Whole Platform LWC11 Submission

Riccardo Solmi Enrico Persiani

May 22, 2011

Contents

Chapter 1

LWC11-Submission

1.1 LWC11 Submission using the Whole Platform

This is the documentation of the **Whole Platform** submission to the **Language Workbench Competition 2011 (LWC11)** . The assignment can be found at http://www.languageworkbenches.net/LWCTask-1.0.pdf . For more details and to find the others submissions see http://www.languageworkbenches.net/ .

How to install the Whole Platform

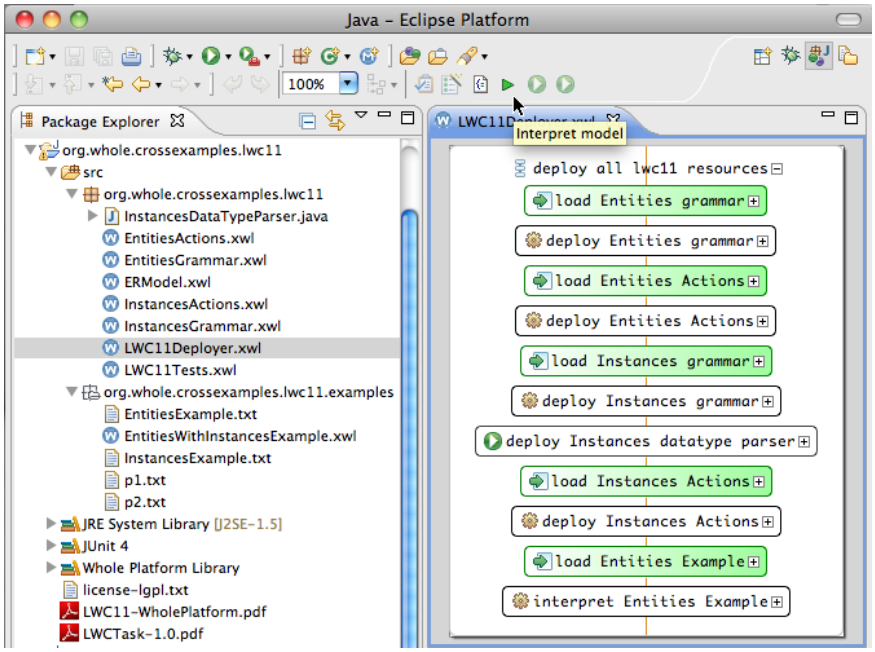
Before installing the Whole Platform be sure to have at least a working Java 5 or higher. The easiest way to install the Whole Platform is to click on the **Download** button on the Whole Platform SourceForge homepage . Upon download completion simply unpack the archive and execute the provided launcher (you can safely use the proposed workspace location). The version used for developing the solution described below is: 1.0.0.v20110512-1609.

How to install the solution

The source code of the solution described below can be found at http://whole.sourceforge.net/docs/LWC11-WholePlatform.zip . A PDF version of this document can be found at http://whole.sourceforge.net/docs/LWC11-WholePlatform.pdf and is also included with the source code archive.

Use the **File > Import... > Existing Projects into Workspace** wizard to import the downloaded archive. Then open

the **LWC11Deployer.xwl** and deploy it by clicking on the *Interpret Model* button on the toolbar. At this point, the Language Workbench should look like this.



Now you are able to open every other artifact in the solution including the examples.

Solution Overview

The solution presented below has been developed having in mind two goals:

- to exploit the benefits of a graphical language workbench by providing a solution at the *domain level* and
- to support an agile approach by using *model interpretation* instead of code generation to apply the solution.

The solution consists of 8 artifacts:

- two grammars for the **Entities** and the **Instances** DSLs
- one metamodel for the **ER** DSL
- two actions for defining and exposing as tooling all of the generators, validator and content assist
- a deployer to hot deploy the solution in the running workbench
- a test suite for testing the grammars and the generators

- a custom DataType parser to support the date format used in the provided examples

With graphical domain languages, *lines of code* is no longer a suitable metric to measure the size and thus the conciseness of a solution. Let us show you that almost all of the solution code can be visualized in a 27 inches monitor.

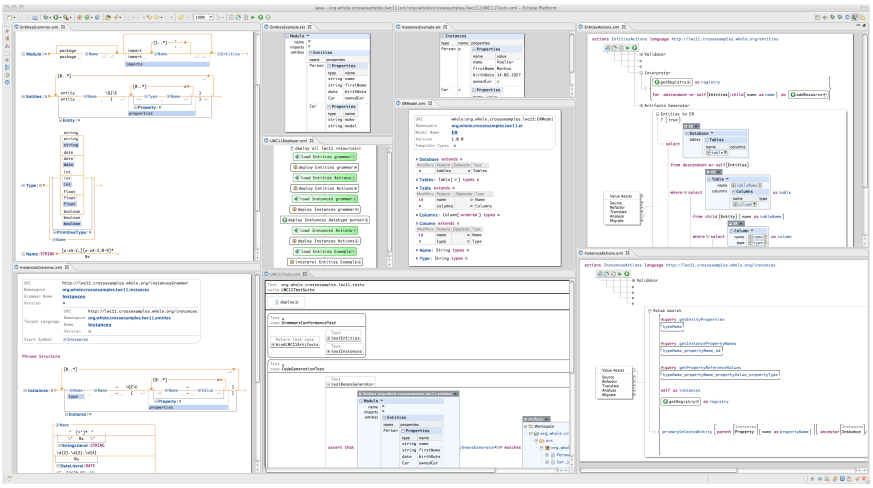


Figure 1.1: Solution Overview

1.1.1 Phase 0 - Basics

This phase is intended to demonstrate basic language design, including IDE support (code completion, syntax coloring, outlines, etc).

- Task 0.1 Simple (structural) DSL without any fancy expression language or such.
- Task 0.2 Code generation to GPL such as Java, C#, C++ or XML
- Task 0.3 Simple constraint checks such as name-uniqueness
- Task 0.4 Show how to break down a (large) model into several parts, while still cross-referencing between the parts

1.1.2 Phase 1 - Advanced

This phase demonstrates advanced features not necessarily available to the same extent in every LWB.

- Task 1.1 Show the integration of several languages
- Task 1.2 Demonstrate how to implement runtime type systems
- Task 1.3 Show how to do a model-to-model transformation
- Task 1.4 Some kind of visibility/namespaces/scoping for references

- Task 1.5 Integrating manually written code (again in Java, C# or C++)
- Task 1.6 Multiple generators

1.1.3 Phase 2 - Non-Functional

Phase 2 is intended to show a couple of non-functional properties of the LWB. The task outlined below does not elaborate on how to do this.

- Task 2.1 How to evolve the DSL without breaking existing models
- Task 2.2 How to work with the models efficiently in the team
- Task 2.3 Demonstrate Scalability of the tools

1.1.4 Phase 3 - Freestyle

Every LWB has its own special "cool features". In phase three we want the participants to show off these features. Please make sure, though, that the features are built on top of the task described below, if possible.

- Task 3.1 Integration with the platform native modeling language
- Task 3.2 Testing
- Task 3.3 Debugging

Chapter 2

LWC11-Task-0.1

This page is part of the Whole Platform LWC11_Submission .

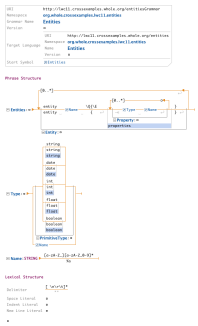
2.1 Task

0.1 Simple (structural) DSL without any fancy expression language or such

Build a simple data definition language to define entities with properties. Properties have a name and a type. It should be possible to use primitive types for properties, as well as other Entities.

2.2 Screenshots

EntitiesGrammar.xwl



2.3 Overview

In a model driven approach, a language consists of a single mandatory part: the structure or (meta)model, and zero or more notations, persistences and other (mainly translational) semantics. By taking a domain specific approach, the Whole Platform provides several languages targeted at language definition. Each language is focused on one domain representing an aspect of language definition (i.e. structure, concrete syntax, notation, tooling, generators).

The more straightforward way to define a language in Whole is to define only its structure using the **Models** DSL and let the Platform provide generic notations and persistences for the new language instances. We will illustrate this solution at the end of this chapter.

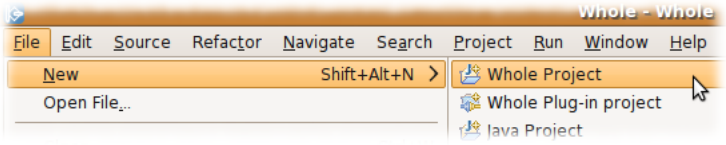
Here, we consider as a requirement the ability to parse/unparse the textual syntax used in the LWCTask examples. So, we start defining the grammar of the language using the **Grammars** DSL and we let the Platform derive the structure and notation.

The two solutions are in fact complementary: we can regard the latter as a way to add a specific textual persistence to a language and conversely we can regard the former as a way to specify the desired structure for a grammar defined language.

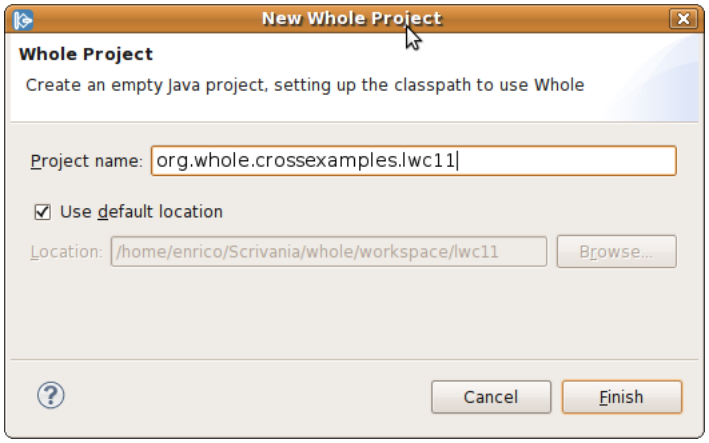
2.4 Details

2.4.1 Creating a Whole Project

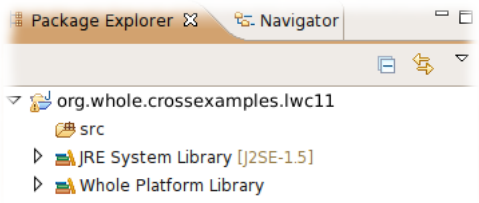
Creating a Whole Project is a fairly straightforward task. From the Eclipse menu bar, select **File > New > Whole Project...**



In the **Project name** field, type `org.whole.crossexamples.lwc11` and click **Finish** when you are done.



Eventually, you'll end up with a workspace containing the newly created Whole Project.



A Whole Project it's basically a Java Project configured to include the **Whole Platform Library** in the build path.

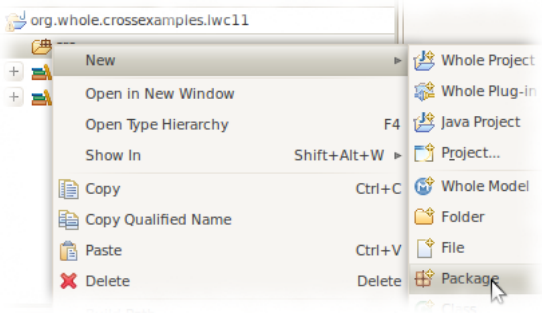
2.4.2 Creating a Grammar Model

The Whole Language Workbench provides several meta-languages that can be used to define new languages. The primary responsibility of a meta-language is to define the abstract syntax of the language being modeled. Meta-languages may also define concrete syntaxes, graphical notations or other language specific features.

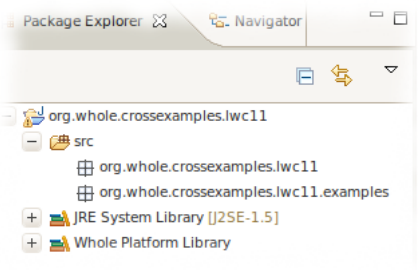
It is important to note that the Whole Language Workbench doesn't impose any distinction between languages and meta-languages. A language can be considered implicitly a meta-language by adding the means to define new languages using its own instances. **Grammars** is such a language, and allows the definition of new languages (both abstract and concrete syntax) using an EBNF like notation.

To create a **Grammar model** we have to use the **New Whole Model** wizard, note that throughout this document we will use the term *model* as a synonym of language instance. But to keep things clean, at first we will create two package. A first package, named `org.whole.crossexamples.lwc11` , to contain the language meta-models and the associated behavior. The second package, named `org.whole.crossexamples.lwc11.examples` , to store the provided sample models.

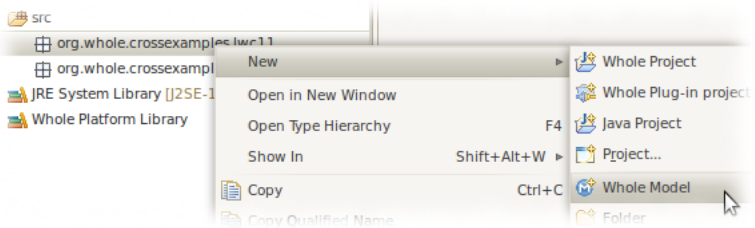
To create a new package, right-click on the **src** folder and select **File** > **New** > **Package** .



Add both packages using the **New Java Package** wizard. Eventually, the updated project will contain the newly created packages.

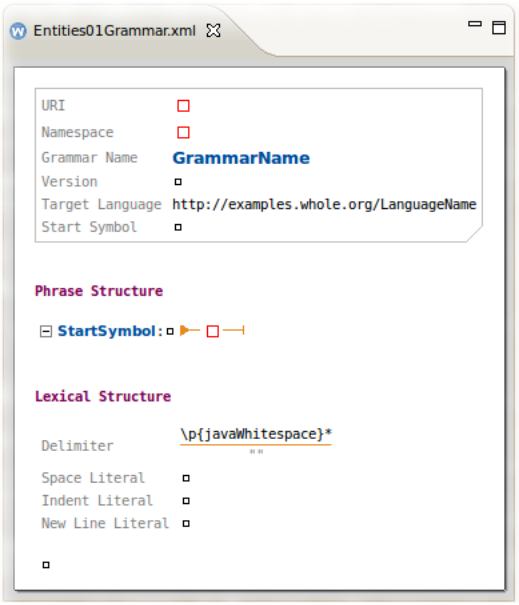


Now we can create a **Grammar model** by right-clicking on the **org.whole.crossexamples.lwc11** package and selecting **File > New > Whole Model** .



The **New Whole Model** wizard window allows us to specify several details regarding the model being created. The upper area allows the selection of the containing folder, that we implicitly choose by right-clicking on the destination package. After the destination path we can type `EntitiesGrammar.xwl` in the **File Name** field. Finally, we can choose **Grammars** as the target language with an **empty** starting template saved using the generic **XML (Whole Template Builder)** persistence.

A resource named **EntitiesGrammar.xwl** will be created in the specified folder, and a corresponding **Whole Graphical Editor** will be opened showing the newly created resource contents.



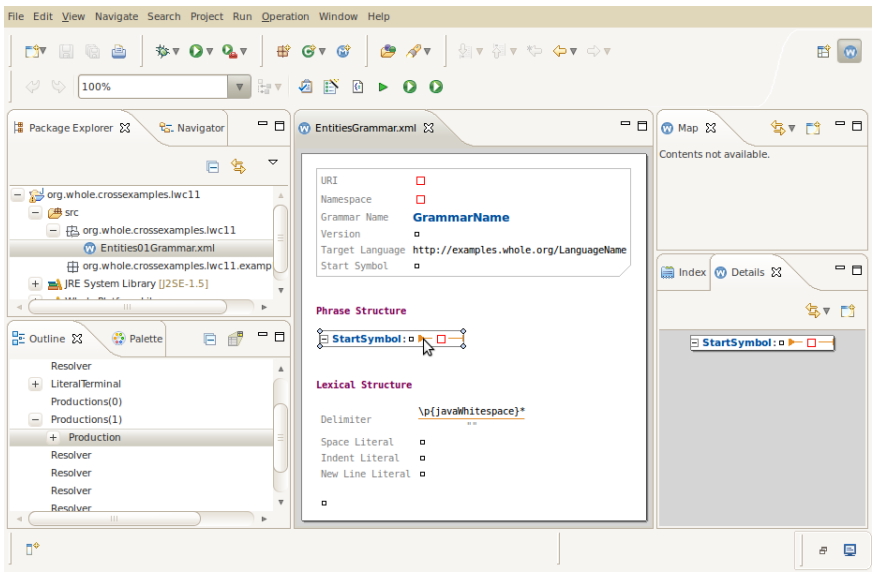
The graphical editor shows a skeleton grammar that can be customized to fit our needs. Before diving into the required grammar customization steps, it is useful to understand some basic editing principles of the Whole Platform. In the Whole Platform, language constructs are generically referred as **Entities** . There are four main kinds of entities:

1. **Simple entities** represent a list of named features
2. **Composite entities** represent collections of entities
3. **Data entities** represent data values
4. **Enum entities** represent enumerated values

The preferred way of editing a model is the **Whole Graphical Editor** . Since we are using a graphical editor, all the editing operations are performed using the provided content assist (by using the context menu or the Meta+Space keystroke) or by means of drag'n'drop operations. The set of allowed operations will be restricted by the abstract syntax constraints of the target entity. Other ways to inspect the model's contents are the **Outline View** (that is always bound to the active editor) and the projection views (i.e. the **Details View**).

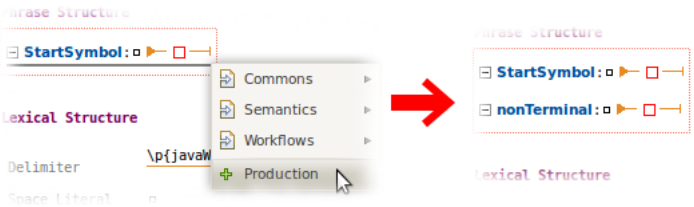
Several notations are provided for every language family, some of them are generic (i.e. available for a family of languages) other are language specific. Some notations may hide parts of the models being edited (often to reduce complexity), but the Outline always shows the entire tree structure. Notations can be changed using the proper context menu item.

In the following screenshot a **Whole Graphical Editor** is shown in the central pane of the workspace window with the **WHITESPACE** lexical production being selected. The Outline on and the Details view show the same selected production in different ways.



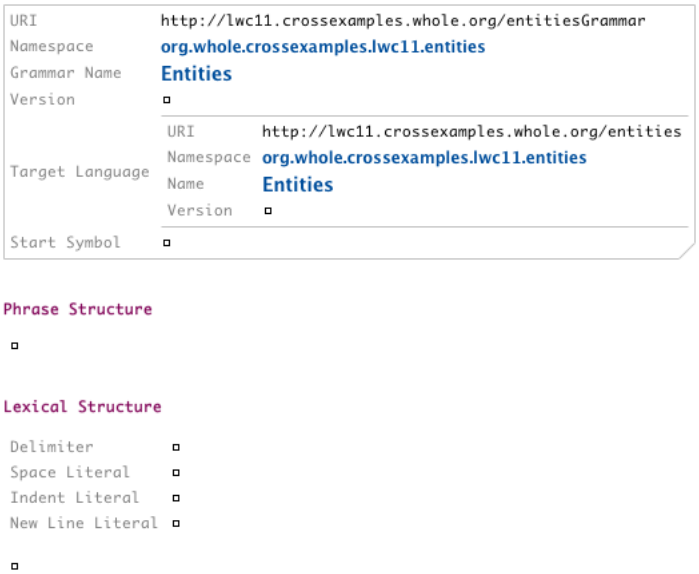
In the graphical editor depicted above several red and black squares are shown. They are placeholders that can be replaced with concrete language entities. In fact, the red ones must be replaced because they represent mandatory features, while the smaller black ones are optional.

A placeholder figure is shown also on collection figures. While hovering a composite figure, the insertion point is highlighted using an horizontal or vertical narrow line. Both the context menu actions and the drag'n'drop operations use that hint to modify the underlying figure.



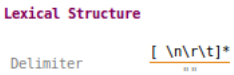
Now we are ready to modify our first grammar model. We start defining some features to identify the grammar: the uri (used to uniquely identify grammars inside the platform), the namespace (used as package prefix for artifacts produced by generation actions) and the grammar name. Finally, we have to insert a *LanguageDescriptor* in the target language feature filled with analogous information to identify the target language to which this grammar is bound.

As we previously said, to replace a placeholder use the content assist. To avoid confusion, in this part of the tutorial simply stick to the **Grammars** submenu of the content assist menu. To enter text editing mode it's enough to double-click on any editable label, to terminate text editing simply press the carriage return or the escape key. Remove any other entity below the top level pane so that the end result is similar to the figure shown below.



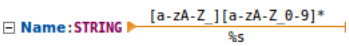
Note that we can have several grammar models bound to a single target language, because the target language defines an abstract syntax while each grammar defines a concrete syntax. Also, if the target language has not been already defined into the Whole Platform, there's a default mechanism to derive it from the grammar model itself.

The next step is to define a delimiter, using a *Literal Terminal* that consumes whitespace sequences. The delimiter can be used to determine the start and end position of the next token to be consumed.



The literal terminal allows to specify a regular expression used to consume input during a parse operation, and a literal string to produce output during an unparse operation. The parse operation is useful to transform a stream of characters into an in memory model, while the unparse operation produces the opposite effect.

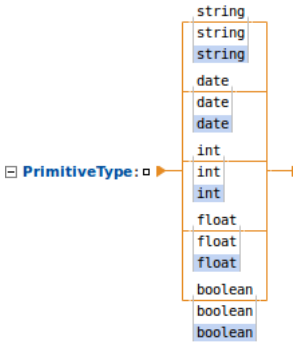
We can now define the first *Production* inside the phrase structure.



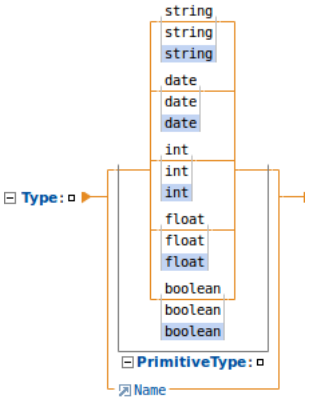
The **Name** production defines a rule to parse an identifier using a *Data Terminal* . During the parse operation data terminals behave similarly to literal terminals, but the captured data is stored in memory for later reuse. When the unparse operation is invoked data terminals output the previously stored data using the provided formatter.

It is worth noting that since the **Name** production contains only a data terminal, it will be mapped to a data entity in the target language. The **STRING** value next to the production name is used define the data-type of the mapped data entity (this is particularly useful in case we have to derive the target language from the grammar model itself).

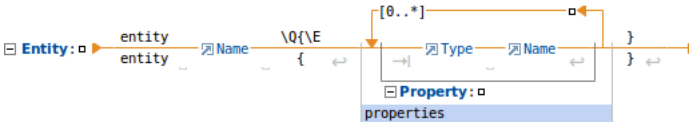
The next production we define is called **PrimitiveType** . Basically, the main rule uses a *Choose* construct to define the set of alternatives that may be encountered during the parse operation, the values are matched using literal terminals. Since the production matches only literal terminals, it will be mapped to an enum entity in the target language. The **As** rules are used to explicit the mapping between literals and enum values.



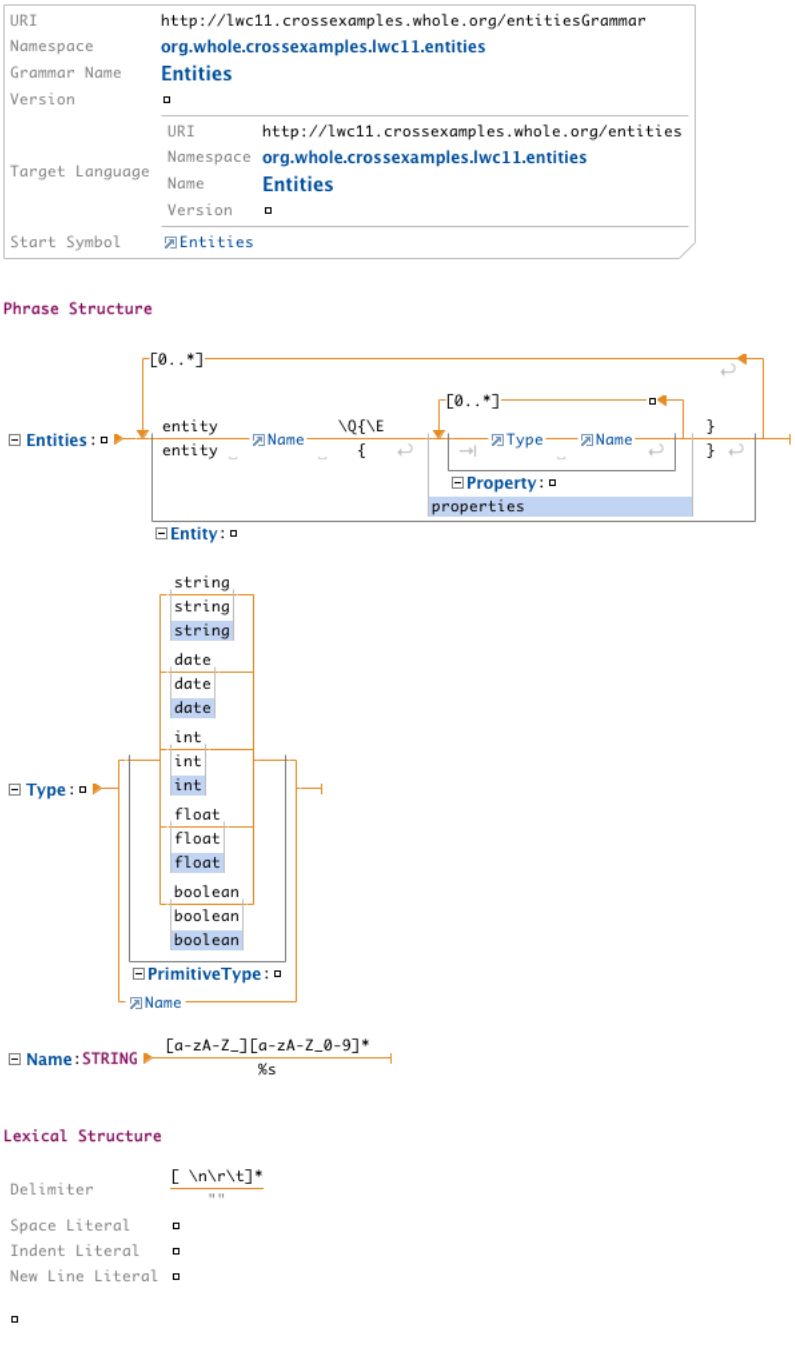
We can now define the **Type** production as the alternative between the **Name** and the **PrimitiveType** rules, note that Grammar models allow arbitrary nesting of productions.



The **Entity** production uses a *Concatenate* rule to define the sequence of *LiteralTerminal* s and *NonTerminal* s to be parsed. It has a nested *Repeat* rule that defines an unbounded collection of properties. Each property will be mapped using its own **Property** production. Note that a few non consuming whitespace rules (*Space* , *Indent* and *NewLine*) are added to produce a prettier output.



Finally we define the **Entities** production to be an unbounded collection of entities. Note that the **Entities** non-terminal is specified as the default starting symbol. The complete grammar is shown below.

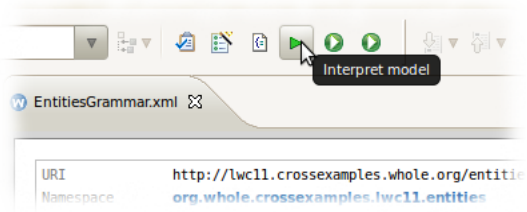


2.4.3 Using the Grammar Model

The easiest way to use the grammar model is to *deploy* it into the Whole Platform. We use the term *deploy* in several contexts to indicate that we are enriching the runtime with the resource being deployed. In this specific case, we expect to

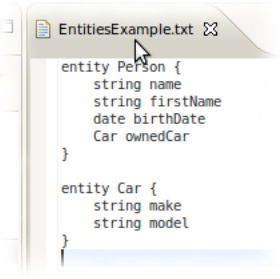
add to the platform the target language derived from the grammar model and the configurations needed to persist such a language using the concrete syntax.

To deploy the grammar model we have to invoke the *interpreter operation* . The Whole Language Workbench provides a toolbar that allows the user to invoke a set of predefined operations on languages. The workbench enables only the actions that can be executed on the active editor. So make sure that the **EntitiesGrammar** is the active editor and click on the interpret action, as shown below.



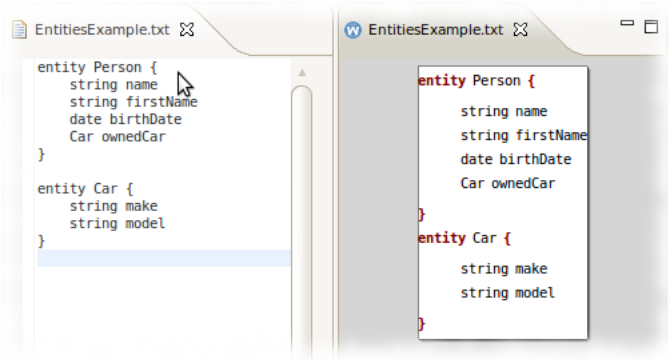
Now we can proceed by creating an example text file to be parsed. Using the **New File** wizard create an empty file inside the *org.whole.crossexamples.lwc11.examples* package and name it `EntitiesExample.txt` . The workbench will open a text editor to modify the newly created file contents. Copy the example provided by *The Task* in the open editor and save.

```
entity Person {
    string name
    string firstName
    date birthDate
    Car ownedCar
}

entity Car {
    string make
    string model
}
```



The example entities file can be opened using the already deployed grammar. Right-click the *EntitiesExample.txt* and select **Open With > Grammar Based Persistence** .

A new graphical editor showing the entities example content is opened. By dragging the graphical editor's tab on the right side of the editor area, you can put the graphical and textual editors side by side.



Experiment with both the editors by performing changes on one side, and looking at how they are reflected on the other side.

Chapter 3

LWC11-Task-0.2

This page is part of the Whole Platform LWC11_Submission .

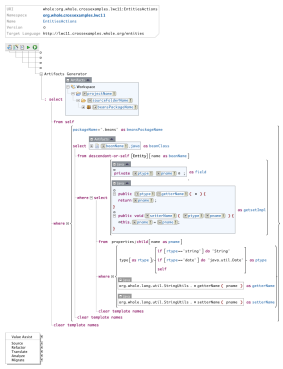
3.1 Task

0.2 Code generation to GPL such as Java, C#, C++ or XML

Generate Java Beans (or some equivalent data structure in C#, Scala, etc.) with setters, getters and fields for the properties.

3.2 Screenshots

EntitiesActions.xwl



3.3 Overview

In the Whole Platform, code generation is performed by the **Artifacts Generator** operation and consists of a regular model-to-model transformation followed by an implicit and automatic model-to-storage synchronization. We don't support the old model-to-text idiom of generating a target code starting directly from a source model for the reasons we will outline below.

In general, code generation is a process involving two orthogonal activities: transformation and persistence. The transformation is tied to a given code gen-

erator whereas the persistence is a reusable service configured by the output of the transformation.

The complexity of the transformation is greatly reduced just by separating the two activities. For instance you are not constrained by the ordering of the target persistence format.

The availability of the target language is not an issue. Firstly, the Platform already includes popular legacy languages such as **Java** , **Objective C** , **XML** , and **XSD** . Secondly, if the target language is not bundled you can choose either to model it or to use the **Text** DSL.

A separate model-to-model transformation can be incremental with respect to the target model. For instance, the source structure-oriented **Queries** DSL can produce multiple target fragments at once whenever it encounter a piece of information on the source model.

Furthermore, a separate model-to-model transformation can be decomposed in multiple simpler and reusable steps, each reflecting a clear intention. For instance, model completion and normalization can be applied both at source and target ends.

To resolve this task, we propose a solution consisting of an Artifacts Generator action that defines a model-to-model transformation from an Entities model to an Artifacts model. The **Artifacts** model defines the output files with their Java model contents that we want to generate into the (Eclipse) Workspace.

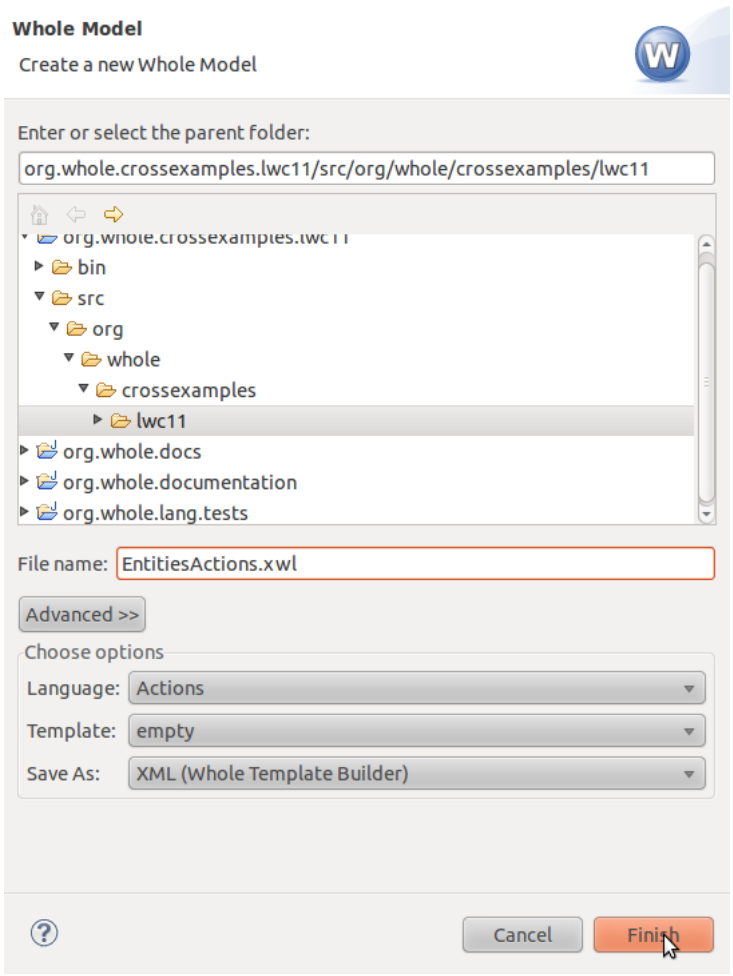
Keep in mind that the generator does not care about the notations and persistences. So, even if you have defined the Entities language starting from its grammar, you have to reason about it in terms of its (meta)model much like as we have done at the end of the previous chapter.

Whole languages used: **Actions** , **Artifacts** , **Queries** , **Commons** , **Java** .

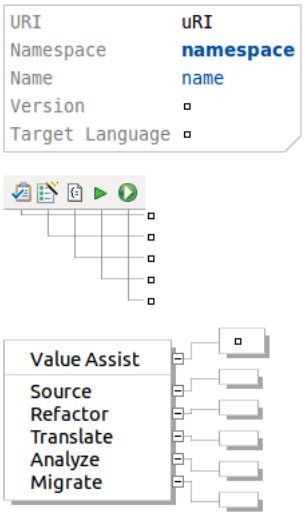
3.4 Details

The following guide describes the steps required for the creation of a Java Beans generator that uses the Entities language as its specification model. More specifically, the generator will create a Java Bean for each declared Entity. Each bean will be placed in the **.beans** sub-package of the Entities model container package.

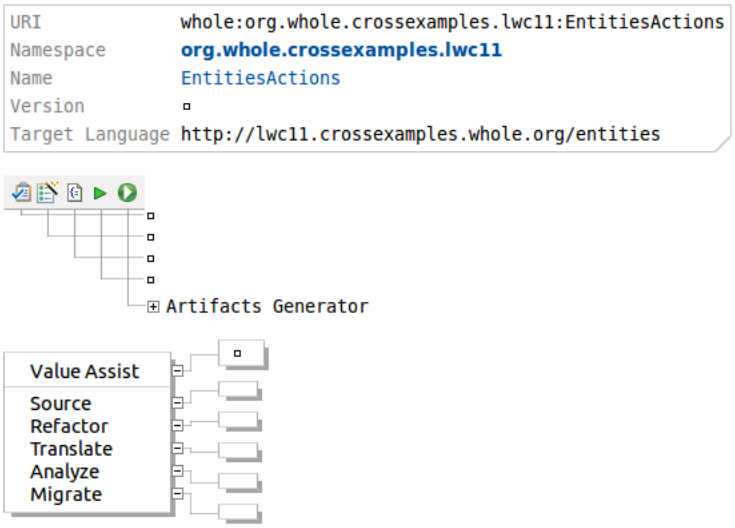
Create a new empty **Actions** model using the new model wizard. Put it in the usual `org.whole.crossexamples.lwc11` package using the name `EntitiesActions.xwl` .



The newly opened editor will show an empty **Language Action Factory** .

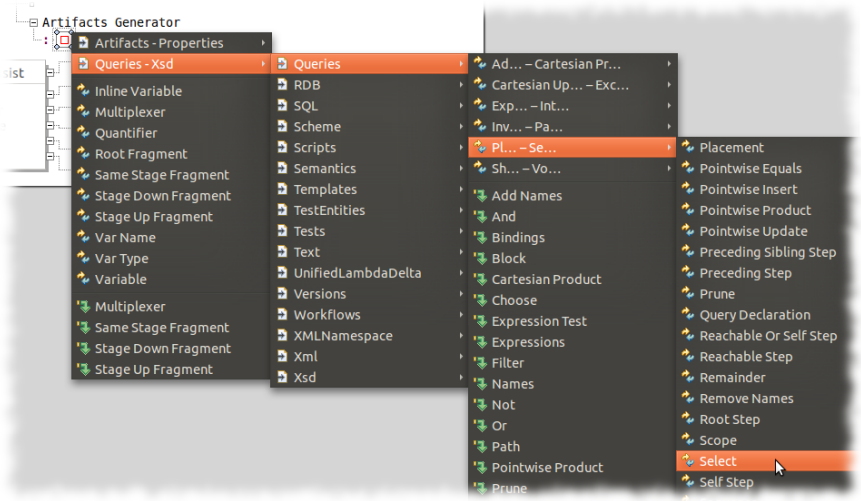


Replace the URI value with `whole:org.whole.crossexamples.lwc11:EntitiesActions` and that of the Namespace with `org.whole.crossexamples.lwc11` . Set `EntitiesActions` as the actions name and `http://lwc11.crossexamples.whole.org/entities` as the Target Language (instead of writing the uri, it is possible to choose the *Entities* language from the content assist menu). Add a `SimpleAction` to the artifacts generator toolbar action, and call it `Artifacts Generator` .



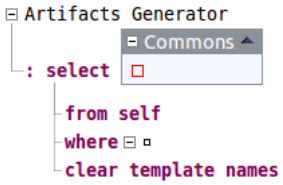
The input of an action is the root entity of the model on which it is being invoked. To generate artifacts we have to develop a model-to-model transformation that takes as input an *Entities* model and produces as output an *Artifacts* model. The resulting model will be used to create concrete artifacts on the file-system.

By clicking the toggle next to the action a placeholder is revealed. An action accepts any entity type for its definition, so the content assist invoked on the placeholder presents us a list of options that includes all the possible entities of the currently deployed languages. We start off by adding a **Select** statement of the **Queries** language.



Queries is the language of choice of the Whole Platform to perform model-to-model transformations. The *Select* statement defines a one to one mapping between the entities matched by the *from* clause and those created by the *select* clause. The from clause is an expression applied to the input and can match many entities. As soon as an entity is matched, it is used as input to the select expression to produce some other output entity. Finally, the *where* clause is evaluated to further refine the output entity.

Our objective is to create an Artifacts model from an Entities model. We will use a template on the select clause and a **SelfStep** of the Queries language on the from clause. To define a template the Whole Platform provides the **StageUpFragment** of the Commons language.

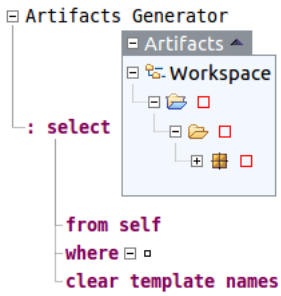


Note

Every *Queries* expression is evaluated in the context of a self entity. An Actions action sets the self entity to be the root entity of the input

model. Queries constructs can also change the self entity for the evaluation of nested expression. In this case **SelfStep** acts like an identity expression, leaving the self entity unchanged.

To create an entity we have to revert to the **StageUpFragment** and populate it with the desired resulting entity. We construct a **Workspace** , containing a **Project** which in turn has a child **Folder** , and eventually a **Package** .



The above fragment resembles a typical Eclipse Java Project structure.

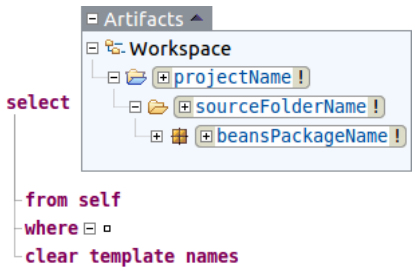
Now we have to define the artifacts names, otherwise it would be impossible for the generator to create them (you should recall that a red placeholder indicates a mandatory feature). Instead of using fixed values we opt for for a solution that allows us to define them at generation time. For this task we'll make use of the **Variable** construct of the **Commons** language.

Commons **Variables** are named entities with a type and a quantifier. They can be used both in *factory semantics* and in *pattern matching semantics* scenarios. Pattern matching is used to match a fragment against a pattern, such pattern contains **Variables** that are used to capture the corresponding parts of the fragment being matched. After a successful match, all the matched variables will be defined in scope using the provided name and the matched value. Factory semantics allows the construction of a fragment starting from a pattern that contains **Variables** to be lazily replaced with variables values in scope.

Both factory semantics and pattern matching semantics constraint replacements and matches using the provided **Variable** type and quantifier. Specific language constructs can enforce **Variable** semantics or refine constraints.

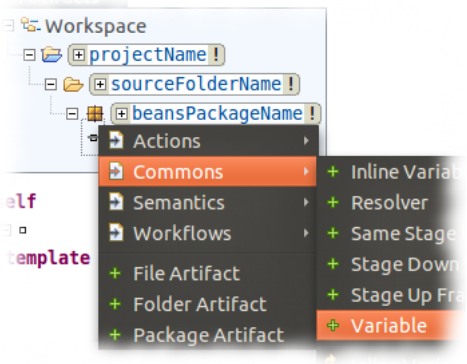
The **StageUpFragment** in the *select* clause of the **Select** statement enforces factory semantics. Through the *clear* clause the strategy of clearing the **Variables** can be altered.

We can proceed by adding the required **Variables** for the names of the artifacts that will be generated. Set variable names as shown in the following figure.

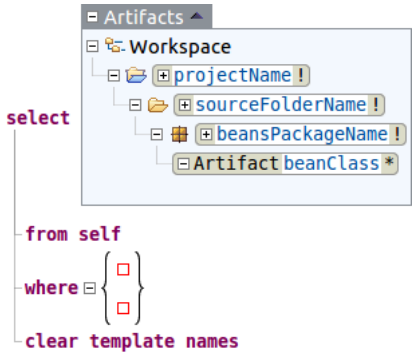


Remember that for the variable's type the content assist proposes only the replacements that are assignable compatible to the variable formal type.

Finally we expand the package artifact node and we add a variable to its artifacts list. Call this last variable it `beanClass` and set `*` as the quantifier (that will allow multiple additions).



To complete the generator, we must add in the *where* clause the required statements to generate the replacements for the **Variables** defined in the *select* clause. Since we will add several statement we start adding a **Sequence** construct of the **Queries** language.

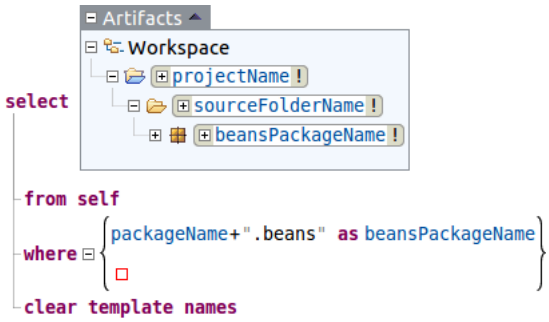


Since the **Sequence** is a collection of other statements, the two red placeholders can be either replaced or deleted.

The first value we want to calculate is the `beansPackageName` substitution, obtained by concatenating the `.beans` suffix to the current package name. To achieve this result we proceed using an **Addition** binary expression, setting a **VariableRefStep** as the left operand and a **StringLiteral** as the right operand. We continue by renaming the left operand to `packageName` (that is already defined in scope as the current package name at generation time) and by changing the value of the right operand to `.beans` .

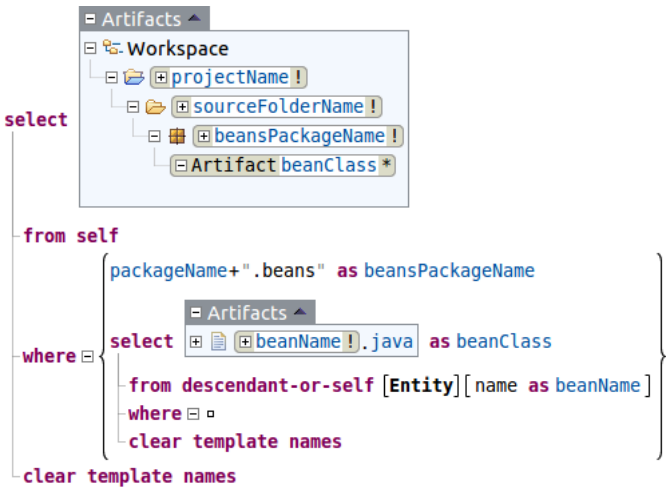
To achieve the substitution we must bind the addition expression to a name, for that purpose we wrap the addition with a **Filter** together with a **VariableTest** predicate. Finally, we rename the newly added **Variable Test** to `beansPackageName` .

The end result is shown in the following figure.



Now that we have defined the name of the target package, we can proceed by defining the file artifacts for the classes we want to generate. For this purpose we will use a nested **Select** statement that will map each **Entity** in the source model to a **FileArtifact** in the target model. Every generated **FileArtifact** will be eventually added to the **PackageArtifact** by replacing the `beanClass` .

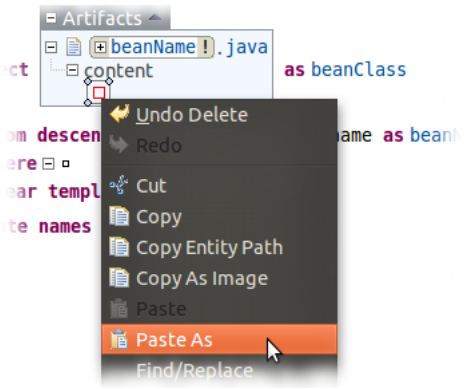
All the **Entities** are extracted from the source model by applying the **TypeTest** to the top down traversal of the source model made through the use of the **DescendantOrSelfStep** construct. Using an **Expression test** , each extracted entity name is bound to the *beanName* variable that will be used later in several places (the variables defined in the *from* clause are in scope also in the *select* and *where* clause). Finally, a **FileNameWithExtension** is used for the file artifact name to fix the `java` file extension.



We continue by defining the contents of the file artifact, that is the generated bean Java code. To simplify this process we provide a simple skeleton Java class that can be copied into the clipboard:

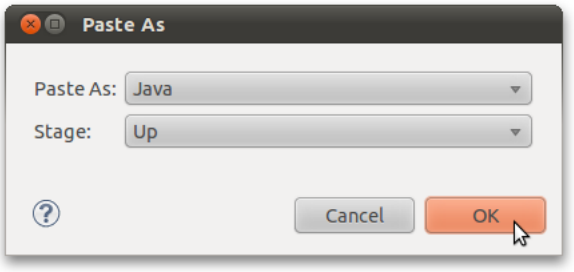
```
package my.pkg;

public class BeanName {
}
```

Clipboard contents can be easily pasted into the file artifact content feature as shown in the following figure.



Note

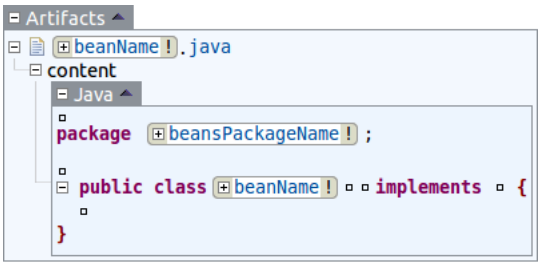
The **Paste As** action is available only in the context menu by clicking the right mouse button. A dialog prompting for a persistence and a stage level will be shown, just select **Java** and **Up** .



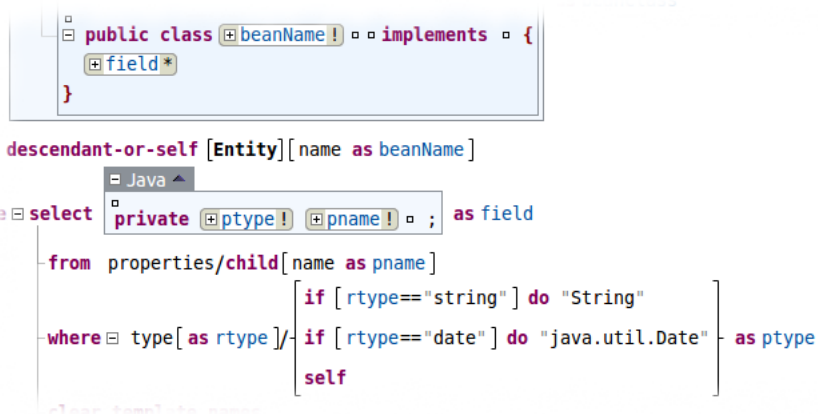
The final result will be a new **StageUpFragment** containing the modeled counterpart of the pasted Java source code.

Now you could ask, why are we adding a new nested stage up fragment? Recall that we are performing a model to model transformation where the target model uses the **Artifacts** language to describe how to generate concrete artifacts on the filesystem. The additional fragment will be retained on the target model, to instruct the final artifacts generator how to set the contents of each created file.

We can introduce some **Variables** of the **Commons** language to parameterize the bean class definition.



Building on the knowledge that we have gathered we add another nested **Select** statement that maps source entity poperties to target Java filed declarations.



The above fragment introduces several new **Queries** language constructs, let's review them synthetically.

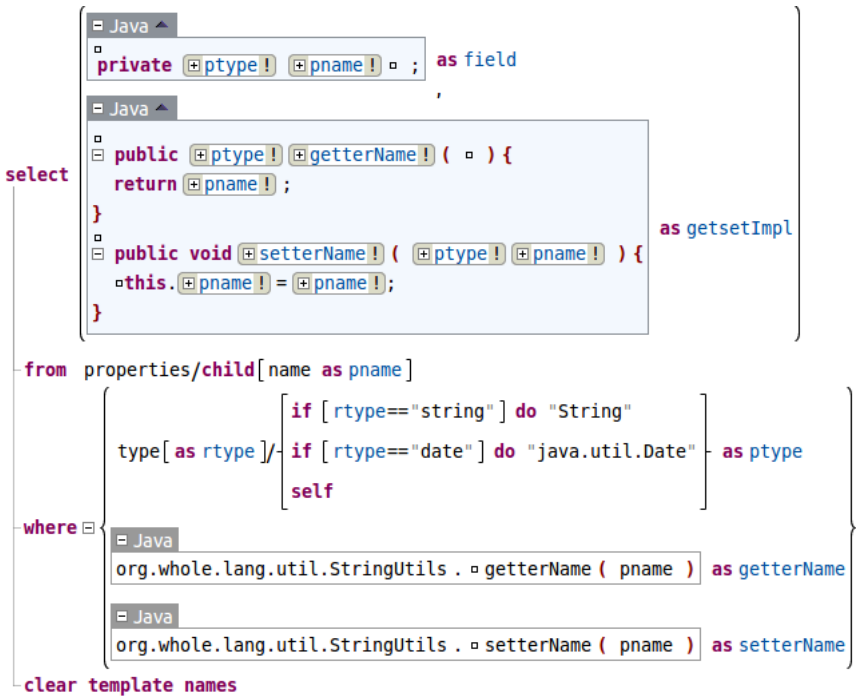
The **ChildStep** allows to iterate over the children of a collection, while the **Path** is used to compose several constucts. In the *from* clause a **Path** is used to compose a **FeatureStep** to a **Filter** , which in turn wraps a **ChildStep** . In this context the path allows to iterate over all the children of the `properties` feature, also binding every child name to the `pname` variable.

The **If** statement is straightforward: it evaluates the *do* clause only if its preadicate is satisfied. The **Choose** statement, that in this example contains the **If** statements followed by a **SelfStep** , chooses the first of the contained constructs that produces a value and consumes it.

To complete the generator we must add the statements to create a getter and a setter for each generated field. To achieve this result, we wrap the *select* clause with a **Tuple** that allows to generate several fragments at a time (in our example two), than we add another **StageUpFragment** containing the method implementations.

In this case `getsetImpl` is an **InlineVariable** (inline varibles are painted with a dashed border) because we want to unwrap the generated **MethodDeclarations** from the **BodyDeclarations** container before substituting them.

Eventually, we add the code needed to calculate the correct getter and setter names.



In this last case we use two **SameStageFragment** to invoke two helper

methods written in Java. The variable `pname` we defined in the *from* clause is referenced as a Java variable in the **MethodInvocations** .

Chapter 4

LWC11-Task-0.3

This page is part of the Whole Platform LWC11_Submission .

4.1 Task

0.3 Simple constraint checks such as name-uniqueness
For example, check the name uniqueness of the properties in the entities.

4.2 Screenshots

EntitiesActions.xwl

4.3 Overview

In a model driven approach a large part of structural constraints are enforced by the definitions at the meta level. More specifically, model instances are always conformant to their metamodels. For instance, the editors are not allowed to violate such constraints and the content assist is limited in accordance with them.

You can provide additional validation rules for your language. These rules behave as soft constraints, so you are allowed to violate them in model instances. The system can perform a validation check to inform you and to disable the execution of certain operations on an invalid model.

In the Whole Platform, operations are system wise and cross language by default. From within the Language Workbench the validation operation can be launched either from the **Operation** menu or the toolbar. The results are shown

both in the *Problems* view and in the tooltips inside the editors. In a headless execution context you can choose where to send the validation events.

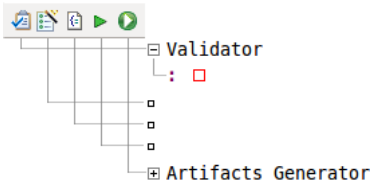
The **Actions** language can be used to define the behavior of the validation operation for your language.

To resolve this task, we propose a solution consisting of three validation rules:

- Property name uniqueness - The name of a property must be unique in the defining entity
- Property name conventions - The name of a property should begin with a lower case letter
- Entity name conventions - The name of an entity should begin with an upper case letter

4.4 Details

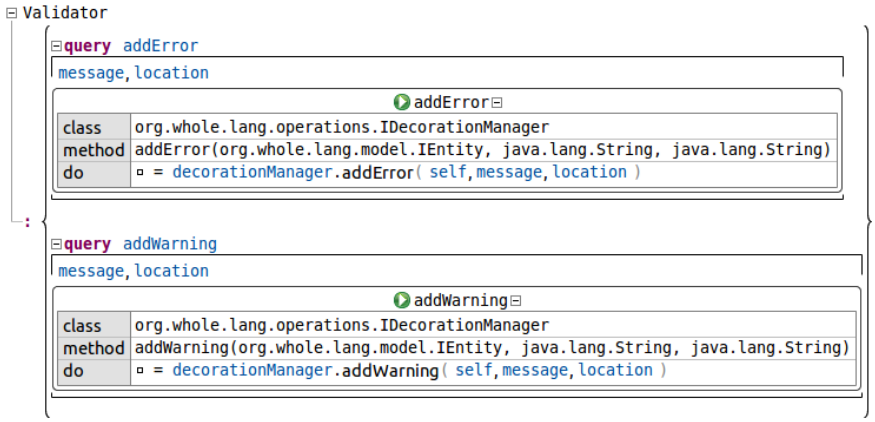
As we did for the artifacts generator action we start by creating a new **SimpleAction** for the validator operation inside the previously created model **EntitiesActions.xwl** .



The first things we have to define are the mechanism to add decorations to the to the *Problems* view. For this task we use two **QueryDeclarations** to help us add error or warning decorations to the `decorationManager` , a variable that is always in scope in the context of a validation operation.

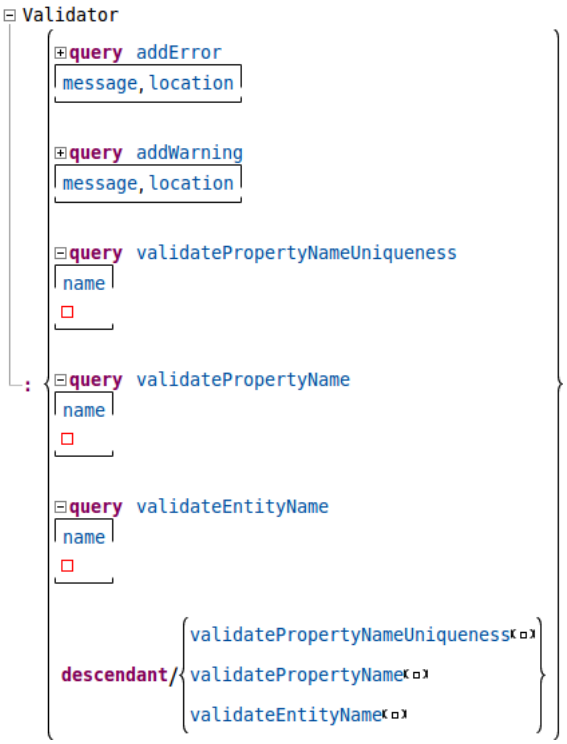
A **QueryDeclaration** is a callable unit of behavior with specific scope rules that can be invoked using the **Call** construct (both of them belong to the **Queries** language).

The **Call** and the **QueryDeclaration** operate on the same self. A **QueryDeclaration** can access all the variables in scope at call time except those listed in its **Names** collection. The expressions declared on a **Call** will be evaluated and bound to the **Names** of the called **QueryDeclaration** in the order they have been declared.



As shown in the above figure, both `addError` and `addWarning` serve as a domain level adapters to the low level API implemented by the `org.whole.lang.operations.IDecorationManager` interface.

We proceed by defining the structural elements of the validation: the **QueryDeclarations** that implement the aforementioned validation rules and the traversal strategy to enforce such validation rules.



Note that on each descendant will be invoked in sequence all the three query declarations `validatePropertyNameUniqueness` , `validatePropertyName` , and `validateEntityName` .

The first query delaration must enforce property name uniqueness. Its implementation verifies that a property does't have a preceding sibling having the same name, otherwise it will add an error describing the constraint violation.

```
query validatePropertyNameUniqueness
name
if [Entities][Property][name as name][some preceding-sibling satisfies [|name|==name]]

do addError( "The name of a property must be unique in the defining entity",
           | ancestor[Entity]/name |+"."+name )
```

The second query delaration must enforce property name validity. Its implementation verifies that a property name is not capitalized, otherwise it will add a warning describing the constraint violation.

```
query validatePropertyName
name
if [Entities][Property][name as name][isUpperCap]

do addWarning( "The name of a property should begin in lower case",
             | ancestor[Entity]/name |+"."+name )
```

The last query delaration, in a manner similar to the previous one, must enforce that entity names are capitalized.

```
query validateEntityName
name
if [Entities][Entity ][name as name][isLowerCap]

do addWarning( "The name of an entity should begin in upper case",name )
```

Chapter 5

LWC11-Task-0.4

This page is part of the Whole Platform LWC11_Submission .

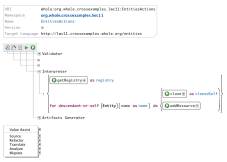
5.1 Task

0.4 Show how to break down a (large) model into several parts, while still cross-referencing between the parts

For example, put the Car and Person entities into different files, while still having the Person -> Car reference work.

5.2 Screenshots

EntitiesActions.xwl



5.3 Overview

The feature of *breaking down a model into several parts, while still cross-referencing between the parts* consists of two orthogonal concerns: structural and behavioral.

From a structural point of view, you should be able, at least, to edit and persist language fragments with the granularity you decide. The Whole Platform poses almost no restrictions on the ability of operating on language fragments. You can experiment by yourself this feature in a Whole editor simply by drag and dropping an arbitrary model entity either over the root entity or in the *Package Explorer* view.

From a behavioral point of view, you should be able to reconstruct, at least logically, a (complete) model starting from the separated fragments. This can be accomplished in different ways depending on the information available on each fragment. The Whole Platform supports the creation of resource registries including the ability to discover and load resources on demand.

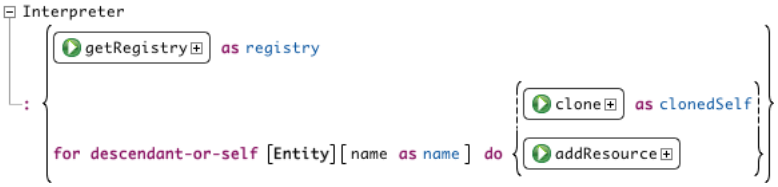
To resolve this task, using the proposed granularity of a single Entity (of the Entities language), no changes are required to the code developed so far. The Entities language does not include information to reference and discover separate modules (for instance the package and import declarations of Java). So, we require the user to interpret a fragment in order to add its entities to the *global* registry of defined entities.

The information collected on the Registry will be used by the runtime type system implemented in Task 1.2 .

5.4 Details

Open the **EntitiesActions.xwl** model developed so far and add a SimpleAction in correspondence with the interpreter tool.

The interpreter lazily creates a global registry identified by the URI of the Entities language and adds to it a resource for each entity found on the model.



Chapter 6

LWC11-Task-1.1

This page is part of the Whole Platform LWC11_Submission .

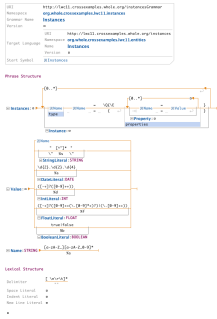
6.1 Task

1.1 Show the integration of several languages

Define a second language to define instances of the entities, including assignment of values to the properties. This should ideally really be a second language that integrates with the first one, not just "more syntax" in the same grammar. We want to showcase language modularity and composition here.

6.2 Screenshots

InstancesGrammar.xwl

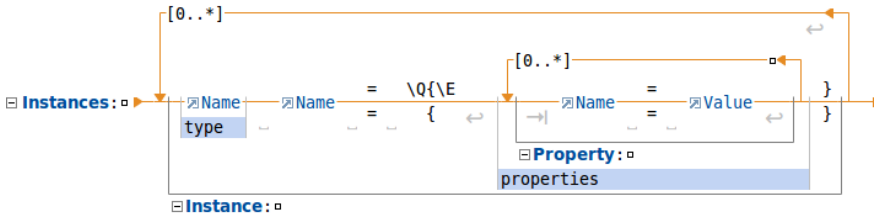


6.3 Details

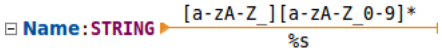
Basically, we proceed by creating an `InstancesGrammar.xwl` as we did in Task 0.1 for the *Entities Grammar* model. At first we customize the grammar global metadata.

URI	http://lwc11.crossexamples.whole.org/instancesGrammar
Namespace	org.whole.crossexamples.lwc11.instances
Grammar Name	Instances
Version	▫
Target Language	URI: http://lwc11.crossexamples.whole.org/instances Namespace: org.whole.crossexamples.lwc11.entities Name: Instances Version: ▫
Start Symbol	Instances

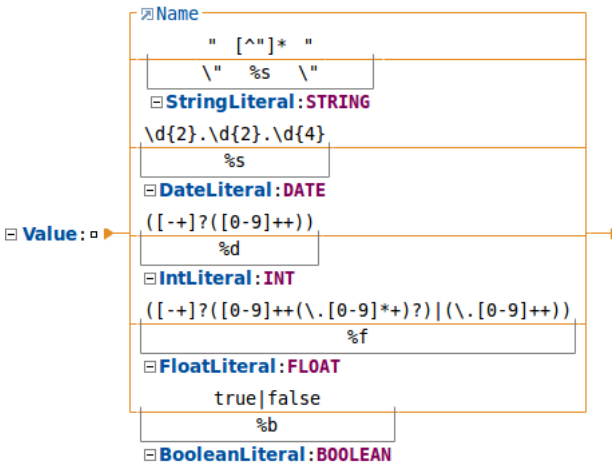
Similarly to what we have already done for the *Entities* language, we add the productions to define the overall language structure.



To complete the grammar we must define the **Name** and **Value** productions. We copy the definition of the **Name** production from the *Entities* grammar as is.



The **Value** production is defined using a **Choose** rule that contains all the allowed literal types.



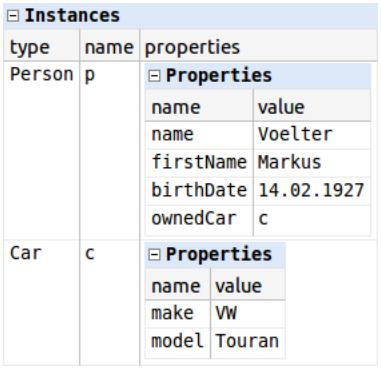
The Whole Platform has the ability to transform every parsed data literal into a value of the declared production data-type. Obviously, many data formats are supported but not all. In the *Instances* example, date types use a format that is not automatically understood by the framework, so we have to extend the language to support such a data-type by adding a customized **IDataTypeParser** .

For this purpose we simply create a class that extends the **GrammarsDefaultDataTypeParser** by adding the date format customizations needed. The class has the responsibilities of parsing literals and unparsing values, both achieved using a standard **DateFormatter** .

```
*/
public class InstancesDataTypeParser extends GrammarsDefaultDataTypeParser {
    private static DateFormat formatter = new SimpleDateFormat("dd.MM.yyyy");

    private InstancesDataTypeParser(GrammarBasedDataTypeParser dataTypeParser) {
        super(dataTypeParser);
        formatter.setTimeZone(TimeZone.getTimeZone("UTC"));
    }

    @Override
    public Date parseDate(EntityDescriptor<?> ed, String value) {
        synchronized (formatter) {
            try {
                return formatter.parse(value);
            } catch (ParseException e) {
                throw new IllegalArgumentException(WholeMessages.no_data_type);
            }
        }
    }

    @Override
    public String unparseDate(EntityDescriptor<?> ed, Date value) {
        synchronized (formatter) {
            return formatter.format(value);
        }
    }

    public static void install(final String languageURI) {
```

The class contains also an **install(...)** helper method meant to simplify its deployment. It is invoked inside the **LWC11Deployer.xwl** as described in the solution overview .

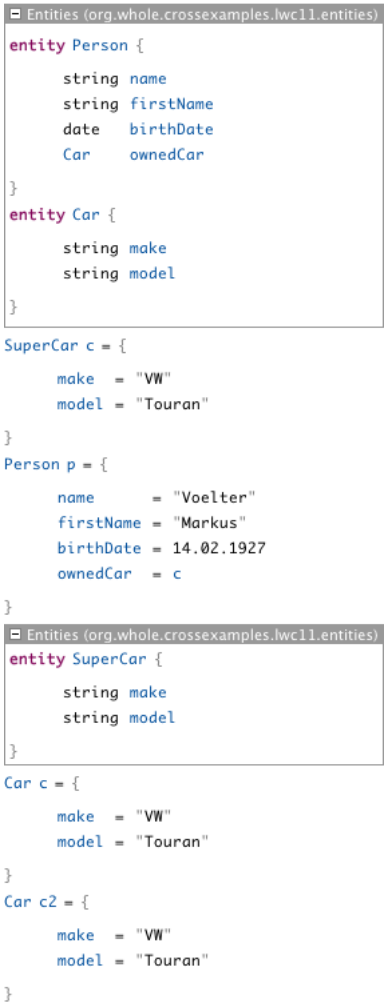
After deploying the *Instances* grammar it is possible to open the provided **InstancesExample.txt** example using the **Grammar Based Persistence** .



Instances

type	name	properties
Person	p	**Properties**
		name: Voelter
		firstName: Markus
		birthDate: 14.02.1927
		ownedCar: c
Car	c	**Properties**
		make: VW
		model: Touran

Now, we want to showcase language composition by allowing definition of entities together with instances in a single model. You can either define explicit composition rules or use the **SameStageFragment** construct of the *Commons* language as glue. The latter solution allows for ununticipated language composition and is available without any specific effort.



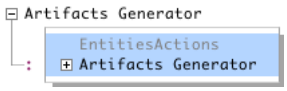
Warning

You have to **File > Save As...** the model using a generic persistence because the grammar based persistences are not composable without an additional effort.

The behavior we already defined for the *Entities* language continue to work as expected when applied to a model containing both entities and instances. In general to support ununticipated composition is sufficient to write path expressions and patterns that capture exactly what we want without assuming that the context (we traverse) contains only the constructs we defined.

Note

The actions defined for a given language are available only when the root of the model is an instance of the language. In order to extend the availability of such actions to a new language acting as a container you have to redefine the actions by calling the original behavior. In order to define an artifacts generator for instances that works on the entities defined inside of the instances model, you have to add the following action to the **InstancesActions.xwl** .



Chapter 7

LWC11-Task-1.2

This page is part of the Whole Platform LWC11_Submission .

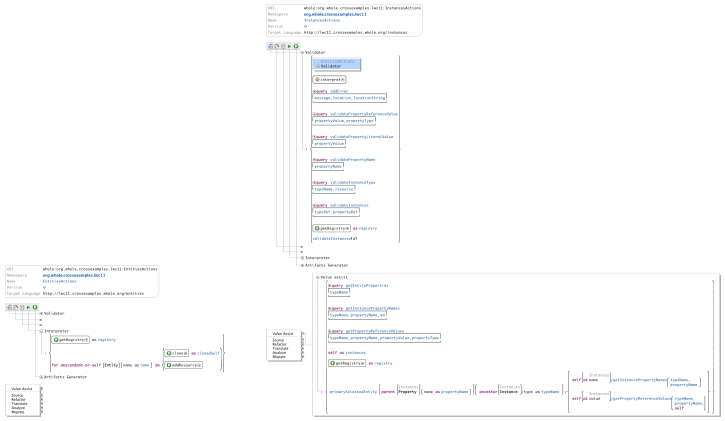
7.1 Task

1.2 Demonstrate how to implement runtime type systems

The initialization values in the instance language must be of the same type as the types of the properties.

7.2 Screenshots

EntitiesActions.xwl InstancesActions.xwl



7.3 Overview

To enforce type checking on Instances models we must leverage the information collected on the Entities Registry described in Task 0.4 . More specifically, we must verify that each Instance is valid with respect to the type it declares to define.

To resolve this task, we add a validation operation to the Instances language. The validation process consists of two complementary tasks: the first is to locate

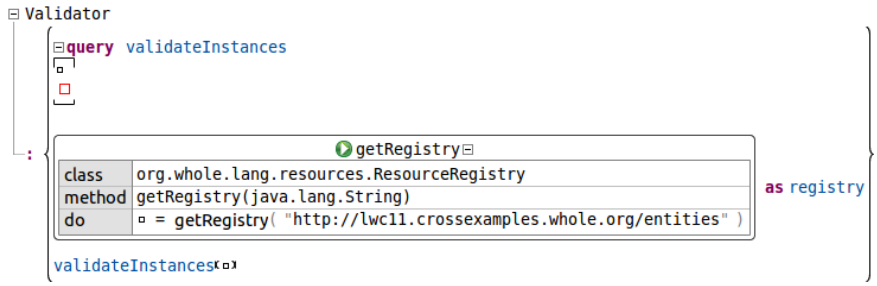
the right type declaration, while the second is to perform type checks.

The list of type checks that are performed on every instance follows:

- Its type declaration must exists in the Registry.
- Its properties must have been declared in its declared type.
- Every property value must be compatible with the declared data type.

7.4 Details

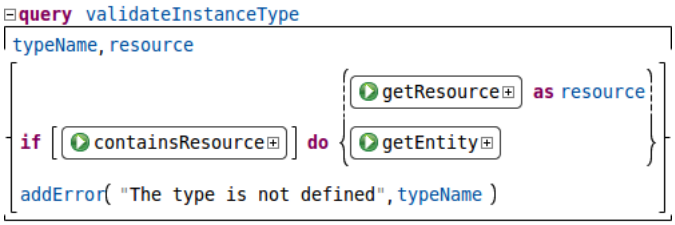
Open the **InstancesActions.xwl** model developed so far and add a Simple-Action in correspondence with the validator tool. Since the validation needs the Entities Registry in scope to extract type definitions, we obtain it using a workflows activity and then we proceed with the validation by calling the **validateInstances** query declaration.



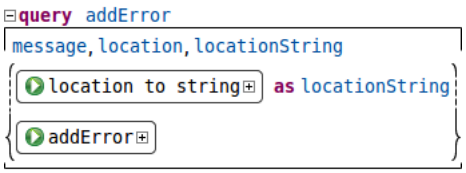
The **validateInstances** query declaration performs a top-down traversal of all the Instances having the declared type contained in the Entities Registry. On such Instances it iterates over the contained properties and enforces the type checks.

The first check verifies that the property has a name, if not it adds an error otherwise it continues validating the property name correctness. The second check verifies that the property has a value: if it has a **Name** value if validates the reference existence otherwise it verifies that the literal type conforms to the type declaration.

The **validateInstanceType** simply tries to extract the type declaration from the Entities Registry, adding an error on failures.



The **addError** adds error decorations to IDecorationManager in a similar way to what we have done in the Entities validator (see Task 0.3).



The **validatePropertyName** ensures that the property name exists in the declared type.

```
query validatePropertyName
propertyName
[ typeDef/properties/child[|name|==propertyName]
addError( "The property is not defined",propertyName )]
```

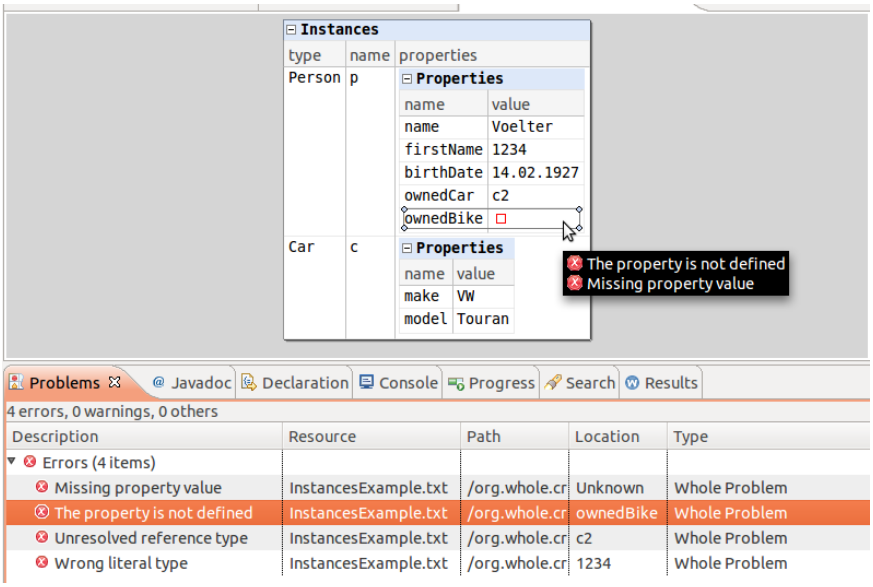
Reference constraints are verified by the **validatePropertyReferenceValue** query declaration by looking for an Instance declaration with a name that matches the property reference value being checked.

```
query validatePropertyReferenceValue
propertyValue,propertyType
[ root/child [name as propertyValue][ propertyDef/type as propertyType ][|type|==propertyType]
propertyValue/addError( "Unresolved reference type",propertyValue )]
```

Finally, the **validatePropertyLiteralValue** checks that the property literal value type conforms to the corresponding property data-type in the declared Entity type.

```
query validatePropertyLiteralValue
propertyValue
[
                    [|self|=="string"][propertyValue[Instances StringLiteral]]
                    [|self|=="date"][propertyValue[Instances DateLiteral]]
propertyDef/type    [|self|=="int"][propertyValue[Instances IntLiteral]]
                    [|self|=="float"][propertyValue[Instances FloatLiteral]]
                    [|self|=="boolean"][propertyValue[Instances BooleanLiteral]]

propertyValue/addError( "Wrong literal type",propertyValue )]
```

We can now open an Instances model and add some errors to test the validation operation.


Instances
type
name
properties
Person
p
Properties
name
value
name
Voelter
firstName
1234
birthDate
14.02.1927
ownedCar
c2
ownedBike
Car
c
Properties
name
value
make
VW
model
Touran
The property is not defined
Missing property value
Problems
Javadoc
Declaration
Console
Progress
Search
Results
4 errors, 0 warnings, 0 others
Description
Resource
Path
Location
Type
Errors (4 items)
Missing property value
InstancesExample.txt
/org.whole.cr
Unknown
Whole Problem
The property is not defined
InstancesExample.txt
/org.whole.cr
ownedBike
Whole Problem
Unresolved reference type
InstancesExample.txt
/org.whole.cr
c2
Whole Problem
Wrong literal type
InstancesExample.txt
/org.whole.cr
1234
Whole Problem

Chapter 8

LWC11-Task-1.3

This page is part of the Whole Platform LWC11_Submission .

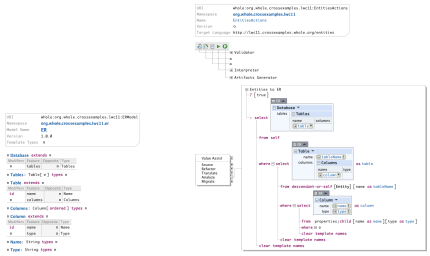
8.1 Task

1.3 Show how to do a model-to-model transformation

Define an ER meta model (Database, Table, Column) and transform the entity model into an instance of this ER meta model.

8.2 Screenshots

ERModel.xwl EntitiesActions.xwl



8.3 Overview

We have to create a DSL, called *ER* , to model a very simplified database schema. Then we have to write a mapping from the **Entities** to the **ER** DSLs.

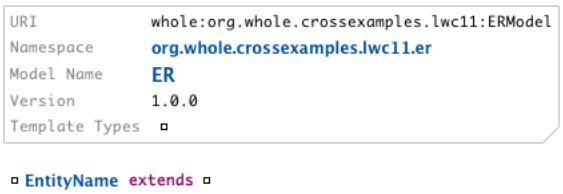
The Task does not require us to use a given concrete syntax for the **ER** persistence, so we are free to write directly the meta model using the **Models** DSL letting the Whole Platform provide both a suitable persistence and notation.

8.4 Details

8.4.1 Creating the ER meta model

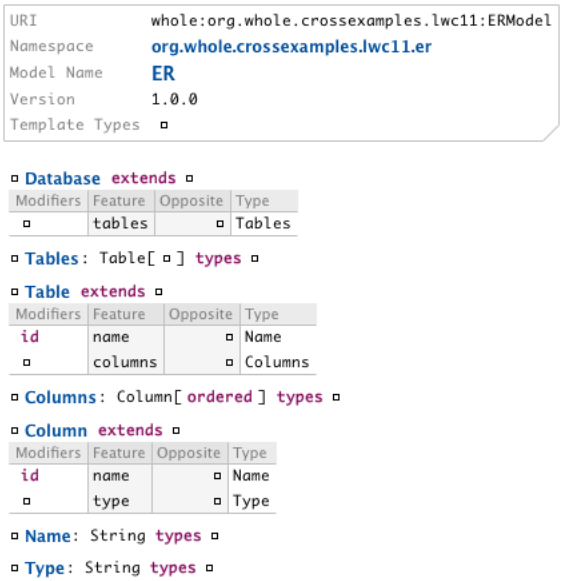
Using the Whole Model Wizard, create a new model called *ERModel* configured with the **Models** language and an *empty* template.

Fill in the meta data information as it is shown in the figure below. Note that the URI used follows a convention that permits the Whole Platform to locate the meta model and deploy it on demand.



For each concept of the domain, add a *SimpleEntity* with the same name and add the features you need to associate it to other concepts and to store basic information. For instance the *Table* entity needs at least a *name* and a *columns* features.

Then add two *CompositeEntity* to represent the collections of tables and columns. Eventually add two *DataEntity* to represent names and types. The resulting model should look like this:

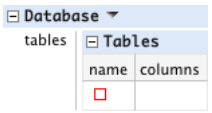


8.4.2 Playing with ER instances

Before proceeding with the transformation, deploy and instantiate the meta model to verify that the expressivity of the new language.

To deploy the meta model you simply have to click the Interpreter button from the Whole toolbar. After deploying a meta model you get a full working language integrated with the other languages and services of the Whole Platform. To create an instance of the *ER* language you have to create a Whole Model and select *ER* as Language. Observe that the procedure is exactly the same as you did to create the meta model itself.

Note that the first concrete entity defined in the meta model will be instantiated as the root entity of the new *empty* model.



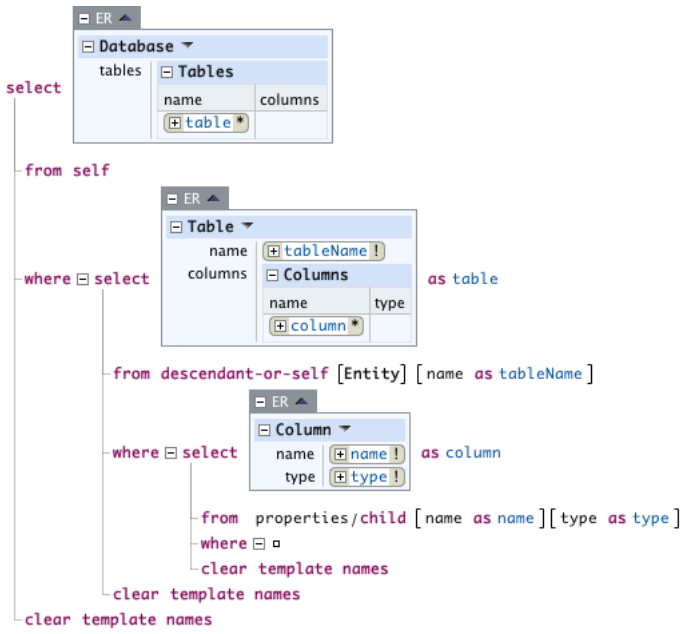
Modeling a domain is an iterative activity because you do not know exactly what you want to model from the beginning. With the Whole Platform you are able to deploy even incomplete meta models. The editor will show a placeholder for the underspecified entities and features. You can also redeploy a model everytime you need to.

8.4.3 Writing the transformation

Open the *EntitiesActions* model and create a *GuardedAction* into the context menu "Analyze". Name the menu item: "Entities to XML".

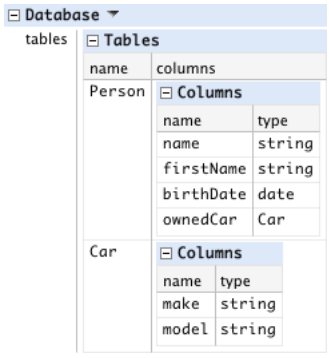
There is almost a one-to-one correspondence between the two meta models that permit to write a very straightforward mapping.

The skeleton of the transformation can be written using three nested *Select* constructs of the **Queries** language. The outer one maps an *Entities* to a *Database* ; the nested one maps an *Entity* to a *Table* ; and the innermost maps a *Property* to a *Column* .



Note that the entities named *Type* in the two languages have a different structure whereas the mapping seams to be one-to-one. This is possible because the Platform automatically maps the two concrete entities of the abstract *Type* of **Entities** to the data entity *Type* of **ER** using their textual representation.

Now we are ready to redeploy the *EntitiesActions* and to open an example of **Entities** . In the context munu choose the newly added action to get the following output in the *Results* view.



Chapter 9

LWC11-Task-1.4

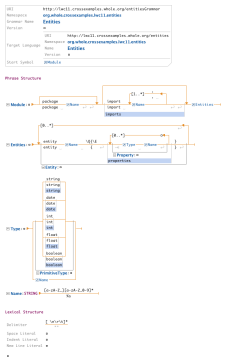
This page is part of the Whole Platform LWC11_Submission .

9.1 Task

1.4 Some kind of visibility/namespaces/scoping for references
Integrate namespaces/packages into the entity DSL.

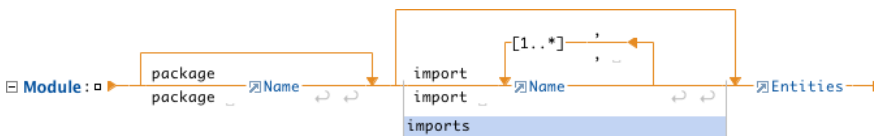
9.2 Screenshots

EntitiesGrammar.xwl



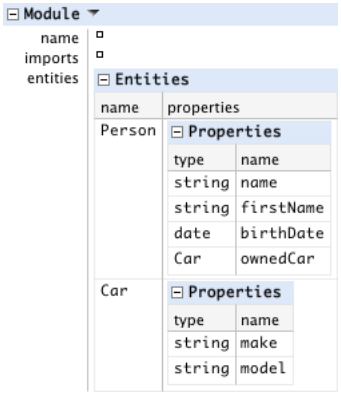
9.3 Details

Open the *EntitiesGrammar* model and add the following production at the beginning of the *Phrase Structure* :



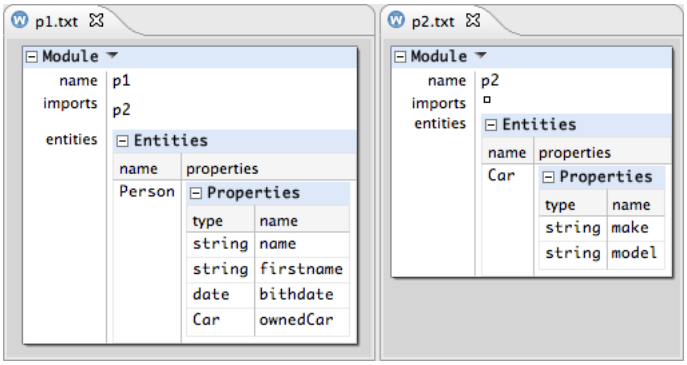
The new production consists of a concatenation of an optional package declaration and an optional imports declaration followed by the *Entities* non terminal. This way, the new production is backward compatible with the previous grammar.

Rename the *Start Symbol* of the grammar to *Module* and redeploy the grammar using the interpreter action. Close and reopen the *Entities* example to verify the backward compatibility of the extended grammar and to discover the differences in the resulting model.



As you can see the root entity is changed to an instance of *Module* with two placeholders for the *name* and *imports* features. The model fragment under the *entities* feature matches exactly the model created with the old grammar.

Now we can take advantage of the new modular syntax and divide the example in two files with the one defining the Person entity importing the other.



Chapter 10

LWC11-Task-1.5

This page is part of the Whole Platform LWC11_Submission .

10.1 Task

1.5 Integrating manually written code (again in Java, C# or C++)

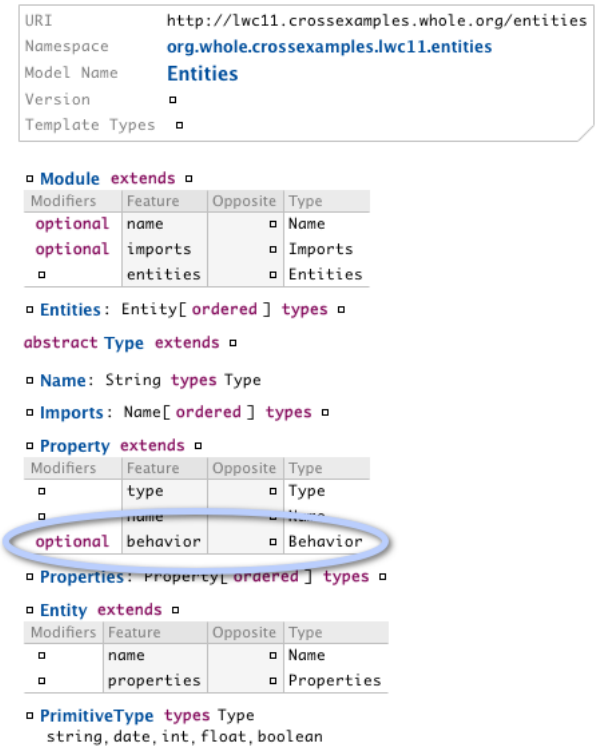
Integrate derived attributes to entities.

10.2 Overview

In the Task 0.1 we have created the **Entities** language by defining a grammar for it. Every time we deploy the grammar, the Whole Platform derives a metamodel and also deploys it, in most cases this is the desirable behavior. Now we want to extend the language by adding the ability to define derived properties using manually written code. To achieve this goal, it is sufficient to work at the metamodel level without the need to change the grammar.

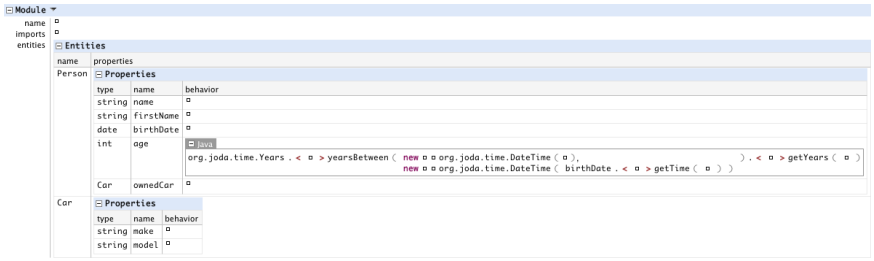
10.3 Details

Open the **EntitiesGrammar.xwl** and launch the *Generate Artifacts* toolbar action. Open the newly created **EntitiesModel.java** file using **Open With > Java Builder Persistence** context menu action. Append an optional feature named `behavior` to the **Property** simple entity, using a type named **Behavior** (it will be implicitly defined as abstract).



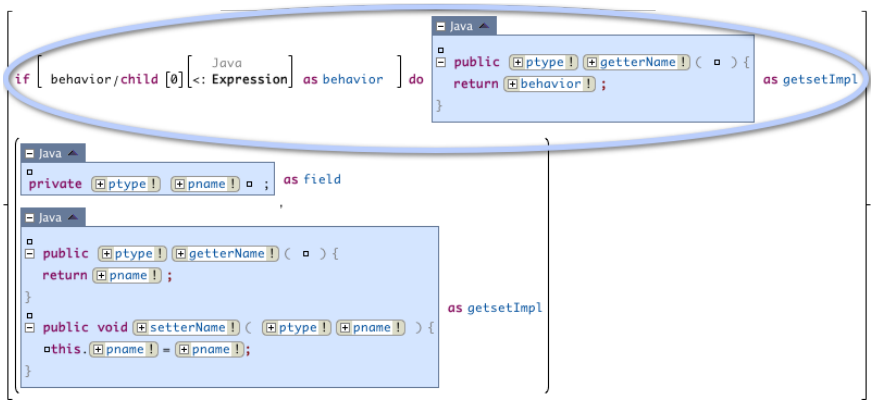
Deploy the **EntitiesModel.java** using the *Interpret model* toolbar action. This way we have replaced the language deployed by the grammar with a backward compatible improved version. Afterwards, we can reopen the **EntitiesExample.txt** and add an *age* property. For the derived behavior, copy the following snippet and use the **Paste As** action as we did in the Task 0.2 , having care to choose **Same** stage level.

```
org.joda.time.Years.yearsBetween(
    new org.joda.time.DateTime(),
    new org.joda.time.DateTime(birthDate.getTime())).getYears()
```



Use the **File > Save As...** menu action to save the modified model using the file name **EntitiesExample2.xwl** and the **XML (Whole Template Builder)** persistence.

Note that all of the behavior defined so far continues to work properly as before the metamodel change. In order to take into account the derived properties open the **EntitiesActions.xwl** and, in the **ArtifactGenerator** action, wrap the innermost select clause with a **Choose** construct. Finally, insert a first case that only produces a derived getter when the behavior is defined.



Now we can redeploy the actions and apply the artifacts generator to the example.

Chapter 11

LWC11-Task-1.6

This page is part of the Whole Platform LWC11_Submission .

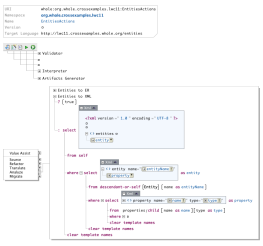
11.1 Task

1.6 Multiple generators

Generate some kind of XML structure from the entity model.

11.2 Screenshots

EntitiesActions.xwl

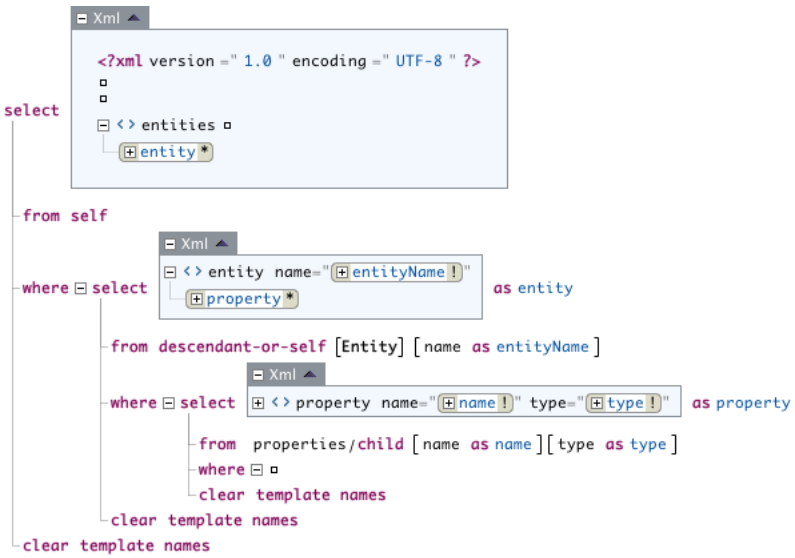


11.3 Details

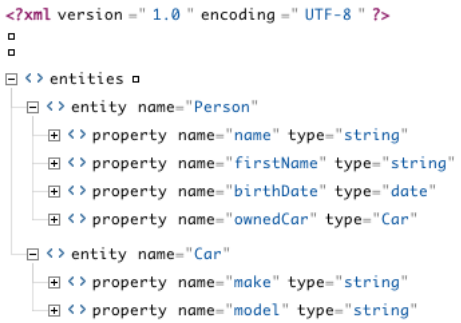
To generate an XML structure from an **Entities** model you can reuse the transformation described in Task 1.3 .

Open the **EntitiesActions** model and drag and drop a copy of the menu action named "Entities to ER". Rename the new menu item to "Entities to XML".

It is sufficient to replace the three templates of **ER** with **XML** based ones. We can map *Entities* to an XML *Document* with a root *Element* having the tag named "entities". Then we can map *Entity* to an "entity" *Element* having a *name* attribute. Finally we can map *Property* with a "property" *Element* having a *name* and *type* attributes. The resulting model should look like this:



Now we are ready to redeploy the *EntitiesActions* and to open an example of **Entities** . In the context menu choose the newly added action to get the following output in the *Results* view.



Chapter 12

LWC11-Task-2.1

This page is part of the Whole Platform LWC11_Submission .

12.1 Task

2.1 How to evolve the DSL without breaking existing models

12.2 Overview

A classic and widely accepted approach to software evolution prescribes backward compatibility and API stability. We think that this approach is too much conservative and fails to take advantage of the model driven technologies.

We are committed to support evolution of (meta)models and model instances including model transformations. To achieve this goal we follow two complementary strategies. We outline, at first, a short term solution using non breaking design and, secondly, a long term solution using model versioning and migration.

Just take into consideration that during the development of the solution we have already seen examples of non breaking design:

- **Grammars** additions don't break backward compatibility of both the parser/unparser and the derived metamodel
- **Models** additions don't break backward compatibility with existing instances
- **Queries** transformations tolerate both the preceding **Grammars** and **Models** evolutions

12.2.1 Non breaking design

The ease of evolution of a DSL can be much improved by having a solid knowledge of the non breaking changes supported by the Language Workbench in use. In a model focused language workbench like the Whole Platform, the separation of the metamodel from its persistences and notations gives us three degrees of freedom to address software evolution.

Metamodels

The non breaking changes of a metamodel are the following:

- addition of entities
- appending of features
- changes to the hierarchy of abstract types (as long as they preserve the feature list on the concrete entities)
- replacing a feature type with a more abstract one

The set of the perceived non breaking changes can be altered by notations and even more by persistences.

Notations

Notations can always be introduced and evolved without breaking existing models and transformations. The opposite is also true for the *Generic notations* that are, by definition, language neutral, even with respect to the metamodel evolution. A non breaking change to a metamodel makes the specific notations less specific and more projective (accidentally). Less specific because additional entities are shown using a generic notation as a fallback. More projective because additional features could be hidden by the specific notation.

Persistences

Persistences can always be introduced and evolved without breaking existing models and transformations. A persistence can be designed to be more or less tolerant to non breaking changes of the metamodel. For instance the **Java Builder Persistence** and the **XML Builder Persistence** by default are tolerant to additions and to feature name changes. The **XSI Persistence** instead is by default tolerant to reordering of features and to XML fragments not conforming to the declared XSD. Nevertheless, the bundled persistences can be configured to be tolerant to a different set of changes.

12.2.2 Model Versioning and Migration

Many languages bundled with the Whole Platform support the specification of an URI to uniquely identify a specific instance, for example **Models** , **Grammars** , **Actions** , and **XSD** . Whenever you want to introduce breaking changes to a model you should also change its URI. This way it is possible to deploy multiple versions of a metamodel, of a grammar and so on. Consequently, it is also possible to define a model to model transformation to allow automatic migration of instances. Given that the transformations are also modeled, it is possible to write specific model to model transformations to migrate them.

In the Task 3.1 we will define two transformations to migrate both **Entities** to **Models** and their respective instances. You can regard the two transformations as examples of model migration.

Chapter 13

LWC11-Task-2.2

This page is part of the Whole Platform LWC11_Submission .

13.1 Task

2.2 How to work with the models efficiently in the team

13.2 Overview

The Whole Platform language workbench is an Eclipse based product. All of the model artifacts written with the Whole Platform are stored by default using a generic XML based persistence. This implies that the facilities provided by the Eclipse platform to support project management, local resource versioning and team collaboration are also available to the Whole Platform users.

It is interesting to observe that the development process also requires agile means for exchanging snippets. Our tool supports exchanging of arbitrary model fragments via clipboard or drag-and-drop through popular collaboration tools such as Skype, DropBox, and Pastebin.

Another important feature of the Whole Platform is that it is easy to realize a customized language workbench product that bundles specialized DSLs (for defining metamodels, rules, mappings. . .) each providing multiple notations targeted to different team roles. This way you can tailor the Whole Platform to your specific development needs or to those of your customers.

Chapter 14

LWC11-Task-2.3

This page is part of the Whole Platform LWC11_Submission .

14.1 Task

2.3 Demonstrate Scalability of the tools

14.2 Overview

The Whole Platform is an open source mature language workbench, already employed in more than 40 commercial products developed since the beginning of 2005. It has been successfully deployed worldwide mainly for customers operating in the financial sector, specifically in the fields of electronic money and payment systems.

The scalability of the Whole Platform is proved by the fact that it is being employed in the definition of a large number, both in size and in quantity, of data definition models (metamodels, grammars, xsd), and of transformation models.

For instance, we have approximatively deployed:

- Several complex data formats (20 SEPA, 900 FIN, 300 RNI)
- 200 configurations of Software Product Lines
- 1000 workflows
- 300 test cases

The memory footprint is improved by the ability of deploying dynamic models and transformations at runtime without generating code. We have products that work with a large number of domain languages that would require a space in the order of gigabytes if generated.

The memory footprint is furthermore improved by the availability of the streaming APIs at the framework level. Several persistence providers, including XML Schema, XML, and grammar based, fully support both read and write operations in streaming. Model streaming permits to apply transformations to very large instances, that otherwise would be impossible to process in memory.

The performance of the Whole Platform is optimized even in dynamic deployment scenarios using a just-in-time compiler for the transformation languages. For instance, the performance of the *Sepa Credit Transfer* product is of 44000 transactions per hour server including for each transaction the time for: parsing, validation, enrichment, translation, assembly, bulking, tracing, routing.

Chapter 15

LWC11-Task-3.1

This page is part of the Whole Platform LWC11_Submission .

15.1 Task

3.1 Integration with the platform native modeling language

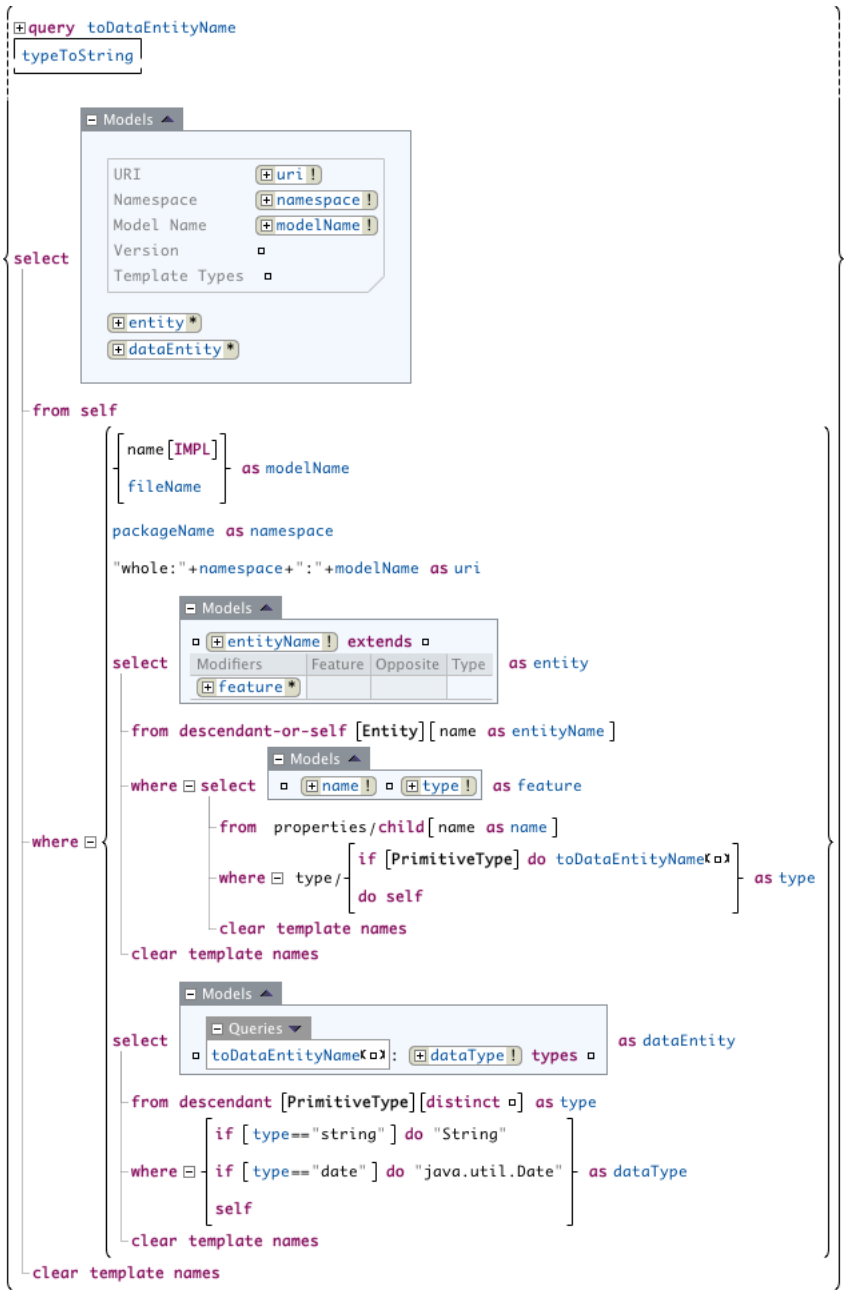
Integrate the **Entities** and **Instances** languages with the native modeling languages of the language workbench. We want to show how to expose the modeling services to other modeling languages.

15.2 Overview

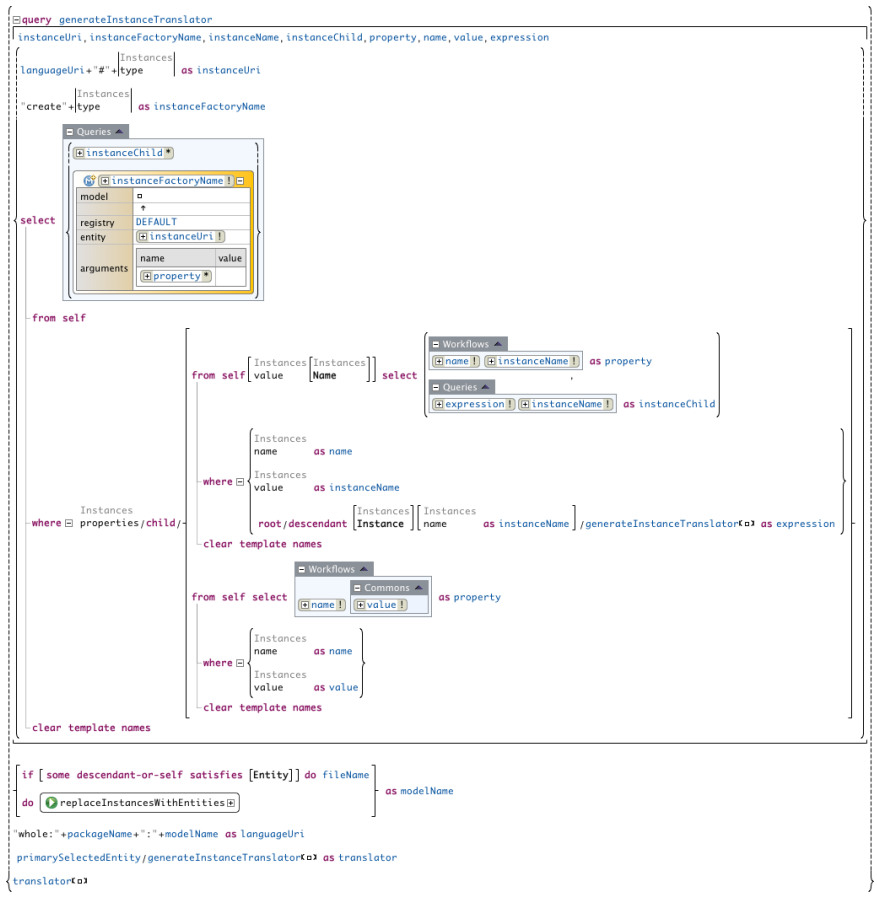
For a language workbench is reasonable to have a main modeling language for defining the structure of a language. This way all the modeling services can be implemented once and are available to other modeling languages via translational semantics.

Let us describe the integration of **XML Schema Definitions** and **XML Schema Instances** into the Whole Platform to outline the kind of integration we expect to achieve for **Entities** and **Instances** . We have defined an **XSD** to **Models** translator, and a bidirectional translator between **XSI** and **Models** instances that uses the knowledge derived by the former translator. Afterwards, we have defined the interpretation of an **XSD** as the application of the translator to **Models** followed by the deployment of its output. Also, we have defined a persistence for **XSI** that applies the bidirectional translator after having located and deployed also the schemas where needed. This way, whenever an XML Schema Instance is loaded you are able to edit the model at a domain specific level, change the persistence and the notation, and so on.

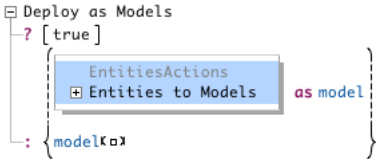
In order to obtain a similar integration for **Entities** and **Instances** we have to define an **Entities** to **Models** translator and a generator from **Instances** to the corresponding models instance. The **Entities** language can be regarded as a small subset of the **Models** language so the translation is straightforward. Note that each primitive type requires a separate data entity in **Models** .



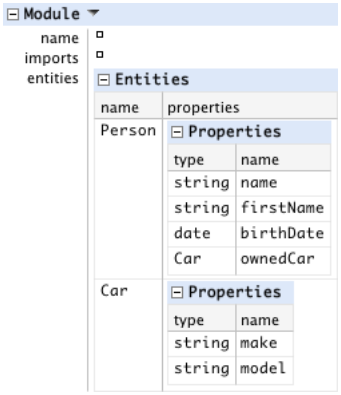
The translation of the instances is almost a one to one mapping between each instance and the code to instantiate the corresponding entity.



To avoid changes to the actions already defined for **Entities** and **Instances** we decided to add three new actions: *Entities to Models* , *Instances to Models instance* and *Deploy as Models* . The first two contain the behavior just defined while the latter is defined as follows:



Now we can load, for instance, the **EntitiesExample.txt** and deploy the corresponding model by calling *Deploy as Models* . Here we show also the derived model as is produced by calling *Entities to Models* .



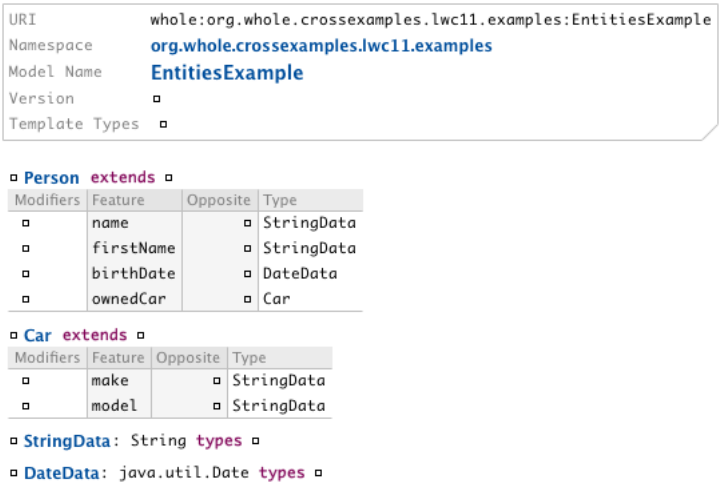
Module
name
imports
entities
Entities

name	properties
Person	Properties

type	name
string	name
string	firstName
date	birthDate
Car	ownedCar

Car	Properties

type	name
string	make
string	model



URI whole:org.whole.crossexamples.lwc11.examples:EntitiesExample
Namespace org.whole.crossexamples.lwc11.examples
Model Name EntitiesExample
Version
Template Types

Person extends

Modifiers	Feature	Opposite	Type
	name		StringData
	firstName		StringData
	birthDate		DateData
	ownedCar		Car

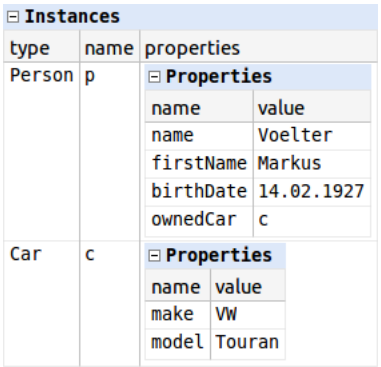
Car extends

Modifiers	Feature	Opposite	Type
	make		StringData
	model		StringData

StringData: String types

DateData: java.util.Date types

Afterwards, we can load the **InstancesExample.txt** and translate it into the corresponding model instance.



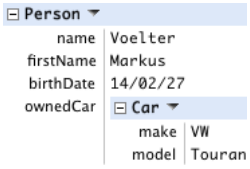
Instances

type	name	properties
Person	p	Properties

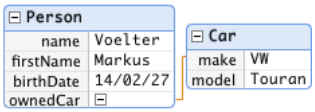
name	value
name	Voelter
firstName	Markus
birthDate	14.02.1927
ownedCar	c

Car	c	Properties

name	value
make	VW
model	Touran



By switching to a *Tree table* notation is even more evident the specificity of the derived model.



Observe that many constraints that before were checked by the validator now are structurally enforced. Furthermore the specific model is more suitable for applying model transformations.

Chapter 16

LWC11-Task-3.2

This page is part of the Whole Platform LWC11_Submission .

16.1 Task

3.2 Testing

Show how to test the grammars and the transformations.

16.2 Overview

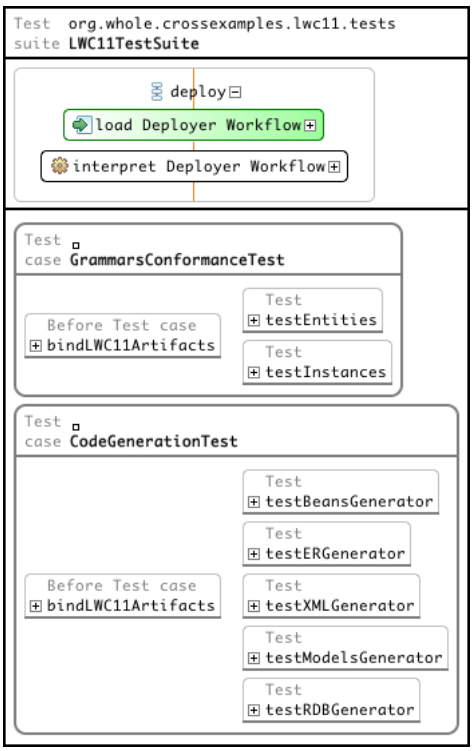
The Whole Platform includes a **Tests** DSL focused on writing and executing unit tests. The **Tests** language is very close to the well-known **JUnit** library; in fact, we provide a translator to Java that uses JUnit 4 in addition to a standalone interpreter.

Furthermore, the DSL has a *learning* mode (see menu actions) to auto-complete tests based on the actual behavior. To cope with differences between repeated executions of a test, it is possible to define filtering rules that remove irrelevant differences. The *learning* mode action features automatic definition of filtering rules!

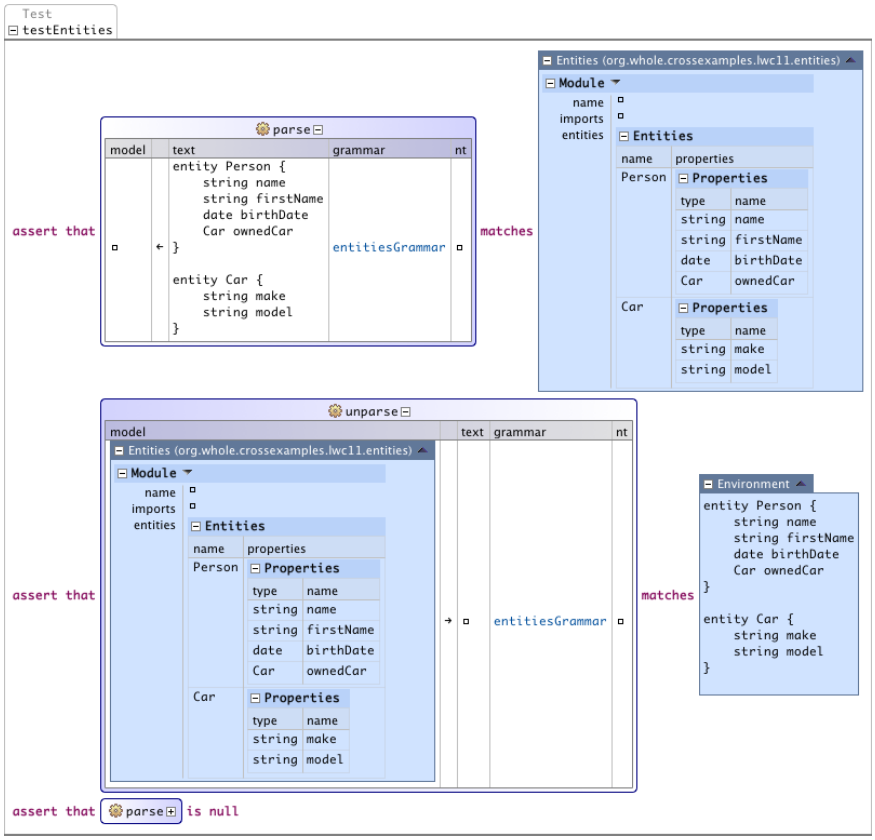
An integrated testing DSL grants more intentionality due to the use of almost plain English assertions on models shown with their own domain specific notations.

16.3 Details

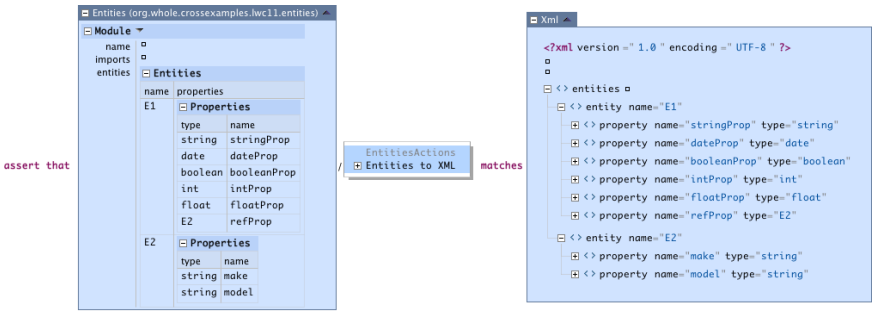
Create a **LWC11Tests.xwl** model with a test suite having two test cases: one for testing grammars and the other for testing the model transformations. Don't forget to define the *deployer* feature of the test suite using a **Workflows** that loads and deploys the **LWC11Deployer.xwl** .



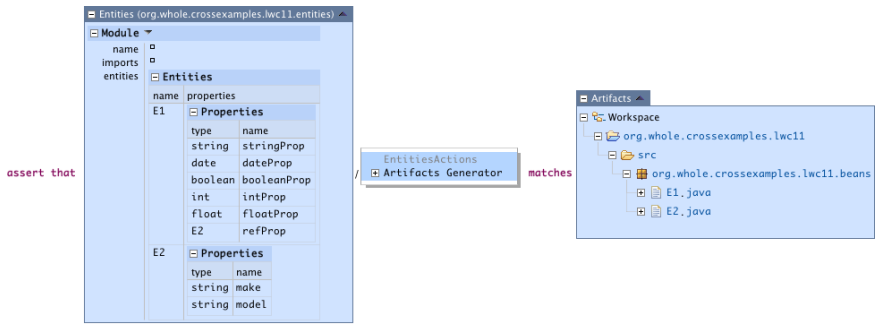
To test a grammar you have to test both the parser and the unparser with assertions covering each production and the more relevant compositions of them. Here, for simplicity, we define just two assertions for each grammar that demonstrate only that the bundled examples are working. Use the **Parse** and **Unparse** activities of the **Workflows** DSL to invoke the relative Grammars' operations.



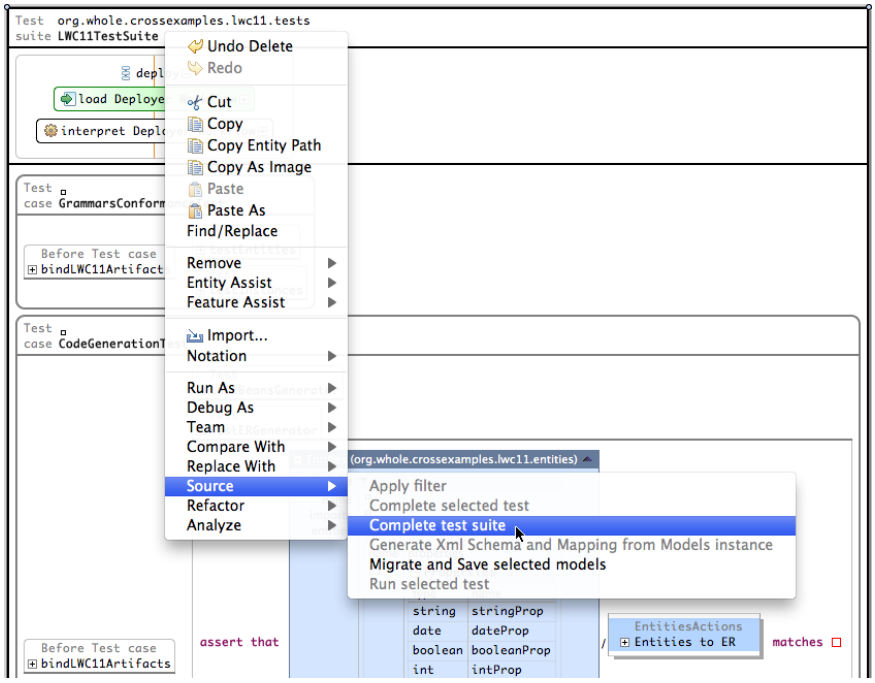
Use an **ActionCall** construct to test the behavior defined in an action of the **EntitiesActions.xwl** .



Note that a programmatic call to an artifacts generator action produces only an artifacts model, while invoking it interactively also produces the expected side effects on the file system. This way we are able to test even actions producing side effects.



Do remember that you are not forced to write by hand the models in the *matches* clauses: you can leave them empty and call the *Complete test suite* action.



Chapter 17

LWC11-Task-3.3

This page is part of the Whole Platform LWC11_Submission .

17.1 Task

3.3 Debugging

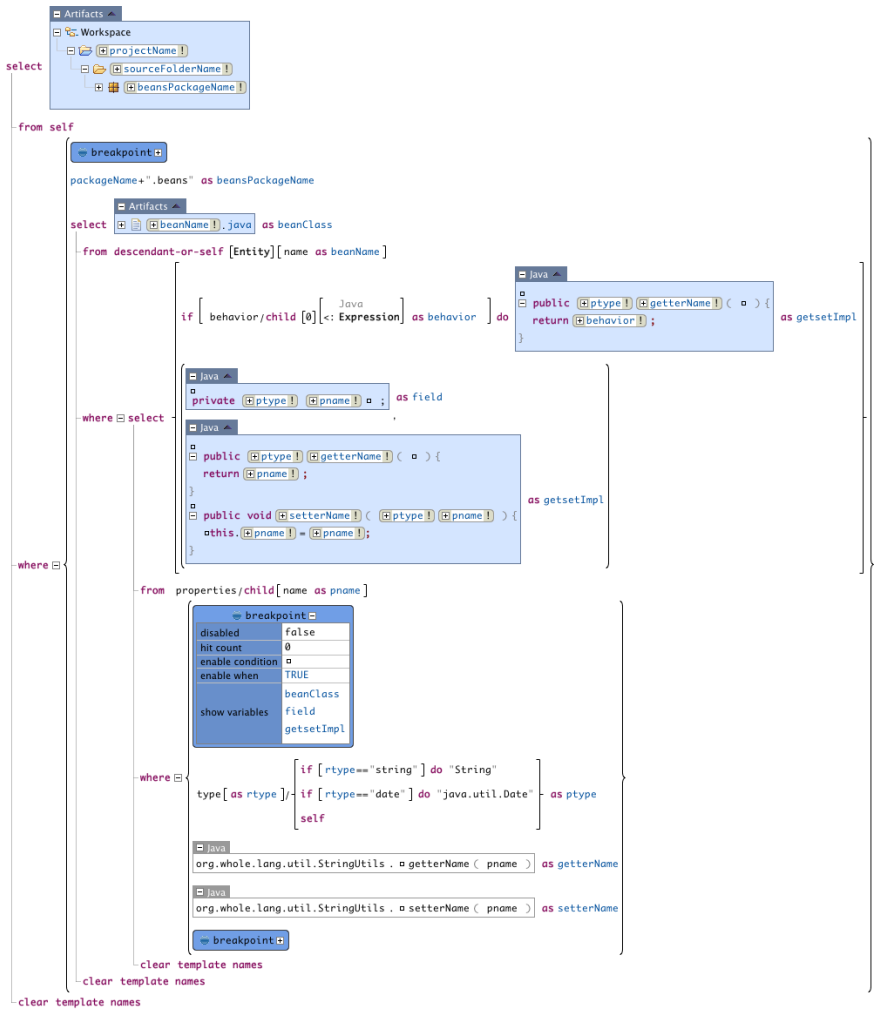
Show how to debug a model transformation.

17.2 Overview

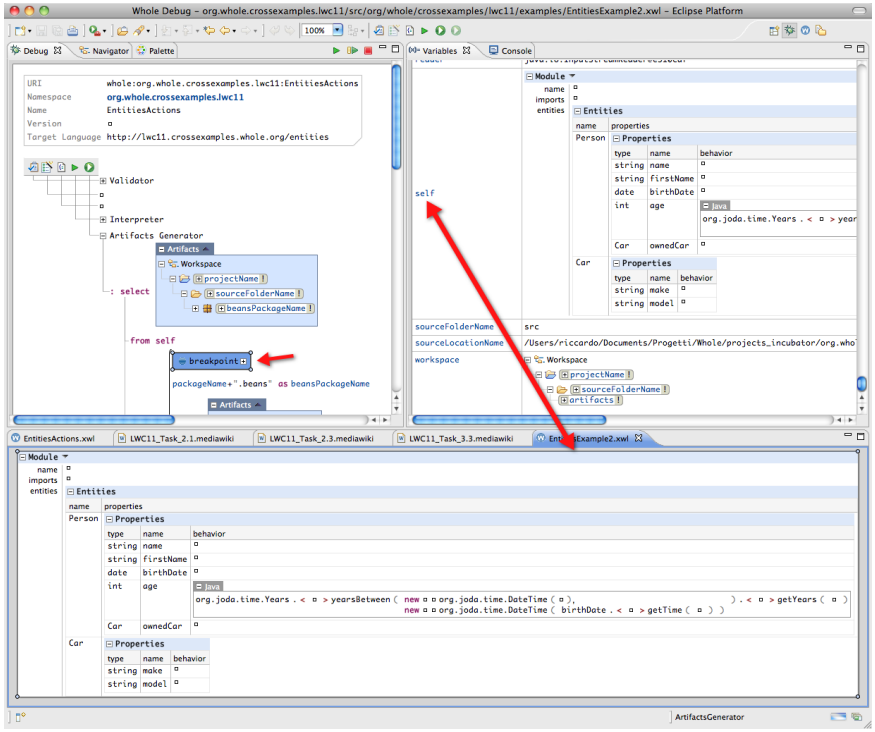
The development of complex model transformations is greatly simplified by the availability of a debugger. The Whole Platform provides a domain level debugger that is able to use the domain specific notations to show both the variables and the behavior being executed.

17.3 Details

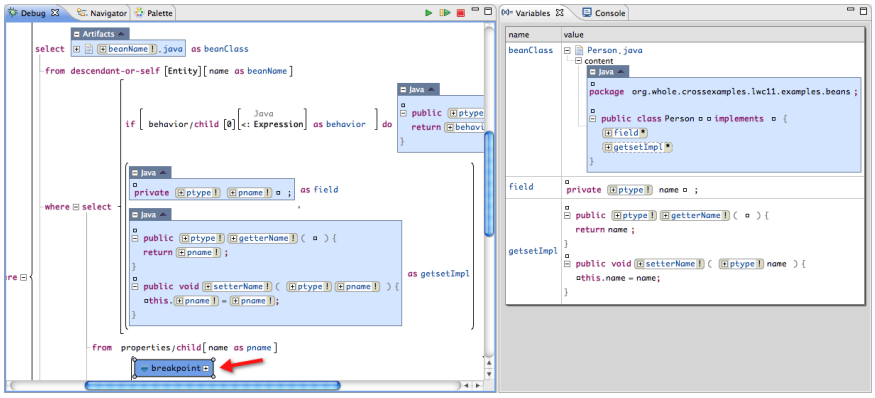
Open the **EntitiesActions.xwl** and add three **Breakpoint** constructs of the **Workflows** DSL as shown in the following figure. In the second and the third breakpoints add a list of variables to reduce the number of variables that will be shown in the **Variables** debug view. Finally, redeploy the actions using the *Interpret model* toolbar action.



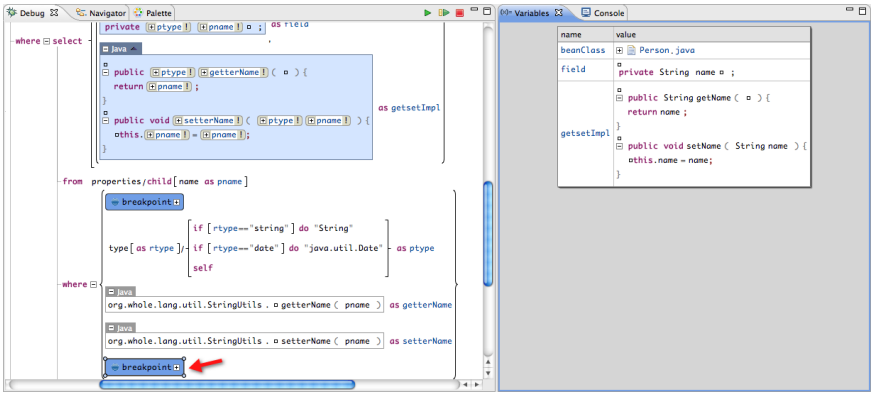
Open the **EntitiesExample2.xwl** and launch the *Generate Artifacts* toolbar action. The Language Workbench will perform a perspective switch automatically, simply put in background the *ArtifactGenerator* job dialog. The **Debug** view shows the executing behavior highlighting the breakpoint on which the debugger suspended. The **Variables** view shows all of the variables in scope, since the first breakpoint doesn't provide a variables list. Note that in the original editor the current *self* entity is highlighted.



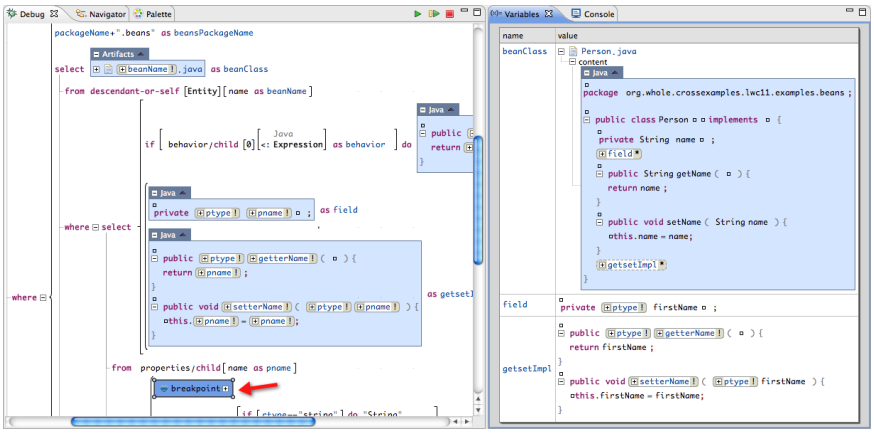
Pressing the **Resume** button in the **Debug** view the execution resumes up to the second breakpoint. As expected, in this case only a subset of the variables in scope is shown in the **Variables** . Note that the shown variables are templates that have been partially filled in.



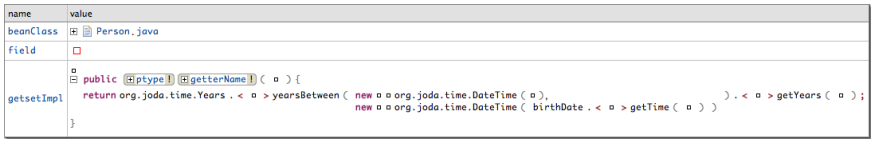
Press the **Resume** button again to reach the last breakpoint at the end of the *where* clause. Note that, at this point, the *field* and the *getsetImpl* variables have been completely filled in.



Pressing the **Resume** button again the execution suspends at the beginning of a new iteration of the *where* clause. Note that the results of the previous iteration are already visible in the *beanClass* variable.



Note that on the property with the derived behavior the *field* variable is not in scope, as shown in the **Variables** view.



Any time, you can either stop the debugger by pressing the **Terminate** button or resume execution with all breakpoints disabled by pressing the **Run** button.