Ingredients

Reading: The manual for the xtend language is available from http://www.eclipse.org/xtend

1 Implementing M2M Transformations in Java

One possible approach to implement M2M transformations would be implementing them in Java. Since our metamodels in EMF have Java implementations, all the persistence code is generated and readily available, we can simply load model data in memory objects, and manipulate them using Java. We can also call Java components from MWE2.

Here is an example of an endo-transforation. It takes a model instance of the Trip model and capitalizes names of all named elements and all vehicles:

```
import java.util.List;

import org.eclipse.emf.mwe2.runtime.workflow.IWorkflowContext;
import org.eclipse.emf.mwe2.runtime.workflow.IWorkflowComponent;

import dk.itu.example.tripLoose.NamedElement;
import dk.itu.example.tripLoose.TripModel;

public class DoCapitalizeNames implements IWorkflowComponent {

    public void invoke(IWorkflowContext ctx) {

        @SuppressWarnings("unchecked")
        List<TripModel> tms = (List<TripModel>) ctx.get("model");
        TripModel tm = tms.get(0);

        for ( NamedElement ne : tm.getElements())
            ne.setName(ne.getName().toUpperCase());
    }

    public void postInvoke() {

    public void preInvoke() {

}
```

Figure 1: Capitalizing name attributes for an instance.

Observe that in the beginning we obtain the model from one of our slots (it must be placed there by some other workflow element). Then we simply iterate over objects implementing the ecore model, introducing the desired changes, just as if these were usual Java implementation objects.

This low-level approach quickly gets complicated. For example complex meta-model graphs give rise to complex, multiple passes over the models.

The transformation, as usual in TMF, is called from a work flow. Here is the work flow that I have written for this purpose.

```
module org.xtext.example.loosetrip.generators.M2M
Workflow {

  bean = org.eclipse.emf.mwe.utils.StandaloneSetup {
    platformUri = "../"
    registerGeneratedEPackage = "dk.itu.example.tripLoose.TripLoosePackage"
    registerGeneratedEPackage = "dk.itu.example.trip.TripPackage"
    registerEcoreFile = "platform:/resource/dk.itu.example.trip/model/tripLoose.ecore"
    registerEcoreFile = "platform:/resource/dk.itu.example.trip/model/trip.ecore"
  }

  component = org.eclipse.xtext.mwe.Reader {
    path = "../org.xtext.example.loosetrip/m2/txt/"
    register = org.xtext.example.loosetrip.LooseTripStandaloneSetup {}

    load = { slot = "model"
             type = "TripModel" }
  }

  component = org.eclipse.emf.mwe.utils.DirectoryCleaner
  {  directory = "../org.xtext.example.loosetrip/model-gen/" }

  component = DoCapitalizeNames { }


  //component = Loose2StrictNaive {}
  //component = Loose2Strict {}

  component = org.eclipse.emf.mwe.utils.Writer {
    modelSlot = "strictmodel"
    // file extension is important here.
    // ".trip" (so the declared xtext syntax extension) saves in textual syntax
    // ".xmi" or ".ecore" saves in xmi syntax.
    uri = "platform:/resource/org.xtext.example.loosetrip/model-gen/strict.xmi"
  }
}
```

The main differences from the workflow presented in the previous lecture is use of the M2M component (DoCapitalizeNames), instead of the xpand generator. Also in the initialization phase we load both metamodels, target and source. For this particular transformation (capitalization), one would be enough, but we need both for the later transformations in this note.

2 Xtend at a Glance

For us, Xtend is a pragmatic language that happens to be used in TMF. Itself it is a reasonable modern OO-language with some extra goodies to make design of DSLs and transformations easier. Here is the characterization from Xtend docs:

Xtend is a statically-typed programming language which is tightly integrated with and runs on the Java Virtual Machine. It has its roots in the Java programming language but improves on a couple of concepts:

- *Advanced Type Inference - You rarely need to write down type signatures*
- *Full support for Java Generics - Including all conformance and conversion rules*
- *Closures - concise syntax for anonymous function expressions*
- *Operator overloading - make your libraries even more expressive*
- *Powerful switch expressions - type based switching with implicit casts*
- *No statements - Everything is an expression*
- *Template expressions - with intelligent white space handling*
- *Extension methods*
- *property access syntax - shorthands for getter and setter access*
- *multiple dispatch aka polymorphic method invocation*
- *translates to Java not bytecode - understand what's going on and use your code for platforms such as Android or GWT*

It is not aiming at replacing Java all together. Therefore its library is a thin layer over the Java Development Kit (JDK) and interacts with Java exactly the same as it interacts with Xtend code. Also Java can call Xtend functions in a completely transparent way. And of course, it provides a modern Eclipse-based IDE closely integrated with the Java Development Tools (JDT).

Package declaration and imports are essentially like in Java. Modulo the the lack of semicolons. And so is the class signature.

The static import makes all static methods of Collections available locally (see the second line of sayHelloTo).

Type inference, like in functional programming languages — return types for functions in the example are inferred (void and string). Everything is an expression. The value of the last expression is returned. So the return statement is also implicit.

Note that we extend a Java class seamlessly (the TestCase class from junit).

Again double colon is used for navigation over types.

This code is translated to the following Java code automatically (the incremental project builder takes care of that):

```
package org.xtext.example.loosetrip.xtend

import junit.framework.Assert
import junit.framework.TestCase

import static extension java.util.Collections.*

class Hello extends TestCase {

    def testHelloWorld() {
        Assert::assertEquals('Hello Joe!', sayHelloTo('Joe'))
        println('Hello Joe!')
    }

    def sayHelloTo(String to) {
        Hello::singletonList(this)
        new Hello().singletonList()
        "Hello "+to+"!"
    }
}
```

```
package org.xtext.example.loosetrip.xtend;

import java.util.Collections;

@SuppressWarnings("all")
public class Hello extends TestCase {

  public String testHelloWorld() {
    String _xblockexpression = null;
    {
      String _sayHelloTo = this.sayHelloTo("Joe");
      Assert.assertEquals("Hello Joe!", _sayHelloTo);
      String _println = InputOutput.<String>println("Hello Joe!");
      _xblockexpression = (_println);
    }
    return _xblockexpression;
  }

  public String sayHelloTo(final String to) {
    String _xblockexpression = null;
    {
      Collections.<org.xtext.example.loosetrip.xtend.Hello>singletonList(this);
      org.xtext.example.loosetrip.xtend.Hello _hello = new org.xtext.example.loosetrip.xtend.Hello();
      Collections.<org.xtext.example.loosetrip.xtend.Hello>singletonList(_hello);
      String _operator_plus = StringExtensions.operator_plus("Hello ", to);
      String _operator_plus_1 = StringExtensions.operator_plus(_operator_plus, "!");
      _xblockexpression = (_operator_plus_1);
    }
    return _xblockexpression;
  }
}
```

Note the inferred types. Most importantly note that this class becomes a Java class that can be used from java, or from anything else, that expects a Java class, seamlessly. In this example you can run it with the junit test case execution mechanism of Eclipse.

You can find the code in the xtend-gen directory. It is occasionally useful to look in there, if you get some weird type errors. Most often you will be able to understand them at the Java level.

Default class visibility is public, and fields are always private.

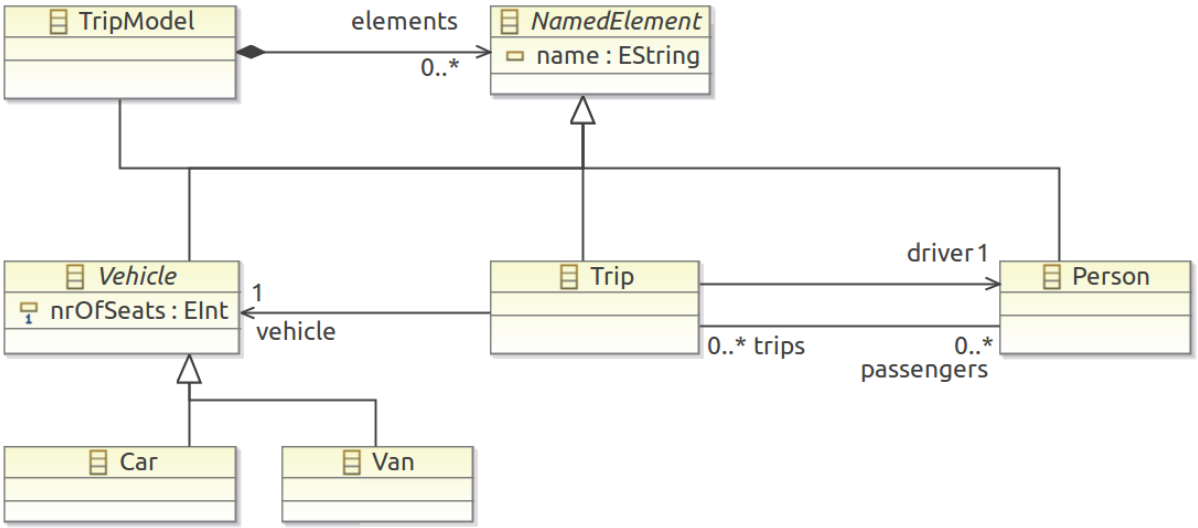

Figure 2: Abstract syntax (the meta-model) of the trip language

You do not need to declare rethrown exceptions. This is done automatically.

Local variables are introduced with the `val` keyword. Examples in the next section.

We will demonstrate the other features of the language trying to implement a simple M2M transformation.

3 Implementing M2M Tx in Xtend

Figure 2 shows the metamodel of our trip language.

We have created a similar metamodel, tripLoose, which only differs from the above, by not requiring that the passengers and trips are opposite references.

We want to write a transformation which reads in a tripLoose instance, and turns it into a trip instance. This is very simple: every object of type `T` in `tripLoose`, needs to be copied into an object of type `T` in `trip`. From type system's point of view these two types are different, even though the contents of objects is the same. On top of that we need to populate the reverse reference Person.trips (in principle we should also populate Trip.passengers but we skip that to keep the code simpler).

Here is the naive way of implementing this in xtend:

```
package org.xtext.example.loosetrip.generators
import dk.itu.example.trip.*
import dk.itu.example.trip.impl.TripFactoryImpl
import org.eclipse.emf.mwe2.runtime.workflow.IWorkflowContext
```

```
class Loose2StrictNaive extends WorkflowComponentWithSlot {

  java.util.List<Object> models
  dk.itu.example.tripLoose.TripModel model
  TripModel strictModel
  TripFactory strictFactory

  def dispatch void addElement (TripModel m, dk.itu.example.tripLoose.Person p) {
    val sp = strictFactory.createPerson();
    sp.name = p.name
    m.elements.add(sp)
  }

  def dispatch void addElement (TripModel m, dk.itu.example.tripLoose.Trip t) {
    val st = strictFactory.createTrip ();
    st.name = t.name
    m.elements.add(st)
  }

  (...)

  def dispatch Boolean namedTrip (Trip t, String n) { t.name == n }
  def dispatch Boolean namedTrip (Person t, String n) { false }
  def dispatch Boolean namedPerson (Trip t, String n) { false }
  def dispatch Boolean namedPerson (Person p, String n) { p.name == n }

  def dispatch void addAttr (TripModel m, dk.itu.example.tripLoose.Person p) {}
  def dispatch void addAttr (TripModel m, dk.itu.example.tripLoose.Trip t) {
    val Trip st = m.elements.findFirst(e | namedTrip(e,t.name)) as Trip
    for (p : t.passengers) {
      val String pname = p.name // clunky compiler it seems ...
      val Person sp = m.elements.findFirst (e | namedPerson(e,pname)) as Person
      st.passengers.add(sp) }
  }
  override void invoke(IWorkflowContext ctx) {
    strictFactory = new TripFactoryImpl ()
    strictModel = strictFactory.createTripModel ()
    models = (ctx.get ("model")) as java.util.List
    model = models.get(0) as dk.itu.example.tripLoose.TripModel
    strictModel.name = model.name

    for (e: model.elements) addElement (strictModel, e)
    for (e: model.elements) addAttr (strictModel, e)
    ctx.put ("strictmodel", strictModel)
  }
}
```

A *dispatch* is a special kind of method that is bound based on the actual type of the objects in actual

parameters at runtime. So a dispatch allows you to write pattern matching on types, like in functional programming languages. A few simple examples are found in the above code.

Also note the use of anonymous functions (closures), which have a syntax akin to comprehended set expressions, making it possible to write set iterations in style very close to OCL.

Start reading the code with the invoke method (which will be called by the same kind of work flow as shown for capitalization example).

The code first obtains a factory for creating trip objects, and creates the root instance — TripModel. Then it obtains the input model from the context. We can, in principle, get a list of models, but for simplicity of the example we will only transform the first one found. This is why we only look at the first element of the list.

Then we do the work in two passes: first, we recreate all objects (named elements). In the second pass we create links between this objects using the addAttr function. This seems necessary, because we cannot create the links, before all objects are instantiated in the trip language (some links would have to be dangling otherwise).

Since M2M transformations are often this kind of graph rewriting, xtend offers a feature (create methods) that uses implicit memoisation, so that you can create your graphs in one pass.

A create function is a function that means to create an object, but it does this in two phases (each one represented by one body) . It first creates the object (the first body is a factory). Then it initializes the object (second body). If the object is created the first time it is cached (the ids of parameters being the key).

If in the initialization phase the object is created recursively again, a reference is simply returned from the cache. This allows to avoid the messy restoration of links, if you build you graphs in some kind of depth-first-search manner.

Here is an example of one such method:

```
def Trip create st: strictFactory.createTrip()
  copyElement(dk.itu.example.tripLoose.Trip t) {
    st.name = t.name
    st.vehicle = copyVehicle (t.vehicle)
    st.driver = copyElement (t.driver)
    st.passengers.addAll(
       t.passengers.map(p | copyElement (p)))
}
```

The important element is that when we get to the creation of the vehicle (or the driver), we can safely create the object since copyVehicle (and copyElement) are create methods, thus they will create the object only once. If the object is to be created again later (because we will see its declaration later, only after the trip declaration), we will simply operate on the copy created before.

Figure 3 shows the complete xtend code of a more concise version of this M2M transformation.

Note the syntax of type casting, using the `as` keyword, unlike in Java.

This code is to be called with the same work flow as the two previous transformation (just change the transformation component in it).

```
package org.xtext.example.loosetrip.generators
import dk.itu.example.trip.*
import dk.itu.example.trip.impl.TripFactoryImpl
import org.eclipse.emf.mwe2.runtime.workflow.IWorkflowContext

class Loose2Strict extends WorkflowComponentWithSlot {

    TripFactory strictFactory
    // there should me a more concise way of testing adherance to an interface
    // I could not figure out, though.
    def dispatch Boolean isTrip (dk.itu.example.tripLoose.Trip t) {true}
    def dispatch Boolean isTrip (dk.itu.example.tripLoose.Person t) {false}

    def Person create sp: strictFactory.createPerson()
        copyElement(dk.itu.example.tripLoose.Person p) { sp.name = p.name }

    def Vehicle create sv: strictFactory.createCar()
        copyVehicle(dk.itu.example.tripLoose.Vehicle v) {
            sv.name = v.name
            sv.nr0fSeats = v.nr0fSeats
        }

    def Trip create st: strictFactory.createTrip()
         copyElement(dk.itu.example.tripLoose.Trip t) {
            st.name = t.name
            st.vehicle = copyVehicle (t.vehicle)
            st.driver = copyElement (t.driver)
            st.passengers.addAll(t.passengers.map(p | copyElement (p)))
        }

    override void invoke(IWorkflowContext ctx) {
        strictFactory = new TripFactoryImpl ()
        val strictModel = strictFactory.createTripModel ()
        val models = (ctx.get ("model")) as java.util.List<Object>
        val model = models.get(0) as dk.itu.example.tripLoose.TripModel
        strictModel.name = model.name

        for(v: model.vehicles) { strictModel.vehicles.add (copyVehicle(v)) }
        for(e: model.elements.filter (e | isTrip(e)))
            { strictModel.elements.add (copyElement(e as dk.itu.example.tripLoose.Trip)) }
        for(e: model.elements.filter (e | !isTrip(e)))
            { strictModel.elements.add (copyElement(e as dk.itu.example.tripLoose.Person)) }
        ctx.put ("strictmodel", strictModel)
    }
}
```

Figure 3: An example of an M2M transformation: convert instances of TripLoose to strict Trip.

4 Odds and Ends

While Xtend seems to be still a fairly low level language to implement M2M transformations in, there are opinions on the market that this is the most practical one. It strikes the balance between usability (effectiveness) and efficiency. There are rumors that tools based on graph transformation are much less efficient.

References

[1] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches, 2006.