TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software- and Multimedia-Technology, Software Technology Group

DRESDEN OCL2 FOR ECLIPSE

MANUAL FOR INSTALLATION, USE AND DEVELOPMENT

Claas Wilke and Michael Thiele

Last update on February 25, 2010

Dresden OCL2 for Eclipse has been developed at the Technische Universität Dresden, Faculty of Computer Science, Software Technology Group. Dresden OCL2 for Eclipse and this manual are available a the Dresden OCL2 for Eclipse project website (`http://dresden-ocl.sourceforge.net/`).

CONTACT

Technische Universität Dresden
Fakultät Informatik
Lehrstuhl Softwaretechnik
Prof. Dr. Uwe Aßmann
Nöthnitzer Str. 46
D-01187 Dresden

LICENSE

We are always looking forward to find new projects that use Dresden OCL2 for Eclipse or at least parts of the toolkit. Thus, please inform us, if you use Dresden OCL2 for Eclipse in your project.

Dresden OCL2 for Eclipse

This Manual

ABSTRACT

This document contains the documentation of Dresden OCL2 for Eclipse. In the first Chapter the general use of Dresden OCL2 for Eclipse and its installation is explained. Afterwards, use cases like OCL interpretation and code generation are presented.

Please be aware, that Dresden OCL2 for Eclipse is a project at the Technische Universität Dresden, Software Technology Group. Parts of the project have been realized and developed during student theses and have been developed as prototypes only. Thus, the toolkit is far from being complete. To report bugs and errors or request additional features or answers to specific questions visit our project site at *Sourceforge* [URL10e] or visit our toolkit website [URL10h].

The procedure described in this manual has been realized and tested with *Eclipse 3.5* [URL10c]. We recommend to use the *Eclipse Modeling Tools Edition* which contains all required plug-ins to run Dresden OCL2 for Eclipse[1]. Otherwise you need to install at least the plug-ins enlisted in Table 1.

[1] Apart from the AJDT that are required by some example plug-ins.

CONTENTS

I USING DRESDEN OCL2 FOR ECLIPSE

1 GETTING STARTED WITH DRESDEN OCL2 FOR ECLIPSE

Chapter written by Claas Wilke

This Chapter generally introduces into *Dresden OCL2 for Eclipse*. Dresden OCL2 for Eclipse is the last version of the *Dresden OCL Toolkit* and is based on a *Pivot Model*. The pivot model was developed by Matthias Bräuer and is described in his Großer Beleg (Minor Thesis) [Brä07]. Further information about the toolkit is available at the website of the Dresden OCL Toolkit [URL10h]. This Chapter starts with the installation of the required *Eclipse* plug-ins to run Dresden OCL2 for Eclipse. Afterwards, it describes how to load a domain-specific model, an instance of such a model, and OCL constraints defined on such a model.

1.1 HOW TO INSTALL DRESDEN OCL2 FOR ECLIPSE

Four different possibilities exist to install Dresden OCL2 for Eclipse.

1. You may install the plug-ins using the update site available at [URL10g],
2. You may install the plug-ins using the binary distribution available at the SourceForge project site [URL10e],
3. You may run the the source code distribution available at the SourceForge project site [URL10e],
4. Or you may checkout and run the source code distribution from the SVN available at [URL10f].

This Section will explain the possibilities (1) and (4).

1.1.1 Installing Dresden OCL2 for Eclipse using the Eclipse Update Site

To install Dresden OCL2 for Eclipse via the *Eclipse Update Site*, you have to start an *Eclipse* instance and select the menu option *Help -> Install New Software ...*. Enter the path

`http://dresden-ocl.sourceforge.net/downloads/updatesite/`

and press the *Add...* button (see Figure 1.1). In the new opened window you can additionaly enter a name for the update site (see Figure 1.2).

Now you can select the features of Dresden OCL2 for Eclipse which you want to install. Select them and click on the *Next >* button (see Figure 1.3). An overview about all features of Dresden OCL2 for Eclipse can be found in Table 2. Follow the wizard and agree with the user license. Then the Toolkit will be installed. Afterwards, you should restart your Eclipse application to finish the installation.

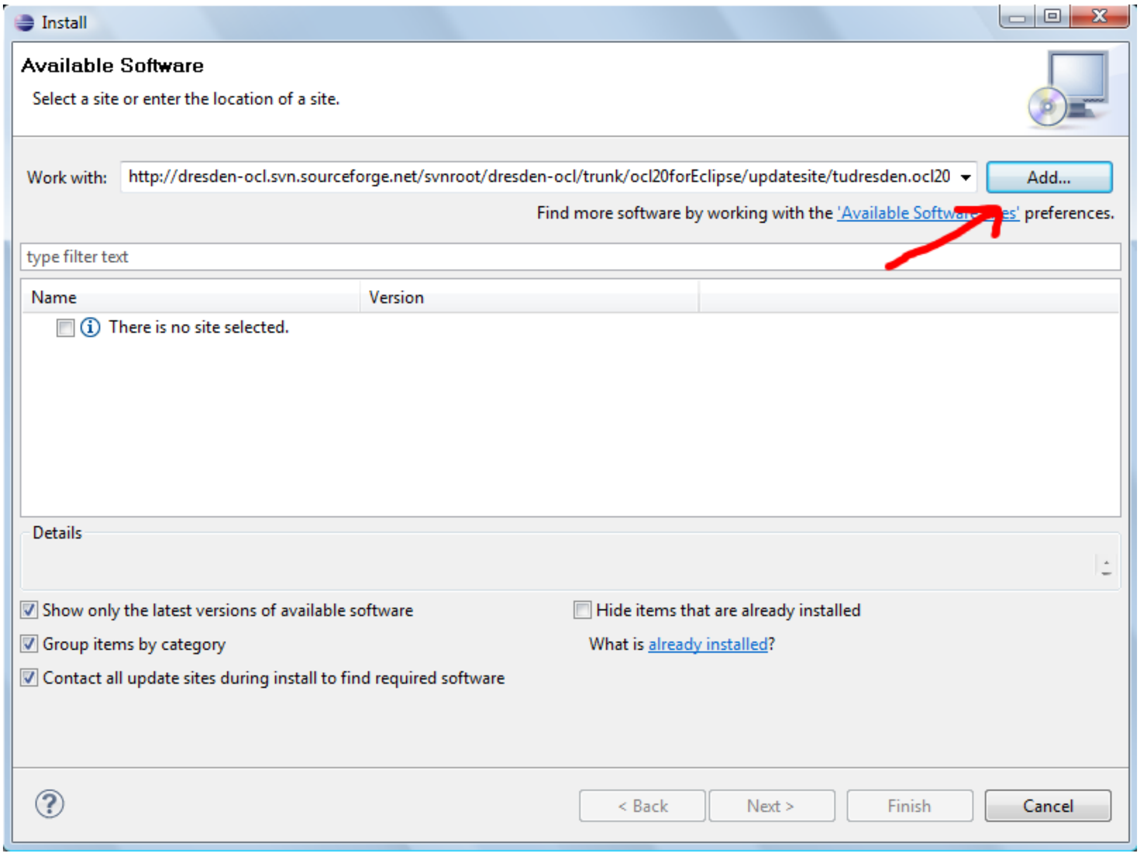

Figure 1.1: Adding an Eclipse Update Site (Step 1).

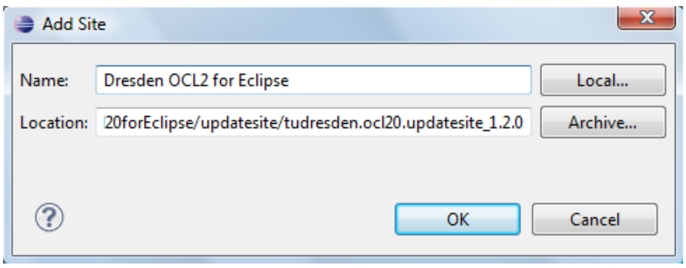

Figure 1.2: Adding an Eclipse Update Site (Step 2).

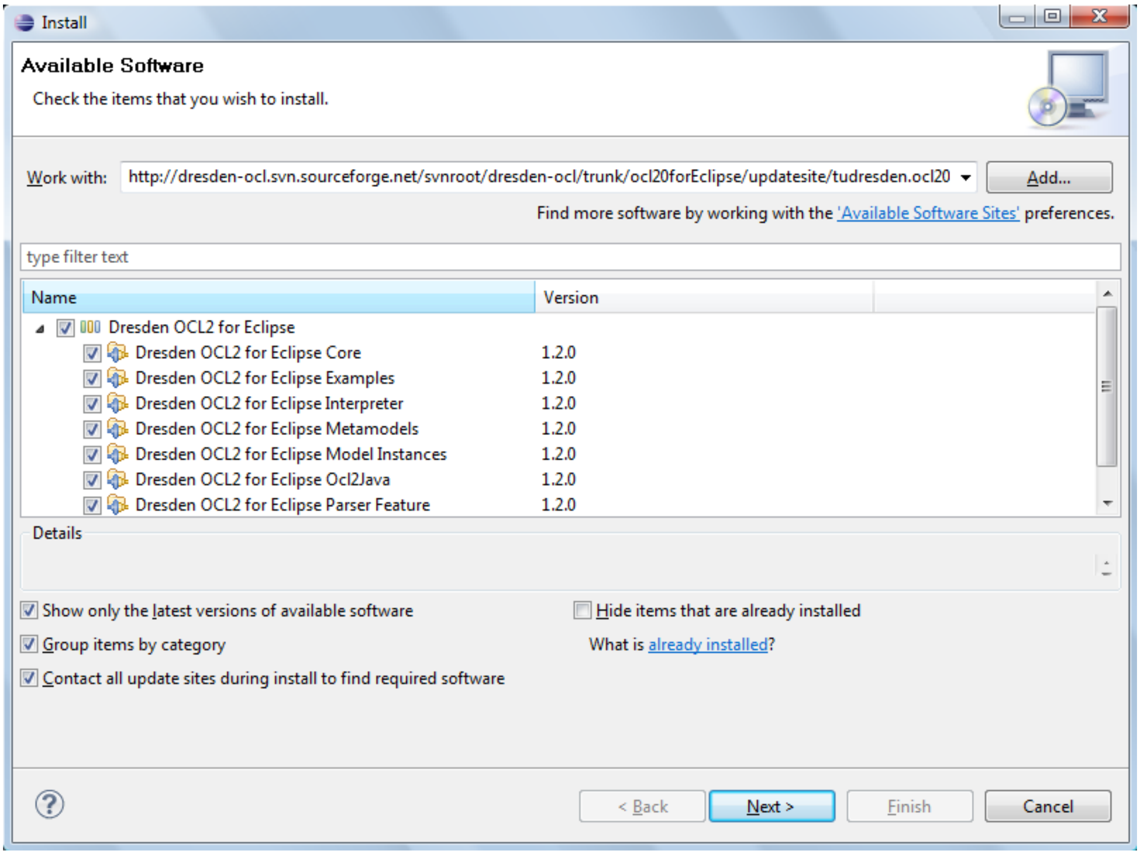

Figure 1.3: Selecting features of Dresden OCL2 for Eclipse.

1.1.2 Importing Dresden OCL2 for Eclipse from the SVN

To use Dresden OCL2 for Eclipse by checking out the source code from the SVN you need to install an SVN client. In the following we use the *Eclipse Subversive* plug-in and at least one of the *SVN Connectors* available at [URL10d].

After installing Eclipse Subversive, a new *Eclispe Perspective* providing access to SVN should exist. The perspective can be opened via the menu *Window > Open Perpective > Other... > SVN Repository Exploring*. In the view *SVN Repositories* you can add a new repository using the URL `https://dresden-ocl.svn.sourceforge.net/svnroot/dresden-ocl/` (see Figure 1.4).

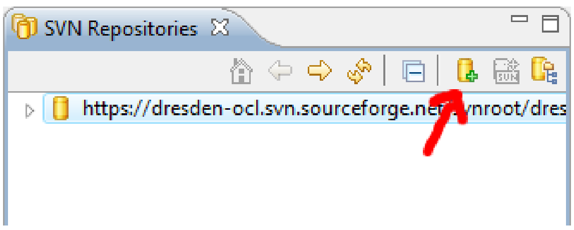

Figure 1.4: Adding an SVN repository.

After pushing the *Finish* button, the SVN repository root should be visible in the *SVN Repositories* view. To checkout the plug-ins, you have to select them in the repository directory `trunk/ocl20-forEclipse/eclipse` and use the *Checkout...* function in the context menu (see Figure 1.5).

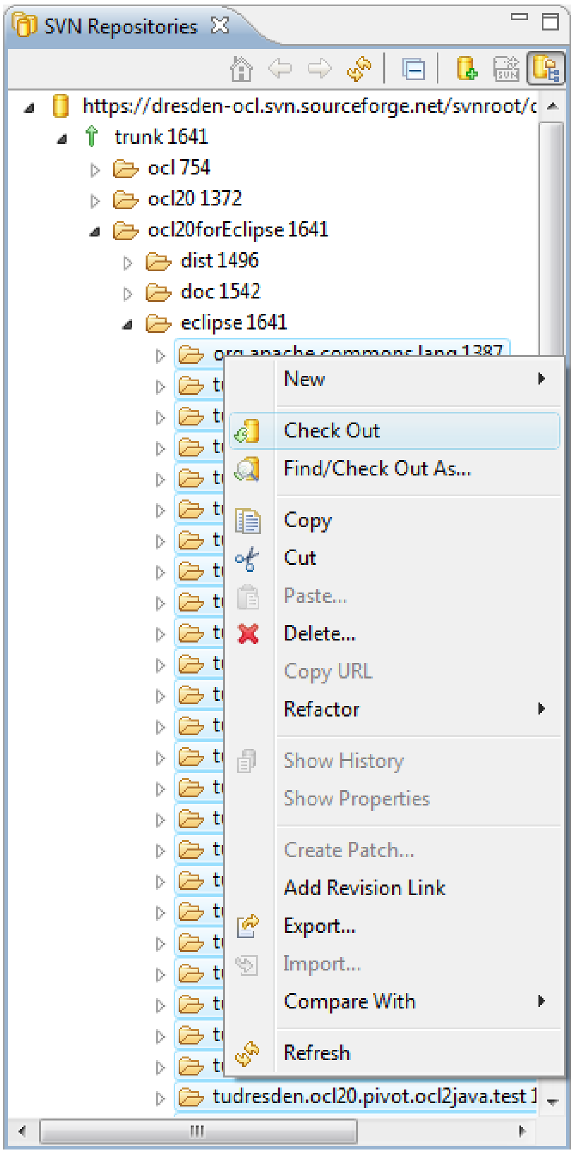

Figure 1.5: Checkout of the Dresden OCL2 Toolkit plug-in projects.

1.1.3 Which Plug-ins do I need at least?

An often asked question is "Which plug-ins are at least required to run Dresden OCL2 for Eclipse?" Well, the answer is: "That depends."

That depends on the things you want to do with Dresden OCL2 for Eclipse. Table 2 shows a list of the currently existing plug-ins of Dresden OCL2 for Eclipse, that are related to different features. You should install at least the *Core* feature, at least one plug-in of the *Metamodels* feature, and the complete *Parser* feature. The *Interpreter* and *OCL2Java* features are only required if you want to interpret constraints or to generate code from constraints, respectively. If you import or interpret model instances, you need to install the *Model Instances* feature as well. The examples of the *Example* feature are only required to run the examples provided in the tutorials available at [URL10h]. We recommend to install all provided features.

1.1.4 Building the OCL2 Parser

If you decided to run Dresden OCL2 for Eclipse as source code plug-ins from an Eclipse workspace, you need to build the *OCL2 Parser* via an *Ant* build script. If you installed the Toolkit using the update site, you can skip this Subsection.

To build the OCL2 Parser select the file `build.xml` in the project `tudresden.ocl20.pivot.ocl2-parser` and open the context menu. Select the function *Run As ... > Ant Build ...* (see Figure 1.6).

A new window should open. Select in the sub menu *JRE* the check box *Run in the same JRE as the workspace* and push the *OK* button (see Figure 1.7). Afterwards, the OCL2 Parser should be generated without errors. If an error like "Unable to find javac compiler." occurs, you might be trying to run the *Ant* script with a *Java Run-time Environment* instead of a *JDK* (For errors like this one) use the *Installed JREs...* button in the same window to select a JDK instead.

After executing the build script successfully you need to update the projects in your workspace. Update the project `tudresden.ocl20.pivot.oclparser` via context menu (*Refresh*, see Figure 1.8).

Additionally, you need to recompile all depending projects. Select the function *Project > Clean... > Clean all projects* in the Eclipse menu to clean all projects. Now all the projects should not contain any errors anymore and should be executable.

1.2 LOADING MODELS, MODEL INSTANCES AND CONSTRAINTS

If you installed the Dresden OCL2 for Eclipse using the update site, you can execute the toolkit by re-starting your Eclipse distribution. If you imported the Toolkit as source code plug-ins into an Eclipse workspace, you have to start a new Eclipse instance. You can start a new instance via the menu *Run > Run As > Eclipse Application*. If the menu *Eclipse Application* is not available or disabled you need to select one of the plug-ins of the toolkit in the *Package Explorer* first.

1.2.1 The Simple Example

The use of Dresden OCL2 for Eclipse is explained using the *Simple Example* which is located in the plug-in `tudresden.ocl20.pivot.examples.simple`. Figure 1.9 shows a class diagram of the Simple Example.

Dresden OCL2 for Eclipse provides more examples than the *Simple Example*. The different examples use different meta-models which is possible with the *Pivot Model* architecture of the Toolkit. An overview about all examples provided with *Dresden OCL2 for Eclipse* is listed in Table 4. The Simple Example can be used with two different meta-models. These are *UML 2.0* (based on *Eclipse MDT UML2*) and *Java*.

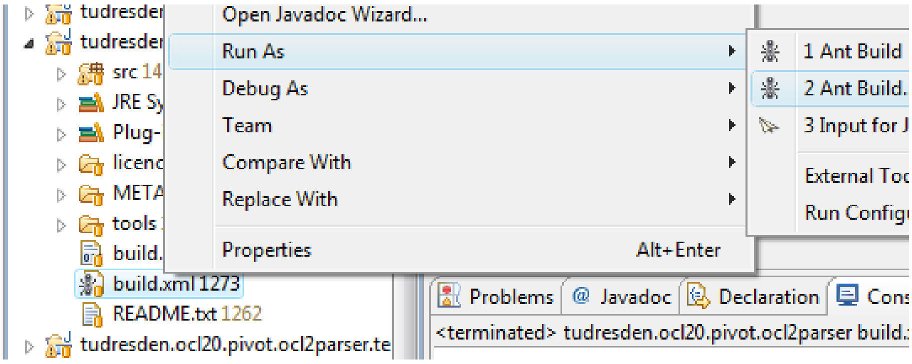

Figure 1.6: Executing the OCL2 Parser build script.

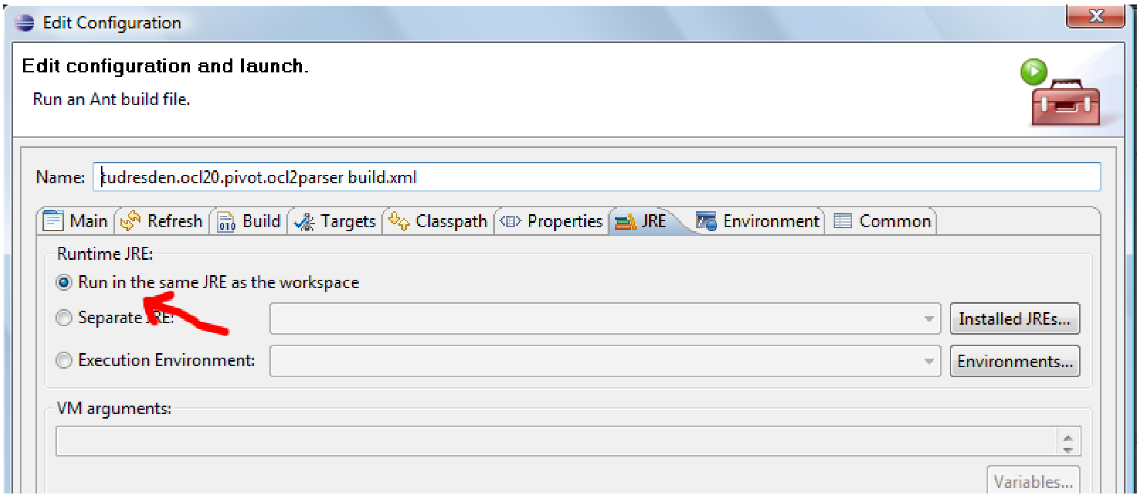

Figure 1.7: Settings of the JRE for the Ant build script.

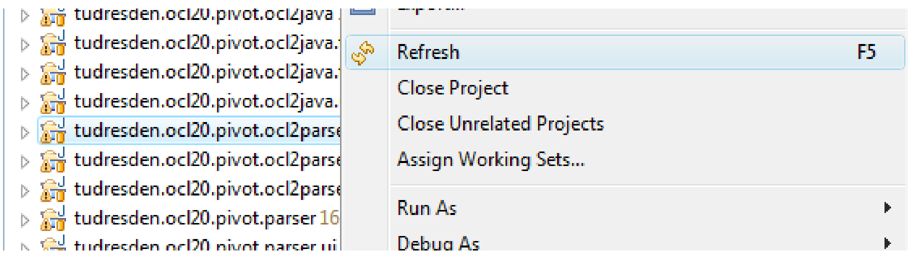

Figure 1.8: Refreshing the project "tudresden.ocl20.pivot.oclparser".

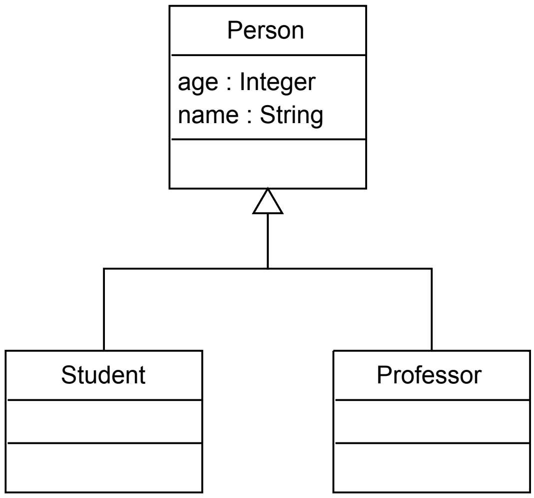

Figure 1.9: A class diagram describing the Simple Example model.

1.2.2 Loading a Domain-Specific Model

After starting Eclipse you have to load a model into the toolkit. If the plug-ins of Dresden OCL2 for Eclipse have been installed using the update site, the Simple Example plug-in has to be imported into the *Workspace* first. Create a new Java project into your workspace and select the *Import Wizard General > Archive File*. In the following window select the *plugins* directory in your Eclipse root folder, select the archive `tudresden.ocl20.pivot.examples.simple_1.0.0.jar` and push the *Finish* button.

Now you can load a model. Select the menu option *Dresden OCL2 > Load Model*. In the opened wizard you have to select a model file and a meta-model for the model (see Figure 1.10). Push the button *Browse Workspace...* and select the file `model/simple.uml` inside the Simple Example Project. Then select the meta-model UML2 and push the *Finish* button.

Figure 1.11 shows the loaded Simple Example model, which uses UML2 as its meta-model. Via the menu button of the *Model Browser* (the little triangle in the right top corner) you can switch between different models loaded into Dresden OCL2 for Eclipse (see Figure 1.12) With the red ex you can close the currently selected model.

1.2.3 Loading a Model Instance

After loading a model, you can load a *model instance* using another wizard. Use the menu option *Dresden OCL2 > Load Model Instance*. In the opened wizard you have to select a model instance (in this tutorial we used the file `bin/tudresden/ocl20/pivot/examples/simple/instance/ModelInstanceProviderClass.class` of the Simple Example (see Figure 1.13). Besides the model instance resource you have to select a model for which the model instance shall be loaded and the type of model instance you want to load (we want to load a *Java Instance*).

Figure 1.14 shows the loaded model instance of the Simple Example model. Like in the model browser you can switch between different model instances and you can close selected instances. Note that the *Model Instance Browser* only shows the model instances of the model actually selected in the model browser. By switching the domain specific model, you also switch the pool of model instances available in the model instance browser.

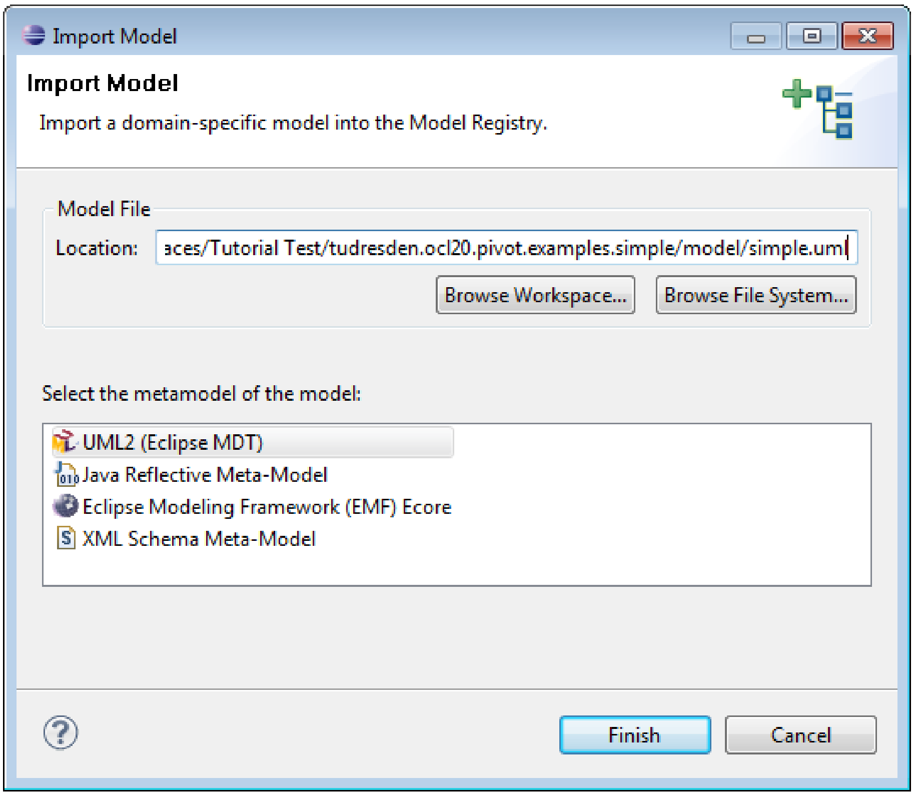

Figure 1.10: Loading a domain-specific model.

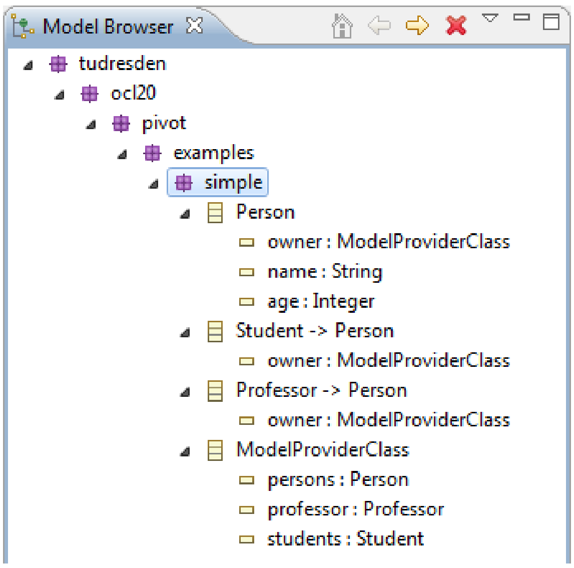

Figure 1.11: The Simple Example model in the model browser.

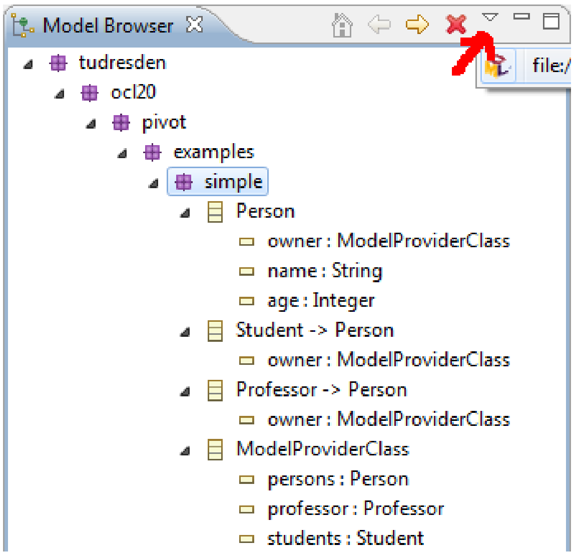

Figure 1.12: You can switch between different models using the little triangle.

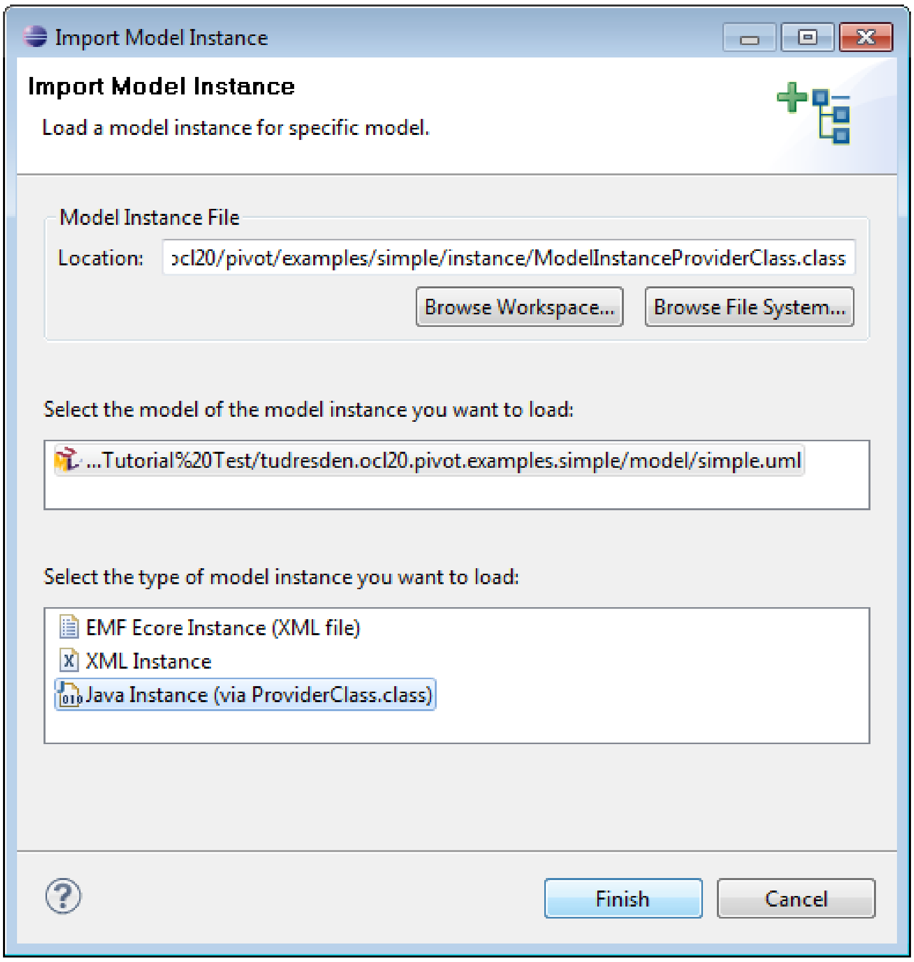

Figure 1.13: Loading a simple model instance.

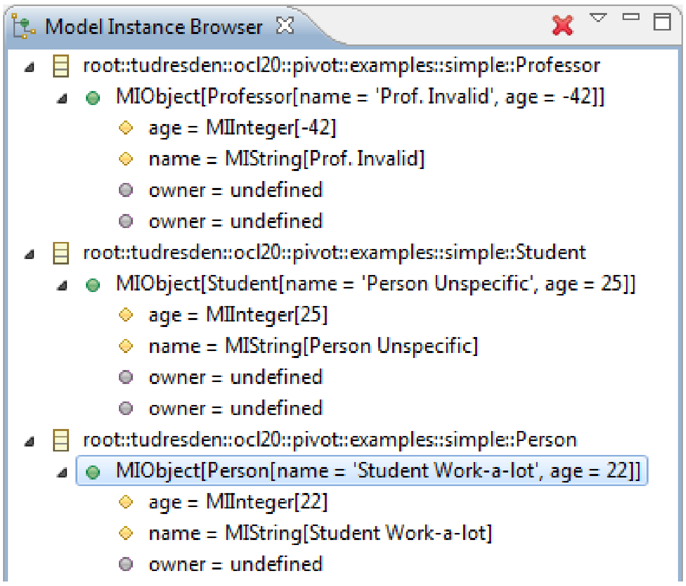

Figure 1.14: A simple model instance in the Model Instance Browser.

1.2.4 Parsing OCL expressions

Before you can work with OCL constraints, you have to load them like the domain-specific model and the model instance into the toolkit. Use the menu option *Dresden OCL2 > OCL Expressions* and select an OCL file. In this tutorial we used the OCL file `constraints/invariants.ocl` of the Simple Example. (see Figure 1.15). The constraints of the file `constraints/invariants.ocl` are shown in Listing 1.1.

The expressions of the selected OCL file are loaded into the actually selected model. Figure 1.15 shows the *Model Browser* containing the model and the parsed expressions.

```
package tudresden::ocl20::pivot::examples::simple

-- The age of Person can't be negative.
context Person
inv: age >= 0

-- Students should be 16 or older.
context Student
inv: age > 16

-- Proffesors should be at least 30.
context Professor
inv: not (age < 30)

endpackage
```

Listing 1.1: The invariants of the simple examples.

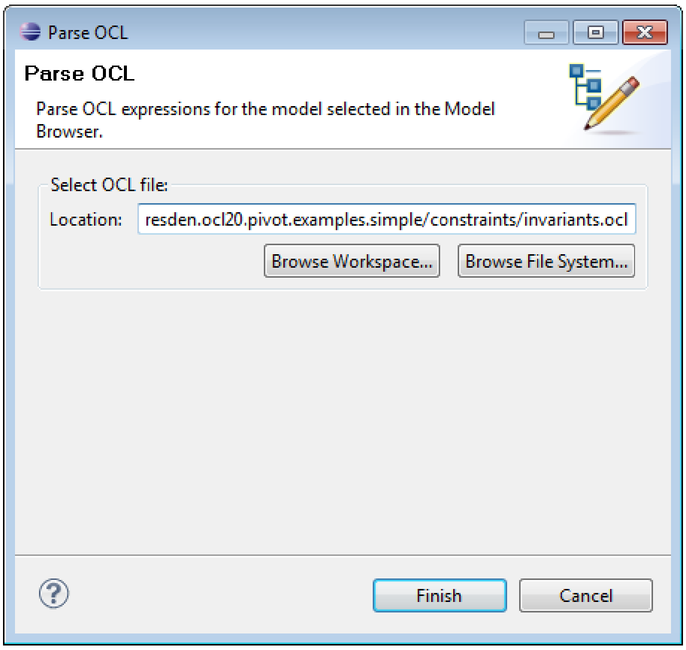

Figure 1.15: The import of OCL expressions.

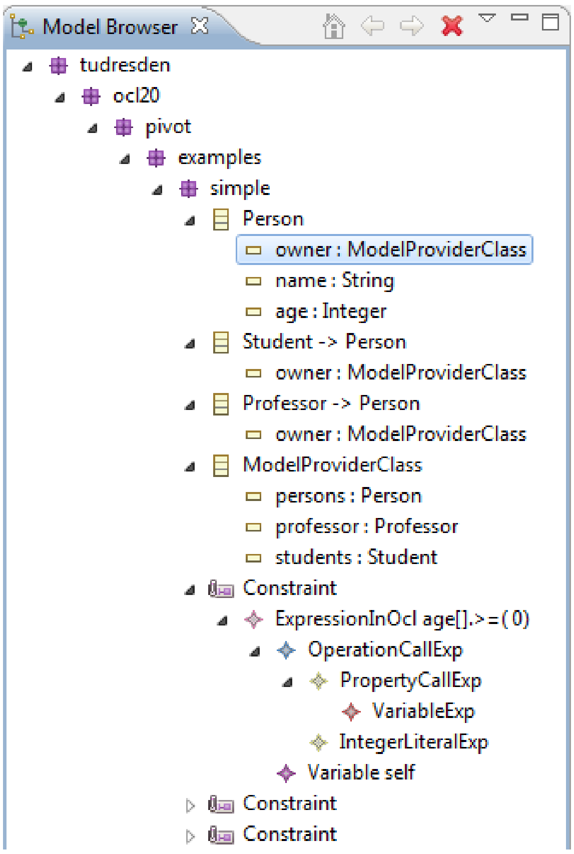

Figure 1.16: Parsed expressions and the model in the Model Browser.

1.3 SUMMARY

This Chapter described how to use Dresden OCL2 for Eclipse. It explained how to install the toolkit's plug-ins. Afterwards, the loading of domain-specific models, model instances and OCL constraints into the toolkit was explained.

Now, the imported models can be used with the tools provided by Dresden OCL2 for Eclipse. For example you can use the *OCL2 Interpreter* of Dresden OCL2 for Eclipse to interpret OCL constraints for a given model and model instance (as explained in Chapter 2) or you can use the *OCL22Java Code Generator* to generate *AspectJ* code for a loaded model and OCL constraints (as explained in Chapter 3).

2 OCL CONSTRAINT INTERPRETATION

Chapter written by Claas Wilke

This Chapter describes how the OCL2 Interpreter provided with Dresden OCL2 for Eclipse can be used. How to install and run *Dresden OCL2 for Eclipse* and how to load models and OCL constraints will not be explained in this Chapter. This Chapter assumes that the user is familiar with such basic uses of the toolkit. A general introduction into Dresden OCL2 for Eclipse can be found in Chapter 1.

2.1 THE SIMPLE EXAMPLE

This Chapter uses the *Simple Example* which is provided with Dresden OCL2 for Eclipse located in the plug-in `tudresden.ocl20.pivot.examples.simple`. An overview over all examples provided with Dresden OCL2 for Eclipse can be found in Table 4. An introduction into the Simple Example can be found in Section 1.2.1. The model of the example defines three classes: The class `Person` has the attributes `age` and `name`. Two subclasses of `Person` are defined, `Student` and `Professor`.

To import the Simple Example into our Eclipse workspace we create a new Java project called `tudresden.ocl20.pivot.examples.simple` and use the import wizard *General > Archive File* to import the example provided as a jar archive. In the following window we select the directory where the jar file is located (probably the `plugins` directory into the Eclipse root folder), select the archive `tudresden.ocl20.pivot.examples.simple.jar` and push the *Finish* button (if you use a source code distribution of Dresden OCL2 for Eclipse instead, you can simply import the project `tudresden.ocl20.pivot.examples.simple` using the import wizard *General -> Existing Projects into Workspace*). Figure 2.1 shows the *Package Explorer* containing the imported project.

The project provides a model file that contains a class diagram (the model file is located at `model/simple.uml`) and the constraint file we want to interpret (located at `constraints/allConstraints.ocl`). Listing 2.1 shows the constraints defined in the constraint file.

First, the constraint file defines three simple invariants that denote, that the `age` of every `Person` must always zero or greater than zero. Furthermore, the `age` of every `Student` must be greater than 16 and the `age` of every `Professor` does not have to be lesser than 30.

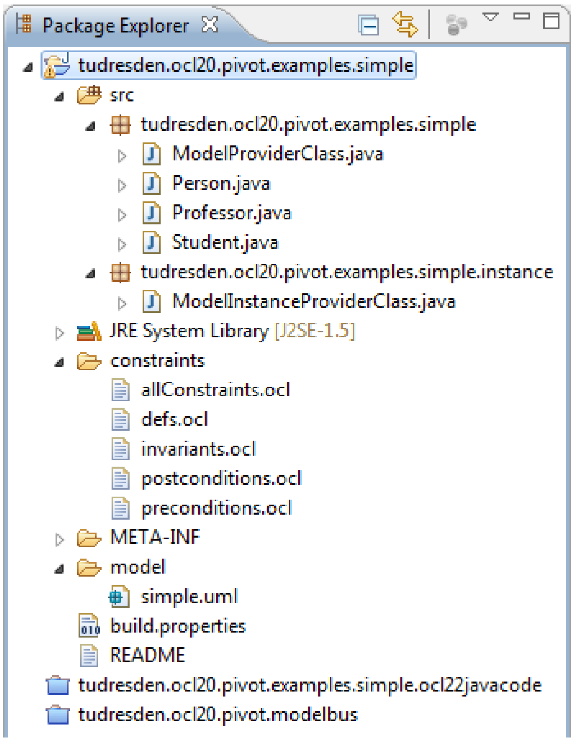

Figure 2.1: The package explorer containing the project which is required to run this tutorial.

```
-- The age of Person can not be negative.
context Person
inv: age >= 0

-- Students should be 16 or older.
context Student
inv: age > 16

-- Proffesors should be at least 30.
context Professor
inv: not (age < 30)

-- Returns the age of a Person.
context Person
def: getAge(): Integer = age

-- Before returning the age, the age must be defined.
context Person::getAge()
pre: not age.oclIsUndefined()

-- The result of getAge must equal to the age of a Person.
context Person::getAge()
post: result = age
```

Listing 2.1: The constraints contained in the constraint file.

In addition to that the constraint file contains a definition constraint that defines a new operation `getAge()` which returns the `age` of a `Person`. A precondition checks, that the `age` must be defined before it can be returned by the operation `getAge()`. And finally, a postcondition which checks, whether or not the result of the operation `getAge()` is the same as the `age` of the `Person`.

2.2 PREPARATION OF THE INTERPRETATION

To prepare the interpretation we have to import the model `model/simple.uml` for which we want to interpret constraints into the *Model Browser*. We use the import wizard for domain-specific models of the toolkit to import the model. This procedure is explained in Section 1.2.3. Furthermore, we have to import a model instance for which the constraints shall be interpreted into the *Model Instance Browser*. We use another import wizard to import the model instance `bin/tudresden/ocl20/pivot/examples/simple/ModelProviderClass.class`. Finally, we have to import the constraint file `constraints/allConstraints.ocl` containing the constraints we want to interpret. The import is done by an import wizard again. Afterwards, the *Model Browser* should look like illustrated in Figure 2.2 and the *Model Instance Browser* should look like shown in Figure 2.3.

The opened model instance contains three instances of the classes defined in the Simple Example model. One instance of `Person`, one instance of `Student` and one instance of `Professor`. For these three instances we now want to interpret the imported constraints.

2.3 OCL INTERPRETATION

Now we can start the interpretation. To open the OCL2 Interpreter we use the menu option *Dresden OCL2 > Open OCL2 Interpreter*. The *OCL2 Interpreter View* should now be visible (see Figure 2.4).

By now, the *OCL2 Interpreter View* does not contain any result. Besides the results table, the view provides four buttons to control the OCL2 Interpreter. The buttons are shown in Figure 2.5. With the first button (from left to right) constraints can be prepared for interpretation. The second button can be used to add variables to the *Interpreter's Environment*. The third button provides the core functionality, it can be used to start the interpretation. And finally, the fourth button provides the possibility to delete all results from the *OCL2 Interpreter View*. The functionality of the buttons will be explained below.

2.3.1 Interpretation of Constraints

To interpret constraints, we simple select them in the *Model Browser* and push the button to interpret constraints (the third button from the left). First, we want to interpret the three invariants defining the range of the `age` of `Persons`, `Students` and `Professors`. We select them in the *Model Browser* (see Figure 2.6) and push the *Interpret* button. The result of the interpretation is now shown in the *OCL2 Interpreter View* (see Figure 2.7).

The invariant `age >= 0` has been interpreted for all three model objects. The results for the `Person` and the `Student` instances are `true` because their `age` is greater than zero. The result for the `Professor` instance is `false` because its `age` is `-42`.

The two other invariants were only interpreted for the `Student` or the `Professor` instance because their context is not the class `Person` but the class `Student` or the class `Professor`, respectively. Again, the `Student's` result is `true` and the `Professor's` result is `false`.

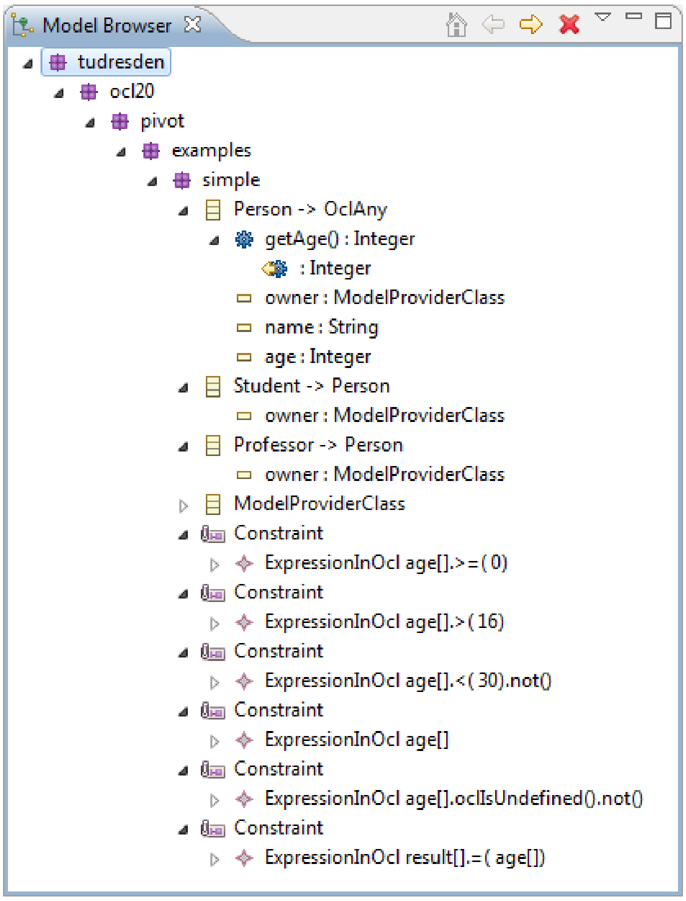

Figure 2.2: The model browser containing the simple model and its constraints.

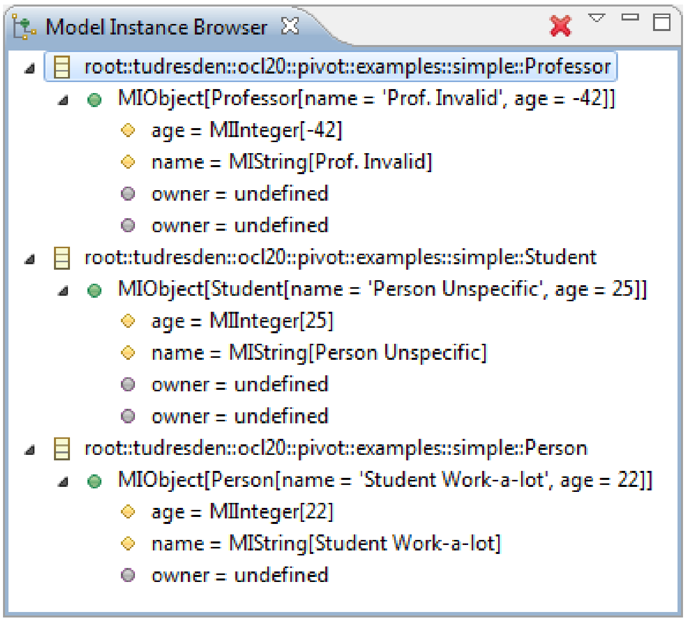

Figure 2.3: The model instance browser containing the simple model instance.

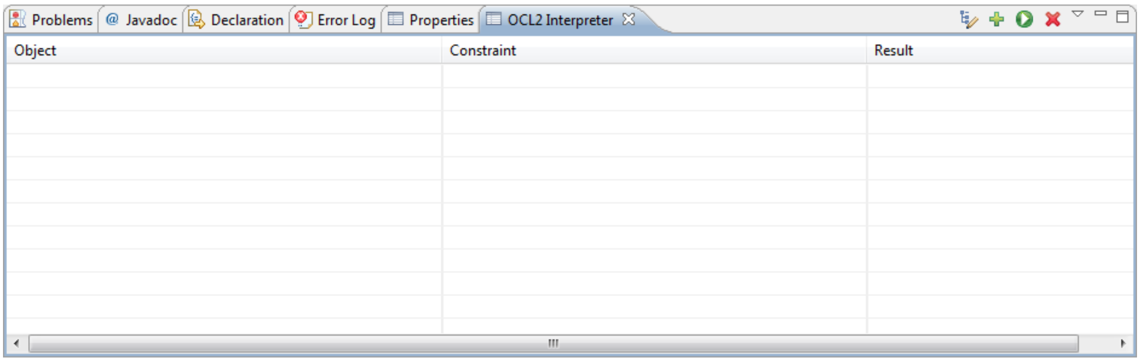

Figure 2.4: The OCL2 Interpreter View containing no results.

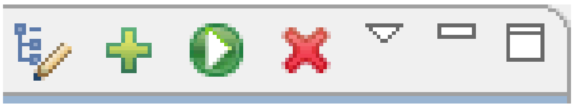

Figure 2.5: The buttons to control the OCL2 Interpreter.

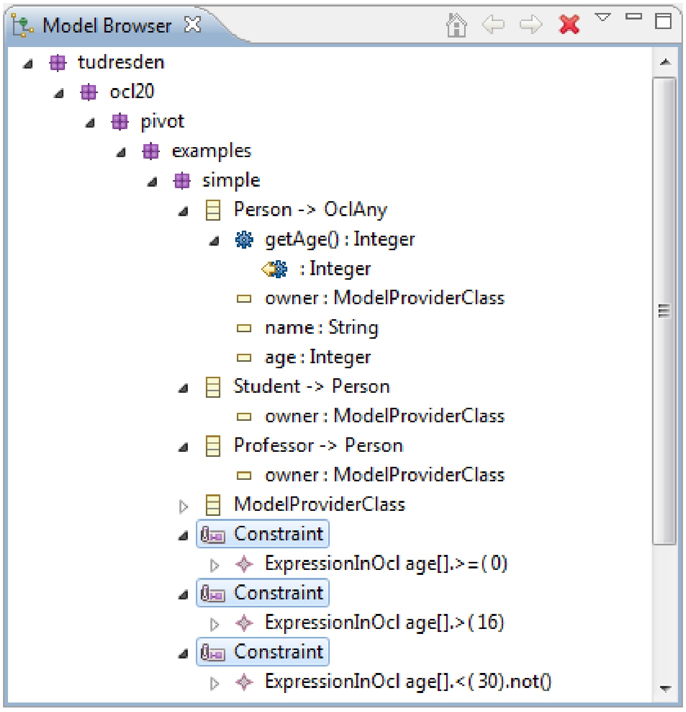

Figure 2.6: The three invariants selected in the Model Browser.

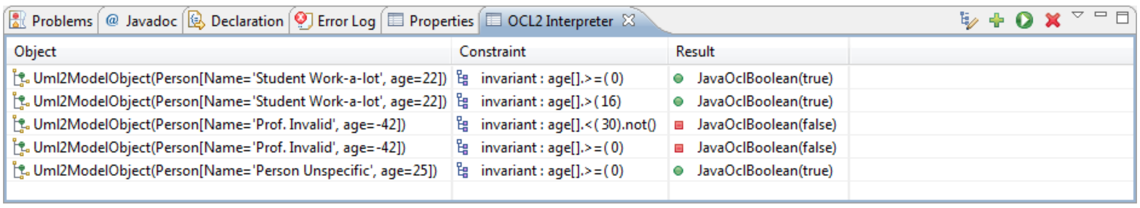

Figure 2.7: The results of the three invariants for all model instances.

2.3.2 Preparation of Constraints

Some constraints cannot be interpreted because they are no constraints in the natural sense of the word constraint. OCL2 enables us to use OCL expressions to define new attributes and methods or to initialize attributes and methods. Such `def`, `init` and `body` constraints cannot be interpreted to `true` or `false`, because their result type has not to be `Boolean`. Furthermore, they can be used to alter the results of other constraints that shall be interpreted.

The `allConstraints.ocl` file contains a definition constraint, which defines the method `getAge()` for the class `Person`. Before we can refer to this method in other constraints we have to prepare the definition constraint to ensure, that the interpretation of other constraints will finish with the right results.

To prepare the definition constraint, we select it in the *Model Browser* (see Figure 2.8) and push the *Prepare* button (the first button from the left).

The preparation does not result with a visible result in the *OCL2 Interpreter View*. But the method definition of the constraint has been added to the *Interpreter Environment* of the OCL2 Interpreter. Thus, we can interpret the next constraint now. This constraint is the precondition which checks that the `age` of any `Person` must be defined before the method `getAge()` can be invoked.

We select the constraint in the *Model Browser* (see Figure 2.9) and push the *Interpret* button. The result of the interpretation is shown in Figure 2.10.

The interpretation finishes for all three instances successfully because the attribute `age` has been set for all three instances.

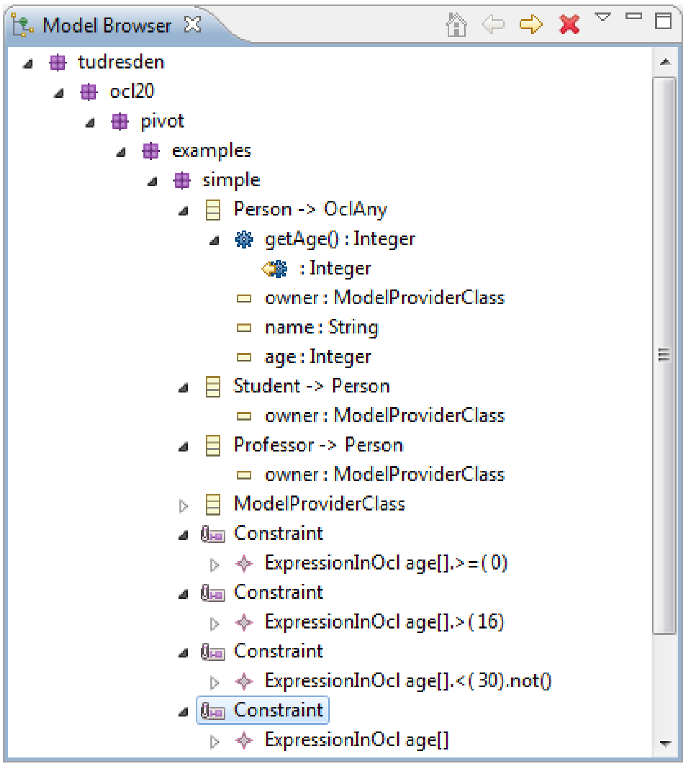

Figure 2.8: The definition constraint selected in the Model Browser.

2.3.3 Adding Variables to the Environment

By preparing the definition constraint we added some information to the *Interpretation Environment* that was necessary to interpret other constraints. For some constraints we have to add further information which can not be provided by the preparation of other constraints.

For example, our last constraint–a postcondition–compares the result of the method `getAge()` with the attribute `age` of the referenced `Person` instance. Therefore, OCL provides the special variable `result` in postconditions which contains the result of the constrained method's execution. Using the *OCL2 Interpreter View* we cannot execute the method `getAge()` and store the result in the `result` variable. We can interpret the postcondition in a specific context which has to be prepared by hand only. We have to set the result variable manually.

If we interpret the postcondition constraint (the sixth and last constraint in the *Model Browser*) without setting the `result` variable, the constraint results in a `undefined` result for all three model instances (see Figure 2.11).

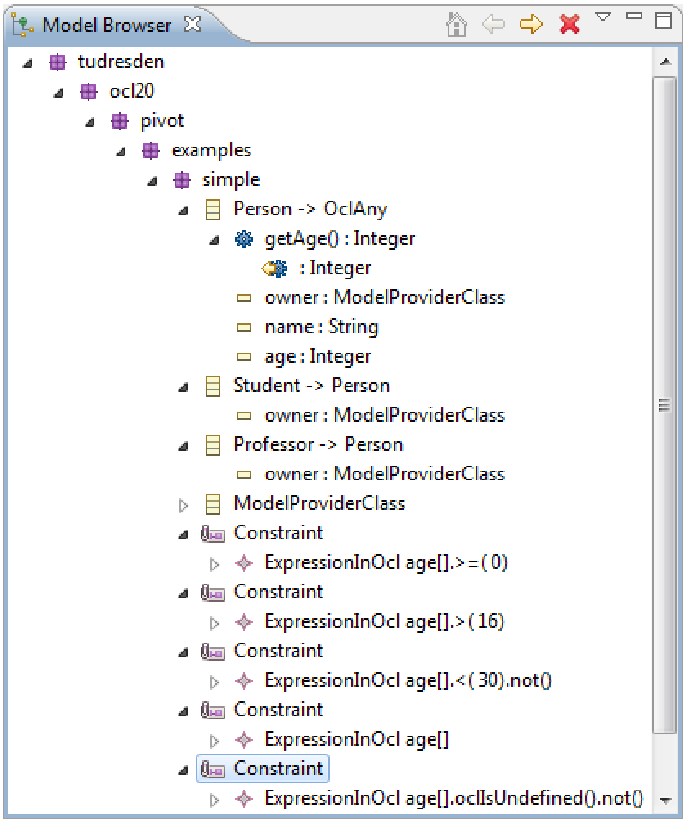

Figure 2.9: The precondition selected in the Model Browser.

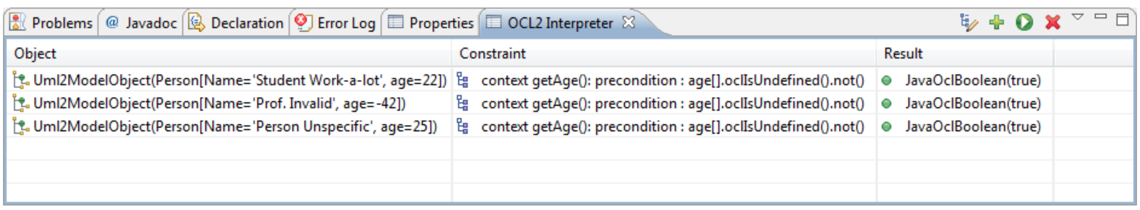

Figure 2.10: The results of the precondition for all instance objects.

To prepare the variable, we push the button to add new variables to the Interpreter Environment (the second button from the left) and a new window opens which we can use to specify new variables. We enter the name `result`, select the variable type `Integer` and enter the value 25. Then we push the *OK* button (see Figure 2.12. The result variable has now been added to the Interpreter's Environment.

Now, we can interpret the postcondition again. The result is shown in Figure 2.13. The results for the `Student` and `Professor` instances are both `false` because their `age` attribute is not equal to 25 and thus the `result` value does not match to the `age` attribute. But the interpretation for the `Person` instance succeeds because its `age` is 25.

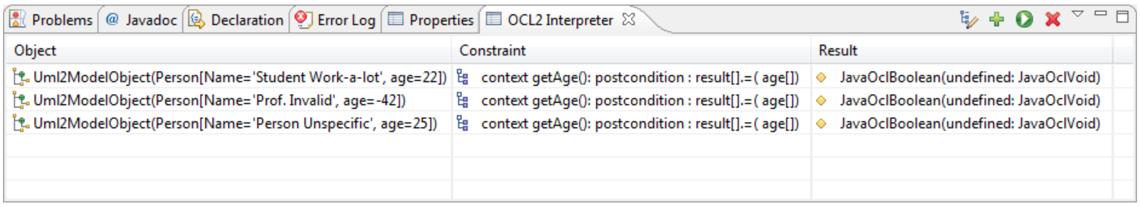

Figure 2.11: The results of the postcondition without preparing the result variable.

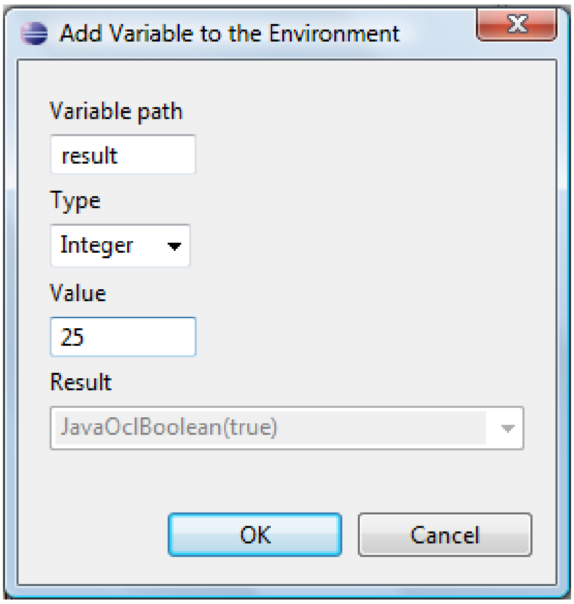

Figure 2.12: The window to add new variables to the environment.

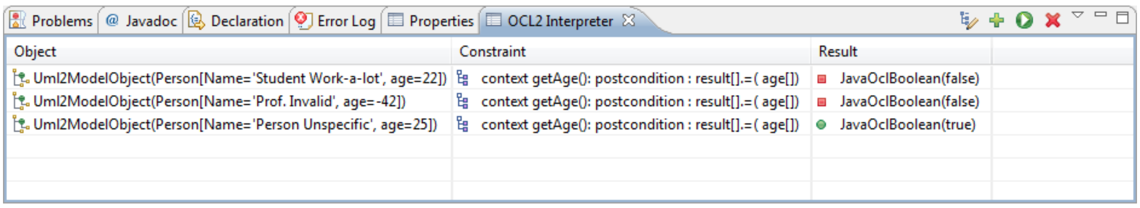

Figure 2.13: The results of the postcondition with result variable preparation.

2.4 SUMMARY

This Chapter described how OCL constraints can be interpreted using the OCL2 Interpreter of Dresden OCL2 for Eclipse. The preparation and interpretation of constraints has been explained, the addition of new variables to the Interpreter Environment has been shown.

3 ASPECTJ CODE GENERATION

Chapter written by Claas Wilke

This Chapter describes how the Java Code Generator *OCL22Java* provided with Dresden OCL2 for Eclipse can be used. A general introduction into Dresden OCL2 for Eclipse can be found in Chapter 1. A detailed documentation of the development of OCL22Java can be found in the Minor Thesis (Großer Beleg) of Claas Wilke [Wil09].

In addition to the general Eclipse installation the *AspectJ Development Tools (AJDT)* are required to execute the code generated with OCL22Java. The AJDT plug-ins can be found at the AJDT website [URL10b].

3.1 CODE GENERATOR PREPARATION

This Chapter uses the *Simple Example* which is provided with Dresden OCL2 for Eclipse and has been introduced in Subsection 1.2.1 To import the Simple Example into our Eclipse workspace we have to create a new Java project into our Workspace (here called `tudresden.ocl20.pivot.examples.simple`) and use the import wizard *General -> Archive File* to import the example provided as a JAR archive. In the following window we select the directory were the JAR file is located (probably the `plugins` or `dropins` directory inside the Eclipse root folder). We select the archive `tudresden.ocl20.pivot.examples.simple.jar` and push the *Finish* button (if you use a source code distribution of Dresden OCL2 for Eclipse instead, you can simple import the project `tudresden.ocl20.pivot.examples.simple` using the import wizard *General -> Existing Projects into Workspace*).

Next, we have to import a second project called `tudresden.ocl20.pivot.examples.simple.ocl22javacode`. We can use the same mechanism explained above, but instead of a Java project we now create an *AspectJ Project* before we import the archive file (if the wizard to create an AspectJ project is not available you have to install the AJDT first). Figure 3.1 shows the *Package Explorer* containing both imported projects.

After importing the second plug-in, we have to add the JUnit4 library to the project's build path. Push the second mous button on the project in the *Package Explorer* and select the menu item *Properties* (see Figure 3.2). In the new opened window select the sub-menu *Java Build Path -> Libraries* and push the *Add Library...* button (see Figure 3.3). In the following window select *JUnit*, push *Next*, select *JUnit 4* and push *Finish*. Push the *OK* button to close the project's properties. The project should not contain any compile errors anymore.

Now we have imported all files we need to run this tutorial. The first project provides a model file which contains the simple class diagram which has been explained in Subsection 1.2.1 (the model file is located at `model/simple.uml`) and the constraint file we want to generate code for (the constraint file is located at `constraints/invariants.ocl`). Listing 3.1 shows one invariant that is contained in the constraint file for which we want to generate code. The invariant declares, that the `age` of any `Person` must be greater or equal to zero at any time during the life cycle of the `Person`.

The second project provides the test class `src/tudresden.ocl20.pivot.examples.simple.constraints.InvTest.java` which contains a JUnit test case that checks, whether or not the mentioned constraint is enforced during run-time. The test case creates two `Persons` and tries to set their `age`. The `age` of the second `Person` is set to `-3` and thus the constraint is violated. The test case expects that a run-time exception is thrown, if the constraint is violated.

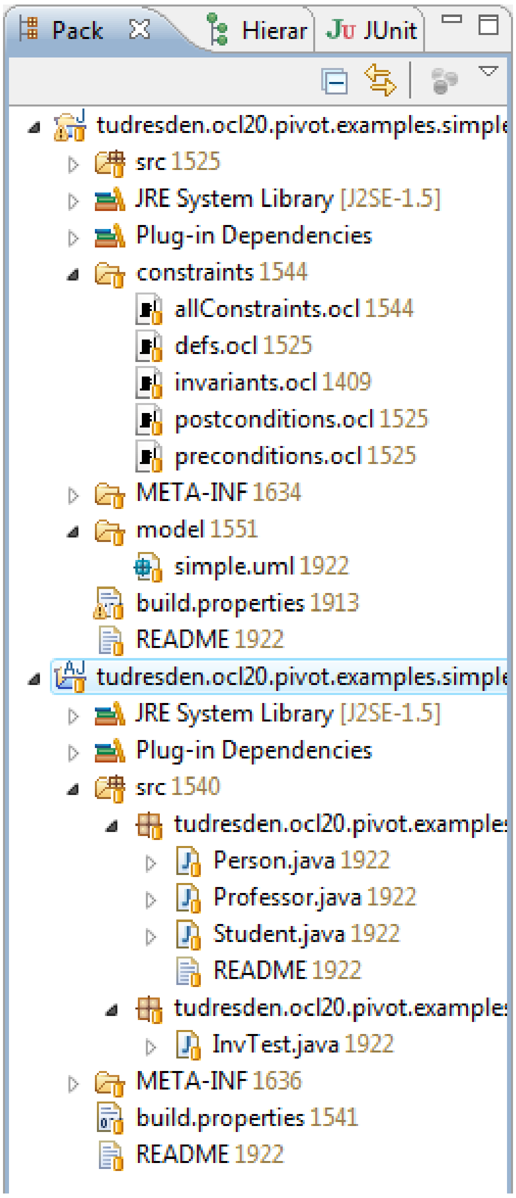

Figure 3.1: The two projects which are required to run this tutorial.

The code for the mentioned constraint has not been generated yet and thus the exception will not be thrown. We run the test case by opening the context menu on the Java class in the *Package Explorer* and selecting the menu item *Run as -> JUnit Test*. The test case fails because the exception is not thrown (see Figure 3.4). To fulfill the test case we have to generate the ApsectJ code for the constraint which enforces the constraint's condition. How to generate such code will be explained in the following.

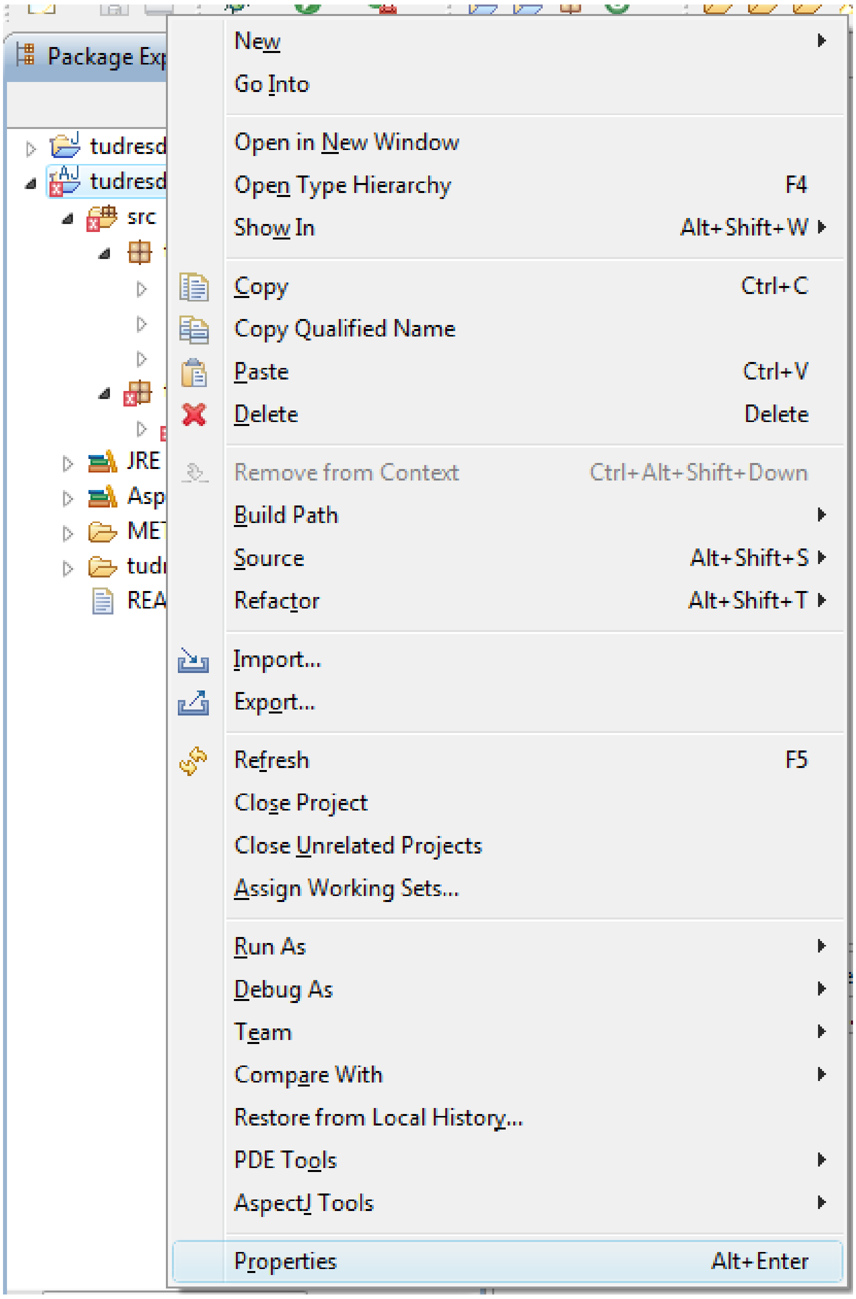

Figure 3.2: Selecting the properties settings.

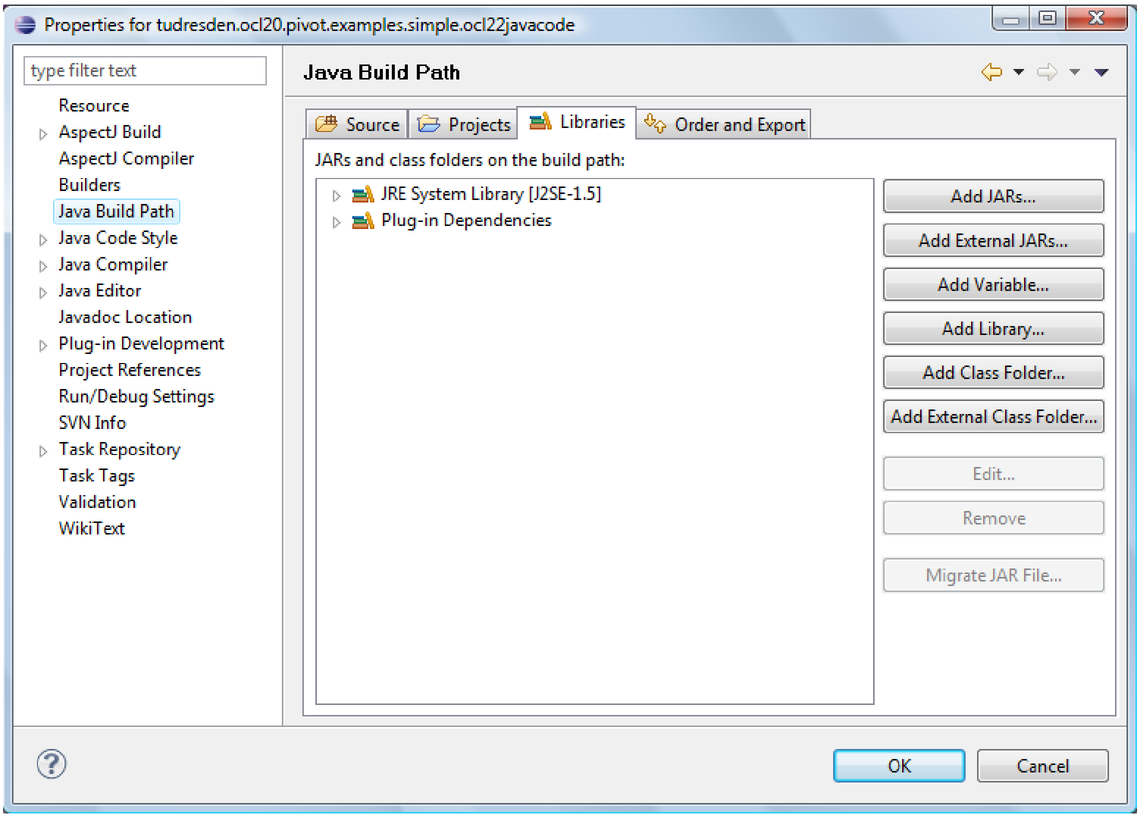

Figure 3.3: Adding a new library to the build path.

```
-- The age of Person can not be negative.
context Person
inv: age >= 0
```

Listing 3.1: A simple invariant.

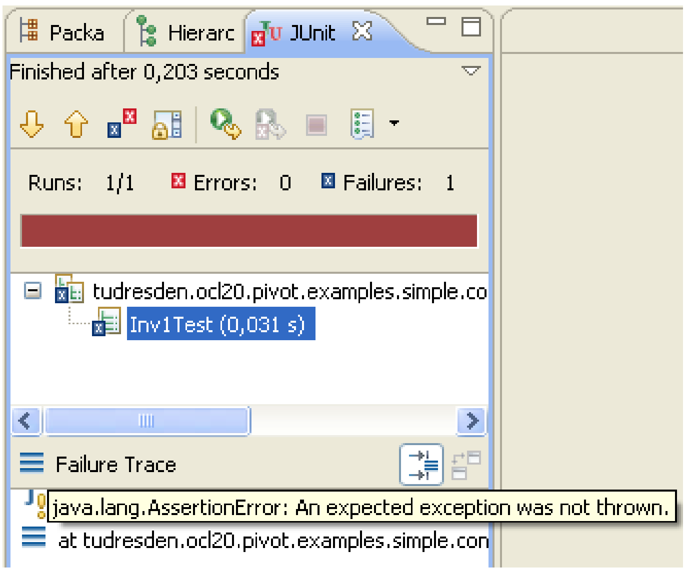

Figure 3.4: The result of the JUnit test case.

3.2 CODE GENERATION

To prepare the code generation we have to import the model `model/simple.uml` into the *Model Browser*. We use the import wizard for domain-specific models of the toolkit to import the model. This procedure is explained in the already mentioned introduction in Chapter 1. Afterwards, we have to import the constraint file `constraints/invariant.ocl` which is done by an import wizard again. After the importation, the *Model Browser* should look like illustrated in Figure 3.5. Now we can start the code generation.

To start the code generation we open the menu *DresdenOCL* and select the item *Generate AspectJ Constraint Code*.

3.2.1 Selecting a Model

A wizard opens and we have to select a model for code generation (see Figure 3.6). We select the `simple.uml` model and push the *Next* button.

3.2.2 Selecting Constraints

As a second step we have to select the constraints for that we want to generate code. We only select the constraint that enforces that the `age` of any `Person` must be equal to or greater than zero and push the *Next* button (see Figure 3.7).

3.2.3 Selecting a Target Directory

Next, we have to select a target directory into that our code generated shall be stored. We select the source directory of our second project (which is `tudresden.ocl20.pivot.examples.simple.`

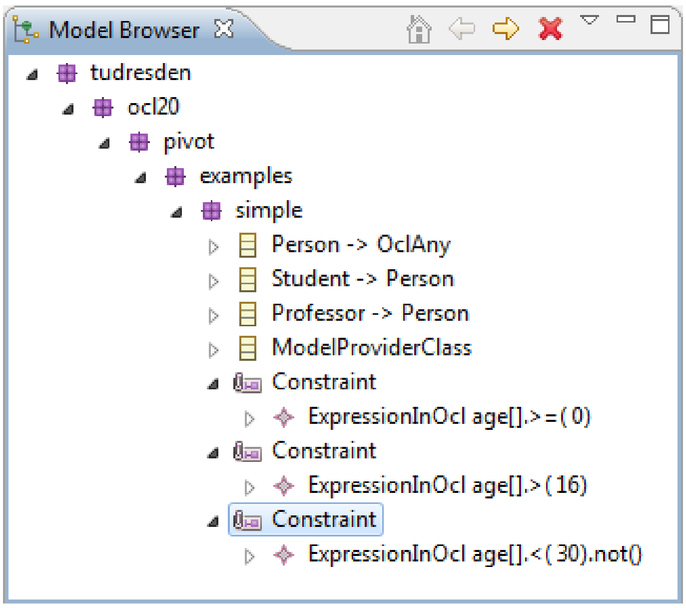

Figure 3.5: The model browser containing the simple model and its constraints.

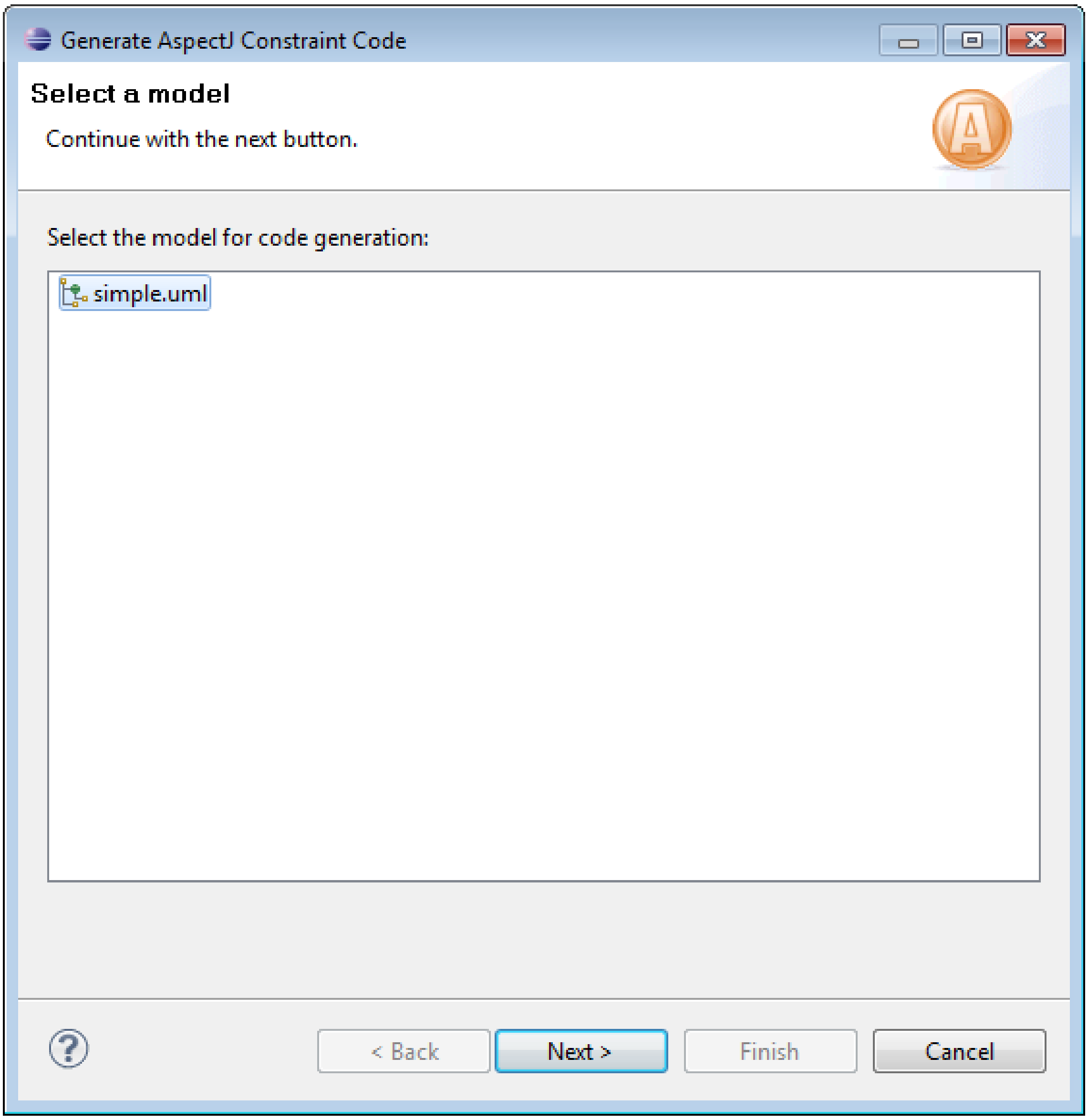

Figure 3.6: The first step: Selecting a model for code generation.

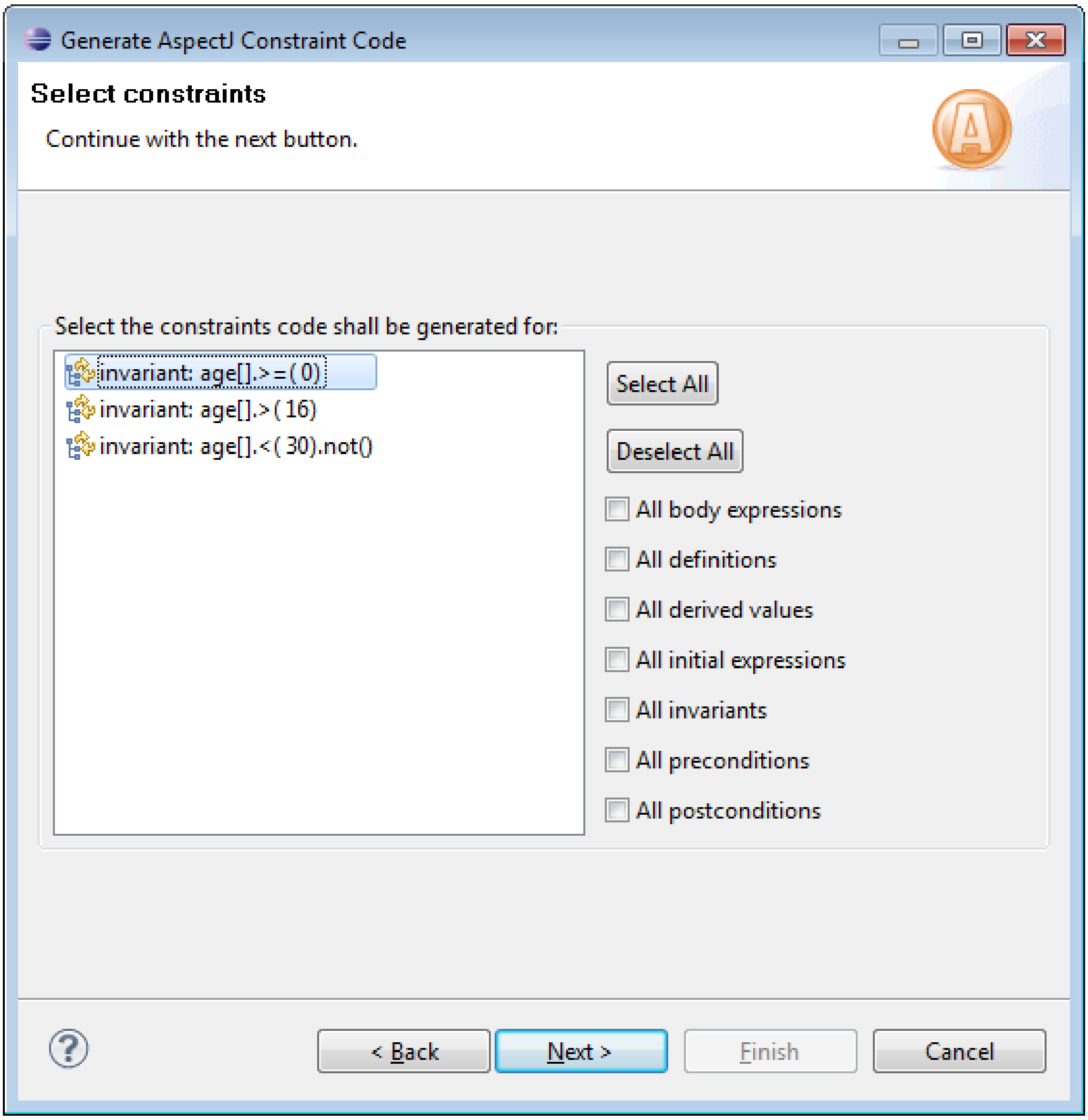

Figure 3.7: The second step: Selecting constraints for code generation.

ocl22javacode/src) (see Figure 3.8). Please note, that we select the source directory and not the package directory into which the code shall be generated! The code generator creates or uses contained package directories depending on the package structure of the selected constraint. Additionally we can specify a sub folder into that the constraint code shall be generated relatively to the package of the constrained class. By default this is a sub directory called `constraints`. We don't want to change this setting and push the *Next* button.

3.2.4 Specifying General Settings

On the following page of the wizard we can specify general settings for the code generation (see Figure 3.9). We can disable the inheritance of constraints (which would not be useful in our example because we want to enforce the constraint for `Persons`, but for `Students` and `Professors` as well). We can also enable that the code generator will generate getter methods for newly defined attributes of `def` constraints. More interesting is the possibility to select one of three provided strategies, when invariants shall be checked during runtime:

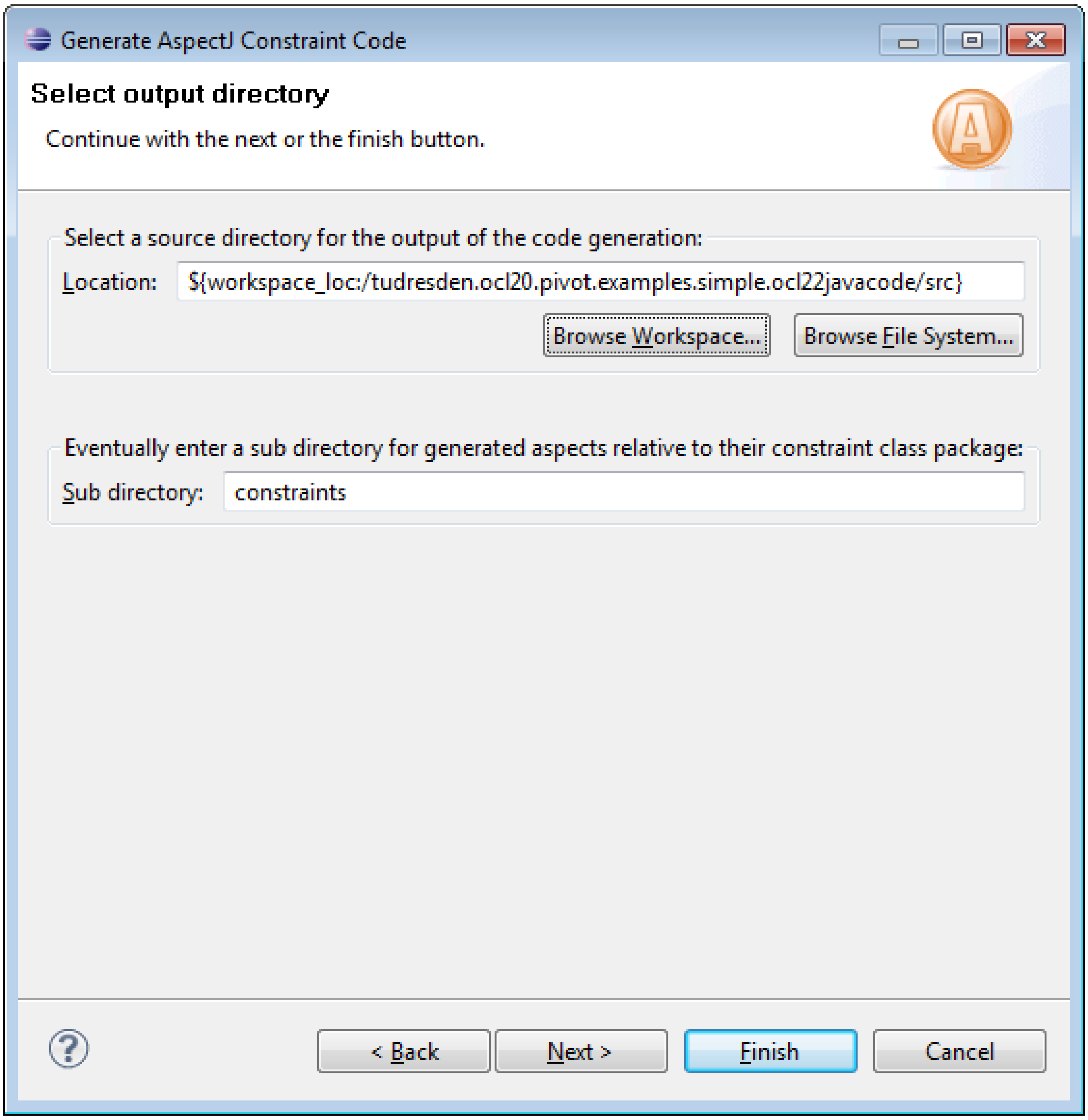

Figure 3.8: The third step: Selecting a target directory for the generated code.

1. Invariants can be checked after construction of an object and after any change of an attribute or association which is in scope of the invariant condition (*Strong Verification*).
2. Invariants can be checked after construction of an object and before or after the execution of any public method of the constrained class (*Weak Verification*).
3. And finally, invariants can only be checked if the user calls a special method at runtime (*Transactional Verification*).

These three scenarios can be useful for users in different situations. If a user wants to verify strongly, that his constraints are verified after any change of any dependent attribute he should use *Strong Verification*. If he wants to use attributes to temporary store values and constraints shall be verified if any external class instance wants to access values of the constrained class only, he should use *Weak Verification*. If the user wants to work with databases or other remote communication and the state of his constraint classes should be valid before data transmission only, he should use the strategy *Transactional Verification*.

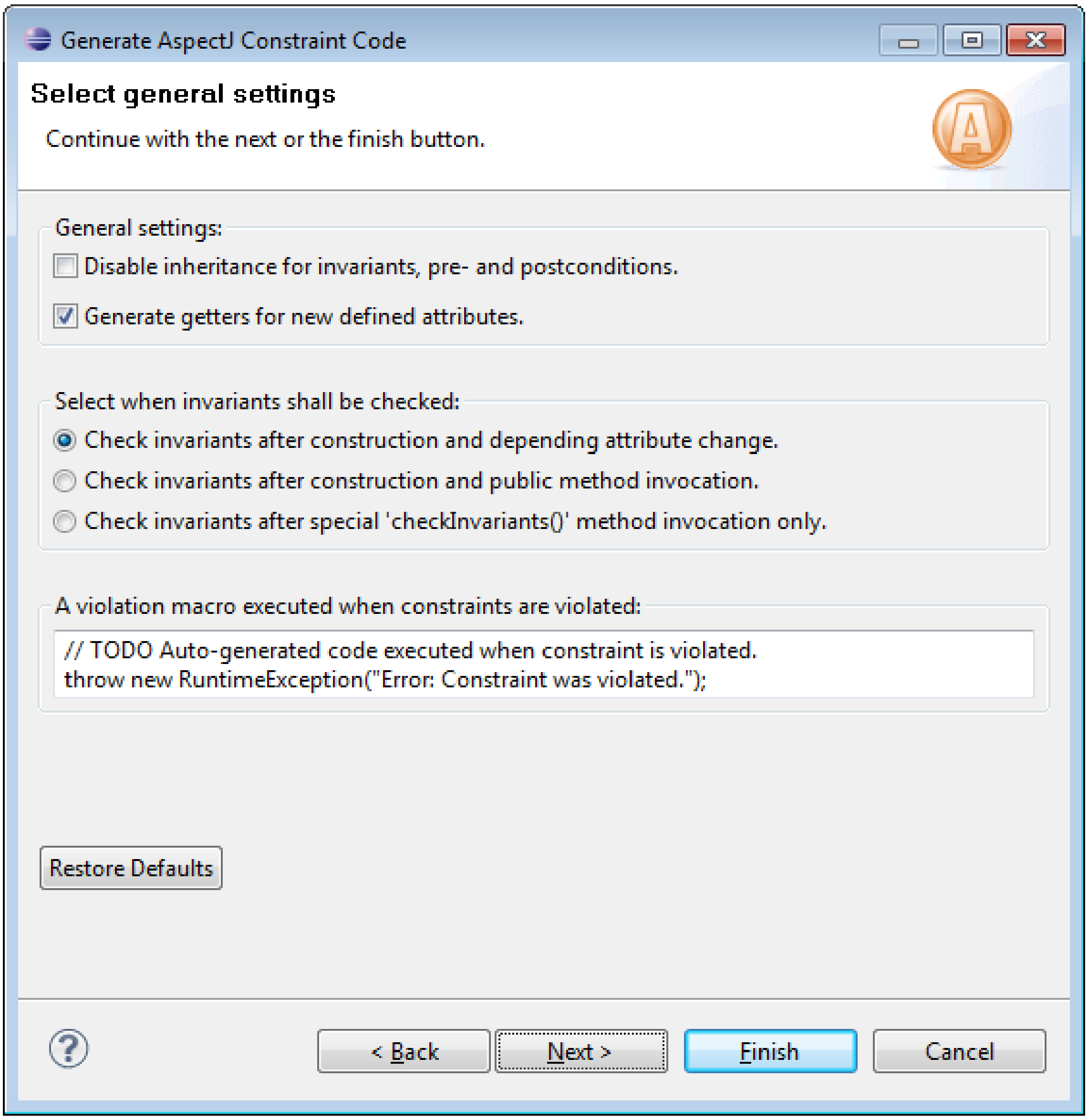

Figure 3.9: The fourth step: General settings for the code generation.

Finally, we can specify a *Violation Macro* which specifies the code, which will be executed when a constraint is violated during runtime. By default, the violation macro throws a run-time exception. We also want to have a run-time exception thrown when our constraint is violated. Thus, we do not change the violation macro and continue with the *Next* button.

3.2.5 Constraint-Specific Settings

The last page of the code generation wizard provides the possibility to configure some of the code generation settings constraint-specific by selecting a constraint and adapting it's settings (see Figure 3.10). We don't want to adapt the settings, thus we can finish the wizard and start the code generation by pushing the *Finish* button.

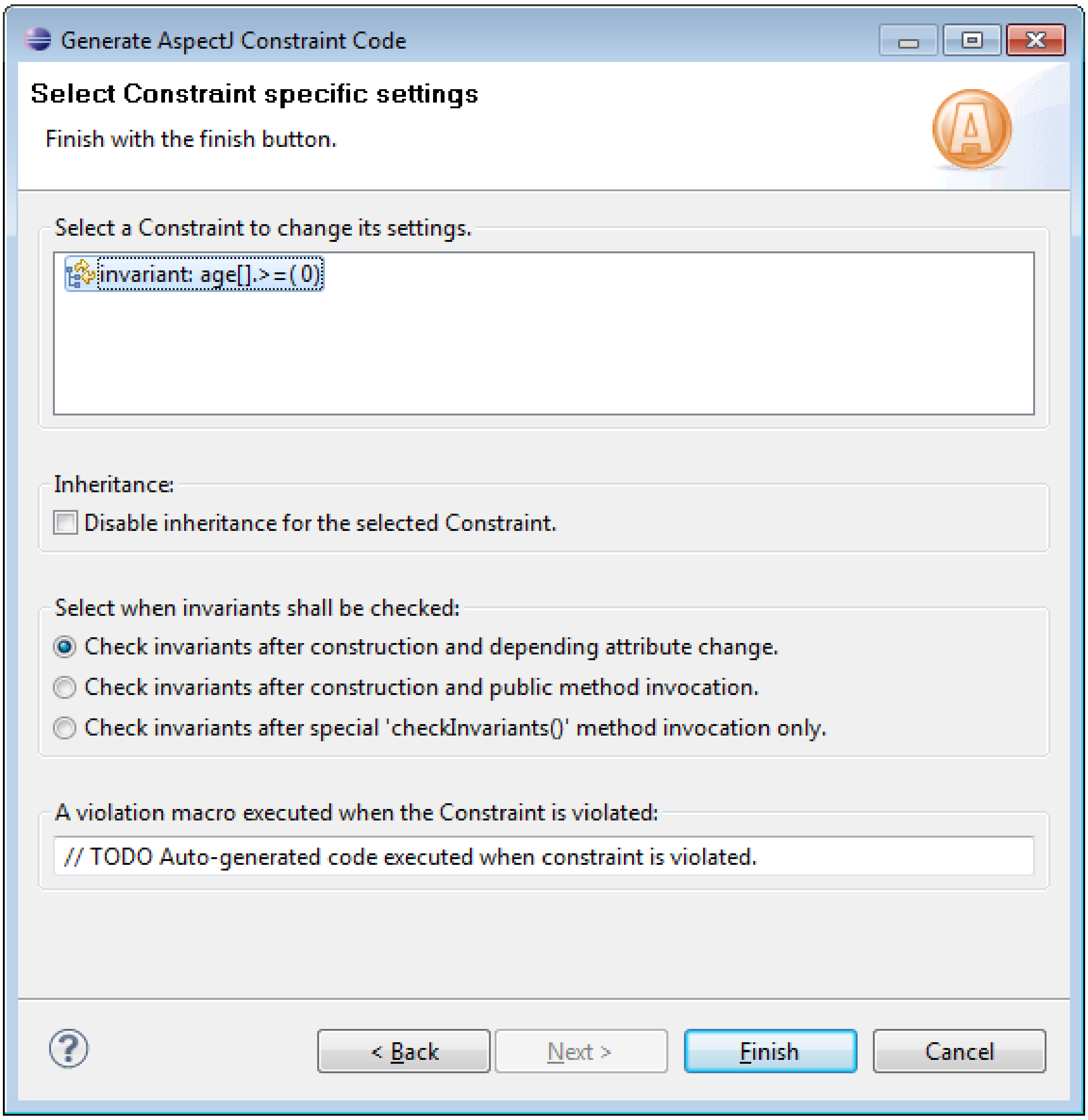

Figure 3.10: The fifth step: Constraint-specific settings for the code generation.

3.3 THE GENERATED CODE

After finishing the wizard, the code for the selected constraint will be generated. To see the result, we have to refresh our project in the workspace. We select the project `tudresden.ocl20.pivot.examples.simple.constraint` in the *Package Explorer*, open the context menu and select the menu item *Refresh*. Afterwards, our project contains a new generated AspectJ file called `tudresden.ocl20.pivot.examples.simple.constraints.InvAspect01.aj` (see Figure 3.11). Now we can rerun our JUnit test case. The test case finishes successfully because the expected runtime exception is thrown (see Figure 3.12).

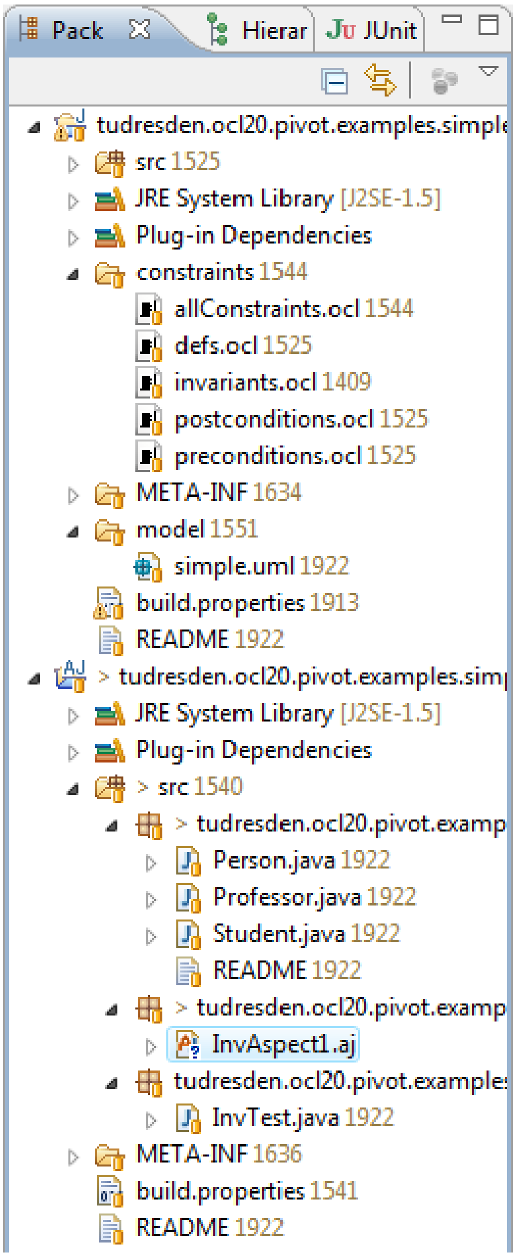

Figure 3.11: The package explorer containing the new generated aspect file.

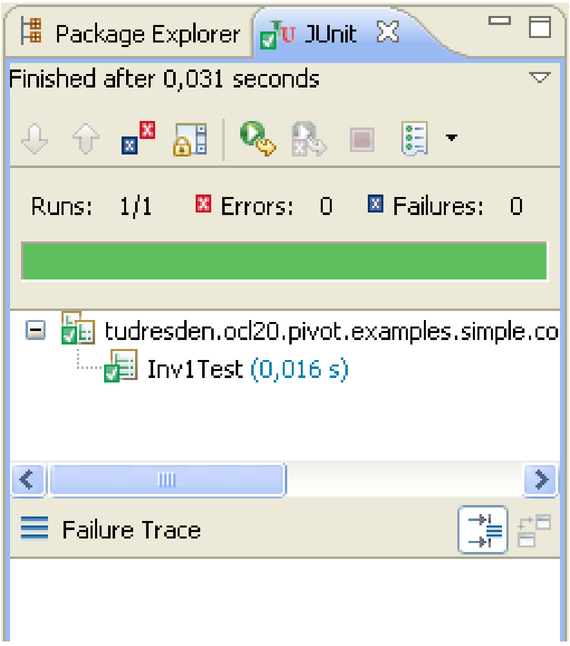

Figure 3.12: The successfully finished jUnit test case.

3.4 SUMMARY

This Chapter described how to generate AspectJ code using the *OCL22Java* code generator of Dresden OCL2 for Eclipse. A more detailed documentation of the OCL22Java code generator can be found in the Minor Thesis (Großer Beleg) of Claas Wilke [Wil09].

II DEVELOPMENT WITH DRESDEN OCL2 FOR ECLIPSE

4 THE ARCHITECTURE OF DRESDEN OCL2 FOR ECLIPSE

Chapter written by Claas Wilke

This chapter introduces into the highly generic architecture of Dresden OCL2 for Eclipse. Before the architecture is explained, some theoretical background is shortly presented.

4.1 THE GENERIC THREE LAYER METADATA ARCHITECTURE

The Object Constraint Language is a language that is always based on another modeling language (usually the UML). Without another language used for modeling, it does not make any sense to define constraints because OCL is used for constraint specification but not for modeling itself. Thus, besides OCL, a modeling language is required to define a model on that OCL constraints can be specified.

Each modeling language is defined in another language, its *Meta-Modeling Language*. For example, the Unified Modeling Language's meta-model is defined using the *Meta Object Facility (MOF)* [OMG06], the standardized meta-meta model of the OMG. The MOF is used to describe the UML meta-model that can be used to model UML models. Generally spoken, each model requires a meta-model that is used to describe the model. The model can be instantiated by model instances (for example a UML class diagram can be instantiated by a UML object diagram). The model can be enriched with OCL constraints that are defined on the model (using an OCL meta-model) and can then be verified for instances of the model afterwards.

The OMG introduced the *MOF Four Layer Metadata Architecture* [OMG06][OMG09, p. 16ff] that is used to arrange and structure the meta-model, the model, and the model's instances into a layered hierarchy (see Figure 4.1). Generally, four layers exist, the *Meta-Meta-Model Layer (M3)*, the *Meta-Model Layer (M2)*, the *Model Layer (M1)*, and the *Model Instance Layer (M0)*.

OCL constraints can be defined on both, meta-models and models to verify models or model instances, respectively. E.g., one may use OCL to define rules on a meta-model that must be ensured for every model modeled with the meta-model. But, one may use OCL to define rules on a model (that must be verified for the model's instances) as well. Thus, the four layer metadata

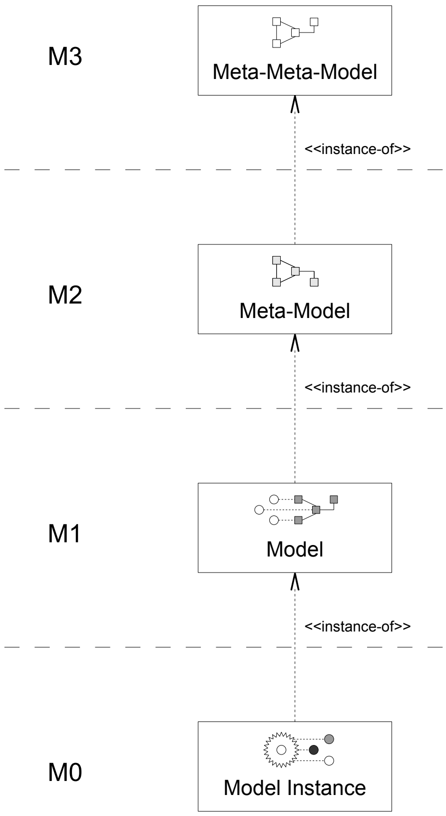

Figure 4.1: The MOF Four Layer Metadata Architecture.

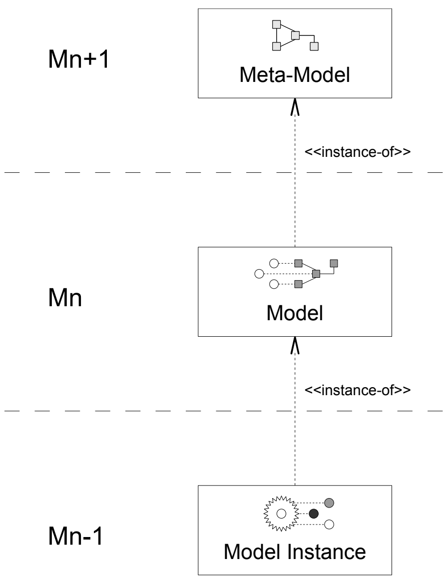

Figure 4.2: The Generic Three Layer Metadata Architecture.

architecture can be generalized to a *Generic Three Layer Metadata Architecture* in the scope of an OCL definition (see Figure 4.2) [DW09]. On the *Mn+1 Layer* lies the meta-model that is used to define the model that shall be constrained. On the *Mn Layer* lies the model that is an instance of the meta-model and can be enriched by the specification of OCL constraints. Finally, on the *Mn-1 Layer* lies the model instance on that the OCL constraints shall be verified. Please note, that in the context of such a generic layer architecture, a model instance can be both a model (like a UML class diagram) or a set of objects (like Java run-time objects).

4.2 THE TOOLKIT'S PACKAGE ARCHITECTURE

The package architecture of Dresden OCL2 for Eclipse is shown in Figure 4.3. The architecture is the result of the work of Matthias Bräuer [Brä07] and can easily be extended. The architecture can be separated into three layers: The *Back-End*, the *Core* and the *Tools Layer*.

The back-end layer contains the meta-model to manage models and run-time objects (or values) that shall be used during interpretation as model instances. Both, meta-models and model instances can easily be exchanged because all other packages of Dresden OCL2 for Eclipse do not directly communicate with the meta-model and the run-time objects but use the *Pivot Model* and the *Model Instance Type Model* that delegate all requests instead. Important is the fact, that both meta-models and model instances can be exchanged independently. Thus, a Java model instance could be both an instance of a UML class diagram and an EMF Ecore model.

The second layer is the toolkit's core layer and contains the *Pivot Model*, *Essential OCL*, the *Model Instance Type Model*, the *OCL Standard Library* and the *Model Bus*. The use of the pivot model

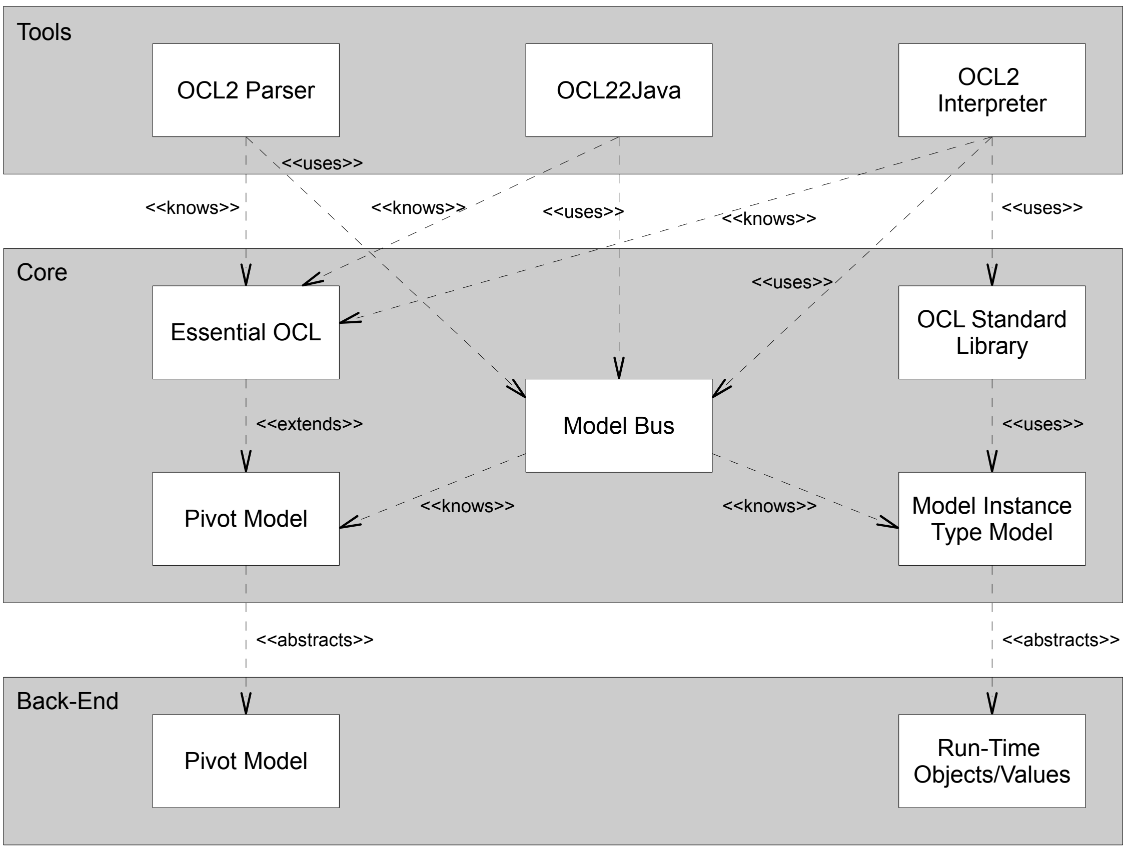

Figure 4.3: The architecture of Dresden OCL2 for Eclipse.

and the model instance type model were explained before. The package *Essential OCL* extends the pivot model and provides the *Abstract Syntax* to extend loaded models with OCL constraints. The OCL standard library provides an implementation of all core operations that exist in OCL for specific datatypes (e.g., collection iterators like `forAll()` or the operation `oclIsTypeOf()`). The package *Model Bus* loads, manages and provides access to models and model instances the user wants to work with and thus, can be considered as the repository of the toolkit.

The third layer contains all tools that are provided with the toolkit. This layer contains the *OCL2 Parser* (which is essential, because the other tools require models that are enriched with already parsed and syntactically and semantically checked constraints), the *OCL2 Interpreter*, and the *OCL22Java Code Generator*. All the tools use the packages of the second layer to access models and model instances and to work on OCL constraints. The OCL standard library, and also managed model instances are only required by the OCL2 interpreter.

Dresden OCL2 for Eclipse has been developed as a set of Eclipse/OSGi plug-ins. All packages that are located in the core and tools layer represent different Eclipse plug-ins. Additionally, Dresden OCL2 for Eclipse contains some plug-ins to provide GUI elements such as wizards and examples to run Dresden OCL2 for Eclipse with some simple models and OCL expressions.

4.3 DRESDEN OCL2 FOR ECLIPSE AND THE GENERIC THREE LAYER METADATA ARCHITECTURE

Figure 4.4 shows the architecture of Dresden OCL2 for Eclipse in respect to the Generic Three Layer Metadata Architecture (introduced in Section 4.1). At the first sight, the architecture seems to be very complex. But do not be afraid! The architecture will now be explained step by step.

4.3.1 The Adaptation of Meta-Models, Models and Model Instances

As you can see, the left part of Figure 4.4 shows the Generic Three Layer Metadata Architecture. Meta-models, models and model instances are adapted and loaded into Dresden OCL2 for Eclipse. It could be argued that such an adaptation is expensive and costly, but the opposite is the truth. The architecture of Dresden OCL2 for Eclipse allows its users to adapt the toolkit to every meta-model and model instance type they want. After the adaptation of a new meta-model or model-instance, they can reuse the rest of the toolkit! Thus, to adapt the OCL2 Interpreter to a new type of model instance, only one adaptation is required. The rest comes for free! How to adapt meta-models and model instance types to Dresden OCL2 for Eclipse is explained in the Chapters 6 and 7.

4.3.2 How Meta-Models and Models are Adapted

As already said, the core feature of Dresden OCL2 for Eclipse is the *Pivot Model*. The pivot model is a meta-model that abstracts from all other meta-models. It contains interfaces to define the structural part of a model such as `Types`, `Namespaces`, `Operations` and `Properties`. Furthermore, these interfaces provide methods to reason on them (e.g., the interface `Namespace` provides a method `getNestedNamespaces()` to retrieve all contained `Namespaces`).

Every meta-model users want to work with can be adapted to the pivot model. The *Adapted Meta-Model* must implement the interfaces of the pivot model and must adapt them to its meta-model elements. E.g., adapting the UML2 meta-model, the interface `Type` from the pivot model

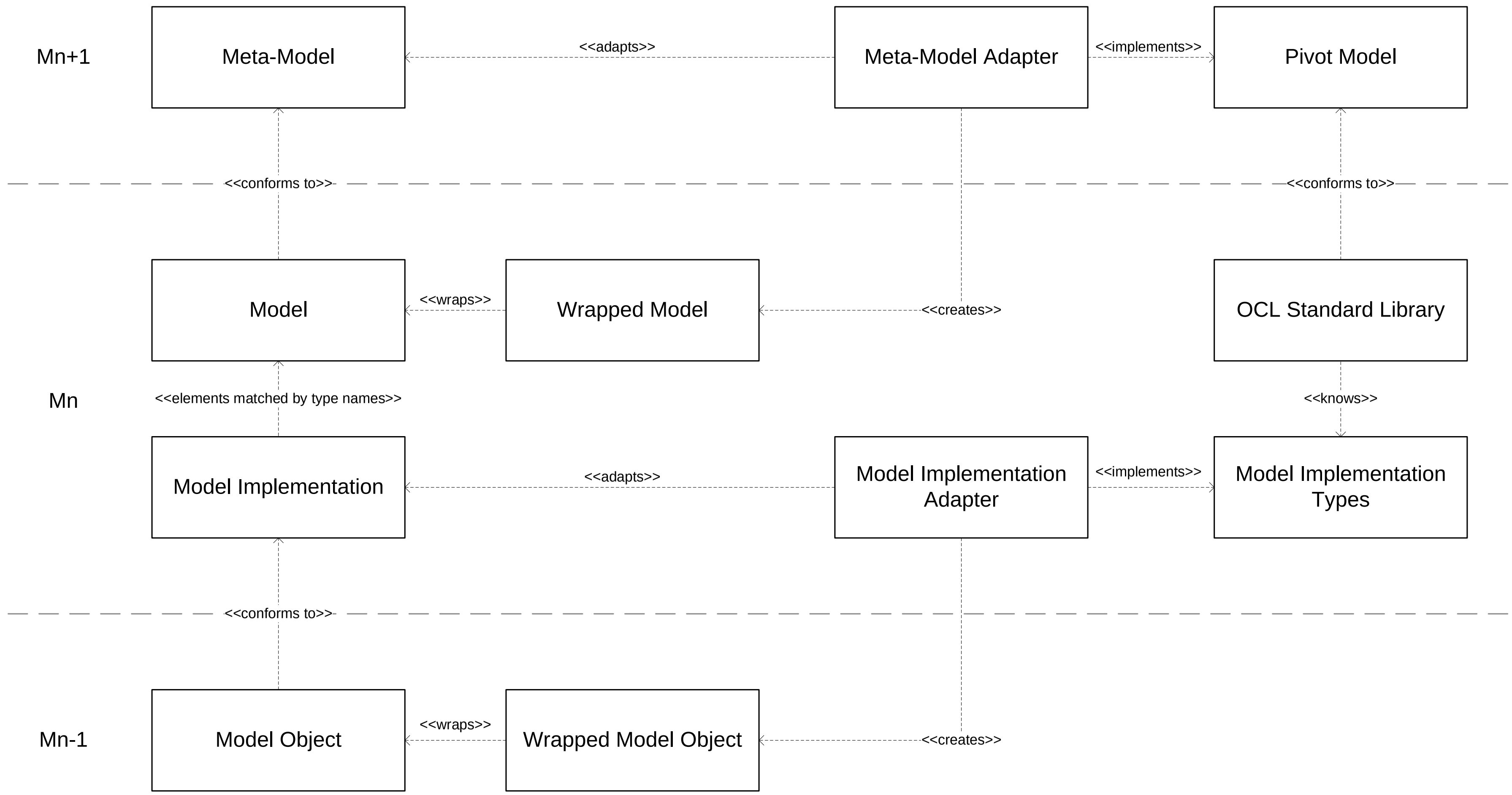

Figure 4.4: The architecture of Dresden OCL2 for Eclipse in respect to the Generic Three Layer Metadata Architecture.

must be adapted to the meta-model element `UML2Class`. Besides the adaptation of the pivot model, each meta-model must provide a `ModelProvider` that provides methods to load model resources of the adapted meta-model and adapts them to its pivot model implementation. The result is a *Wrapped Model*, Dresden OCL2 for Eclipse can work with. Further details about the adaptation of meta-models to the pivot model can be found in Chapter 6.

4.3.3 How Model Instances are Adapted

If the OCL2 Interpreter shall be used to interpret instances of a loaded model, a second adaptation is required, an adaptation to the *Model Instance Type Model*. The model instance type model can be considered as similar to the pivot model, but the purpose is different. If OCL constraints shall be interpreted on run-time objects or values, operations and properties of these run-times values must be accessible. E.g., if a constraint like `context Person inv: age >= 0` shall be interpreted, the property `age` of a run-time object of the class `Person` must be accessed. The OCL2 interpreter does not accesses this property directly, but delegates the request to a model instance type adaptation to let the model instance type become exchangeable[1]. Thus, Dresden OCL2 for Eclipse contains a second model the *Model Implementation Type Model* that can be considered as an abstraction of all model instance types. The model instance type model defines a set of interfaces to describe the elements of a model instance (e.g., `IModelInstancePrimitiveType` and `IModelInstanceObject`). These interfaces provide operations to reflect on the model instance objects (e.g., the operation `IModelInstanceType.invokeOperation()` or the operation `IModelInstanceElement.isTypeOf()`). The reflection mechanism can be considered as similar to the mechanism provided by Java in the package `java.lang.reflect`.

Each model instance type users want to work with must be adapted to the model instance type model. Besides an adaptation of the interfaces, also a `ModelInstanceProvider` must be implemented that is responsible to adapt model instance objects to the implemented interfaces. Due to the fact of this second adaptation, Dresden OCL2 for Eclipse is able to use the same OCL2 Interpreter for different types of model instances! More details about the adaptation of model instances to Dresden OCL2 for Eclipse are available in Chapter 7.

4.3.4 Coupling between Models and their Instances

As mentioned above, different types of models can be connected with different types of instances. E.g., a UML class diagram could be implemented by a set of Java Classes (and their objects) or by an XML document. To maintain this loose coupling, meta-models and model implementation types do not know each other. If a model instance is imported into Dresden OCL2 for Eclipse, a model (and thus also a meta-model) has to be selected, to that the instance belongs to. The objects of the instance are matched to the types of the selected model by the name of their types. E.g., a Java instance's objects are matched by associating their classes' names to the names of the types of the selected model.

4.3.5 Essential OCL and OCL Constraints

To enrich models loaded into Dresden OCL2 for Eclipse with OCL constraints, a meta-model for OCL constraints is required (also called the *Abstract Syntax*). This meta-model contains elements like `OperationCallExpressions` or `StringLiterals` to describe the different tokens of OCL constraints and to enrich loaded models with parsed OCL constraints. *Essential OCL* is this OCL meta-model that is used by the OCL2 Parser to build the model representation of parsed OCL constraints. Essential OCL extends the pivot model, because OCL constraints can also contain `Types`, `Operations` and `Properties` defined in the model.

[1] To be honest, the interpreter does not delegate the access directly but uses the OCL standard library instead which delegates the request to the model instance. But this is a technical detail and not important in this context.

4.3.6 The OCL Standard Library

The *OCL Standard Library* is required when OCL constraints shall be interpreted. The OCL standard library is the implementation of all operations that are defined in the OCL standard and are available for all types in OCL or for different kinds of types (e.g., the operations `String.concat()` or `OclAny.oclIsUndefined()`. The OCL standard library is invoked by the OCL2 Interpreter at any time, the interpreter wants to interpret the value of an `OperationCallExpression` or a `PropertyCallExpression`. The OCL standard library either computes the result itself or delegates the request to an adapted model instance (via the *Model Implementation Type Model*'s interfaces) if a model-specific operation or property shall be accessed.

4.4 SUMMARY

This Chapter introduced into the architecture and package structure of Dresden OCL2 for Eclipse. The *Pivot Model* and the *Model Implementation Type Model* have been explained shortly. Also the relationships between the pivot model, *Essential OCL* and the *OCL Standard Library* have been presented. One may argue that the architecture seems to be complex and complicate. Nevertheless, it should be remembered that Dresden OCL2 for Eclipse was designed as generic as possible. Thus, Dresden OCL2 for Eclipse can be adapted to various different kinds of meta-models and model instances without changing the OCL2 Parser nor the OCL2 Interpreter!

5 HOW TO INTEGRATE DRESDEN OCL2 FOR ECLIPSE

Chapter written by Claas Wilke

In Chapter 4 the architecture of Dresden OCL2 for Eclipse has shortly been explained. This chapter will explain, how Dresden OCL2 for Eclipse can be integrated into other tools, toolkits or projects.

5.1 THE INTEGRATION FACADE OF DRESDEN OCL2 FOR ECLIPSE

Since the release 2.1.0, Dresden OCL2 for Eclipse contains an *Integration Facade*, that combines all required interfaces of Dresden OCL2 for Eclipse in one interface, also called a *Facade* [GHJV95]. The facade contains self-explanatory static methods that provide access to the repository (modelbus) and all tools of Dresden OCL2 for Eclipse. A documentation of the complete facade's interface would be too large for this documentation. Thus, please investigate the facade directly in the code. The facade called `Ocl2ForEclipseFacade` is located in the plug-in `tudresden.ocl20.pivot.facade`. Please be aware of the fact that if you use the facade, you will result in dependencies to all major parts of Dresden OCL2 for Eclipse. Thus, if you want to use one of the tools only (e.g, the OCL2 Parser) you could access these tools directly as explained below.

5.2 HOW TO ACCESS META-MODELS, MODELS AND INSTANCES

The central component of Dresden OCL2 for Eclipse is the *Model-Bus* which is implemented by the Eclipse plug-in `tudresden.ocl20.pivot.modelbus`. The main class of this plug-in (`tudresden.ocl20.pivot.modelbus.ModelBusPlugin`) provides methos, to access meta-models, models and model instances and to import new resources of these kinds into the toolkit (see Figure 5.1).

The class provides four different static methods to access different registries, the `MetamodelRegistry`, the `ModelRegistry`, the `ModelInstanceTypeRegistry` and the `ModelInstanceRegistry`.

5.2.1 The Meta-Model Registry

The Meta-Model Registry provides methods to add and get meta-models to and from Dresden OCL2 for Eclipse. Normally, the method `addMetamodel(IMetamodel)` is not required because by starting Eclipse, all meta-models register themselves via their extension point in the registry. To get a meta-model from the toolkit, the methods `getMetamodels()` and `getMetamodel(id: String)` can be used. The method `getMetamodels()` returns all meta-models that are currently registered in the registry. The method `getMetamodel(id: String)` can be used to get a meta-model by its ID (Normally, the ID of a meta-model is equal to the name of its plug-in. E.g., the UML2 meta-model has the ID `tudresden.ocl20.pivot.metamodels.uml2`).

5.2.2 How to load a Model

First, to load a model into Dresden OCL2 for Eclipse, the meta-model the model is an instance of has to be selected . E.g., for an UML2 class diagram the UML2 meta-model should be selected (see Listing 5.1). Each meta-model has its own `IModelProvider` that can be accessed by using the method `IMetamodel.getModelProvider()`. The `IModelProvider` provides three methods to load a model. A model can be loaded by using the method `getModel(..)` with

1. A `File` object representing the model as argument,
2. a `String` representing the path of the file there the model is located,
3. or a `URL` leading to the file there the model is located.

After loading the model, the model can be added to the `ModelRegistry`, that manages all models currently loaded into Dresden OCL2 for Eclipse (see Figure 5.3). The `ModelRegistry` can also be used to set an active model which represents the `IModel` that is currently selected in the *Model Browser* of Dresden OCL2 for Eclipse.

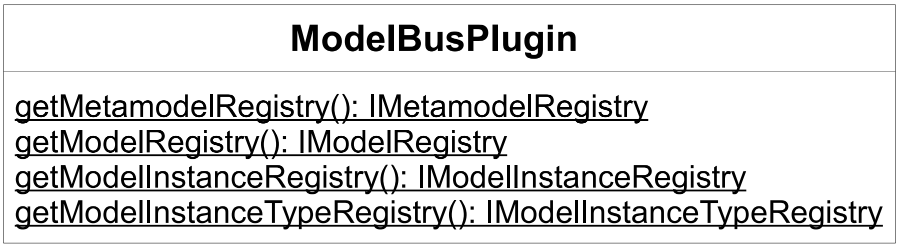

Figure 5.1: The main class of the Model-Bus plug-in.

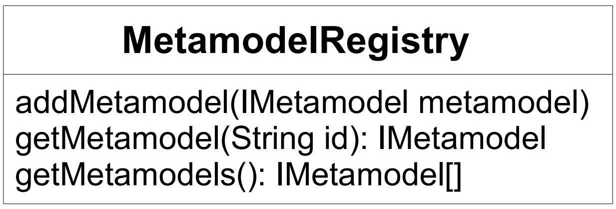

Figure 5.2: The Meta-Model Registry.

```
IMetamodel metaModel;
IModel model;

metaModel = ModelBusPlugin.getMetamodelRegistry()
                .getMetamodel("tudresden.ocl20.pivot.metamodels.uml2");
model = metaModel.getModelProvider().getModel(modelURL);
```

Listing 5.1: How to load a model.

5.2.3 The Model Instance Type Registry

Similar to the Meta-Model Registry, the Model Instance Type Registry provides methods to add and get model instance types to and from Dresden OCL2 for Eclipse. Normally, the method `addModelInstanceType(IModelInstanceType)` is not required because by starting Eclipse, all model instance types register themselves via their extension point in the registry. To get a model instance type from the toolkit, the methods `getModelInstanceTypes()` and `getModelInstanceType(id: String)` can be used. The method `getModelInstanceTypes()` returns all model instance types that are currently registered in the registry. The method `getModelInstanceType(id: String)` can be used to get a model instance types by its ID (Normally, the ID of a model instance type is equal to the name of its plug-in. E.g., the Java model instance type has the ID `tudresden.ocl20.pivot.modelinstancetype.java`).

5.2.4 How to load a Model Instance

First, to load a model instance into Dresden OCL2 for Eclipse, the model instance type must be selected the instance is an instance of. E.g., for a set of Java objects the Java model instance type should be selected (see Listing 5.2). Each model instance type has its own `IModelInstanceProvider` that can be accessed by using the method `IModelInstanceType.getModelInstanceProvider`. The `IModelInstanceProvider` provides three methods to load a model instance. A model instance can be loaded by using the method `getModelInstance(..)` with

1. A `File` object representing the model instance as argument,

ModelRegistry
addModel(IModel model) setActiveModel(IModel model) getActiveModel(): IModel getModels(): IModel[]

Figure 5.3: The Model Registry.

ModelInstanceTypeRegistry
addModelInstanceType(IModelInstanceType miType) getModelInstanceType(String id): IModelInstanceType getModelInstanceTypes(): IModelInstanceTypes[]

Figure 5.4: The Model Instance Type Registry.

```
IModelInstanceType miType;
IModelInstance modelInstance;

miType = ModelBusPlugin.getModelInstanceTypeRegistry()
           .getModelInstanceType(
             "tudresden.ocl20.pivot.modelinstancetype.java");
modelInstance = miType.getModelInstanceProvider()
                  .getModelInstance(modelInstanceUrl, model);
```

Listing 5.2: How to load a model instance.

2. a `String` representing the path of the file there the model instance is located,
3. or a `URL` leading to the file there the model instance is located.

Additionally, each of these methods requires the `IModel` as a second argument, the model instance is an instance of. Thus, the model must be loaded before the model instance can be loaded.

After loading the model instance, the model instance can be added to the `ModelInstanceRegistry`, that manages all model instances currently loaded into Dresden OCL2 for Eclipse (see Figure 5.5). The `ModelInstanceRegistry` can also be used to set an active model instance that represents the `IModelInstance` that is currently selected in the *Model Instance Browser* of Dresden OCL2 for Eclipse.

5.3 HOW TO ACCESS THE OCL2 PARSER

The OCL2 Parser of Dresden OCL2 for Eclipse is located in the plug-in `tudresden.ocl20.pivot.ocl2Parser`. The parser provides a very simple interface and can be used as shown in Listing 5.3. First, a new `OCL2Parser` must be created. The constructor requires two arguments, a `Reader` used to read all constraints that shall be parsed, and an `IModel` for that the constraints shall be parsed. After creating the parser instance, the method `OCL2Parser.parse()` can be used to parse the constraints.

5.4 HOW TO ACCESS THE OCL2 INTERPRETER

To use the OCL2 Interpreter, a model and a model instance must be loaded into the toolkit before. Additionally, at least one constraint must be parsed that shall be interpreted for the objects contained in the model instance. The interpreter is located in the plug-in `tudresden.ocl20.pivot.`

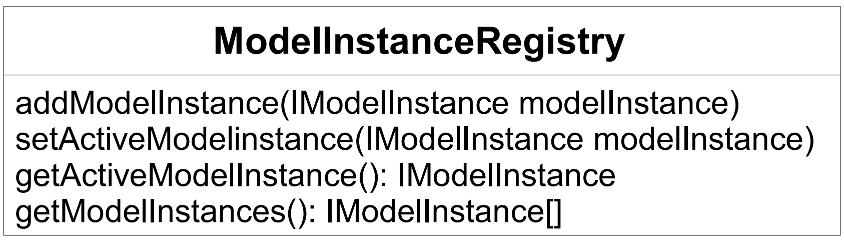

Figure 5.5: The Model Instance Registry.

`interpreter`. It has a more complex interface than the other tools and contains many different operations to interpret different kinds of constraints.

Listing 5.4 shows how the `OclIntepreter` can be used. First, a model and a model instance must be loaded on which the constraints shall be verified. Furthermore, at least one constraint must be parsed that shall be interpreted (lines 1-6). Afterwards, an `IOclInterpreter` can be created for the loaded model instance by using the factory method of the `OclInterpreterPlugin` (line 14). Finnaly, the parsed constraints can be interpreted for all `IModelInstanceObjects` of the model instance by iterating over them (lines 21-24). The result of each interpretation will be an `IInterpretationResult` which internally contains a triple of (1) an `IModelInstanceObject` on which (2) a `Constraint` have been interpreted resulting in (3) an `OclAny`.

5.5 SUMMARY

This chapter shortly presented, how the tools of Dresden OCL2 for Eclipse can be accessed and used via their interfaces. First, the integration facade has been presented, afterwards, direct access of specific tools and the modelbus has been explained. Please be aware of the fact, that direct code documentation is always error-prone and will be outdated very soon. Thus, please do not hesitate to contact us if some parts of this chapter are written in a unclear manner or are inconsistent with the code.

```
FileReader oclFileReader;
OCL2Parser parser;

oclFileReader = new FileReader(oclFile);
parser = new OCL2Parser(model, oclFileReader);
parser.parse();
```

Listing 5.3: How to parse constraints.

```
1  IModel model;
2  IModelInstance modelInstance;
3
4  /*
5   * Load model, model instance and constraints. ...
6   */
7
8  IOclInterpreter oclInterpreter;
9
10 List<Constraint> constraints;
11 List<IModelInstanceObject> modelInstanceObjects;
12 List<IInterpretationResult> results;
13
14 oclInterpreter = OclInterpreterPlugin.createInterpreter(modelInstance);
15
16 constraints = model.getRootNamespace().getOwnedAndNestedRules();
17 modelInstanceObjects = modelInstance.getAllModelInstanceObjects();
18
19 results = new ArrayList<IInterpretationResult>();
20
21 for (IModelInstanceObject aModelInstanceObject : modelInstanceObjects) {
22   results.addAll(oclInterpreter.interpretConstraints(constraints,
23          aModelInstanceObject));
24 }
```

Listing 5.4: How to interpret constraints.

6 ADAPTING A META-MODEL TO THE PIVOT MODEL

Chapter written by Michael Thiele

Dresden OCL2 for Eclipse is built to work with different meta-models / DSLs. In order to use new meta-models one has to create an adapter plug-in that adapts the meta-model to the pivot model of the toolkit. To ease this process, Dresden OCL2 for Eclipse includes a code generator that creates adapter stubs.

The code generator is located in the plug-in `tudresden.ocl20.pivot.codegen.adapter`. The only prerequisites are the core features (`tudresden.ocl20.feature.core`) of the Dresden OCL2 for Eclipse and the meta-model to adapt that has to be modeled in Ecore.

In the following Ecore itself will be adapted to the pivot model (this has been done already for Dresden OCL2 for Eclipse, but serves the purpose of showing the adaption mechanism on a well known meta-model). Figure 6.1 shows the Eclipse standard editor for Ecore models with the Ecore model opened. Since the adaptation of different repositories is allowed, one has to specify the resource from which a model can be loaded. This can be done via an annotation as shown in figure 6.2. In the *Properties View* enter `http://www.tu-dresden.de/ocl20/pivot/2007/pivot-model` at `Source` (see figure 6.3). Create a new details entry for the annotation (figure 6.4) and enter `Resource` as `Key` and `org.eclipse.emf.ecore.resource.Resource` as `Value` (see figure 6.5).

To adapt types of the meta-model to the pivot model, choose the type to adapt and create a new annotation (similar to resource) and a corresponding details entry with `PivotModel` as `Key` and the specific pivot model type name as `Value`. In figure 6.6 the meta-element `EClass` is adapted to the pivot model type `Type`. This step has to be repeated for every meta-model type that can be adapted to the pivot model. In this example this would be:

- `EClass` –>`Type`
- `EDataType` –>`PrimitiveType`
- `EEnum` –>`Enumeration`
- `EEnumLiteral` –>`EnumerationLiteral`

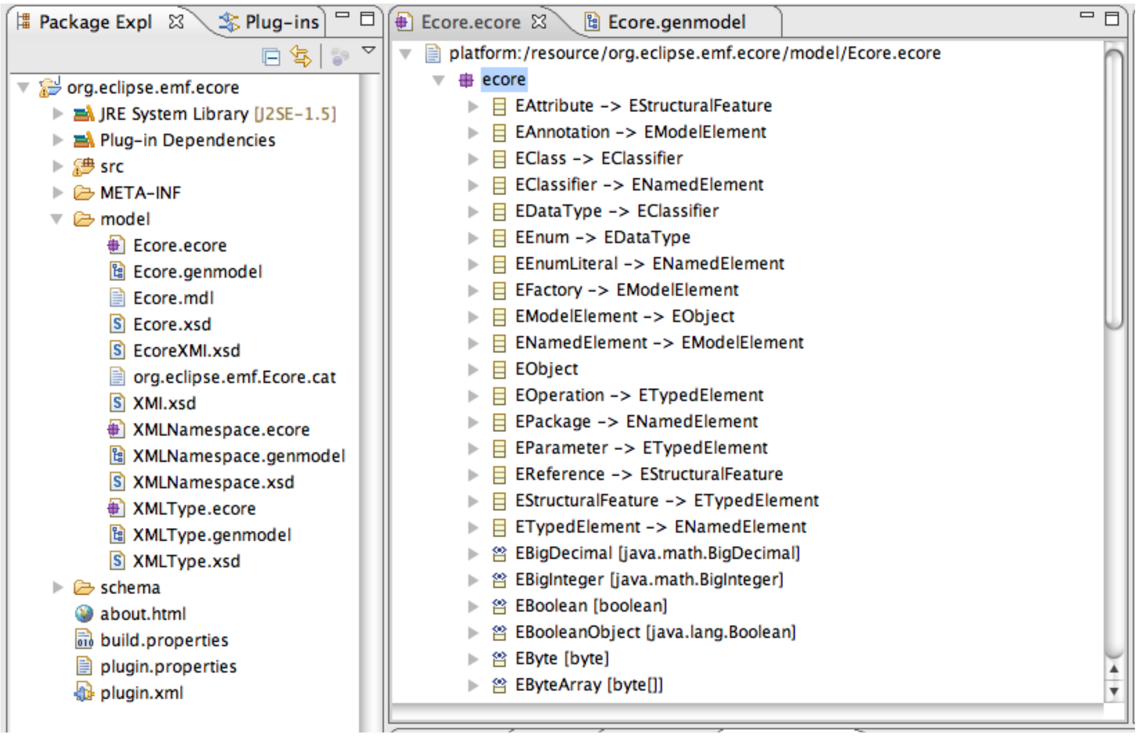

Figure 6.1: The Ecore model opened in Eclipse.

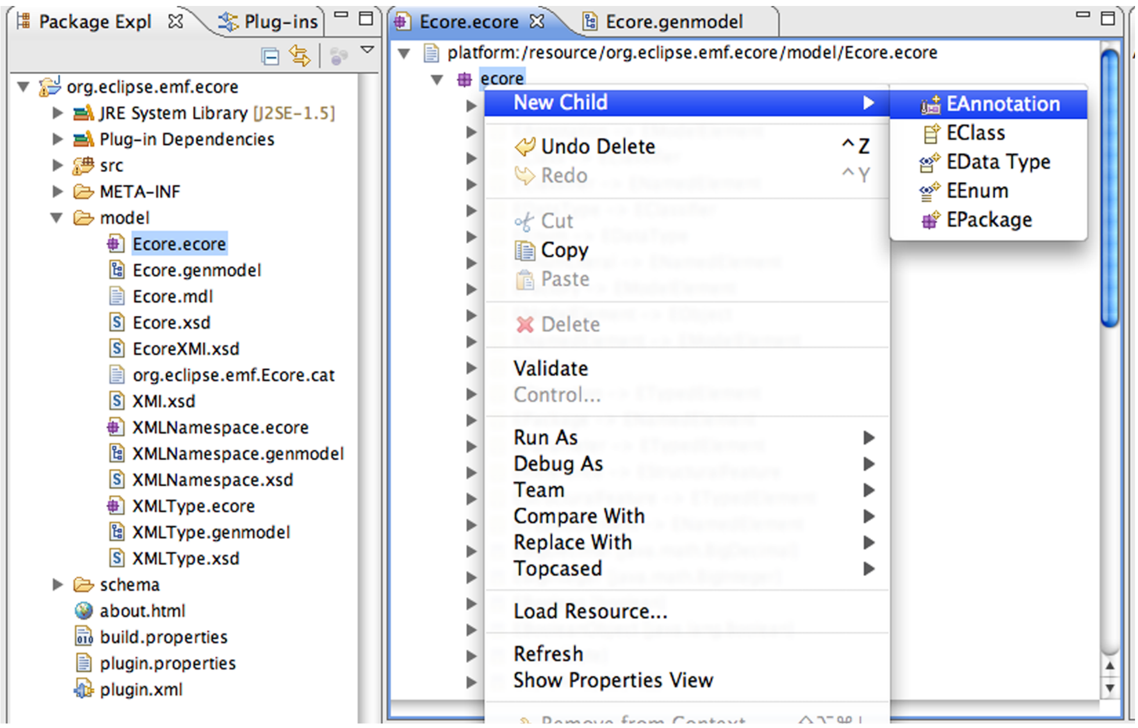

Figure 6.2: Create an annotation for the Ecore package.

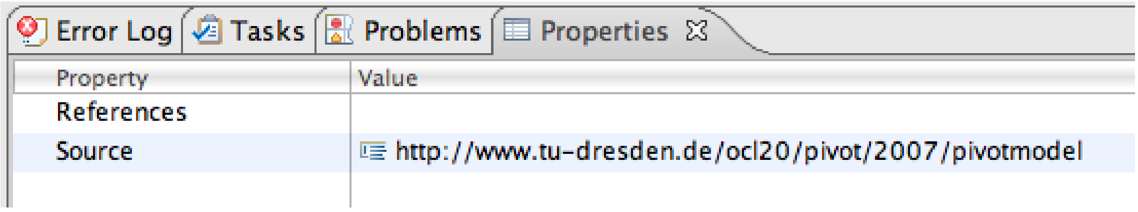

Figure 6.3: The Properties View for the annotation.

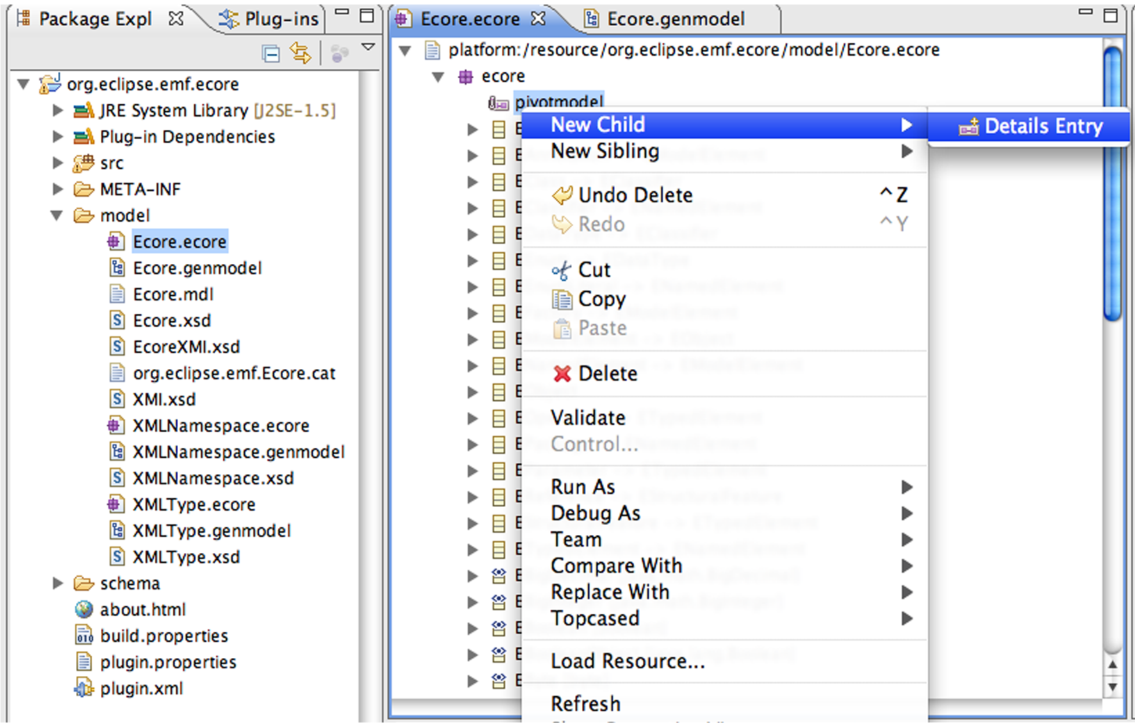

Figure 6.4: Create annotation details for the annotation.

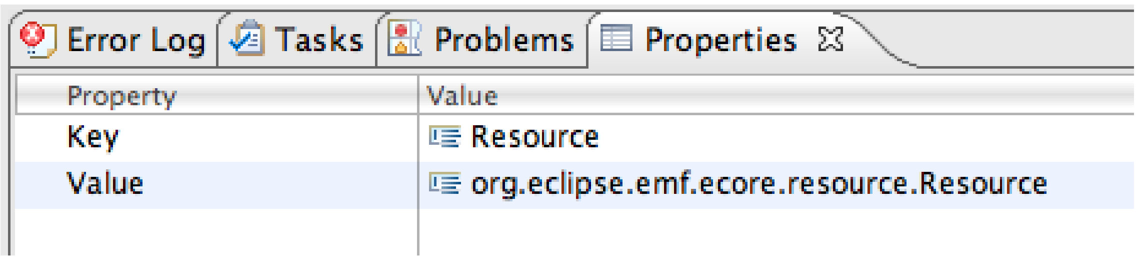

Figure 6.5: The Properties View for the annotation details.

- `EOperation` ->`Operation`
- `EPackage` ->`Namespace`
- `EParameter` ->`Parameter`
- `EStructuralFeature` ->`Property`

When finished, change to the genmodel (see figure 6.7). Right-click on the root element and choose `Generate Pivot Model adapters` like shown in figure 6.8. A new plug-in is created. It is named after the conventions of the Dresden OCL2 for Eclipse: `tudresden.ocl20.pivot.metamodels.<meta-model name>`. The package explorer is shown in figure 6.9.

Every created class marked blue (`http://www.eclipse.org/modeling/emft/?project=mint/`) has methods with an `@generated` annotation. The locations where to alter the generated code are also marked with `TODO`s. After replacing the `TODO`s with real code, the adapter is complete and can be used together with the Dresden OCL2 for Eclipse. To test the new adapter, read chapter 10 on the meta-model test suite that allows test-driven development of the adapter.

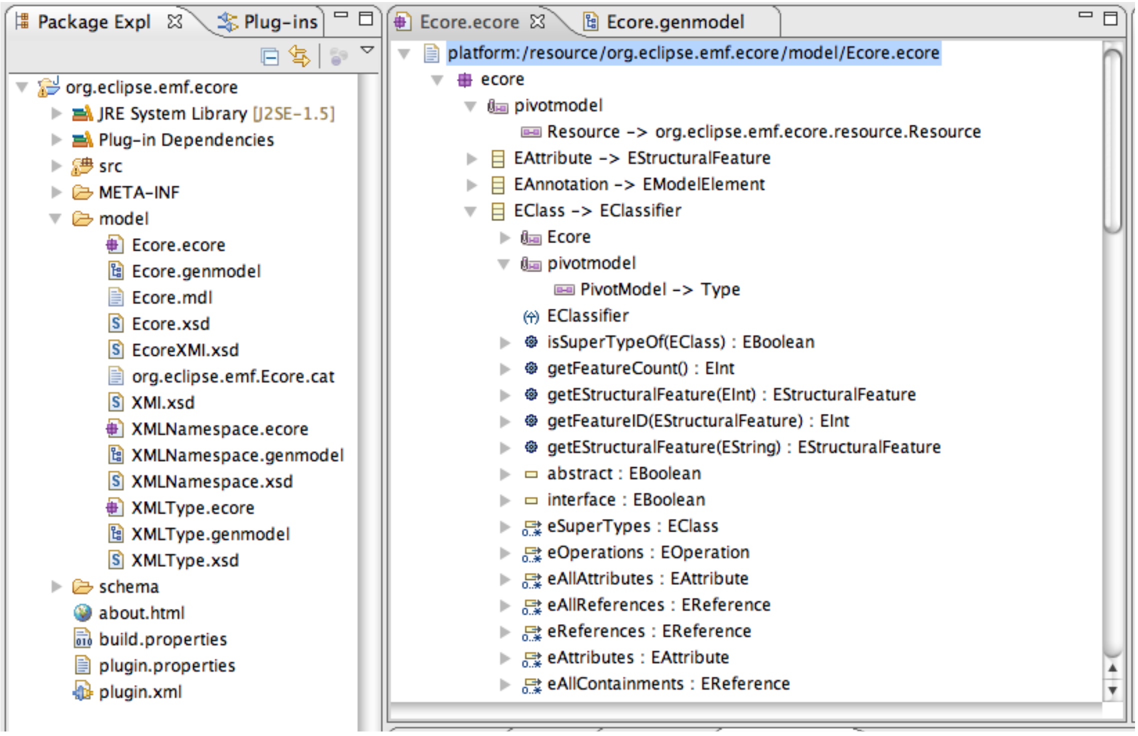

Figure 6.6: The EClass annotation.

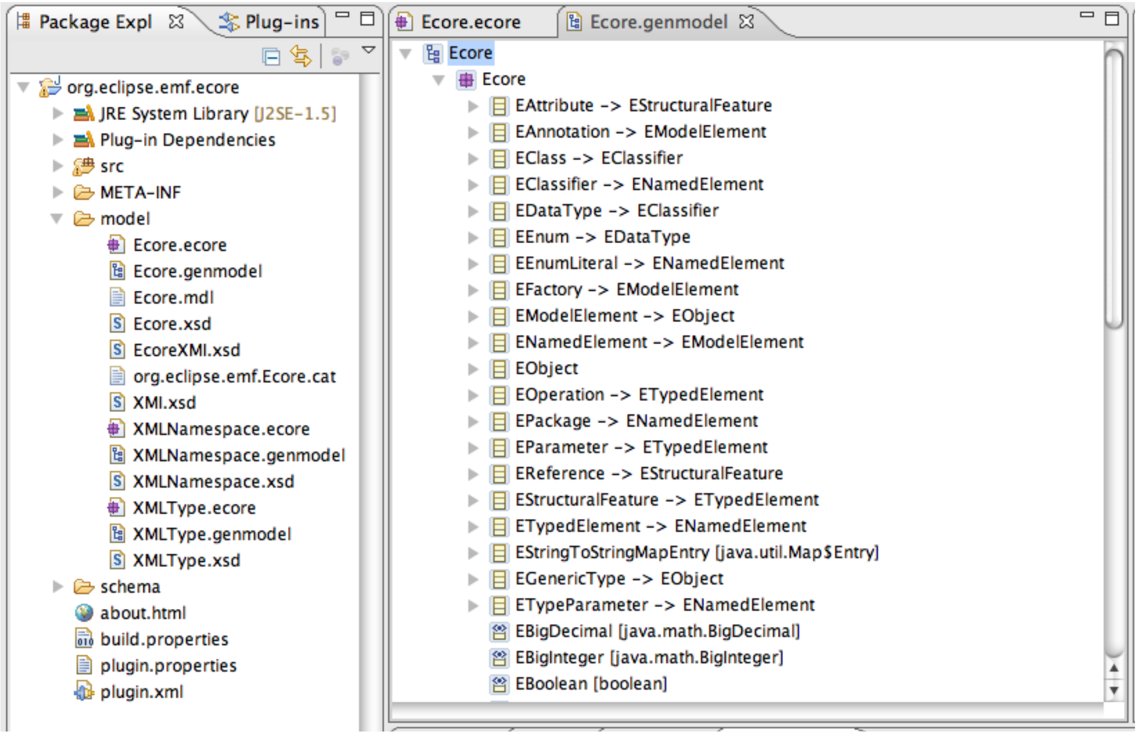

Figure 6.7: The genmodel of the Ecore model.

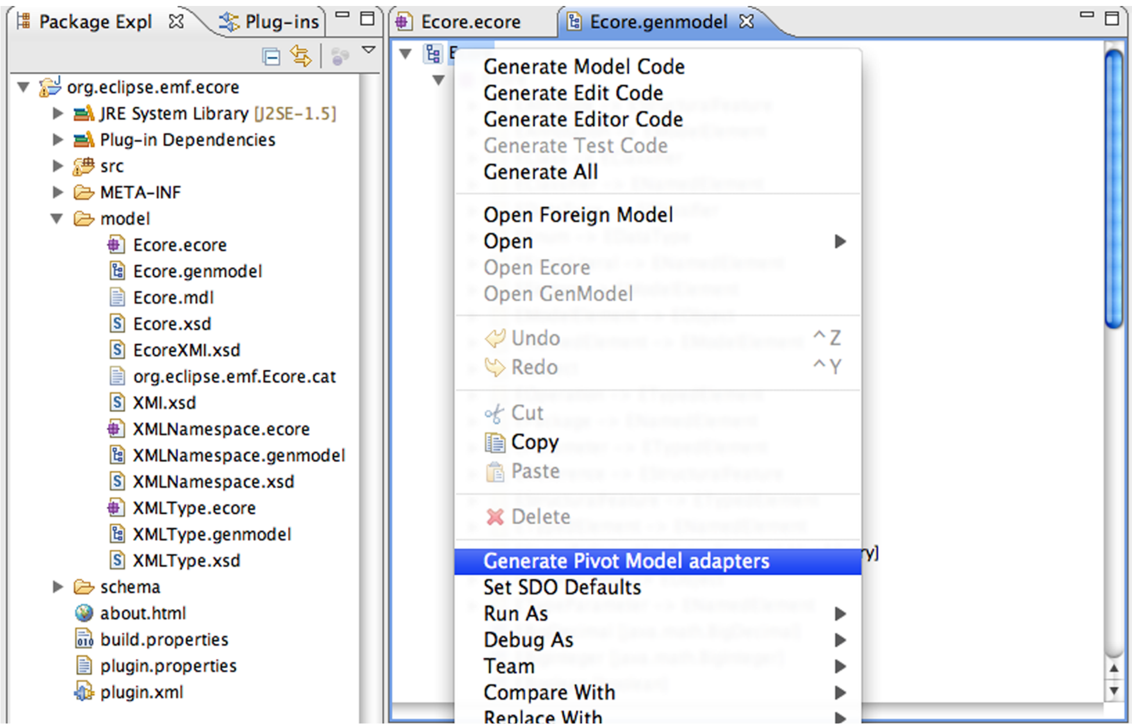

Figure 6.8: Right-click on Ecore and select 'Generate Pivot Model adapters'.

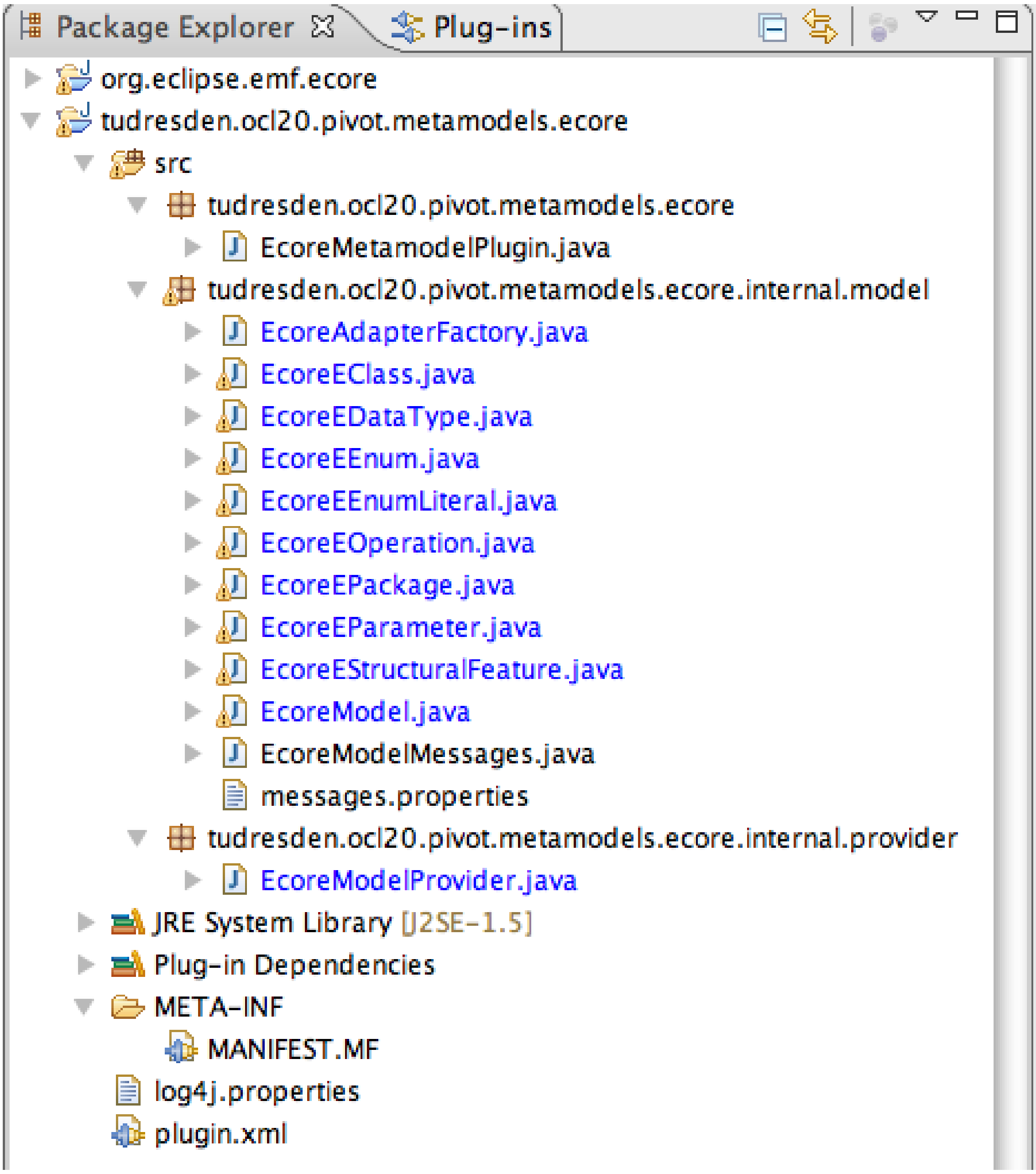

Figure 6.9: The structure of the generated plug-in.

7 ADAPTING A MODEL INSTANCE TYPE TO DRESDEN OCL2 FOR ECLIPSE

Chapter written by Claas Wilke

As mentioned in Chapter 4, Dresden OCL2 for Eclipse is able to interpret OCL constraints on different types of model instances. E.g., the same constraints can be interpreted on Java `Objects`, EMF `EObjects` and XML files. This is possible because the toolkit abstracts the instance's elements as `IModelInstanceElements`. Thus, each type of model instance that shall be connected with the OCL2 Interpreter requires its own Model Instance Type Adaptation. How such an adaptation has to be implemented is explained below. First, the different elements that can belong to a model instance type are presented. Afterwards, the `IModelInstanceProvider`, `IModelInstance` and `IModelInstanceFactory` interfaces are explained.

7.1 THE DIFFERENT TYPES OF MODEL INSTANCE ELEMENTS

Similar to a model, a model instance can have different types of elements. The element types are similar to the different types that can be expressed in models adapted to Dresden OCL2 for Eclipse. Figure 7.1 shows all different types of `IModelInstanceElements` that can exist. The different types are explained in the following.

7.1.1 The IModelInstanceElement Interface

Each `IModelInstanceElement` has to provide a set of methods that is required to handle the adapted objects during interpretation. The methods are shortly explained in the following. Some of these methods are implemented in an abstract `IModelInstanceElement` implementation and have not to be implemented. Nevertheless, they are presented for completeness reasons.

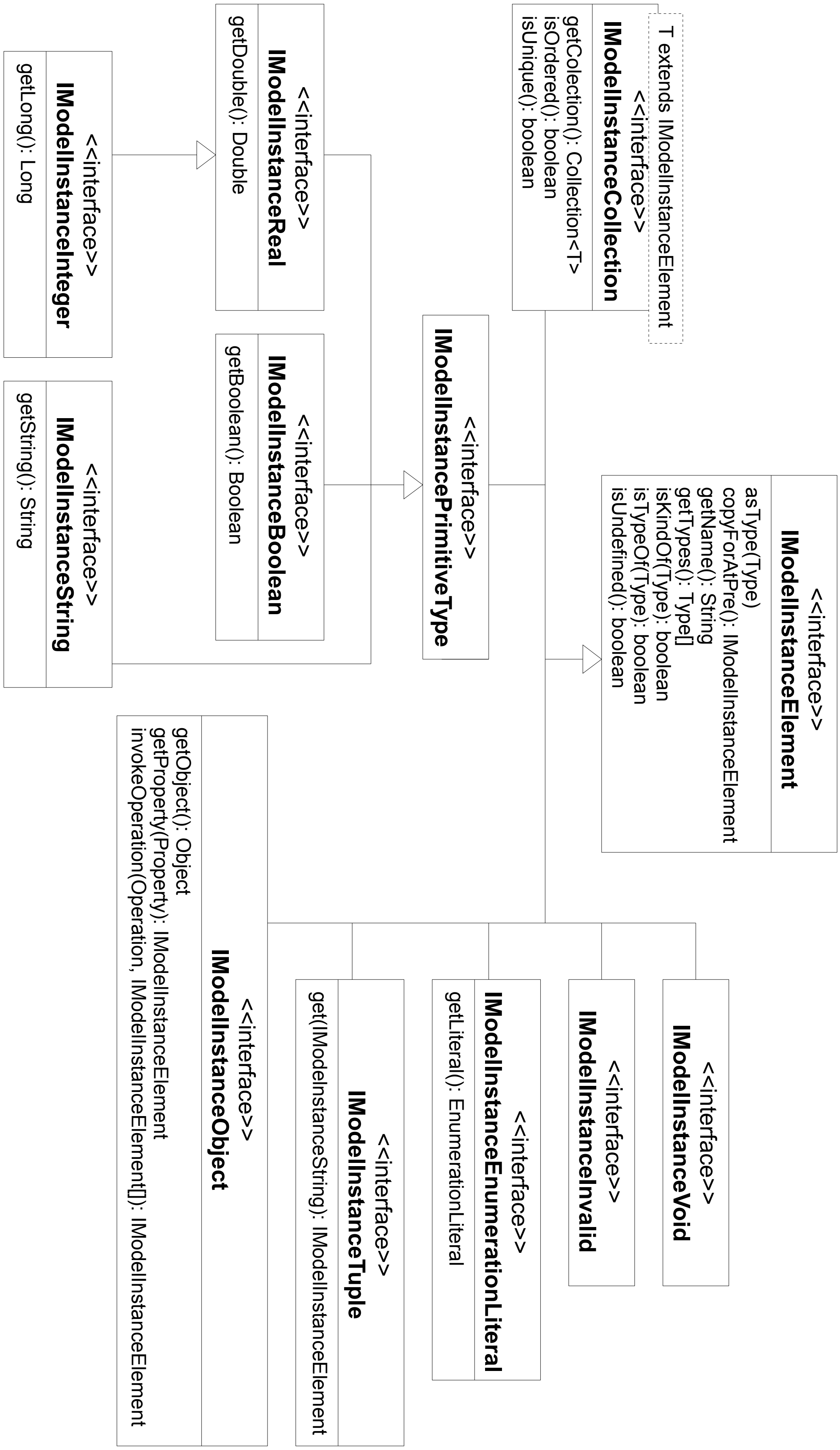

Figure 7.1: The different types of IModelInstanceElements.

asType(Type)

The method `asType(Type)` is required to cast an element to a given type of its model. E.g., in OCL the primitive type `Integer` can be casted to `Real`. In general, this method should check if the given Type conforms to the adapted element and if so, the result is a new `IModelInstanceElement` of the given type. Else an `AsTypeCastException` is thrown.

copyForAtPre()

The method `copyForAtPre()` is required to create a copy of the element if its value shall be stored during interpretation as an *@pre-value*. E.g., during the interpretation of the constraint

```
context Person::birthdayHappens()
post:  self.age = self.age@pre + 1
```

the interpreter has to store the value of the property `age`. The value has to be copied, because if `age` is incremented during the method's execution, a simple reference would refer to the incremented value[1]. As far as we know, it is rather complicate to copy some objects at runtime - e.g., in Java where the `clone()` method is not available for every object. Thus, a `CopyForAtPreException` can be thrown, if an element cannot be copied.

getName()

The method `getName()` returns a string representation of the element. This additional operation exists to provide a different string to display the element in the GUI components besides the general `toString()` method's result.

getTypes()

The method `getTypes()` returns a set containing all `Types` of the element. In most cases, this set contains exactly one Type. But sometimes, this set can contain multiplye types. This is the case if an `IModelInstanceElement` represents an object that inherits from multiple types in the model.

isKindOf(Type) and isTypeOf(Type)

The methods `isKindof(Type)` and `isTypeOf(Type)` are required to check if an `IModelInstance-Element` conforms to a given type or is of a given type, respectively.

isUndefined()

The method `isUndefined()` checks if an `IModelInstanceElement`'s adapted element is `null` or not.

[1] This is true although age is a primitive type. The integer instance is modeled in Java and thus can be referenced.

7.1.2 The Adaptation of Model Instance Objects

The most important IModelInstanceElement is the IModelInstanceObject. It encapsulates the standard objects of an model instance such as a Java Object or an EMF Ecore EObject. IModelInstanceObject must be implemented by every model instance type because without this kind of element a model instance type does not do any sense. Besides the inherited methods of IModelInstanceElement three additional methods have to be implemented. They are explained below.

getObject()

The method getObject() returns the adapted Object of the IModelInstanceObject.

getProperty(Property)

The method getProperty(Property) is required to get the property values of an object during the interpretation of OCL constraints. The method should return the adapted value of the given property (probably an IModelInstanceCollection of values if the property is multiple or the instance of IModelInstanceVoid if the property's value is null) or throws a PropertyNotFoundException if the given property does not exist. A PropertyAccessException can be thrown, if an unexpected exception occurs during accessing the object's property.

invokeOperation(Operation, List<IModelInstanceElement>)

The method invokeOperation(Operation, List<IModelInstanceElement>) is required to invoke the adapted object's operations during interpretation. The list of arguments (adapted as IModelInstanceElements) may be empty but not null. The method should return the adapted value of the operation's invocation (probably an IModelInstanceCollection of values if the operation is multiple or the instance of IModelInstanceVoid if the result is null) or throws an OperationNotFoundException if the given operation does not exist. A OperationAccessException can be thrown, if an unexpected exception occurs during invoking the object's operation. This operation is one of the most complicated operations in the complete model instance implementation because it must be able to reconvert adapted model instance elements given as parameters. E.g., a given IModelInstanceObject has to be unwrapped by calling its getObject() method. For primitive and collection type implementation instances this is more complicate because they do not contain an adapted object that can be returned. They have to be converted. For details of such a reconvert mechanism investigate the Java implementation that uses *Java Reflections* to check, whether an operation requires an int, Integer, byte, or Long (for example) as input.

7.1.3 The Adaptation of Primitive Type Instances

To adapt primitive type instances, the interfaces IModelInstanceBoolean, IModelInstanceInteger, IModelInstanceReal and IModelInstanceString exist. Each of them contains an additional method to return the adapted value as a Java Object[2]. Because primitive instances do not have a state, they do not have to but can be implemented by a model instance type. Instead of implementing own primitive type instances, the predefined instances JavaModelInstanceBoolean, JavaModelInstanceInteger, JavaModelInstanceReal and JavaModelInstanceString (located in the plug-in tudresden.ocl20.pivot.modelbus can be reused.

[2] Precisely, a Boolean, a Long, a Double or a String.

7.1.4 The Adaptation of Collections

Besides primitive type instances and objects, a collection implementation is required to describe sets of `IModelInstanceElements`. The interface `IModelInstanceCollection<T extends IModelInstanceElement>` provides three additional methods explained below. The `IModelInstanceCollection` must not be implemented by every model instance type, the predefined implementation `JavaModelInstanceCollection` can be reused instead.

getCollection()

The method `getCollection()` returns a Java collection containing the `IModelInstanceElments` that are contained in the `IModelInstanceCollection`.

isMultiple() and isOrdered()

The methods `isMultiple()` and `isOrdered()` identify the different types of OCL collections.

7.1.5 IModelInstanceEnumerationLiteral

The interface `IModelInstanceEnumerationLiteral` represents instances of `Enumerations`. Because enumerations do not have a state, they do not need any adapted `Object`. Thus, a standard `ModelInstanceEnumerationLiteral` implementation located in the plug-in `tudresden.ocl20.pivot.modelbus` can be reused. During adaptation, an enumeration literal existing in the model instance just has to be associated to its related `EnumerationLiteral` in the instance's model. For details investigate the existing `IModelInstance` implementations for Java and EMF Ecore.

7.1.6 IModelInstanceTuple

The interface `IModelInstanceTuple` represents key (`IModelInstanceString`) value `IModelInstanceElement` data structure called *Tuple*. Tuples are required during OCL interpretation only. Thus, a standard `ModelInstanceTuple` implementation located in the plug-in `tudresden.ocl20.pivot.modelbus` exists.

7.1.7 IModelInstanceVoid and IModelInstanceInvalid

The interfaces `IModelInstanceVoid` and `IModelInstanceInvalid` exist to define the singleton instances of the types `OclVoid` and `OclInvalid`. Their instances can be accessed via the static property `IModelInstanceVoid.INSTANCE` or `IModelInstanceInvalid.INSTANCE` respectively. For example, the `IModelInstanceVoid` instance is required when a method's invocation shall return a `null` value.

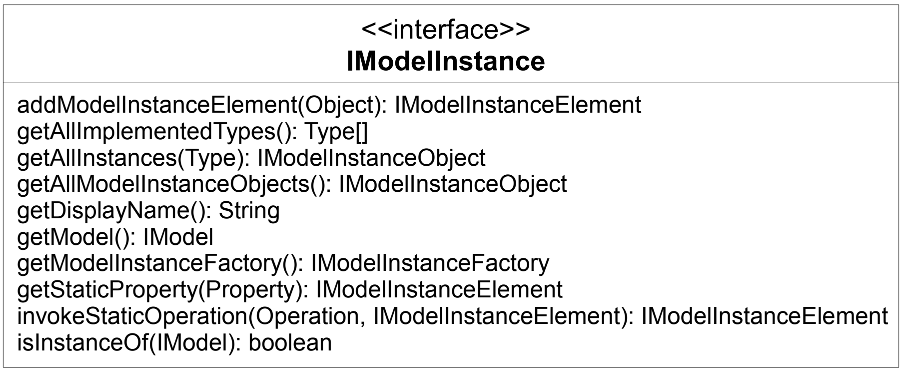

Figure 7.2: The IModelInstance Interface.

7.2 THE IMODELINSTANCEPROVIDER INTERFACE

Besides the `IModelInstaceElements`, a model instance type has to implement an `IModelInstanceProvider` that has to be registered at the model-bus plug-in via the extension point `tudresden.ocl20.pivot.modelbus.modelinstancetypes`. The model instance provider provides the methods to load a resource (given as a `URL` or `File`) into an `IModelInstance` object. You can use the abstract implementation `AbstractModelInstanceProvider` to implement your model instance provider. The two remaining methods to be implemented are explained below.

7.2.1 getModelInstance(URL, IModel)

The method `getModelInstance(URL, IModel)` is responsible to load a given model instance (as a `URL`) as an instance of a given model. For implementation details investigate the existing implementations for Java and EMF Ecore.

7.2.2 createEmptyModelInstance(IModel)

The method `createEmptyModelInstance(IModel)` can be used to create an empty model instance for a given model. The model instance can be enriched with objects during runtime via the method `IModelInstance.addModelInstanceElement(Object)`.

7.3 THE IMODELINSTANCE INTERFACE

Figure 7.2 shows the interface `IModelInstance`. Many of its operations are implemented in the abstract basis implementation `AbstractModelInstance`. The remaining operations that must be implemented are explained below.

7.3.1 The Constructor

The most important operation of a model instance is the constructor. Inside the constructor the resource given to the `IModelInstanceProvider` is opened and adapted to `IModelInstanceElements`. To adapt the elements, an `IModelInstanceFactory` (explained below) is used. For details investigate the existing `IModelInstance` implementations for Java and EMF Ecore.

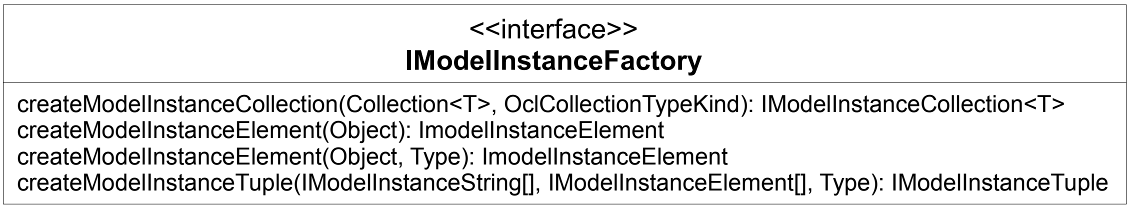

Figure 7.3: The IModelInstance Interface.

7.3.2 addModelInstanceElement(IModelInstanceElement)

The method `addModelInstanceElement(IModelInstanceElement)` can be used to add another `Object` to the model instance during runtime. The implementation should use its `IModelInstanceFactory` to adapt the `Object` and throw a `TypeNotFoundInModelException` if the given `Object` can not be adapted to the model instance.

7.3.3 getStaticProperty(Property)

The method `getStaticProperty(Property)` is required to get the property values of static properties during the interpretation of OCL constraints. The method should return the adapted value of the static property (probably an `IModelInstanceCollection` of values if the property is multiple or the instance of `IModelInstanceVoid` if the property's value is `null`) or throws a `PropertyNotFoundException` if the given property does not exist. A `PropertyAccessException` can be thrown, if an unexpected exception occurs during accessing the object's property.

7.3.4 invokeStaticOperation(Operation, List<IModelInstanceElement>)

The method `invokeStaticOperation(Operation, List<IModelInstanceElement>)` is required to invoke static operations during interpretation. The list of arguments (adapted as `IModelInstanceElements`) may be empty but not `null`. The method should return the adapted value of the operation's invocation (probably an `IModelInstanceCollection` of values if the operation is multiple or the instance of `IModelInstanceVoid` if the result is `null`) or throws an `OperationNotFoundException` if the given operation does not exist. An `OperationAccessException` can be thrown, if an unexpected exception occurs during invoking the object's operation. This operation is one of the most complicated operations in the complete model instance implementation because it must be able to reconvert adapted model instance elements given as parameters. E.g., a given `IModelInstanceObject` can be unwrapped by invoking its `getObject()` method. For primitive and collection type implementations this is more complicate because they do not contain an adapted object that can be simply returned. They have to be converted. For details of such a reconvert mechanism investigate the Java implementation that uses *Java Reflections* to check, whether an operation requires an `int`, `Integer`, `byte`, or `Long` (for example) as input.

7.4 THE IMODELINSTANCEFACTORY INTERFACE

The `IModelInstanceFactory` implementation of a model instance is responsible to adapted the instance's objects to the `IModelInstanceElement` implementations. It investigates the types of the objects to decide if they shall be adapted as `IModelInstanceObjects`, `IModelInstancePrimitiveTypes` or `IModelInstanceEnumerationLiterals`. An `IModelInstanceFactory` has to implement four methods as shown in Figure 7.3. A default implementation for the basis elements such as `IModelInstanceTuples` and `IModelInstanceCollections` called `BasisJavaModelInstanceFactory` exists. Normally, an `IModelInstanceFactory` should extend the `BasisJavaModelInstanceFactory` and should call the methods of the basis implementation as often as possible (e.g., to adapt the `IModelInstanceTuples`). For details investigate the existing implementations for EMF Ecore and Java.

7.5 ADAPTING AN OWN MODEL INSTANCE TYPE

We know that adapting a model instance type sounds easy but can be a lot of pain. Every kind of model instance comes with its own problems and own solutions. Some may be simple, others may be complicate or impossible. But never forget, if you adapted your own type of model instance, you can connect your instances with Dresden OCL2 for Eclipse and you can reuse the OCL2 Parser and OCL2 Interpreter!

If you are confused and still do not know how to adapt your model instance type, investigate the existing adaptations for Java (`tudresden.ocl20.pivot.modelinstancetype.java`) and EMF Ecore (`tudresden.ocl20.pivot.modelinstancetype.ecore`). To check your adaptation, have a look at the *Generic Model Instance Type Test Suite* (presented in Chapter 11) as well.

8 THE LOGGING MECHANISM OF DRESDEN OCL2 FOR ECLIPSE

Chapter written by Claas Wilke

Dresden OCL2 for Eclipse uses a *Log4j Logger* to log method entries, exits and errors during the toolkit's execution. If you run Dresden OCL2 for Eclipse as source code plug-ins from an Eclipse workspace, you might receive exceptions like following, although the toolkit works correctly.

```
log4j:ERROR Could not connect to remote log4j server at [localhost].
```

The reason is that the Log4j Logger tries to sent the logged events to a server running at `localhost`. To solve this problem (if you want to) you have to install and setup a logging server at your computer. One logging server you might use is called *Chainsaw* and available at the Apache Logging website [URL10a]. If you start Chainsaw, set up a *SocketReceiver* at port *4445 (Old Style/Standard Chainsaw Port)* (see figure 8.1).

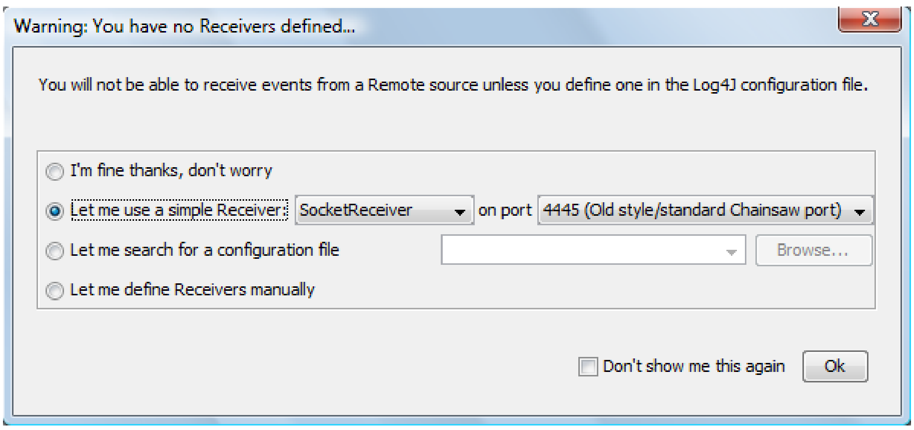

Figure 8.1: Setting up a simple SocketReceiver in Chainsaw.

9 THE EXTENSIBLE TEST SUITE OF DRESDEN OCL2 FOR ECLIPSE

Chapter written by Michael Thiele

Dresden OCL2 for Eclipse is a collection of Eclipse plug-ins that either are used together or some of these plug-ins are used together with plug-ins of other parties. This modular structure imposes one problem when trying to combine all test cases of Dresden OCL2 for Eclipse into one test suite, since there is no guaranty that all test plug-ins are available.

Therefore, an extensible test suite has been created. It can be found under the name `tudresden.ocl20.pivot.testsuite`. Basically it is a test suite that searches a specific extension point for registered tests or test suites. It includes these tests in its own test suite and executes the test suite. Thus, on any change of the source code all tests can be run at once to check the integrity of the toolkit.

To extend the extensible test suite with new tests or test suites, a plug-in has to implement the extension point `tudresden.ocl20.pivot.testsuite`. This extension has to specify the tests it wants to add. JUnit3 as well as JUnit4 tests or test suites are allowed. If, for some reason, the test wants to emit some warnings to the user, it can use the Log4j mechanism for that purpose. Simply extend the `logger.properties` with the following code:

```
# Extensible Test Suite appender
log4j.appender.stringbuffer=tudresden.ocl20.logging.appender.StringBufferAppender
log4j.appender.stringbuffer.layout = org.apache.log4j.PatternLayout
log4j.appender.stringbuffer.layout.ConversionPattern = %C1:  %m%n%n
```

Then add this appender to the logging of the plug-in. An example usage of this mechanism can be found in the plug-in `tudresden.ocl20.pivot.metamodels.test`.

To run the extensible test suite, right-click on `OCL2TestSuiteRunner` in the package `tudresden.ocl20.pivot.testsuite.runner`. Choose `Run As ->JUnit Plug-in Test` as in figure 9.1.

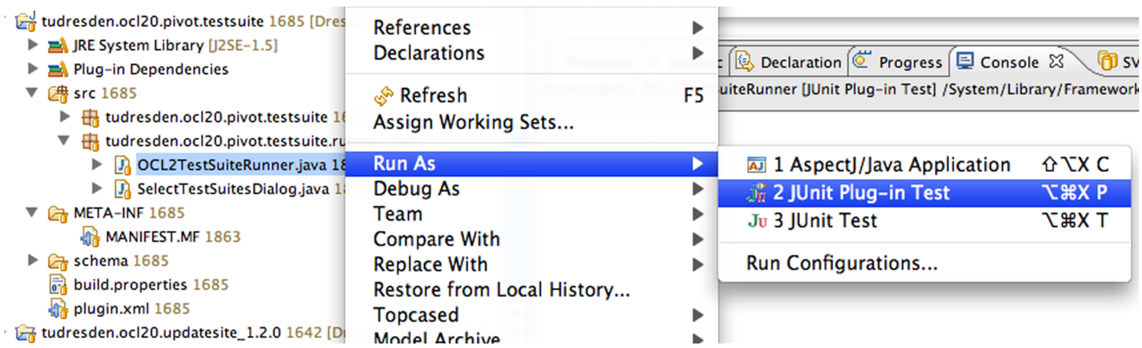

Figure 9.1: Run the extensible test suite.

10 THE GENERIC META-MODEL TEST SUITE

Chapter written by Claas Wilke

To test the adaptation of a meta-model to the pivot model of Dresden OCL2 for Eclipse, the toolkit provides a generic test suite that can be simply instantiated by each adapted meta-model. This chapter shortly presents, how the generic meta-model test suite can be instantiated to test an adapted meta-model.

10.1 THE TEST SUITE PLUG-IN

The generic meta-model test suite is located in the plug-in `tudresden.ocl20.pivot.metamodels.test`. The test suite provides a set of JUnit tests, that check the functionality of all operations that must be adapted by every meta-model that shall be adapted to the pivot model. The adaptation of a meta-model to the pivot model is explained in Chapter 6. The test suite contains about 150 Junit tests.

To instantiate the generic test suite for a new adapted meta-model, only two resources must be provided: (1) a model modeled in the newly adapted meta-model that contains instances of all pivot model types that shall be tested, and (2) a Java class that instantiates the test suite with the modeled model. During test execution, the generic test suite uses the provided model to test the meta-model (see Figure 10.1). Both, the model and the Java class are shortly presented in the following sections.

10.2 THE REQUIRED MODEL TO TEST A META-MODEL

Figure 10.2 shows the test model that must be implemented in a meta-model that shall be tested with the generic meta-model test suite. At a first sight, the model seems to be very complex. But many of the contained features are optional, because some data structures and types of the pivot model could be (but do not have to be) implemented by a meta-model. E.g., a meta-model

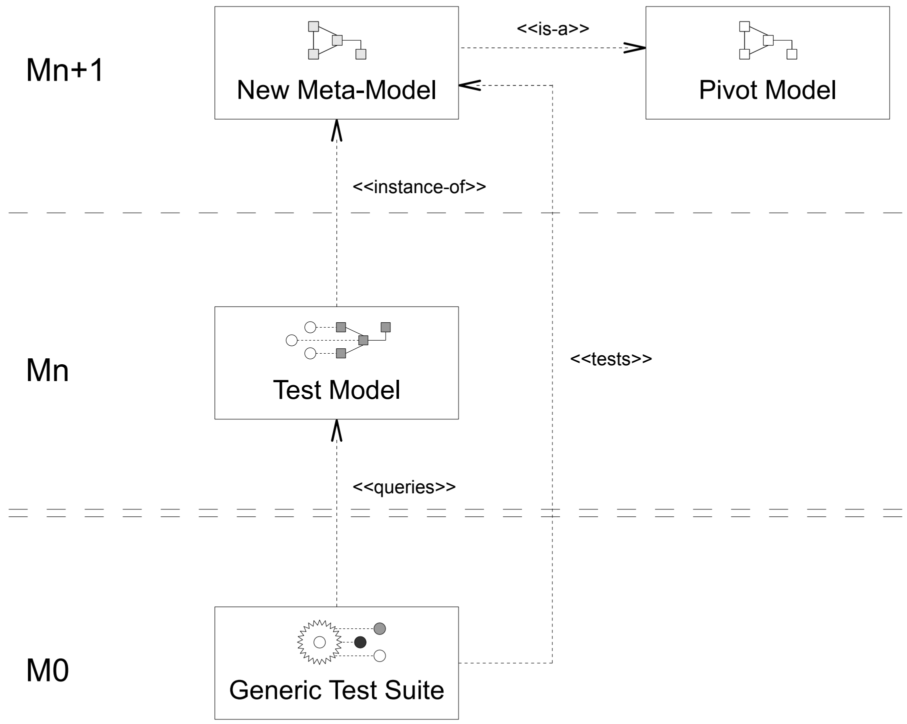

Figure 10.1: The Generic Meta-Model Test Suite in respect to the Generic Three Layer Architecture

can provide a enumeration type but not has to. If a structure is not provided by a test model, the test suite will print a warning during test execution that the expected structure has not been found. If the structure is not adapted intentionally, the warnings can be ignored. In the following, all types and relations of the test model are explained shortly.

10.2.1 TestTypeClass1 and TestTypeClass2

As their name already tells us, the classes `TestTypeClass1` and `TestTypeClass2` are used to test the adaptation of `Types`. Each meta-model has to provide types, thus, these classes are required. Both classes provide an operation and a property. The association between the two classes is optional because not all meta-models contain associations. But the generalization between `TestTypeClass2` and `TestTypeClass1` is required.

10.2.2 TestTypeInterface1 and TestTypeInterface2

Besides classes, some meta-models also provide a second type that must be mapped to the pivot model type `Type`, which is the interface type. To test the adaptation of the `Type` element for meta-models that have both, classes (or types) and interfaces, the test model contains two interfaces `TestTypeInterface1` and `TestTypeInterface2`. They are optional and can be used to test the adaptation of interfaces. E.g., a meta-model that adapts both classes and interfaces is the UML2 meta-model located in the plug-in `tudresden.ocl20.pivot.metamodels.uml2`.

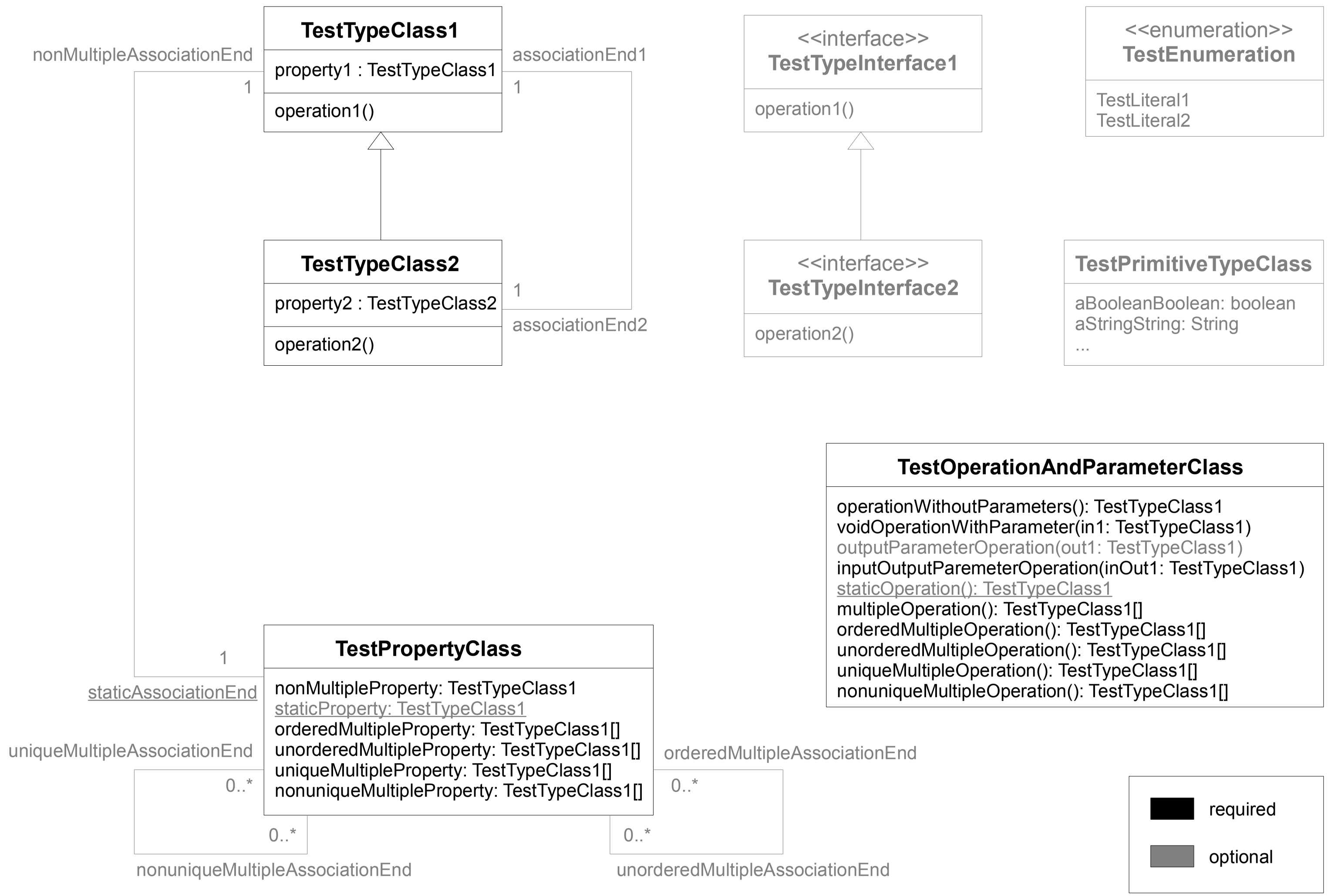

Figure 10.2: The required Test Model to test a Meta-Model's adaptation. The gray parts are optional.

10.2.3 TestEnumeration

To test the adaptation of a `Enumeration` type, the class `TestEnumeration` can be used. Because enumerations are not part of every meta-model, this class of the test model is optional.

10.2.4 TestPrimitiveTypeClass

A special class in the test model is the class `TestPrimitiveTypeClass`. This class contains a property for each primitive type of the adapted meta-model that shall be tested. Each property has the type of the `PrimitiveType` whose adaptation shall be tested. Important is the name of the property. If the property's name starts with `aBoolean`, the type is tested as adapted to a pivot model's `PrimitiveType` of the kind `Boolean`. If the name starts with `anInteger` instead, the types is tested as an `Integer`. E.g., the example property `aStringString` shown in `TestPrimitiveTypeClass` in Figure 10.2 is tested as adapted to a `String`. Table 10.1 shows the adaptation of the different property name prefixes to the different `PrimitiveTypeKinds`.

Property Name Prefix	**Expected PrimitiveTypeKind**
`aBoolean...`	`Boolean`
`anInteger...`	`Integer`
`aReal...`	`Real`
`aString...`	`String`

Table 10.1: The adaptation of properties' name prefixes to PrimitiveTypeKinds.

10.2.5 TestPropertyClass

The class `TestPropertyClass` contains properties to test the right adaptation of the pivot model element `Property`. Additionally, the class has many associations that can be used to test the adaptation of a second property type for associations (like in the UML2 meta-model, plug-in: `tudresden.ocl20.pivot.metamodels.uml2`). Thus, all associations are optional and are not required to test a meta-model's adaptation. The names of the contained properties are self-explainable: The property `nonMultipleProperty` is used to test the adaptation of a `Property` that cannot contain multiple values. The property `staticProperty` represents a static `Property` and is optional because not all meta-models contain a `static` modifier. The other properties are used to test multiple properties that are ordered, unordered, unique and non-unique.

10.2.6 TestOperationAndParameterClass

Similar to the class `TestPropertyClass` (presented above), the class `TestOperationAndParameterClass` contains operations to test the adaptation of all different kinds of `Operations`. Additionally, the class is also used to test the adaptation of `Parameters` of operations. Some of the operations are optional (like the static operation and the operation with an output value), others are required.

```java
1  import tudresden.ocl20.pivot.metamodels.test.MetaModelTestPlugin;
2  import tudresden.ocl20.pivot.metamodels.test.MetaModelTestSuite;
3
4  @Suite.SuiteClasses(value = { MetaModelTestSuite.class })
5  public class TestUML2MetaModel extends MetaModelTestSuite {
6
7    /** The id of the {@link IMetamodel} which shall be tested. */
8    private static final String META_MODEL_ID = UML2MetamodelPlugin.ID;
9
10   /** The path of the model which shall be tested. */
11   private static final String TEST_MODEL_PATH = "model/testmodel.uml";
12
13   /**
14    * <p>
15    * Prepares the {@link MetaModelTestSuite}.
16    * </p>
17    */
18   @BeforeClass
19   public static void setUp() {
20
21     MetaModelTestPlugin.prepareTest(UML2MetaModelTestPlugin.PLUGIN_ID,
22       TEST_MODEL_PATH, META_MODEL_ID);
23   }
24 }
```

Listing 10.1: An instantiation of the generic meta-model test suite.

10.3 INSTANTIATING THE GENERIC TEST SUITE

As mentioned above, to initialize the generic meta-model test suite, only one Java class must be implemented that instantiates the test suite with the test model implemented in the adapted meta-model. Listing 10.1 shows a Java class that instantiates the test suite to test the UML2 meta-model.

Important is that the class provides a JUnit test suite (according to JUnit 4 conventions), that contains the `MetaModelTestSuite` (line 4). Additionally, the class only has to provide a `setUp()` method that can be used to setup the test suite before execution (lines 13 to 23). Inside the `setUp()` method, the operation `MetaModelTestPlugin.prepareTest(String, String, String)` must be invoked. The method initializes the environment of the generic test suite by setting three arguments:

1. The `ID` of the plug-in that contains the test model used for testing (e.g., `tudresden.ocl20.pivot.metamodels.uml2.test`,
2. The location of the test model relative to the plug-in's root folder (e.g., `model/testmodel.uml`),
3. And the `ID` of the meta-model that shall be tested (e.g. `tudresden.ocl20.pivot.metamodels.uml2`.

Afterwards, the implemented Java class can be executed as a *JUnit Plug-in Test* in Eclipse. The test suite should then inform you (by failed test cases) which parts of your meta-model adaptation are wrong implemented or missing. As mentioned above, warnings caused by missing parts of the test model–that were not implemented intentionally–can be ignored.

10.4 SUMMARY

This chapter shortly introduced into the generic meta-model test suite of Dresden OCL2 for Eclipse. For further details of the test suite investigate the test suite plug-in `tudresden.ocl20.pivot.metamodels.test` or the existing test suite instantiations in the plug-ins `tudresden.ocl20.pivot.metamodels.uml2.test`, `tudresden.ocl20.pivot.metamodels.ecore.test`, or `tudresden.ocl20.pivot.metamodels.java.test`.

11 THE GENERIC MODEL INSTANCE TYPE TEST SUITE

Chapter written by Claas Wilke

To test the adaptation of a model instance type to Dresden OCL2 for Eclipse, the toolkit provides a generic test suite that can be simply instantiated by each adapted model instance type. This chapter shortly presents, how the generic model instance type test suite can be instantiated to test an adapted model instance type.

11.1 THE TEST SUITE PLUG-IN

The generic model instance type test suite is located in the plug-in `tudresden.ocl20.pivot.modelinstancetype.test`. The test suite provides a set of JUnit tests, that check the functionality of all operations that must be adapted by every model instance type that shall be supported for Dresden OCL2 for Eclipse. The adaptation of a model instance type to Dresden OCL2 for Eclipse is explained in Chapter 7. The test suite contains about 100 JUnit tests.

To instantiate the generic test suite for a new adapted model instance type, only two resources must be provided: (1) a model instance of the newly adapted model instance type that contains instances of a set of model types defined in a special test model, and (2) a Java class that instantiates the test suite with the model instance. During test execution, the generic test suite uses the provided model instance to test the model instance type (see Figure 11.1). Both, the model instance and the Java class are shortly presented in the following sections.

11.2 THE REQUIRED MODEL INSTANCE TO TEST A META-MODEL

The Figures 11.2 and 11.3 show the test model of which a model instance must be provided. At a first sight, the model seems to be very complex. In the following, all types and relations of the test model are explained shortly. Because a detailed explanation would be too large for this document we recommend to investigate the already existing implementations of the test model provided with the test suite implementations for the Java and the EMF Ecore model instance types (the plug-ins `tudresden.ocl20.modelinstancetype.java.test` and `tudresden.ocl20.modelinstancetype.ecore.test`).

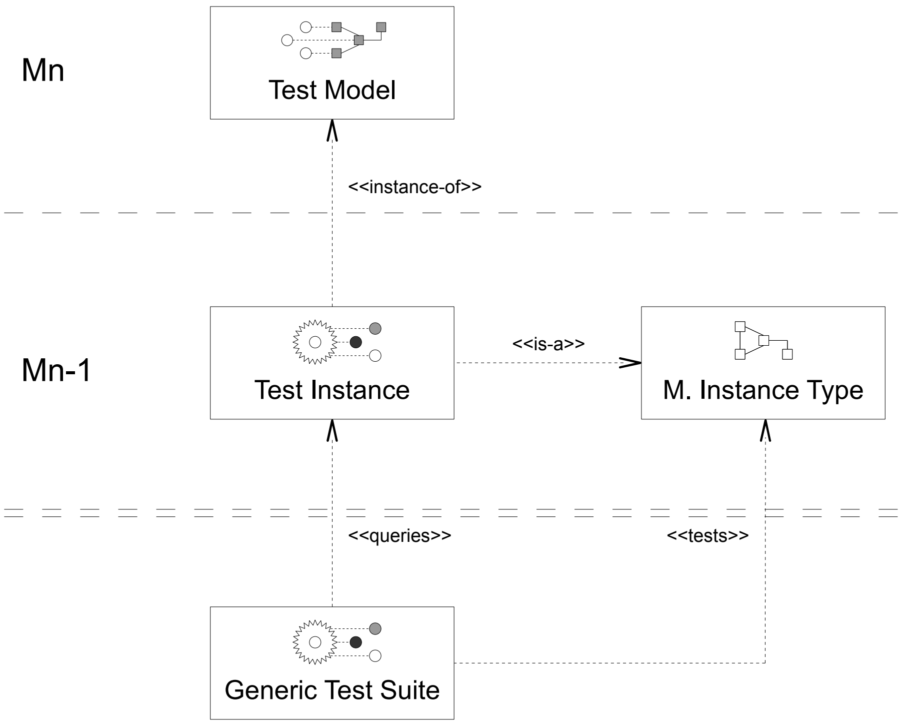

Figure 11.1: The Generic Model Instance Type Test Suite in respect to the Generic Three Layer Architecture

11.2.1 The ContainerClass

The `ContainerClass` is part of the model because in some model instance types, each model instance must have exactly one root element that contains all other instance elements (e.g., EMF Ecore instances). Thus, the `ContainerClass` is responsible to manage Sets of all other classes that should be instantiated to test the model instance type implementation. The `ContainerClass` should be instantiated by at least one object of the model instance. **Please be aware that if the collections of other implementation types contained by this object are not instantiated appropriately, the whole test suite may fail.**

11.2.2 Class1

`Class1` is the major type of the model that is used to test various functionalities of the model instance type. It provides a set of different properties that are used to test the right adaptation of non-multiple, multiple, ordered, unordered, unique and non-unique properties (`nonMultipleProperty`, `multiple...Property`). Further properties are required to provide default values that can be used to invoke the operations provided by `Class1` (`argumentProperty...`).

Besides properties, `Class1` provides a enormous set of operations as well. Some operations are responsible to test the invocation of operations with different result types (`voidOperation()`, `nonMultipleOperation()` and `multiple...Operation()`), others are required to test the invocation of operations with different argument types (`voidOperationWith...Argument()`).

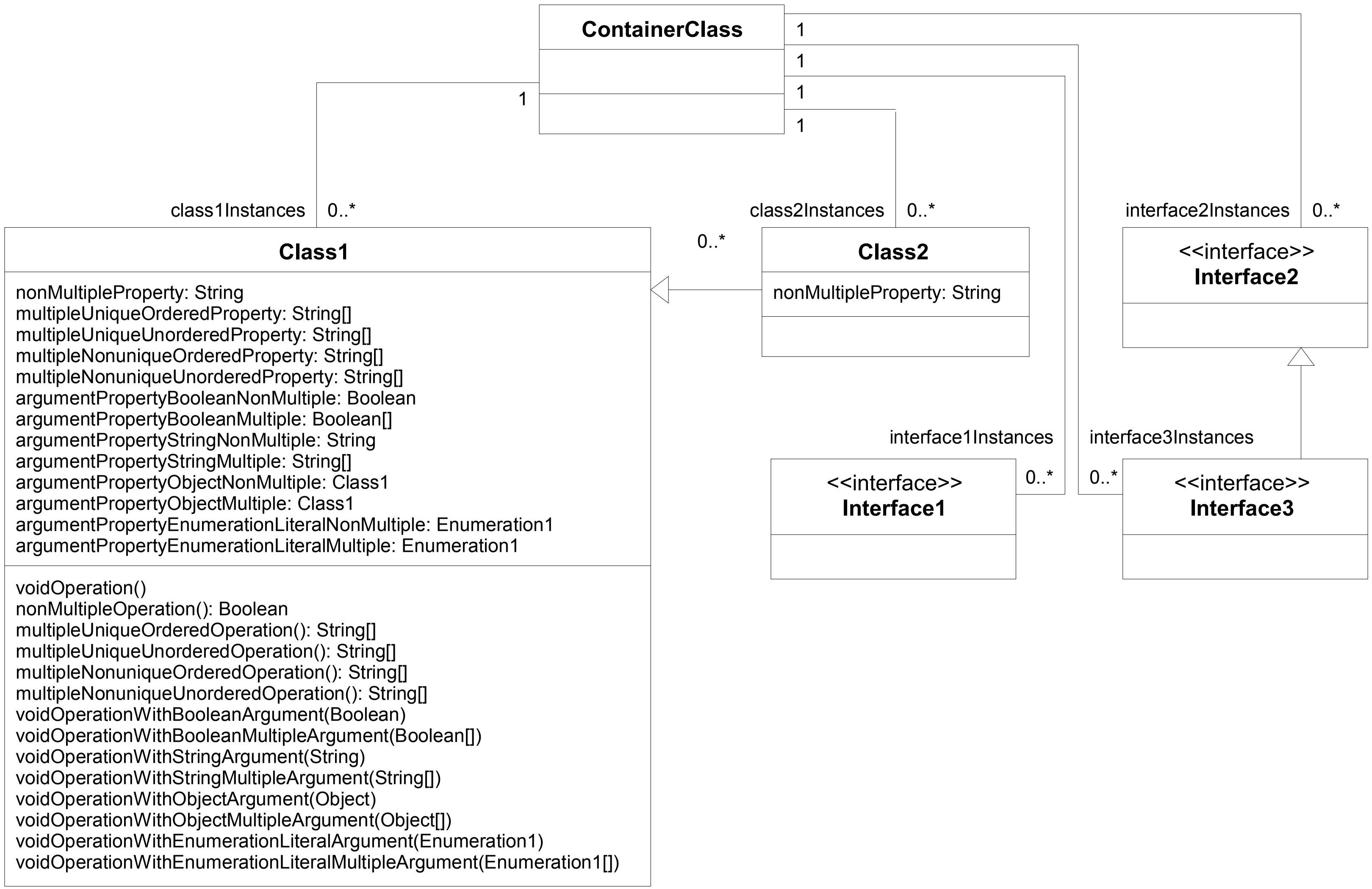

Figure 11.2: The required Test Model to test a Model Instance Type’s adaptation (part 1).

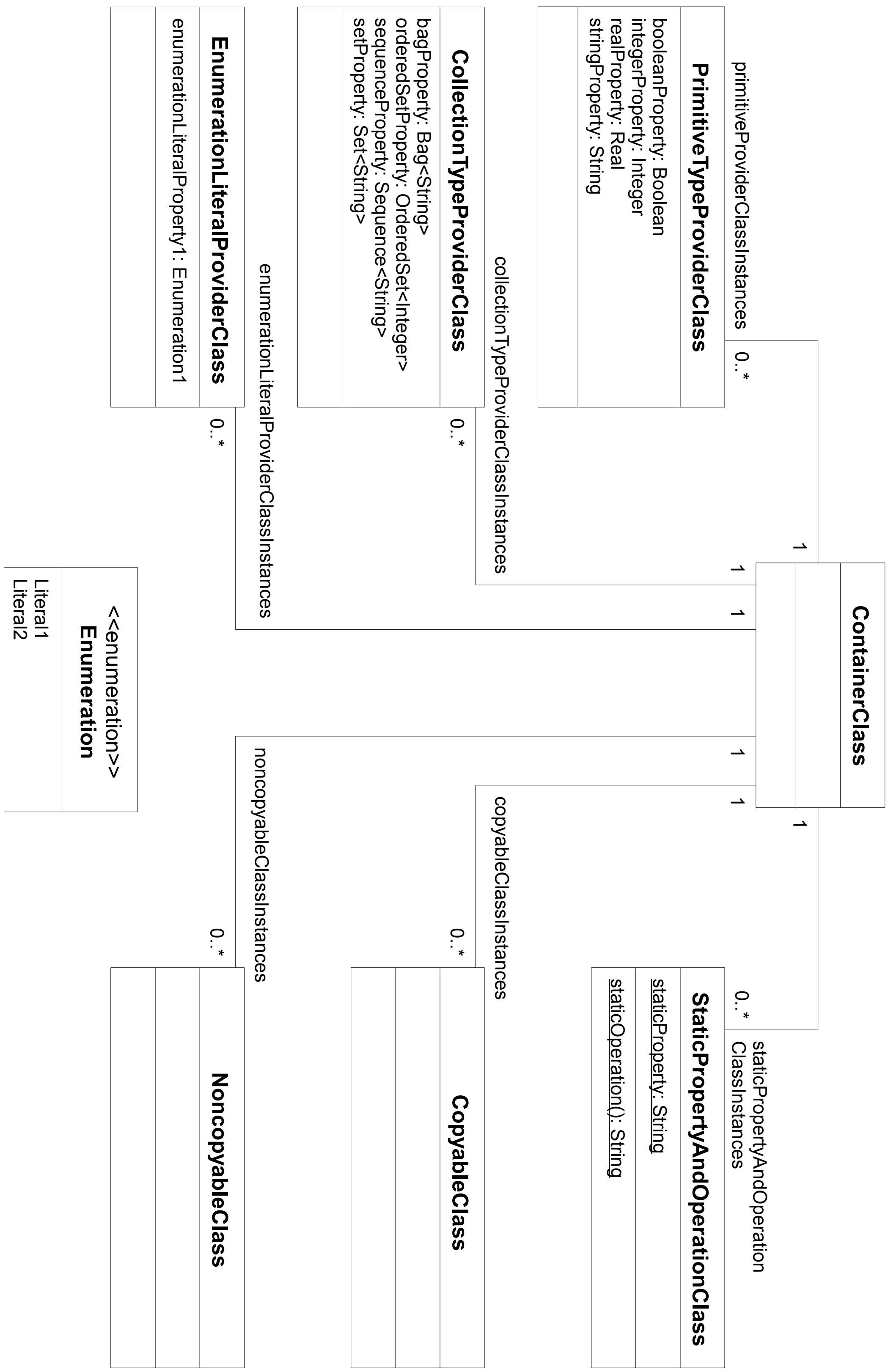

Figure 11.3: The required Test Model to test a Model Instance Type's adaptation (part 2).

11.2.3 Class2

The test model contains a second class called `Class2` that is used to test casting between classes and subclasses. For the same purpose, the only property of the class called `nonMultipleProperty` is required. It is used to test the special access on parent properties in OCL e.g., by the statement `aClass2.oclAsType(Class1).nonMultipleProperty` which returns the property of `Class1` instead of the property of `Class2`.

11.2.4 Interface1, Interface2 and Interface3

The test model provides three other types called `Interface1`, `Interface2` and `Interface3`. Although the *Pivot Model* itself does not differentiate between classes and interfaces (they are all handled as types internally), these three types are used to test further inheritance and casting relationships. The interface types should be used by the adapted model instance type to implement objects that extends more than one type (multiple inheritance). If multiple inheritance is not possible for the adapted model instance type (not even for interfaces!) these types can be ignored.

11.2.5 PrimitiveTypeProviderClass, CollectionTypeProviderClass and EnumerationLiteralProviderClass

The classes `PrimitiveTypeProviderClass`, `CollectionTypeProviderClass` and `EnumerationLiteralProviderClass` are responsible to provide instances of all model object types that shall be adapted to special-handled model instance objects (which are primitive types, collections (and arrays) and enumeration literals). E.g., the `PrimitiveTypeProviderClass` should provided values as their properties that should be handled as Booleans, Integers, Reals and Strings.

Because some model instance types provide different types that shall be mapped to the same kind of type in Dresden OCL2 for Eclipse (e.g., the types `int` and `java.lang.Integer` of Java should be mapped both to the primitive type `Integer`), these classes can provide multiple properties for the same type. The properties should then be numbered, ignoring the number for the first property and starting with the number two for the second property. E.g., if the `PrimitiveTypeProviderClass` should provide multiple different `Integer` instances their properties should be called `integerProperty`, `integerProperty2`, `integerProperty3` and so on. If multiple properties for the same type are provided, the test suite must be informed during setup to load all the different properties (see also Section 11.3).

11.2.6 StaticPropertyAndOperationClass

The `StaticPropertyAndOperationClass` is used to test the invocation of static properties and operations. Because not every model instance type supports the invocation of static properties and operations (e.g., EMF Ecore does not), these tests are based on an extra type. If the model instance type that shall be tested does not support static features, this class could be ignored.

11.2.7 Copyable- and NonCopyableClass

OCL supports the special operation `@pre`, that can be used in OCL to access to values of properties and operations that existed before the execution of an operation that is the current context of the constraint, the `@pre` expression belongs to. To support this operation, the OCL2 Interpreter must copy and store values of model instance objects.

In some model instance types it is not possible to support such a `copy()` or `clone()` method for every single object. Thus, the test model contains two further types called `Copyable` and `NonCopyableClass` that should be used to implement data structures that can be used to test the copy mechanism during run-time explicitly.

11.3 INSTANTIATING THE GENERIC TEST SUITE

As mentioned above, to initialize the generic model instance type test suite, only one Java class must be implemented that instantiates the test suite with the test model instance. Listing 11.1 shows a Java class that instantiates the test suite to test the Java model instance type.

Important is that the class provides a JUnit test suite (according to JUnit 4 conventions), that contains the `ModelInstanceTypeTestSuite` (line 8). Additionally, the class only has to provide a `setUp()` method that can be used to setup the test suite before execution (lines 22 to 45). Inside the `setUp()` method, the operation `ModelInstanceTypeTestPlugin.prepareTest(String, String, String)` must be invoked. The method initializes the environment of the generic test suite by setting three arguments:

1. The `ID` of the plug-in that contains the test model instance used for testing (e.g., `tudresden.ocl20.pivot.modelinstancetype.java.test`,
2. The location of the test model relative to the plug-in's root folder (e.g., `bin/tudresden/ocl20/pivot/modelinstancetype/java/test/modelinstance/ProviderClass.class`),
3. And the `ID` of the model instance type that shall be tested (e.g. `tudresden.ocl20.pivot.modelinstancetype.java`.

Additionally, the method could set different counters that can be used to inform the test suite that more than one implementation of a primitive or collection type is provided in their provider class (lines 43 to 43, see also Subsection 11.2.5).

Afterwards, the implemented Java class can be executed as a *JUnit Plug-in Test* in Eclipse. The test suite should then inform you (by failed test cases) which parts of your model instance type adaptation are wrong implemented or missing. Warnings caused by missing parts of the test model instance–that were not implemented intentionally–can be ignored.

11.4 SUMMARY

This chapter shortly introduced into the generic model instance type test suite of Dresden OCL2 for Eclipse. For further details of the test suite investigate the test suite plug-in `tudresden.ocl20.pivot.modelinstancetype.test` or the existing test suite instantiations in the plug-in `tudresden.ocl20.pivot.modelinstancetype.java.test`, and the plug-in `tudresden.ocl20.pivot.modelinstancetype.ecore.test`.

```java
import tudresden.ocl20.pivot.modelinstancetype.test.
        ModelInstanceTypeTestPlugin;
import tudresden.ocl20.pivot.modelinstancetype.test.
        ModelInstanceTypeTestServices;
import tudresden.ocl20.pivot.modelinstancetype.test.
        ModelInstanceTypeTestSuite;

@Suite.SuiteClasses(value = { ModelInstanceTypeTestSuite.class })
public class TestJavaModelInstanceType extends
    ModelInstanceTypeTestSuite {

  /** The id of the {@link IModelInstanceTypeObject}
    * which shall be tested. */
  private static final String MODEL_INSTANCE_ID =
    JavaModelInstanceTypePlugin.PLUGIN_ID;

  /** The path of the model which shall be tested. */
  private static final String TEST_MODELINSTANCE_PATH =
    "bin/tudresden/ocl20/pivot/modelinstancetype/" +
    "java/test/modelinstance/ProviderClass.class";

  /**
   * <p>
   * Prepares the {@link ModelInstanceTypeTestSuite}.
   * </p>
   */
  @BeforeClass
  public static void setUp() {

    ModelInstanceTypeTestPlugin.prepareTest(
      JavaModelInstanceTypeTestPlugin.PLUGIN_ID,
      TEST_MODELINSTANCE_PATH, MODEL_INSTANCE_ID);

    ModelInstanceTypeTestServices.getInstance()
      .setBooleanPropertyCounter(2);
    ModelInstanceTypeTestServices.getInstance()
      .setIntegerPropertyCounter(10);
    ModelInstanceTypeTestServices.getInstance()
      .setRealPropertyCounter(4);
    ModelInstanceTypeTestServices.getInstance()
      .setStringPropertyCounter(4);
    ModelInstanceTypeTestServices.getInstance()
      .setSequencePropertyCounter(2);
  }
}
```

Listing 11.1: An instantiation of the generic model instance test suite.

III APPENDIX

Tables

Software	Available at
Eclipse 3.4.x	http://www.eclipse.org/
Eclipse Modeling Framework (EMF)	http://www.eclipse.org/modeling/emf/
Eclipse Model Development Tools (MDT) (only with the UML2.0 meta model)	http://www.eclipse.org/modeling/mdt/
Eclipse Plug-in Development Environment (only to run the toolkit using the source code distribution)	http://www.eclipse.org/pde/

Table 1: Software needed to run Dresden OCL2 for Eclipse (**If not using the Eclipse MDT Distribution)**.

Feature	Plug-ins
Core	**Required:** tudresden.ocl20.pivot.logging tudresden.ocl20.pivot.essentialocl tudresden.ocl20.pivot.essentialcol.edit tudresden.ocl20.pivot.essentialocl.editor tudresden.ocl20.pivot.essentialocl.standardlibrary tudresden.ocl20.pivot.examples.royalandloyal tudresden.ocl20.pivot.modelbus tudresden.ocl20.pivot.modelbus.ui tudresden.ocl20.pivot.pivotmodel tudresden.ocl20.pivot.pivotmodel.edit tudresden.ocl20.pivot.standardlibrary.java **Optional:** tudresden.ocl20.pivot.essentialocl.tests tudresden.ocl20.pivot.pivotmodel.tests

Table 2: The plug-ins of Dresden OCL2 for Eclipse related to their feature (part 1).

Feature	Plug-ins
Examples	**Optional:** tudresden.ocl20.pivot.examples.living tudresden.ocl20.pivot.examples.pml tudresden.ocl20.pivot.examples.royalandloyal.ocl22javacode tudresden.ocl20.pivot.examples.simple tudresden.ocl20.pivot.examples.simple.ocl22javacode
Interpreter	**Required (for interpretation):** tudresden.ocl20.interpreter tudresden.ocl20.interpreter.ui **Optional:** tudresden.ocl20.interpreter.test
Metamodels	**Required (at least one of the following):** tudresden.ocl20.pivot.metamodels.ecore tudresden.ocl20.pivot.metamodels.java tudresden.ocl20.pivot.metamodels.uml2 **Optional:** tudresden.ocl20.pivot.metamodels.test tudresden.ocl20.pivot.metamodels.ecore.test tudresden.ocl20.pivot.metamodels.java.test tudresden.ocl20.pivot.metamodels.uml2.test
Model Instances	**Required (at least one of the following for interpretation):** tudresden.ocl20.pivot.modelinstancetype.ecore tudresden.ocl20.pivot.modelinstancetype.java **Optional:** tudresden.ocl20.pivot.modelinstancetype.test tudresden.ocl20.pivot.modelinstancetype.test.ecore tudresden.ocl20.pivot.modelinstancetype.test.java
Ocl2Java	**Required (for code generation):** tudresden.ocl20.pivot.ocl2java tudresden.ocl20.pivot.ocl2java.ui **Optional (eventually for code execution):** tudresden.ocl20.pivot.ocl2java.types **Optional:** tudresden.ocl20.pivot.ocl2java.test
Dresden OCL2 for Eclipse Parser Feature	**Required:** tudresden.ocl20.pivot.ocl2parser tudresden.ocl20.pivot.parser tudresden.ocl20.pivot.parser.ui **Optional:** tudresden.ocl20.pivot.ocl2parser.test

Table 3: The plug-ins of Dresden OCL2 for Eclipse related to their feature (part 2).

Living Example	
Plug-in Package	tudresden.ocl20.pivot.examples.living
Meta-Model	Java Meta-Model
Model	bin/tudresden/ocl20/pivot/examples/living/ModelProviderClass.class
OCL Expressions	constraints/*.ocl
Model Instance Type	Java
Model Instance	bin/tudresden/ocl20/pivot/examples/living/ModelInstanceProvider-Class.class
Simple Example	
Plug-in Package	tudresden.ocl20.pivot.examples.simple
Meta-Models	Java or MDT UML2
Model	src/tudresden.ocl20.pivot.examples.simple.ModelProviderClass.java, model/simple.uml
OCL Expressions	constraints/*.ocl
Model Instance Type	Java
Model Instance	src/tudresden.ocl20.pivot.examples.simple.instance.Model-InstanceProviderClass.java
PML Example	
Plug-in Package	tudresden.ocl20.pivot.examples.pml
Meta-Model	EMF Ecore
Model	model/pml.ecore
OCL Expressions	constraints/*.ocl
Model Instance Type	EMF Ecore
Model Instances	modelinstance/goodModelinstance.pml, modelinstance/badModelinstance.pml
Royal and Loyal Example	
Plug-in Package	tudresden.ocl20.pivot.examples.royalandloyal
Meta-Model	MDT UML2
Model	model/royalsandloyals.ecore model/royalsandloyals.uml
OCL Expressions	constraints/*.ocl
Model Instance Type	Java
Model Instance	src/tudresden.ocl20.pivot.examples.royalandloyal.instance.Model-InstanceProviderClass.java

Table 4: The examples provided with Dresden OCL2 for Eclipse.

LIST OF FIGURES

LISTINGS

LIST OF ABBREVIATIONS

AJDT	AspectJ Development Tools
DOT	Dresden OCL Toolkit
DOT4Eclipse	Dresden OCL2 for Eclipse
Eclipse MDT	Eclipse Modeling Development Tools
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
JAR	Java Archive
JDK	Java Development Kit
JRE	Java Run-time Environment
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OSGi	Open Services Gateway initiative
SVN	Subversion
UML	Unified Modeling Language
XML	Extensible Markup Language

BIBLIOGRAPHY

[Brä07] BRÄUER, Matthias: *Models and Metamodels in a QVT/OCL Development Environment*, Technische Universität Dresden, Germany, Großer Beleg (Minor Thesis), May 2007

[DW09] DEMUTH, Birgit; WILKE, Claas: Model and Object Verification by Using Dresden OCL. In: *Proceedings of the Russian-German Workshop "Innovation Information Technologies: Theory and Practice", July 25-31, Ufa, Russia, 2009*, Ufa State Aviation Technical University, Ufa, Bashkortostan, Russia, 2009

[GHJV95] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns - Elements of Reusable Object-Oriented Software*. 2nd. Addison-Wesley Professional, Indianapolis, IN, USA, 1995

[OMG06] Object Management Group (OMG), Needham, MA, USA: *Meta Object Facility (MOF) Core Specification, OMG Available Specification*. http://www.omg.org/spec/MOF/2.0/. Version: 2.0, January 2006

[OMG09] Object Management Group (OMG), Needham, MA, USA: *OMG Unified Modeling Language™ (OMG UML), Infrastructure*. http://www.omg.org/spec/UML/2.2/. Version: 2.2, February 2009

[URL10a] *Apache Chainsaw*. Apache Logging Services. http://logging.apache.org/chainsaw/. Version: 2010

[URL10b] *AspectJ Development Tools (AJDT)*. Eclipse AJDT Project Website hosted by the the Eclipse Foundation. http://www.eclipse.org/aspectj/. Version: 2010

[URL10c] *Eclipse project Website*. Eclipse project Website. http://www.eclipse.org/. Version: 2010

[URL10d] *Polarion Software: Subversive*. Polarion.org Community. http://www.polarion.com/products/svn/subversive.php. Version: 2010

[URL10e] *Sourceforge Project Site of the Dresden OCL Toolkit*. Sourceforge project website. http://sourceforge.net/projects/dresden-ocl/. Version: 2010

[URL10f] *SVN of the Dresden OCL Toolkit*. Subversion Repository. http://dresden-ocl.svn.sourceforge.net/svnroot/dresden-ocl. Version: 2010

[URL10g] *Update Site of the Dresden OCL Toolkit*. Eclipse Update Site. http://dresden-ocl.sourceforge.net/downloads/updatesite/. Version: 2010

[URL10h] *Website of the Dresden OCL Toolkit*. Project website. http://dresden-ocl.sourceforge.net/. Version: 2010

[Wil09] WILKE, Claas: *Java Code Generation for Dresden OCL2 for Eclipse*, Technische Universität Dresden, Germany, Großer Beleg (Minor Thesis), February 2009