek int típusú mezői. Vegyünk fel értékadás utasítást, és aritmetikai kifejezéseket (összeadás, kivonás, szorzás stb.) is támogassunk!

A feladat megoldásához módosítani kell az Xcore metamodellt, az Xtext fordítót és az Xtend generátort is.

12.4 Függvények hozzáadása (nehéz)

Egészítsük ki a nyelvet úgy, hogy függvényeket is lehessen definiálni az állapotgépen belül, amelyek látják a mezőket, de jump utasítást nem tartalmazhatnak. A függvények meg tudják hívni egymást, és meg lehessen hívni őket az eseménykezelőkből!

A feladat megoldásához módosítani kell az Xcore metamodellt, az Xtext fordítót és az Xtend generátort is.

12.5 Párhuzamos régiók támogatása (nehéz)

Egészítsük ki a nyelvet, hogy a többi UML szabvány által biztosított lehetőséget is támogassa, például a párhuzamos régiókat.