A Domain Specific Language with oAW and Xtext

Implementation and Integration of a Domain Specific Language with oAW and Xtext

by Volker Koster

Implementation and Integration of a Domain Specific Language with oAW and Xtext

by
Volker Koster

Abstract

This article is about textual Domain Specific Languages (DSLs) and Language Workbenches. The level is introductory.
The DSL we will develop here is based on an example taken from Martin Fowler's article "Language Workbenches: The Killer-App for Domain Specific Languages?".
Implementation and Eclipse integration of this DSL will be demonstrated using the Xtext component of the openArchitectureWare (oAW) framework.

The article is structured into three parts.

Part I summarises the exemplary problem domain described by Marin Fowler and gives an overview of a rudimentary manual Java implementation that serves as a target for our code generation efforts.

In Part II we will concern ourselves with the extraction and implementation of a DSL for our little problem. Using oAW, we will then generate and customize a specific text editor for this DSL and see how it smoothly integrates into the Eclipse IDE.
Now that we have an editor that "speaks" our language, we will use it to build a model that conforms to our manually coded software solution.

In Part III we will generate code from this model targeting the hand written code of Part I. We will finish by summing up the advantages gained from these efforts and give some further reading tips.

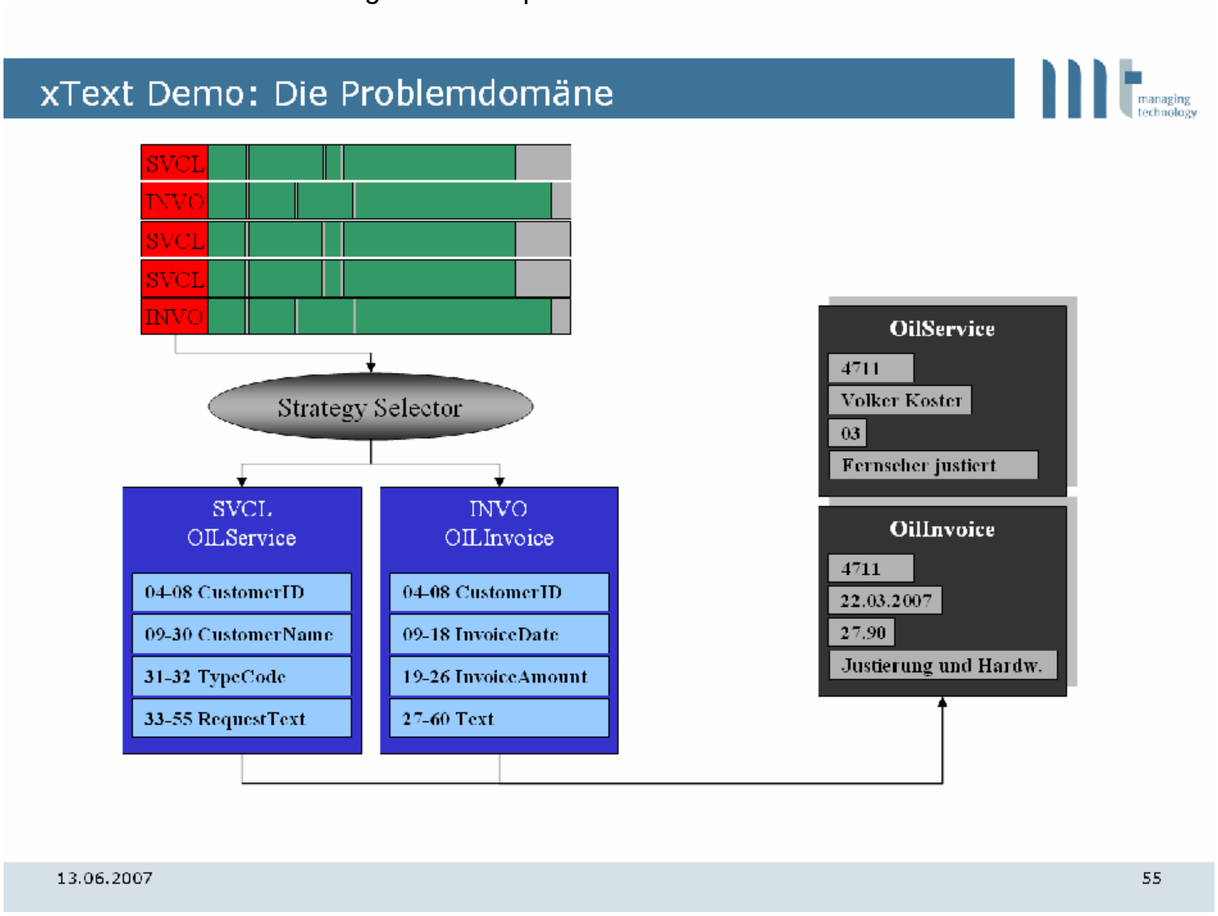
Part I: Problem Domain, Abstraction and Configuration

In case this is your first contact with the term "Domain Specific Language", I suggest you read Martin Fowlers article "Language Workbenches: The Killer-App for Domain Specific Languages?" first. If you are short on time, at least work through the introductory example. I found that to be a perfect introduction on just a couple of pages. You will find the article here:http://www.martinfowler.com/articles/languageWorkbench.html#ASimpleExampleOfLanguageOrientedProgramming.

In short, what we have to deal with in this example is a structured flat file that we have to process in a certain manner.
The file is structured into lines of data.
A text line is of a certain type, identified by the first 4 characters of the line.
Lines of the same type are all structured the same way, which means that relevant data for a given type of line can always be found at fixed positions within that line.
We will process the text file by instantiating an object for each line of text associating a line type with a Java class. The relevant data of each line has to be injected into the properties of the corresponding Java object.

That's about it. If you like, you can imagine yourself as being responsible for processing invoices from different customers all bundled up as individual lines of text in a single flat file. Of course, each customer has his own invoice structure that you have to deal with. The knowledge of the structures of a given invoice is captured in your customer-specific invoice Java class. So now the problem comes down to instantiating the appropriate invoice Java class for each line and to populate its properties with the line's data. Our implementation will follow the one given by Martin, except we are coding in Java here.

Picture 1 describes the design of this implementation.



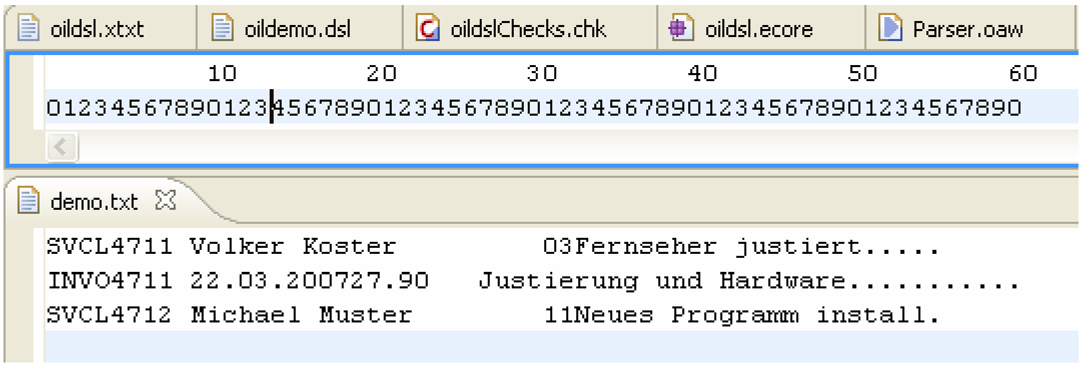
Picture 1: Designing the Solution

The text file is processed by reading one line at a time. For any given line, we examine the first 4 characters, also called 'type code' from now on, to identify the type of the line we are dealing with. Depending on its type code, we delegate the line string to an appropriate Strategy object for line processing.
The Strategy knows exactly how to deal with the given line type. It creates an object of the Java class that is associated with the line's type code. We will call this Java class "Target Class" for short. Line processing knowledge is encapsulated in a list of "Extractors" held by the Strategy.
An Extractor is responsible for extracting a well defined amount of data from an equally well known position within the text line and assigning this data to its associated property of the new born target object.

Just for the sake of being able to display some results, we collect the target objects in a list that we simply print out to the console in the end.

Whether you like this implementation or not, the crucial point here, as Martin Fowler pointed out, is the separation of our solution into code representing the abstraction of the problem domain and into code that simply configures this abstraction. This separation into an abstraction and a configuration is the key to DSL development. We will talk some more about this later on.

In order to get started, we need an example text file that we want to process (see Picture 2)



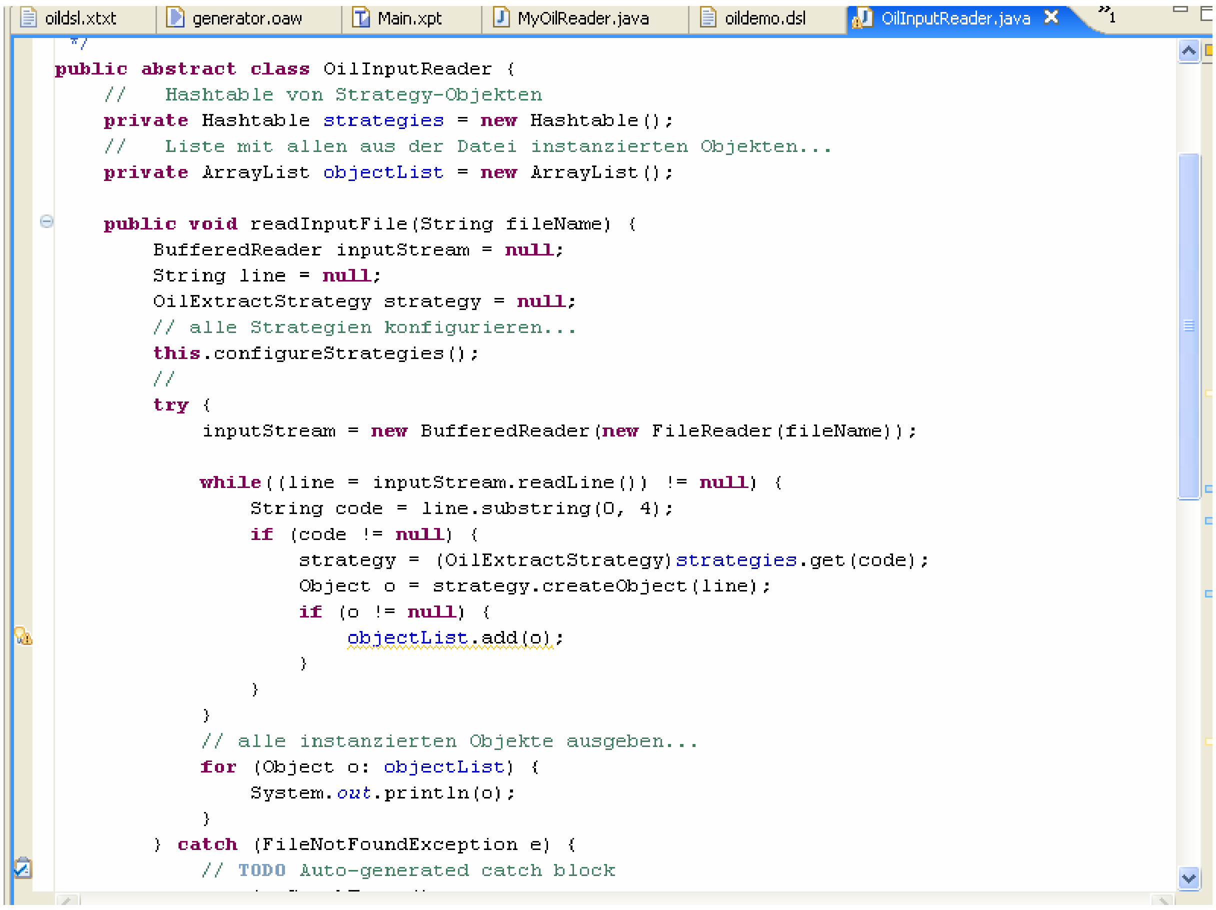
```
                10        20        30        40        50        60
0123456789012345678901234567890123456789012345678901234567890
```

```
SVCL4711 Volker Koster          03Fernseher justiert.....
INVO4711 22.03.200727.90    Justierung und Hardware..........
SVCL4712 Michael Muster         11Neues Programm install.
```

Picture 2: Example Text File

The content of the file is in German. Don't worry about that, for this is of no importance. Let's see how to deal with this text by looking at some code fragments.

Picture 3 shows the main processing loop iterating over all the lines of our file.



```
 */
public abstract class OilInputReader {
    //   Hashtable von Strategy-Objekten
    private Hashtable strategies = new Hashtable();
    //   Liste mit allen aus der Datei instanzierten Objekten...
    private ArrayList objectList = new ArrayList();

    public void readInputFile(String fileName) {
        BufferedReader inputStream = null;
        String line = null;
        OilExtractStrategy strategy = null;
        // alle Strategien konfigurieren...
        this.configureStrategies();
        //
        try {
            inputStream = new BufferedReader(new FileReader(fileName));

            while((line = inputStream.readLine()) != null) {
                String code = line.substring(0, 4);
                if (code != null) {
                    strategy = (OilExtractStrategy)strategies.get(code);
                    Object o = strategy.createObject(line);
                    if (o != null) {
                        objectList.add(o);
                    }
                }
            }
            // alle instanzierten Objekte ausgeben...
            for (Object o: objectList) {
                System.out.println(o);
            }
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
```

Picture 3: Main Processing Loop

(I called the Eclipse project “OILLanguage”, where OIL is short for “Object Instantiation Language”)

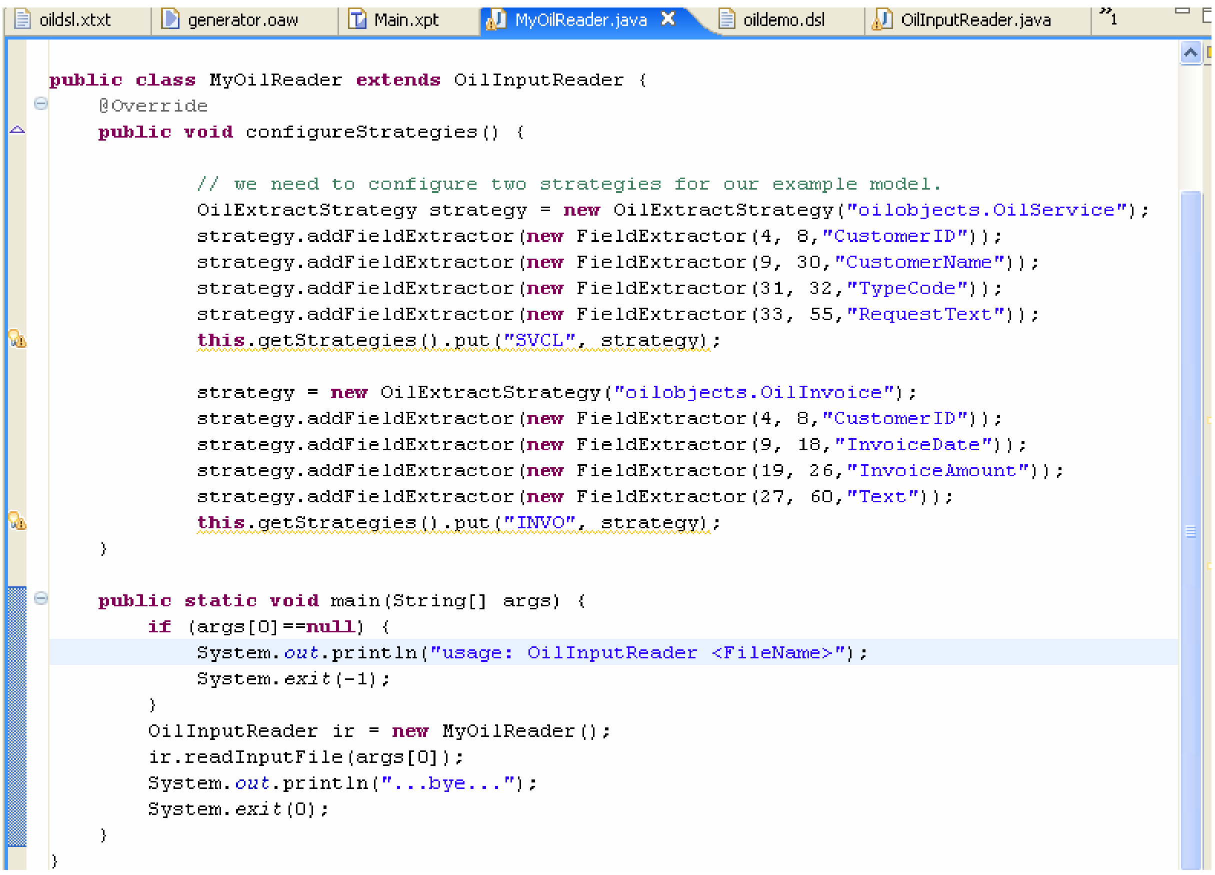
As you can see, we read a line, choose the right strategy from a hash table based on the line’s type code and hand further processing to the Strategy object. The resulting target object is simply added to our output list.

The concepts of a Strategy and its associated Field Extractors are the cornerstones of our Abstraction. They work together in the process described above, to process any given text file. But in order to actually process a given file, this abstraction has to configured, according to the files structure. Every line type needs exactly one Strategy and every Strategy has to be carefully configured with the name of the class to instantiate and with exactly one Extractor for each data field to be extracted from that line. Again, every Extractor has to be configured with exact position of the data to be extracted and with the name of the target class property that will receive the data.

Different file structures will be taken care of, by adjusting the configuration. The abstraction, our concepts, will never be touched.

You probably guessed where this is leading to. Our goal will be, to design a language for describing a given file structure and simply generate the corresponding configuration code from these descriptions. But let’s do this step by step.
To make the distinction between abstraction and configuration a bit more prominent in our code, I choose to make the OilInputReader abstract, so that the configuration code can be further isolated.

Take a look at its concrete implementation in Picture 4:



```
oildsl.xtxt   generator.oaw   Main.xpt   MyOilReader.java   oildemo.dsl   OilInputReader.java
```

```java
public class MyOilReader extends OilInputReader {
    @Override
    public void configureStrategies() {

            // we need to configure two strategies for our example model.
            OilExtractStrategy strategy = new OilExtractStrategy("oilobjects.OilService");
            strategy.addFieldExtractor(new FieldExtractor(4, 8,"CustomerID"));
            strategy.addFieldExtractor(new FieldExtractor(9, 30,"CustomerName"));
            strategy.addFieldExtractor(new FieldExtractor(31, 32,"TypeCode"));
            strategy.addFieldExtractor(new FieldExtractor(33, 55,"RequestText"));
            this.getStrategies().put("SVCL", strategy);

            strategy = new OilExtractStrategy("oilobjects.OilInvoice");
            strategy.addFieldExtractor(new FieldExtractor(4, 8,"CustomerID"));
            strategy.addFieldExtractor(new FieldExtractor(9, 18,"InvoiceDate"));
            strategy.addFieldExtractor(new FieldExtractor(19, 26,"InvoiceAmount"));
            strategy.addFieldExtractor(new FieldExtractor(27, 60,"Text"));
            this.getStrategies().put("INVO", strategy);
    }

    public static void main(String[] args) {
        if (args[0]==null) {
            System.out.println("usage: OilInputReader <FileName>");
            System.exit(-1);
        }
        OilInputReader ir = new MyOilReader();
        ir.readInputFile(args[0]);
        System.out.println("...bye...");
        System.exit(0);
    }
}
```

Picture 4: Configuring our Abstraction

Just take a moment to verify, that this configuration code corresponds to the properties of our little demo text file from above.
We have two types of lines here, identified by the type code “SVCL” and “INVO”, respectively.
Accordingly, we have to configure our abstraction with two instances of the strategy class OilExtractStrategy.
Each strategy instance is configured with the target class to instantiate and a list of extractor instances. The extractor class is called FieldExtractor.
A FieldExtractor again is configured with the ‘from’ and ‘to’ position of the data to be extracted and the name of the property that will hold this data within the target class.

That’s about all there is.
We now have a configuration language that we can call a DSL in all rights.
I do call it a language because it possesses both, structure and semantics. The structure is defined by the method signatures, the semantics are defined by the code itself. A domain expert, even one with little programming experience, can easily use this language to configure a new strategy in order to provide for a new line type in the text file. From a domain expert’s point of view, this high level language is a significant raise of the level of abstraction as far as dealing with the domain is concerned. From now on, we can discuss this domain in terms of Strategies and Extractors.

We discovered this little language by separating our abstraction from its configuration. You can as well take this as a rule of thumb. In search for a DSL associated with an abstraction, the first places to look at are constructors, factories and configuration files.
Next, try to distinguish between dynamic and stable parts of the system. The abstraction is the part of the problem domain that is likely to be quite stable, whereas the configuration is actually supposed to change.
Martin explains this (and more) in much greater detail, so again, don’t miss his article.

As I tried to point out, the configuration is actually supposed to change and, luckily, we isolated its code into one single method. Nevertheless, we have to modify our code whenever the configuration needs to be updated. Maybe the structure of a line type changes (your customer changes the structure of his invoices) or new line types come up (you found a new customer).
On top of that, we will only be able to discover (or should I say ‘experience’) configuration errors at runtime. Configuration syntax will be checked by your Java IDE but things like accidentally switching ‘from’ and ‘to’ position or assigning different data fragments of a line to the same property (this actually happened to me as a result of a copying & pasting) will only come up at runtime.

As you can see, the problem with our configuration DSL is that it is embedded in our code. It is an internal DSL.

In the next part of this article, we will extract the language from the code and make it an external language in its own right.
We will base it on a grammar, introduce some keywords, define a syntax and generate a plug-in editor for Eclipse so we can “speak” our language with some decent IDE support.

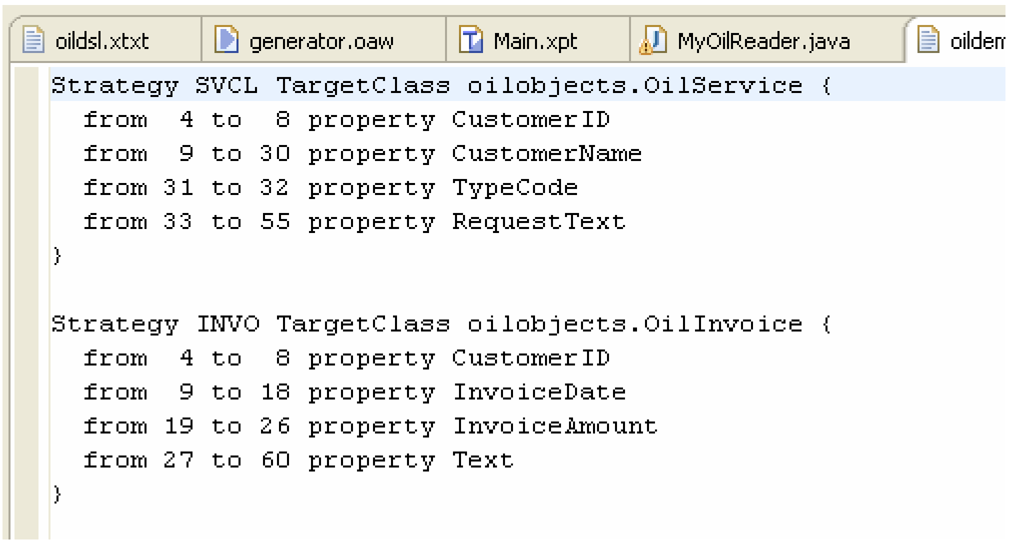
Using the OpenArchitectureWare framework (oAW) we will accomplish this with (in my opinion) minimal effort.

Part II: Externalizing the DSL

We will now externalize our little language, generate a text editor to support it and plug the editor into our Eclipse Workbench. This would probably be a non-trivial task, could we not rely on the oAW framework, especially on its Xtext component.

You will see a couple of screenshots showing oAW in action in the following pages, but we will take on more of a user's point of view, not diving too deep into oAW's details here.
A click-by-click tutorial covering everything presented here in much more detail is in progress and will be available on our website soon. Of course you will find the link to the oAW project in the 'Further Reading' section at the end of this article.

Let's start with a look at the language as it will present itself in the end (Picture 5).



```
Strategy SVCL TargetClass oilobjects.OilService {
  from  4 to  8 property CustomerID
  from  9 to 30 property CustomerName
  from 31 to 32 property TypeCode
  from 33 to 55 property RequestText
}

Strategy INVO TargetClass oilobjects.OilInvoice {
  from  4 to  8 property CustomerID
  from  9 to 18 property InvoiceDate
  from 19 to 26 property InvoiceAmount
  from 27 to 60 property Text
}
```

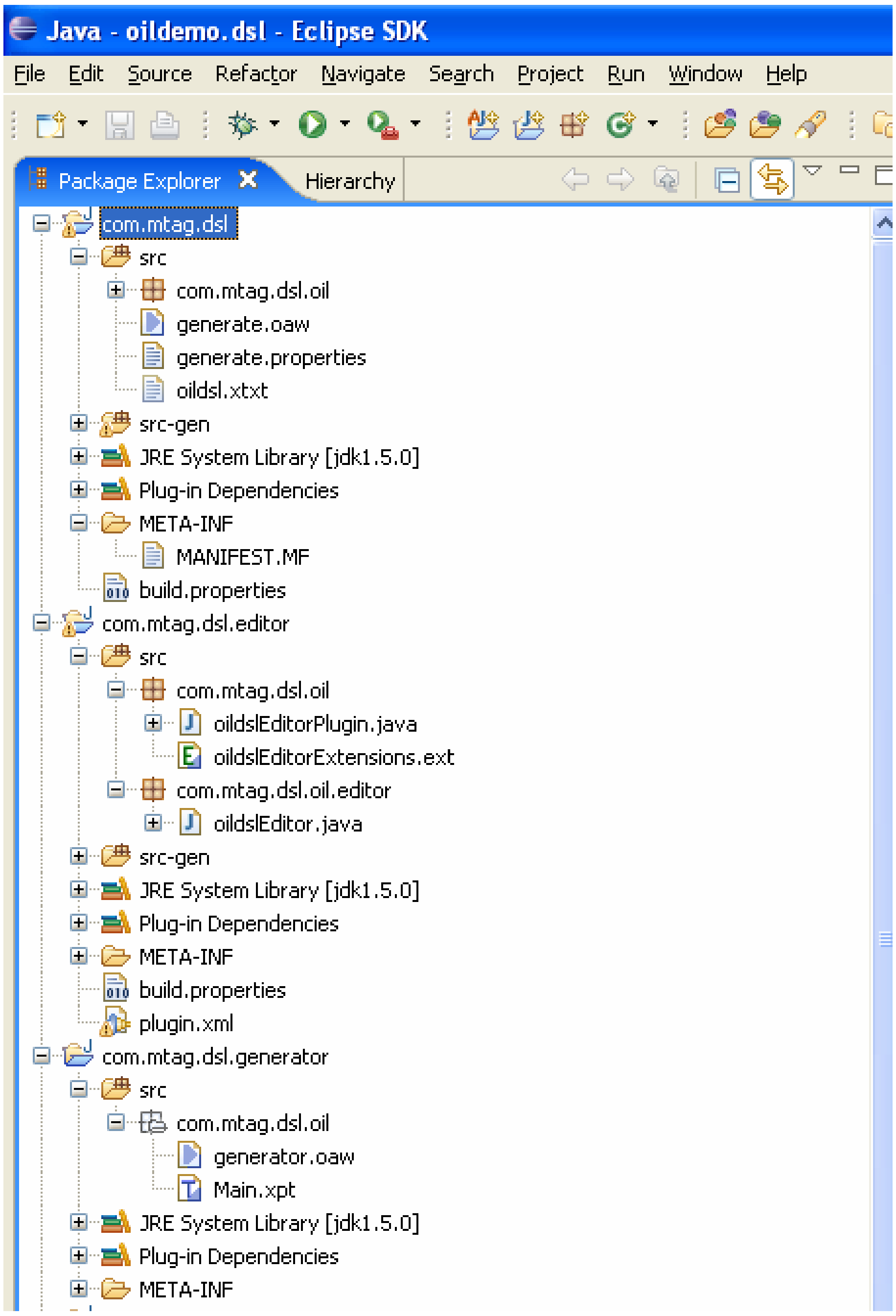
Picture 5: A view of our configuration language (OIL)

You will recognize all the ingredients that made up our configuration code.
Our top level configuration unit is called a Strategy. Well will make 'Strategy' the first keyword of our language.
The keyword Strategy is followed by the type code identifying a certain line type. Like in our handwritten code, we configure two strategies with the type codes 'SVLC' and 'INVO', respectively.
After the type code, we introduce our next keyword 'TargetClass' followed by the fully qualified name of the Java class to be instantiated by this Strategy.
A Strategy contains a number of extractors bundled together in curly braces.
An Extractor consist of the keywords 'from' and 'to', each followed by the appropriate position of the data to be extracted. An Extractor concludes with the keyword 'property' followed by the name of the property the data will be assigned to.

Of course this is only one out of many possible representations for our language and I admit to probably being over the top with the introduction of keywords. The language could have been more concise, but those keywords will help us to show the editor's capabilities like keyword completion, syntax highlighting and error checking later on.

Now that we know what our language should look like in the end, let's get our hands dirty with a little bit of oAW and Xtext.

Xtext is oAW's component for designing and implementing textual DSLs.
It comes along with its own project wizard called 'Xtext projects' in Eclipse.
The wizard creates three projects that we will call the 'dsl project', the 'editor project' and the 'generator project', respectively. All of these are Eclipse plug-in projects already configured with the plug-in dependencies to the oAW framework you need.
Picture 6 shows my workspace after the wizard performed its magic.

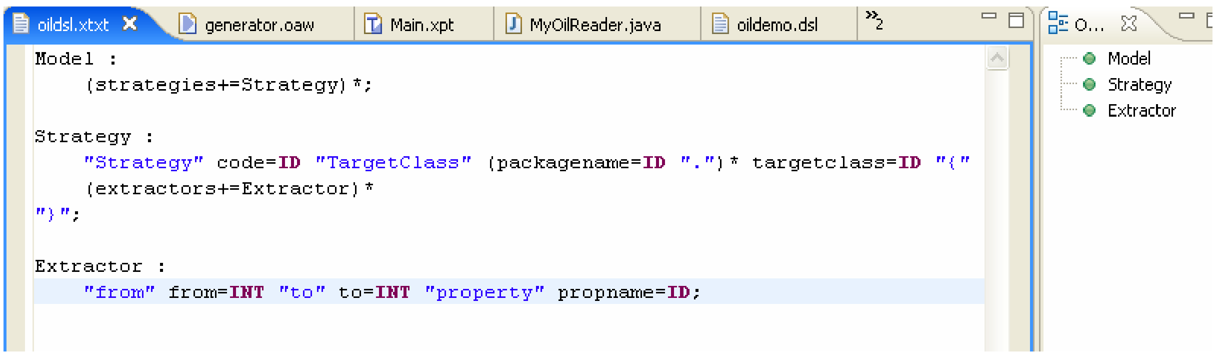

Java - oildemo.dsl - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Hierarchy

```
com.mtag.dsl
    src
        com.mtag.dsl.oil
        generate.oaw
        generate.properties
        oildsl.xtxt
    src-gen
    JRE System Library [jdk1.5.0]
    Plug-in Dependencies
    META-INF
        MANIFEST.MF
    build.properties
com.mtag.dsl.editor
    src
        com.mtag.dsl.oil
            oildslEditorPlugin.java
            oildslEditorExtensions.ext
        com.mtag.dsl.oil.editor
            oildslEditor.java
    src-gen
    JRE System Library [jdk1.5.0]
    Plug-in Dependencies
    META-INF
    build.properties
    plugin.xml
com.mtag.dsl.generator
    src
        com.mtag.dsl.oil
            generator.oaw
            Main.xpt
    JRE System Library [jdk1.5.0]
    Plug-in Dependencies
    META-INF
```

Picture 6: Post-Wizard Workspace

The first project we need to work on is the dsl-project.
The dsl project is for developing a grammar for our language.
Xtext supports us with a grammar editor that recognizes grammar files by their '.xtxt' extension.
In Xtext a grammar is specified in an Extendend Backus Naur Form (EBNF). No reason to stop breathing, for we will only have to deal with very few concepts here (of course, if you like, everything is explained in great detail in your old Niklaus Wirth Compiler Construction Book or on the ANTLR Language Recognition Project by Terence Parr. See the 'Further Reading' section for details).

The Xtext project wizard already created an empty grammar file for us, called oildsl.xtxt, based on the information we entered into the wizard.
Xtext recognizes this as a grammar file, so that Eclipse uses the Xtext grammar editor when opening the file.
The next picture shows the grammar behind our language displayed in the Xtext editor.
Note that the editing process is supported by syntax highlighting, keyword completion and outline view just like we would have it for our own language.



```
Model :
    (strategies+=Strategy)*;

Strategy :
    "Strategy" code=ID "TargetClass" (packagename=ID ".")* targetclass=ID "{"
    (extractors+=Extractor)*
"}";

Extractor :
    "from" from=INT "to" to=INT "property" propname=ID;
```

Picture 7: Our grammar displayed in the Xtext editor

As nice and helpful as the editor is, you are the one responsible for writing a sound grammar for your language. So let's take a closer look at the text above.

Explaining a grammar may be a little technical, but this is the heart of the matter, so any effort to get a grasp on it will dearly pay off.
I will try to explain this from two different angles and, depending on your IT background, you may prefer one explanation to the other. Just read them both and work through the one that suits you more.

An Xtext grammar is structured into rules. These are identified by the text left of the colon.
As you can see, we have a Model rule, a Strategy rule and an Extractor rule.
The Model rule consists of zero or more Strategy rules, indicated by the notation '(…)*', where the '*' represents 'zero or more'.
All the Strategy rules are assigned to a list called 'strategies', indicated by the assignment operator '+='.

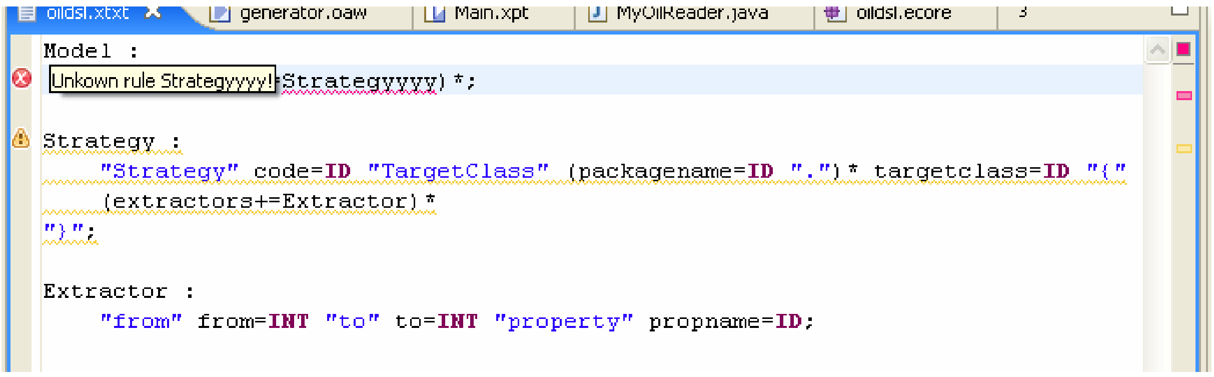
The next rule explains what a Strategy is. It starts with the keyword “Strategy” followed by an ID. ID is one of Xtext’s built-in keywords and is defined quite similar to an identifier in Java. In order to be able to address this ID later on, we give it a name and that name is ‘code’.
We introduce the next keyword of the language and call it ‘TargetClass’.
Following this keyword we want to specify the fully qualified name of the class to be instantiated by this strategy.
The pattern (packagename=ID “.”)* indicates our way of specifying a package name. We would like to be able to repeat the pattern packagename-dot zero or more times. Then we finish with the name of the class itself (this is not good enough for professional purposes, because, for example, we allow for spaces left and right of the dot, but let’s say its good enough for this example).
The next expression, ‘(extractors+=Extractor)*’, means that a Strategy maintains a list of zero or more Extractors and that this list can be addressed as ‘extractors’.
An Extractor is defined as a rule of its own.
To bundle all the Extractors of a given Strategy visually together, we put them in curly braces.
Our grammar closes with the definition of the Extractor rule.
It consists for the keywords ‘from’ and ‘to’, which are both followed by integers that can be addressed as ‘from’ and ‘to’, respectively. This is where we specify the position of the data to be extracted from a given line of text.
The final keyword of our language is ‘property’. It is followed by an identifier denoting the property of the target class we want to assign the extracted data to.

Here is the second approach to understanding an Xtext grammar.
This approach will be more appealing to you, if you already gathered some experience with the Eclipse Modelling Framework.
The oAW framework is based on the Eclipse Modelling Framework (EMF), which again, is based on the eCore meta-modelling language.
ECore Meta-Models constitute the abstract syntax for our models.
As we will see later on, there is a tight association between an Xtext grammar and its corresponding eCore model. In fact, the eCore model will be generated from the Xtext grammar.
If you are familiar with the EMF and eCore models, you will probably find an Xtext grammar easier to understand, by focusing on the model that will be generated based on your grammar.

Every grammar rule corresponds to a meta-type of the generated eCore meta-model. The type is named after the rule identifier left of the colon.
In our example, Model, Strategy and Extractor are the meta-types of our meta-model.
Model is our top level container. It will contain an attribute called ‘strategies’, which is a list containing zero or more model elements of type Strategy.
A Strategy contains the attributes ‘code’, ‘packagename’ and ‘tragetclass’. It also contains a reference to zero or more model elements of the Extractor type.

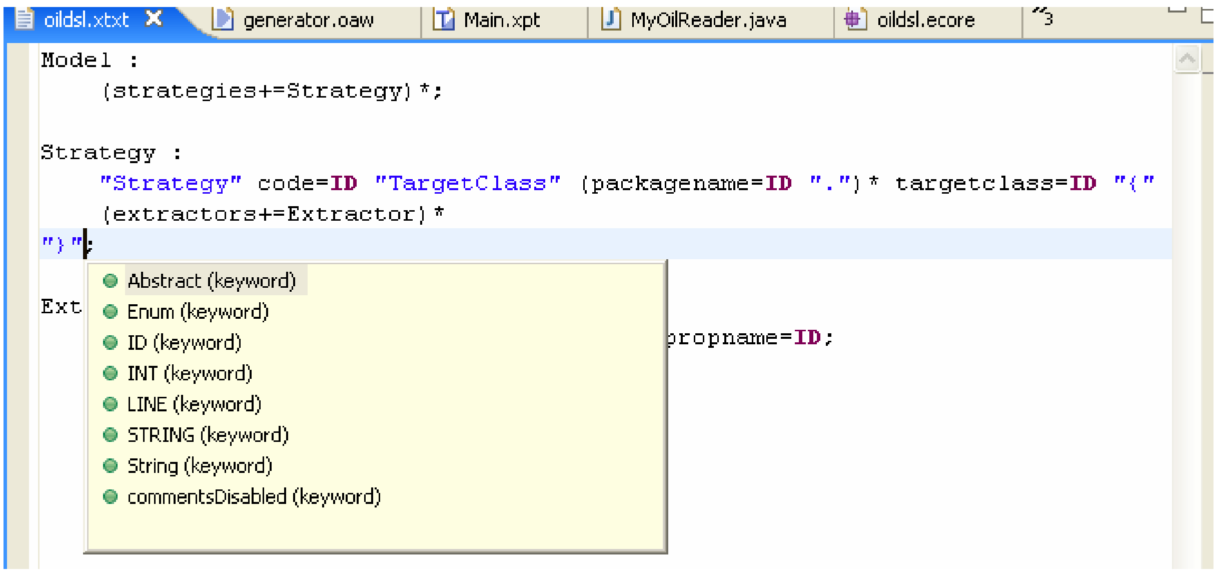
Extractor is our last meta-type. It has the properties ‘from’, ‘to’, and ‘propname’, holding information regarding data extraction positions and the data target property, respectively.

Whatever view of a grammar suits you better, the features of the Xtext grammar editor will give you extra support while editing your grammar.
The next two pictures show the editor in action with its syntax checking capability and its support for keyword completion and keyword dropdown lists.



```
Model :
Unkown rule Strategyyyy! Strategyyyy)*;

Strategy :
     "Strategy" code=ID "TargetClass" (packagename=ID ".")* targetclass=ID "{"
     (extractors+=Extractor)*
"}";

Extractor :
     "from" from=INT "to" to=INT "property" propname=ID;
```

Picture 8: Syntax checking and error indication



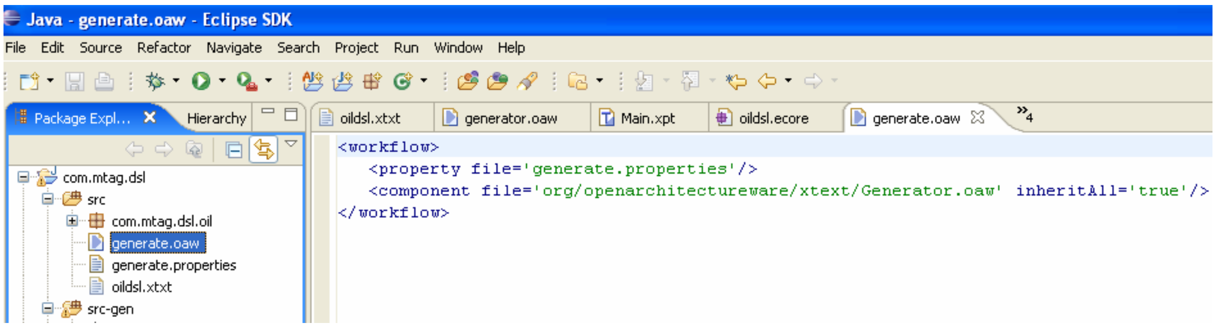
```
Model :
     (strategies+=Strategy)*;

Strategy :
     "Strategy" code=ID "TargetClass" (packagename=ID ".")* targetclass=ID "{"
     (extractors+=Extractor)*
"}";

Ext                                                  propname=ID;
```

Picture 9: Keyword dropdown list

There's a lot more to Xtext grammars than we made use of in this example. You will find all the details about Xtext grammars in the Xtext reference manual on the oAW homepage. Some Xtext examples are also shipped with the oAW installation.

Ok, we now have a grammar to support our OIL language shown in picture 5.
We will now use this grammar to generate a special text editor for OIL and we want all the support we are used to from other language-specific editors like Java or XML editors. We want syntax highlighting, keyword completion and error checking. We want a smooth integration into our Eclipse IDE.

The first and fully functional version of this editor will be generated by the oAW framework. With our grammar in place, we can make use of another artefact generated by the Xtext project wizard. It is an oAW workflow called generator.oaw, as can be seen in picture 10.



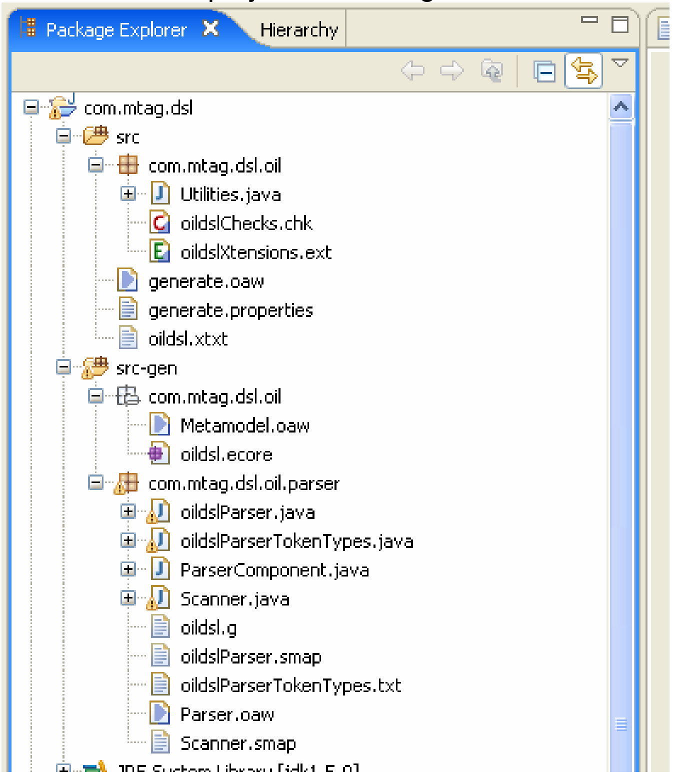
```
<workflow>
   <property file='generate.properties'/>
   <component file='org/openarchitectureware/xtext/Generator.oaw' inheritAll='true'/>
</workflow>
```

Picture 10: Generator.oaw workflow in the DSL project

An oAW workflow is a sequence of so-called Workflow Components, where each Component is responsible for one specific task within the workflow. A typical oAW Workflow consists of Components for model input and parsing, model validation and code generation. oAW ships with a number of predefined Components but its not too difficult to implement your own, if you have to. You will find all the information you need on oAW workflows in the workflow reference on the oAW homepage.

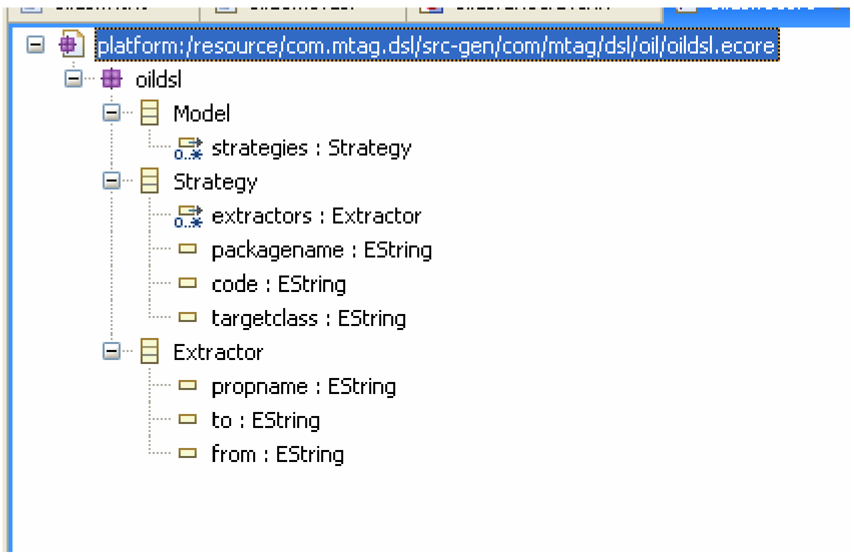
The workflow we are dealing with just hands over to a predefined Xtext workflow and configures it with the information we entered into the Xtext project wizard.

Start the workflow by right clicking the ‘generate.oaw’ file and selecting ‘Run as oAW Workflow’. A lot of things have been generated and added to our three projects. Picture 11 shows the DSL project after the generator run. Let’s do a quick scan of what’s new.



Picture 11: After the Generator Run

The first file to mention is oildslChecks.chk in the com.mtag.ds.oil package. This is an empty template specified in oAW's Check language. We will use this file later on to implement some constraints on our meta-model that will enhance the usability of our editor and make our modelling more failsafe.
Probably the most important artefact generated is the eCore model based on our grammar. It has been generated into the src-gen folder and is named oildsl.eCore.
Picture 12 shows the eCore model in its own tree editor. See for yourself, how closely the model and our grammar are related.


```
platform:/resource/com.mtag.dsl/src-gen/com/mtag/dsl/oil/oildsl.ecore
  oildsl
    Model
      strategies : Strategy
    Strategy
      extractors : Extractor
      packagename : EString
      code : EString
      targetclass : EString
    Extractor
      propname : EString
      to : EString
      from : EString
```
Picture 12: eCore model based on the OIL grammar

Finally, again located in the src-gen folder, a package com.mtag.dsl.oil.parser has been generated. It all starts with the oildsl.g file. This is an ANTLR grammar file which was fed to ANTLR in order to generate a parser for files written in our OIL grammar.

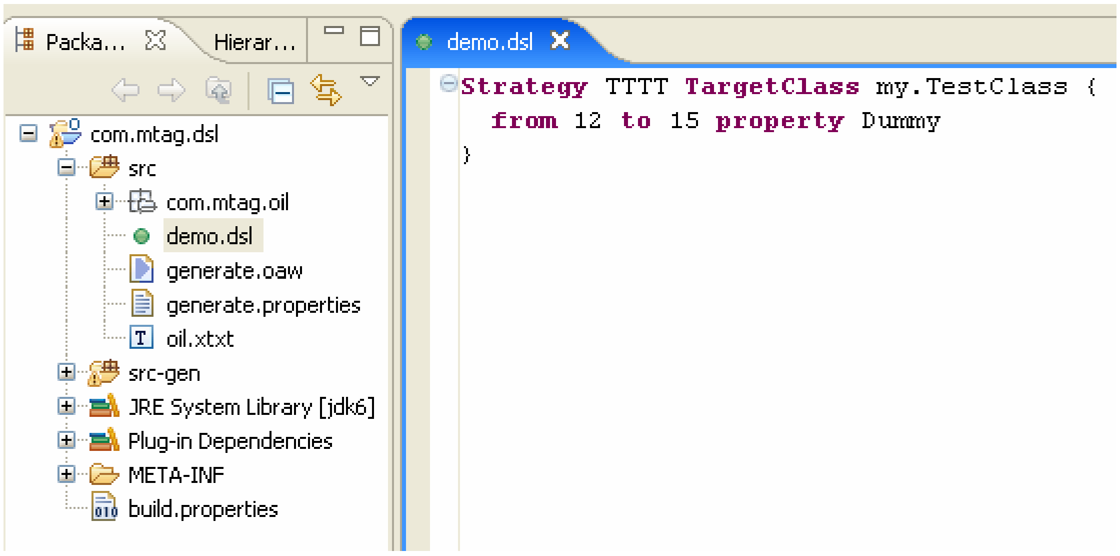
As we will not dig much deeper, the thing to remember is that Xtext uses EMF and ANTLR to accomplish the job of parsing an OIL text file into an eCore model.

You have already seen that the dsl project deals with grammars, parsers and models. Now let's take a look at the editor project. The generator added a lot of stuff here, so that we now have a working text editor conforming to the OIL grammar as an Eclipse plug-in.
The one file I would like to mention here is the oildslEditorExtension.ext file. Xtend is another oAW language and, as the name suggests, it is used to add features to a meta-model without affecting the model itself. For reasons I'm going to explain later, the file does not help us right away, but we will soon make good use of it while enhancing our editor.

While talking about enhancing the editor, I should point out that our editor is already fully functional. Just like editing Java files, it supports features like syntax highlighting, keyword completion and error checking.

In order to use the editor, you must understand that it is an Eclipse plug-in still under development and therefore not yet deployed as a plug-in to Eclipse.
The Eclipse way of testing your own plug-ins during development is to start them in a so-called ‘Runtime Workbench’. Let’s do this to see our ‘Out of the Box Editor’ in action.

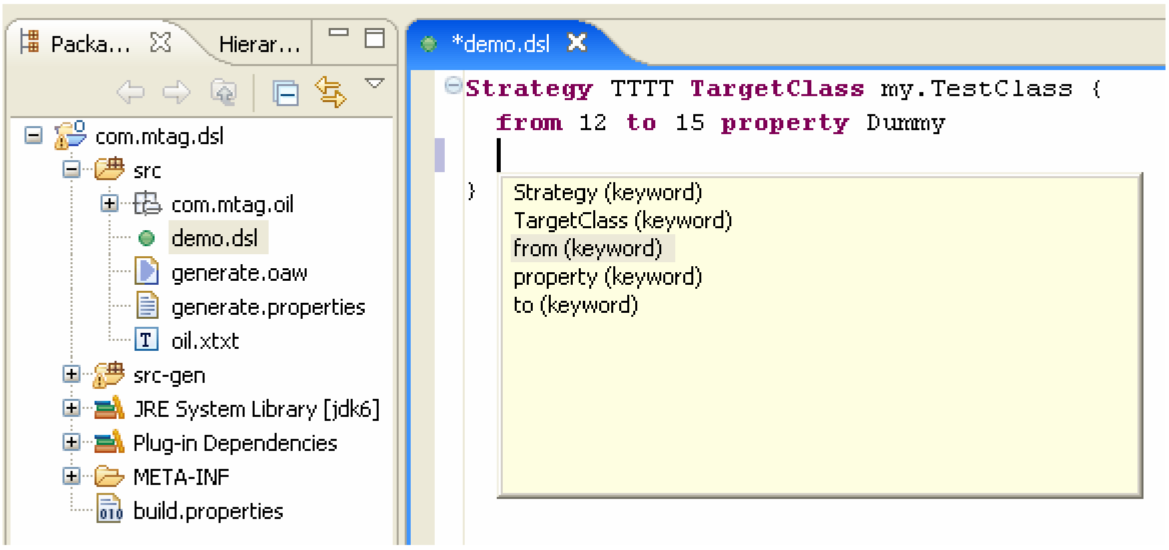
Probably the easiest way to test the editor is to create a file ‘demo.dsl’ in the src folder of your dsl project (I will introduce a more sophisticated approach in Part III). The extension of the file is important and must correspond to what you entered into the Xtext project wizard for Xtext to recognize it as a file containing text written in your language.
Right click the dsl project and select ‘Run as – Eclipse Application’.
This will start a workbench coming up with an empty workspace. Import your dsl project by clicking ‘File-Import-Existing Projects into Workspace’. Click ‘Next’, browse for your Eclipse Workspace and select the dsl project. Then click ‘Finish’ to import the dsl project into the Runtime Workbench.
Picture 13 shows my workspace after opening the demo.dsl file and entering some text.



```
Strategy TTTT TargetClass my.TestClass {
  from 12 to 15 property Dummy
}
```

Picture 13: Using the Editor

You can already see in the navigator, that oAW has recognized our demo.dsl file by prefixing it with a little icon. While entering text, you will find the keywords of our language highlighted.
Try the keyword completion feature as shown in the next picture.

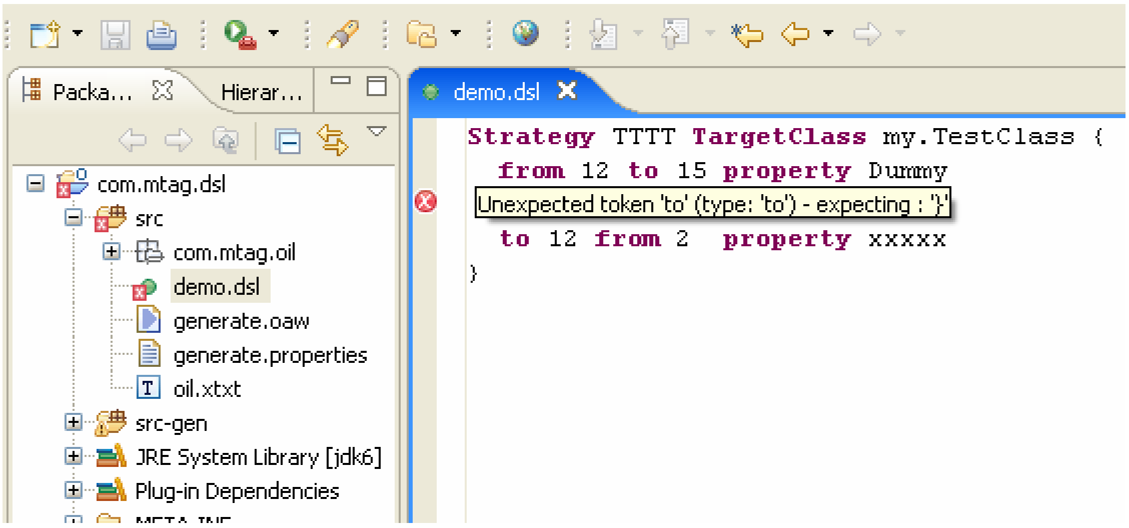


```
Strategy TTTT TargetClass my.TestClass {
  from 12 to 15 property Dummy

}
```

```
Strategy (keyword)
TargetClass (keyword)
from (keyword)
property (keyword)
to (keyword)
```

Picture 14: Keyword Completion Feature

Finally, enter some nonsense to verify that the syntax of your input will be checked and must conform to the rules of our grammar.



```
Strategy TTTT TargetClass my.TestClass {
  from 12 to 15 property Dummy
Unexpected token 'to' (type: 'to') - expecting : '}'
  to 12 from 2  property xxxxx
}
```

Picture 15: Syntax Checking

The navigator propagates errors along the project structure, just as Eclipse would do with errors in a Java file. The generated editor integrates our language smoothly into the IDE.

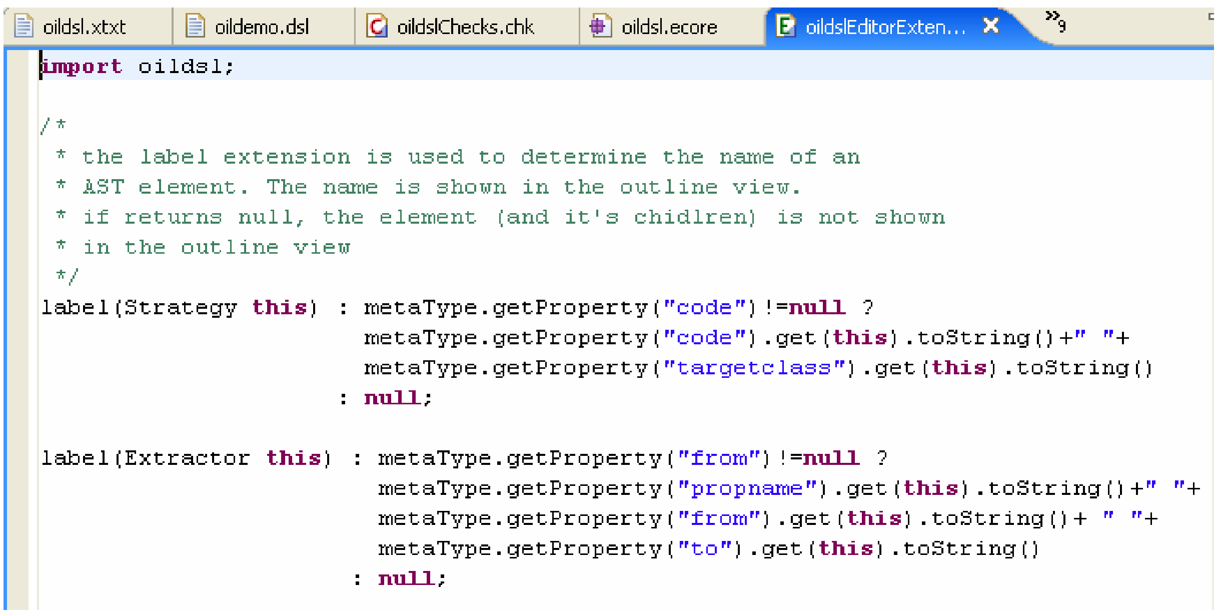
Before we proceed, let's sum up what we did, what we got and what's left to do.
We fed some information into the Xtext project wizard and ended up with three projects, a dsl project, an editor project and a generator project.
The only thing we actually did was to specify a grammar for our language to be. Though the explanations where lengthy, the grammar itself was only a couple of lines of code.
We then started a pre-configured generation process and ended up with a fully functional text editor for editing text files based on our own grammar.

The rest of Part II will deal with enhancing the editor. We will improve its integration into Eclipse, but most of all we will improve its ability to detect logical errors within our text so that configuring our abstraction with an external DSL will become much safer than modifying code and experiencing errors at runtime.

We'll start by adding an outline view to our editor.

The generator created the file 'oildslEditorExtensions.ext', already mentioned above, for the purpose of extending the editor. We will edit this file to provide for an outline view.
The file extension 'ext' is associated with oAW's Xtend language, which we now have to deal with.
The main purpose of the Xtend language, as the name suggests, is the extension of your meta-model without actually polluting the model itself (other purposes, like model transformations, are out of the scope of this article).
Xtend is a functional language based, like all oAW languages, on a common type system and expression language. It's not too difficult to find your way around because - like in all oAW language editors - keyword completion drop down lists indicate your options.
We will make further use of the Xtend language while implementing checks on our model later on while working on the dsl project.

'oildslEditorExtensions.ext' contains a function called 'label()', which is used by the editor to set up the outline view. By default, label() inspects a given meta type for a property called 'name'. If such a property exists, its content is displayed in the outline view, simple as that. So we would have seen an outline view right from the start, if only our meta types had a property called 'name'. Our Strategy meta type has properties like 'code' and 'targetclass', but no name property, so we have to invest some extra work and re-code the label function to implement an outline view.
Picture 16 shows the code.

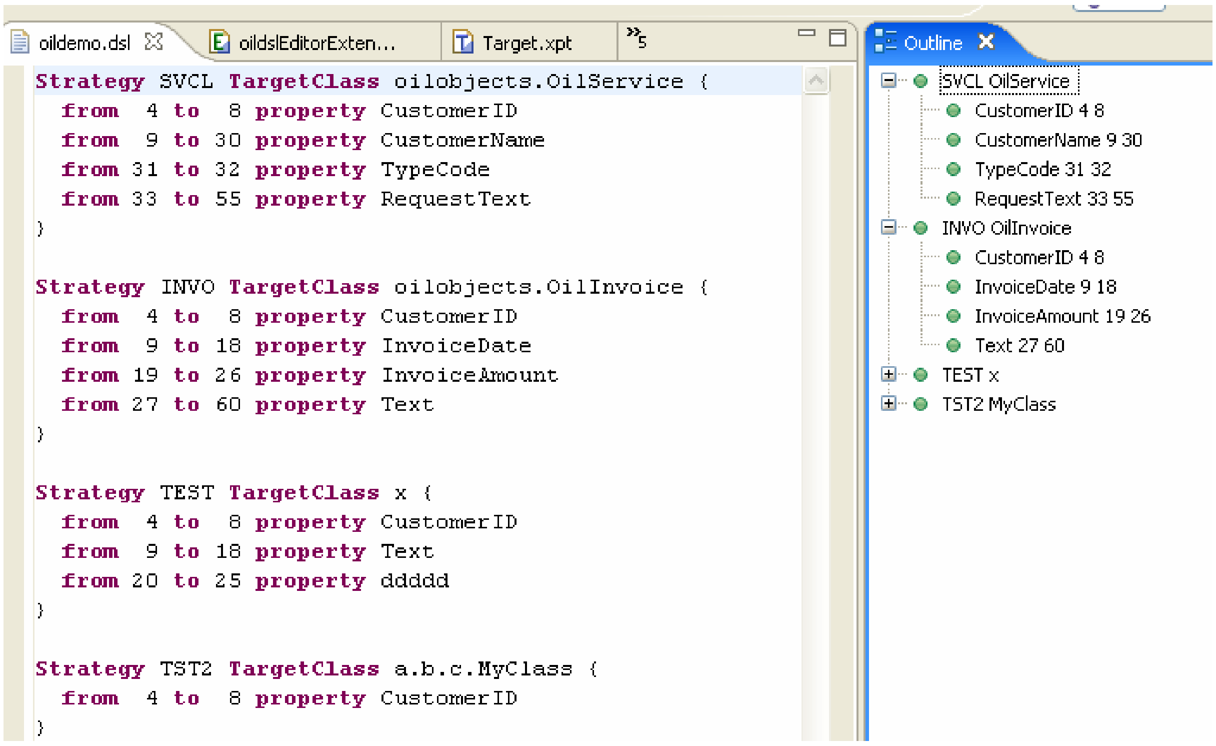


```
import oildsl;

/*
 * the label extension is used to determine the name of an
 * AST element. The name is shown in the outline view.
 * if returns null, the element (and it's chidlren) is not shown
 * in the outline view
 */
label(Strategy this) : metaType.getProperty("code")!=null ?
                        metaType.getProperty("code").get(this).toString()+" "+
                        metaType.getProperty("targetclass").get(this).toString()
                      : null;

label(Extractor this) : metaType.getProperty("from")!=null ?
                         metaType.getProperty("propname").get(this).toString()+" "+
                         metaType.getProperty("from").get(this).toString()+ " "+
                         metaType.getProperty("to").get(this).toString()
                       : null;
```

Picture 16: Outline View Code

We implemented two versions of the label function. One is for Strategy types, one is for Extractor types. In both cases, the result of the function is the string to be displayed in the outline view. For Strategy types, the result string is made up of the code and targetclass properties. For Extractor types, we concatenate the name of the target property with the 'from' and 'to' positions, respectively. Note that we have to import the namespace of our meta-model for the Xtend editor to recognise our meta types. Take that away and you will end up with a lot of ugly error markers.

The result of our implementation is an outline view such as the one shown in Picture 17.

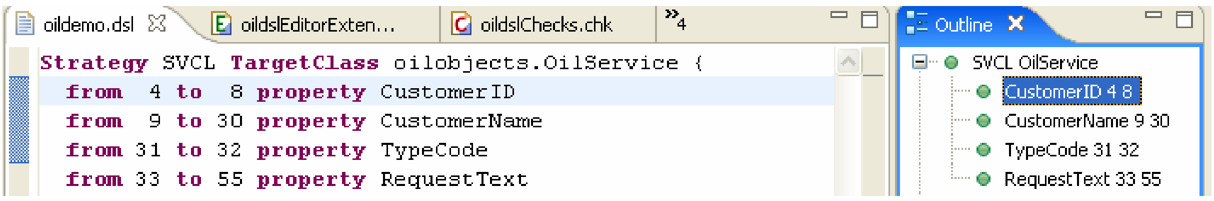


```
Strategy SVCL TargetClass oilobjects.OilService {
  from  4 to  8 property CustomerID
  from  9 to 30 property CustomerName
  from 31 to 32 property TypeCode
  from 33 to 55 property RequestText
}

Strategy INVO TargetClass oilobjects.OilInvoice {
  from  4 to  8 property CustomerID
  from  9 to 18 property InvoiceDate
  from 19 to 26 property InvoiceAmount
  from 27 to 60 property Text
}

Strategy TEST TargetClass x {
  from  4 to  8 property CustomerID
  from  9 to 18 property Text
  from 20 to 25 property ddddd
}

Strategy TST2 TargetClass a.b.c.MyClass {
  from  4 to  8 property CustomerID
}
```

```
SVCL OilService
    CustomerID 4 8
    CustomerName 9 30
    TypeCode 31 32
    RequestText 33 55
INVO OilInvoice
    CustomerID 4 8
    InvoiceDate 9 18
    InvoiceAmount 19 26
    Text 27 60
TEST x
TST2 MyClass
```

Picture 17: Outline View

What goes on behind the scenes?
The text above is parsed into an Abstract Syntax Tree (AST) according to our grammar. In order to display the outline view, Xtext walks this AST and calls the label function for every meta type it encounters. The results of these function calls are used to construct the outline view. This is more proof of how our self made language has become part of the development environment. The IDE works directly with the abstract representation of our language, just as it does with Java, for example.
This can also be seen when selecting an item in the outline view. The selection is immediately projected into the editor.



```
Strategy SVCL TargetClass oilobjects.OilService {
  from  4 to  8 property CustomerID
  from  9 to 30 property CustomerName
  from 31 to 32 property TypeCode
  from 33 to 55 property RequestText
```

```
SVCL OilService
    CustomerID 4 8
    CustomerName 9 30
    TypeCode 31 32
    RequestText 33 55
```

Picture 18: Editor and outline view relationship

Let's round up Part II with defining and implementing some check constraints on our meta-model. The checks we've seen so far were on a mere syntactical level based on the grammar of our language.
We now want to go beyond syntax and impose some logical checks on our model.
I consider this to be one of the most interesting aspects of an external DSL. Error checking can now be done while editing the model and not only at program runtime.

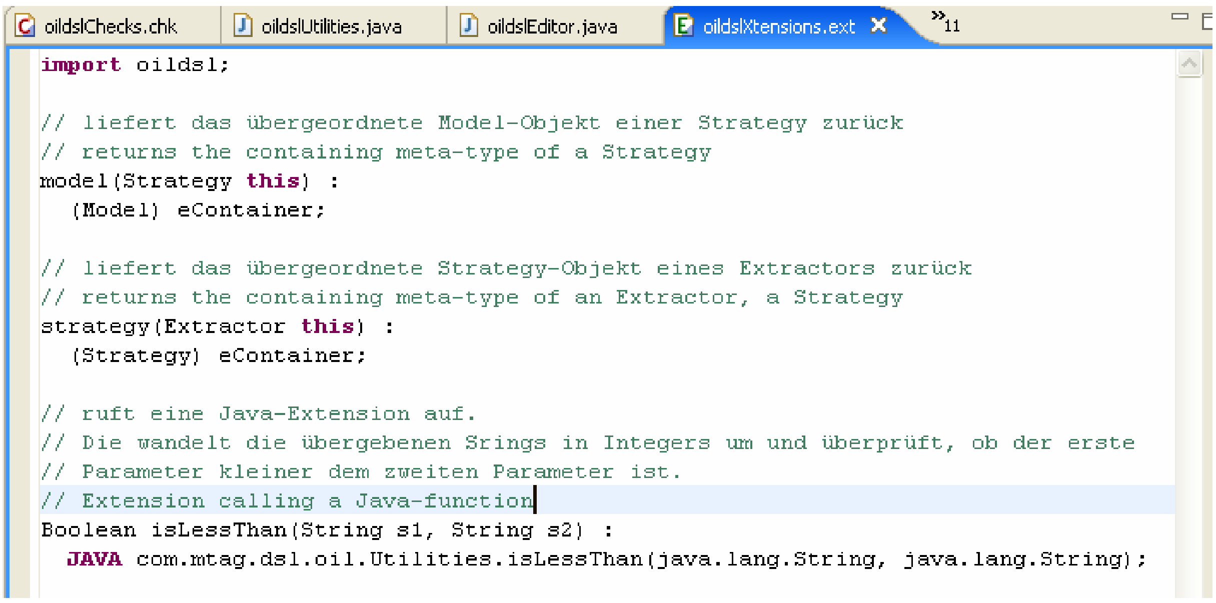
Let's decide on a couple of rules we want to hold true for our language.

- Every Strategy needs a unique type code.
  We don't want two or more Strategies with the type code 'SVCL' for example.
- Within a Strategy we want the Extractors to assign their data to unique properties. Otherwise we would be overwriting previously written data.
- Within an Extractor, the 'from' value must always be less than the 'to' value.

For implementing checks on meta-models, oAW offers the Check language. Check, again, is based on oAW's type system and expression language but comes along with a special syntax for implementing checks on meta-models.

These checks will be defined directly on the meta-model, so we have to switch back to the dsl project again.
The generator created a basic template, 'oildslChecks.chk', to start with. It's located within the dsl project in the folder 'com.mtag.dsl.oil'.

Before we modify this file to implement our checks, we use the now familiar Xtend language again to write a couple of extensions that will make life a little easier for us.
We create a file 'oildslXtensions.ext' in the same folder. Here's the code:



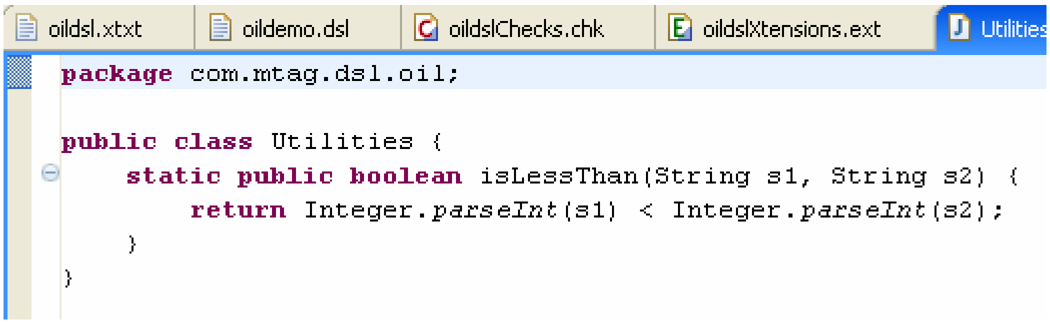
```
import oildsl;

// liefert das übergeordnete Model-Objekt einer Strategy zurück
// returns the containing meta-type of a Strategy
model(Strategy this) :
   (Model) eContainer;

// liefert das übergeordnete Strategy-Objekt eines Extractors zurück
// returns the containing meta-type of an Extractor, a Strategy
strategy(Extractor this) :
   (Strategy) eContainer;

// ruft eine Java-Extension auf.
// Die wandelt die übergebenen Srings in Integers um und überprüft, ob der erste
// Parameter kleiner dem zweiten Parameter ist.
// Extension calling a Java-function
Boolean isLessThan(String s1, String s2) :
  JAVA com.mtag.dsl.oil.Utilities.isLessThan(java.lang.String, java.lang.String);
```

Picture 19: Useful extensions

Fist of all we have to import the meta-model namespace again, in order to make the meta types known to the editor.
Next we create two extensions to make navigation within the model more convenient.
The fist extension is called model(). It takes a Strategy type as an argument and returns its containing type. We use the ECore property ‘eContainer’ to determine the containing type.
Remember that oAW uses EMF and therefore ECore as the meta meta-model for our meta-model. eContainer references the containing type. We know from our grammar that the Model is the container of a Strategy, so the cast of eContainer to Model will be fine.
Calling this function on a Strategy will return the containing Model.

We implement a similar function to be called on an Extractor returning the containing Strat-egy.

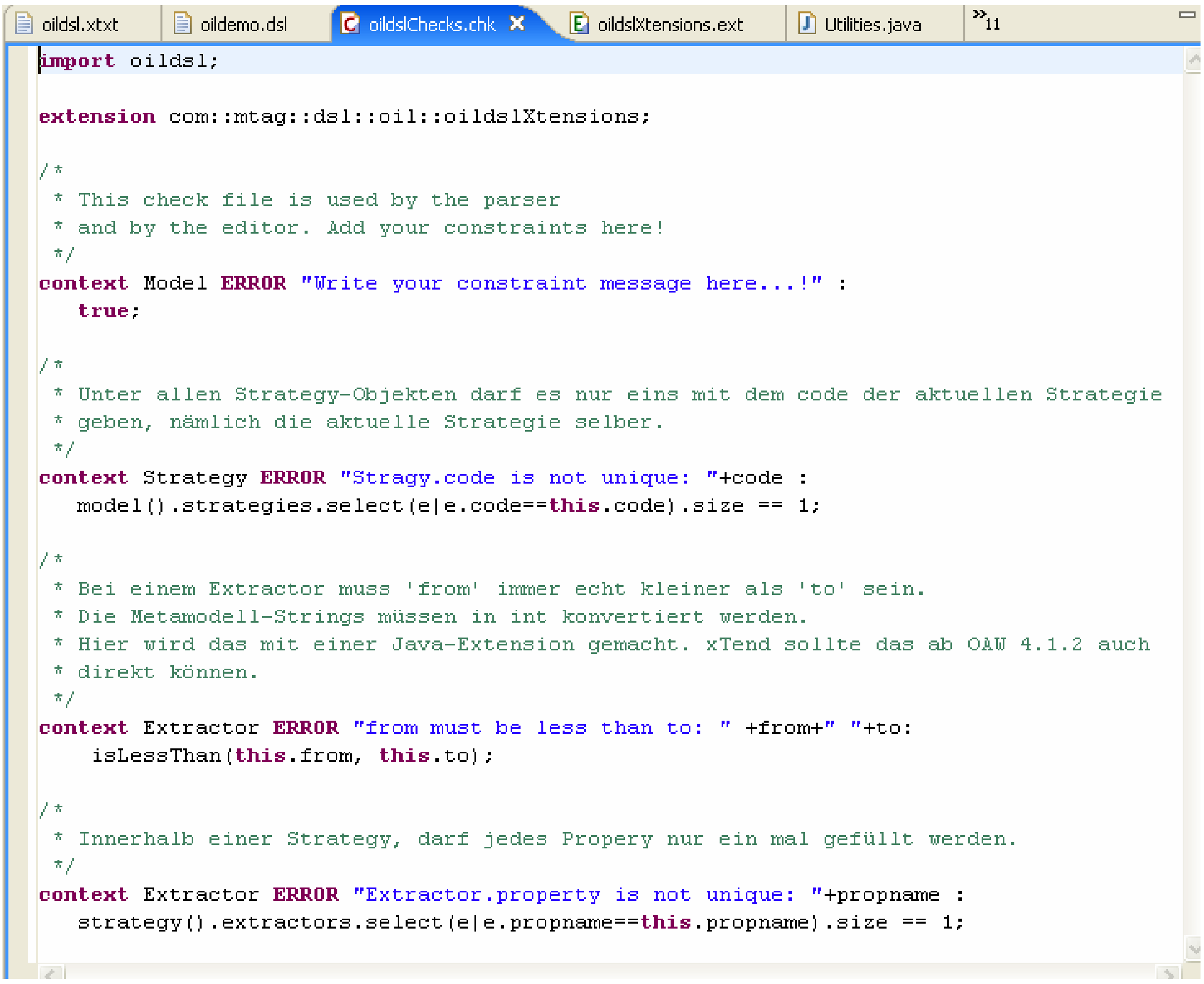
The third function is a Java extension. We use Java to transform two Strings into Integers and perform a comparison bewteen them. The isLessThan() function is coded in a separate file called Utilities.java. Here is the code, more for completeness than for being interesting.



```
package com.mtag.dsl.oil;

public class Utilities {
    static public boolean isLessThan(String s1, String s2) {
        return Integer.parseInt(s1) < Integer.parseInt(s2);
    }
}
```

Picture 20: isLessThan() Function

You see it's easy to call Java code from within Xtend. Just make sure that your Java function is static.
Now we are ready to implement our checks.



oildsl.xtxt | oildemo.dsl | oildslChecks.chk | oildslXtensions.ext | Utilities.java | »11

```
import oildsl;

extension com::mtag::dsl::oil::oildslXtensions;

/*
 * This check file is used by the parser
 * and by the editor. Add your constraints here!
 */
context Model ERROR "Write your constraint message here...!" :
   true;

/*
 * Unter allen Strategy-Objekten darf es nur eins mit dem code der aktuellen Strategie
 * geben, nämlich die aktuelle Strategie selber.
 */
context Strategy ERROR "Stragy.code is not unique: "+code :
   model().strategies.select(e|e.code==this.code).size == 1;

/*
 * Bei einem Extractor muss 'from' immer echt kleiner als 'to' sein.
 * Die Metamodell-Strings müssen in int konvertiert werden.
 * Hier wird das mit einer Java-Extension gemacht. xTend sollte das ab OAW 4.1.2 auch
 * direkt können.
 */
context Extractor ERROR "from must be less than to: " +from+" "+to:
    isLessThan(this.from, this.to);

/*
 * Innerhalb einer Strategy, darf jedes Propery nur ein mal gefüllt werden.
 */
context Extractor ERROR "Extractor.property is not unique: "+propname :
   strategy().extractors.select(e|e.propname==this.propname).size == 1;
```

Picture 21: Checks on meta-types

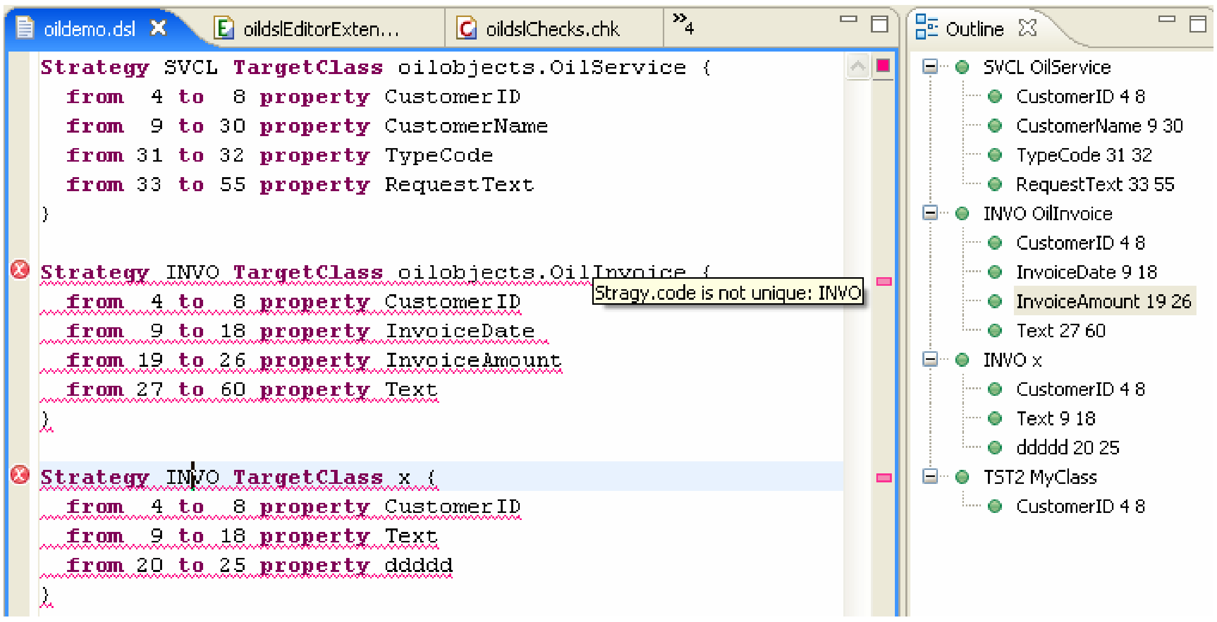
Again we make our meta-types known by importing the model namespace.
Because we want to use our self made extensions, we have to include them too.

The first check we see is on the Model meta-type. This one has been created by the wizard. We could have deleted it but instead use it for a brief description of the structure of the Check language.
A check always starts with the keyword 'context' followed by the meta-type we want to impose the constraint upon. The next keyword is either ERROR or WARNING indicating the severity of the constraint violation. The severity level is followed up by the message we would like to see associated with an error or warning respectively. Finally a colon separates the message from the actual code performing the check, which has to be an expression evaluating to a Boolean. In our case the Model check always returns true, so you will never see its error message.

The check we impose on the Strategy meta-type is more interesting because the expression involves model navigation and collections. As you can easily look up the syntax in oAW's expression language reference, I'll try to put into words, what's going on.
The context of this check is the meta-type Strategy, so this check will be performed on all Strategy instances within our model. Because we want to check the uniqueness of the code property of a given Strategy, we have to take into account all the other Strategy instances. In order to get a list of these, we make use of our model() extension to navigate to the container of the given Strategy, which is the Model. From there, we perform a 'select' on the Model's Strategy list selecting all Strategy instances with the same 'code' as the given one. If the result of this selection is a collection of size 1, 'true' will be returned. The function returns false otherwise and an error marker will be displayed at the respective Strategy instance within the editor.

We placed two checks on Extractors. One is very similar to the Strategy uniqueness check, only we're checking the uniqueness of the 'property' property of the target class.
The second check is comparing the 'from' and 'to' properties of an Extractor. We simply pass the respective properties to our isLessThan() extension, which then calls our static Java function comparing theses parameters as Integers.

Let's have a look at the new features in action.
Re-launch your Eclipse runtime workbench and use the editor to type some errors to test each check.
The first test involves checking Strategy code uniqueness. After saving the editor, error markers will be placed on every model instance that did not pass all its checks. Pointing the cursor to a marker displays the error message.
Picture 22 shows the result.



```
Strategy SVCL TargetClass oilobjects.OilService {
  from  4 to  8 property CustomerID
  from  9 to 30 property CustomerName
  from 31 to 32 property TypeCode
  from 33 to 55 property RequestText
}

Strategy INVO TargetClass oilobjects.OilInvoice {
  from  4 to  8 property CustomerID
  from  9 to 18 property InvoiceDate
  from 19 to 26 property InvoiceAmount
  from 27 to 60 property Text
}

Strategy INVO TargetClass x {
  from  4 to  8 property CustomerID
  from  9 to 18 property Text
  from 20 to 25 property ddddd
}
```

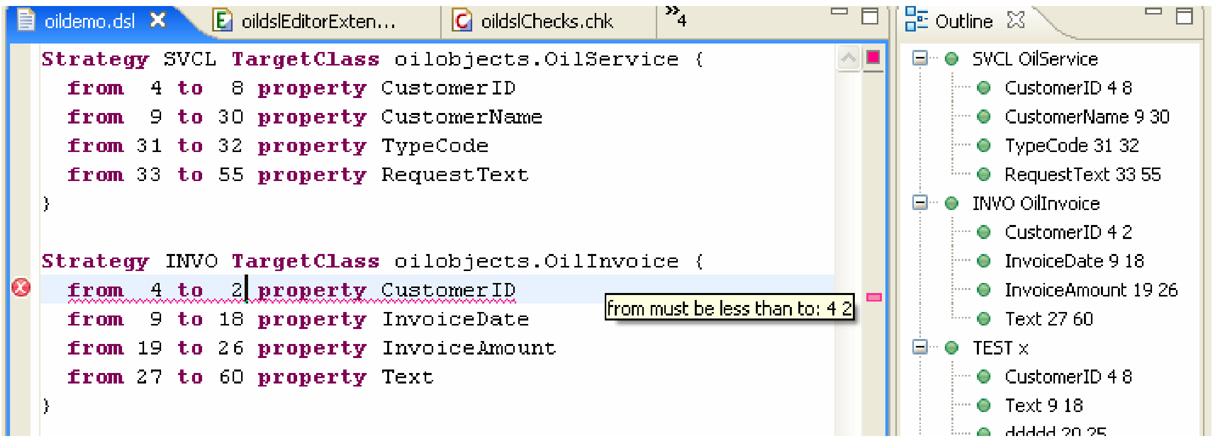
```
Stragy.code is not unique: INVO
```

```
SVCL OilService
    CustomerID 4 8
    CustomerName 9 30
    TypeCode 31 32
    RequestText 33 55
INVO OilInvoice
    CustomerID 4 8
    InvoiceDate 9 18
    InvoiceAmount 19 26
    Text 27 60
INVO x
    CustomerID 4 8
    Text 9 18
    ddddd 20 25
TST2 MyClass
    CustomerID 4 8
```

Picture 22: Strategy uniqueness check

As you can see, we won't be able to enter non unique codes on Strategies any more. Note that the check has been performed on all Strategy instances and both INVOs have been marked.

The next picture shows a 'from-to' violation.


oildemo.dsl | oildslEditorExten... | oildslChecks.chk | »4 | Outline

```
Strategy SVCL TargetClass oilobjects.OilService {
  from  4 to  8 property CustomerID
  from  9 to 30 property CustomerName
  from 31 to 32 property TypeCode
  from 33 to 55 property RequestText
}

Strategy INVO TargetClass oilobjects.OilInvoice {
  from  4 to  2 property CustomerID
  from  9 to 18 property InvoiceDate
  from 19 to 26 property InvoiceAmount
  from 27 to 60 property Text
}
```

from must be less than to: 4 2

```
SVCL OilService
  CustomerID 4 8
  CustomerName 9 30
  TypeCode 31 32
  RequestText 33 55
INVO OilInvoice
  CustomerID 4 2
  InvoiceDate 9 18
  InvoiceAmount 19 26
  Text 27 60
TEST x
  CustomerID 4 8
  Text 9 18
  ddddd 20 25
```

Picture 23: From-to violation

Of course we could have implemented all these checks directly within our abstraction, but there is no way around the fact that error checking would have to be done at runtime.
Our external language on the other hand can be made as fail save as we think appropriate to ensure that the abstraction is always soundly configured.

Before we proceed by generating code in Part III, here's another short summary.
We finished Part II with a couple of enhancements to our editor and the meta-model it is based upon.
Modelling with the editor is more fun now, but most of all it is much safer because we taught it some semantics. These semantics were coded as extensions to the meta-model, not the editor. The editor only visualizes what happens. This makes the meta-model the heart of the matter with the editor as its manipulation and visualisation tool.
In Part III we will generate the code implemented manually in Part I, based only on our meta-model. The editor project will have no part in this.

Part III: Generating Code

After coming this far, you will probably be disappointed to find the code generation process to be the easiest part of it all. All it takes is just one more oAW language, called Xpand, to put to use.

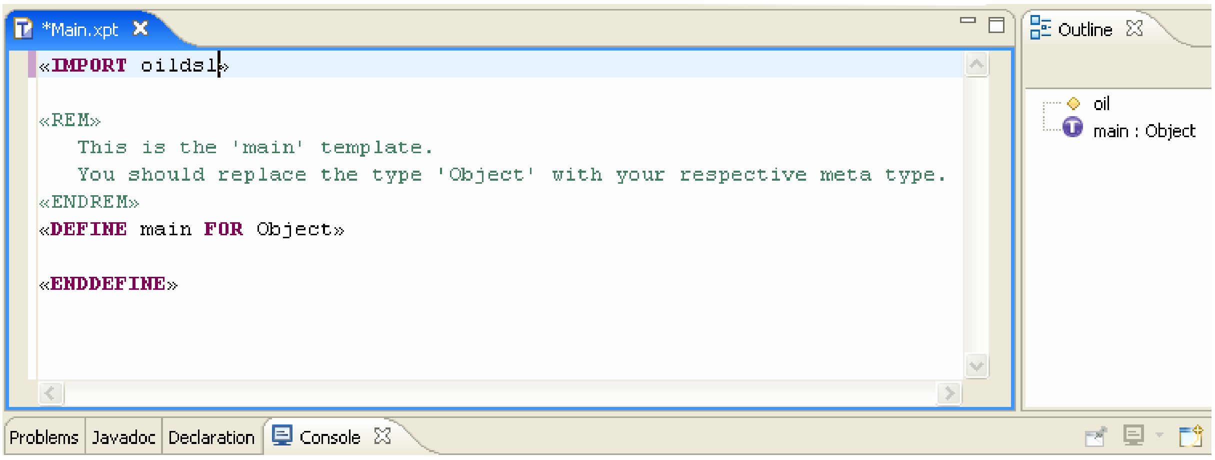
I found it to be more difficult to set up all the projects within Eclipse correctly in order to have everything, projects and plug-ins, working smoothly together, than to program the code generator itself. I will therefore take some extra time to describe a simple set up that will work nicely for developing and testing our code generator. Look out for the compiled Eclipse configuration tips to be published on our web site.

Here is the story board for Part III.

We will first turn our attention to the generator project. Here we will actually implement the code generator for our DSL. After that we are finished with the three projects generated by the Xtext project wizard as far as implementation goes. What's left to do with them is a little bit of Eclipse configuration to make them available to other projects.
Finally we will bring it all together by creating a new pure java project that will make use of all the stuff we've done so far. In this new project, we will create an OIL file with the help of our self made OIL editor. The last step will be the creation of a very simple oAW workflow to call the generator project, invoke the code template on our model and start the code generation.

Here is how to implement the code generator as an oAW Xpand template.

Open the generator project and take a look at the file 'com.mtag.dsl.oil.Main.xpt' located in the 'src' folder.



```
«IMPORT oildsl»

«REM»
   This is the 'main' template.
   You should replace the type 'Object' with your respective meta type.
«ENDREM»
«DEFINE main FOR Object»

«ENDDEFINE»
```

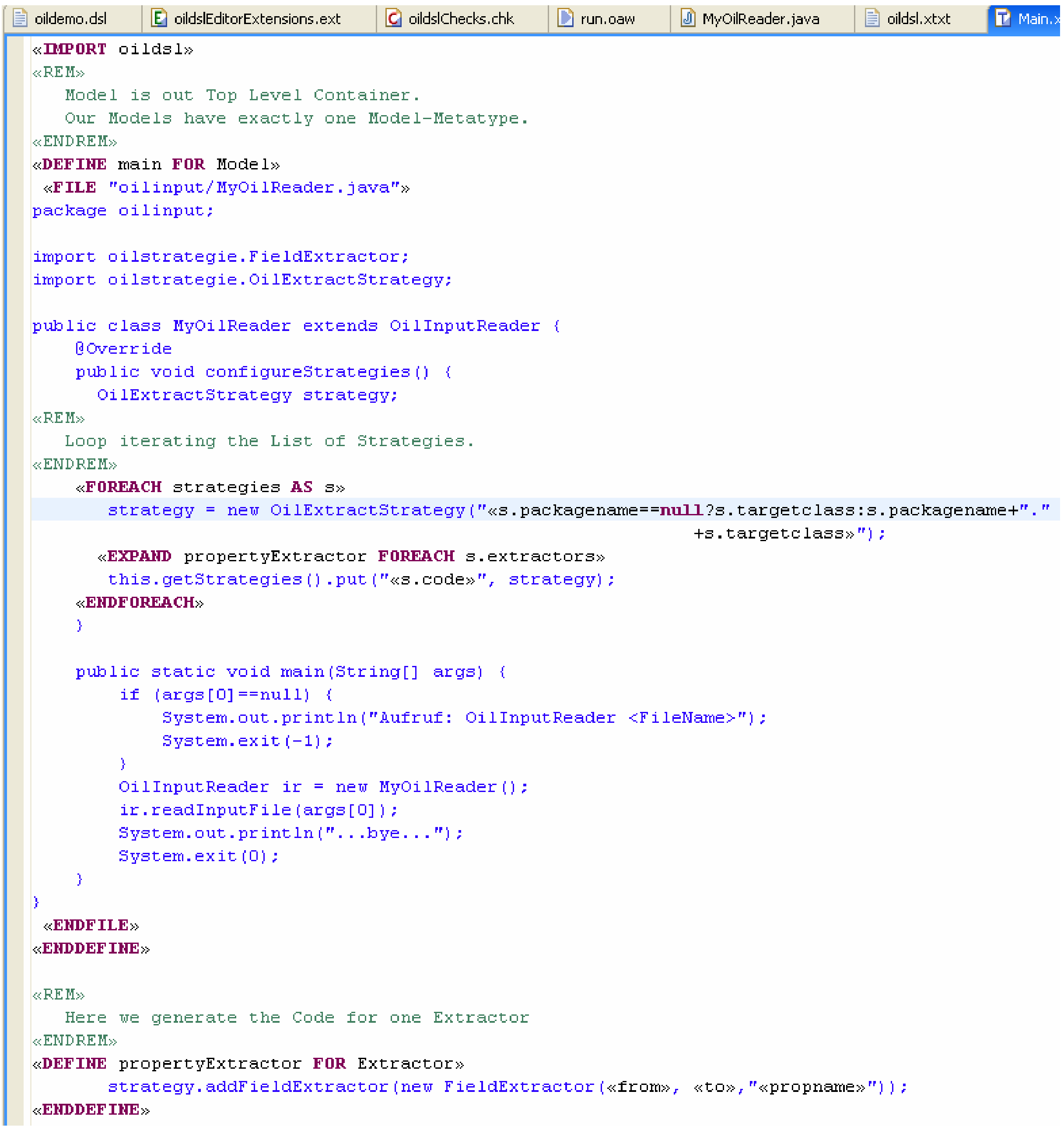
Picture 26: The main Xpand Template

This is another file initially created by the Xtext project wizard.
Its file extension 'xpt' marks it as an Xpand template file. Xpand is oAW's template language for code generation. Xpand is quite powerful on its own but can also seamlessly integrate extensions written in Xtend. Read all about Xpand in the language reference documentation on the oAW homepage.

Basically, with Xpand you walk along the abstract syntax tree (AST) of your model and generate your code along the way. That's why the first thing to do is to import your meta-model namespace (the Xtext project wizard already did this for us) again.
The building blocks of Xpand are called 'templates' and are declared by a 'DEFINE' statement.
Every template consists of a template name and a meta-type on which the template can be called. In this way a template is much like a subroutine, parameterized by a meta-type and optional parameters.
Although we don't make use of it here, templates do support dynamic polymorphism. This allows for more than one template sharing the same name but differ in their respective parameter types. When expanding (calling) a template, the meta-type of the parameter supplied by the caller decides which template will be called at runtime.

Again, we don't use this feature, but of course it is possible to import extensions (written in oAW's Xtend language) into a template file, so your Xpand template can make use of every feature you added to your meta-model by way of extensions.

Let's now modify the template Main.xpt, in order to generate the code already laid out in Picture 4 of Part I.



```
«IMPORT oildsl»
«REM»
   Model is out Top Level Container.
   Our Models have exactly one Model-Metatype.
«ENDREM»
«DEFINE main FOR Model»
 «FILE "oilinput/MyOilReader.java"»
package oilinput;

import oilstrategie.FieldExtractor;
import oilstrategie.OilExtractStrategy;

public class MyOilReader extends OilInputReader {
    @Override
    public void configureStrategies() {
      OilExtractStrategy strategy;
«REM»
   Loop iterating the List of Strategies.
«ENDREM»
    «FOREACH strategies AS s»
       strategy = new OilExtractStrategy("«s.packagename==null?s.targetclass:s.packagename+"."
                                                      +s.targetclass»");

      «EXPAND propertyExtractor FOREACH s.extractors»
       this.getStrategies().put("«s.code»", strategy);
    «ENDFOREACH»
    }

    public static void main(String[] args) {
        if (args[0]==null) {
            System.out.println("Aufruf: OilInputReader <FileName>");
            System.exit(-1);
        }
        OilInputReader ir = new MyOilReader();
        ir.readInputFile(args[0]);
        System.out.println("...bye...");
        System.exit(0);
    }
}
 «ENDFILE»
«ENDDEFINE»

«REM»
   Here we generate the Code for one Extractor
«ENDREM»
«DEFINE propertyExtractor FOR Extractor»
       strategy.addFieldExtractor(new FieldExtractor(«from», «to»,"«propname»"));
«ENDDEFINE»
```

Picture 27: The generator template

We have created two templates, a main template to be called on a Model and a ‘propertyExtractor’ template to be called on an Extractor. Remember that Model, Strategy and Extractor are the meta-types derived from our grammar.

‘main’ is our top level template and will be called from outside of this file (how exactly, we will see in a minute).
The first thing ‘main’ does, is to create a file ‘oilinput/MyOilReader.java’ using the FILE statement. Everything between FILE and ENDFILE will end up in this file, except other Xpand statements.

What follows is some boilerplate text to be directly inserted into the file, until we reach the FOREACH block.
The FOREACH statement takes the strategies collection of the Model as input and iterates through it aliasing each Strategy so that we can refer to the current Strategy by a shortcut. For every Strategy in the strategies collection, we generate the OilExtractStrategy instantiation.

Next, we expand the ‘propertyExtractor template for a Strategy’s ‘extractor’ collection.
The ‘properyExtractor’ template generates just one line of code for adding a newly instantiated and configured ‘FieldExtractor’ object to the Strategy.

You can see, how static template code and access to meta-type properties are mixed together in an Xpand template (Xpand keywords and meta type references are always enclosed in “«” and “»” characters, the so called guillemets).

When we are done with all the Strategies the Model holds, all that’s left to do is to add some boilerplate code for the main() method.

As I told you: too easy to be real fun.

Let’s finish with the three Xtext projects by adding some Eclipse specific configuration to make them accessible from other projects.

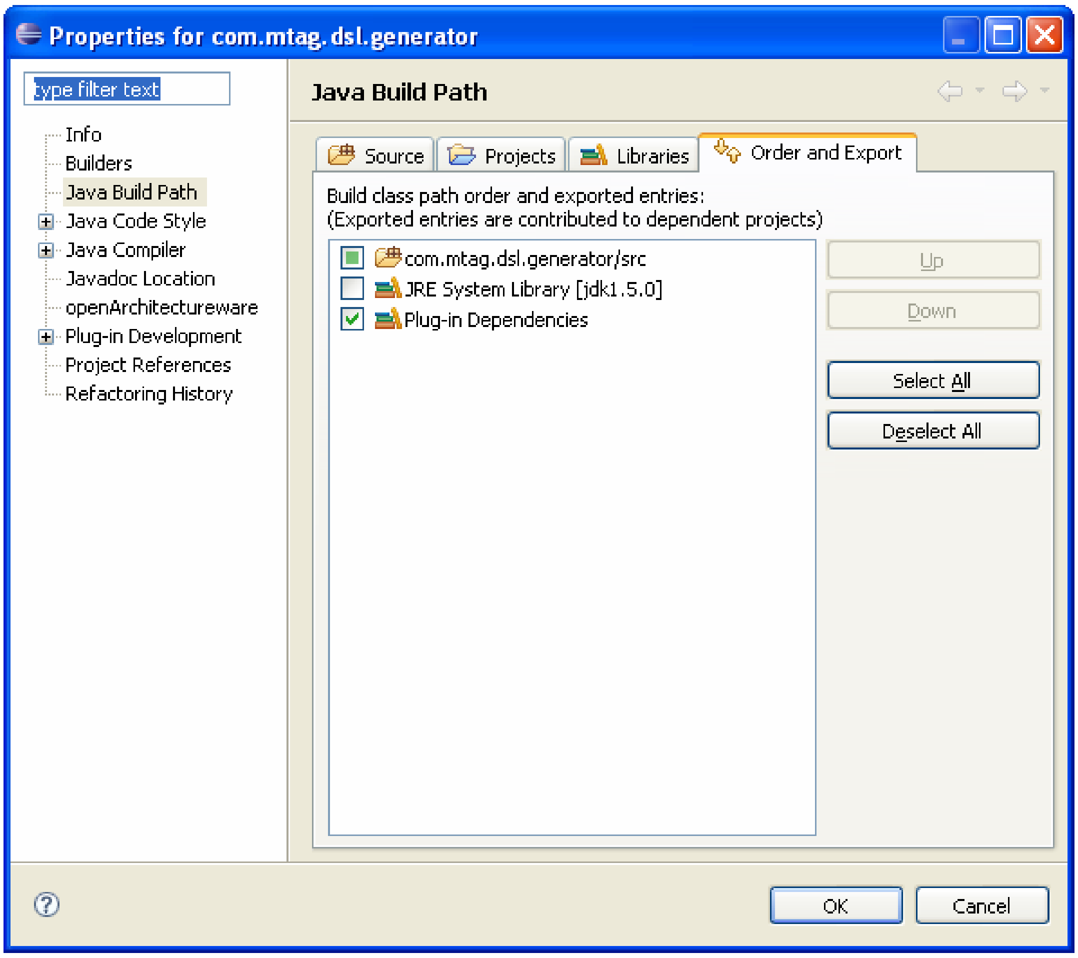
We’ve already seen that the tree projects created by the Xtext project wizard are Eclipse plug-in projects. They are pre-configured with all the plug-in dependencies they need in order to work properly with the oAW engine. Take a closer look at these dependencies and you will notice that the three projects are also interrelated with one another, such that the editor and the generator project depend on the dsl project.

Our yet to be created Java project cannot make use of the generator project and the dsl project right away. It can only make use of them and the work we put into them, when these projects are deployed as Eclipse plug-ins or alternatively are run within an Eclipse runtime workbench. We will take this second approach because our plug-ins are still under development. When implementing an extension or an Xpand template, you want to test what you did as soon as possible by starting a runtime workbench and test the generator.

What we have to do is to create the invoking Java project and make some adjustments to ensure that this program has access to the generator projects and all of its dependencies.

The first thing to do, in order to get these dependencies in line, is to export the plug-in dependencies of the generator project. This will make all theses dependencies available to any project using the generator project. We will make our soon to be created Java project depend on the generator project and will therefore have access to the oAW engine.

To re-export the generator project's dependencies, select it in the navigator and select 'Project-Properties' from the menu. Click 'Java Build Path' and activate the 'Order and Export' tab. Click the plug-in dependencies like in Picture 24.

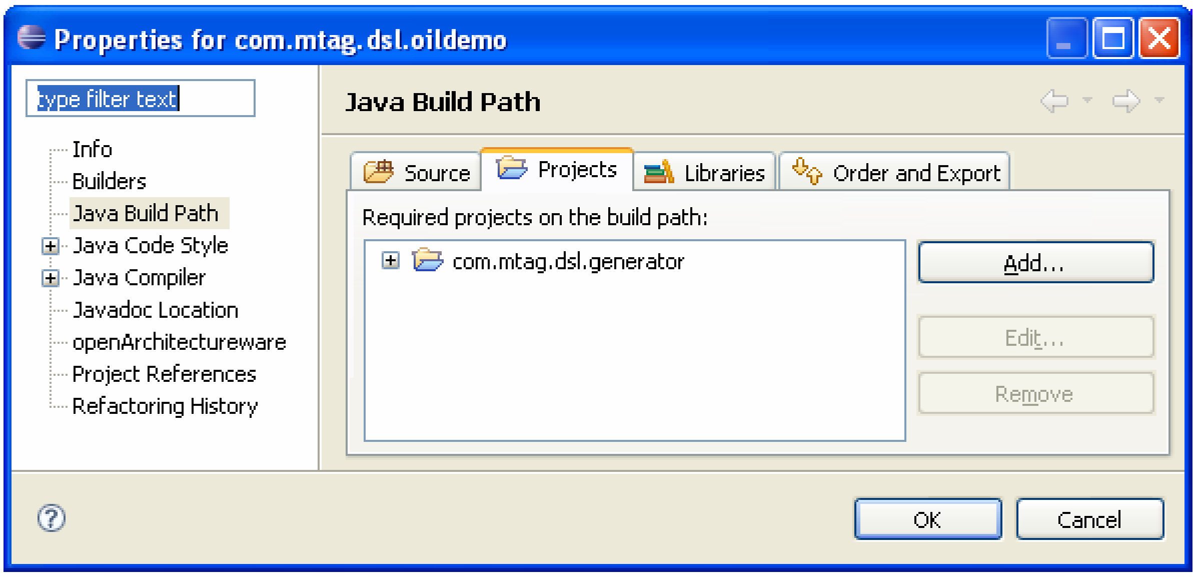


Build class path order and exported entries:
(Exported entries are contributed to dependent projects)

- ▣ com.mtag.dsl.generator/src
- ☐ JRE System Library [jdk1.5.0]
- ✓ Plug-in Dependencies

Picture 24: Export the plug-in dependencies of the generator project

Now create the Java project and make it depending on the generator project. I called my Java project 'com.mtag.dsl.oildemo'. Select the project properties again and activate the 'Projects' tab. Now add the generator project as a dependency, like in Picture 25.

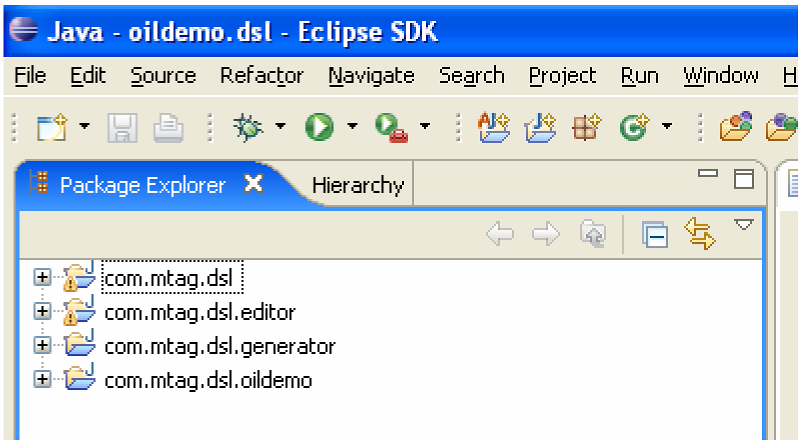


Picture 25: Generator Project dependency

The rest has to be done within a runtime workbench.

Again select the generator project, activate the context menu and select ‘Run as’-‘Eclipse Application’.
Now import all the projects we’ve done so far: the three Xtext projects and the new Java project we just created. This is done from the ‘File’ menu by selecting ‘Import’-‘General’ – ‘Existing Projects into Workspace’. Browse for your workspace directory and select the respective projects.

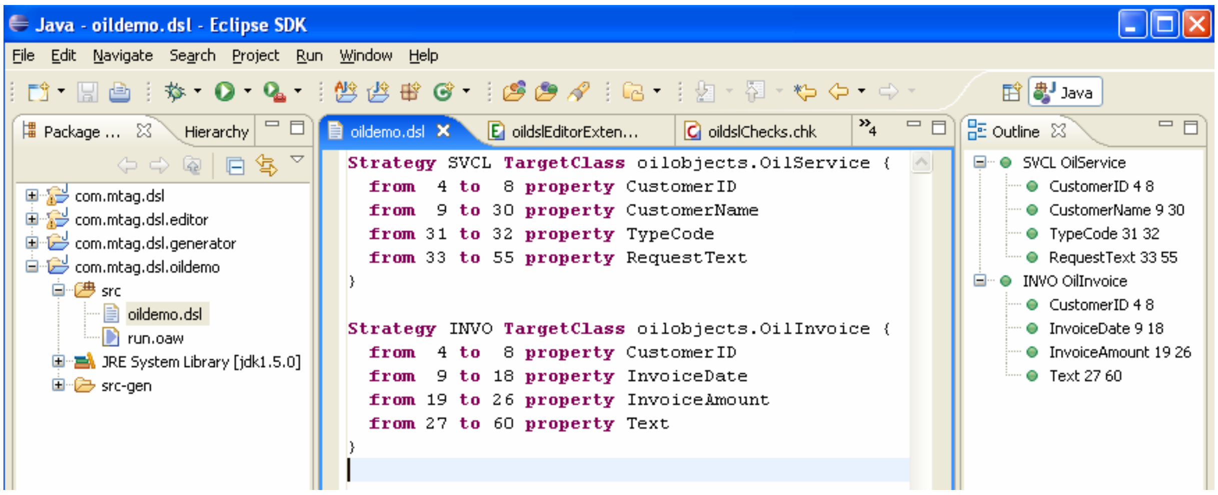
If all went well, your runtime workbench should look similar to Picture 26.



Picture 26: Runtime workbench

The new Java project is the place to create the configuration using our configuration language and editor.

Picture 27 shows all this.

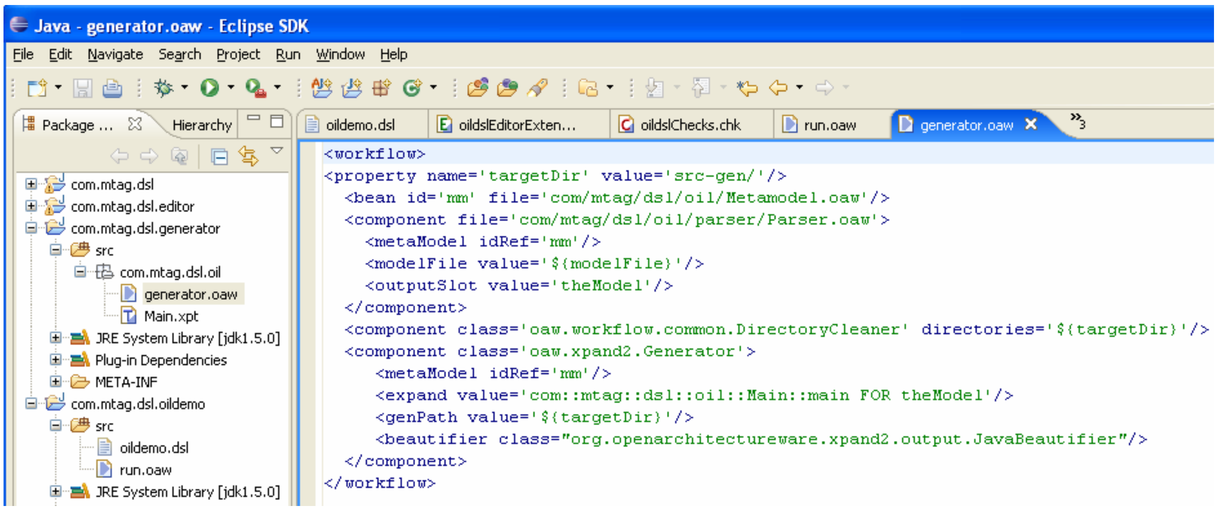


```
Strategy SVCL TargetClass oilobjects.OilService {
  from  4 to  8 property CustomerID
  from  9 to 30 property CustomerName
  from 31 to 32 property TypeCode
  from 33 to 55 property RequestText
}

Strategy INVO TargetClass oilobjects.OilInvoice {
  from  4 to  8 property CustomerID
  from  9 to 18 property InvoiceDate
  from 19 to 26 property InvoiceAmount
  from 27 to 60 property Text
}
```

Picture 27: Using the configuration language

We are finally coming to an end and all that's left to do is to invoke the code generation workflow that Xtext already generated for us within the generator project.

Before we invoke it, let's take a look at it.



```
<workflow>
<property name='targetDir' value='src-gen/'/>
  <bean id='mm' file='com/mtag/dsl/oil/Metamodel.oaw'/>
  <component file='com/mtag/dsl/oil/parser/Parser.oaw'>
    <metaModel idRef='mm'/>
    <modelFile value='${modelFile}'/>
    <outputSlot value='theModel'/>
  </component>
  <component class='oaw.workflow.common.DirectoryCleaner' directories='${targetDir}'/>
  <component class='oaw.xpand2.Generator'>
     <metaModel idRef='mm'/>
     <expand value='com::mtag::dsl::oil::Main::main FOR theModel'/>
     <genPath value='${targetDir}'/>
     <beautifier class="org.openarchitectureware.xpand2.output.JavaBeautifier"/>
  </component>
</workflow>
```

Picture 28: The generator workflow

The workflow starts with the definition of a property 'targetDir' set to 'src-gen'.
Next, a bean element pointing to the 'Metamodel.oaw' workflow is used to create a simple object to wire up the meta-model. If you look at the 'Metamodel.oaw' workflow, you can guess that it creates an eCore meta-model from the eCore file generated from our grammar.

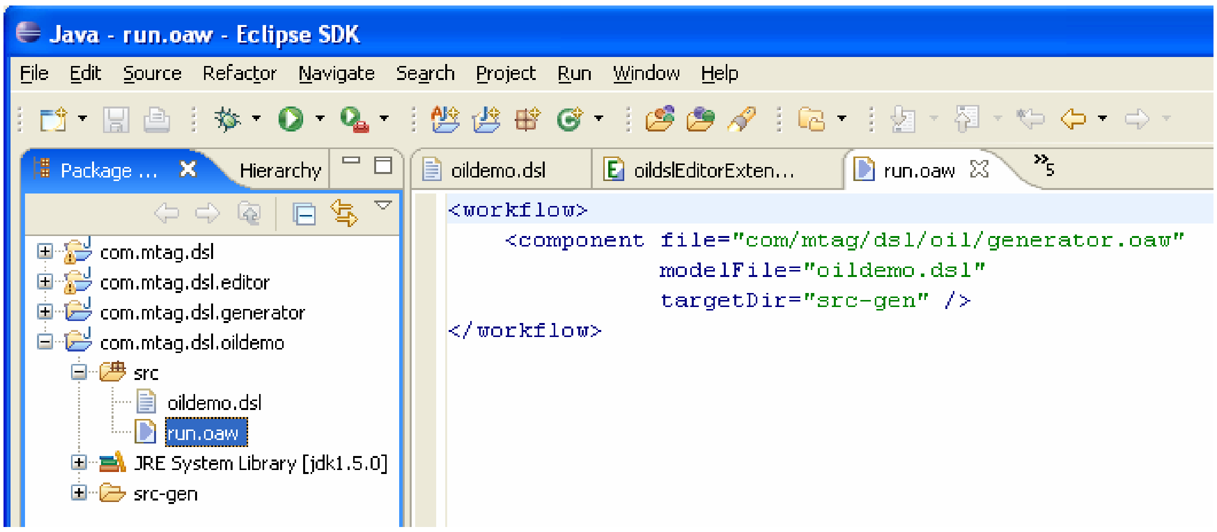
The bean element is followed up by a component for parsing our model. It references another workflow generated by Xtext. This component takes the meta-model and the location of the text file to parse as parameters. The model will be parsed, validated and sent to an output slot named ‘theModel’.

Slots are oAW’s means to communicate results between workflow components. Other components may access the parsed model by referring to the slot by name.

The next component simply deletes the target directory. Like in Ant, you can declare formal parameters like ‘$(targetDir)’ or ‘$(modelFile)’ instead of hard coding everything. Let’s just remember to supply values for these later, when we call this workflow.

The final component invokes the generator. The <expand> tag points to our Xpand template file and calls the main template for the model waiting in the output slot ‘theModel’. The generator component offers the option to specify a code beautifier to be run over the generated files. We use the one shipped with oAW, but you are free to install the one you prefer.

Now, having seen how model parsing, validating and code generation basically work, we are ready to invoke this workflow from within our Java project. We could use Ant or Maven to call an oAW workflow, but it can also be done completely within the oAW framework by creating another very simple workflow for the task.


Java - run.oaw - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package ... Hierarchy
oildemo.dsl oildslEditorExten... run.oaw
com.mtag.dsl
com.mtag.dsl.editor
com.mtag.dsl.generator
com.mtag.dsl.oildemo
src
oildemo.dsl
run.oaw
JRE System Library [jdk1.5.0]
src-gen

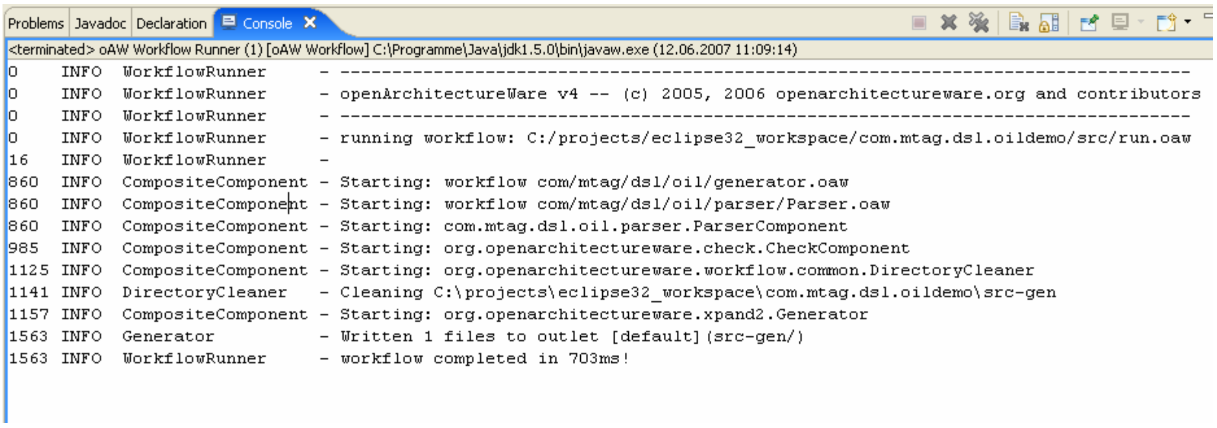
```
<workflow>
    <component file="com/mtag/dsl/oil/generator.oaw"
               modelFile="oildemo.dsl"
               targetDir="src-gen" />
</workflow>
```

Picture 29: Invoking the generator workflow

Within our new Java project, I’ve created a very small workflow called ‘run.oaw’. It consists of just one component pointing to the workflow we want to call within the generator project. As you remember, we have to supply some parameters to the generator workflow. This is done by simply defining the respective properties within the component.

Now, to finish this article, let’s run this workflow and have a look at the generated code. Right click the run.oaw file and select ‘Run as – oAW Workflow’.

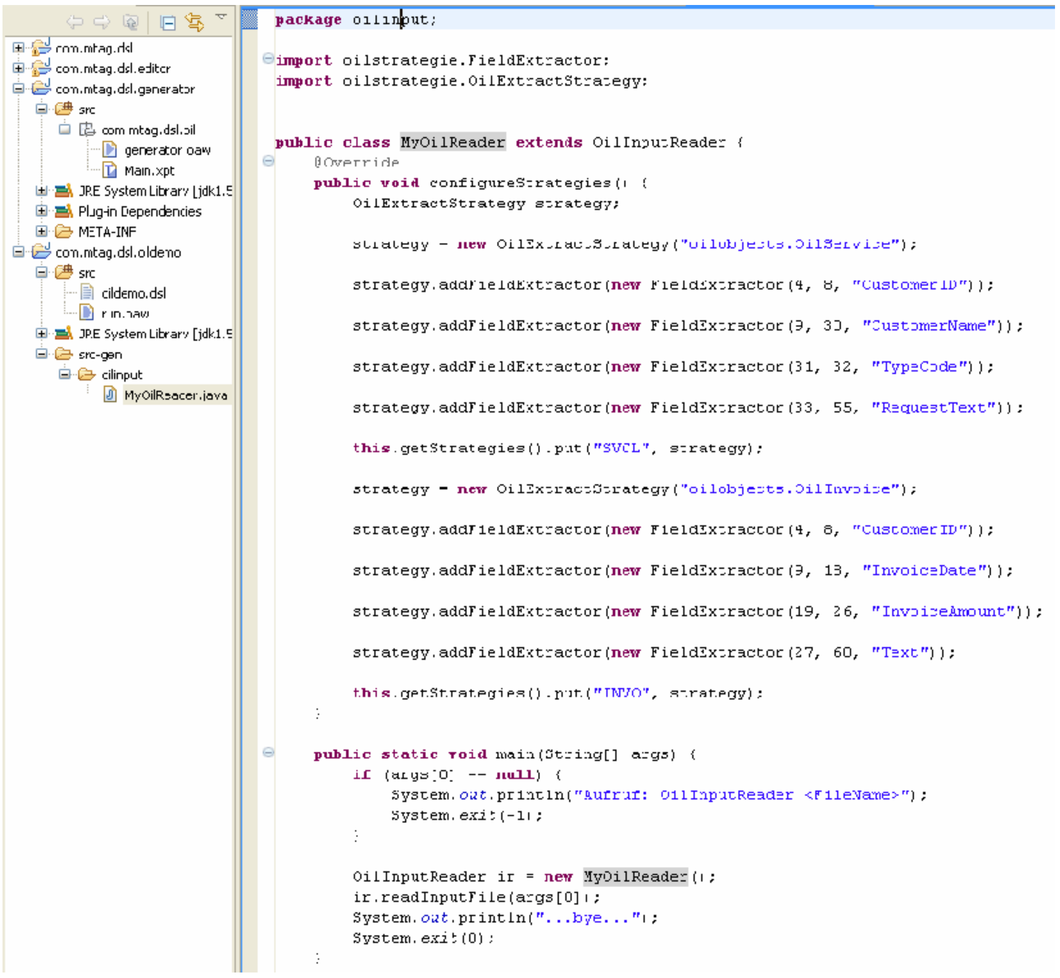
You should see something like this on your console:



```
<terminated> oAW Workflow Runner (1) [oAW Workflow] C:\Programme\Java\jdk1.5.0\bin\javaw.exe (12.06.2007 11:09:14)
0    INFO  WorkflowRunner      - --------------------------------------------------------------------------------
0    INFO  WorkflowRunner      - openArchitectureWare v4 -- (c) 2005, 2006 openarchitectureware.org and contributors
0    INFO  WorkflowRunner      - --------------------------------------------------------------------------------
0    INFO  WorkflowRunner      - running workflow: C:/projects/eclipse32_workspace/com.mtag.dsl.oildemo/src/run.oaw
16   INFO  WorkflowRunner      -
860  INFO  CompositeComponent - Starting: workflow com/mtag/dsl/oil/generator.oaw
860  INFO  CompositeComponent - Starting: workflow com/mtag/dsl/oil/parser/Parser.oaw
860  INFO  CompositeComponent - Starting: com.mtag.dsl.oil.parser.ParserComponent
985  INFO  CompositeComponent - Starting: org.openarchitectureware.check.CheckComponent
1125 INFO  CompositeComponent - Starting: org.openarchitectureware.workflow.common.DirectoryCleaner
1141 INFO  DirectoryCleaner    - Cleaning C:\projects\eclipse32_workspace\com.mtag.dsl.oildemo\src-gen
1157 INFO  CompositeComponent - Starting: org.openarchitectureware.xpand2.Generator
1563 INFO  Generator           - Written 1 files to outlet [default](src-gen/)
1563 INFO  WorkflowRunner      - workflow completed in 703ms!
```

Picture 30: The generator console

Open the src-gen folder and see what our template produced:



```
package oilinput;

import oilstrategie.FieldExtractor;
import oilstrategie.OilExtractStrategy;


public class MyOilReader extends OilInputReader {
    @Override
    public void configureStrategies() {
        OilExtractStrategy strategy;

        strategy = new OilExtractStrategy("oilobjects.OilService");

        strategy.addFieldExtractor(new FieldExtractor(4, 8, "CustomerID"));

        strategy.addFieldExtractor(new FieldExtractor(9, 30, "CustomerName"));

        strategy.addFieldExtractor(new FieldExtractor(31, 32, "TypeCode"));

        strategy.addFieldExtractor(new FieldExtractor(33, 55, "RequestText"));

        this.getStrategies().put("SVCL", strategy);

        strategy = new OilExtractStrategy("oilobjects.OilInvoice");

        strategy.addFieldExtractor(new FieldExtractor(4, 8, "CustomerID"));

        strategy.addFieldExtractor(new FieldExtractor(9, 18, "InvoiceDate"));

        strategy.addFieldExtractor(new FieldExtractor(19, 26, "InvoiceAmount"));

        strategy.addFieldExtractor(new FieldExtractor(27, 60, "Text"));

        this.getStrategies().put("INVO", strategy);
    }

    public static void main(String[] args) {
        if (args[0] == null) {
            System.out.println("Aufruf: OilInputReader <FileName>");
            System.exit(-1);
        }

        OilInputReader ir = new MyOilReader();
        ir.readInputFile(args[0]);
        System.out.println("...bye...");
        System.exit(0);
    }
```

Picture 31: The generated code

Apart from some minor differences in formatting, this reflects our manually written code from Part I.
From now on, we will do all the configuration of our object extraction and instantiation problem solely within our domain specific editor. These configurations will be transformed into real code by just a single mouse click.

You can now process any input file, regardless of its structure. Just use our language editor and describe the file's structure in our specific language. Then generate the configuration code. No more manual coding is needed.

Summary

This article described the process of discovering and implementing a simple domain-specific language. We started out with an example given by Martin Fowler, externalised its DSL and based it on an appropriate grammar. Next we generated an Eclipse editor to support this grammar and 'speak' our new language with full IDE support. Finally, after enhancing the editor and adding some logic to the underlying meta-model, we implemented a code generator to transform our textual configurations into real code.

This is by no means an overly complicated problem domain, so the advantages of implementing a DSL for it may not be too obvious. Nevertheless, let's briefly summarise the gains:

- **Configuring the file extractor is no longer done by coding. By externalising the DSL, configuration errors will now be detected during modelling, not at program runtime.**
- **We've raised the level of abstraction for configuring the file processing. Originally, configuration took place on code level, which is probably out of scope for most domain experts (don't take the word 'expert' too seriously here). Now it can by done by anyone who knows the structure of the input file and 'speaks' our configuration language.**

Even within this simple domain, we clearly scored on the grounds of quality improvement and complexity reduction. Transfer this to more demanding domains and the potential of external DSL will become obvious.

Remember, we did not generate 100% of the application. Instead, we hand coded our concepts, hopefully only once, and generate 100% of the configuration code for any given file structure.

Finally, consider the effort we had to invest. It took a couple of pages to describe the process, but when you count the actual lines of code we had to write and the number

of clicks we had to perform, you may agree that this does not amount to anything worth worrying about. The framework kept most of the complexities away from us.

It is the power of frameworks like oAW that let us leverage the potential of DSLs and related concepts into our daily projects.

Further Work

While writing this article, two opportunities for DSL problem solutions showed up in the project I'm currently working on. One deals with maintaining a couple of database tables with a special DSL for defining table contents. The other is about describing the transformation of financial data into a format readable by SAP's Basel II Bank Analyser Application.
You will find something about these on our website as soon as time (and client) permits.

A click-by-click tutorial for this article is making progress. It will add nothing to the problem domain but will elaborate on Eclipse and oAW configuration issues. You won't miss a click there.

Eclipse is a powerful but non-trivial framework. More than once, I found myself caught up more in Eclipse than in oAW or problem domain issues. A list of tips and best practices I found on the web and put to trial will be compiled and published here.

Further Reading

Here's some further reading that I found very helpful. It is a personal selection and, of course, incomplete.

Martin Fowler
Workbenches: The Killer-App for Domain Specific Languages?
http://www.martinfowler.com/articles/languageWorkbench.html#ASimpleExampleOfLanguageOrientedProgramming.
A Must Read.

openArchitectureWare Framework (oAW)
http://www.openarchitectureware.org
A most innovative open source project for model driven code generation.

Eric Evans
Domain-Driven Design: Tackling Complexity in the Heart of Software.
Read this, to learn about modelling and abstractions.

Sergey Dmitriev
Language Oriented Programming: The Next Programming Paradigm

http://www.onboard.jetbrains.com/is1/articles/04/10/lop/
Read about JetBrains' visions concerning Language Workbenches. I love this article, because it describes the problems of us software developers precisely.

The ANTLR Project
http://www.antlr.org/
Terence Parr's "Another Tool for Language Recognition" Project Homepage. This is the place for gaining deeper understanding of how Xtext works.

Niklaus Wirth
Compiler Construction
Zürich, November 2005
I must admit that compiler construction is not my favourite topic. But Wirth presents it in a readable fashion.

About the Author

Volker Koster is an Executive Architect at MT AG, Ratingen, Germany.
He's been on software projects since 1990, mostly in Oracle and Java environments.

You can contact him by e-mail at volker.koster@mt-ag.com.
Your comments are very welcome.

Balcke-Dürr-Allee 9
40882 Ratingen
Tel. +49 (0) 21 02 309 61-0
Fax +49 (0) 21 02 309 61-10
info@mt-ag.com
www.mt-ag.com