



User Manual

Oclarity
for Rational Rose™

Version 1.8

Copyright Notice

© 2004-2007 EmPowerTec AG, Taubenweg 20, 85238 Petershausen, Germany.

All rights reserved. This product and related documentation are protected by copyright and are distributed under licenses restricting their use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of EmPowerTec AG, and its licensors, if any.

Third Party Website Reference

EmPowerTec AG is not responsible for the availability of third-party Web sites mentioned in this document. EmPowerTec AG does not endorse and is not responsible or liable for any content, advertising, products, or other material on or available from such sites or resources. EmPowerTec AG will not be responsible or liable for any damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through any such sites or resources.

Trademarks

EmPowerTec is a trademark of EmPowerTec AG.

Rational Rose™ is a trademark of IBM Inc.

Other brands and their products are trademarks of their respective holders and should be noted as such.

1	Introduction	4
2	Installation	5
2.1	Requirements	5
2.2	Installation	5
2.3	Updates	5
2.4	Installing the license key.....	5
2.5	Uninstalling	5
2.6	Printing this manual.....	6
3	The light version	7
4	Adding OCL expressions to a model.....	8
4.1	Introduction	8
4.2	Adding OCL expressions in notes.....	8
4.3	Add OCL expressions to classes.....	9
5	Using the OCL editor	11
5.1	Introduction	11
5.2	Basic editor window.....	11
5.3	Menu 'code'	12
5.4	Menu 'edit'	12
5.5	Leaving the code editor.....	12
5.6	Editing code in notes.....	12
6	Checking OCL code	14
6.1	Introduction	14
6.2	Triggering checks	14
6.3	Definition expressions.....	16
6.4	Viewing check results.....	17
6.4.1	Warnings.....	18
6.4.2	Check again.....	18
6.4.3	First error.....	18
6.4.4	Next error.....	19
6.4.5	Previous error.....	19
6.4.6	Editing code from the result window	19
7	Exporting OCL expressions	20
7.1	Triggering the export.....	20
7.2	Structure of the XML file.....	21
8	Integration with the Rose repository.....	22
8.1	Overview	22
8.2	The OCL-property editor for operations	22
8.3	Package handling	24
8.3.1	Package names with spaces	24
8.3.2	Default package.....	24
8.4	Mapping to Rose properties.....	24
8.5	Enumerations	24
8.6	Attributes of stereotype <<reference>>	25
8.7	States	25
8.8	Instantiated types.....	25

1 Introduction

OCL is a formal language intended to phrase expressions in object models, in particular in UML models. The purpose of OCL is to add precision to a model and complement the better known UML diagrams and use cases. If used properly, adding OCL expressions to a UML model can significantly increase the precision and ultimately the value of a UML model. Furthermore, the communication between the persons working on a software project is improved because all business logic described with OCL is available for all contributors to your project at any time.

OCL expressions can be used with varying intentions. Best known is the use as 'constraints'. Constraints state conditions that must be true in certain points of time. Other applications of OCL expressions are initialization expressions for attributes and associations, derivation rules for additional auxiliary attributes and methods and the language independent description of method implementations.

OCL is language independent and thus helps you to specify more knowledge at the abstract level of the UML model than without using OCL. Instead of burying the business logic in complex programming language statements in software implementation files, it is stored at the heart of your software system: in your UML models.

It is beyond the scope of this document to provide an introduction to OCL. On our website <http://www.empowertec.de>, you can find links to OCL-resources.

Rational Rose™ does not support OCL by itself and thus adding OCL expressions to Rational Rose models is not reasonably feasible. Oclarity adds comprehensive support for adding OCL expressions to an OCL model:

- Context sensitive editor
- Full syntactic and semantic checking of OCL expressions ensures consistency with the model.
- Capability, to check all OCL expression in a model at once. This is particularly useful if properties of a model are changed, e.g. class- or attribute names, method signatures and so on.
- Smooth integration with the Rational Rose GUI.
- Flexible export capabilities.

2 Installation

2.1 Requirements

Oclarity requires an installation of Rational Rose 2000 or newer. Oclarity can be used with all language specific versions of Rational Rose.

2.2 Installation

Load the current version of Oclarity from our website using this link:

<http://www.empowertec.de/downloads/OclaritySetup.exe>

Execute the file.

There are no choices to make during the setup.

2.3 Updates

Regularly, we make new versions with the same major version number of Oclarity available for download. These new versions contain bug fixes and minor functional improvements.

Our license agreement authorizes you to install any available update with the same major version number than the version you have licensed. To install a new version simply download it from our website, uninstall the old version and install the new version as described in chapter 2.2.

2.4 Installing the license key

To use Oclarity after the 30 day trial period, a valid license key must be installed. License keys can be purchased in our online shop <http://www.empowertec.de/buy/> or by sending an email to <mailto:sales@empowertec.de>.

As long as you are using a trial key, Oclarity will offer you the possibility to install a valid license key at every startup of Rational Rose.

If Rational Rose is already started, you can use the menu entry Tools/OCL/About to display a dialog for installing a license key.

2.5 Uninstalling

To uninstall this software you may simply choose the according menu entry in the start menu.

Alternatively, you can use the Windows control panel to remove the software.

2.6 Printing this manual

If you print the manual with Adobe Acrobat Reader with default print settings, the pages are smaller than intended because Acrobat Reader scales the pages. To get a printout in original size, please uncheck all the scaling options in the Copies and Adjustments section of the Acrobat Reader print dialog box.

3 The light version

EmPowerTec provides a free light version of Oclarity which contains the same advanced OCL checker than the standard version but misses the productivity features of the standard version. The light version is intended for users that want to occasionally check OCL expressions and do not use OCL in commercial software projects where the time of the staff is the most crucial factor for project success.

When using the light version, OCL expressions are stored in external text files. These files contain just the raw OCL expressions as suggested in the OCL standard description.

The user interface consists of a simple window.

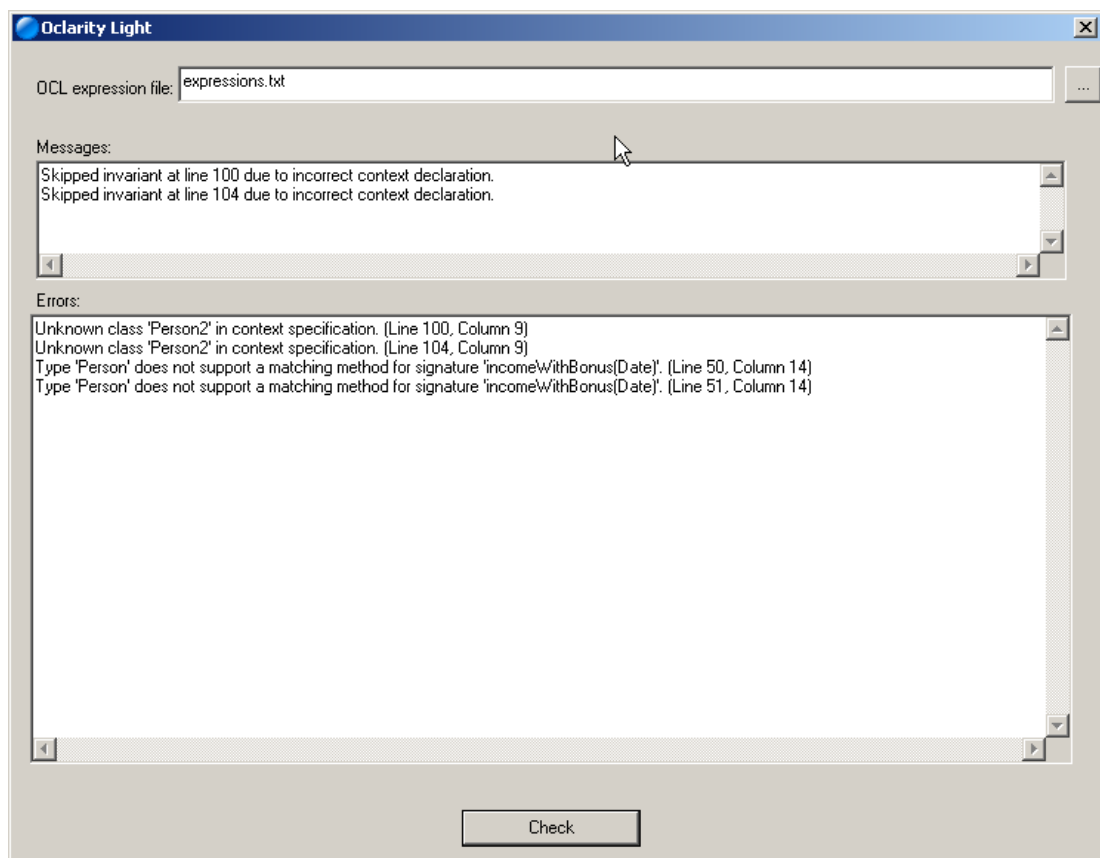


Figure 1: Oclarity Light user interface

To perform a check, the file with the expression that shall be checked must be selected and the 'Check' button must be pressed.

EmPowerTec is not obliged to give any support for the light version. However, feel free to send us any questions and problem reports. If we have free resources, we will try to solve your issues.

4 Adding OCL expressions to a model

4.1 Introduction

OCL expressions can be added in two basic ways.

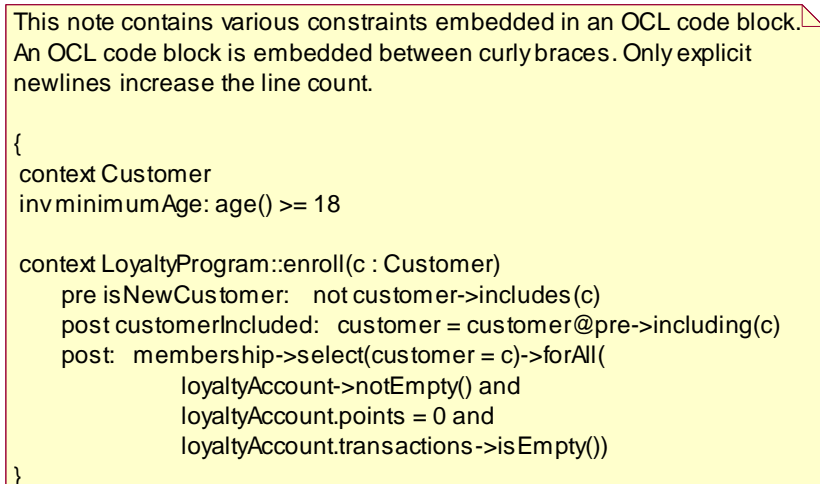
First, you can write OCL expressions in notes in diagrams. This is particularly useful if the expression is relatively small but important so that it should be immediately perceived.

Second, you can use the OCL editor that is integrated in Oclarity. Using the OCL editor, the OCL code is always associated with a specific class. This association is only a technical one, the semantic association is done by 'context' declarations within the OCL code. However, we recommend adding only those OCL expressions to a given class that are also semantically associated with this class.

4.2 Adding OCL expressions in notes

A note on a diagram can be used to hold all kind of text. Therefore, a convention has to be followed so that a portion of text is treated as OCL expression by our OCL parser. It is important to follow these conventions; otherwise, the OCL code would not be checked and could contain any kind of errors without notice.

The convention is to embed OCL code in curly brackets. Here is an example:



```
{
context Customer
inv minimumAge: age() >= 18

context LoyaltyProgram::enroll(c : Customer)
pre isNewCustomer: not customer->includes(c)
post customerIncluded: customer = customer@pre->including(c)
post: membership->select(customer = c)->forAll(
    loyaltyAccount->notEmpty() and
    loyaltyAccount.points = 0 and
    loyaltyAccount.transactions->isEmpty())
}
```

Figure 2: Editing OCL code in a note

It is not necessary to use a separate line for the starting and ending curly braces and the final curly brace may be omitted. Multiple blocks of OCL code may be defined in a single note although we recommend putting all OCL code in a single OCL block in a given note.

A note object reformats its content if it is resized by breaking the lines but it does not insert newline characters into the text. Therefore, you must explicitly insert newline characters (by pressing the return key) wherever you want the OCL code to start a new line. If an error is detected in such an OCL expression, the errors line number refers to the position of the offending line relative to the beginning of the note (that is, it includes all non OCL text).

4.3 Add OCL expressions to classes

Whenever you select a single class in the Rational Rose GUI, the selected item has an additional entry in its context menu labeled 'edit OCL'. This may be the case on a class diagram or in Rational Roses browser:

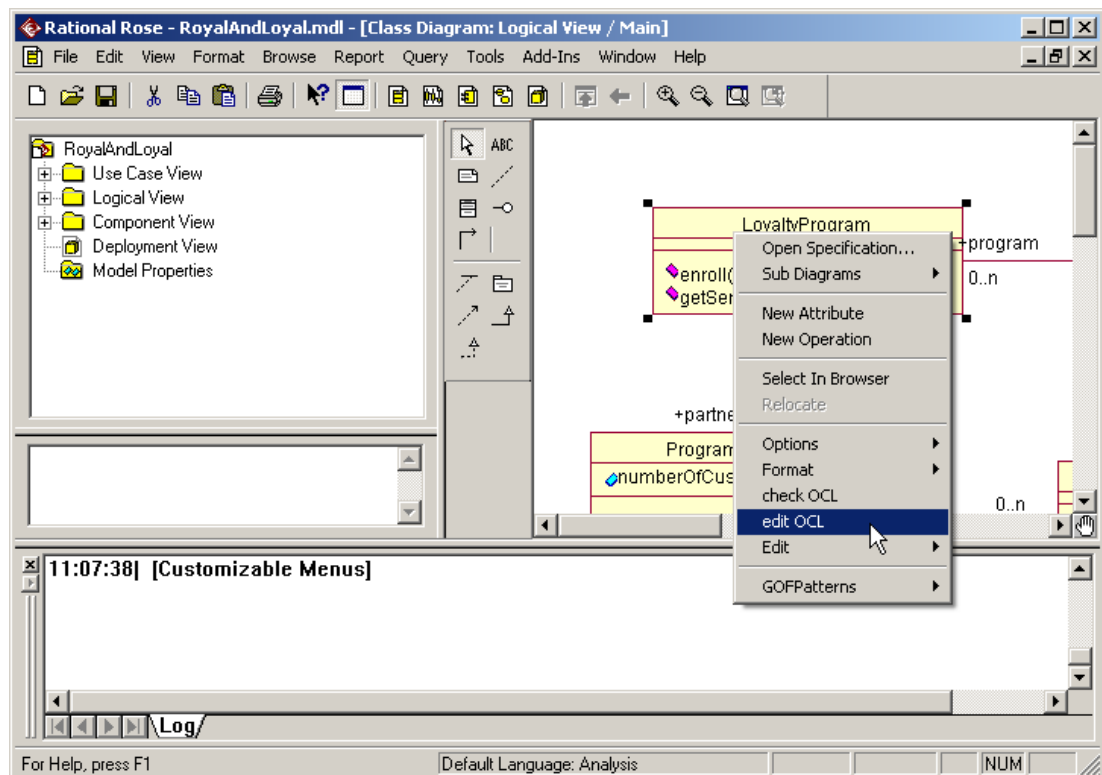


Figure 3: selected class in diagram

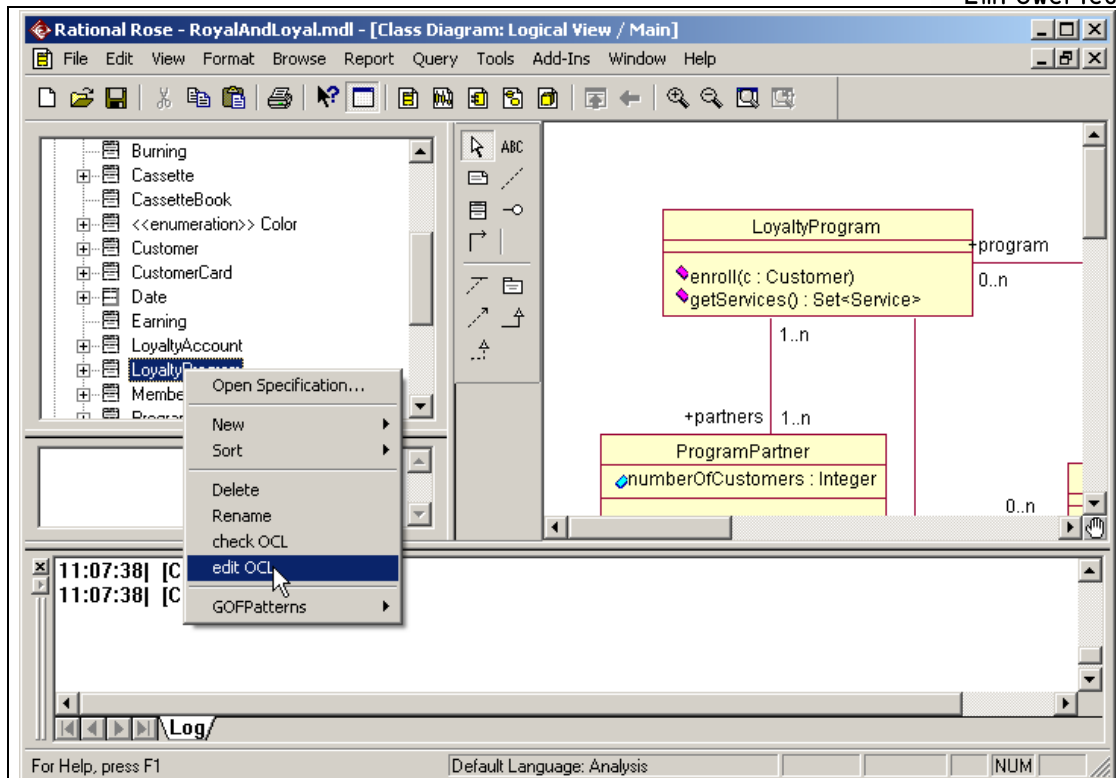


Figure 4: selected class in browser

In either case, after selecting the menu entry 'edit OCL', the OCL code editor is displayed and the according OCL code may be edited.

5 Using the OCL editor

5.1 Introduction

The OCL code editor offers powerful features for editing OCL code. This chapter introduces the various features.

5.2 Basic editor window

This screenshot illustrates the code editor:

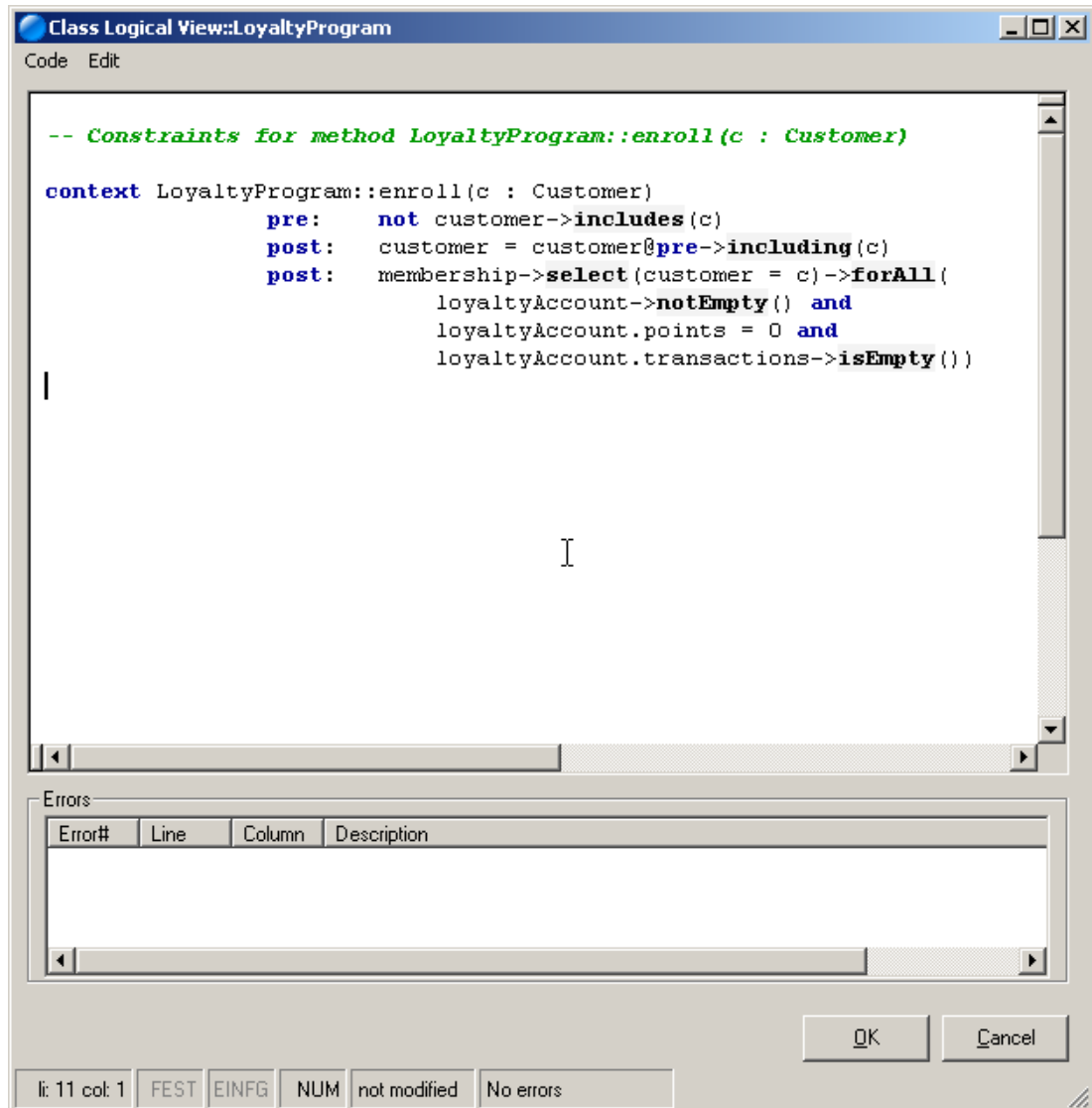


Figure 5: OCL editor window

The large area beneath the menu bar contains the actual OCL code, which is displayed with syntax highlighting. Underneath this area, a table is displayed that contains the errors that

have been detected during a check of the OCL code. If any errors are display, a double click on the line containing the error positions the cursor on the respective code element.

5.3 Menu 'code'

This menu contains a single entry that can be used to check the code. As a shortcut, you can press alt+c to trigger the check.

The result of the check is displayed in a panel in the status bar of the editor window. This panel is reset to an empty string, if the code is modified after the check.

5.4 Menu 'edit'

This menu provides all basic editing functionality:

- Undo/Redo
- Copy/Paste
- Search/Replace

5.5 Leaving the code editor

You can leave the code editor either by clicking on the 'OK'-button or by clicking on the 'cancel'-button (clicking in the editor windows close-button in the upper right corner is equivalent to clicking on the 'OK'-button).

If 'OK' is clicked the code is implicitly checked and a warning is displayed if the code contains errors.

You may then correct the errors or save the code as it is.

If the code is saved, this means only storing the changed code in Rational Roses internal repository - **the changed code is not saved to disk until the complete model is saved**. Rational Rose will warn you, if you want to leave Rational Rose without saving the changes to disk.

If 'cancel' is clicked and the code is changed, you are asked whether you really want to discard your changes.

5.6 Editing code in notes

Editing code in notes requires some special handling. Because a note may contain a mixture of OCL code and arbitrary other text, all non OCL code must be hidden from the parser.

Note that you cannot directly invoke the code editor for OCL code in notes (due to restrictions of Rational Roses extensibility model). Invoking the code editor for OCL code in notes is only possible from the analysis result window (see chapter 6.4 for details).

Oclarity temporarily prepends each line of non OCL code with the string '-- ##' ('-' starts a comment in OCL) thus making this text look like a comment to the parser. After closing the code editor, these marks are removed again.

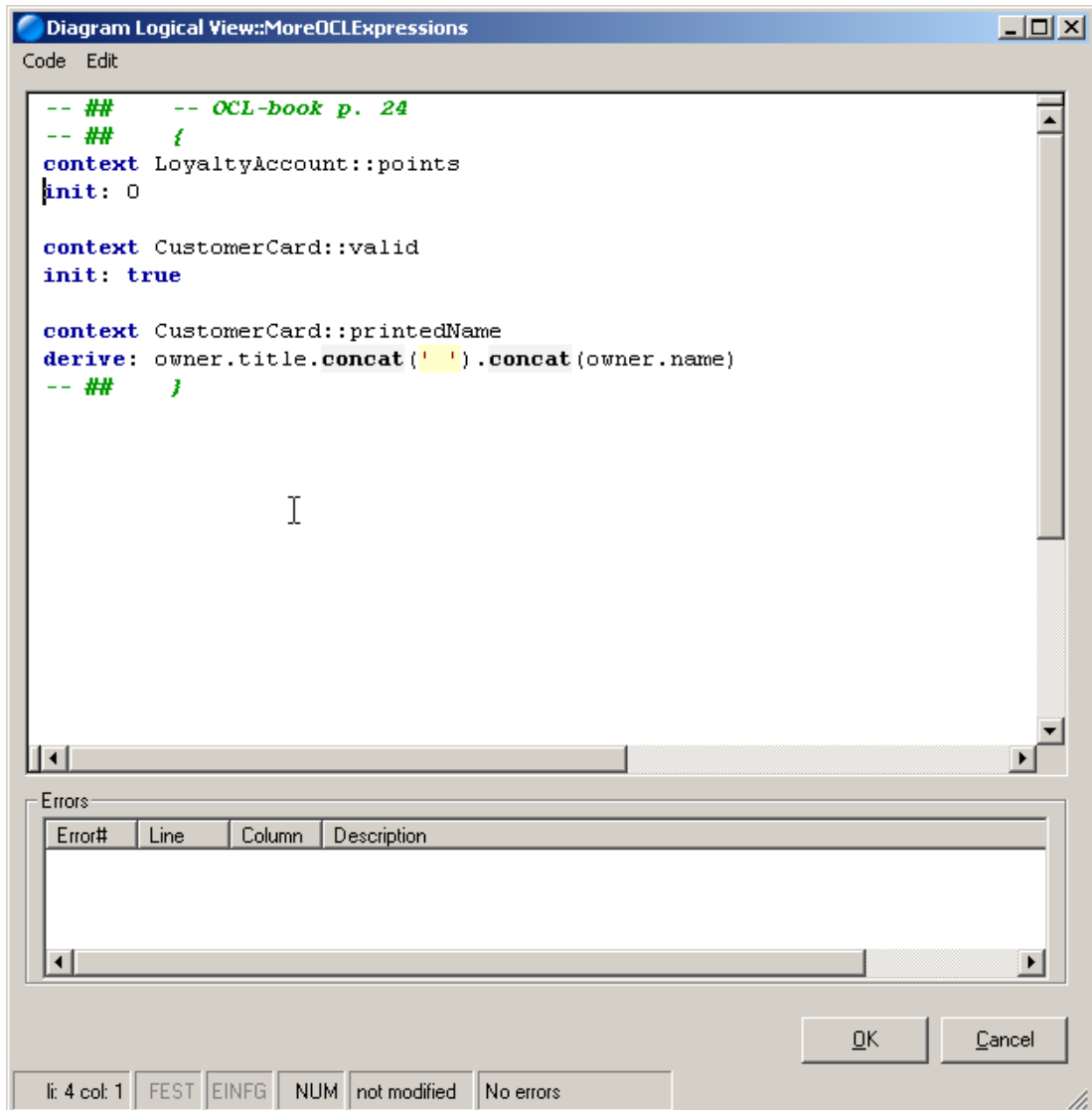


Figure 6: Edit code from note

These marks must be preserved during editing to avoid syntax errors.

6 Checking OCL code

6.1 Introduction

Checking the validity of the OCL code is the most important functionality of Oclarity. After all, only correct OCL code adds value to your UML models. Without checks, the risk of having wrong or outdated code is too high to improve the overall productivity of your development team.

6.2 Triggering checks

Checks can be triggered in the following ways:

- Using the context menu of one or more selected classes on a class diagram or in the browser.
In this case, the code associated with all selected classes is checked.
- Using the context menu on the background area of a class diagram.
In this case, the code associated with all classes contained in the diagram and all OCL code contained in the notes on the diagram is checked. The same effect can be achieved by selecting the menu entry 'Tools/OCL/Check OCL expressions of current element'.
- Using the menu entry 'Tools/OCL/Check all OCL expressions'.
In this case, all OCL expressions contained in the whole model are checked. Depending on the size of your model and the amount of OCL code this may take a considerable amount of time. A dialog containing a progress bar and a cancel button is displayed during the check.

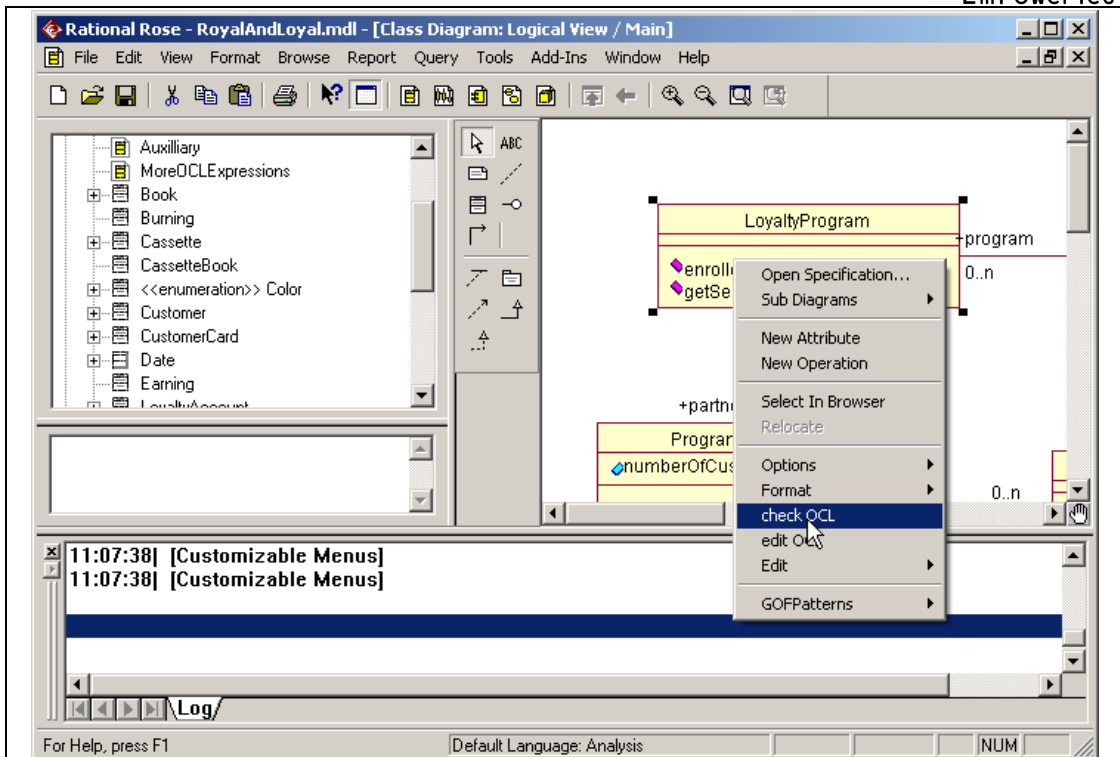


Figure 7: Trigger check on selected class

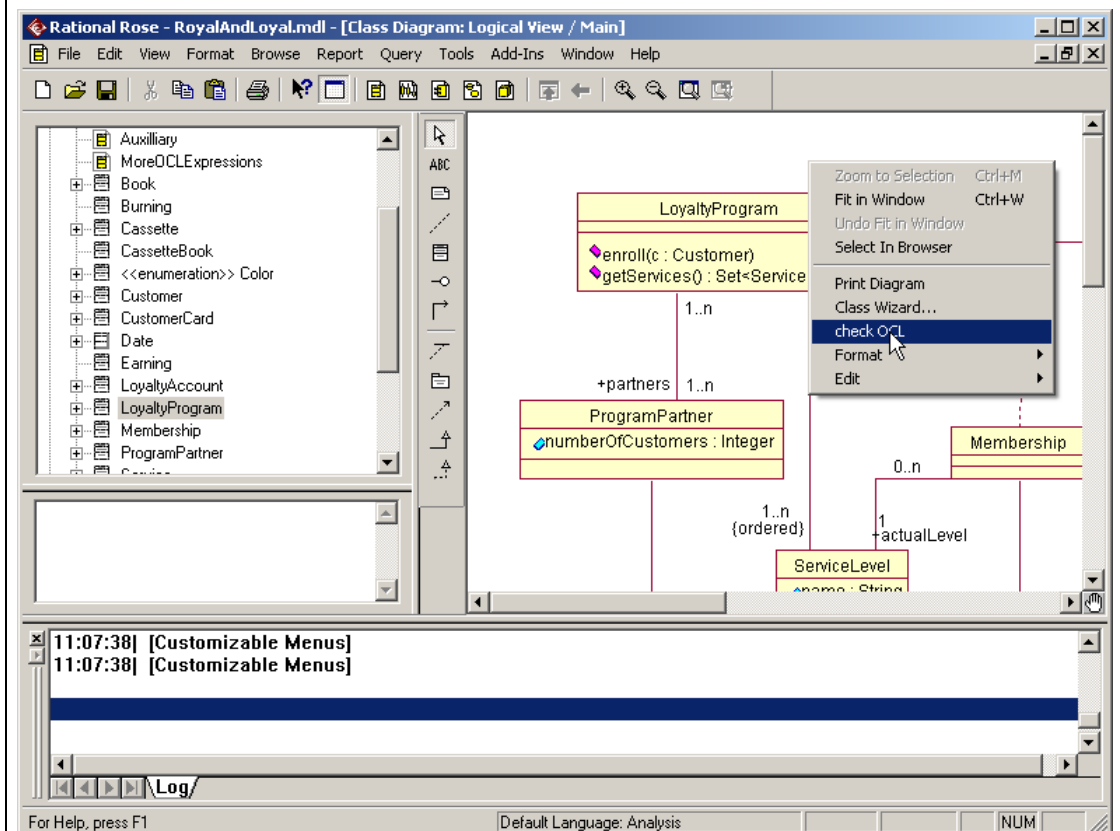


Figure 8: Trigger check for whole class diagram

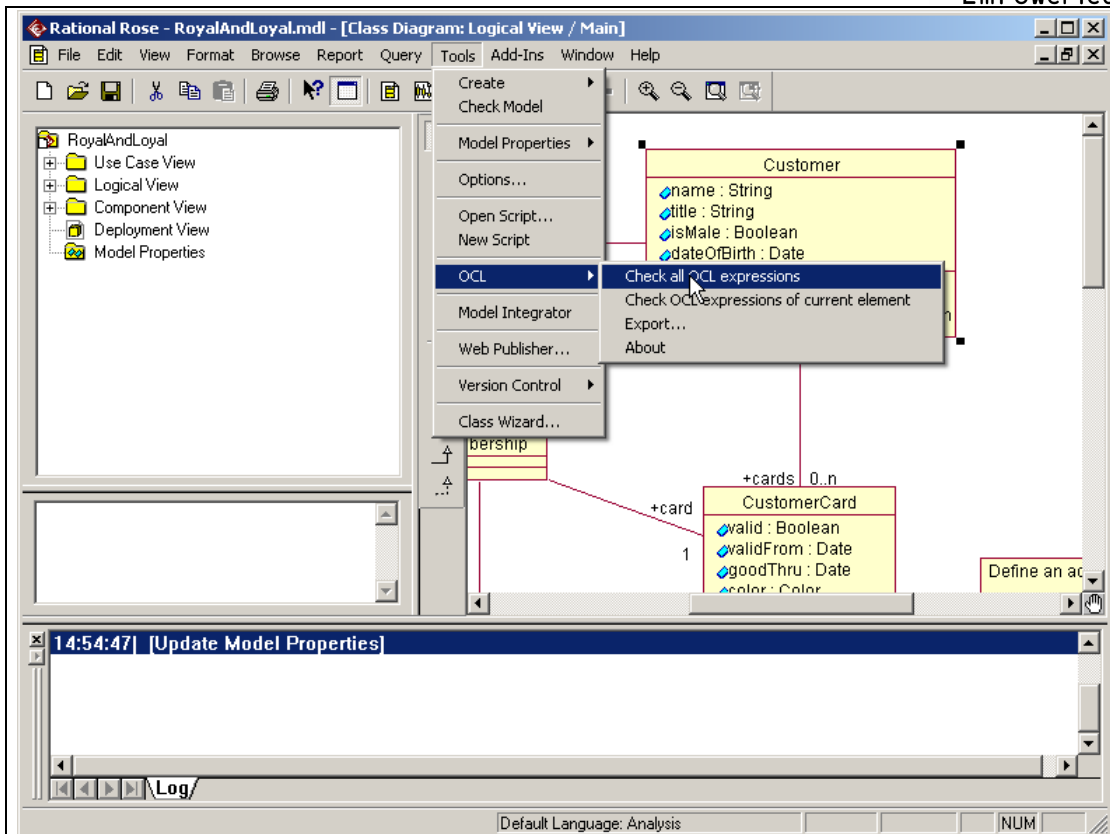


Figure 9: Trigger checks on all OCL expressions in a model

6.3 Definition expressions

OCL expressions of type *def* have a special characteristic: they add attributes or queries to types defined in the model.

```
context Person
def: income : Integer = self.job.salary->sum()
def: hasTitle(t : String) : Boolean = self.job->exists(title = t)
```

In the example above, an attribute named *income* of type *Integer* and a method named *hasTitle(String)* are added to class *Person*. The problem with this kind of expression is that they are not available until the OCL expressions that defines them have been processed. This means, that if another OCL expression references such a definition expression, the definition expression must be checked before. Since the order of evaluation of the various model elements (classes and notes) is basically arbitrary, the check is done in two passes. In pass 1, only definition expression are checked and thus temporarily added to the respective classes. In pass 2, all expressions are checked. The passes are done automatically, no user intervention is required. But even with the 2 pass approach, it is the users responsibility to include the model element, which holds the referenced definition expressions, in the check.

6.4 Viewing check results

After one or more OCL expressions have been checked, the results of the checks are displayed in the result window. The basic purpose of this window is to quickly find any errors contained in the checked expressions. In addition, the window allows the easy editing of OCL expressions and quick rechecks of the expressions.

Here is a screenshot that we will use to explain the various properties of this window:

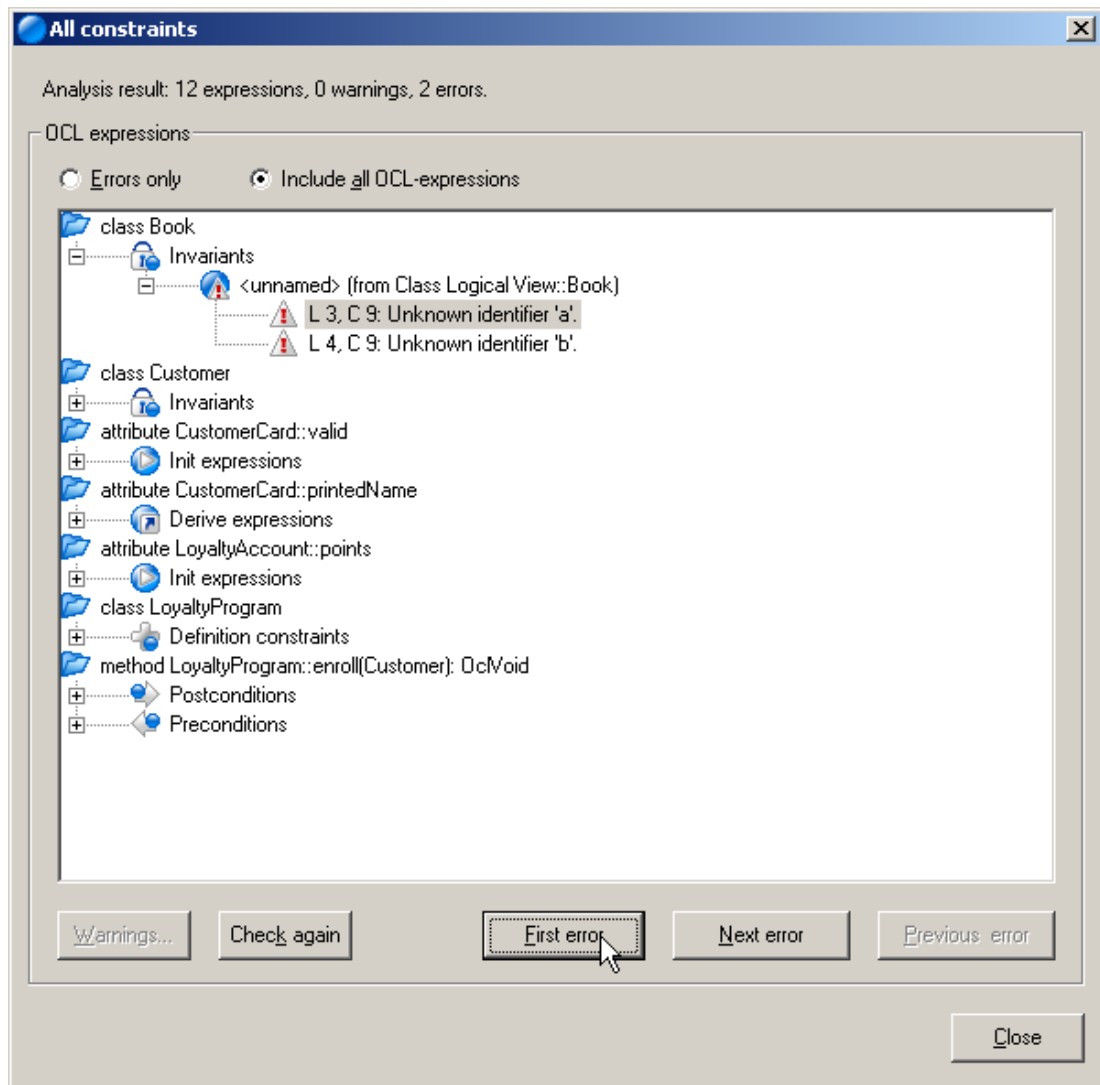


Figure 10: Result window

The window provides a tree like view of the different contexts that contain OCL expressions. For each context, the types of OCL expressions are displayed. Within the expression type, the single expressions are displayed, either with their name if specified or with the canonical name '<unnamed>'. If an expression contains errors, a distinct entry is displayed for each error. If a given context does not contain an expression

of a certain type (e.g. no 'derive'-expressions), this type is not displayed.

At the top of the window, you can choose, whether you want to see all OCL expressions or only expressions that contain errors.

The various buttons have the following meanings:

6.4.1 Warnings...

Usually, errors are displayed within their context. But some kind of errors, e.g. errors in a context specification or errors during accessing the internal repository of Rational Rose, cannot be attributed to a specific context. Such errors are displayed as warnings in a separate window.

This button is only enabled, if actually errors of this kind have occurred during the last check.

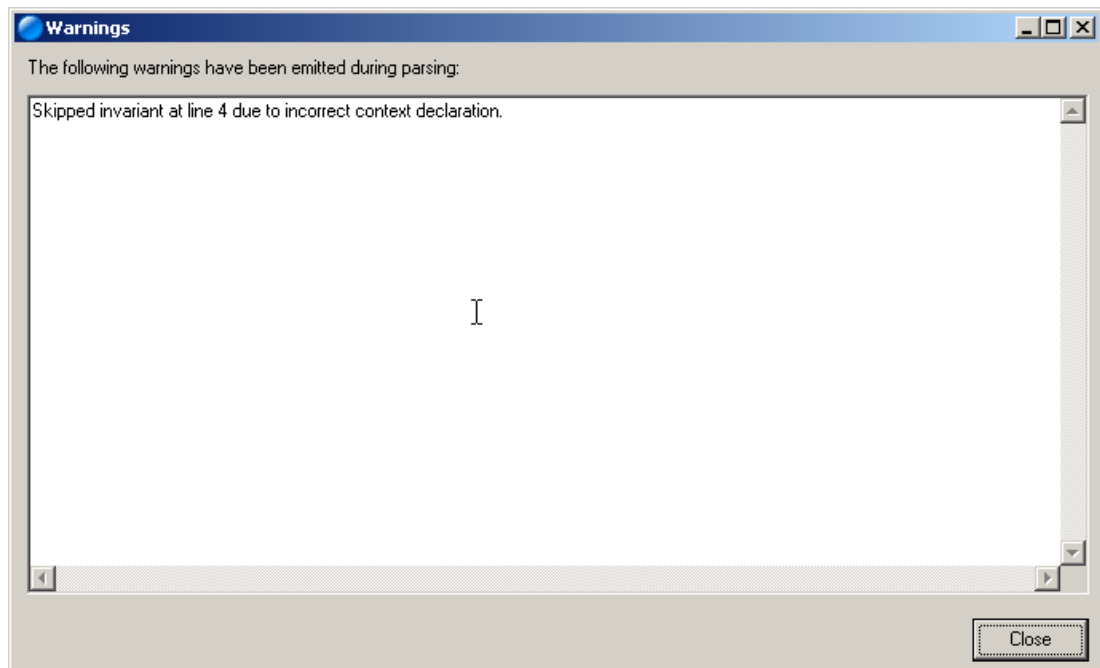


Figure 11: Warnings

6.4.2 Check again

After code has been edited from the result window, this button can be used to trigger a new check of all OCL expressions that have been checked during the initial check.

6.4.3 First error

Pressing this button selects the first error.

If no errors have been found during analysis this button is disabled.

6.4.4 Next error

Pressing this button selects the next error. If the last error is already selected or no errors have been found during analysis, this button is disabled.

6.4.5 Previous error

Pressing this button selects the previous error. If the first error is already selected or no errors have been found during analysis, this button is disabled.

6.4.6 Editing code from the result window

By double clicking or pressing the right mouse button and choosing menu entry 'edit code' on a node, the underlying OCL code can be edited. This allows for fast correction of errors encountered during analysis of the OCL expressions. Not all kind of nodes support this behavior.

Please see chapter 5.6 for special treatment of OCL expressions in notes.

7 Exporting OCL expressions

Exporting the OCL expressions contained in your models may be useful for various reasons:

- You might want to make the OCL expressions easily accessible for your team, e.g. by publishing them on a website.
- You might want to use the expressions in another tool.

Therefore, Oclarity supports the export of all OCL expressions to an XML file.

7.1 Triggering the export

To trigger the export, choose menu entry Tools/OCL/Export... and follow the instructions shown in the displayed dialog. The settings made in this window are preserved across invocations of Rational Rose.

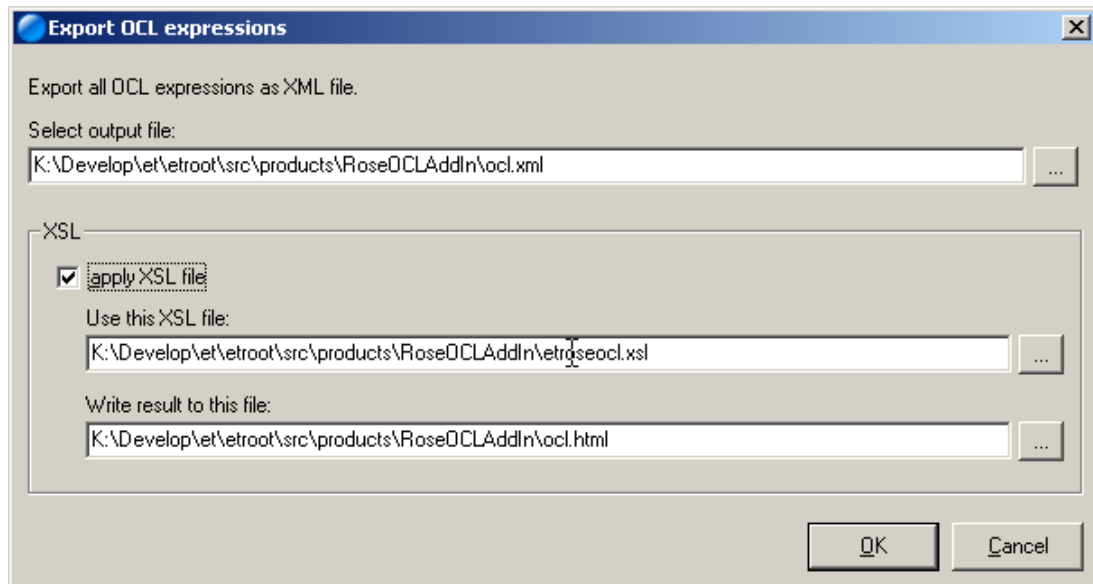


Figure 12: Triggering an export

You can specify the name and path of the destination XML document. Optionally, an XSL-file can automatically be applied to the newly exported XML file.

A sample XSL file that you can use as basis for your own XSL files is shipped with Oclarity. It is stored in the subdirectory OCL in the Rational Rose installation directory.

7.2 Structure of the XML file

In the listing below, the structure of an exported XML file is listed. Bold text indicates an XML element whereas normal text indicates an attribute of the associated element. An increased indentation indicates either a sub element of the element in the line above or an attribute of the element above.

```
ocl_expressions
  general
    model_name
    user_name
    computer_name
    export_time
  contexts
    context
      name
      category
        type
        expression
          name
          source
          code
```

8 Integration with the Rose repository

8.1 Overview

The OCL language definition is free of any reference to concrete programming languages or tools. Therefore, when using OCL in the context of a specific programming language or software engineering tool there is a certain degree of freedom in mapping the features of OCL to the programming language or tool at hand. This chapter describes how various features of OCL are to be used in the context of Oclarity for Rational Rose.

OCL uses the following properties of model elements:

- isQuery-status of operations
- whether an operation is a class operation or an instance operation
- the direction of operation parameters

A method that is a 'query' is an operation that does not alter the systems state during its execution. Only queries can be called in OCL expressions.

The direction of operation parameters is important, because if an output (or input/output) parameter is used, the method returns a TupleType in OCL and not the specified return type.

Since none of these properties can be edited efficiently and uniform across all versions of Rational Rose, Oclarity comes with its own dialog to edit these OCL-related method properties. We store the according information in the model in a way that is compatible with Rational Roses C++ language AddIn.

8.2 The OCL-property editor for operations

To edit the OCL-related properties of an operation or operation parameter, the according operation has to be selected in the browser. Clicking the right mouse button and choosing the menu entry 'OCL properties...' opens the editor for these properties.

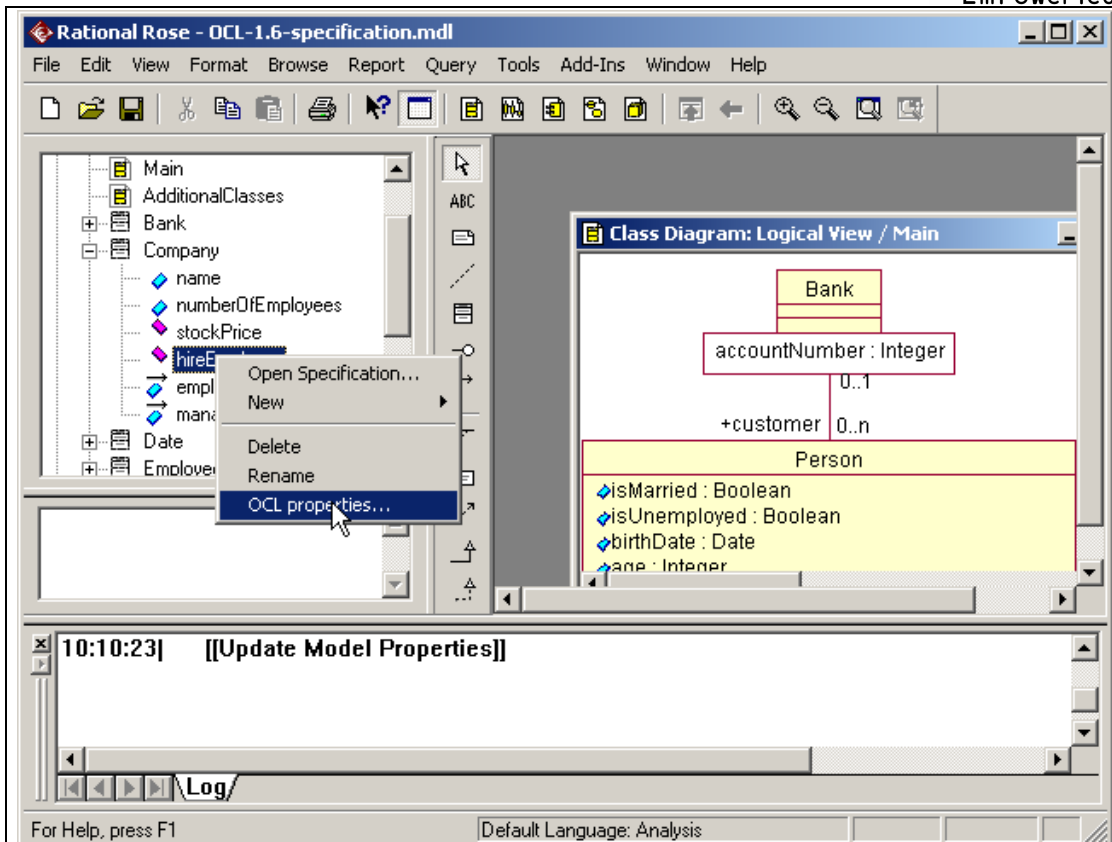


Figure 13: Opening the operation property editor

In the following window, all OCL relevant properties of an operation and its parameters can be modified.

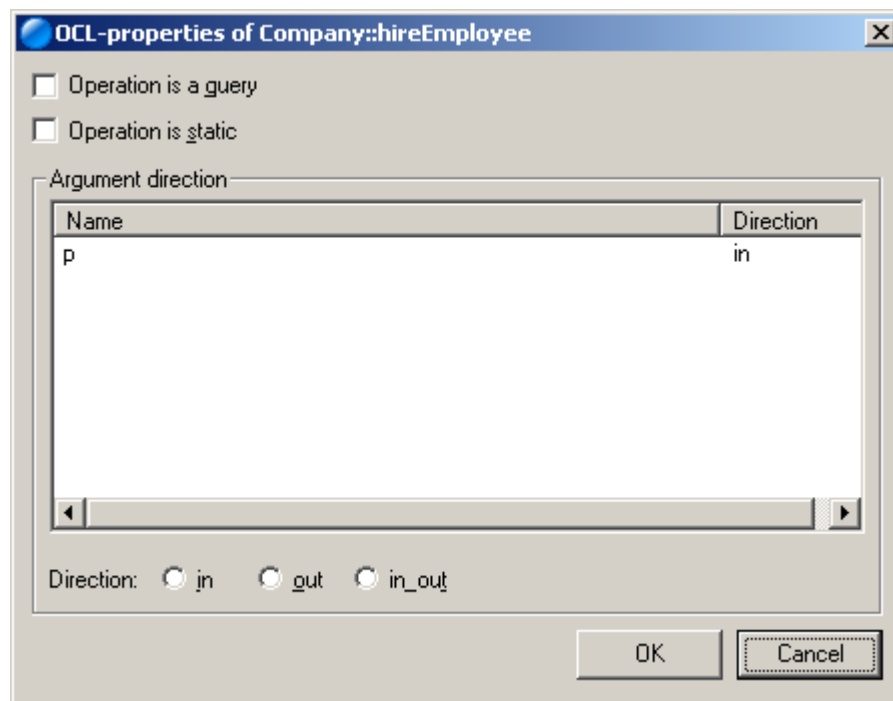


Figure 14: Edit operation properties

8.3 Package handling

8.3.1 Package names with spaces

In Rational Rose, package names may contain spaces but this is not supported by the OCL grammar. As a workaround the spaces have to be replaced by underscore characters ('_') in the OCL code.

For example, if you want to reference a type named 'Customer' in the package 'Logical View::Business Objects', the OCL code must use 'Business_Objects::Customer' (note that 'Logical View' is not specified - see chapter 8.3.2).

It is not possible to mix underscores and spaces in package names. Our recommendation is not to use package names with spaces in your Rational Rose models.

8.3.2 Default package

Per default, Oclarity searches types in the package 'Logical View'. The package 'Logical View' must never be specified explicitly.

However, it is also possible to refer to types in the 'Use Case View'. In this case, the package must be specified and the spaces must be replaced by underscores:

```
Use_Case_View::Customer
```

8.4 Mapping to Rose properties

This chapter describes how Oclarity stores the information that is modified in the operation editor.

Information	Tool/property name	Used values
isQuery	MOF.rose2mof.isQuery	true false
isStatic	MOF.rose2mof.scope	instance_level classifier_level
parameter direction	MOF.rose2mof.direction	in_dir out_dir inout_dir

8.5 Enumerations

As defined by the OCL standard, an enumeration type is defined through a class with the stereotype <<enumeration>>.

The attributes, who must be of type Integer or specified without type, define the members of the enumeration type.

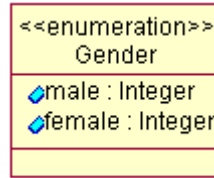


Figure 15: Defining an enumeration type

8.6 Attributes of stereotype <<reference>>

Attributes of stereotype <<reference>> are ignored by Oclarity, because we assume that there is also an association that describes the relation to the other class. This is useful, if Oclarity is used to check expressions in the Rose model of the UML Metamodel published by the OMG.

8.7 States

All states defined in state diagrams can be used in method `oclInState()` without additional provisions.

8.8 Instantiated types

An instantiated type is a type that is not used on its own but only in combination with another type. This concept is called 'templates' in C++ or 'generics' in Java and C#. For example, a container class 'Set' may not be used on its own but only as container for specific contained elements, e.g. instances of class 'Service'.

In OCL, the syntax to express such an instantiation is

```
Type1(Type2)
```

e.g.

```
Set(Service)
```

Since this syntax deviates from the syntax of some widely used programming languages, we decided to allow both notations interchangeably.

This means, if you are modeling a PSM and thus using concrete types from your target language, this syntax can be used:

```
Type1<Type2>
```

e.g.

```
Set<Service>.
```

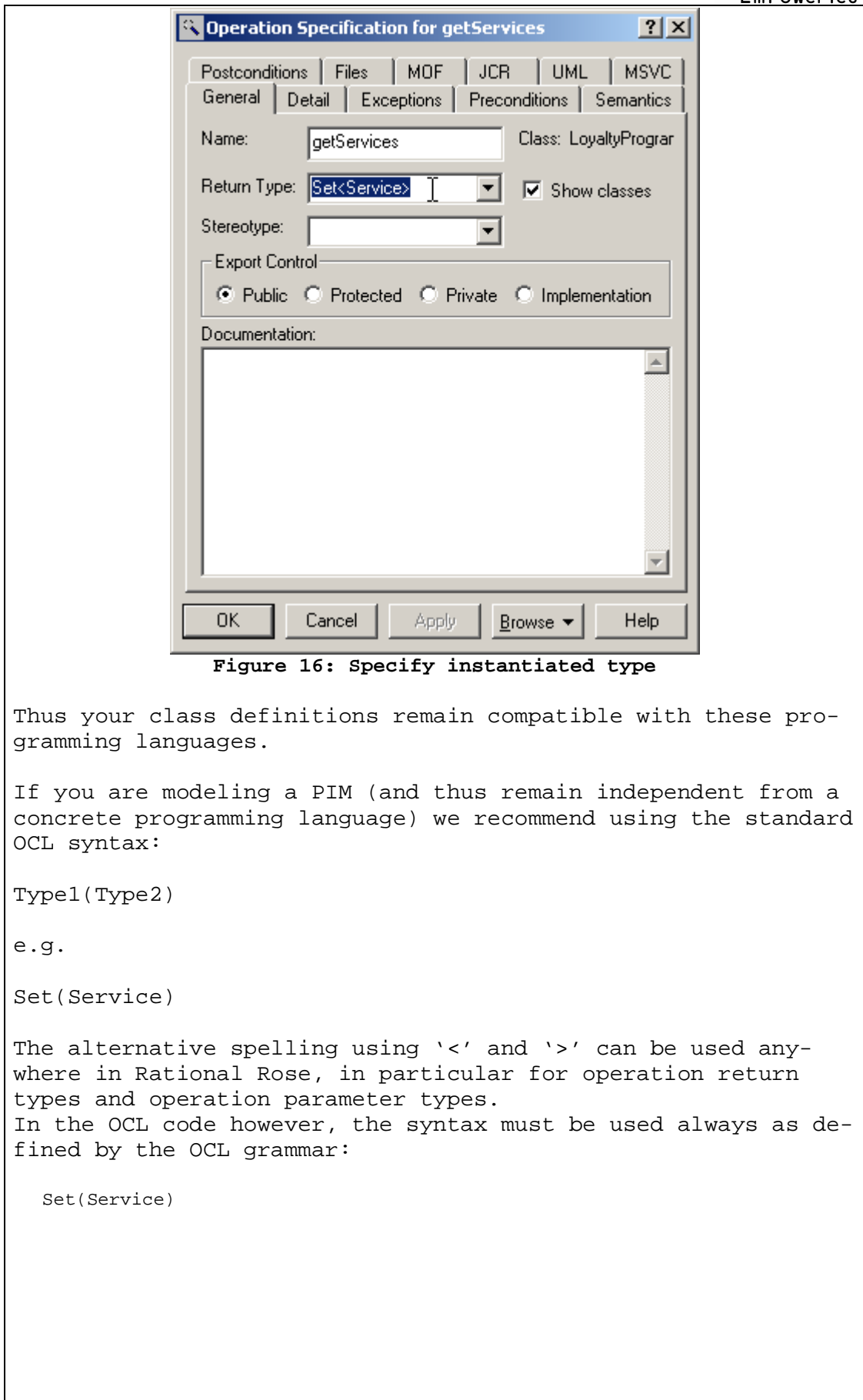


Figure 16: Specify instantiated type

Thus your class definitions remain compatible with these programming languages.

If you are modeling a PIM (and thus remain independent from a concrete programming language) we recommend using the standard OCL syntax:

```
Type1(Type2)
```

e.g.

```
Set(Service)
```

The alternative spelling using '<' and '>' can be used anywhere in Rational Rose, in particular for operation return types and operation parameter types.

In the OCL code however, the syntax must be used always as defined by the OCL grammar:

```
Set(Service)
```

8.9 Associations

In Rational Rose it is possible to create associations between classes and other types of model elements, e.g. an actor from a use case diagram. Such associations are not allowed in UML and are ignored by Oclarity (a warning is emitted).