TECHNISCHE UNIVERSITÄT DRESDEN

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

DRESDEN OCL

MANUAL FOR INSTALLATION, USE AND DEVELOPMENT

Claas Wilke, Michael Thiele and Björn Freitag

Last update on April 28, 2012

Dresden OCL has been developed at the Technische Universität Dresden, Department of Computer Science, Software Technology Group. Dresden OCL and this manual are available a the Dresden OCL project website (http://www.dresden-ocl.org/).

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 OCL or at least parts of it. Thus, please inform us, if you use Dresden OCL wthin your project, tool or application.

Dresden OCL

This Manual

ABSTRACT

This document contains the documentation of Dresden OCL. In the first part of this manual the general use of Dresden OCL is explained. Afterwards, use cases like OCL interpretation and code generation are presented. The second part of this manual contains the technical documentation of Dresden OCL, like its architecture and the adaptation of further types of models and model instances to Dresden OCL.

Please be aware that Dresden OCL is a project developed at the Technische Universität Dresden, Software Technology Group. Parts of the project have been designed and implemented during student theses and have been developed as prototypes only. Thus, Dresden OCL is far from being complete. To report bugs and errors or request additional features or answers to specific questions visit Dresden OCL's website [**?**] or the project site at *Sourceforge* [**?**].

The procedure's described in this manual were run and tested with *Eclipse 3.6* [**?**]. We recommend to use the *Eclipse Modeling Tools Edition* which contains most required plug-ins to run Dresden OCL. Otherwise you need to install at least the plug-ins enlisted in Table 1. Alternatively, Dresden OCL may be used as a stand-alone library for Java. If you want to use the stand-alone distribution, you cannot use the GUIs and editors provided with Dresden OCL since the GUI elements depend on Eclipse. The use of the stand-alone distribution is documented in Chapter 9.

CONTENTS

I USING DRESDEN OCL

1 ABOUT DRESDEN OCL

Chapter written by Claas Wilke and Michael Thiele

Dresden OCL is developed as a project at the Technische Universität Dresden (TUD), Software Technology Group since 1999. Its latest version is released as a set of Eclipse plug-ins and thus, is sometimes also referred as *Dresden OCL2 for Eclipse*. Dresden OCL consists of a set of OCL tools, including an OCL parser, an OCL interpreter, an OCL-to-Java and an OCL-to-SQL code generator.

In this chapter, some general information on Dresden OCL is presented. The supported OCL version and the differences to the official OCL specification are documented. Supported metamodels, models and model instances are shortly presented. If you are not interested in such information, you can skip this chapter and continue with the installation and general use of Dresden OCL as documented in Chapter 2.

1.1 SUPPORTED VERSION OF OCL

Dresden OCL generally supports OCL 2.3 as specified in [**?**]. Nevertheless, some differences between Dresden OCL and OCL 2.3 exist as documented in the following:

1.1.1 Comformance of Dresden OCL to the OCL 2.3 specification

The OCL 2.3 specification defines a set of compliance points that can be addressed by tools implementing OCL [**?**, p. 1]. Table 1.1 summarises the compliance points addressed by Dresden OCL.

Besides the features enlisted in Table 1.1 some minor differences exist between Dresden OCL and the OCL specification:

- Dresden OCL does not support the operation `OclAny.oclLocale()` introduced within OCL 2.3 that can be used to implement different behavior for the `String` operations `toLowerCase()`, `toUpperCase()`, `<(String)`, `>(String)`, `<=(String)`, and `>=(String)`. Since Dresden OCL is implemented in Java, it uses the semantics of `java.lang.String.toLowerCase()`, `toUpperCase()`, and `compareTo(String)` instead.

Compliance Point	Support in Dresden OCL
Syntax	Fully supported (Except OclMessage, its related operations, and OclAny.oclLocale()).
XMI	XMI export import of OCL is currently not supported.
Evaluation - allInstances() - @pre in postconditions - OclMessage - Navigating non-navigable association ends - accessing private and protected features	 supported (with some limitations to specific technological spaces, e.g., Java) supported not supported not supported supported

Table 1.1: Compliance points addressed by Dresden OCL (v. 3.x) according to [**?**, p. 1].

a	b	not a	a or b	a xor b	a and b	a implies b
false	false	true	false	false	false	true
false	true	- " -	true	true	false	true
false	null	- " -	invalid	invalid	false	true
false	invalid	- " -	invalid	invalid	false	true
true	false	false	true	true	false	false
true	true	- " -	true	false	true	true
true	null	- " -	true	invalid	invalid	invalid
true	invalid	- " -	true	invalid	invalid	invalid
null	false	invalid	invalid	invalid	false	invalid
null	true	- " -	true	invalid	invalid	true
null	null	- " -	invalid	invalid	invalid	invalid
null	invalid	- " -	invalid	invalid	invalid	invalid
invalid	false	invalid	invalid	invalid	false	invalid
invalid	true	- " -	true	invalid	invalid	true
invalid	null	- " -	invalid	invalid	invalid	invalid
invalid	invalid	- " -	invalid	invalid	invalid	invalid

Table 1.2: Decision Table for boolean operators in Dresden OCL.

- Dresden OCL does not support the expression of UnlimitedNaturalExpressions to define multiplicities within constraints such as `(1..*)`.

1.1.2 Different Semantics of OCL Expressions in Dresden OCL

The current OCL 2.3 specification [**?**] contains some inconsistencies and misses some definitions especially when evaluating `invalid` or `null` values. Thus, we had to assume or change the semantics of OCL during evaluation of some OCL statements. In the following we present the differences of the OCL semantics used in Dresden OCL compared with the official OCL specification [**?**].

Boolean Operators

Boolean operators in Dresden OCL are interpreted as documented in Table 1.2. As documented, the operator `and` always results in `false` as long as one of its operands is `false`, ignoring whether the other operand is `null` or even `invalid`. The operator `or` results in `true`, as long as one of its operands is `true`. Both `false implies null` and `false implies invalid` result in `true`. The

a	b	a = b	a <> b	a < b	a <= b	a > b	a >= b
42	42	true	false	false	true	false	true
42	7	false	true	false	false	true	true
42	null	false	true	invalid	invalid	invalid	invalid
null	null	true	false	invalid	invalid	invalid	invalid
42	invalid	false	true	invalid	invalid	invalid	invalid
invalid	invalid	true	false	invalid	invalid	invalid	invalid

Table 1.3: Decision Table for equality operators in Dresden OCL.

```
1 Bag { true, null } -> exists(true) => true
2 Bag { true, null } -> forAll(true) => invalid
3 Bag { true, null } -> one(true) => true
4 Bag { true, null } -> one(false) => invalid
5 Bag { true, null } -> select(true) => invalid
6 Bag { true, null } -> select(oclIsUndefined()) => Bag { null }
```

Listing 1.1: Some example Iterator Expressions and their evaluation results.

operator `xor` can only be evaluated if both arguments are neither `null` nor `invalid`. Otherwise the operand will result in `invalid`.

Equality in Dresden OCL

OCL defines the two operators = and <> to compute equality of two given OCL expressions. Since OCL values can be both `null` or `invalid`, it is important to know how these operators behave when used with `null` and `invalid` values. According to the specification, comparing two OCL values results in `invalid` when one of the two values is either `null` or `invalid` [**?**, p. 195]. In some situations this can lead to problems during evaluation, e.g., when iterating on an OCL collection containing `null` values.

Thus in Dresden OCL, using the operators = and <> for comparison will always result in `true` or `false` as documented in Table 1.3. But please be aware, that this behaviour is only implemented for = and <>. Using other comparison operators such as <, <=, >, and >= for numeric values can result in `invalid`!

Collections

In Dresden OCL, collections can contain `null` values but cannot contain `invalid` values! If an `invalid` value is contained in a collection, the complete collection will be `invalid` as well.

Iterators

In Dresden OCL, iterators will result in a value as long as they can be computed, even if their collection contains `null` values. Listing 1.1 shows some examples for iterator expressions and their results in Dresden OCL. E.g., an `any` iterator will result in `true` as long as one element of its source's collection fulfills its condition even if other elements are `null` values. On the other hand, a `select` iterator will result in `invalid` if the condition for any element in its collection results in `null`. However, if the collection contains `null` values but the condition results in `true` or `false` (e.g., `oclIsUndefined()`), the result will not be `invalid`.

```
1 null + 2 => invalid
2 null.asSet() => Set { }
3 null.oclIsUndefined() => true
4 invalid.oclIsUndefined() => invalid
```

Listing 1.2: Evaluation of null values in Dresden OCL.

```
1 invalid + 2 => invalid
2 invalid.asSet() => invalid
3 invalid.oclIsInvalid() => true
4 undefined.oclIsInvalid() => false
```

Listing 1.3: Evaluation of invalid values in Dresden OCL.

A further difference between the OCL specification and the semantics in Dresden OCL exists for the `closure()` iterator. According to the specification, the `closure()` iterator computes its result based on a depth-first search [**?**, Sect. 7.6.5], whereas results in Dresden OCL are computed based on breadth-first search, due to performance reasons.

Handling of Null values

According to the OCL 2.3 specification [**?**], all operation calls invoked on `null` values will result in `invalid`. An exception is the operation `oclIsUndefined()` which results in `true` if its source is `null` and `false` otherwise. According to the OCL 2.3 specification the expression `invalid.oclIsUndefined()` will result in `true` as well. **In Dresden OCL,** `invalid.oclIsUndefined()` **results in** `invalid` **instead!** According to the OCL 2.3. specification, the implicit OCL operation `asSet()` can be invoked on `null` values and will result in an empty `Set`. Listing 1.2 shows some examples for evaluations on `null` values.

Handling of Invalid values

According to the OCL 2.3 specification [**?**], all operation calls invoked on `invalid` values will result in `invalid`. Exceptions are the operations `oclIsInvalid()` and `oclIsUndefined()`. The evaluation of `oclIsInvalid()` will result in `true` when invoked on `invalid` values. Otherwise it will result in `false`, even when invoked on `null` values. `invalid.asSet()` will result in `invalid` because sets cannot contain invalid values. Listing 1.3 shows some examples for evaluations on `invalid` values.

1.2 SUPPORTED MODELS IN DRESDEN OCL

Dresden OCL is adapted to multiple meta-models and thus allows the import of multiple kinds of models. Which kinds of models and which model file formats are supported by Dresden OCL is documented in this section. Possible use cases for the different models and model instances supported by Dresden OCL are given in Section 2.3.

1.2.1 EMF Ecore Models

Dresden OCL allows to import Ecore models modelled with the *Eclipse Modeling Framework (EMF)* [**?**]. Typically, Ecore models are stored in XMI files matching to the naming pattern `*.ecore`.

```
1  package tudresden.ocl20.pivot.examples.simple;
2
3  public class ModelProviderClass {
4
5    protected Person person;
6
7    protected Professor professor;
8
9    protected Student student;
10 }
```

Listing 1.4: Example for a Java Model Provider Class.

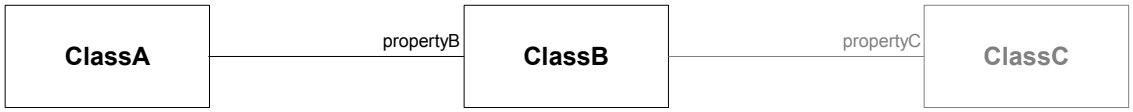

Figure 1.1: Transitive Import of Java Classes as a Model into Dresden OCL.

1.2.2 Java Classes as Models

Dresden OCL supports to import Java classes as models and thus to define OCL constraints directly on Java types and their fields and methods. If a Java class is imported into Dresden OCL, all types used inside the class' declaration are handled as also being a part of the imported model. If a `ClassA` is imported into Dresden OCL containing an attribute of the type `ClassB`, `ClassB` is also imported into Dresden OCL. `ClassB` could contain an operation having the return type `ClassC` and thus `ClassC` could be imported as well. After short consideration it should be clear that such a transitive mechanism could lead to a more or less complete import of the Java standard library into Dresden OCL. Thus, only the types that are used during OCL parsing and evaluation are imported into Dresden OCL.

If a `ClassA` is imported into Dresden OCL, the type of `ClassA` and all types that are directly related to this class are imported as well. E.g., a `ClassB` used in a property of `ClassA` would be imported, a related `ClassC` used in `ClassB` would not be imported (see Figure 1.1). If other types are requested during the work of tools on the imported model (e.g., if a tool requests the operation `ClassB.getOwnedOperations()`, all newly required types will be imported as well. This deferred transitive adaptation mechanism avoids that all types never used inside Dresden OCL are imported and adapted and cause overhead and maintenance problems. Listing 1.4 shows an example for a Java class provided with the *Simple Example* of Dresden OCL. The class contains properties having the types `Person`, `Professor` and `Student`. Thus, the imported model in Dresden OCL will contain a package `tudresden.ocl20.pivot.example.simple`, containing the four classes `ModelProviderClass` `Person`, `Professor` and `Student`.

To import Java classes into Dresden OCL two possibilities exist:

1. `*.class` files can be imported as a model. All directly referenced classes (either as types of properties and operations or their arguments) are imported as well. **Please be aware, that only byte code classes (`*.class`) and not source code classes `*.java` can be imported into Dresden OCL!**
2. Alternatively, the path leading to the Java class to be imported can be declared in a text file matching to the file naming pattern `*.javamodel`. This alternative was implemented to support the import of Java classes referencing other external Java classes provided as JARs. An example for a `*.javamodel` text file is shown in Listing 1.5. The first line references the `JarClassProvider.class` that shall be imported as a model. The second line references a

```
../bin/package1/package2/JarClassProvider.class
../lib/simple.jar
```

Listing 1.5: Example for a .javamodel File.

JAR file that contains classes whose types are referenced in the `JarClassProvider.class`. Both, the class and the JAR are referenced via relative URLs from the directory where the `*.javamodel` file is located in the file system. Further lines of the `*.javamodel` text file could be used to reference additional JARs to be imported as well. Please be aware, that only referenced classes from the JARs are imported into Dresden OCL. Again, further classes are imported when required during OCL parsing or evaluation.

1.2.3 MDT UML Class Diagrams

Dresden OCL allows to import UML class diagrams modelled with the *Eclipse Modeling Development Tools (Eclipse MDT)* [**?**]. Typically, MDT UML models are stored as XMI files matching to the naming pattern `*.uml`. Since many Eclipse-based modelling tools built on top of MDT UML, their models can be imported as well. Examples for such modelling tools are the *Graphical Modeling Framework (GMF)* UML class diagram editor and the *Topcased* UML class diagram editor.

1.2.4 XML Schemas as Models

Dresden OCL allows to import XML Schema Definitions (XSD) as models. Typically, XSDs are stored in XML files matching to the naming pattern `*.xsd`.

1.3 SUPPORTED MODEL INSTANCES IN DRESDEN OCL

Dresden OCL is adapted to and thus allows the import of multiple types of model instances. Which types of model instances are supported by Dresden OCL is documented in this section. Possible use cases for the different models and model instances supported by Dresden OCL are given in Section 2.3.

1.3.1 EMF Ecore-Based Model Instances

Besides the creation of meta-models or DSLs, the Eclipse Modeling Framework (EMF) allows the generation of simple model editors that can be used to model instances of the Ecore-based models created before. E.g., you can create a simple DSL using EMF. Afterwards you can model an instance of this DSL using an Ecore-generated model editor. Now, you can import your DSL as a model into Dresden OCL and you can parse OCL constraints defined on your DSL. To check these constraints on the created model instance, you have to import this instance into Dresden OCL as well.

Thus, Dresden OCL allows to import instances of Ecore-based models as model instances. Typically, the instances are stored as XMI files that match to a file naming pattern that ends with the name of your Ecore model. E.g., if you modelled an `myDSL.ecore` and created an instance of this model via an Ecore-generated model editor, the instance matches to the naming pattern `*.mydsl`.

```java
public class ModelInstanceProviderClass {

  /**
   * @return A {@link List} of {@link Object}s that are part of the
   *         {@link IModelInstance}.
   */
  public static List<Object> getModelObjects() {

    List<Object> result;
    result = new ArrayList<Object>();

    Person person1;
    person1 = new Person();
    person1.setName("Person Unspecific");
    person1.setAge(25);
    result.add(person1);

    /* Add further elements ... */

    return result;
  }
}
```

Listing 1.6: Example for a ModelInstanceProviderClass programmed in Java.

Besides complete models it can be useful to import single model instance elements into Dresden OCL when using Dresden OCL directly from another software application via its API. How this is possible is documented in Section 11.3.2. For Ecore model instances, this mechanism can be used to add single `EObjects` to a model instance in Dresden OCL.

1.3.2 Java Model Instances

Java objects can be regarded as instances of model elements described in class diagrams or similar models. Thus, the import of Java objects as model instances for OCL constraint interpretation is a common use case. Dresden OCL supports two possibilities to import Java objects into Dresden OCL.

1. It is possible to create a `ModelInstanceProviderClass` containing a static method called `getModelObjects()` that returns a `List` of `java.lang.Objects` that shall be imported as a model instance. Listing 1.6 shows an example of such a `ModelInstanceProviderClass`.

2. Similar to Ecore model instances, it is possible to add single `java.lang.Objects` to an existing model instance via Dresden OCL's API at runtime as documented in Section 11.3.2.

1.3.3 XML Model Instances

Dresden OCL supports to import XML files as model instances. Although XML files cannot contain executable code, their elements can be used as data to be verified by structural OCL integrity constraints. Typically XML files conform to the file name matching pattern `*.xml`.

1.4 SUMMARY

This chapter documented which version of OCL is supported by Dresden OCL. Furthermore, supported types of models, and model instances were presented. The following chapter will explain how to install and use Dresden OCL.

2 GETTING STARTED WITH DRESDEN OCL

Chapter written by Claas Wilke and Michael Thiele

This chapter generally introduces into *Dresden OCL*. Dresden OCL is based on a *Pivot Model* developed by Matthias Bräuer [**?**] which is shortly explained in Chapter 7. Further information about Dresden OCL is available at the project's website [**?**].

This chapter explains the installation of Dresden OCL and how to load a model, an instance of such a model, and OCL constraints defined on such a model into Dresden OCL. Besides the Eclipse distribution, Dresden OCL can also be used as a stand-alone Java Library. If you plan to use the stand-alone distribution you can skip this chapter and continue with Chapter 9. However, this chapter explains the basic concepts of Dresden OCL. Although you cannot use the shown GUI wizards and browsers when using the stand-alone version, this chapter can be helpful to understand the terms used in and the mechanisms provided by Dresden OCL.

2.1 HOW TO INSTALL DRESDEN OCL

The following different possibilities exist to install Dresden OCL within Eclipse.

1. You may install Dresden OCL using the *Eclipse Marketplace Client*.
2. You may install Dresden using the update site available at [**?**],
3. You may checkout and run the source code distribution from the SVN available at [**?**].

This section will explain all three possibilities.

2.1.1 Installing Dresden OCL using the Eclipse Marketplace Client

Since Eclipse 3.6, the new Eclipse Marketplace Client allows easy installation of Eclipse-based tools such as Dresden OCL.

To install Dresden OCL via the Eclipse Marketplace Client, select the menu option *Help -> Eclipse Marketplace...*. Probably you have to select a marketplace catalog afterwards. If so, select the *Eclipse Marketplace* catalog and proceed.

Type `Dresden OCL` into the search text field and press the return key. Select Dresden OCL from the search results and click the *Install* button (cf. Fig. 2.1). Afterwards, click through the installation dialog and Dresden OCL will be installed. Finally you have to restart your Eclipse distribution to complete the installation.

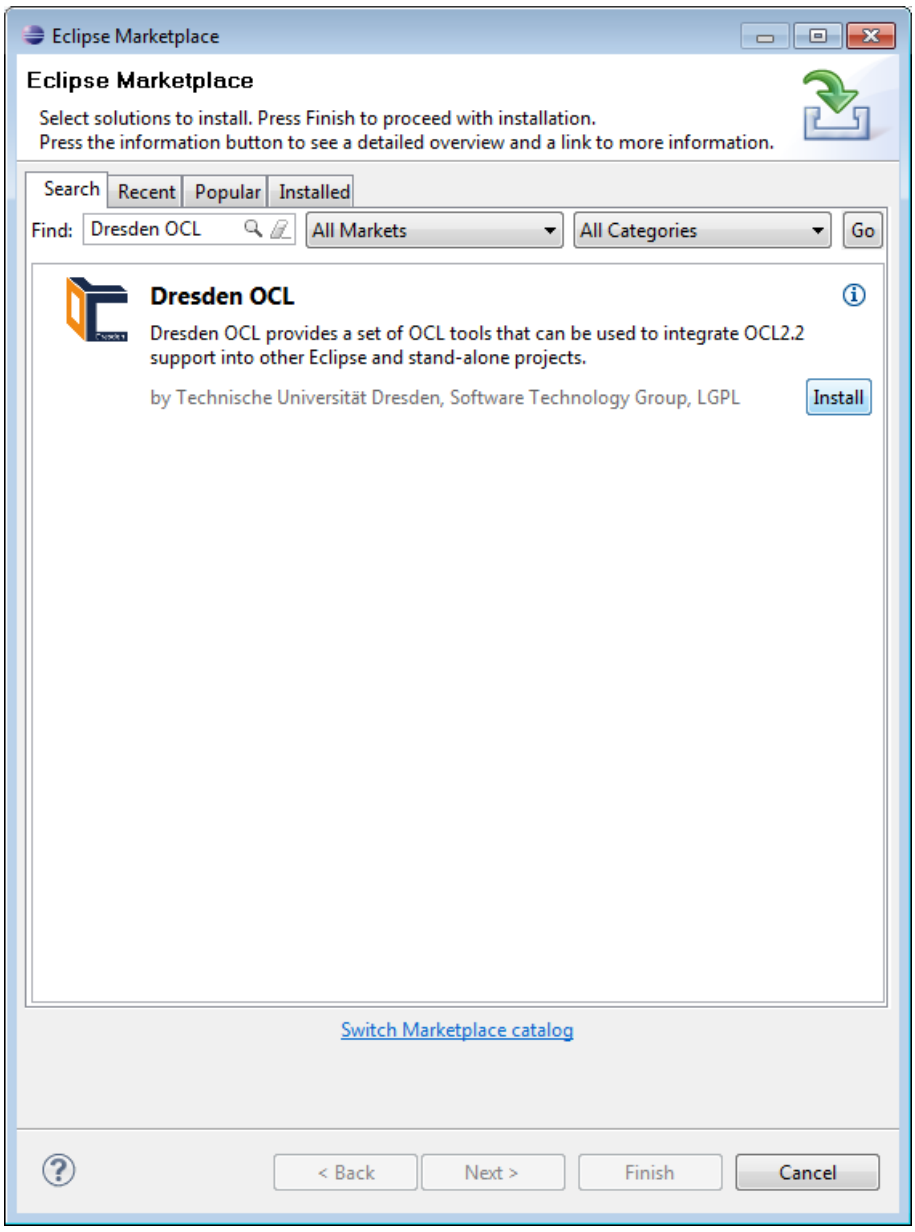

Figure 2.1: Installing Dresden OCL using the Marketplace Client.

2.1.2 Installing Dresden OCL using the Eclipse Update Site

To install Dresden OCL 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://www.dresden-ocl.org/update/site.xml` and click the *Add...* button (cf. Fig. 2.2). In the new opened window you can additionaly enter a name for the update site (cf. Fig. 2.3).

Now you can select the features of Dresden OCL which you want to install. Select them and click the *Next >* button (cf. Fig. 2.4). An overview on all features of Dresden OCL can be found in Table 2 in the appendix of this manual. Follow the wizard and agree with the user license. Then

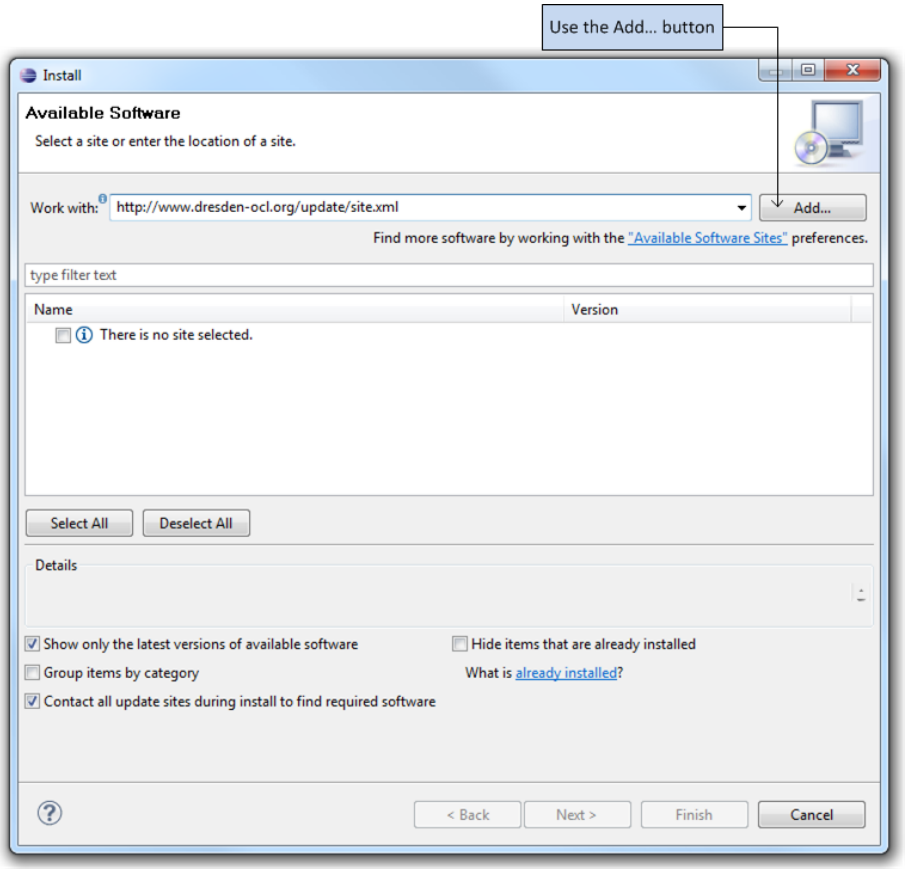

Figure 2.2: Adding an Eclipse Update Site (Step 1).

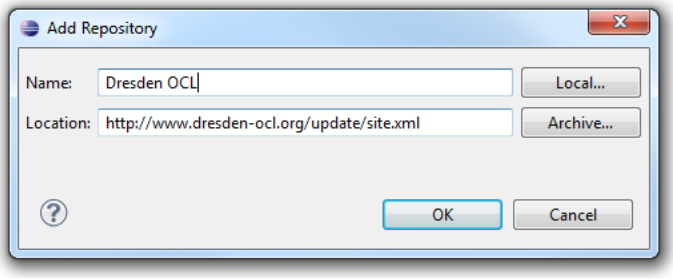

Figure 2.3: Adding an Eclipse Update Site (Step 2).

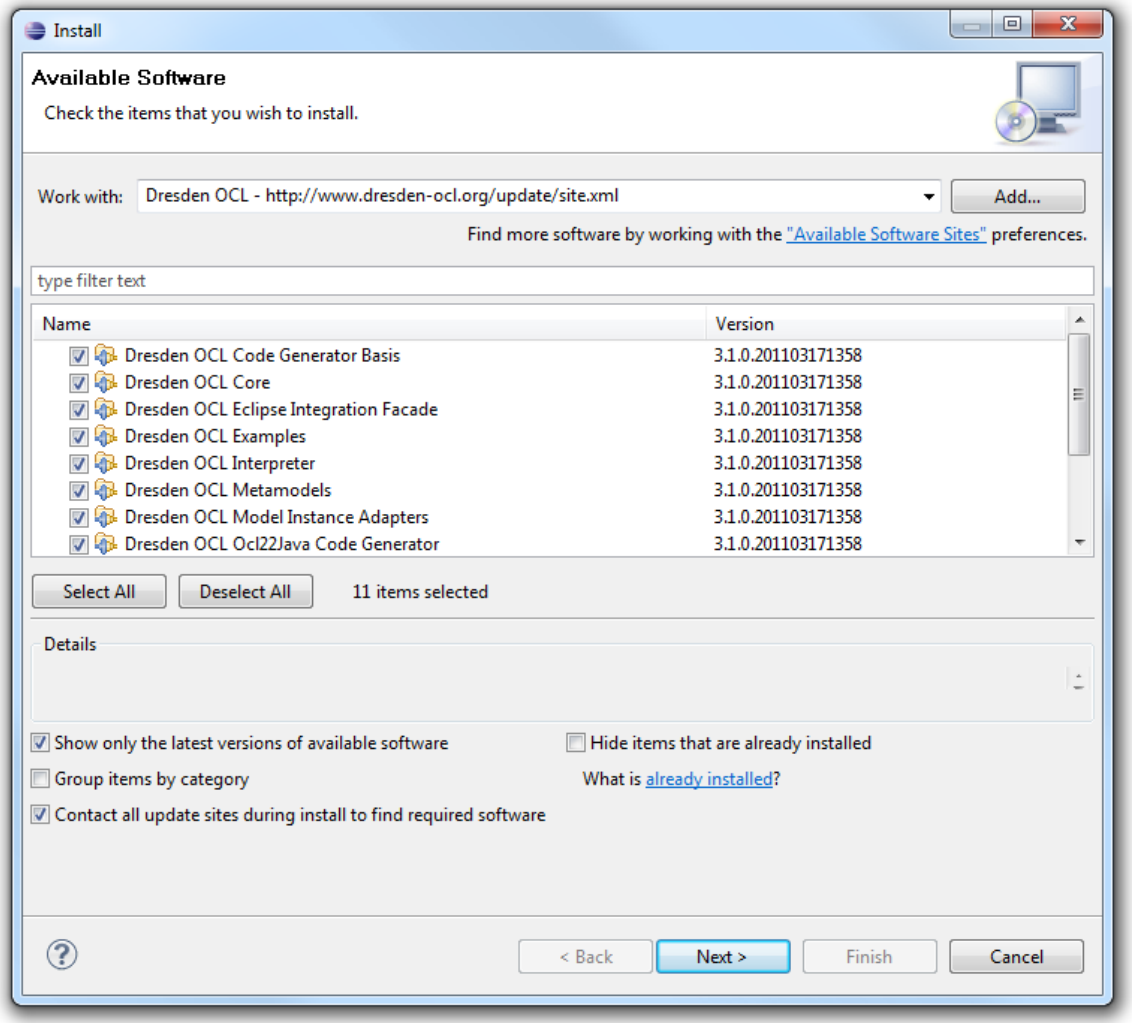

Figure 2.4: Selecting features of Dresden OCL.

Dresden OCL will be installed. Afterwards, you should restart your Eclipse application to finish the installation.

2.1.3 Importing Dresden OCL from the SVN

To use Dresden OCL by checking out the source code from the SVN you need to install an SVN client. In the following the *Eclipse Subversive* plug-in is used.

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 `http://svn-st.inf.tu-dresden.de/svn/dresdenocl/` (cf. Fig. 2.5).

After clicking 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 (cf. Fig. 2.6).

2.1.4 Which Plug-ins do you need at least?

Often people wonder which plug-ins of Dresden OCL they require for a minimum installation. The answer to this question depends on the things you plan to do with Dresden OCL. Table 2 in the appendix of this manual shows a list of the currently existing plug-ins of Dresden OCL, that are related to different features. You should install at least the *Core* feature, at least one meta-model of the *Meta-Models* feature, and the complete *Parser* feature. The *Interpreter*, the *OCL22Java*m and the *OCL2SQL* 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 this manual. We recommend to install all provided features.

2.1.5 Building the OCL2 Parser

The new Dresden OCL parser/editor is partially written in Scala. In order to build the sources of the parser without having to have the `Scala IDE` installed, Dresden OCL comes with various *Ant* scripts that compile the Scala code to byte code.

After a checkout, the build script should be called automatically. Be aware that the compilation might take a while to finish. If other projects that depend on the parser like the facade still do not compile correctly, try to perform a *refresh* on the plug-ins that contain Scala code.

If the *Ant* script is not invoked automatically, you can call it either be cleaning the `tudresden.ocl20.pivot.language.ocl.staticsemantics` plug-in or by running the *Ant* task `clean all` of the same plug-in.

Each plug-in that contains Scala code (`org.kiama.attribution`, `tudresden.ocl20.pivot.language.ocl.semantics`, `tudresden.ocl20.pivot.language.ocl.staticsemantics` and `tudresden.ocl20.pivot.pivotmodel.semantics`) contains a *build.xml* file that comes with three targets: `clean all` to clean the selected project and all dependent projects, `clean` to only clean the selected project and `compile` to simply compile changes that have been made since the last build.

In either case, you have to run the *ANT* scripts in the same *JRE* as Eclipse. Figures 2.7 and 2.8 show how to achieve this. 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.

If you want to make changes to the static semantics evaluation of the parser you should consider installing the *Scala IDE* from `http://www.scala-lang.org/scala-eclipse-plugin`. Be aware that the Scala code is version 2.7.7 which is not compatible with Scala 2.8 and therefore you cannot use the current *Scala IDE* which supports only Scala 2.8. The *Scala IDE* runs with Eclipse 3.5 and 3.6.

In order to use the Scala compiler of the IDE, you have to go to the *Properties* of each Scala plug-in, select the tab *Builders*, check the *Scala Builder* and possibly uncheck the *Ant* script for building.

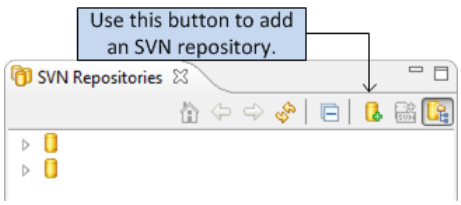

Figure 2.5: Adding an SVN repository.

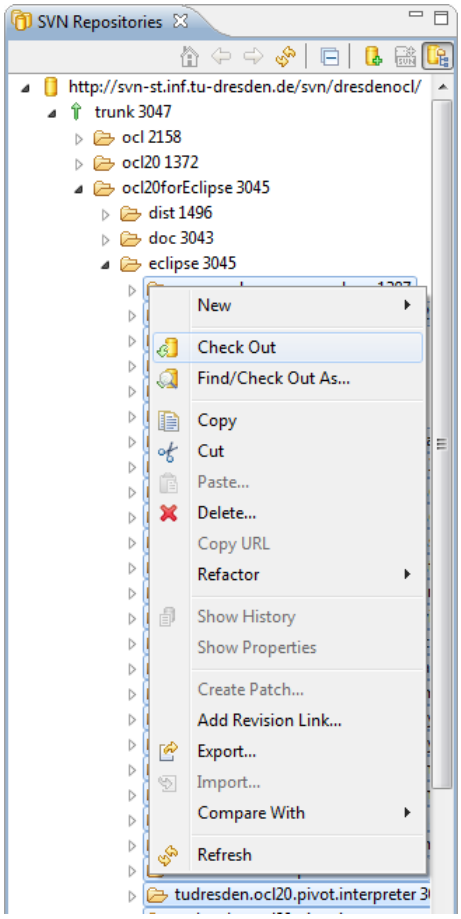

Figure 2.6: Checkout of Dresden OCL's plug-in projects.

2.2 LOADING MODELS, MODEL INSTANCES AND CONSTRAINTS

If you installed the Dresden OCL using the market place client or update site, you can execute the toolkit within 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.

2.2.1 The Simple Example

The use of Dresden OCL is explained using the *Simple Example* which is located in the plug-in `tudresden.ocl20.pivot.examples.simple`. Figure 2.9 shows a class diagram of the Simple Example.

Dresden OCL 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 OCL is listed in Table 4 in the appendix of this manual. The Simple Example can be used with two different meta-models. These are *UML 2* (based on *Eclipse MDT UML*) and *Java*.

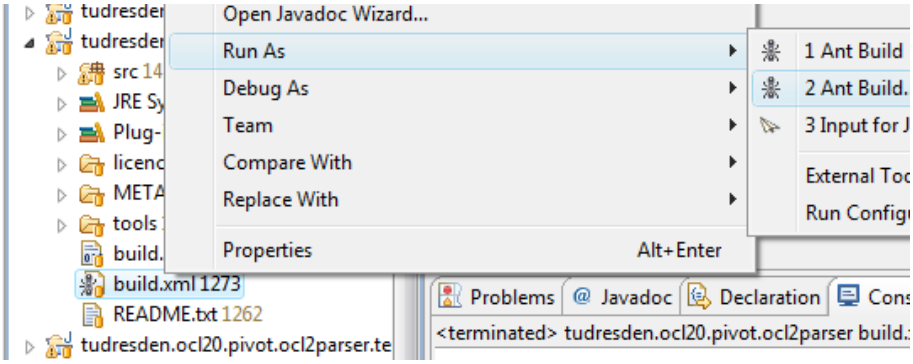

Figure 2.7: Executing the OCL2 Parser build script.

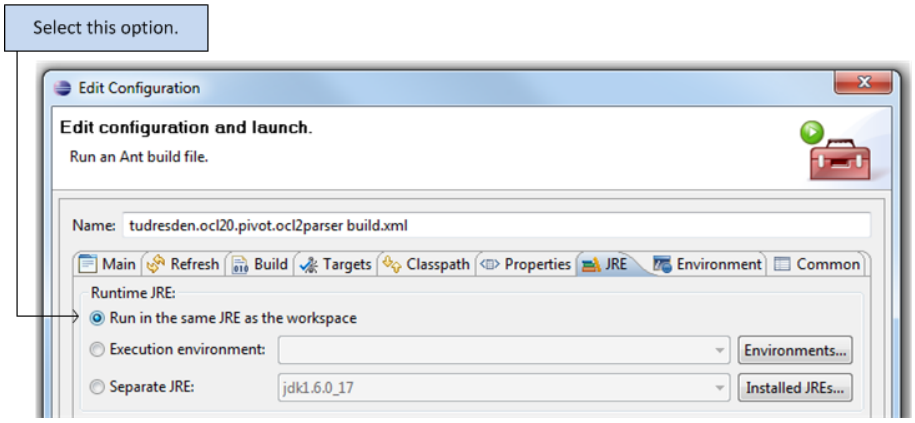

Figure 2.8: Settings of the JRE for the Ant build script.

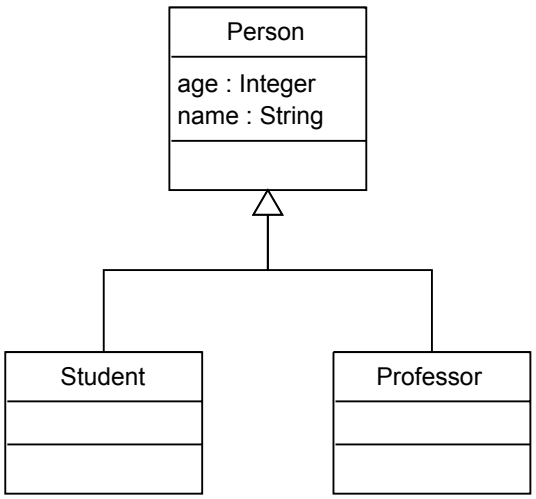

Figure 2.9: A class diagram describing the Simple Example model.

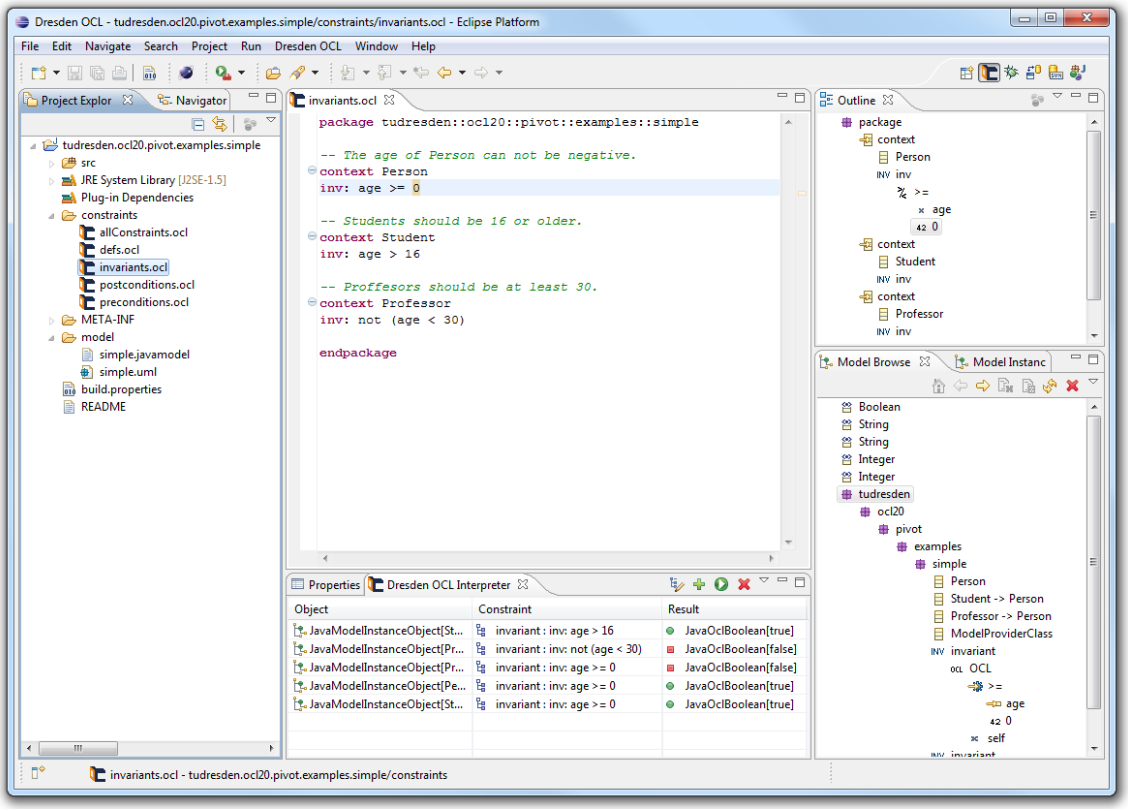

Figure 2.10: The Dresden OCL Perspective.

2.2.2 Dresden OCL Perspective

Dresden OCL provides its own perspective within Eclipse that contains all views and editors provided with Dresden OCL. To ease the work with Dresden OCL, you should now switch to the Dresden OCL perspective. Select the menu option *Window -> Open Perspective -> Other ...* and select the perspective *Dresden OCL* (cf. Fig. 2.10)

On the left hand side the perspective contains the *Project Explorer* of Eclipse to manage different projects. The right hand side contains the *Outline View* for opened OCL files. Below, the *Model Browser* and *Model Instance Browser* of Dresden OCL allow to explore models and instances imported into Dresden OCL. At the bottom of the perspective the *OCL Intepreter* is located. The center of the perspective contains the *OCL Editor* of Dresden OCL that allows to edit and parse OCL files for an opened model. How to use the tools provided with Dresden OCL is explained in the following.

2.2.3 Loading a Model

For this tutorial you first have to load a model into Dresden OCL. To ease the use of the Simple Example project, this project should be imported into the *Workspace* first. Select the menu option *File -> New -> Other* and select the option *Dresden OCL Examples -> Simple Example* within the new opened window (cf. Fig 2.11). Click the *Finish* button to import the project into your workspace. Afterwards, the workspace should contain the Simple Example project as shown in Figure 2.10, left hand side.

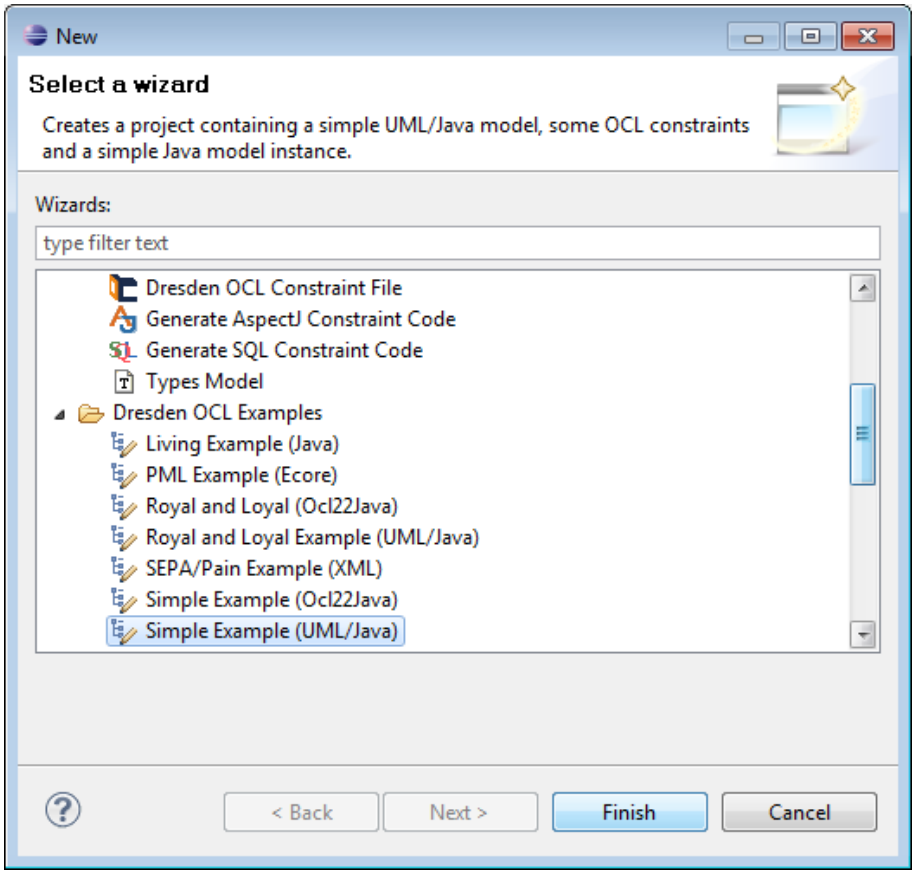

Figure 2.11: Importing the Simple Example project.

Now you can import the model into Dresden OCL. Select the `model/simple.uml` file in the *Project Explorer* and open the context menu (right mouse click). Select the menu option *Dresden OCL > Load Model* (cf. Fig. 2.12). In the opened wizard you have to select the meta-model UML2 and click the *Finish* button (cf. Fig. 2.13).

Figure 2.14 shows the imported 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 imported into Dresden OCL (cf. Fig. 2.15). With the two circled arrows icon you can reload a model into Dresden OCL, with the red *X* you can close the currently selected model.

2.2.4 Loading a Model Instance

After loading a model, you can load an instance of this model using another wizard. The model instance is required to interpret OCL constraints on elements instantiating the classes described in the opened model. Which kinds of model instances are supported in Dresden OCL is documented in Section 1.3. Since the Simple Example provides a Java model instance, we now have to select a `class` file. `class` files are not displayed in the *Project Explorer*. Thus, switch to the *Navigator* that should be visible at the left hand side of the Dresden OCL perspective as well and select the file `bin/tudresden/ocl20/pivot/examples/simple/instance/ModelInstanceProviderClass.class` of the Simple Example in the *Project Explorer*. Open the context menu and select the menu option *Dresden OCL > Load Model Instance* (cf. Fig. 2.16). In the opened wizard you have to select a model for which the model instance shall be loaded and the

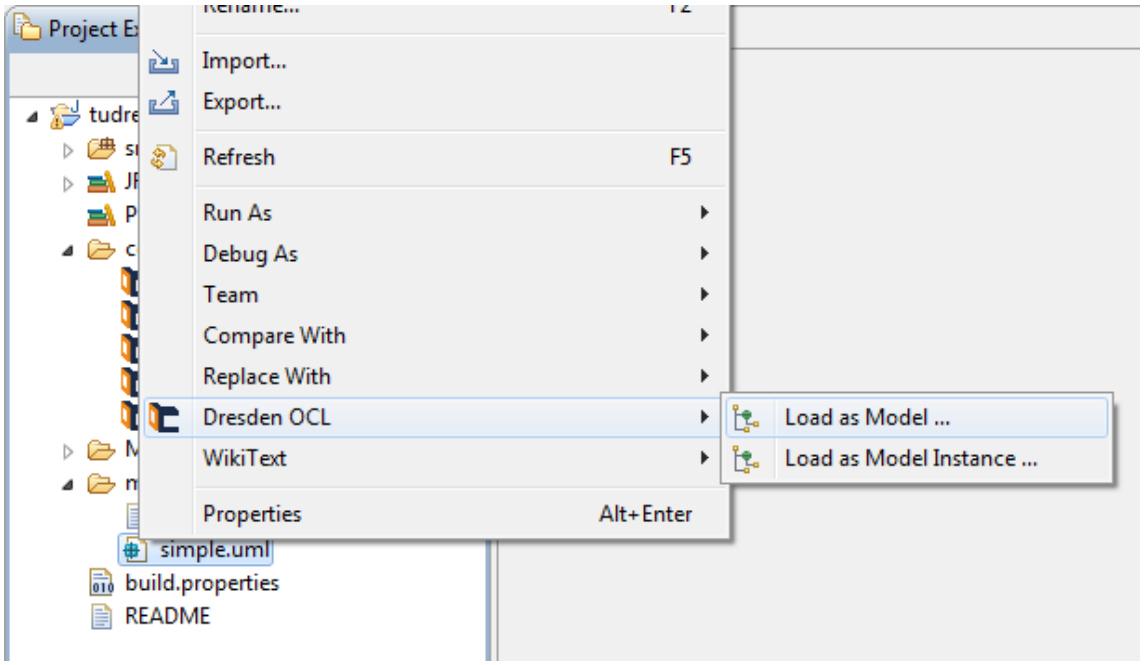

Figure 2.12: Loading a Model.

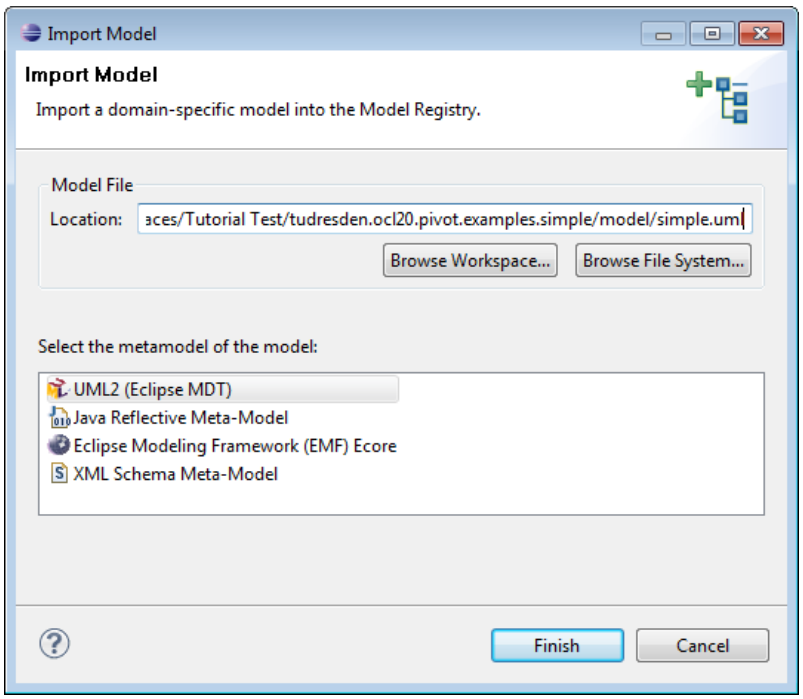

Figure 2.13: Loading a Model.

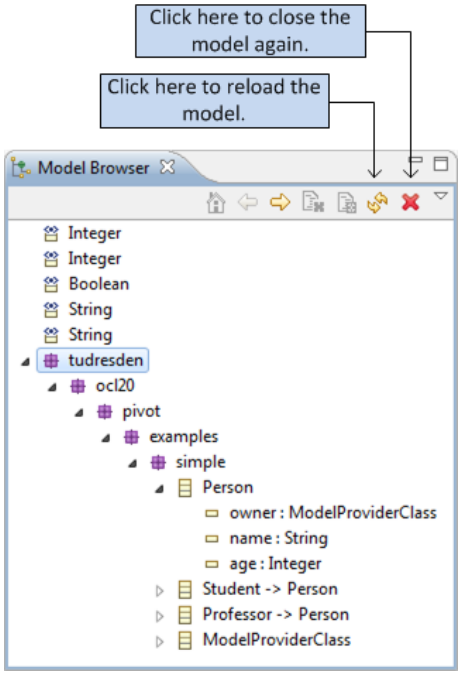

Figure 2.14: The Simple Example model within the Model Browser.

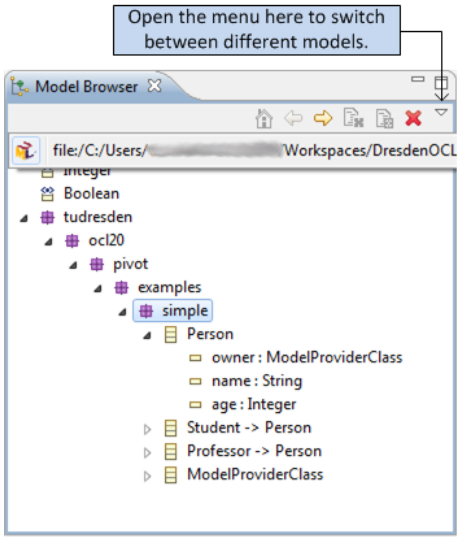

Figure 2.15: You can switch between different Models using the little triangle.

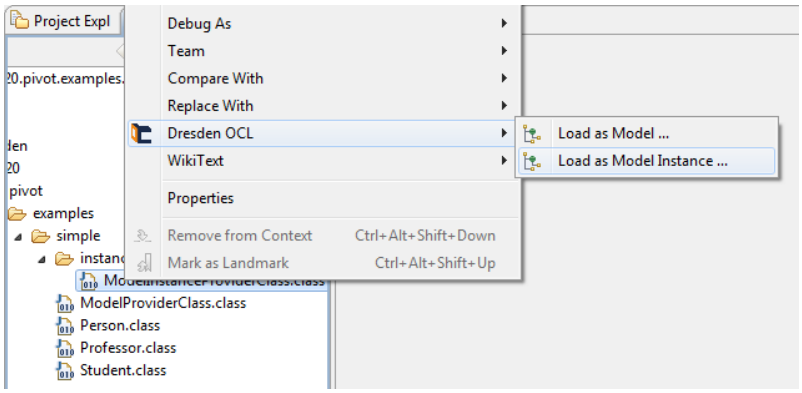

Figure 2.16: Loading a Simple Model Instance.

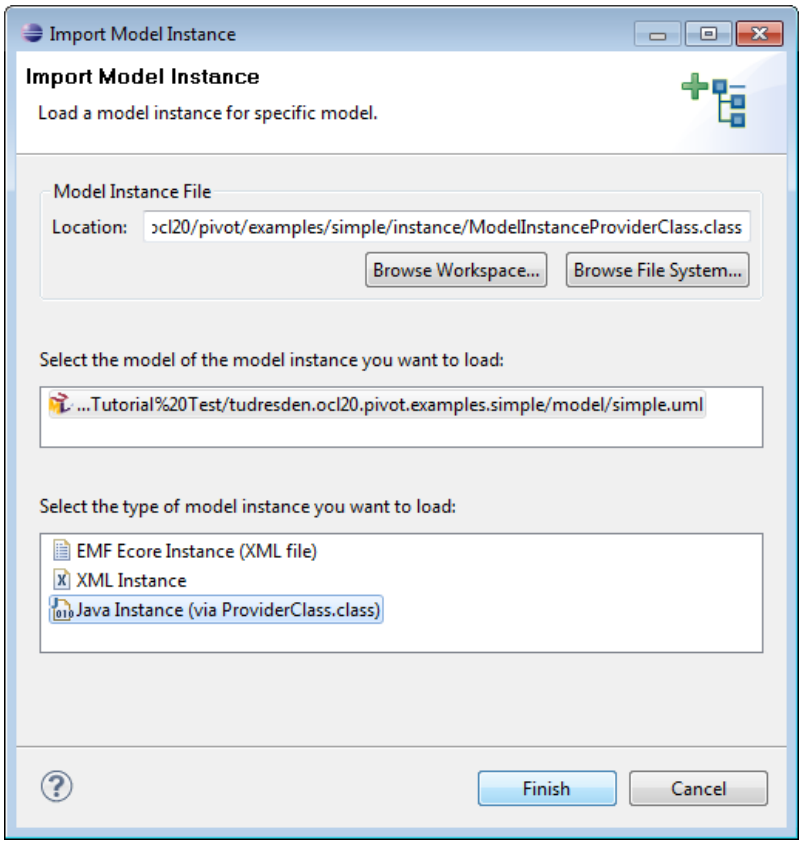

Figure 2.17: Loading a Simple Model Instance.

type of model instance you want to load (cf. Fig. 2.17). Select the *Java Instance* type and click the *Finish* button.

Figure 2.18 shows the imported model instance. 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 model in the model browser, you also switch the pool of model instances available in the model instance browser.

2.2.5 Parsing OCL Expressions

Any file with the file extension `.ocl` can be opened with the *Dresden OCL Editor*. Once opened, syntactic checks are performed to analyse whether the given file contains valid OCL code. If currently there is no active model selected in the *Model Browser*, the editor will fail to perform the static semantics analysis and will yield that there is no active model. You can load a model and then re-parse the OCL file by changing the OCL file (e.g., by introducing and immediately deleting a whitespace character).

The editor/parser will automatically add parsed constraints to the model as well as *definitions* to the appropriate classes. You can inspect the changes on the model in the *Model Browser*. Note that the *definitions* and constraints are not added to your model – they belong to the view of Dresden OCL on the model. The result can be seen in Figure 2.19. You also can remove parsed constraints from the model which is shown in Figure 2.20.

2.3 POSSIBLE USE CASES OF DRESDEN OCL USING DIFFERENT MODELS AND MODEL INSTANCES

Dresden OCL can be used in the context of different kind of models and instances and even at different modeling layers. This section tries to name some prominent examples for possible use cases of Dresden OCL w.r.t. different kinds of models and instances. Readers who are not interested in these details are encouraged to skip this section.

In general, Dresden OCL supports the use of OCL at two different levels: First, OCL constraints can be defined on a metamodel and evaluated on instances of this metamodel (i.e., models). In this context, OCL constraints are often called *Well-Formedness Rules (WFRs)*. Second, OCL constraints can be defined on a model and evaluated on instances of this model (i.e., runtime objects or data). These constraints are often called *Business Rules (BRs)*. Examples for both use cases are shown in Table 2.1 and shortly explained in the following.

2.3.1 Use Cases of Dresden OCL for Well-Formedness Rules

Dresden OCL supports different scenarios, where OCL rules can be specified on metamodels as WFRs. The mose prominent scenarios are shortly explained below.

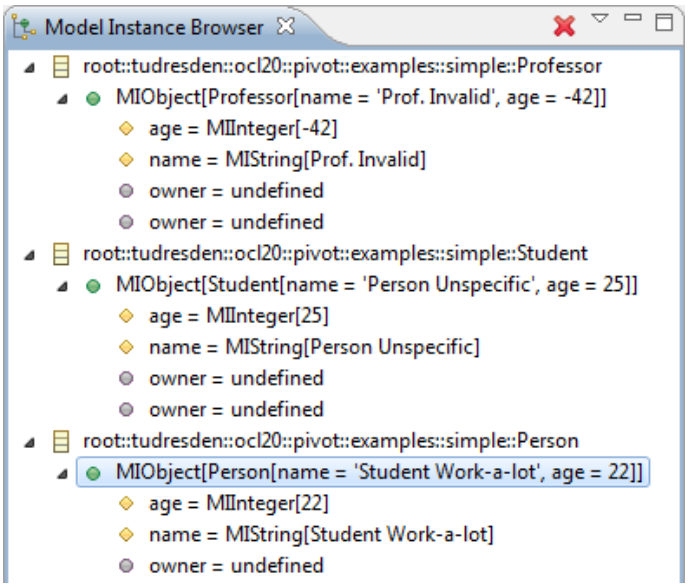

Figure 2.18: A simple model instance in the Model Instance Browser.

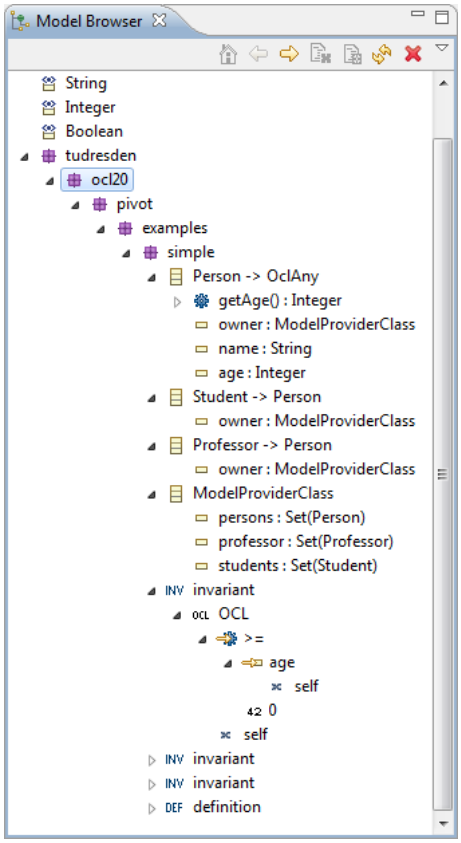

Figure 2.19: Parsed expressions and the model in the Model Browser.

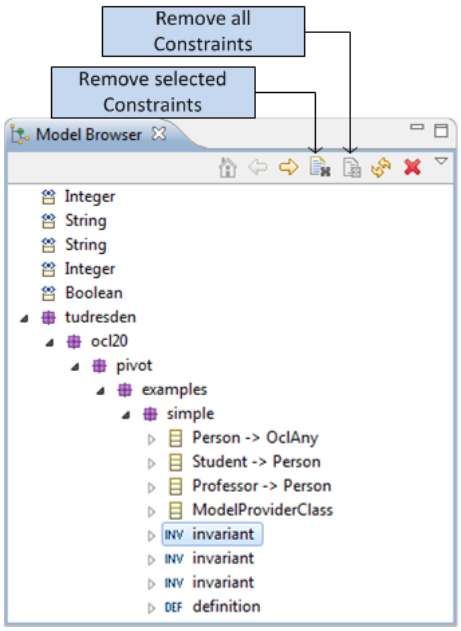

Figure 2.20: How to remove Constraints from a Model again.

WFR Specification and Evaluation	
EMF/Ecore-based model, e.g., `mydsl.ecore`	EMF/Ecore-based instance (model), e.g., `model01.mydsl`
UML metamodel, `uml.ecore` of MDT/UML	UML instance (model), e.g., `model01.uml`
BR Specification and Evaluation	
UML-based model, e.g., `model01.uml`	Java-based instance (runtime objects), e.g., `Instance01.class`
Java classes, e.g., `Model01.class`	Java-based instance (runtime objects), e.g., `Instance01.class`
XML Schema (XSD), e.g., `mySchema.xsd`	XML instance (data), e.g., `Instance01.xml`

Table 2.1: Different possible use cases of Dresden OCL.

WFRs for EMF/Ecore Models

EMF/Ecore is often used as a metamodeling language to develop Domain-Specific Languages (DSLs). To specify OCL constraints on an Ecore-based metamodel, you import the model file (e.g., `mydsl.ecore`) as a model into Dresden OCL using the model importer for EMF/Ecore-based models. Afterwards, you can specify OCL constraints using the OCL parser/editor of Dresden OCL. You can import istances of your DSL into Dresden OCL using the model instance importer for EMF/Ecore-based instances (e.g., `model01.mydsl`) to interprete the specified constraints on them.

WFRs for the UML Metamodel

Another common use case is the specification of OCL constraints on the UML metamodel and their evaluation for instances of the metamodel (i.e., UML models). You can do this by importing the EMF/Ecore-based UML-metamodel of the Eclipse Modeling Development Tools (MDT). You can find the required `uml.ecore` within the Eclipse plug-in `org.eclipse.uml.uml` of Eclipse MDT.

Please note, when importing this metamodel into Dresden OCL, you have to import using the EMF/Ecore model importer and not a UML model importer (since the UML metamodel was modelled in EMF)! Afterwards, you can import a UML model as an instance of the metamodel to evaluate constraints on it (e.g., `model.uml`). Again, you have use the EMF/Ecore model instance importer.

2.3.2 Use Case of Dresden OCL for Busines Rules

Besides the evaluation of WFRs, multiple use cases for the evaluation of Business Rules (BRs) are supported and explained below.

BRs for UML models

A common use case is the definition of OCL constraints on a UML model (e.g., `model01.uml`), typically containing a class model. You can import UML class models into Dresden OCL using the corresponding model importer. For OCL evaluation a possible model instance is a set of runtime objects by using a Java class and the Java model instance importer (e.g., `Instance01.class`). Details how a Java class usable as a Java model instance must be implemented are documented in Section 1.3.2.

BRs for Java Classes

Another possible use case is the use of a set of Java classes as a model for OCL constraint specification (e.g., `Model01.class`). A Java class can be imported as a model using the Java model importer. Details how the class is imported as a model are documented in Section 1.2.2. Again, a typical model istance would be a set of Java objects as mentioned above (e.g., `Instance01.class`).

BRs for XML Schemata

Dresden OCL supports the definition of OCL rules on **XSDs!** (XSDs) as well (e.g., `mySchema.xsd`) using the XSD model importer. Obviously an XML file would be an appropriate model instance, using the XML model instance importer (e.g., `Instance01.xml`).

2.3.3 Further Use Cases

Of course, the use cases presented above are not a complete list of all possible use cases. Theoretically, every supported type of model importer can be combined with every type of model instances. And besides, you can even define your own importers for further use cases, if necessary. How to adapt Dresden OCL to other types of models and instances is documented in the Chapters 10 and 11.

2.4 SUMMARY

This chapter described how to use Dresden OCL. It was explained how to install the plug-ins of Dresden OCL. Afterwards, the import of models, model instances and OCL constraints into Dresden OCL was explained.

Now, the imported models can be used with the tools provided by Dresden OCL. For example you can use the *OCL Interpreter* to interpret OCL constraints for a given model and model instance (as explained in Chapter 3) or you can use the *OCL22Java Code Generator* to generate *AspectJ* code for a loaded model and OCL constraints (as explained in Chapter 4). How the *OCL2SQL Code Generator* can be used to generated SQL schema and integretiy views is documented in Chapter 5.

If you do not want to use Eclipse, but still want to interpret OCL constraints or generate AspectJ code, you can use Dresden OCL as a stand-alone library outside of Eclipse. A detailed description on how to do this is given in Chapter 9.

3 OCL INTERPRETATION

Chapter written by Claas Wilke

This chapter describes how the OCL Interpreter provided with Dresden OCL can be used and how to trace such interpretations. How to install and run Dresden OCL and how to load models and OCL constraints was explained in Chapter 2. If you are not familiar with such basic uses of Dresden OCL, read Chapter 2 first.

3.1 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 OCL can be found in Table 4 in the appendix of this manual. An introduction into the Simple Example can be found in Section 2.2.1. 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 run this tutorial, import the Simple Example as explained in Section 2.2.1. Figure 3.1 shows the *Project 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 3.1 shows the constraints defined in the constraint file.

First, the constraint file defines three simple invariants that ensure 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`.

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`.

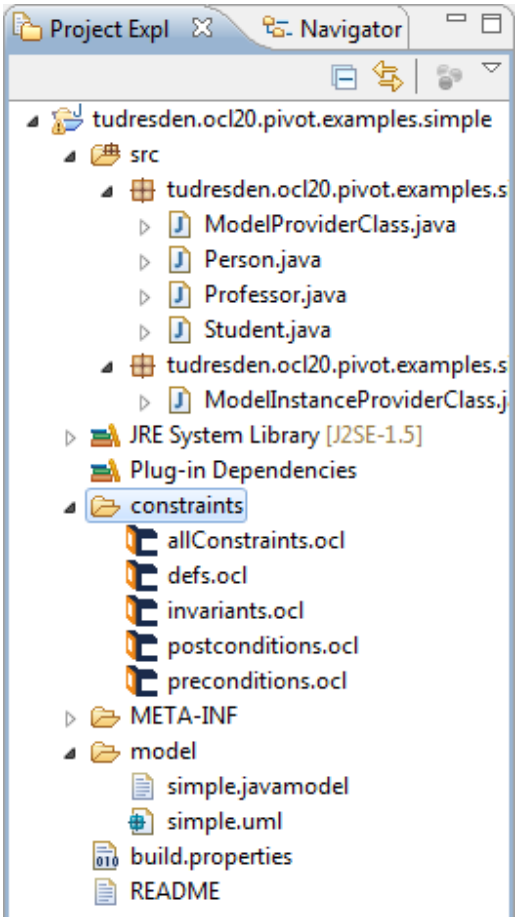

Figure 3.1: The Project Explorer containing the Project which is required to run this Tutorial.

3.2 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 model import wizard of Dresden OCL to import the model. This procedure is explained in Section 2.2.4. 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 open the constraint file `constraints/allConstraints.ocl` containing the constraints we want to interpret. Afterwards, the *Model Browser* should look like illustrated in Figure 3.2 and the *Model Instance Browser* should look like shown in Figure 3.3.

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.

3.3 OCL INTERPRETATION

Now we can start the interpretation. To open the OCL Interpreter we use the menu option *Dresden OCL2 > Open OCL2 Interpreter*. The *OCL Interpreter View* should now be visible (cf. Fig. 3.4).

```
1  -- The age of Person can not be negative.
2  context Person
3  inv: age >= 0
4
5  -- Students should be 16 or older.
6  context Student
7  inv: age > 16
8
9  -- Proffesors should be at least 30.
10 context Professor
11 inv: not (age < 30)
12
13 -- Returns the age of a Person.
14 context Person
15 def: getAge(): Integer = age
16
17 -- Before returning the age, the age must be defined.
18 context Person::getAge()
19 pre: not age.oclIsUndefined()
20
21 -- The result of getAge must equal to the age of a Person.
22 context Person::getAge()
23 post: result = age
```

Listing 3.1: The Constraints contained in the Constraint File.

By now, the *OCL Interpreter View* does not contain any result. Besides the results table, the view provides four buttons to control the OCL Interpreter. The buttons are shown in Figure 3.5. 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 *OCL Interpreter View*. The functionality of the buttons will be explained below.

3.3.1 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* and push the *Interpret* button. The result of the interpretation is now shown in the *OCL Interpreter View* (see Figure 3.6).

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`.

Besides invariants, OCL allows 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`. Now,

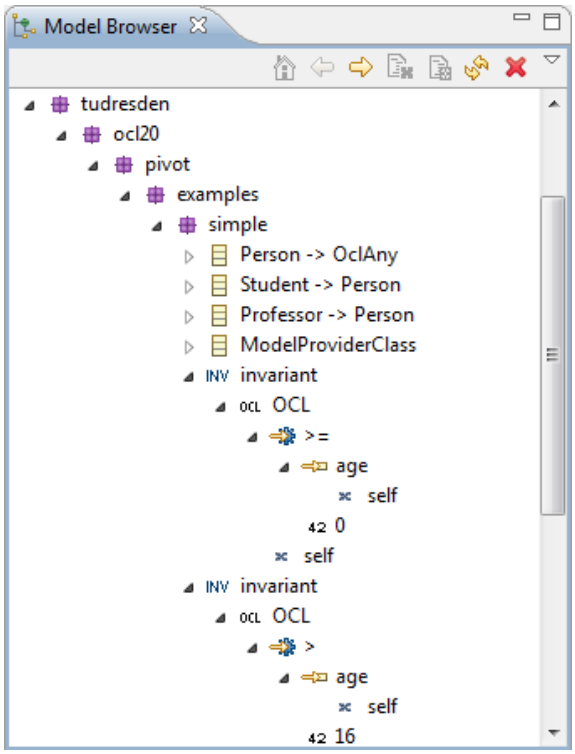

Figure 3.2: The Model Browser containing the Simple Model and its Constraints.

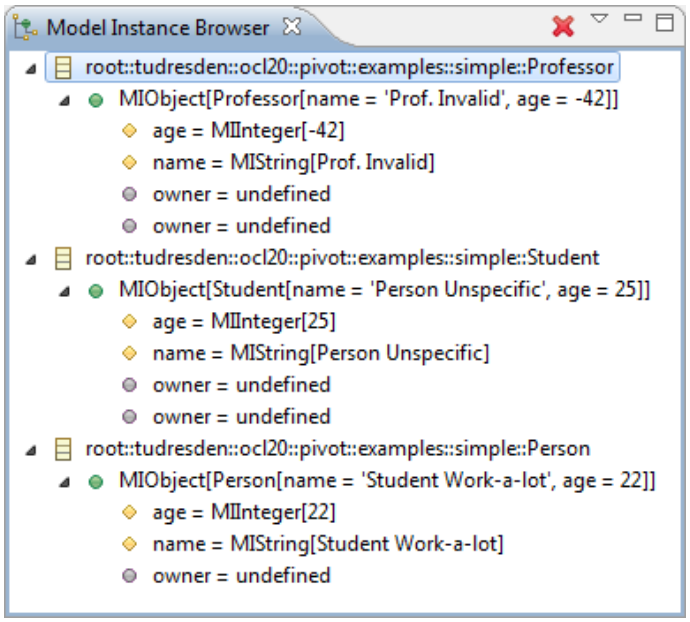

Figure 3.3: The Model Instance Browser containing the Simple Model Instance.

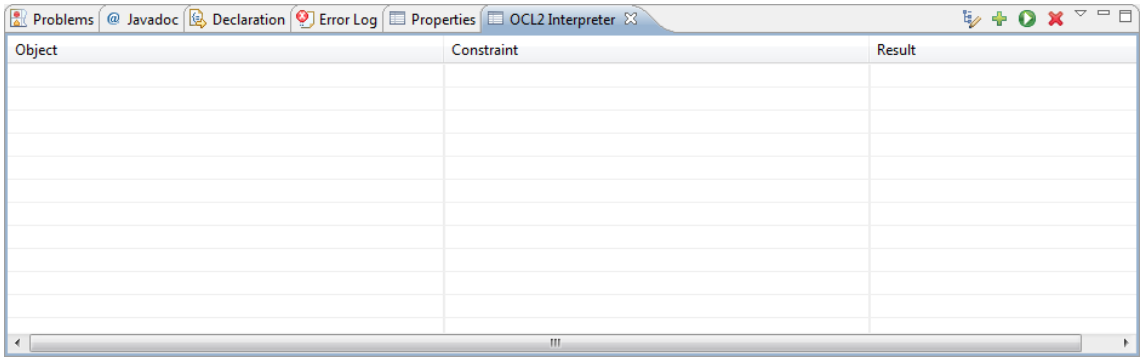

Figure 3.4: The OCL2 Interpreter View containing no results.

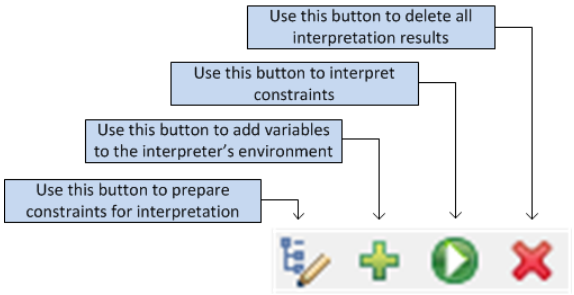

Figure 3.5: The Buttons to Control the OCL2 Interpreter.

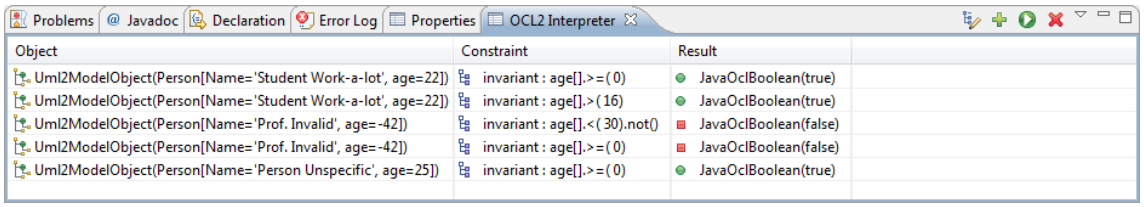

Figure 3.6: The results of the three Invariants for all Model Instance Elements.

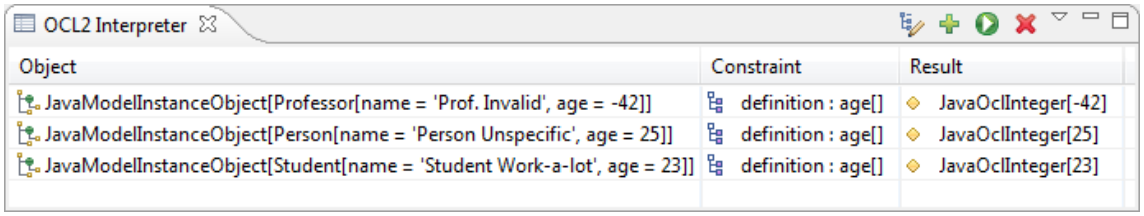

Figure 3.7: The results of the Definition for all Model Instance Elements.

we want to interpret this definition constraint. We select the constraint in the *Model Browser* and click the *Interpret* button. The result of the interpretation is shown in Figure 3.7. The interpretation finishes for all three instances successfully because the attribute `age` has been set for all three instances.

3.3.2 Adding Variables to the Environment

When interpreting OCL constraints from the GUI, we have to add further context information to interpret some pre- and postconditions. For example, the postcondition contained in the constraint file 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 *OCL 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 (cf. Fig. 3.8).

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 click the *OK* button (cf. Fig. 3.9). 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 3.10. 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`.

Other examples requiring manual addition of context information are pre- and postconditions that are defined on operations containing arguments. Listing 3.2 shows a precondition that is defined on an operation `setAge(arg01)`. If the argument `arg01` is referred during interpretation, the interpreter has to know the value of the argument. Thus, we would have to add the value of `arg01` before the constraint's interpretation manually as shown for the `result` variable.

3.3.3 Preparation of Constraints

The interpretation of some postconditions requires a preparation of the Interpreter's environment before the operation defined in the context of the postcondition is invoked. Listing 3.3 shows such a postcondition. The postcondition is defined on an operation `birthdayHappens()` that increments the `age` of a `Person`. The postcondition checks, whether the `age` was incremented or not. Thus, the Interpreter has to store the value of `age` before the operation `birthdayHappens()` is invoked. Therefore, the Interpreter View provides a button to prepare constraints (the first button from the left). If you want to interpret such postconditions, first select and prepare your constraint. The value of `age@pre` is then stored in the Interpreter's environment. Then you can invoke your model instance's operation (which is quite complicate from the GUI). Afterwards you can interpret the postcondition.

The preparation of postconditions is not that useful when interpreting constraints from the GUI of Dresden OCL because you cannot invoke your operations here to alter your model instance's state. Nevertheless, like the possibility to add variables to the Interpreter's environment you

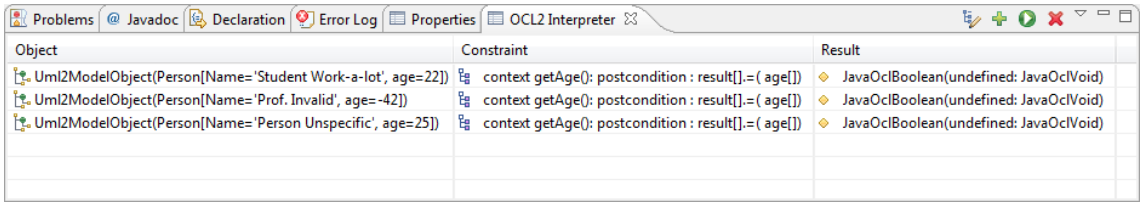

Figure 3.8: The results of the Postcondition without preparing the Result Variable.

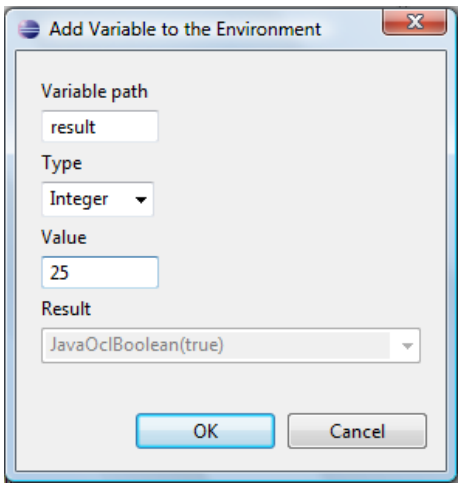

Figure 3.9: The Window to add new Variables to the Environment.

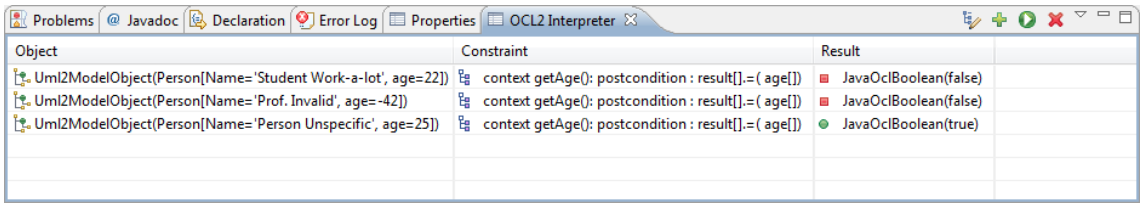

Figure 3.10: The Results of the Postcondition with Result Variable Preparation.

```
1 -- arg01 must be defined.
2 context Person::setAge(arg01: Integer)
3 pre: not arg01.oclIsUndefined()
```

Listing 3.2: An example Precondition defined on an Operation with Argument.

```
1 -- age must be incremented by one.
2 context Person::birthdayHappens()
3 post: age = age@pre + 1
```

Listing 3.3: An example Postcondition that must be prepared.

can prepare postconditions from the GUI. These operations are much more useful when using Dresden OCL via its API and using the OCL Interpreter to check OCL constraints during the runtime of other software. Then you can prepare constraints before methods are invoked and check postconditions afterwards, e.g., by using *Aspect-Oriented Programming (AOP)*.

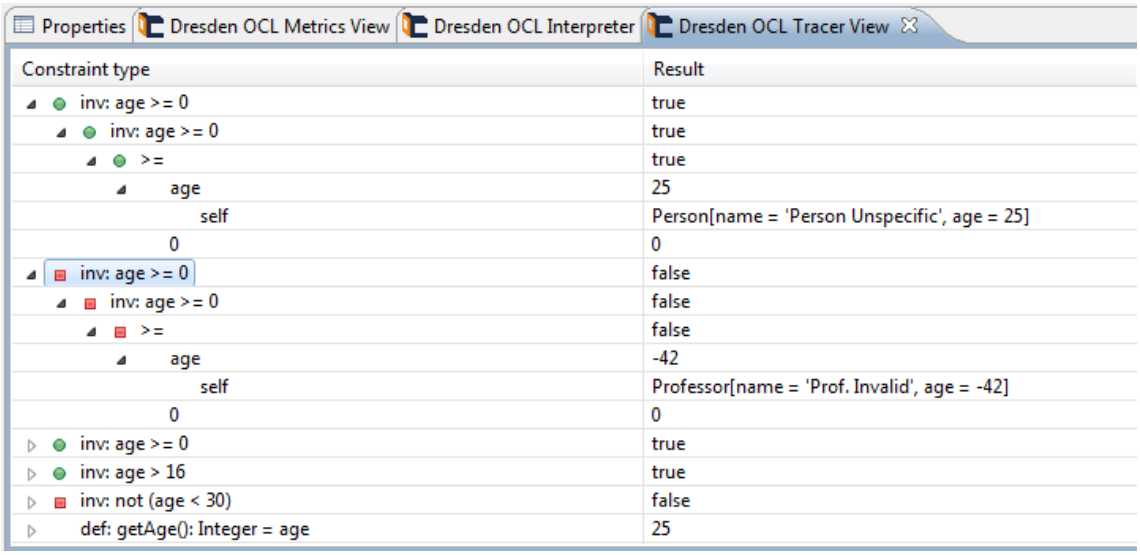

Figure 3.11: The results of the interpretation in the tracer view.

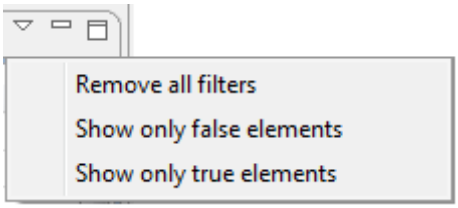

Figure 3.12: The menu to apply filters to the tracer.

3.3.4 Tracing

Tracing is applied for logging and visualizing the execution path of the interpretation of an OCL expression. Thus, it can be used to debug OCL expressions if interpretation do not correspond to expected results. To receive interim results during interpretation of OCL expressions, open the Dresden OCL Tracer from the menu option *Dresden OCL > Open Dresden OCL Tracer* before you start the interpretation. Once opened, the execution path of an OCL expression will be logged and shown in the OCL *Tracer View* (see Figure 3.11).

When debugging OCL expressions users typically intend to find failures. Thus, the OCL *Tracer View* also provides a menu to filter for special criteria, e.g. all `true` or `false` results (see Figure 3.12).

3.4 SUMMARY

This chapter described how OCL constraints can be interpreted using the OCL Interpreter of Dresden OCL. The preparation and interpretation of constraints has been explained, the addition of new variables to the Interpreter Environment has been shown. The tracing of expressions has been demonstrated. Besides the use of the Interpreter via Dresden OCL's GUI, you can also invoke the Interpreter via Dresden OCL's API. The easiest way to connect to Dresden OCL is via its *Facade* providing interfaces for all services of Dresden OCL. How to use Dresden OCL's facade is documented in Chapter 8.

4 ASPECTJ CODE GENERATION

Chapter written by Claas Wilke

This chapter describes how the Java Code Generator *OCL22Java* provided with Dresden OCL can be used. A general introduction into Dresden OCL can be found in Chapter 2. A detailed documentation of the development of OCL22Java can be found in the Minor Thesis (Großer Beleg) of Claas Wilke [**?**].

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 [**?**].

4.1 CODE GENERATOR PREPARATION

This chapter uses the *Simple Example* which is provided with Dresden OCL and has been introduced in Section 2.2.1. Since for this scenario we require two different projects and not just the one imported in Chpater 2, we use another example wizard now. By using the menu option *File -> New -> Dresden OCL Examples -> Simple Example (Ocl22Java)* the two projects are imported into the Eclipse workspace. Afterwards, the workspace should contain two projects named `tudresden.ocl20.pivot.examples.simple` and `...simple.ocl22javacode` (cf. Fig. 4.1).

The first project provides a model file which contains the simple class diagram which has been explained in Section 2.2.1 (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 4.1 shows one invariant that is contained in the constraint file. 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.

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

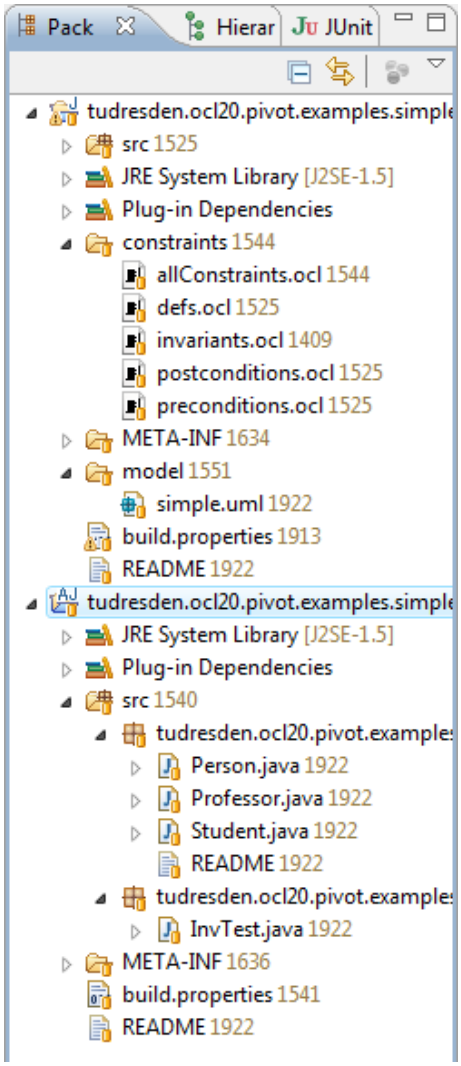

Figure 4.1: The two Projects which are required to run this Tutorial.

Package Explorer and selecting the menu item *Run as -> JUnit Test*. The test case fails because the exception is not thrown (cf. Fig. 4.3). 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.

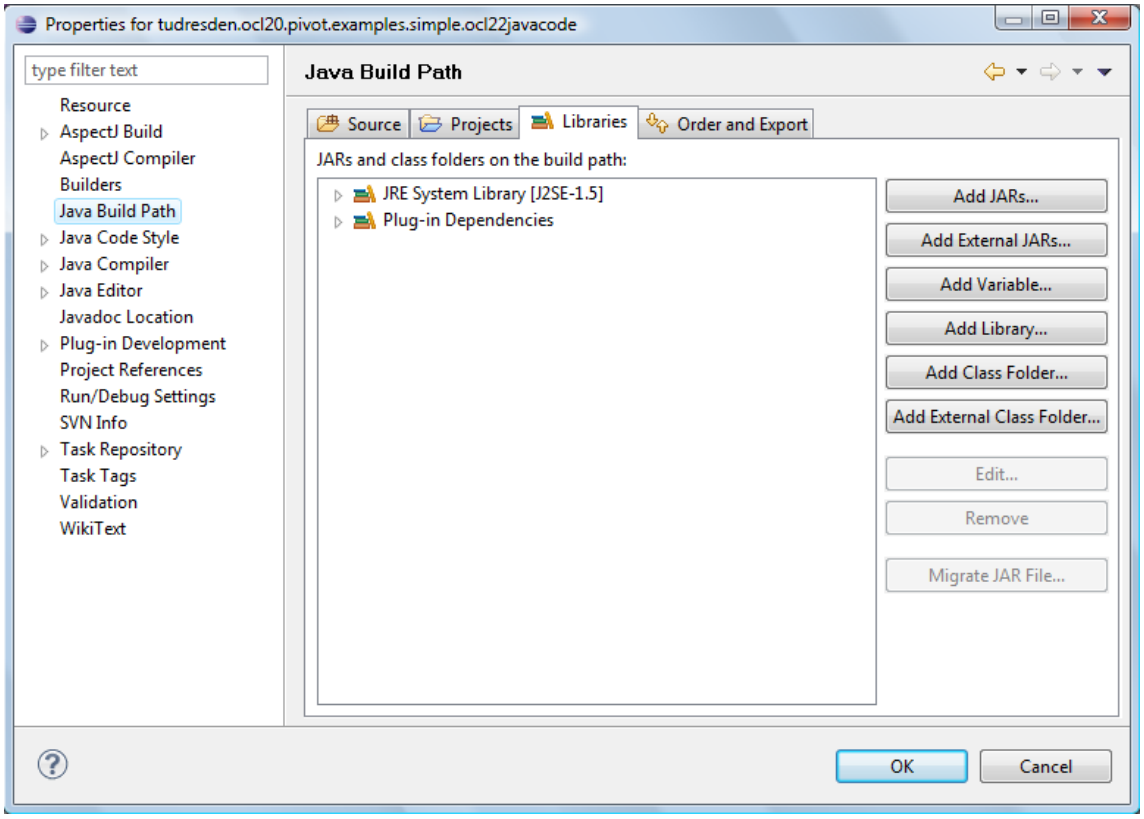

Figure 4.2: Adding a new Library to the Build Path.

```
1 -- The Age of a Person can not be Negative.
2 context Person
3 inv: age >= 0
```

Listing 4.1: A Simple Invariant.

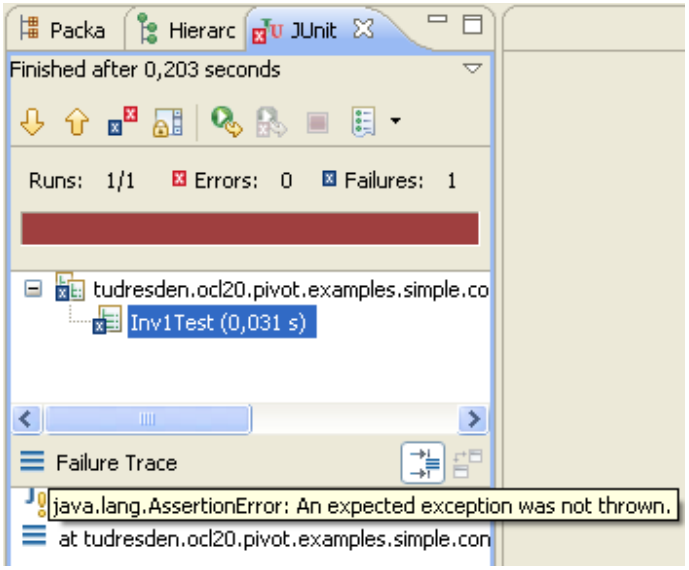

Figure 4.3: The Result of the JUnit Test Case.

4.2 CODE GENERATION

To prepare the code generation we have to import the model `model/simple.uml` into the *Model Browser*. We use the model import wizard of Dresden OCL to import the model. This procedure is explained in Chapter 2. Afterwards, we have to open the constraint file `constraints/invariant.ocl`. After the importation, the *Model Browser* should look like illustrated in Figure 4.4. Now we can start the code generation.

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

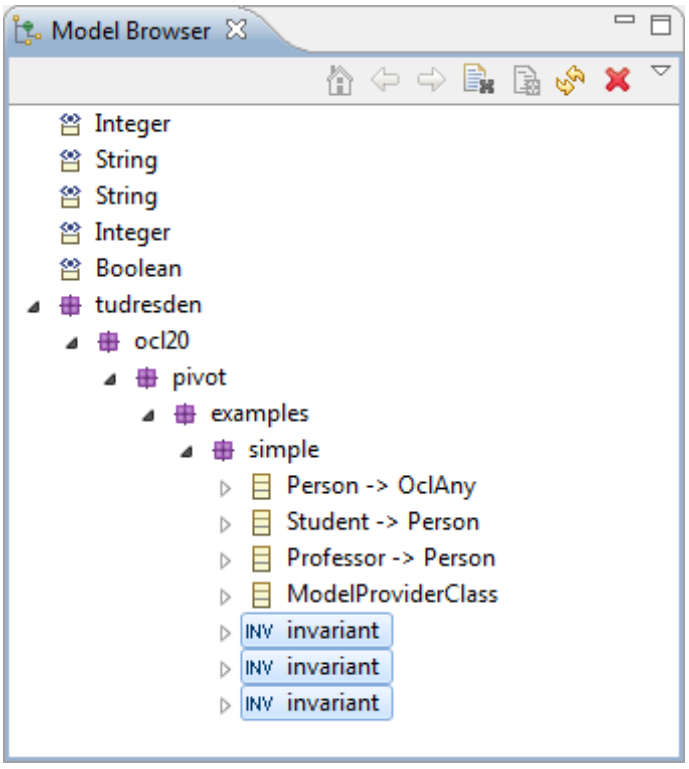

Figure 4.4: The Model Browser containing the Simple Model and its Constraints.

4.2.1 Selecting a Model

A wizard opens and we have to select a model for code generation (cf. Fig. 4.5). We select the `simple.uml` model and click the *Next* button.

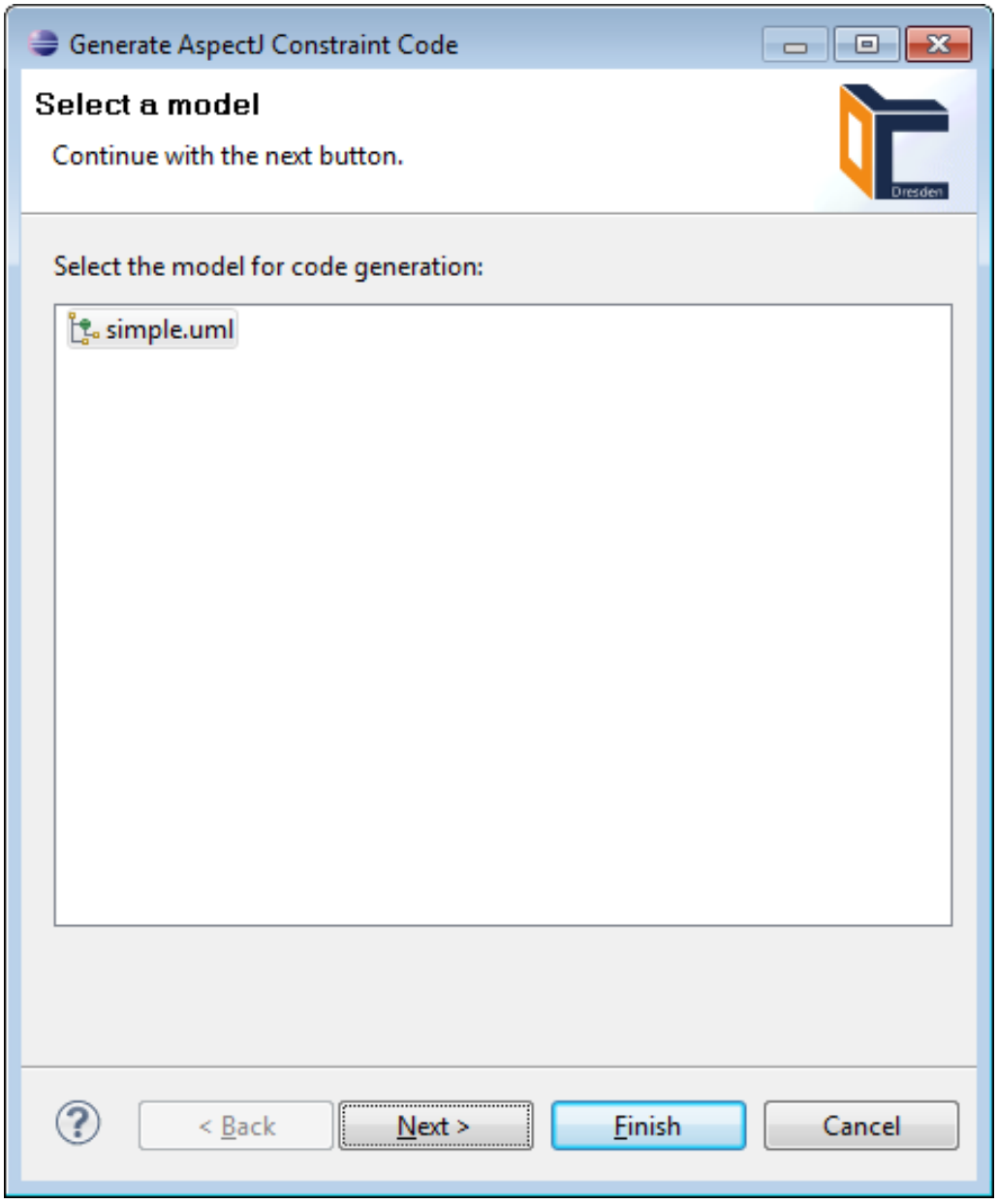

Figure 4.5: The first Step: Selecting a Model for Code Generation.

4.2.2 Selecting Constraints

As a second step we have to select the constraints for which 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 click the *Next* button (cf. Fig. 4.6).

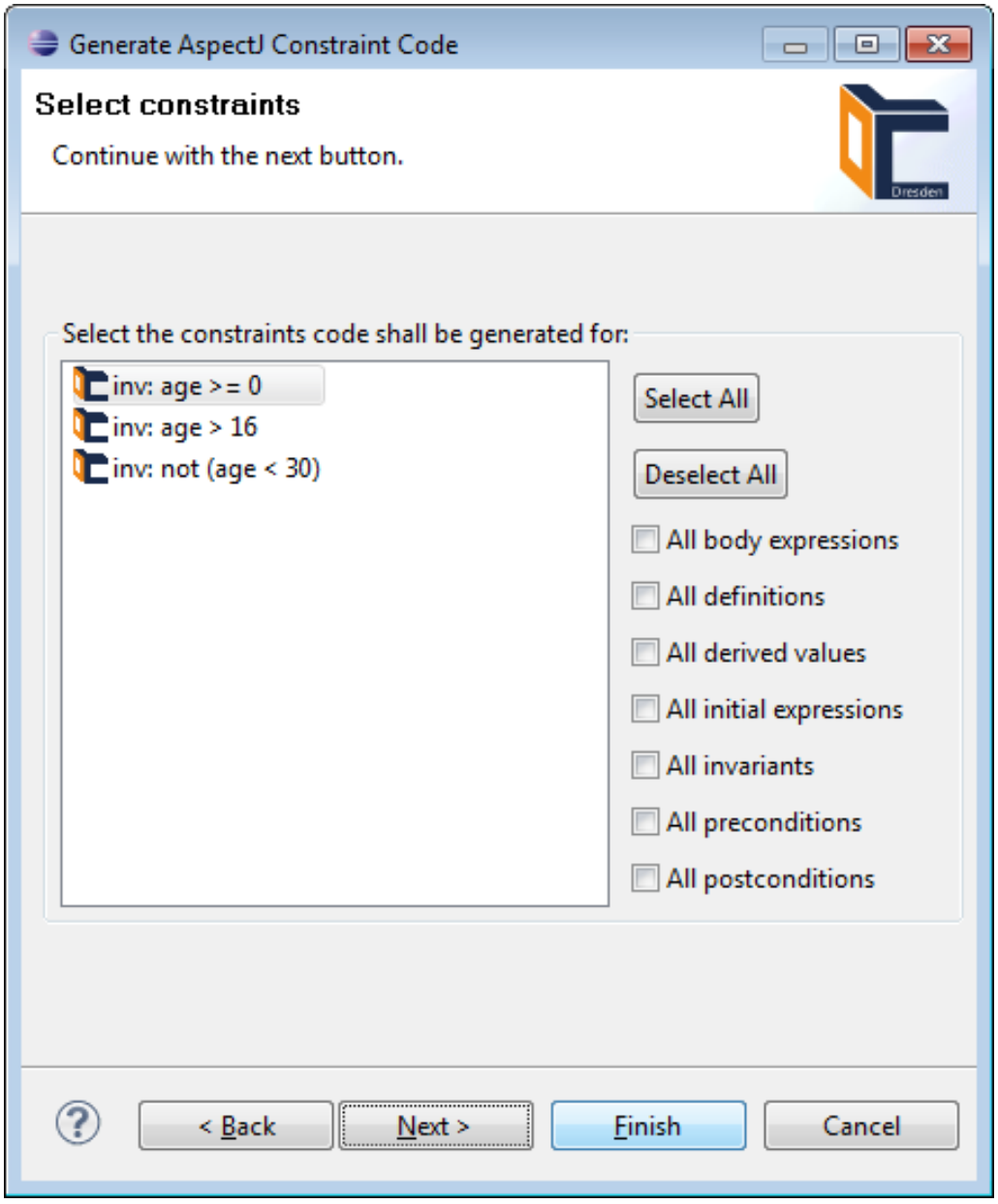

Figure 4.6: The second Step: Selecting Constraints for Code Generation.

4.2.3 Selecting a Target Directory

Next, we have to select a target directory into that the generated code shall be stored. We select the source directory of our second project (which is `tudresden.ocl20.pivot.examples.simple.ocl22javacode/src`) (cf. Fig. 4.7). 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 click the *Next* button.

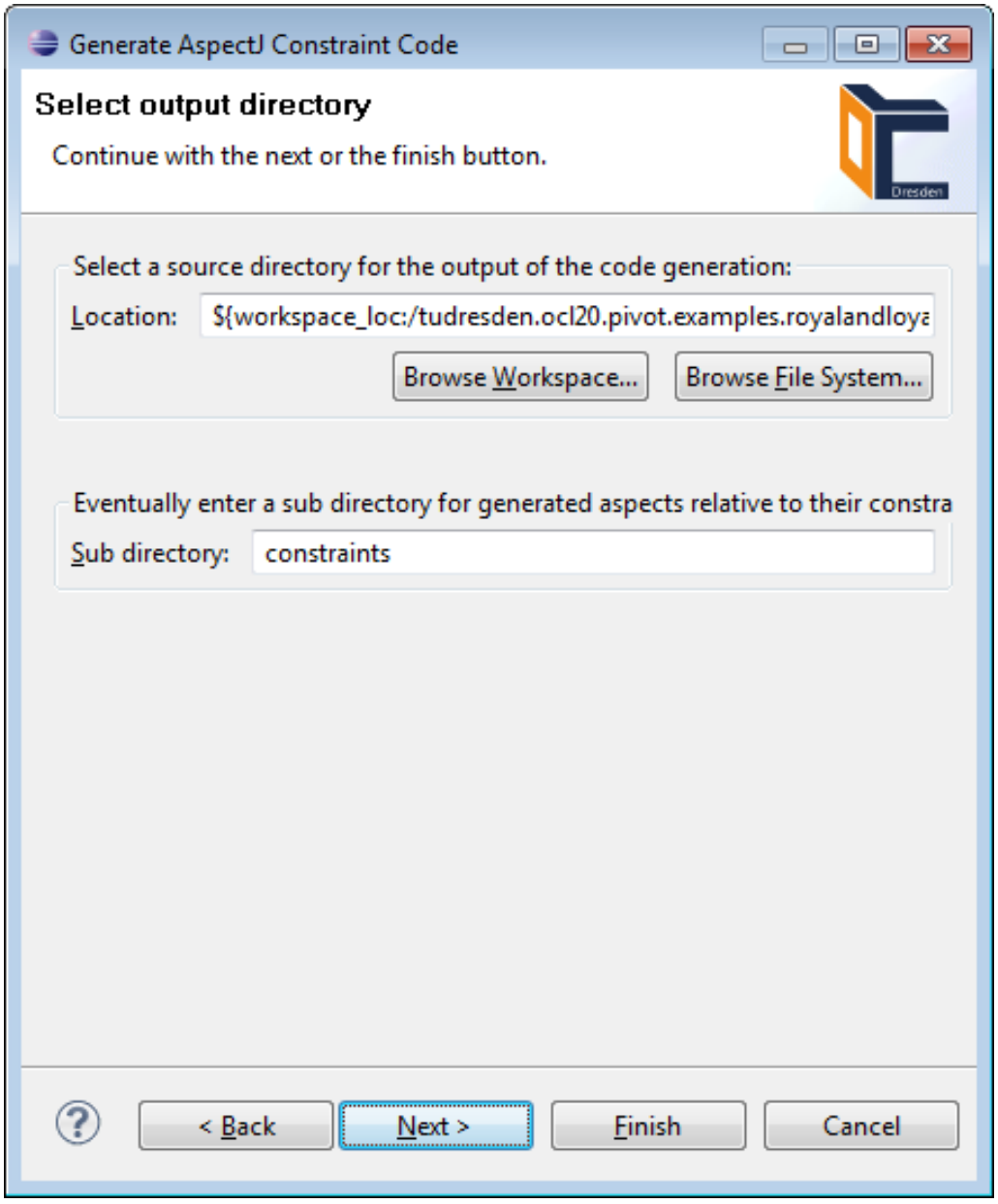

Figure 4.7: The third Step: Selecting a Target Directory for the Generated Code.

4.2.4 Specifying General Settings

On the following page of the wizard we can specify general settings for the code generation (cf. Fig. 4.8). 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:

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*).

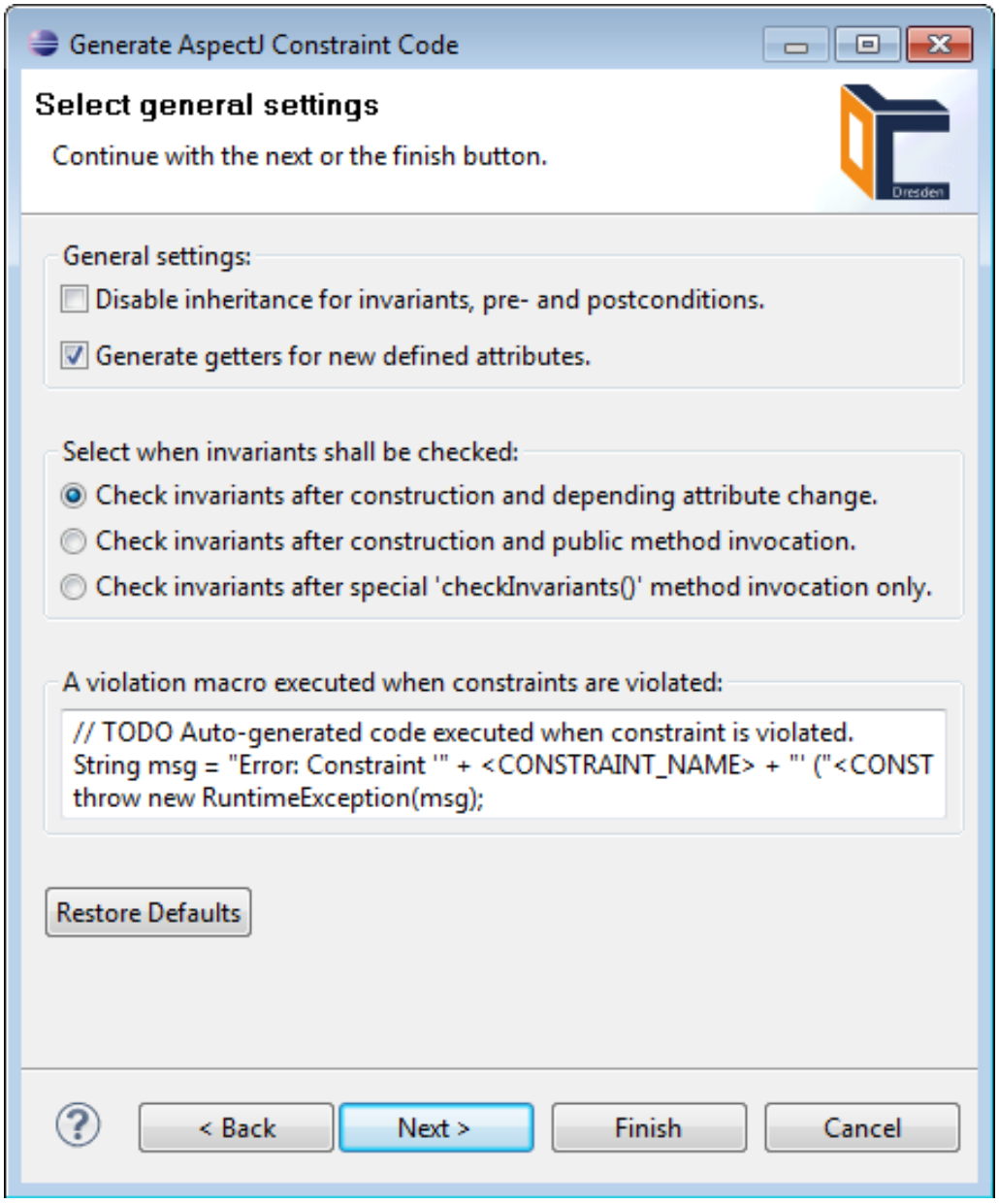

Figure 4.8: The fourth Step: General Settings for the Code Generation.

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*.

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 containing a message saying which constraint was violated by which runtime object. Whithin such messages, some specific keywords can be used to declare information such as the violated constraint's name or body. Table 4.1 gives an overview over the allowed keywords and their meaning. 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.

Keyword	**Meaning within generated error message**
`<CONSTRAINT_NAME>`	The name of the constraint violated during runtime. If the constraint has no name, the term `undefined` will be used instead.
`<CONSTAINTS_BODY>`	The body (OCL expression) of the constraint violated during runtime.
`<OBJECT_IN_ILLEGAL_STATE>`	The Object causing the constraint violation at runtime. The keyword will be replaced by calling the `.toString()` method of the object.

Table 4.1: Keywords allowed to parameterize messages within violation macros.

4.2.5 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 (cf. Fig. 4.9). We don't want to adapt the settings, thus we can finish the wizard and start the code generation by clicking the *Finish* button.

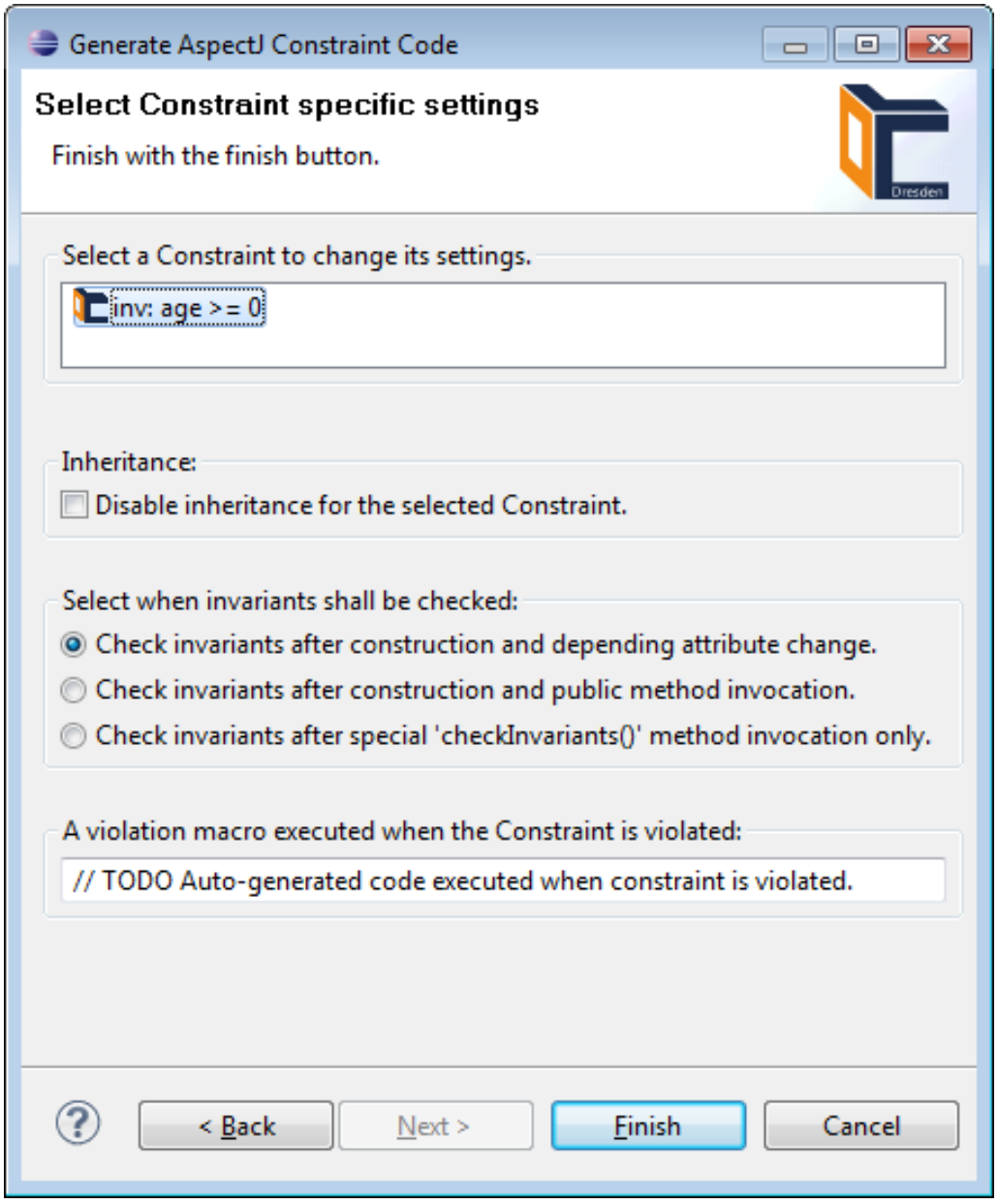

Figure 4.9: The fifth Step: Constraint-Specific Settings for the Code Generation.

4.3 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 newly generated AspectJ file called `tudresden.ocl20.pivot.examples.simple.constraints.InvAspect01.aj` (cf. Fig. 4.10). Now we can rerun our JUnit test case. The test case finishes successfully because the expected run-time exception is thrown (cf. Fig. 4.11).

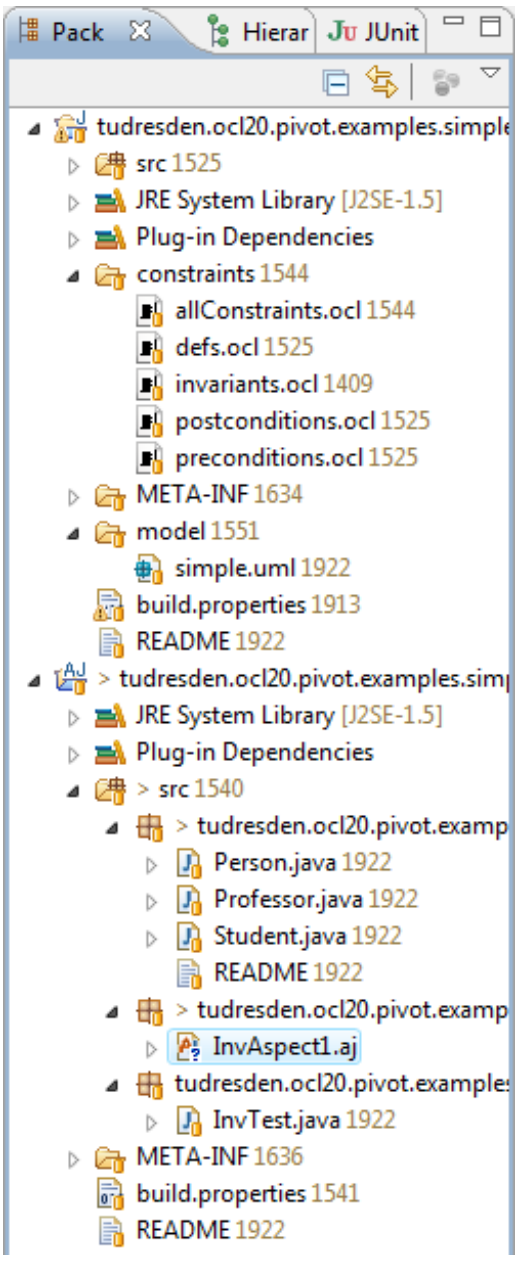

Figure 4.10: The Package Explorer containing the newly generated AspectJ File.

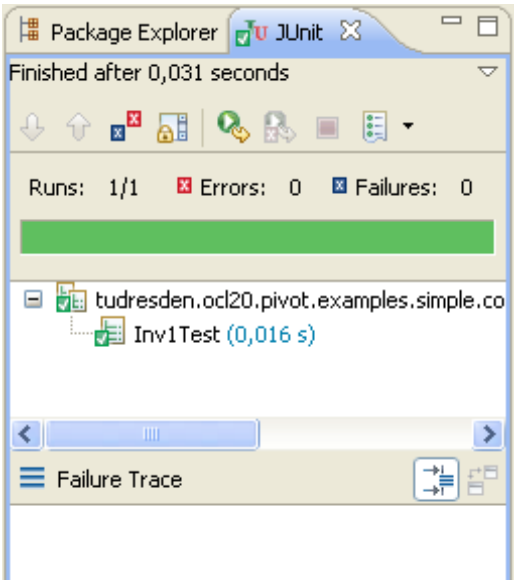

Figure 4.11: The successfully executed jUnit Test Case.

4.4 SUMMARY

This chapter described how to generate AspectJ code using the *OCL22Java* code generator of Dresden OCL. A more detailed documentation of the OCL22Java code generator can be found in the Minor Thesis (Großer Beleg) of Claas Wilke [**?**]. Besides the use of OCL22Java via Dresden OCL's GUI, you can also invoke OCL22Java via Dresden OCL's API. The easiest way to connect to Dresden OCL is via its *Facade* providing interfaces for all services of Dresden OCL. How to use Dresden OCL's facade is documented in Chapter 8.

5 SQL CODE GENERATION

Chapter written by Björn Freitag

This chapter describes how the SQL Code Generator *OCL2SQL* provided with Dresden OCL can be used. A general introduction into Dresden OCL is provided in Chapter 2.

The SQL Code Generator is able to generate the following code of SQL dialects (target languages):

- Standard SQL 2008
- PostgreSQL 9
- Oracle SQL 11g
- MySQL 5.1 und 5.5

5.1 CODE GENERATOR PREPARATION

In the following the *University Example* is used (Figure 5.1). To import the University Example into the Eclipse workspace have to use the wizard *File -> New -> Other. . . -> Dresden OCL Examples -> University Example (UML)*. Afterwards, the project `tudresden.ocl20.pivot.examples.university` should be available within the workspace.

The project provides a model file which contains the university class diagram (the model file is located at `model/university.uml`) and the constraint file (located at `constraints/university.ocl`) we want to generate code for. One invariant is shown in Listing 5.1. It is contained in the constraint file we want to generate code for. The invariant declares, that the `grade` of any `Person`'s superviser must be greater thant its own `grade`.

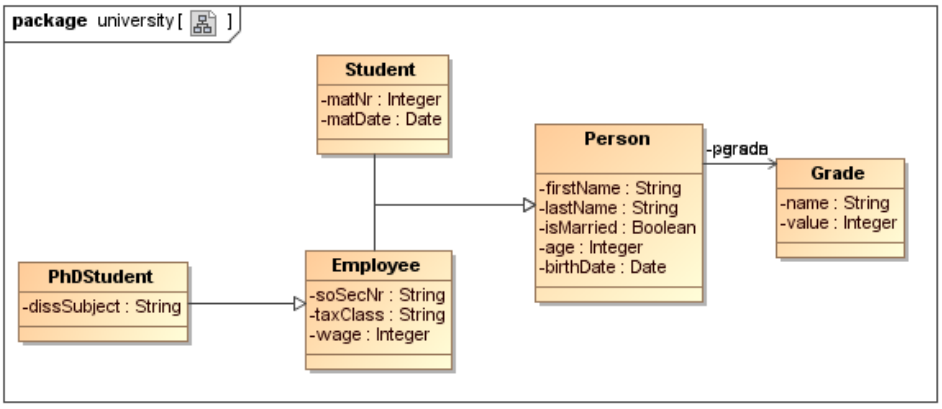

Figure 5.1: the UML diagram of the university example.

```
1 context Person
2 inv tudOclInv1: self.supervisor.grade.value > self.grade.value
```

Listing 5.1: a simple invariant.

5.2 CODE GENERATION

To prepare the code generation we have to import the model `model/university.uml` into the *Model Browser*. We use the model import wizard of Dresden OCL to import the model. This procedure is explained in Chapter 2. Afterwards, we have to open the constraint file `constraints/university.ocl`. Afterwards, import the *Model Browser* should look like illustrated in Figure 5.2. Now we can start the code generation. By selecting the item *Generate SQL Code* of the menu *Dresden OCL* the SQL code generation can be started.

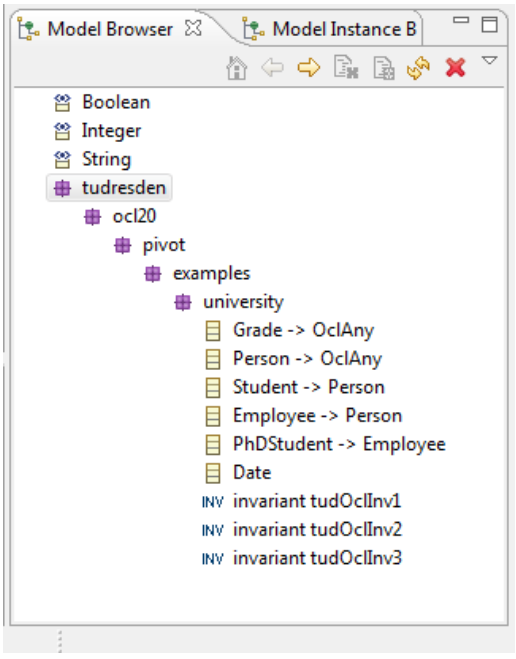

Figure 5.2: The Model Browser showing the university model and its constraints.

5.2.1 Selecting a Model

At first, a wizard is opened and we have to select a model for code generation (cf. Fig. 5.3). We select the `university.uml` model and click the *Next* button.

5.2.2 Selecting Constraints

As a second step we have to select the constraints we want to generate code for. We only select the constraints `inv: oclInv1` and `inv: oclInv2`. After the selection of the constraints we click on the *Next* button (cf. Fig. 5.4).

5.2.3 Selecting a target directory

Now, we have to choose the directory where the generated code will be stored. We select the folder `sql` in this project (which is `tudresden.ocl20.pivot.examples.university/sql`) (cf. Fig. 5.5). Then, we click the *Next* button.

5.2.4 General settings

On the following page of the wizard we can specify general settings for the code generation (cf. Fig. 5.6). We can choose a SQL dialect for the generated code. The modus of SQL generation decides how inheritance relationships are mapped to the SQL schema. In the typed modus all subclass properties will be written to one table of the super class (often called generalization table). Any class has its own table with the own properties in the vertical modus. If you wish only to generate the code for invariant you must choose *Only Integrity View* otherwise the table schema will be generated as well. The other parameter will set the prefix for the different parts of the SQL schema. We can finish the settings and start the code generation with the *Finish* button.

5.3 THE GENERATED CODE

After finishing the wizard, the code for the selected constraints will be generated. To review the results it can be necessary to refresh the project (F5). Our project contains two new SQL files in the folder `sql` (cf. Fig. 5.7). The file `2010-09-29-09-34_schema.sql` contains the table and view definitions (called object views) of the model. Every class of the model has its own view. Via this view the data(objects) of the class can be accesssed. The other file `2010-09-29-09-34_view.sql` contains the SQL code for the invariants (cf. Fig. 5.8), that is one view definition (called integrity view) for each invariant. The semantics of such an integrity view is that after evaluation of an invariant the corresponding integrity view contains all objects that violate the invariant (cf. Listing 5.2).

5.4 SUMMARY

Basically this chapter describes the translation of OCL invariants to SQL integrity views by the *OCL2SQL* code generator of Dresden OCL. Besides the use of *OCL2SQL* via Dresden OCL's GUI, you can also invoke OCL2SQL via Dresden OCL's API. The easiest way to connect to Dresden OCL is via its *Facade* providing interfaces for all services of Dresden OCL. How to use Dresden OCL's facade is documented in Chapter 8.

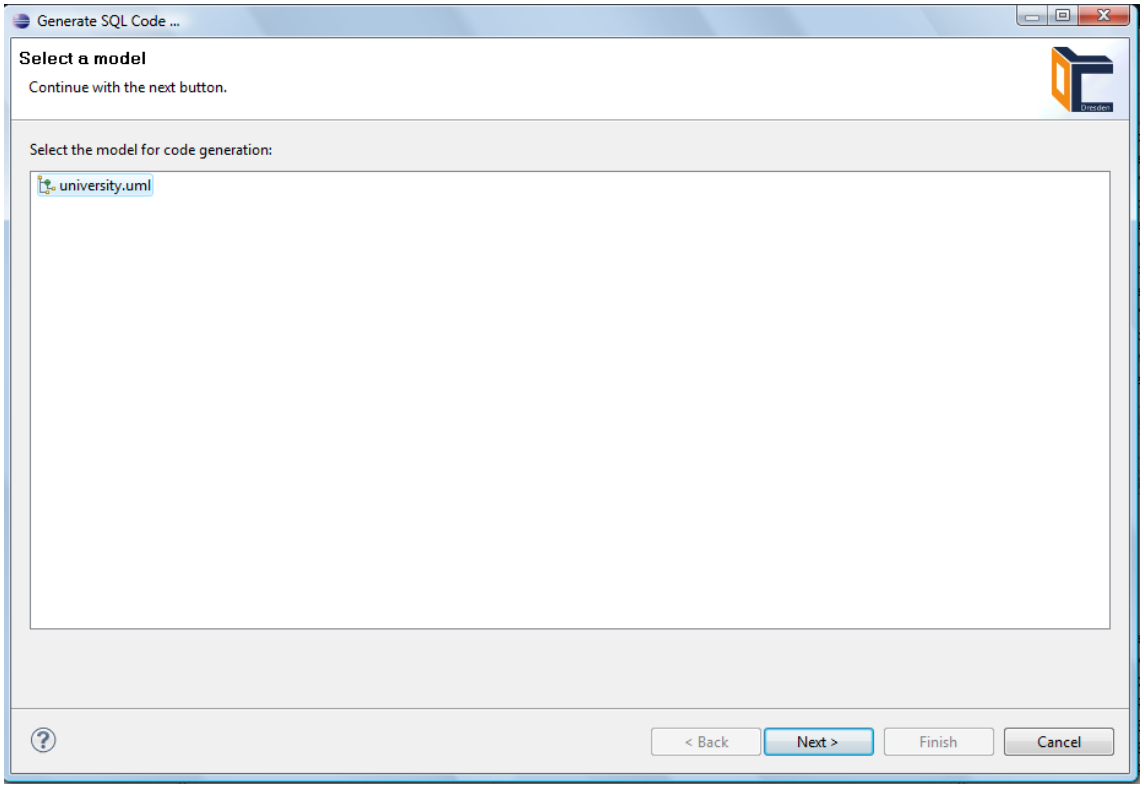

Figure 5.3: The first step: Selecting a model for code generation.

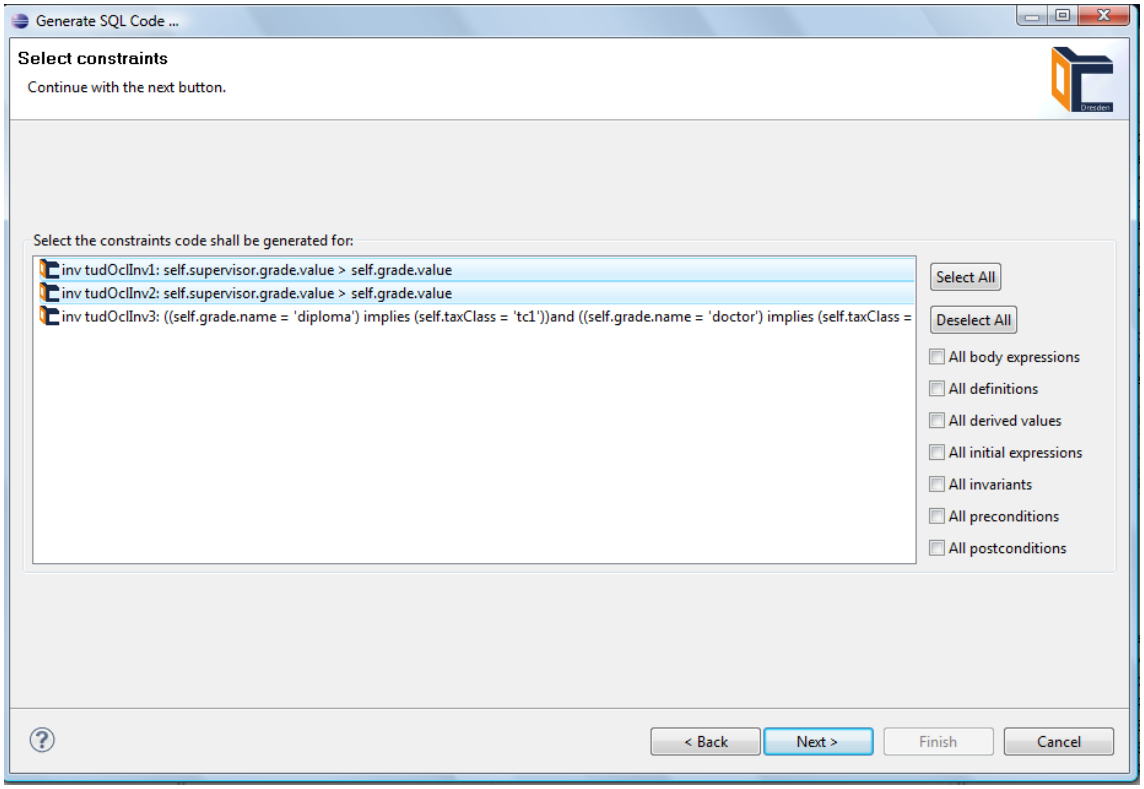

Figure 5.4: The second step: Selecting constraints for code generation.

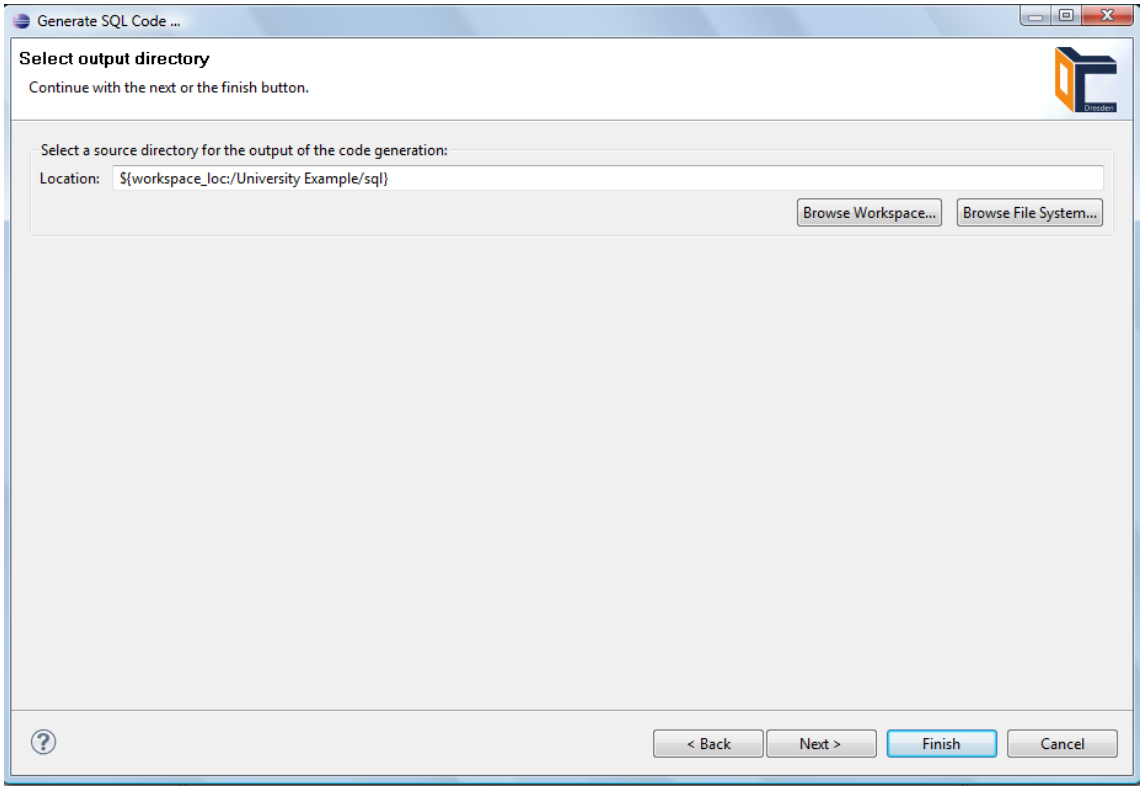

Figure 5.5: The third Step: Selecting a target directory for the generated code.

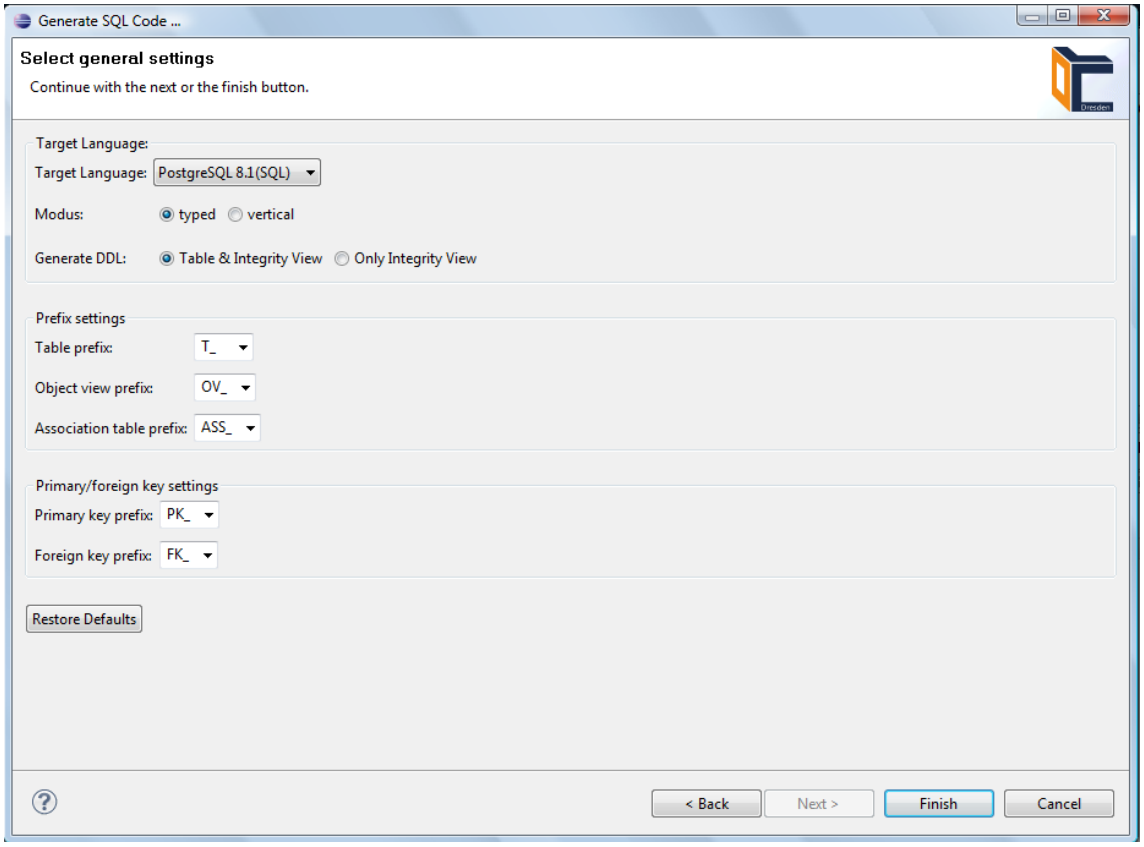

Figure 5.6: The fourth Step: General Settings for the code generation.

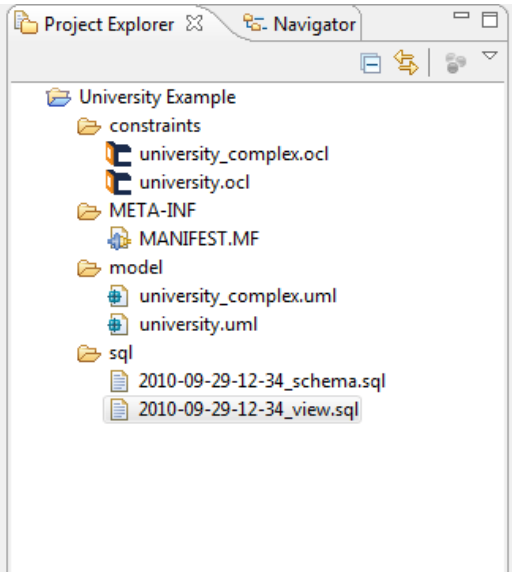

Figure 5.7: The Package Explorer containing the new SQL code files.

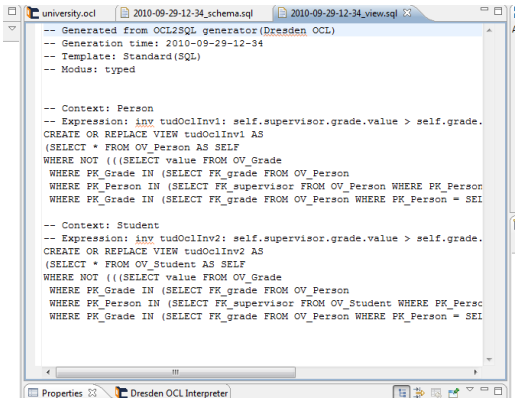

Figure 5.8: The Package Explorer containing the new SQL code files.

```
1  -- Expression: inv tudOclInv1: self.supervisor.grade.value > self.grade.
       value
2  CREATE OR REPLACE VIEW tudOclInv1 AS
3  (SELECT * FROM OV_Person AS SELF
4  WHERE NOT (((SELECT value FROM OV_Grade
5   WHERE PK_Grade IN (SELECT FK_grade FROM OV_Person
6   WHERE PK_Person IN (SELECT FK_supervisor FROM OV_Person WHERE PK_Person
        = SELF.PK_Person))) > (SELECT value FROM OV_Grade
7   WHERE PK_Grade IN (SELECT FK_grade FROM OV_Person WHERE PK_Person = SELF
       .PK_Person)))));
```

Listing 5.2: SQL Code for invariant oclInv1.

6 DRESDEN OCL METRICS

Chapter written by Claas Wilke

This chapter describes how the Dresden OCL Metrics tool provided with Dresden OCL can be used. A general introduction into Dresden OCL can be found in Chapter 2. This chapter uses the *Simple Example* which is provided with Dresden OCL and has been introduced in Section 2.2.1.

6.1 USING DRESDEN OCL METRICS

The Dresden OCL Metrics tool is a simple tool that allows to compute some statistics for a selected or a set of selected OCL constraints.

In summary, the following metrics can be computed:

- The number of constraints selected,
- The number of constraints by kind (Body, Definition, Derived, Init, Invariant, Postcondition, Precondition),
- The number of expressions used within the selected constraints (including minimum, maximum, average, and median number of expressions per constraint),
- The depth of the expressions within the selected constraints (minimum, maximum, average, and median),
- The number of if-expressions used within the selected constraints,
- The number of let-expressions used within the selected constraints,
- The number and kinds of literals used within the selected constraints,
- The number and kinds of iterators used within the selected constraints,
- The number and names of operations called within the selected constraints,
- The number and names of properties called within the selected constraints.

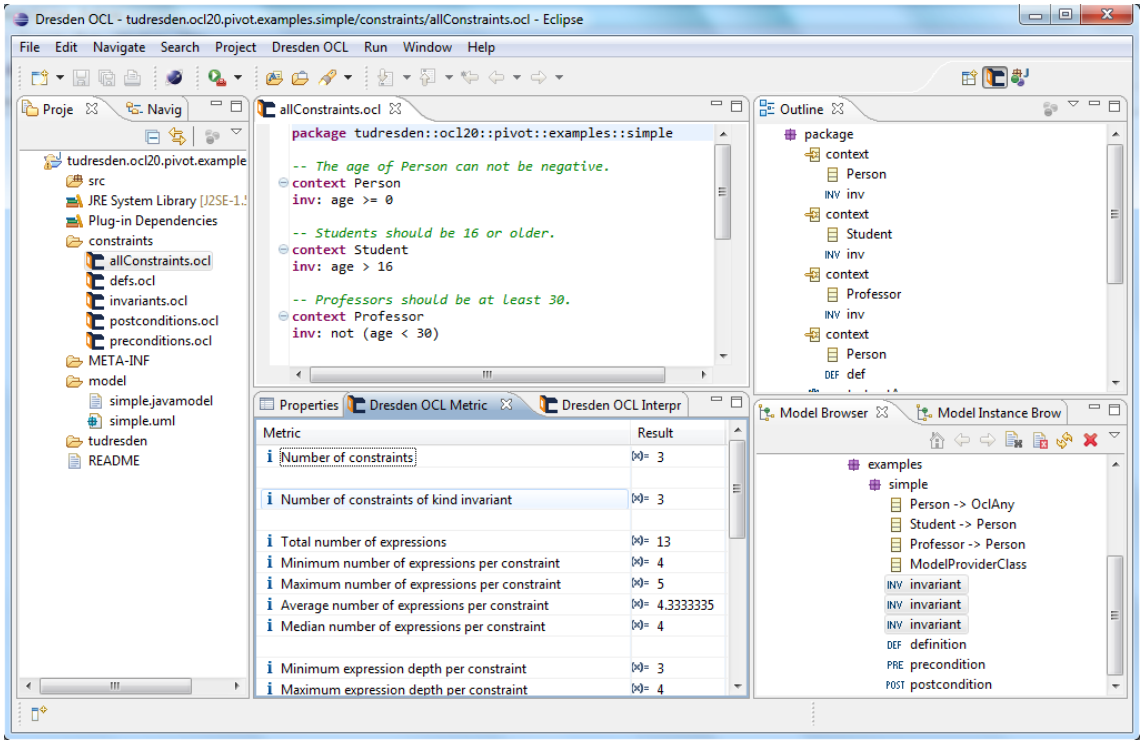

Figure 6.1: The Dresden OCL perspective showing the Dresden OCL Metrics view at the center bottom.

Dresden OCL Metrics consists of a single view that should be visible at the center bottom of the Dresden OCL perspective (cf. Fig. 6.1). If not, the view can be opened by the menu option *Dresden OCL -> Open OCL Metrics*.

Any time, you select a set of constraints or a model element containing a set of constraints within the *Model Browser*, the results will be automatically displayed within the *Dresden OCL Metrics View* (cf. Fig 6.1).

6.2 SUMMARY

This chapter described how to use the Dresden OCL Metrics. Feel free to explore the metrics using your own example. If you have ideas for further metrics, please do not hesitate to let us know. Maybe we will implement your metrics within a following version of Dresden OCL . . .

II TOOL DEVELOPMENT USING DRESDEN OCL

7 THE ARCHITECTURE OF DRESDEN OCL

Chapter written by Claas Wilke

This chapter introduces into the highly generic architecture of Dresden OCL. Before the architecture is explained, some theoretical background is shortly presented. Further details of Dresden OCL's architecture can be found in [**?**] and [**?**].

7.1 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)* [**?**], 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 / abstract syntax) and can then be verified for instances of the model afterwards.

The OMG introduced the *MOF Four Layered Metadata Architecture* [**?**][**?**, 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 7.1). Generally, four layers can be identified, the *Meta-Meta-Model Layer (M3)*, the *Meta-Model Layer (M2)*, the *Model Layer (M1)*, and the *Model Instance Layer (M0)*. The latest version of MOF allows to define as much layers as required for a certain use case [**?**, p. 8f].

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

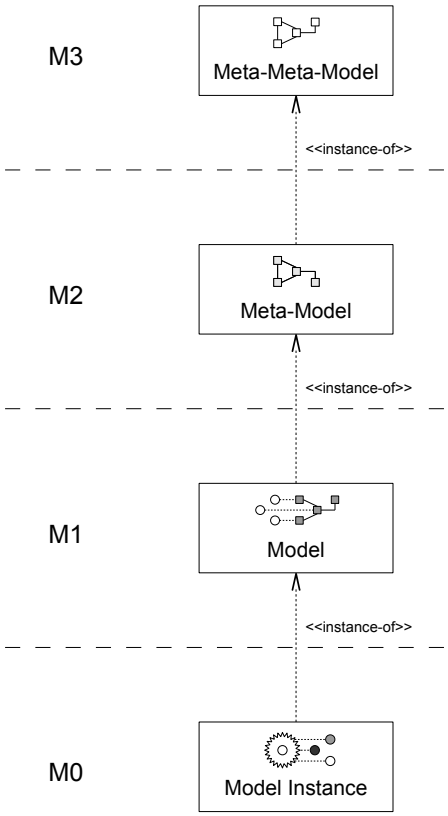

Figure 7.1: The MOF Four Layer Metadata Architecture.

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 architecture can be generalized to a *Generic Three Layer Metadata Architecture* [**?**] in the scope of an OCL definition (see Figure 7.2).

On the *Mn+1 Layer* lies the meta-model that is used to define the model that shall be constrained. The OCL abstract syntax extends the meta-model to allow contexts of OCL constraints refering to elements of the model.

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 EMF Ecore model in Figure 7.2 (c))) or a set of objects (like Java run-time objects in Figure 7.2 (b)). To provide a relationship between the model instance's elements and the model's elements a mapping is required between both. Thus, the model instance requires a description of a model realisation that allows to reflect on the instance's elements (e.g., for Java run-time objects such a realisation can be considered as a set of Java classes. Each `java.lang.Object` allows to reflect on its `java.lang.Class`. This `Class` can then be mapped to the corresponding model element in the model by comparing the `Classes` name with the names of `Types` existing in the model).

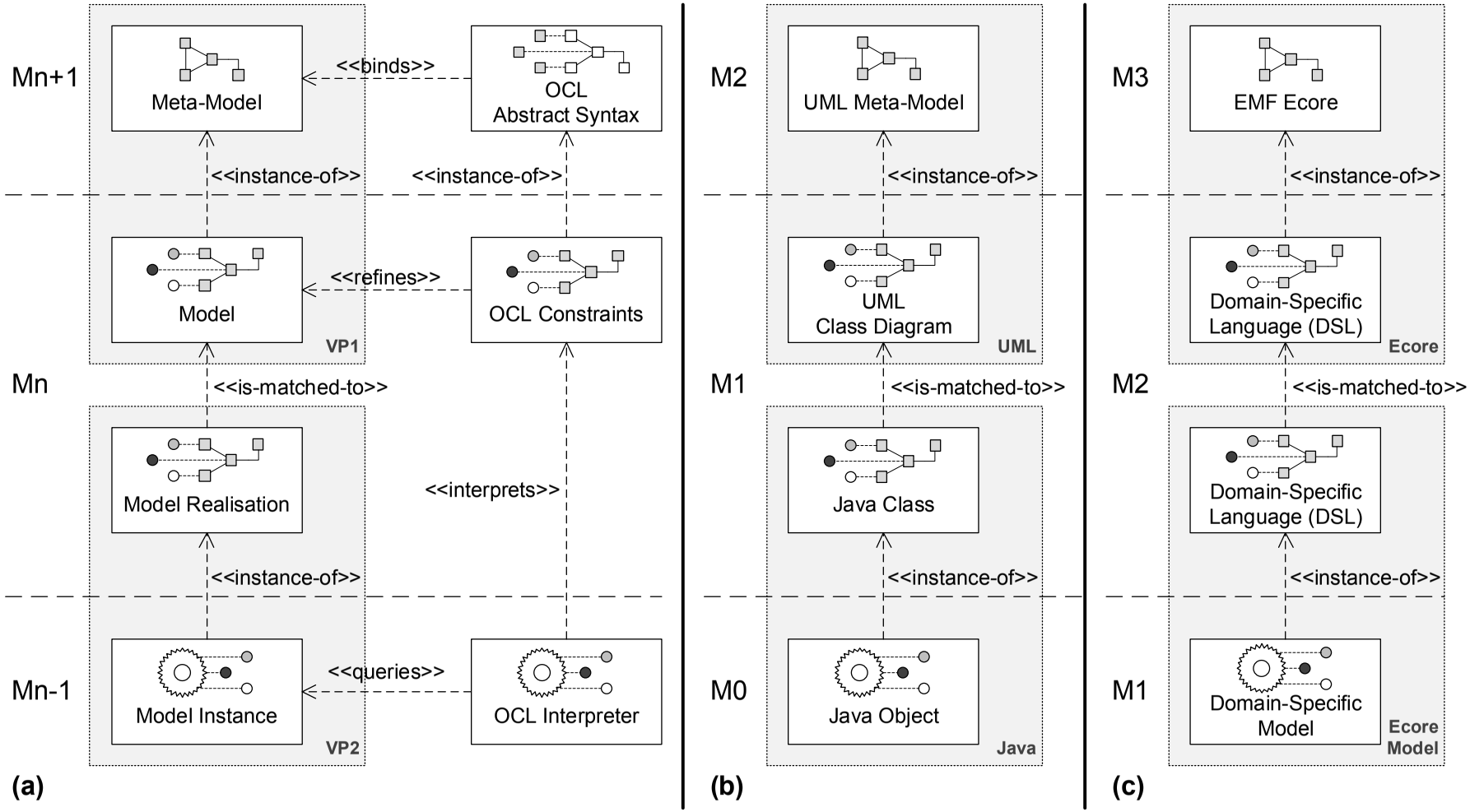

Figure 7.2: The Generic Three Layer Metadata Architecture.

7.2 DRESDEN OCL'S PACKAGE ARCHITECTURE

The package architecture of Dresden OCL is shown in Figure 7.3. The architecture can be separated into five layers:

1. The *API Layer*,
2. the *Tools Layer*,
3. the *Registry Layer*,
4. the *OCL Layer*,
5. and the *Variability Layer*.

The API layer provides the Facade of Dresden OCL that can be used by other plug-ins to access the OCL tools of Dresden OCL. The Facade of Dresden OCL is introduced in Chapter 8.

The Tools Layer contains the Tools provided by Dresden OCL. These are: The OCL Parser/Editor introduced in Section 2.2.5, the OCL Interpreter introduced in Chapter 3, the OCL22Java Code Generator introduced in Chapter 4, and the OCL2SQL Code Generator introduced in Chapter 5. Finally, the *Model Browser* can be used to store imported models, and model instances. The backend of the Model Browser is the *Model Bus* that manages all meta-models and model instance types currently provided by Dresden OCL. The model bus is located at the Registry Layer.

The OCL layer contains the OCL syntax and semantics used by the tools of Dresden OCL. The OCL syntax and semantics are separated into three different packages. *Essential OCL* contains the abstract syntax (or meta-model) of OCL that is used by Dresden OCL and its tools. The *OCL Standard Library Model* provides a modeled description of all operations that are provided by the

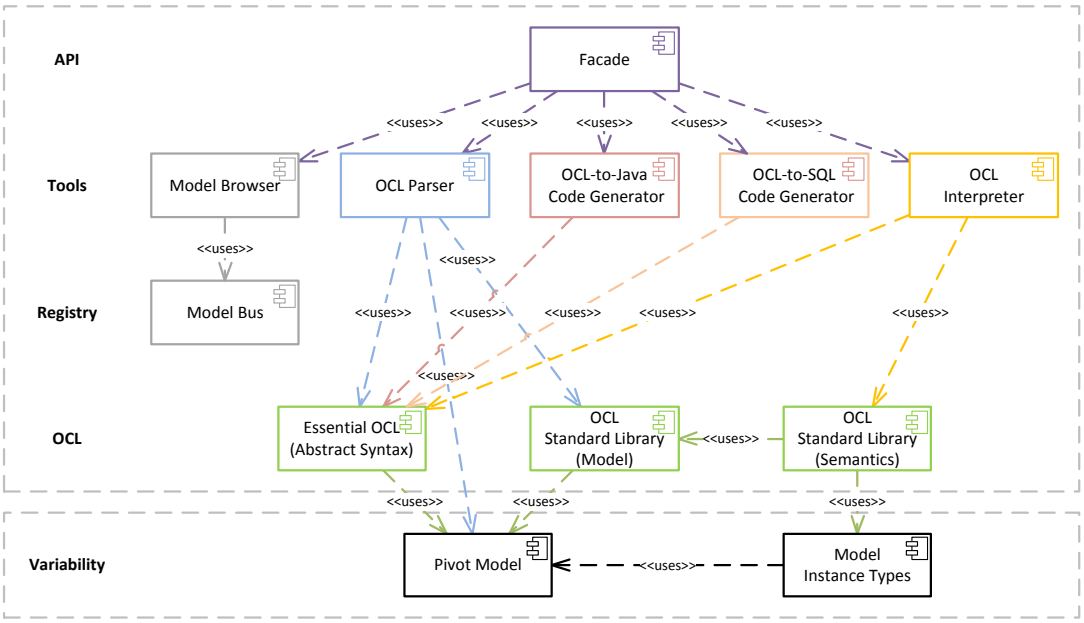

Figure 7.3: The package architecture of Dresden OCL.

OCL Standard Library (as defined in [**?**, Ch. 11]). Finally the *OCL Standard Library Semantics* provides an implementation of all these operations that are used by the OCL Interpreter of Dresden OCL for OCL interpretation.

The last layer is the Variability Layer. The Variability layer contains the *Pivot Model* and the *Model Instance Types* that are used to connect Dresden OCL with various meta-models or types of model instances, respectively. The Pivot Model is described more detailed in Section 7.3.2, the Model Instance Types are explained in Section 7.3.3. Further details on both the Pivot Model and the Model Instance Types can be found in [**?**] and [**?**].

Dresden OCL has been developed as a set of Eclipse/OSGi plug-ins. All packages 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. An overview over all plug-ins provided with Dresden OCL2 for Eclipse can be found in Table 2 in the appendix of this manual.

7.3 DRESDEN OCL AND THE GENERIC THREE LAYER METADATA ARCHITECTURE

Figure 7.4 shows the architecture of Dresden OCL with respect to the Generic Three Layer Metadata Architecture (introduced in Section 7.1). 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.

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

As you can see, the left part of Figure 7.4 shows the Generic Three Layer Metadata Architecture. Meta-models, models and model instances are adapted and loaded into Dresden OCL. It could be argued that such an adaptation is expensive and costly, but the opposite is the truth. The architecture of Dresden OCL allows its users to adapt the toolkit to every meta-model and model instance type they want to. After the adaptation of a new meta-model or model-instance type, they can reuse the complete set of tools provided by Dresden OCL! Thus, to adapt the OCL Interpreter to a new type of model instance, only one adaptation is required. The rest comes for free!

7.3.2 How Meta-Models and Models are Adapted

As already said, the core feature of Dresden OCL is the *Pivot Model* (a.k.a. *Model Types*). The pivot model is a meta-model that abstracts from all other meta-models. It contains interfaces to define the structural parts 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 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 an *Adapted Model*, Dresden OCL can work with. Further details about the adaptation of meta-models to the pivot model can be found in Chapter 10.

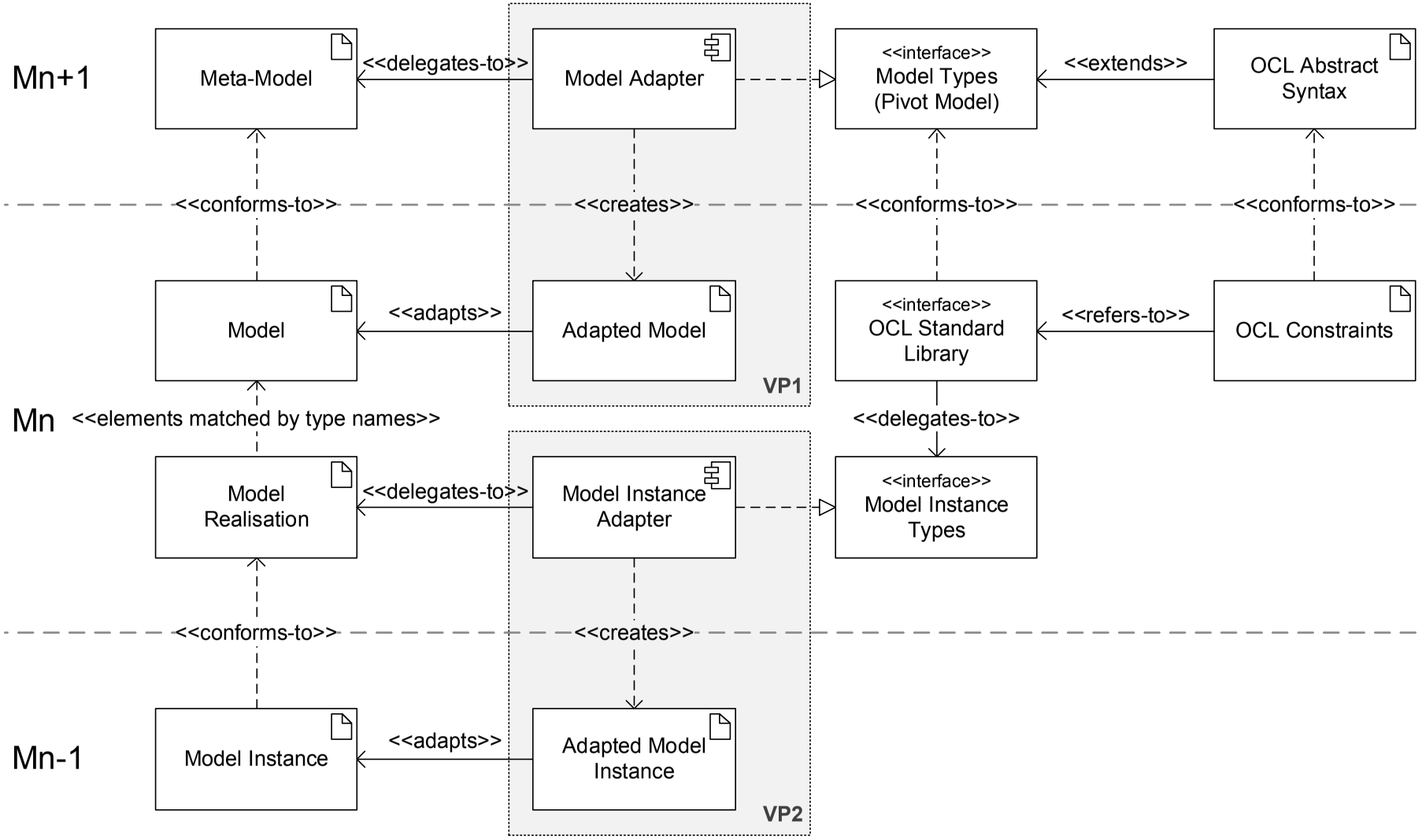

Figure 7.4: The architecture of Dresden OCL with respect to the Generic Three Layer Metadata Architecture.

7.3.3 How Model Instances are Adapted

If the OCL Interpreter shall be used to interpret instances of amn opened 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-time 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 OCL interpreter does not accesses this property directly, but delegates the request to a *Model Instance Element* adaptation to let the model instance type become exchangeable[1]. Thus, Dresden OCL contains a second model the *Model Instance Types* that can be considered as an abstraction of all model instance types. The model instance types define 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 elements (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 types. 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 OCL is able to use the same OCL Interpreter for different types of model instances! More details about the adaptation of model instances to Dresden OCL are available in Chapter 11.

7.3.4 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 OCL, 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.

7.4 SUMMARY

This Chapter introduced into the architecture and package structure of Dresden OCL. The *Pivot Model* and the *Model Instance Types* 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 OCL was designed as generic as possible. Thus, Dresden OCL can be adapted to various different kinds of meta-models and model instances without changing the OCL Parser nor the OCL Interpreter! The currently existing adaptations to meta-models and model instances are documented in Section 1.2 and Section 1.3, respectively.

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

8 HOW TO INTEGRATE DRESDEN OCL INTO ECLIPSE PROJECTS

Chapter written by Claas Wilke

In Chapter 7 the architecture of Dresden OCL has shortly been explained. This chapter will explain, how Dresden OCL can be integrated into other Eclipse plug-ins. If you plan to use Dresden OCL as a JAR library without Eclipse plug-in structure, you should use the *Standalone Integration Facade* presented in Chapter 9 instead.

8.1 THE INTEGRATION FACADE OF DRESDEN OCL

Since the release 2.2.0, Dresden OCL contains an *Integration Facade*, that combines all required interfaces of Dresden OCL in one interface, also called a *Facade* [**?**]. The facade contains self-explanatory static methods that provide access to the repository (modelbus) and all tools of Dresden OCL. 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 OCL. Thus, if you want to use one of the tools only (e.g, the OCL Parser) you could access these tools directly as explained below.

8.2 HOW TO ACCESS META-MODELS, MODELS AND INSTANCES

The central registry component of Dresden OCL 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 methods, to access meta-models, models and model instances and to import new resources of these kinds into the toolkit (see Figure 8.1).

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

8.2.1 The Meta-Model Registry

The Meta-Model Registry provides methods to add and get meta-models to and from Dresden OCL. 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`).

8.2.2 How to load a Model

First, to load a model into Dresden OCL, 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 8.1). 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 OCL (see Figure 8.3). 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 OCL's GUI.

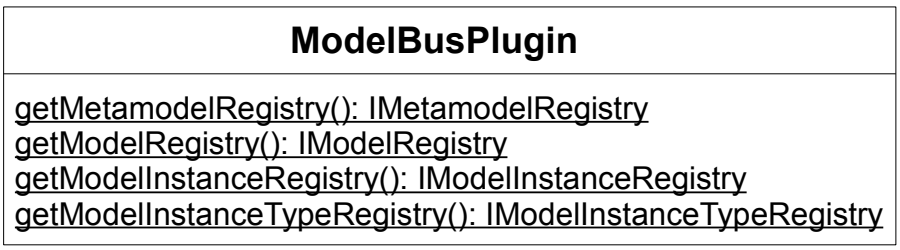

Figure 8.1: The main class of the Model-Bus plug-in.

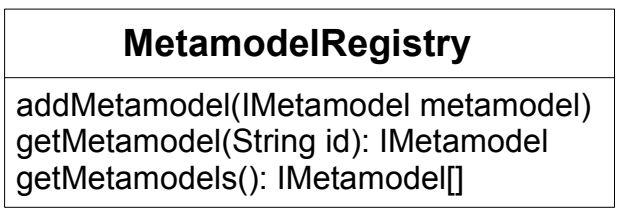

Figure 8.2: The Meta-Model Registry.

```
1 IMetamodel metaModel;
2 IModel model;
3
4 metaModel = ModelBusPlugin.getMetamodelRegistry()
5                .getMetamodel("tudresden.ocl20.pivot.metamodels.uml2");
6 model = metaModel.getModelProvider().getModel(modelURL);
```

Listing 8.1: How to load a model.

8.2.3 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 OCL. 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 Dresden OCL, 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`).

8.2.4 How to load a Model Instance

First, to load a model instance into Dresden OCL, the model instance type must be selected of that the instance is an instance of. E.g., for a set of Java objects the Java model instance type should be selected (see Listing 8.2). 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,

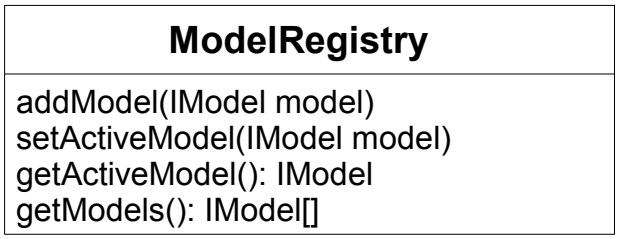

Figure 8.3: The Model Registry.

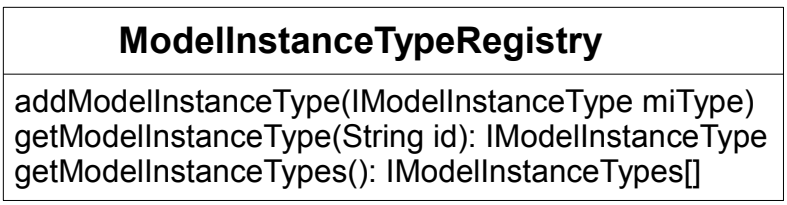

Figure 8.4: The Model Instance Type Registry.

```
IModelInstanceType miType;
IModelInstance modelInstance;

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

Listing 8.2: 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 OCL (see Figure 8.5). 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 OCL's GUI.

8.3 HOW TO ACCESS THE OCL PARSER

The OCL2 Parser of Dresden OCL2 for Eclipse is located in the plug-in `tudresden.ocl20.pivot.language.ocl.staticsemantics` within the package `tudresden.ocl20.pivot.language.ocl.resource.ocl`. The parser provides a very simple interface and can be used as shown in Listing 8.3. You have to provide an `IModel` and an `URI` where the OCL file can be found. You can specify whether constraints and definitions should be added to the `IModel` by a third parameter named `addToModel`. It can be left out and is then assumed to be `true`.

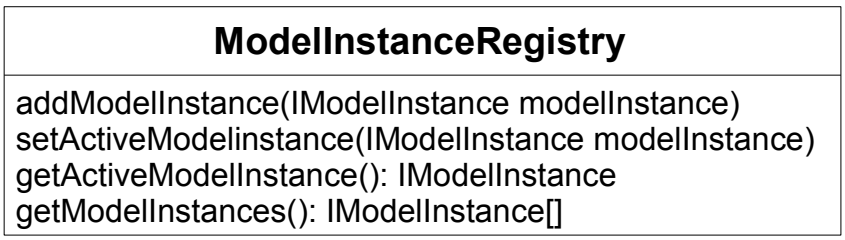

Figure 8.5: The Model Instance Registry.

```
URI uri = URI.createFileURI(oclFile);
boolean addToModel = true;
List<Constraint> parsedConstraints;

parsedConstraints = Ocl22Parser.INSTANCE.doParse(model, uri, addToModel)
```

Listing 8.3: How to parse constraints.

8.4 HOW TO ACCESS THE OCL INTERPRETER

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

Listing 8.4 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`.

8.5 SUMMARY

This chapter shortly presented, how the tools of Dresden OCL 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.

```
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 8.4: How to interpret constraints.

9 STANDALONE – USING DRESDEN OCL OUTSIDE OF ECLIPSE

Chapter written by Michael Thiele

Although Dresden OCL is targeted at the Eclipse platform, it can be used outside of it as a standalone application. All GUI components cannot be used, but all the core components like model loading, model instance loading, parsing OCL constraints, interpreting them or generating AspectJ code out of them are available.

9.1 THE EXAMPLE APPLICATION

The easiest way to use the standalone application is to check out the provided example at `http://svn-st.inf.tu-dresden.de/svn/dresdenocl/trunk/ocl20forEclipse/standalone/tudresden.ocl20.pivot.standalone.example`. It can be used as an Eclipse Java project (not plug-in project), can be imported into Netbeans or can be used from the command line as an Eclipse runtime is not required. The structure of the example is described below.

9.1.1 Classpath

The `lib` folder contains all libraries that are needed when executing the example. This includes several Eclipse jar files as well as stringtemplate.jar that is needed for the code generation facilities of the application and others. The `plugins` folder contains all jar files that are provided by Dresden OCL. When using the example as an Eclipse or Netbeans project all jars are already on the classpath. When using the command line or other tools, you probably have to add these jars to the classpath manually.

9.1.2 Resources

The `resources` folder has three sub-folders. In `constraints` the OCL constraint files for three different examples are located. The `model` folder contains the model files. Note, that in order

to use the Java models, you need the appropriate `.class` files that are contained in a folder structure that represents the package name of the classes. Also, all classes that are referenced by the model need to be located in the folder structure or have to be on the classpath. Otherwise, a `NoClassDefFoundError` will be thrown. The `modelInstance` folder contains model instance files that again, in the case of Java model instances, have the same conditions as described for Java models.

9.1.3 Logging

In order to log different parts of Dresden OCL, you can alter the `log4j.properties` file. There, two appenders and a root logger are already defined. In order to log only specific parts of the toolkit, for example, you can enter: `log4j.logger.tudresden.ocl20.pivot.interpreter =debug, stdout` that puts all log messages of the interpreter on the console (please be aware that this will significantly slow down the execution time).

9.1.4 Using the Example

The `src` folder contains one class, named `StandaloneDresden OCLExample.java`. It contains a `main` method that executes numerous examples.

Royal & Loyal The first example is the Royal & Loyal example originally introduced by WARMER & KLEPPE [**?**]. The model is loaded by calling `loadUMLModel(File, File)` on the `StandaloneFacade`. The method takes two arguments, the first being the file where the UML model is located. The second argument needs to be a file that points to the resources jar of the Eclipse UML project. In the example project, this jar file is located at `lib/org.eclipse.uml2.uml.resources_3.0.0.v200906011111.jar`. The example UML model provided here was built using Eclipse UML 3.x. In order to load older models, you have to update the namespace in line 2 of the model (`xmlns:uml="http://www.eclipse.org/uml2/3.0.0/UML"`) or use an older resources jar file. You have to provide the resources to be able to use primitive types in your model that are not mapped correctly when no or the wrong resources are specified.

Then, the Java model instance is loaded. Note, that this needs to be a class that has a method with this signature: `public static List<Object> getModelObjects()`. In the returned list all objects that shall be part of the model instance have to be contained. The actual loading is done by calling `StandaloneFacade.INSTANCE.loadJavaModelInstance(IModel, File)`.

Parsing and interpretation should be self explanatory. The method `StandaloneFacade.INSTANCE.interpretEverything(IModelInstance, List<Constraint>)` returns a list of `IInterpretationResults` that contain the context (`getModelObject()`), the executed constraint (`getConstraint()`) and the result of the interpretation (`getResult()`).

In order to generate AspectJ code, one needs to define the `IOcl22CodeSettings` first. The settings can be created by `Ocl22JavaFactory.getInstance().createJavaCodeGeneratorSettings()`. There are lots of non obligatory settings; only the directory where the code should be generated to has to be specified invariably. Afterwards the code can be generated calling `StandaloneFacade.INSTANCE.generateAspectJCode(List<Constraint>, IOcl22CodeSettings)`.

PML The PML example is nearly analogous to the Royal & Loyal example described above. Before being able to load an Ecore model instance, you have to register the Ecore model at the global Ecore registry. In case of PML this is done by the lines

```
1 Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap()
2   .put("pml", new XMIResourceFactoryImpl());
3 PmlPackage.eINSTANCE.eClass();
```

Other Ecore models require other factory mappings and their package to be initialised.

Simple The difference to the other examples is the model instance that is not loaded from a class file, but built in the example itself. To create an empty Java model instance call

```
1 IModelInstance modelInstance = new JavaModelInstance(model);
```

Calling the method `addModelInstanceElement(Object)` on the model instance will cause the given object to be adapted to the model instance. Thus, constraints can be evaluated on it.

9.2 THE STANDALONE FACADE

The example already contains a jar file of the the facade. In order to change the implementation of the `StandaloneFacade`, check out the Java project available in the SVN `https://dresden-ocl.svn.sourceforge.net/svnroot/dresden-ocl/` at `trunk/ocl20forEclipse/standalone/tudresden.ocl20.pivot.standalone.facade`.

9.2.1 Classpath and OCL Standard Library

The project contains all required jars in the folders `lib` and `plugins` like the example. The `resources` folder also needs to be on the classpath. It contains the modelled version of the OCL standard library. If you want to use your own version of the OCL standard library, you can manipulate the given file or create a new one and change the `initialize(URL)` method in the `StandaloneFacade` to point to your version.

9.2.2 Adding and Removing Methods

The central class for standalone applications is the `StandaloneFacade`. Adding new methods to the facade should pose no problems as long as all needed libraries are on the classpath. If the facade offers too much possibilities, you can delete all methods you do not need. Then, you are able to remove libraries from the classpath that are no longer needed (e.g., all UML related jar files when loading UML models is not an option). Thus, you can significantly reduce the size of needed libraries for standalone applications.

9.3 SUMMARY

The chapter showed how to use Dresden OCL without Eclipse. An extensive example has been explained and how to modify the standard behaviour of the standalone facade that should be used by standalone applications.

10 ADAPTING A META-MODEL TO THE PIVOT MODEL

Chapter written by Michael Thiele

Dresden OCL 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 OCL includes a code generator that creates adapter stubs.

10.1 THE ADAPTER CODE GENERATION

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 Dresden OCL and the meta-model to adapt that has to be modeled in EMF Ecore.

In the following Ecore itself will be adapted to the pivot model (this has been done already for Dresden OCL, but serves the purpose of showing the adaption mechanism on a well known meta-model). Figure 10.1 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 10.2. In the *Properties View* enter `http://www.tu-dresden.de/ocl20/pivot/2007/pivotmodel` as `Source` (see Figure 10.3). Create a new details entry for the annotation (Figure 10.4) and enter `Resource` as `Key` and `org.eclipse.emf.ecore.resource.Resource` as `Value` (see Figure 10.5).

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 10.6 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`

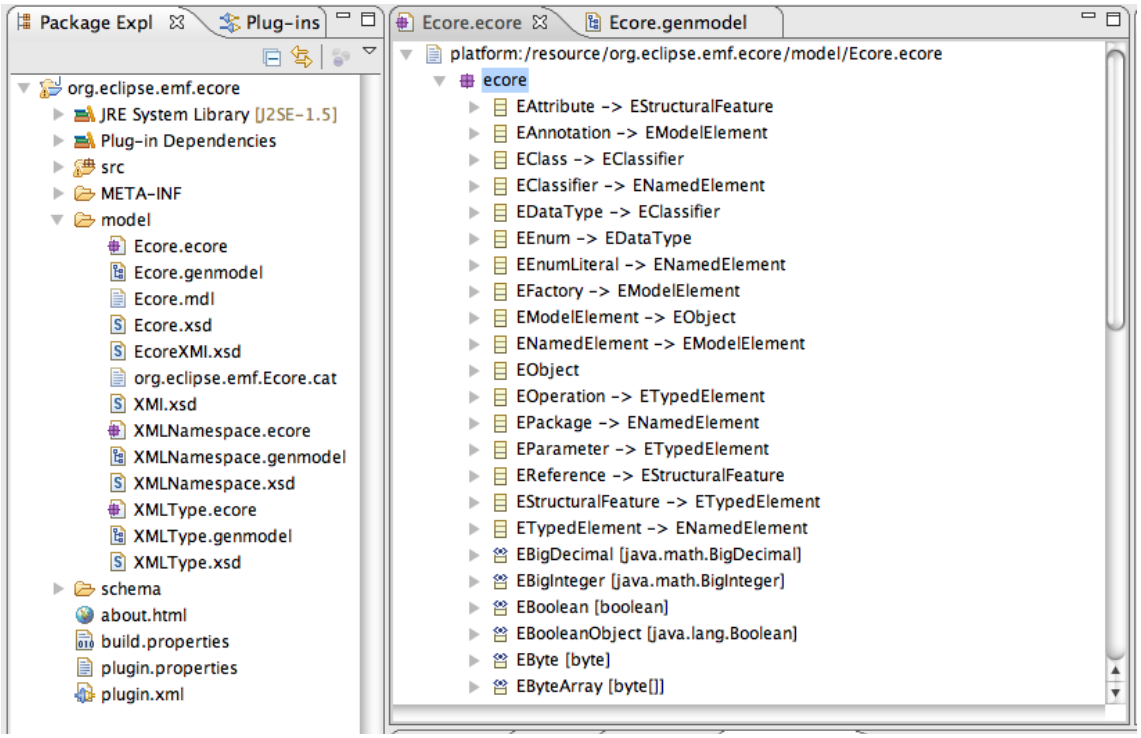

Figure 10.1: The Ecore model opened in Eclipse.

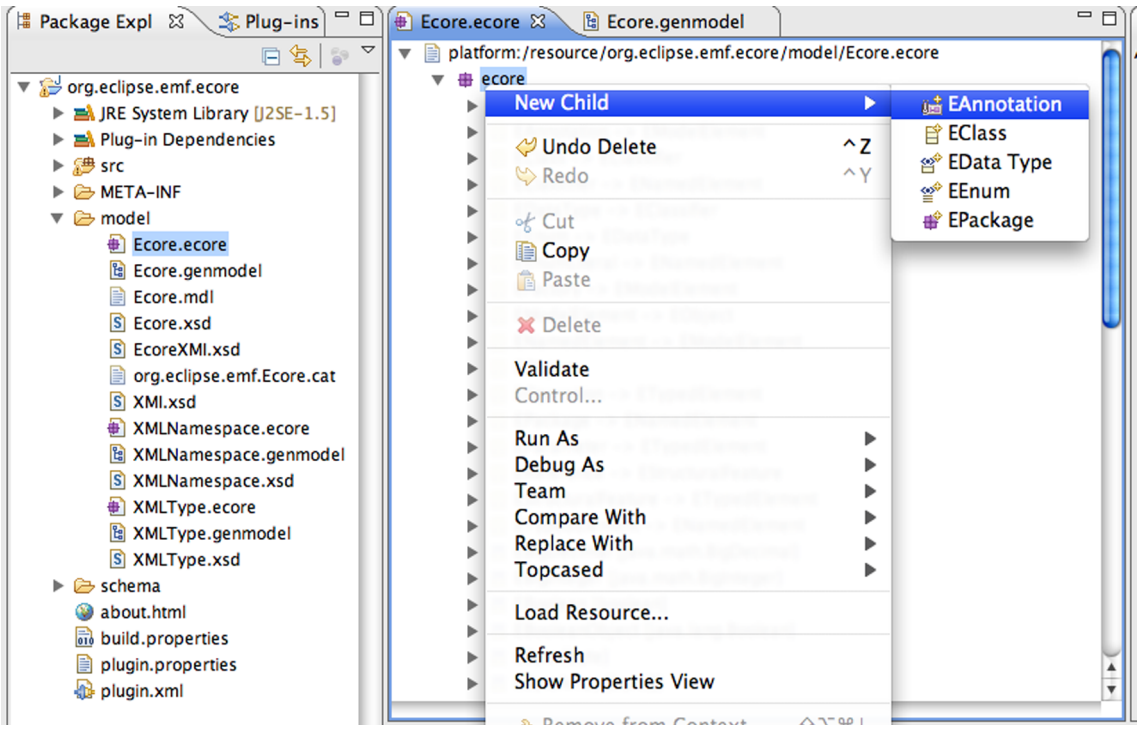

Figure 10.2: Create an annotation for the Ecore package.

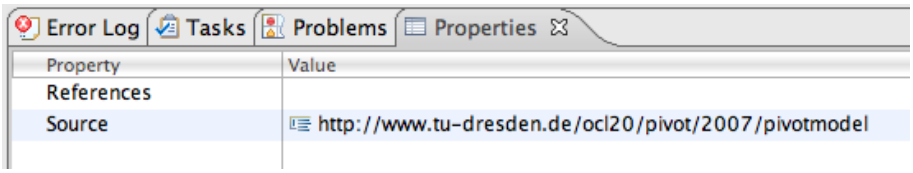

Figure 10.3: The Properties View for the annotation.

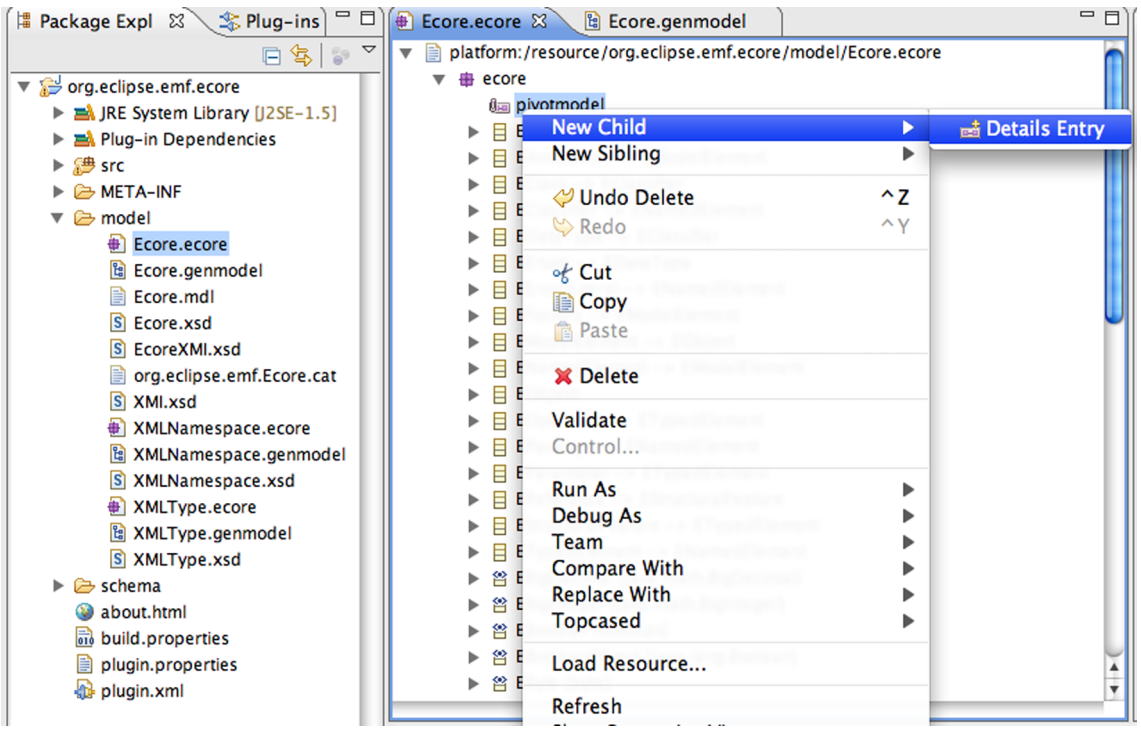

Figure 10.4: Create annotation details for the annotation.

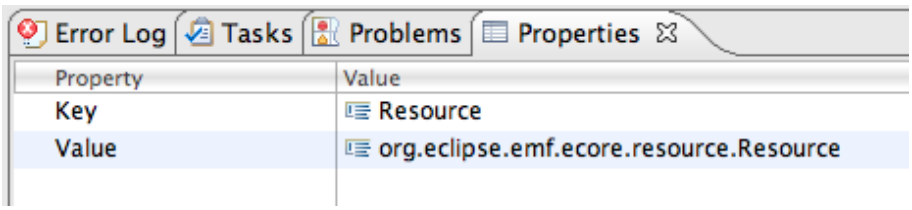

Figure 10.5: The Properties View for the annotation details.

- `EDataType ->PrimitiveType`
- `EEnum ->Enumeration`
- `EEnumLiteral ->EnumerationLiteral`
- `EOperation ->Operation`
- `EPackage ->Namespace`
- `EParameter ->Parameter`
- `EStructuralFeature ->Property`

When finished, switch to the *Generator Model* (see Figure 10.7). Open the context menu on the root element and choose `Generate Pivot Model adapters` like shown in Figure 10.8. A new plug-in is created. It is named after the conventions of the Dresden OCL: `tudresden.ocl20.pivot.metamodels.<meta-model name>`. The package explorer is shown in Figure 10.9.

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 OCL.

10.2 SUMMARY

This chapter explained, how an annotated Ecore model can be used to generate adapter stubs for a meta-model that shall be adapted to the pivot model of Dresden OCL. For further implementation details investigate the source code of the existing adapted meta-models, such as Java, UML and Ecore. To test the new adapter, read Chapter 14 that introduces the *Generic Meta-Model Test Suite* that allows test-driven development of the adapter code.

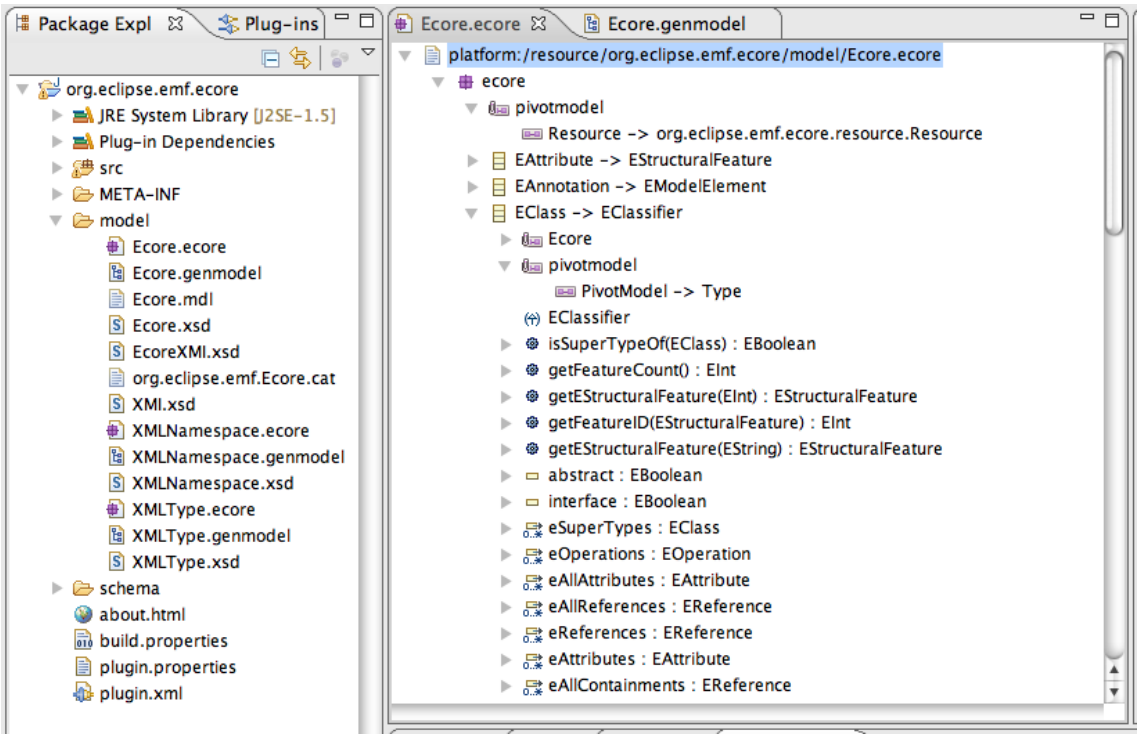

Figure 10.6: The EClass annotation.

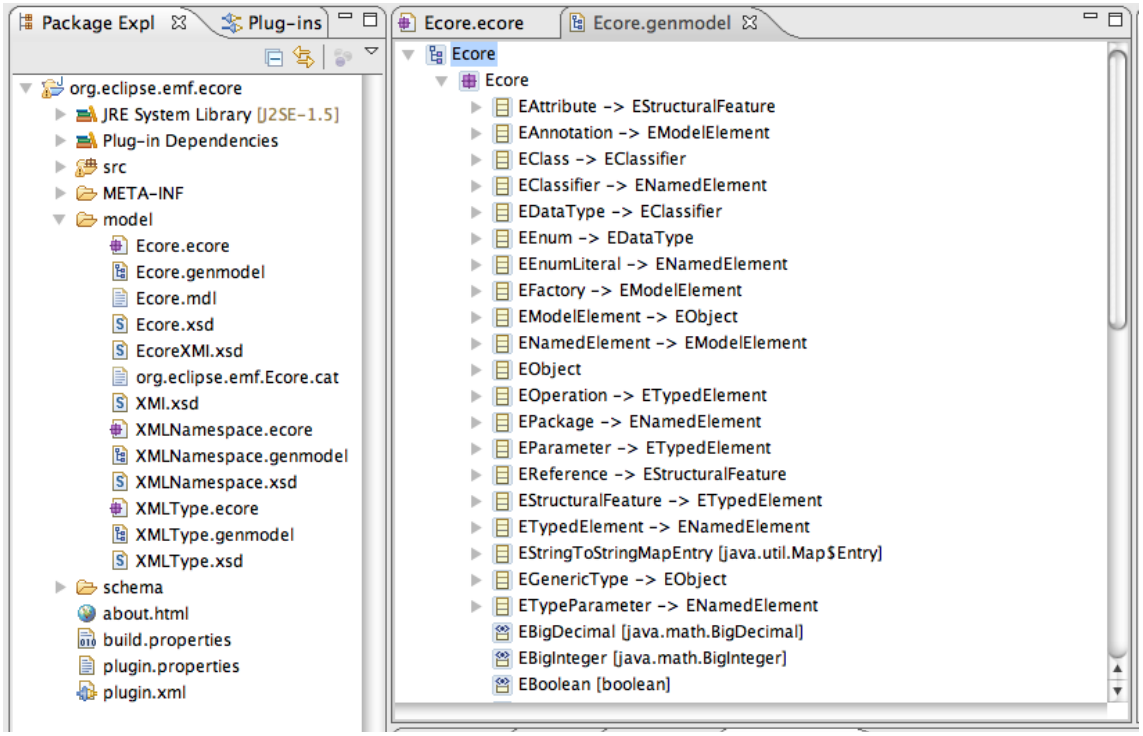

Figure 10.7: The genmodel of the Ecore model.

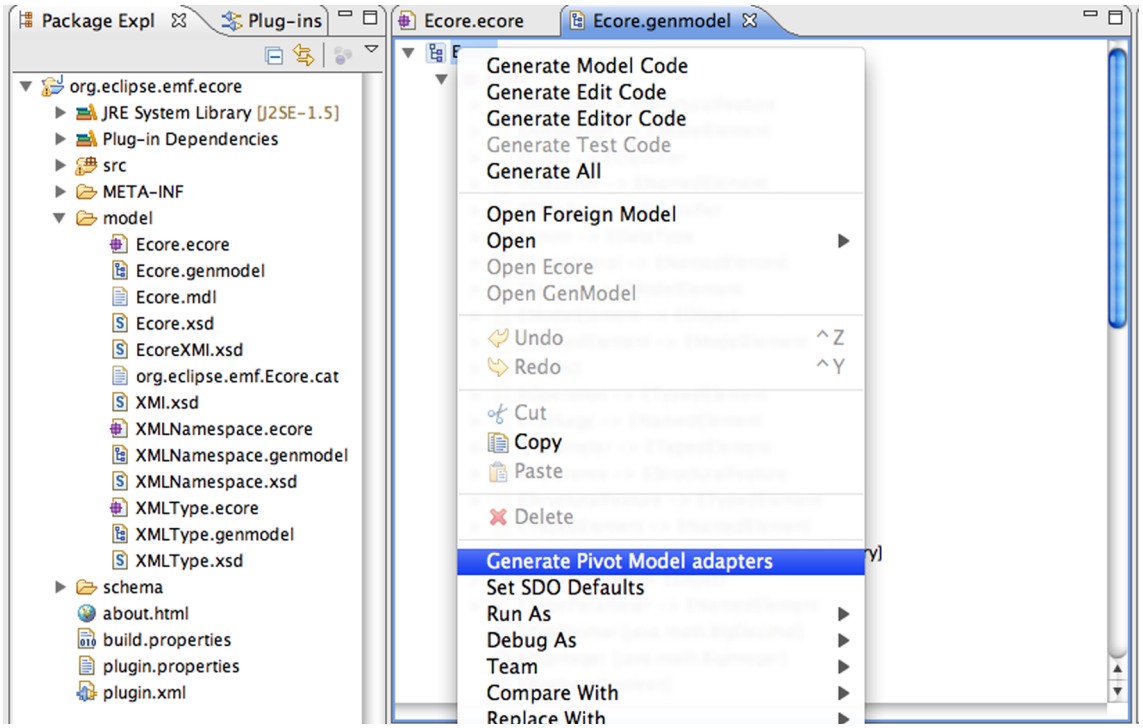

Figure 10.8: Right-click on Ecore and select 'Generate Pivot Model adapters'.

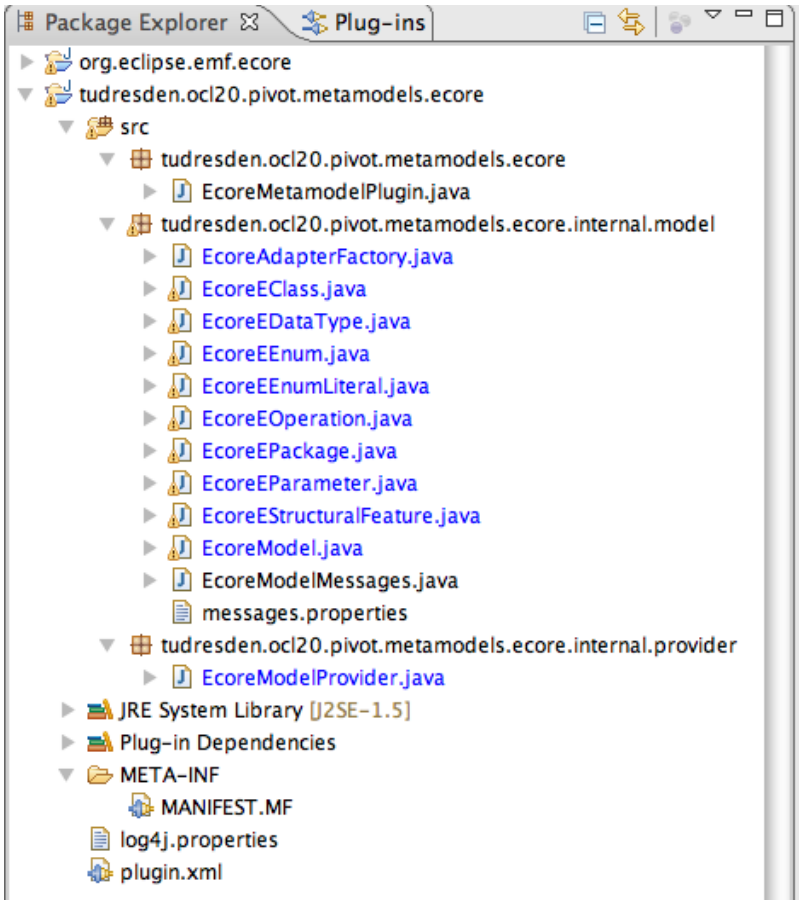

Figure 10.9: The structure of the generated plug-in.

11 ADAPTING A MODEL INSTANCE TYPE TO DRESDEN OCL

Chapter written by Claas Wilke

As mentioned in Chapter 7, Dresden OCL 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 Dresden OCL abstracts the instance's elements as `IModelInstanceElements`. Thus, each type of model instance that shall be connected with the OCL 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.

11.1 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 OCL. Figure 11.1 shows all different types of `IModelInstanceElements` that can exist. The different types are explained in the following.

11.1.1 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 by every adapter. Nevertheless, they are presented for completeness reasons.

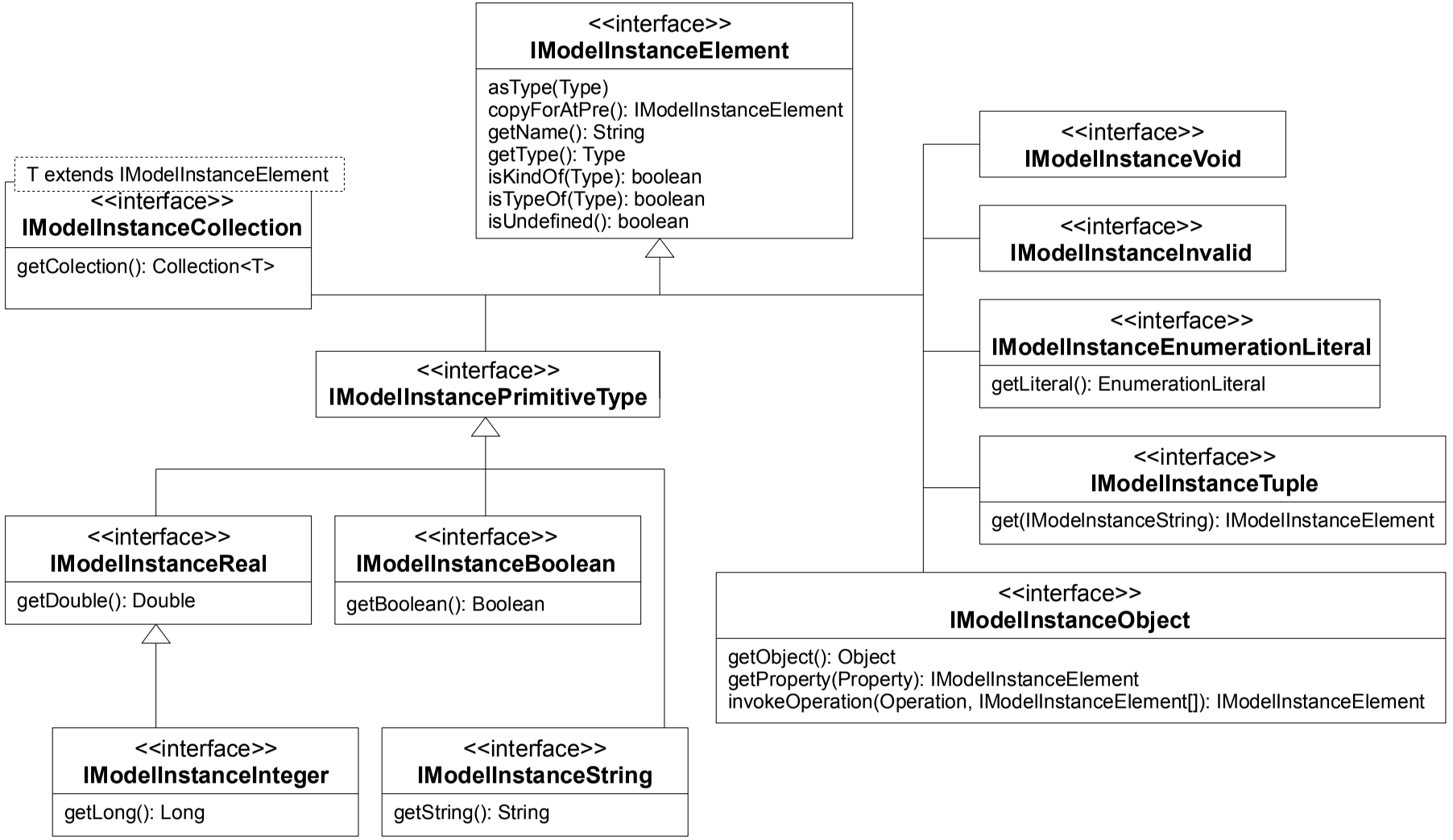

Figure 11.1: The different types of IModelInstanceElements.

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 visible for every class. 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.

getType()

The method `getType()` returns the implemented `Type` of the element.

isKindOf(Type) and isTypeOf(Type)

The methods `isKindof(Type)` and `isTypeOf(Type)` are required to check if an `IModelInstanceElement` 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.

11.1.2 The Adaptation of Model Instance Objects

The most important `IModelInstanceElement` is the `IModelInstanceObject`. It encapsulates the standard objects of a model instance such as a Java `Object` or an EMF Ecore `EObject`. An `IModelInstanceObject` implementation must be provided by every model instance type because without this kind of element a model implementation 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 properties' 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 of a collection type 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's value.

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 of a collection type 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` 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.

11.1.3 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 any 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.modelinstance` can be reused.

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

11.1.4 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`.

11.1.5 IModelInstanceEnumerationLiteral

The interface `IModelInstanceEnumerationLiteral` represents instances of `Enumerations`. Because enumerations do not have any state, they do not need any adapted `Object`. Thus, a standard `ModelInstanceEnumerationLiteral` implementation located in the plug-in `tudresden.ocl20.pivot.modelinstance` 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, EMF Ecore and XML.

11.1.6 IModelInstanceTuple

The interface `IModelInstanceTuple` represents key (`IModelInstanceString`) value (`IModelInstanceElement`) data structures called *Tuples*. Tuples are required during OCL interpretation only and cannot be part of a model instance. Thus, a standard `ModelInstanceTuple` implementation located in the plug-in `tudresden.ocl20.pivot.modelinstance` exists.

11.1.7 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. E. g., the `IModelInstanceVoid` instance is required when a method's invocation shall return a `null` value.

11.2 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.

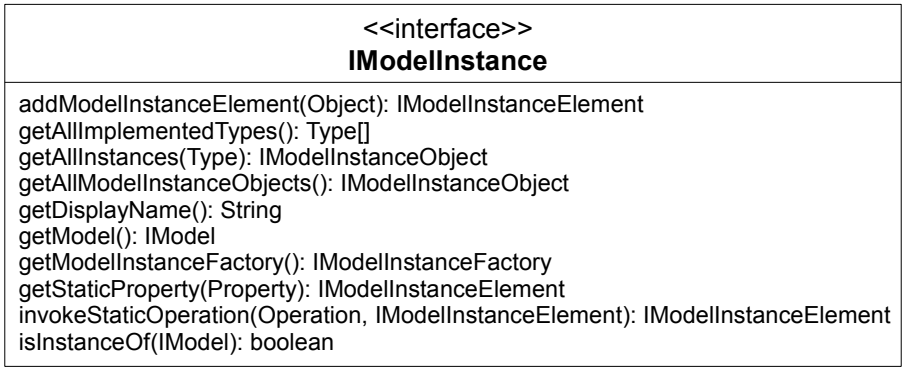

Figure 11.2: The IModelInstance Interface.

11.2.1 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, EMF Ecore and XML.

11.2.2 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)`.

11.3 THE IMODELINSTANCE INTERFACE

Figure 11.2 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.

11.3.1 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.

11.3.2 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.

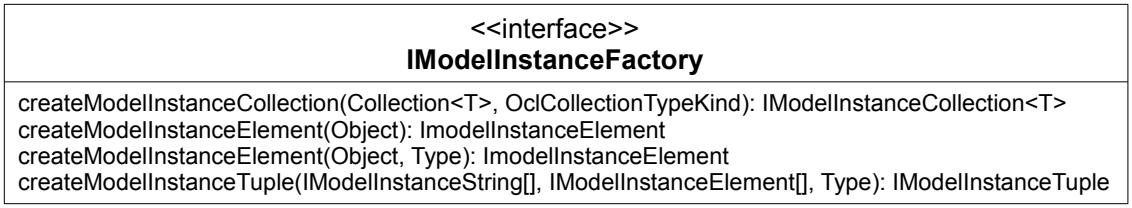

Figure 11.3: The IModelInstance Interface.

11.3.3 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 of a collection type 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.

11.3.4 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 of a collection type 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.

11.4 THE IMODELINSTANCEFACTORY INTERFACE

The `IModelInstanceFactory` implementation of a model instance is responsible to adapt 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 11.3. 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, Java and XML.

11.5 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 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 OCL and you can reuse the OCL Parser and OCL 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`), EMF Ecore (`tudresden.ocl20.pivot.modelinstancetype.ecore`) and XML (`tudresden.ocl20.pivot.modelinstancetype.xml`). To check your adaptation, have a look at the *Generic Model Implementation Type Test Suite* (presented in Chapter 15) as well.

12 THE LOGGING MECHANISM OF DRESDEN OCL

Chapter written by Claas Wilke

Dresden OCL uses a *Log4j Logger* to log method entries, exits and errors during Dresden OCL's execution. Each plug-in of Dresden OCL provides a `log4j.properties` file that can be used to configure the logger for the plug-in. Listing 12.1 shows such file of the Model Bus plug-in.

The last line of this configuration file (line 25) configures the logger. You can configure the logger to log messages of three different levels (`info`, `debug` and `error`). Furthermore, you can use different appenders to collect the logging messages such as the console (`stdout`) or an socket-based logging service (`socket`). The standard configuration of Dresden OCL's plug-ins is to log only error messages using the `stdout` appender.

If you use the `socket` appender instead, you might receive exceptions on the console like the 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 [**?**]. If you start Chainsaw, set up a *SocketReceiver* at port *4445 (Old Style/Standard Chainsaw Port)* (see Figure 12.1). Afterwards, the logging error message should not occur anymore.

Listing 12.1: The Log4J Configuration of the ModelBus plug-in

```
1  ### stdout appender ###
2  log4j.appender.stdout = org.apache.log4j.ConsoleAppender
3  log4j.appender.stdout.Target = System.out
4  log4j.appender.stdout.layout = org.apache.log4j.PatternLayout
5  log4j.appender.stdout.layout.ConversionPattern = %p %c: %m%n
6
7  # the error log appender
8  log4j.appender.errorLog = tudresden.ocl20.logging.appender.
       ErrorLogAppender
9  log4j.appender.errorLog.layout = org.apache.log4j.PatternLayout
10 log4j.appender.errorLog.layout.ConversionPattern = %c: %m%n
11
12 # the plugin log file appender
13 log4j.appender.pluginLogFile = tudresden.ocl20.logging.appender.
       PluginLogFileAppender
14 log4j.appender.pluginLogFile.File = plugin.log
15 log4j.appender.pluginLogFile.layout = org.apache.log4j.PatternLayout
16 log4j.appender.pluginLogFile.layout.ConversionPattern = %d{DATE} %p %c: %
       m%n
17
18 # socket appender
19 log4j.appender.socket=org.apache.log4j.net.SocketAppender
20 log4j.appender.socket.RemoteHost=localhost
21 log4j.appender.socket.Port=4445
22 log4j.appender.socket.LocationInfo=true
23
24 log4j.logger.tudresden.ocl20.pivot.modelbus = error,stdout
```

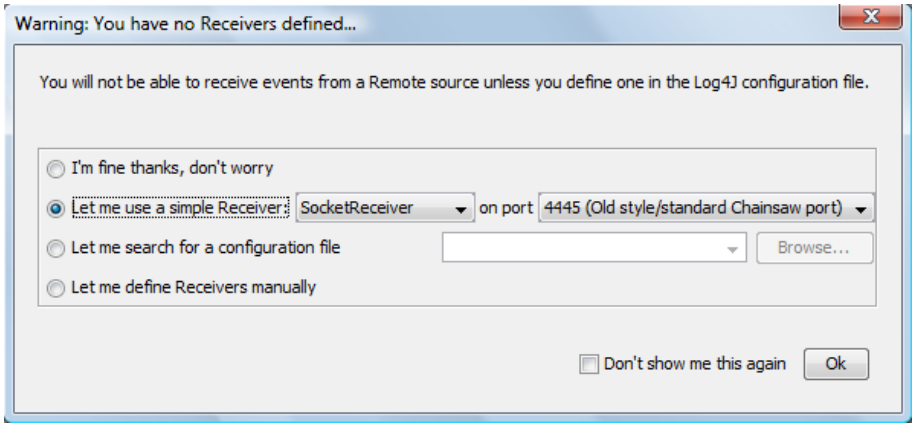

Figure 12.1: Setting up a simple SocketReceiver in Chainsaw.

13 THE EXTENSIBLE TEST SUITE OF DRESDEN OCL

Chapter written by Michael Thiele

Dresden OCL 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 OCL 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 `log4j.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, open the context menu on the class `OCL2TestSuiteRunner` of the package `tudresden.ocl20.pivot.testsuite.runner` in the *Package Explorer*. Choose `Run As ->JUnit Plug-in Test` as shown in Figure 13.1.

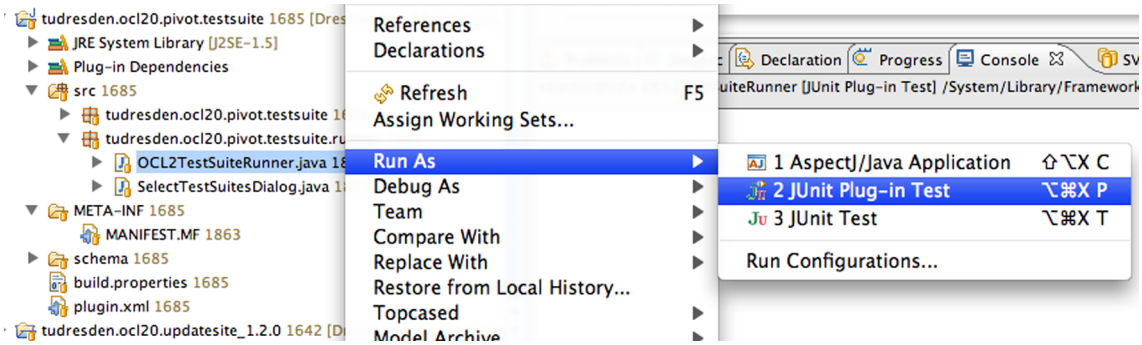

Figure 13.1: Run the Extensible Test Suite.

14 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 OCL, 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.

14.1 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 implemented 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 10. The test suite contains about 150 Junit tests.

To instantiate the generic test suite for a newly 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 14.1). Both, the model and the Java class are shortly presented in the following sections.

14.2 THE REQUIRED MODEL TO TEST A META-MODEL

Figure 14.2 shows the test model that must be modeled using the 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 can provide an enumeration type but does not have to. If a structure is not provided by a

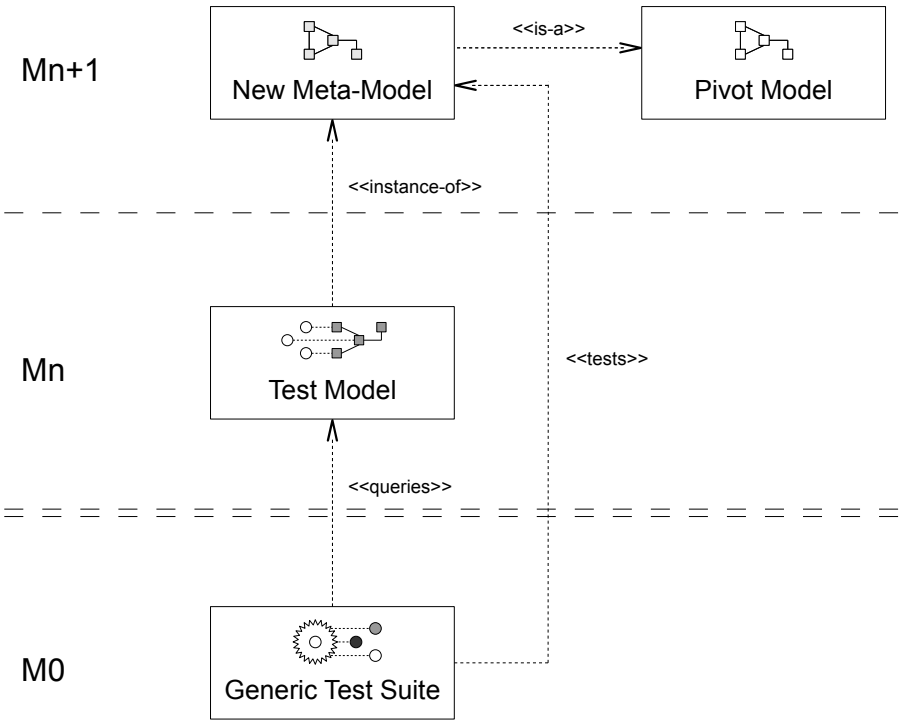

Figure 14.1: The Generic Meta-Model Test Suite in respect to the Generic Three Layer Architecture (as presented in Section 7.1).

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 warning can be ignored. In the following, all types and relations of the test model are explained shortly.

14.2.1 TestTypeClass1 and TestTypeClass2

As their names already tell 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.

14.2.2 TestTypeInterface1 and TestTypeInterface2

Besides classes, some meta-models also provide a second type that must be mapped to the pivot model's interface `Type`, which are `Interfaces`. 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`.

14.2.3 TestEnumeration

To test the adaptation of an `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.

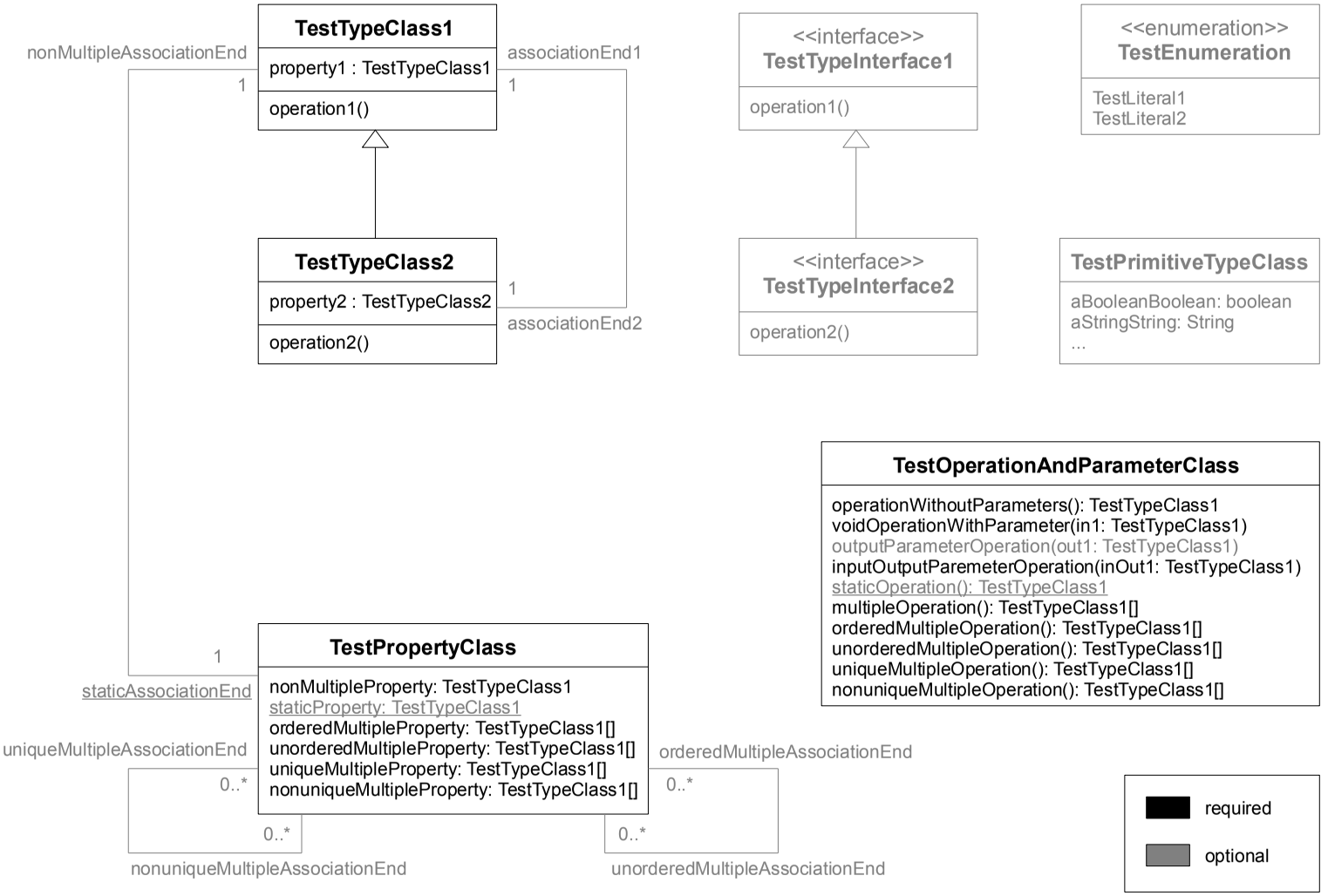

Figure 14.2: The required Test Model to test a Meta-Model's adaptation. The gray parts are optional.

14.2.4 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 14.2 is tested as adapted to a `String`. Table 14.1 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 14.1: The adaptation of properties' name prefixes to PrimitiveTypeKinds.

14.2.5 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.

14.2.6 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.

14.3 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 modeled useing the adapted meta-model. Listing 14.1 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 has to provide a `setUp()`

```
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 14.1: An instantiation of the generic meta-model test suite.

method only that can be used to setup the test suite before its 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.

14.4 SUMMARY

This chapter shortly introduced into the generic meta-model test suite of Dresden OCL. 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`.

15 THE GENERIC MODEL INSTANCE TYPE TEST SUITE

Chapter written by Claas Wilke

To test the adaptation of a model instance type to Dresden OCL, 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.

15.1 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 OCL. The adaptation of a model instance type to Dresden OCL is explained in Chapter 11. The test suite contains about 100 JUnit tests.

To instantiate the generic test suite for a newly 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 15.1). Both, the model instance and the Java class are shortly presented in the following sections.

15.2 THE REQUIRED MODEL INSTANCE TO TEST A MODEL INSTANCE TYPE

The Figures 15.2 and 15.3 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

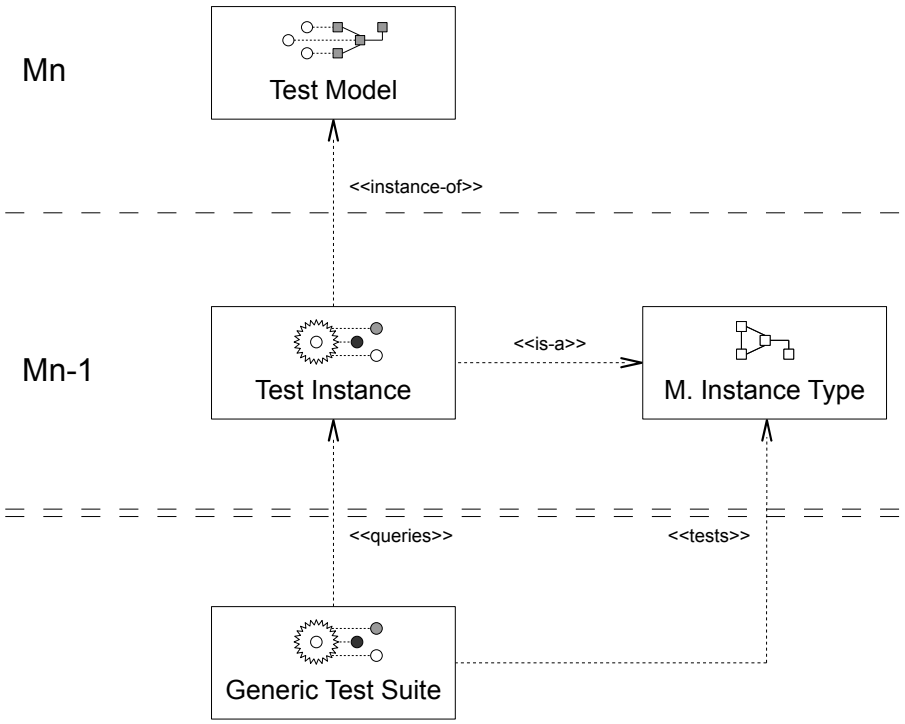

Figure 15.1: The Generic model instance type Test Suite in respect to the Generic Three Layer Architecture (as presented in Section 7.1).

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`).

15.2.1 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. The `ContainerClass` should be instantiated by at least one object of the model instance. **Please be aware that if the collections containing the other instance types' instances are not instantiated appropriately, the whole test suite may fail.**

15.2.2 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 an enormous set of operations as well. Some operations are responsible to test the invocation of operations with different result types (`voidOperation()`,

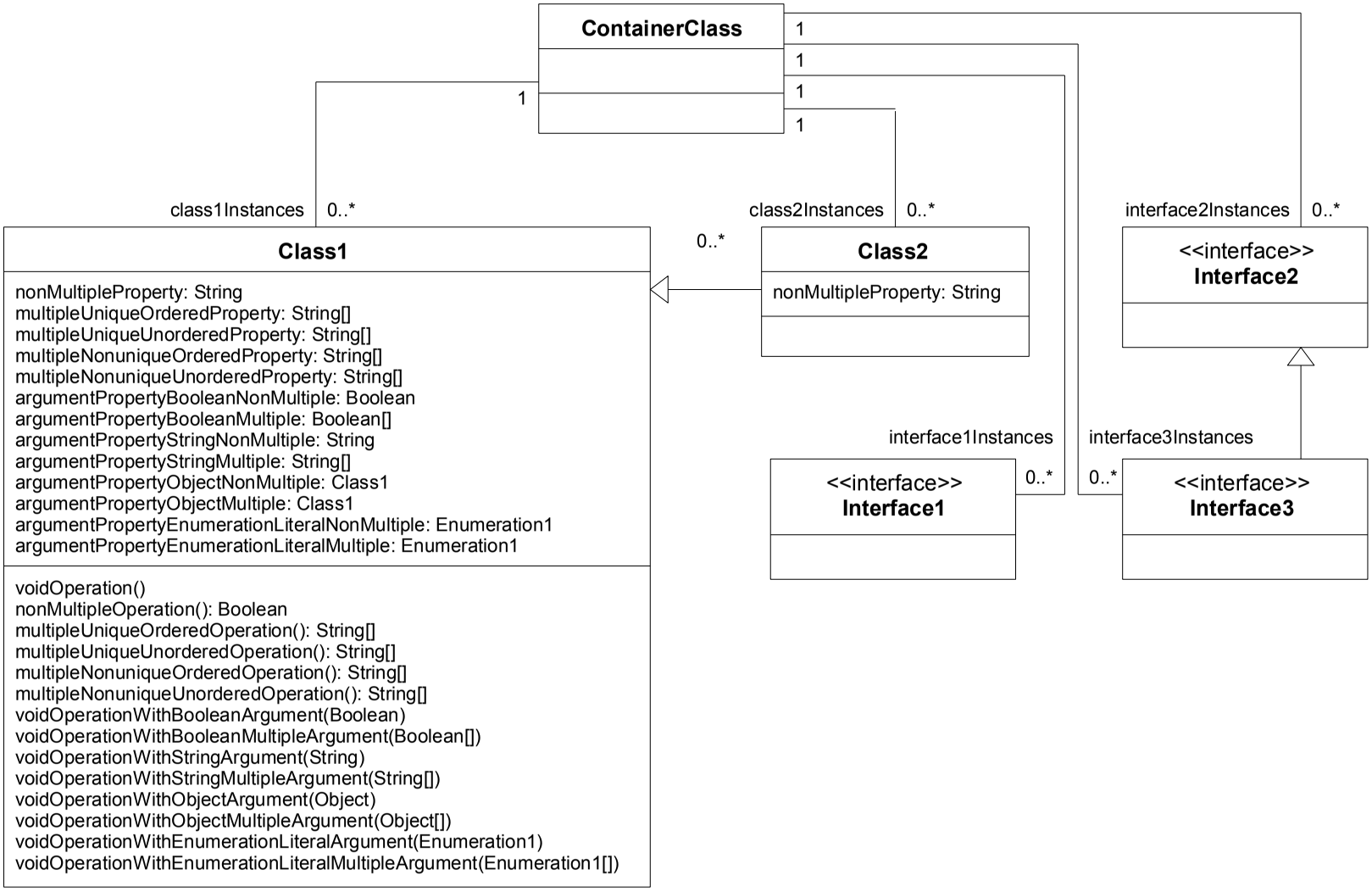

Figure 15.2: The required Test Model to test a model instance type's adaptation (part 1).

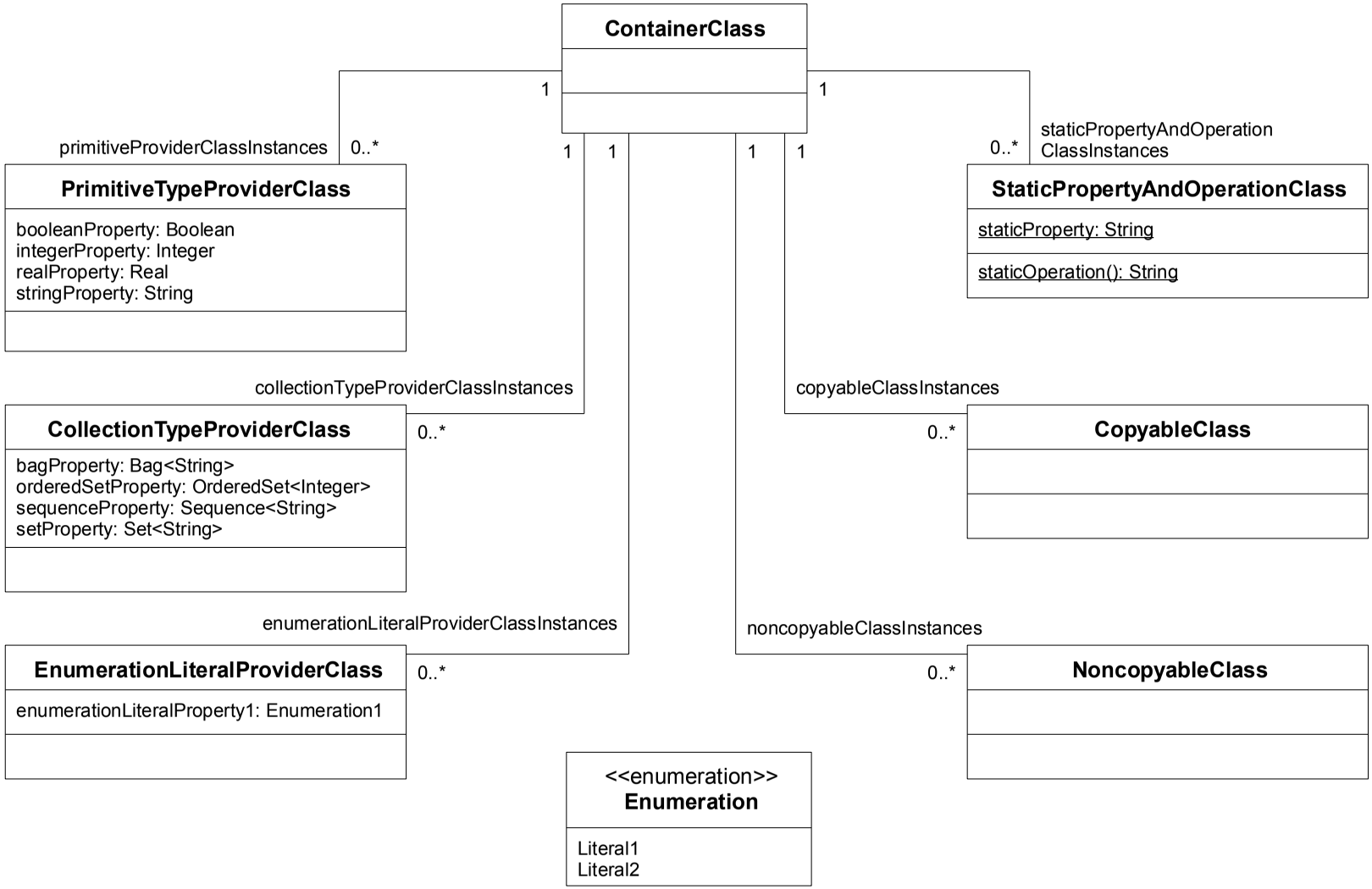

Figure 15.3: The required Test Model to test a model instance type's adaptation (part 2).

nonMultipleOperation() and multiple...Operation()), others are required to test the invocation of operations with different argument types (voidOperationWith...Argument()).

15.2.3 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.

15.2.4 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 extend 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.

15.2.5 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 provide 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 OCL (e.g., the types int and java.lang.Integer of Java should both be mapped 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 15.3).

15.2.6 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.

15.2.7 Copyable- and NonCopyableClass

OCL supports the special operation `@pre` in postconditions, that can be used in OCL to access to values of properties' values before the execution of an operation that is the current context of the postcondition. To support this operation, the OCL 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.

15.3 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 15.1 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 its 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 can 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 Section 15.2.5).

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.

15.4 SUMMARY

This chapter shortly introduced into the generic model instance type test suite of Dresden OCL. 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`.

```
1  import tudresden.ocl20.pivot.modelinstancetype.test.
2         ModelInstanceTypeTestPlugin;
3  import tudresden.ocl20.pivot.modelinstancetype.test.
4         ModelInstanceTypeTestServices;
5  import tudresden.ocl20.pivot.modelinstancetype.test.
6         ModelInstanceTypeTestSuite;
7
8  @Suite.SuiteClasses(value = { ModelInstanceTypeTestSuite.class })
9  public class TestJavaModelInstanceType extends
10     ModelInstanceTypeTestSuite {
11
12   /** The id of the {@link IModelInstanceTypeObject}
13     * which shall be tested. */
14   private static final String MODEL_INSTANCE_ID =
15     JavaModelInstanceTypePlugin.PLUGIN_ID;
16
17   /** The path of the model which shall be tested. */
18   private static final String TEST_MODELINSTANCE_PATH =
19     "bin/tudresden/ocl20/pivot/modelinstancetype/" +
20     "java/test/modelinstance/ProviderClass.class";
21
22   /**
23    * <p>
24    * Prepares the {@link ModelInstanceTypeTestSuite}.
25    * </p>
26    */
27   @BeforeClass
28   public static void setUp() {
29
30     ModelInstanceTypeTestPlugin.prepareTest(
31       JavaModelInstanceTypeTestPlugin.PLUGIN_ID,
32       TEST_MODELINSTANCE_PATH, MODEL_INSTANCE_ID);
33
34     ModelInstanceTypeTestServices.getInstance()
35       .setBooleanPropertyCounter(2);
36     ModelInstanceTypeTestServices.getInstance()
37       .setIntegerPropertyCounter(10);
38     ModelInstanceTypeTestServices.getInstance()
39       .setRealPropertyCounter(4);
40     ModelInstanceTypeTestServices.getInstance()
41       .setStringPropertyCounter(4);
42     ModelInstanceTypeTestServices.getInstance()
43       .setSequencePropertyCounter(2);
44   }
45 }
```

Listing 15.1: An instantiation of the generic model instance test suite.

III APPENDIX

TABLES

Software	Available at
Eclipse 3.6.x	http://www.eclipse.org/
Eclipse Modeling Framework (EMF)	http://www.eclipse.org/modeling/emf/
Eclipse Model Development Tools (MDT) (required for 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/
AspectJ Development Tools (AJDT) (only to run the generated code of Ocl22Java)	http://www.eclipse.org/ajdt/

Table 1: Software required to run Dresden OCL (v. 3.x) with Eclipse
(Most parts are contained in the Eclipse MDT Distribution).

Feature	Plug-ins
Codegen	(Required for code generation) tudresden.ocl20.pivot.tools.codegen tudresden.ocl20.pivot.tools.codegen.ui tudresden.ocl20.pivot.tools.template tudresden.ocl20.pivot.tools.template.test tudresden.ocl20.pivot.tools.transformation tudresden.ocl20.pivot.tools.transformation.test
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.essentialocl.tests tudresden.ocl20.pivot.examples.royalandloyal tudresden.ocl20.pivot.model tudresden.ocl20.pivot.modelbus tudresden.ocl20.pivot.modelbus.tests tudresden.ocl20.pivot.modelbus.ui tudresden.ocl20.pivot.modelinstance tudresden.ocl20.pivot.modelinstancetype tudresden.ocl20.pivot.pivotmodel tudresden.ocl20.pivot.pivotmodel.edit tudresden.ocl20.pivot.pivotmodel.tests tudresden.ocl20.pivot.standardlibrary.java tudresden.ocl20.pivot.standardlibrary.java.tests tudresden.ocl20.pivot.testsuite
Examples	(Optional) tudresden.ocl20.pivot.examples.living tudresden.ocl20.pivot.examples.pain tudresden.ocl20.pivot.examples.pml tudresden.ocl20.pivot.examples.simple tudresden.ocl20.pivot.examples.ui tudresden.ocl20.pivot.examples.university
Integration Facade	(Optional) tudresden.ocl20.pivot.facade
Interpreter	(Required for interpretation) tudresden.ocl20.interpreter tudresden.ocl20.interpreter.test tudresden.ocl20.interpreter.ui
Metamodels	(Required) tudresden.ocl20.pivot.codegen.adapter tudresden.ocl20.pivot.metamodels.ecore tudresden.ocl20.pivot.metamodels.ecore.test tudresden.ocl20.pivot.metamodels.java tudresden.ocl20.pivot.metamodels.java.test tudresden.ocl20.pivot.metamodels.test tudresden.ocl20.pivot.metamodels.uml2 tudresden.ocl20.pivot.metamodels.uml2.test tudresden.ocl20.pivot.metamodels.xsd

Table 2: The plug-ins of Dresden OCL (v. 3.x) related to their features (part 1).

Feature	Plug-ins
Model Instances	(Required) tudresden.ocl20.pivot.modelinstancetype.ecore tudresden.ocl20.pivot.modelinstancetype.java tudresden.ocl20.pivot.modelinstancetype.test tudresden.ocl20.pivot.modelinstancetype.test.ecore tudresden.ocl20.pivot.modelinstancetype.test.java tudresden.ocl20.pivot.modelinstancetype.test.xml tudresden.ocl20.pivot.modelinstancetype.xml
Ocl2Sql	(Required SQL code generation) tudresden.ocl20.pivot.tools.codegen.declarativ tudresden.ocl20.pivot.tools.codegen.declarativ.ocl2sql tudresden.ocl20.pivot.tools.codegen.declarativ.ocl2sql.test tudresden.ocl20.pivot.tools.codegen.declarativ.ocl2sql.ui tudresden.ocl20.pivot.tools.CWM tudresden.ocl20.pivot.tools.template.sql tudresden.ocl20.pivot.tools.transformation.pivot2sql tudresden.ocl20.pivot.tools.transformation.pivot2sql.test
Ocl2Java	(Required for Java/AspectJ code generation) tudresden.ocl20.pivot.tools.codegen.ocl2java tudresden.ocl20.pivot.tools.codegen.ocl2java.types tudresden.ocl20.pivot.tools.codegen.ocl2java.test tudresden.ocl20.pivot.tools.codegen.ocl2java.ui
Ocl22Java Examples	(Optional) tudresden.ocl20.pivot.examples.ocl22java tudresden.ocl20.pivot.examples.royalandloyal.ocl22javacode tudresden.ocl20.pivot.examples.simple.ocl22javacode tudresden.ocl20.pivot.tools.codegen.ocl22java.test.aspectj
OCL2 Parser	(Required) org.emftext.commons.antlr3_2_0 (provided by EMFText) org.kiama.attribution scala.library tudresden.ocl20.pivot.language.ocl tudresden.ocl20.pivot.language.ocl.resource.ocl tudresden.ocl20.pivot.language.ocl.resource.ocl.ui tudresden.ocl20.pivot.language.ocl.semantics tudresden.ocl20.pivot.language.ocl.staticsemantics tudresden.ocl20.pivot.ocl2parser.test tudresden.ocl20.pivot.parser tudresden.ocl20.pivot.pivotmodel.semantics

Table 3: The plug-ins of Dresden OCL (v. 3.x) related to their features (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
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
Generated AspectJ Code	tudresden.ocl20.pivot.examples.royalandloyal.ocl22javacode
SEPA Example	(Provided by Nomos Software (`http://www.nomos-software.com/`))
Plug-in Package	tudresden.ocl20.pivot.examples.pain
Meta-Model	XSD (XML Schema)
Model	model/pain.008.001.01corrected.xsd
OCL Expressions	constraints/*.ocl
Model Instance Type	XML
Model Instances	modelinstances/BusEx1.xml, modelinstances/BusEx2.xml, modelinstances/Pain8Invalid.xml
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
Generated AspectJ Code	tudresden.ocl20.pivot.examples.simple.ocl22javacode
University Example	
Plug-in Package	tudresden.ocl20.pivot.examples.university
Meta-Models	MDT UML2
Model	model/university.uml, model/university_complex.uml
OCL Expressions	constraints/*.ocl
Model Instance Type	–
Model Instance	–

Table 4: The examples provided with Dresden OCL.

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ABBREVIATIONS

AJDT	AspectJ Development Tools
API	Application Programming Interface
AOP	Aspect-Oriented Programming
BRs	Business Rules
DOT4Eclipse	Dresden OCL2 for Eclipse
DSL	Domain-Specific Language
Eclipse MDT	Eclipse Modeling Development Tools
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
JAR	Java Archive
JDK	Java Development Kit
JRE	Java Run-time Environment
MDT	Modeling Development Tools
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OSGi	Open Services Gateway initiative
SVN	Subversion
UML	Unified Modeling Language
URL	Uniform Resource Locator
WFRs	Well-Formedness Rules
XSD	XML Schema Definition
XMI	XML Metadata Interchange
XML	Extensible Markup Language

ACKNOWLEDGEMENTS

We like to thank all people that are or have been involved into the development of Dresden OCL. Below is a list (alphabetical order) of people involved. If we forgot to mention a person please do not blame but inform us.

- Ronny Brandt
- Matthias Bräuer
- Birgit Demuth
- Katrin Eisenreich
- Frank Finger
- Björn Freitag
- Kai Uwe Gärtner
- Florian Heidenreich
- Heinrich Hußmann
- Sebastian Käppler
- Andreas Kling
- Ansgar Konemann
- Nikolai Krambrock
- Martin Krebs
- Sten Löcher
- Ronny Marx
- Michael Muck
- Christian Nil
- Sven Obermaier
- Stefan Ocke

- Andreas Schmidt
- Lars Schütze
- Irina Soudnik
- Mirko Stölzel
- Michael Thiele
- Niels Thieme
- Christian Wende
- Ralf Wiebicke
- Claas Wilke
- Steffen Zschaler