SDMetrics

User Manual

V2.31

July 3, 2013

For product support and latest product news and updates, visit www.sdmetrics.com

e-mail: info@sdmetrics.com

Jürgen Wüst
In der Lache 17
67308 Zellertal
Germany

All rights reserved. No part of this manual may be reproduced, in any form or by any means, without permission in writing from the author.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Linux is a trademark of Linus Torvalds. UNIX is a registered trademark of X/Open Company Limited. Windows, Excel are registered trademarks of Microsoft, Inc. MOF, UML and XMI are either registered trademarks or trademarks of the Object Management Group, Inc. MagicDraw is a trademark or registered trademark of No Magic, Inc. XML is a trademark of the World-Wide Web Consortium. SDMetrics is a registered trademark of Jürgen Wüst. All other product names and company names mentioned herein are the property of their respective owners.

Table of Contents

1 Introduction....1
2 Installation....2
2.1 System Requirements....2
2.2 Quick Installation and Start....2
2.3 Installing SDMetrics....2
2.3.1 Single User Installation....2
2.3.2 Multiple Users Installation....3
2.4 Updating SDMetrics from an Older Version....3
2.5 Invoking SDMetrics....4
2.6 Uninstalling SDMetrics....4
3 Getting Started....5
4 The SDMetrics User Interface....9
4.1 Getting Help....9
4.2 Specifying Project Settings....9
4.2.1 Specifying Project Files....9
4.2.2 Specifying Filters....12
4.2.3 Saving Project Settings....13
4.2.4 Loading Project Settings....14
4.3 Calculating and Viewing Metric Data....14
4.3.1 Common controls in views....15
4.4 The View 'Metric Data Tables'....16
4.4.1 Highlighting Outliers....17
4.5 The View 'Histograms'....18
4.6 The View 'Kiviat Diagrams'....19
4.7 The View 'Rule Checker'....20
4.7.1 Filtering Design Rules....22
4.7.2 Accepting Design Rule Violations....23
4.8 The View 'Descriptive Statistics'....24
4.9 The View 'Design Comparison'....25
4.9.1 Calculating and Viewing Metric Deltas....26
4.9.2 Metric Deltas Table....26
4.9.3 Comparative Descriptive Statistics Table....27
4.9.4 Mapping Design Elements....28
4.9.5 Exporting Metric Deltas....29
4.10 The View 'Relation Matrices'....29
4.11 The View 'Graph Structures'....30
4.11.1 Viewing Cycles....31
4.11.2 Viewing Connected Components....32
4.12 The View 'Model'....33
4.13 The View 'Catalog'....34
4.14 The View 'Log'....35
4.15 Exporting Data....36
4.15.1 Exporting Data Tables....36
4.15.2 Exporting Graphs....38

© 2002-2013. All rights reserved.

4.16 Setting Preferences....39
4.16.1 Project File Sets....39
4.16.2 Percentiles....43
4.16.3 Output....44
4.16.4 Appearance....45
4.16.5 Behavior....46
5 Running SDMetrics from the Command Line....48
6 Design Measurement....52
6.1 Design Metrics and System Quality....52
6.2 Structural Design Properties....53
6.2.1 Size....53
6.2.2 Coupling....54
6.2.3 Inheritance....55
6.2.4 Complexity....56
6.2.5 Cohesion....56
6.3 Data Analysis Techniques....57
6.3.1 Descriptive Statistics....58
6.3.2 Dimensional Analysis....58
6.3.3 Rankings....58
6.3.4 Quality Benchmarks....59
6.3.5 Prediction Models....60
7 SDMetrics Metamodel and XMI Transformation Files....63
7.1 SDMetrics Metamodel....64
7.2 XMI Transformation Files....66
7.2.1 XMI Transformation File Format....66
7.2.2 XMI Transformations and Triggers....67
7.2.3 Tips on Writing XMI Transformations....73
8 Defining Custom Design Metrics and Rules....77
8.1 Definition of Metrics....78
8.1.1 Projection....78
8.1.2 Compound Metrics....88
8.1.3 Attribute Value....89
8.1.4 Nesting....90
8.1.5 Signature....91
8.1.6 Connected Components....91
8.1.7 Value Filter....92
8.1.8 Subelements....93
8.1.9 Substring....93
8.2 Definition of Sets....95
8.2.1 Projection....96
8.2.2 Subelements....98
8.3 Definition of Design Rules....99
8.3.1 Violation....100
8.3.2 Cycle....101
8.3.3 Projection for Rules....101
8.3.4 Valueset for Rules....102
8.3.5 Word lists....103
8.3.6 Exempting Approved Rule Violations....104

© 2002-2013. All rights reserved.

8.4 Definition of Relation Matrices......104
8.5 Expression Terms......107
8.5.1 Constants and Identifiers......107
8.5.2 Metric Expressions......108
8.5.3 Set Expressions......111
8.5.4 Condition Expressions......111
8.5.5 Expression Terms and XML......113
8.6 Writing Descriptions......113
8.7 Defining Metrics for Profiles......116
8.7.1 Profiles in UML 2......116
8.7.2 Profiles in SDMetrics......116
8.7.3 XMI Serialization of Profile Extensions......117
8.7.4 Profile Extensions with Regular Model Elements......117
8.7.5 Extension References without Inheritance......118
8.7.6 Extension References with Inheritance......121
8.7.7 Tips on Creating Metrics and Rules for Profile Extensions......122
9 Extending the Metrics and Rule Engine......125
9.1 Metric Procedures......125
9.1.1 Conception of a New Metric Procedure......125
9.1.2 Implementation of the Metric Procedure......126
9.1.3 Using the New Metric Procedure......128
9.2 Set Procedures......129
9.3 Rule Procedures......130
9.4 Boolean Functions......132
9.4.1 Conception of a New Boolean Function......132
9.4.2 Implementation of the Boolean Function......133
9.4.3 Using the New Boolean Function......134
9.5 Scalar Functions......135
9.6 Set Functions......136
9.7 Metrics Engine Extension Guidelines......138
A: Metamodels......141
A.1 Metamodel for UML 1.3/1.4......141
A.2 Metamodel for UML 2.x......144
B: List of Design Metrics......152
B.1 Class Metrics......152
B.2 Interface Metrics......158
B.3 Package Metrics......159
B.4 Interaction Metrics......164
B.5 Usecase Metrics......165
B.6 Statemachine Metrics......166
B.7 Activity Metrics......167
B.8 Component Metrics......169
B.9 Node Metrics......171
B.10 Diagram Metrics......172
C: List of Design Rules......174
C.1 Class Rules......174
C.2 Interface Rules......178
C.3 Datatype Rules......179

© 2002-2013. All rights reserved.

C.4 Property Rules....180
C.5 Operation Rules....181
C.6 Parameter Rules....183
C.7 Package Rules....184
C.8 Association Rules....186
C.9 Associationclass Rules....187
C.10 Generalization Rules....187
C.11 Interfacerealization Rules....188
C.12 Dependency Rules....189
C.13 Interaction Rules....189
C.14 Actor Rules....189
C.15 Usecase Rules....190
C.16 Statemachine Rules....192
C.17 Region Rules....193
C.18 State Rules....195
C.19 Activitygroup Rules....200
C.20 Action Rules....200
C.21 Controlnode Rules....201
C.22 Objectnode Rules....203
C.23 Pin Rules....204
C.24 Controlflow Rules....205
C.25 Objectflow Rules....205
D: List of Matrices....206
E: Project File Format Definitions....207
F: Glossary....213
G: References....215

© 2002-2013. All rights reserved.

1 Introduction

Welcome to SDMetrics, the quality measurement tool for UML™ designs. This manual will get you up and running with SDMetrics, and introduce you to the basic and advanced features SDMetrics has to offer.

Part I of this manual addresses all users of SDMetrics. Part II addresses power users who want to define metrics or design rules of their own, and/or adapt SDMetrics to a specific XMI® exporter.

Part I - Basic SDMetrics Usage	
Section 2 "Installation"	describes the installation of SDMetrics.
Section 3 "Getting Started"	is a brief guided tour of SDMetrics, taking you through the steps to calculate a set of metrics for your UML designs.
Section 4 "The SDMetrics User Interface"	describes all features of the SDMetrics user interface in detail.
Section 5 "Running SDMetrics from the Command Line"	shows how SDMetrics can be run from a command line.
Section 6 "Design Measurement"	discusses general design measurement principles and provides guidelines how to interpret measurement data.
Part II - Advanced SDMetrics Features	
Section 7 "SDMetrics Metamodel and XMI Transformation Files"	explains how to define the UML metamodels used by SDMetrics, and how SDMetrics extracts UML design information from XMI files.
Section 8 "Defining Custom Design Metrics and Rules"	shows how metrics and design rules are defined in SDMetrics, and how you can define your own UML design metrics and rules.
Section 9 "Extending the Metrics and Rule Engine"	describes how to extend the calculation capabilities of the metrics engine itself.
Part III - Appendices	
Appendix A: "Metamodels"	shows SDMetrics' metamodels for UML1.x and UML2.x.
Appendix B: "List of Design Metrics"	lists the design metrics that ship with SDMetrics.
Appendix C: "List of Design Rules"	lists the design rules that ship with SDMetrics.
Appendix D: "List of Matrices"	lists the relation matrices that ship with SDMetrics.
Appendix E: "Project File Format Definitions"	is a reference to the metamodel, metric definition, and XMI transformation file formats.

© 2002-2013. All rights reserved.

2 Installation

2.1 System Requirements

SDMetrics runs on platforms supporting the Java™ 1.6 or better runtime environment (JRE). Memory usage depends on the size of the models analyzed and typically ranges from 40MB for small systems (a few hundred model elements) to 500MB-1GB for large systems (several hundred thousand model elements). The GUI requires at least a 17" color display. Disk space required for the installation is less than 10MB (not counting the JRE). Typically, the hardware that runs your UML modeling tools will also be sufficient to run SDMetrics.

If you do not already have the JRE installed on your machine, obtain the latest version of the JRE for your platform at http://www.java.com/. Follow the download and installation instructions provided there.

2.2 Quick Installation and Start

- Unpack the compressed archive (zip file) you received or downloaded to a folder of your choice.
- Make sure you have read and agreed with the license agreement `License.txt` before using SDMetrics.
- To start SDMetrics, double-click the file `SDMetrics.jar`, or invoke the script `SDMetrics.bat` (Windows platforms) or `SDMetrics.sh` (Unix/Linux platforms).

For installation on other platforms, and further installation options, see the following sections.

2.3 Installing SDMetrics

2.3.1 Single User Installation

Installing SDMetrics is easy. The installation will not affect your system configuration in any way. You do *not* require administrative rights on your system for the installation.

To install SDMetrics, simply unpack the compressed archive (zip file) you received or downloaded to a folder of your choice. Among others, you should find these artifacts in the folder:

© 2002-2013. All rights reserved.

File/folder name	Purpose
`SDMetrics.jar`	The SDMetrics program files.
`sdmetrics.bat`	Script to start SDMetrics on Windows platforms.
`sdmetrics.sh`	Script to start SDMetrics on Unix/Linux platforms.
`Readme.html`	Latest program version information.
`License.txt`	The SDMetrics end user license agreement.
`SDMetricsUserManual.html`	Opens the SDMetrics User Manual in your web browser.
`manual/`	Contains the SDMetrics User Manual files.
`bin/`	Any extensions your create for SDMetrics go in here.

Make sure you have read and agreed with the license agreement before using SDMetrics.

2.3.2 Multiple Users Installation

If you have licensed SDMetrics for a larger number of users, the system administrator may choose to deploy SDMetrics on a network or multi-user system (network drive, terminal servers, etc).

- A single installation copy can be used by more than one user at a time.
- The installation directory can (and should) be read-only for users, as SDMetrics does not write to its installation directory at runtime.

Alternatively, you can also perform separate installations at each local client, or use a mixed approach of local and central installations.

2.4 Updating SDMetrics from an Older Version

If you have previously installed an older version of SDMetrics that you no longer wish to use, simply replace the old installation directory with the new version.

If you choose to use the old and new version of SDMetrics in parallel, put the new version in a new installation directory of its own.

In either case, your application preferences will automatically be taken over from the old version, with two exceptions:

- The default project file sets (see Section 4.16.1 "Project File Sets") will always be restored to factory settings for the new version. If you have changed the metamodel/metric definition/XMI transformation files for the default project file sets, you need to register them again with the new version of SDMetrics.
- SDMetrics will look for the user manual in its default location (see Section 4.16.5 "Behavior").

© 2002-2013. All rights reserved.

2.5 Invoking SDMetrics

Navigate to the SDMetrics installation directory and double-click file `SDMetrics.jar` to launch SDMetrics. If this does not work on your system, invoke SDMetrics with the script `SDMetrics.bat` (for Microsoft Windows platforms) or `SDMetrics.sh` (for Unix/Linux platforms).

On a command line prompt, start SDMetrics with the command:

```
java -jar "/path/to/SDMetrics.jar"
```

The name of the Java class to launch SDMetrics is `com.sdmetrics.SDMetrics`, so the following command will also start SDMetrics:

```
java -classpath "/path/to/SDMetrics.jar" com.sdmetrics.SDMetrics
```

2.6 Uninstalling SDMetrics

1. SDMetrics stores the user's preferences in a file `.SDMetricsXYZ.props` in the user's home directory. XYZ indicates the program version. You can find the precise location of the user preferences file in the "About" dialog (select "Help -> About"). Quit SDMetrics and delete this file.
2. Delete the SDMetrics installation directory you created during the installation, and all of its contents.

This completely removes SDMetrics from your system.

© 2002-2013. All rights reserved.

3 Getting Started

The following guided tour takes you through the steps to calculate a set of metrics for your UML design. Let's assume you have exported a UML design you want to analyze as an XMI file from your UML design tool. Refer to the manual of your UML modeling tool how to accomplish this. Start SDMetrics as described in Section 2 "Installation", and you are presented with the SDMetrics main window.

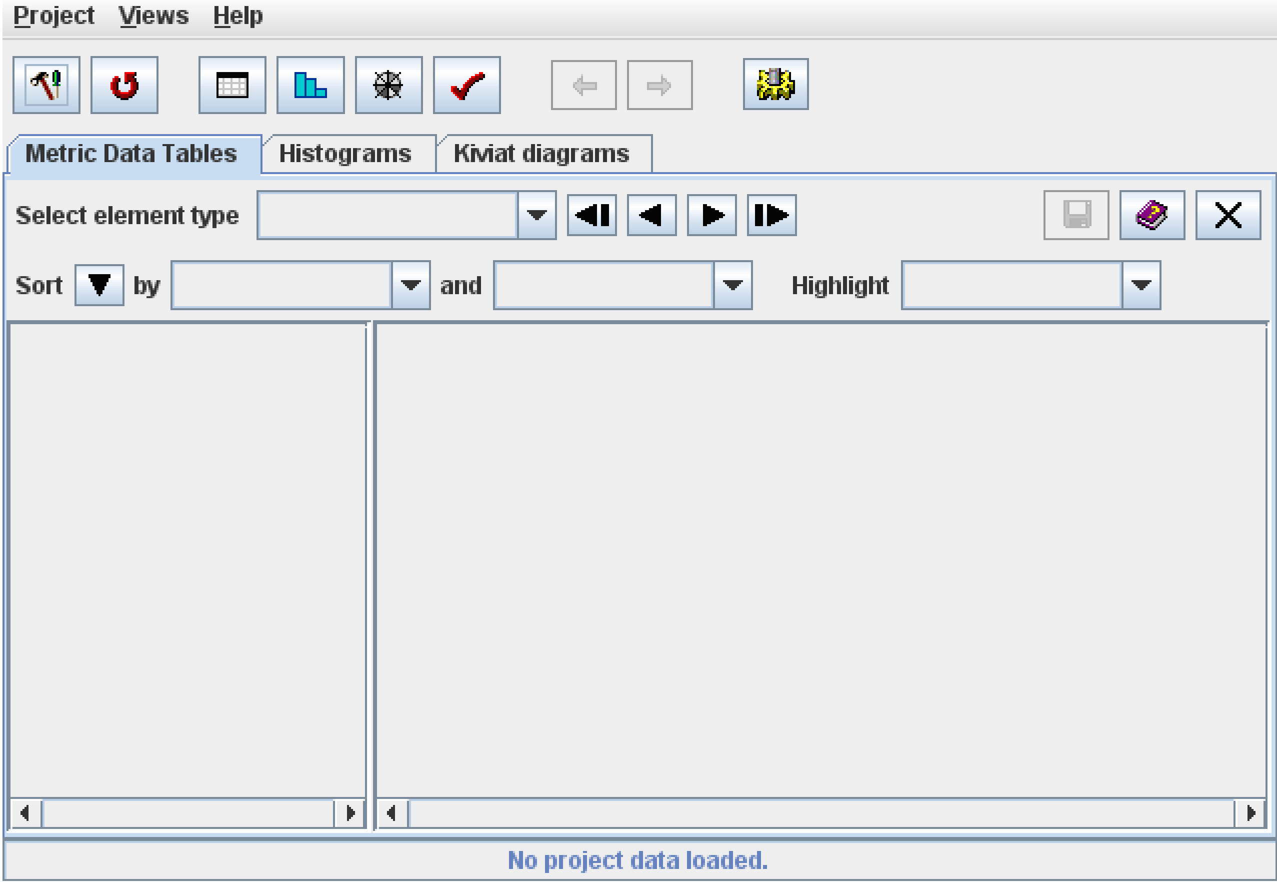


Figure 1: SDMetrics main window

Calculating a set of metrics takes four steps:

1. Specify the XMI file with your UML design
2. Calculate the metrics
3. Explore the metric data
4. Export the metric data for further processing

© 2002-2013. All rights reserved.

Step 1: Specify the XMI file to read

Click the button or select "Project -> Edit Project Settings" from the menu bar.

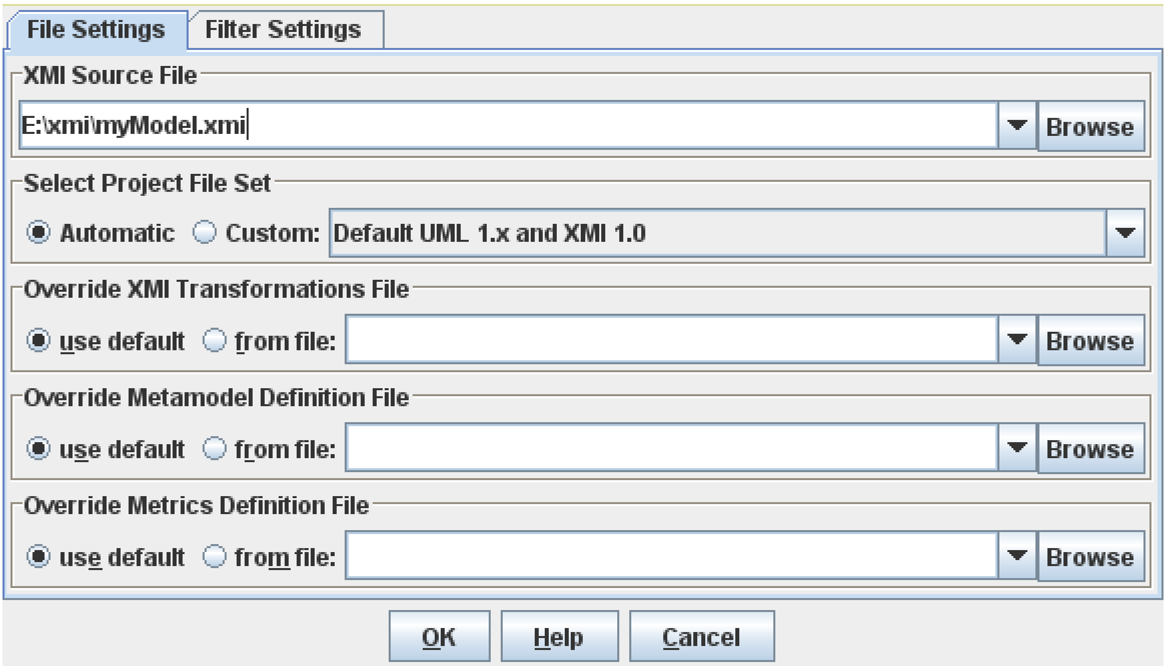


Figure 2: Project File Settings Dialog

You specify the XMI file to read in the "XMI Source File" box at the top of the project settings dialog. Click the topmost "Browse..." button and use the file chooser dialog to select your XMI file.

At this point, you don't need to worry about the remaining options in the Project File Settings dialog; leave everything at "Automatic" and "use default", and click the "OK" button to confirm your selection.

Step 2: Calculate the metrics

After you specified your project files, select "Project->Calculate Metrics" from the menu bar, or click the button on the tool bar. SDMetrics will read your XMI design file and calculate the metrics. This is a fully automated process that usually takes a few seconds. You can monitor its progress on the status bar.

© 2002-2013. All rights reserved.

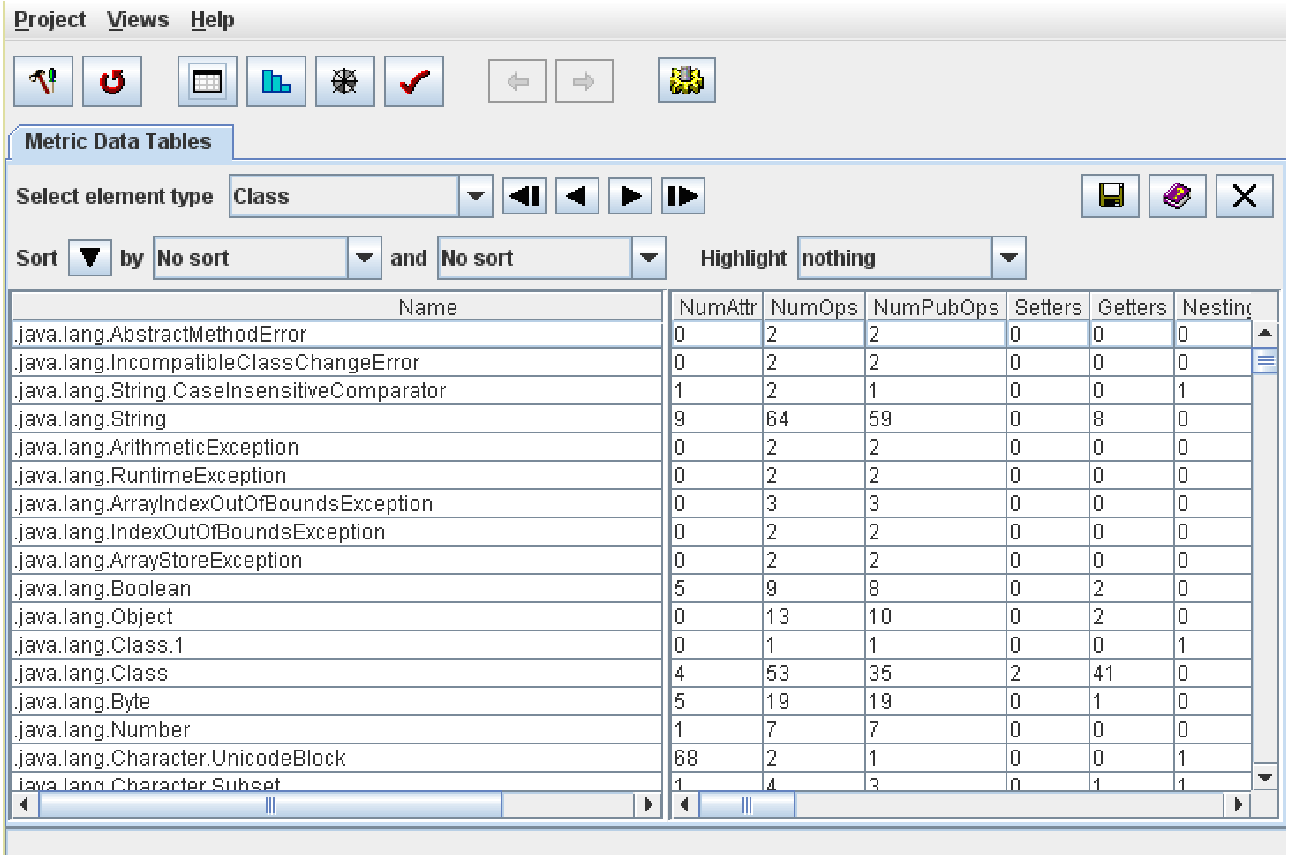


Figure 3: Table view of calculated metrics

Step 3: Explore the metric data

To explore the metric data, SDMetrics provides several views of the data, for instance:

- Metric data tables
  Presents a tabular view of all metric data. Select the type of elements (packages, classes, etc) you want to see from the dropdown list in the upper tool bar of the view. You can sort the table according to metric values, and highlight outliers (model elements with relatively high metric values).
- Histograms
  In this view you can browse histograms and cumulative distribution charts for the metrics. The view also displays descriptive statistics such as mean/minimum/maximum values and percentiles for one metric at a time.
- Kiviat Diagrams
  This view shows Kiviat diagrams for the model elements. The diagrams provide a summary of all metric values for one design element at a time.

Open these views and more from the "Views" menu in the menu bar.

© 2002-2013. All rights reserved.

Step 4: Export the metrics data for further processing

You can export the metric data tables for further processing with your favorite spreadsheet software or statistical software packages. Go back to the "Metric Data Tables" view and click the button on the upper right corner of the view. This opens the data export dialog window.

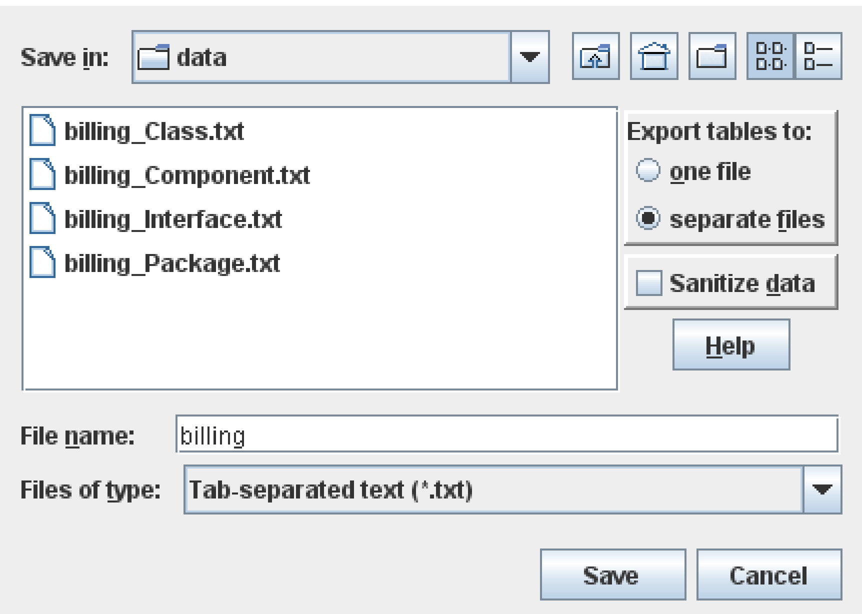


Figure 4: Data Export Dialog

Here you can specify whether to write the data tables to one or several output files, and select the file format for the output files (tab-separated text, HTML, etc).

This completes the brief guided tour of some of SDMetrics' most important features. The following section describes these and all other features of SDMetrics in detail.

© 2002-2013. All rights reserved.

4 The SDMetrics User Interface

This section describes how to use SDMetrics via its interactive graphical user interface. For batch processing see Section 5 "Running SDMetrics from the Command Line".

4.1 Getting Help

All views and dialogs of SDMetrics contain help buttons ( ) that will show the relevant part of the user manual in a web browser. Or select "Help -> User Manual Table of Contents" from the menu bar, or press the F1 key at any time. If you have problems opening the manual from within SDMetrics, see Section 4.16.5 "Behavior".

4.2 Specifying Project Settings

4.2.1 Specifying Project Files

4.2.1.1 Overview of Project Files

To control the way SDMetrics processes your UML designs, SDMetrics requires a set of project files, such as the file containing the definition of the metrics to be calculated. Figure 5 illustrates the role of the project files:

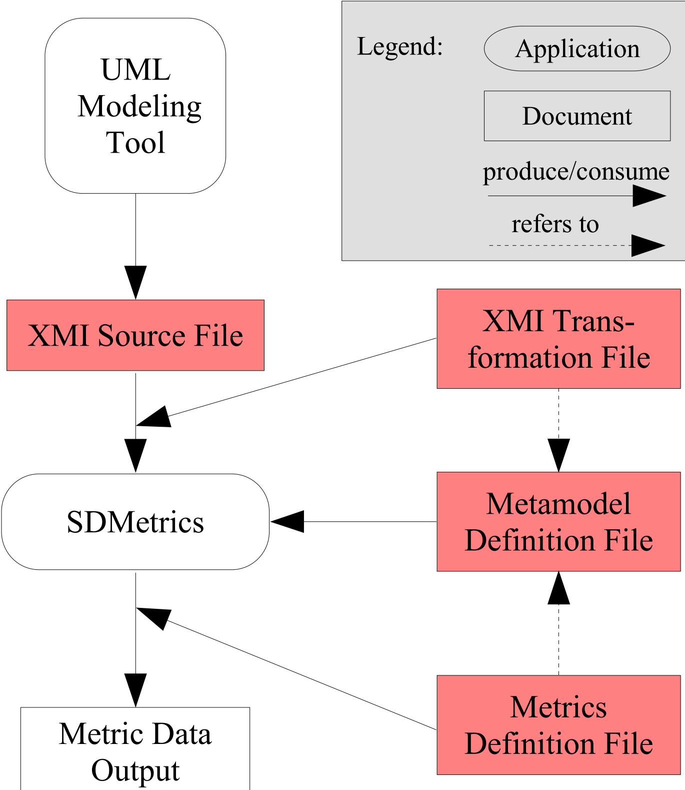


Figure 5: SDMetrics Project Files

© 2002-2013. All rights reserved.

- XMI Source File
  The XMI file that stores the UML design you want to analyze. You create this XMI file with your UML modeling tool or some other application that exports XMI files.
- Metamodel Definition File
  The SDMetrics metamodel (see Section 7.1 "SDMetrics Metamodel") defines which UML elements SDMetrics knows about. The metamodel provides the basis for the definition of the metrics to be calculated.
- XMI Transformations File
  The XMI transformations file (see Section 7.2 "XMI Transformation Files") defines how the model elements in the SDMetrics metamodel are retrieved from the XMI source file.
- Metrics Definition File
  The metrics definition file (see Section 8 "Defining Custom Design Metrics and Rules") defines the set of metrics to be calculated for your UML design.

4.2.1.2 The Project File Settings Dialog

To edit the project file settings, select "Project -> Edit Project Settings" from the menu bar, or click the button in the tool bar. Select the "File Settings" tab.

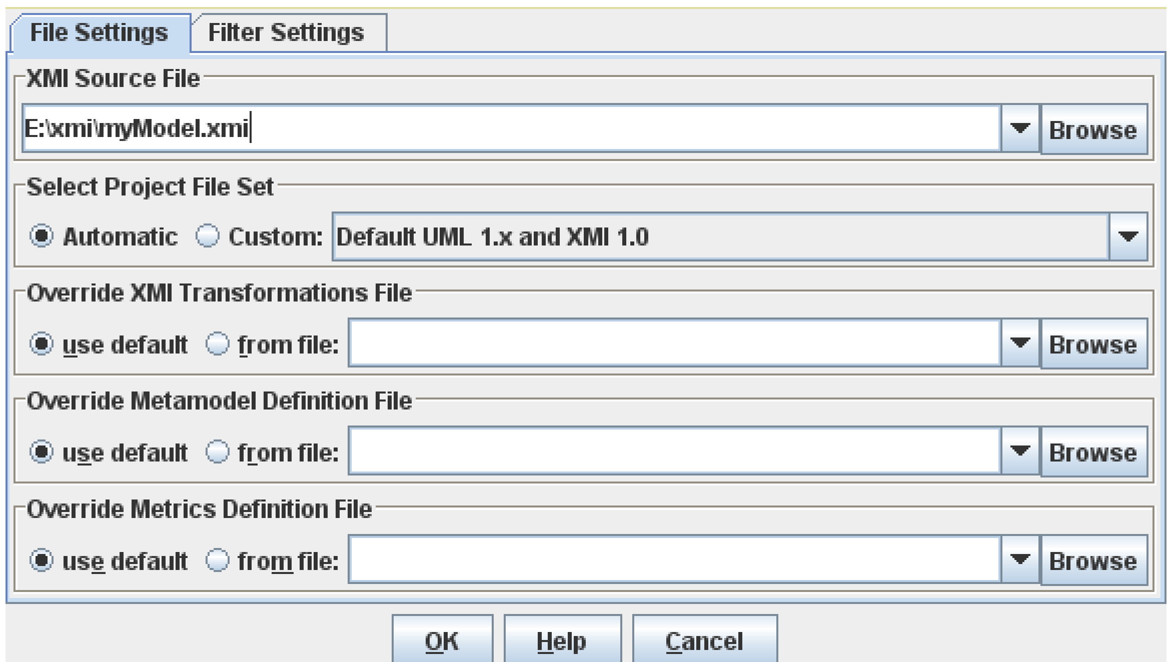


Figure 6: Project File Settings Dialog

On the file settings dialog, you specify the project files to use. From top to bottom:

XMI Source File
To specify the XMI source file,

- enter the name of the XMI file with your UML design in the text field,
- or click the arrow down button next to the text field to pick a file from a list of recently used XMI source files,

© 2002-2013. All rights reserved.

- or click the "Browse..." button next to the text field; this opens a standard file chooser dialog where you can navigate the file system to select the file,
- or if your system supports file drag and drop, you can drop the XMI file anywhere into the main window or the project file settings dialog.

Select Project File Set
SDMetrics ships with several default project file sets. Each project file set includes a consistent combination of one metamodel definition file, one XMI transformation file, and one metric definition file:

Name	Description
Default UML 1.x and XMI 1.0	XMI 1.0 import and metrics for UML 1.x models
Default UML 1.x and XMI 1.1-1.3	XMI 1.1/1.2/1.3 import and metrics for UML 1.x models
Default UML 2.x and XMI 2.x	XMI 2.x import and metrics for UML 2.x models

Table 1: Default project file sets

When you select the radio button "Automatic", SDMetrics will automatically determine the most suitable project file set based on the XMI version and exporter of the XMI source file at hand. When you select the radio button "Custom", SDMetrics will always use the project file set selected from the dropdown list. This feature is useful when you have created your own project file sets (see Section 4.16.1 "Project File Sets"), or if SDMetrics has problems determining the XMI version of your XMI source file, or if SDMetrics cannot match the XMI version to one of the available project file sets.

Most of the time, you will want to use a project file set "as is". You can, however, replace any or all files of the project file set with modified or custom project files of your own, e.g., to calculate a different set of design metrics.

Override XMI Transformations File
If you do not wish to use the XMI transformation file of the applicable project file set, you can specify an alternative XMI transformation file to use here. Select the "from file" radio button, and specify your transformation file in the text field next to it (using the browse button, the list of previously used XMI transformations, or drag and drop the file into the dialog).
See Section 7.2 "XMI Transformation Files" for more information on how to create your own XMI transformation files.

Override Metamodel Definition File
To use a modified or custom metamodel of your own, select the "from file" radio button, and specify your metamodel definition file in the text field next to it (using any of the means described above).
See Section 7.1 "SDMetrics Metamodel" for more information on how to create your own metamodel definitions.

Metrics Definition File
To use a metrics definition file of your own, select the "from file" radio button, and specify your metrics definition file in the text field next to it (using any of the means described above).

© 2002-2013. All rights reserved.

See Section 8 "Defining Custom Design Metrics and Rules" for more information on how to create your own metrics definition files.

4.2.2 Specifying Filters

Your XMI files may contain design elements representing APIs of standard libraries or 3rd party components. In most cases, you do not want to calculate design metrics for such elements. With filters, you can instruct SDMetrics to ignore such design elements.

4.2.2.1 Qualified Element Names

SDMetrics' filtering mechanism is based on the fully qualified names of design elements. For example, if your design model is called "BankingApplication", the fully qualified name of a class "Account" in a top-level package "businesslogic" is:
`BankingApplication.businesslogic.Account`
If class "Account" has a method "deposit" with a parameter "amount", the fully qualified name of parameter "amount" is:
`BankingApplication.businesslogic.Account.deposit.amount`

The fully qualified name of a model element is the name of element, prepended by the fully qualified name of element's owner and a period. Note that SDMetrics element filters use the period as namespace separator, not the double colon (::) of the UML.

4.2.2.2 Specifying Filters

Filters are specified based on fully qualified element names. You define filters to match one or more parts of the beginning of fully qualified names. For example, a filter `BankingApplication.businesslogic` matches the above mentioned package "businesslogic" and all elements it contains.

The # character serves as wildcard that matches any name. For instance, the filter `#.org.xml` matches the package org.xml and all elements contained within, regardless of the design model name.

4.2.2.3 Filter Dialog

To open the filter dialog, select "Project -> Edit Project Settings" from the main menu, or click the button on the tool bar. Select the "Filter Settings" tab.

© 2002-2013. All rights reserved.

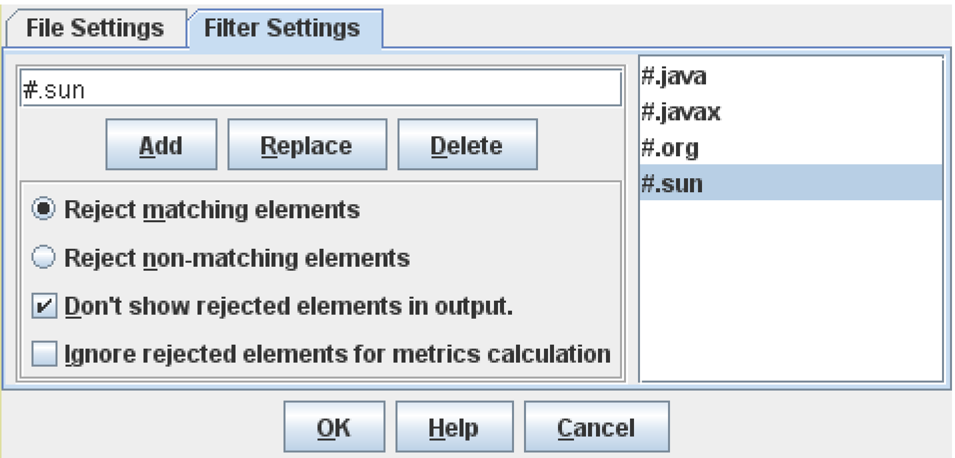


Figure 7: Filter Settings Dialog

The right hand side of the window shows the current list of filters. To add a new filter to the list, enter the filter string in the text field on the left side and click the "Add" button (or press the enter or return key). Click the "Replace" button to replace the currently selected filter on the list with the contents of the text field. The "Delete" button removes the currently selected filter on the list.

The meaning of the radio buttons and checkboxes is as follows:

- "Reject matching elements" - if this option is selected, all elements that match at least one of the filters will be rejected.
- "Reject non-matching elements" - if this option is selected, all elements that match none of the filters will be rejected. This is useful if, for instance, you are only interested in elements of one or two specific packages.
- "Don't show rejected elements in output" - if the option is selected, the rejected elements do not appear in any data output (GUI displays and exported data files). Deselecting this option disables the filter mechanism.
- "Ignore rejected elements for metrics calculation" - this option determines the treatment of links from or to rejected elements. Accepted elements can have all kinds of relationships (associations, inheritance relationships etc.) with rejected elements. Whether to count such relationships or not affects for example coupling and inheritance metrics for the accepted elements.
  By default, this option is deselected. Links to rejected elements are counted; the measurement values you obtain for the accepted elements are the same as when the filter mechanism is disabled.
  If this option is selected, links to rejected elements are ignored during metric calculation or rule checking. The measurement values you obtain will usually be lower than without filters.

To apply the modified filter settings, you need to re-calculate the design metrics by clicking the ↺ button on the tool bar.

4.2.3 Saving Project Settings

You can save the current project file and filter settings to a configuration file for later retrieval. This is also convenient if you intend to invoke SDMetrics via the command line (see Section 5 "Running SDMetrics from the Command Line"): you can instruct SDMetrics to read the file and filter settings

© 2002-2013. All rights reserved.

from a configuration file instead of having to specify each file and filter individually as command line arguments.

To save the current project settings, select "Project -> Save Project Settings" from the main menu. This opens a standard file chooser dialog where you can specify the configuration file to write the settings to.

4.2.4 Loading Project Settings

To load a project settings configuration, select "Project -> Load Project Settings" from the main menu. This opens a standard file chooser dialog where you can specify the configuration file to read the file and filter settings from.

4.3 Calculating and Viewing Metric Data

After you specified your project file and filter settings (see Section 4.2 "Specifying Project Settings"), select "Project->Calculate Metrics" from the menu bar, or click the button on the tool bar. SDMetrics will read your UML design file and calculate the metrics as specified. This is a fully automated process that usually takes a few seconds. You can monitor the calculation progress on the status bar.

If an error occurs during data processing (e.g., a file was not found, metric calculation failed due to a semantic error in a custom metric definition file), the calculation will abort and you are prompted with a description and location of the error. After you fixed the error (e.g., specified the correct project file, corrected the metric definition file), press the button again to restart the calculation.

Upon successful completion, you can explore the metric data in the main window. SDMetrics provides several views:

View name	Description
Metric Data Tables	Presents metric data as a set of tables
Histograms	Displays histograms for the metrics
Kiviat Diagrams	Displays Kiviat diagrams for the model elements
Rule Checker	Shows design rule violations for the UML model
Design Comparison	Compares metric values to those of a second UML model
Relation Matrices	Shows relations such as "class uses class", "actor associated with use case"
Descriptive Statistics	Concise summary of the descriptive statistics for the metric data
Graph Structures	Shows circular dependencies and connected components in model element relation graphs
Model	Displays the UML model in a tabular format
Catalog	Shows the definitions of all metrics, design rules, and relation matrices
Log	Keeps a log of previous calculation runs

Table 2: Overview of SDMetrics' views

© 2002-2013. All rights reserved.

To open a view, use the "Views" menu from the menu bar. The following views are also accessible via toolbar buttons: Metric Data Tables (▦), Histograms (▙), Kiviat Diagrams (✵), and the Rule Checker view (✔).

Many views allow for navigation to other views. For example, views displaying metric data provide links to the "Catalog" view to display the definition of the metrics, and links to the "Histogram" view to view the distribution of the metrics. After following such links, the history buttons ⇦ ⇨ on the toolbar navigate back and forth between the previously visited views.

4.3.1 Common controls in views

There are a number of control elements that are common to all or most views. This section describes those frequently used controls.

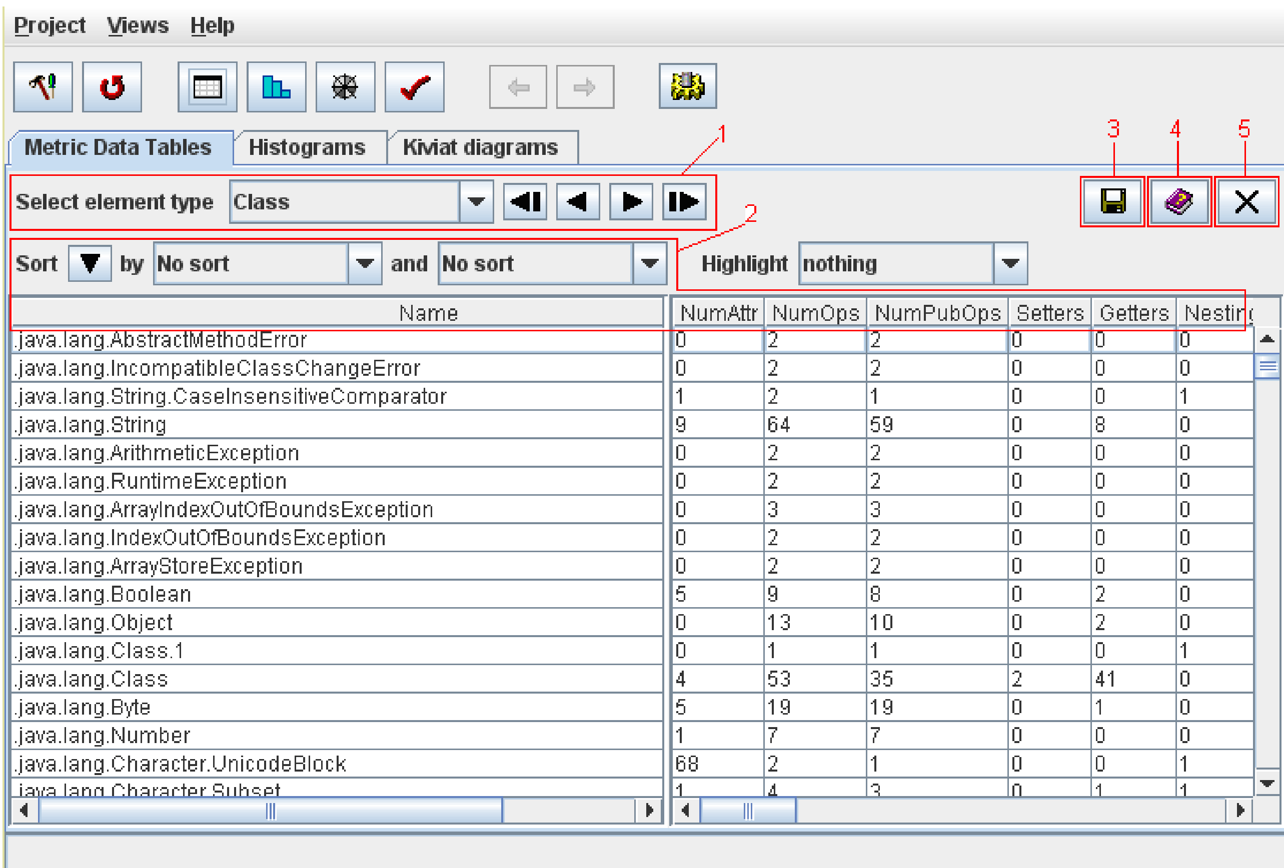


Figure 8: Common controls used in most views

1. Most views show metric data for one model element type at a time, e.g. metrics for all classes, all packages, all state diagrams, etc. These views provide a dropdown list from which to select a model element type. With the "tape deck" controls next to the list you can easily browse the various model element types.

2. Many views provide tables of data that can be sorted by columns. To sort a table by a column, click the column header. This sorts the table rows in descending order. Click the column header a second time to change the sorting order to ascending. Click the column

© 2002-2013. All rights reserved.

header a third time to restore the original order of the rows (no sort). Alternatively, you can use the control panel provided by these views to sort the table:

- Select the column to sort by from the first dropdown list labeled "by".
- You can also specify a secondary column to sort by from the second dropdown list labeled "and". Rows with equal values in the primary sort column are then sorted by the secondary column.
- Press the ▲ or ▼ button to toggle between ascending and descending sort.
- To restore the original order of table rows, select the first entry "No sort" from the sort dropdown list.

3. Most views have a save button to export the data presented in the view (tables of data or graphs, see Section 4.15.1 "Exporting Data Tables" and Section 4.15.2 "Exporting Graphs").

4. All views feature a help button to open a description of the view in the user manual.

5. All views feature a close button ✕ in the upper right corner to close the view. Alternatively, the tabs of the views feature a context menu to close the view or all other views currently opened.

4.4 The View 'Metric Data Tables'

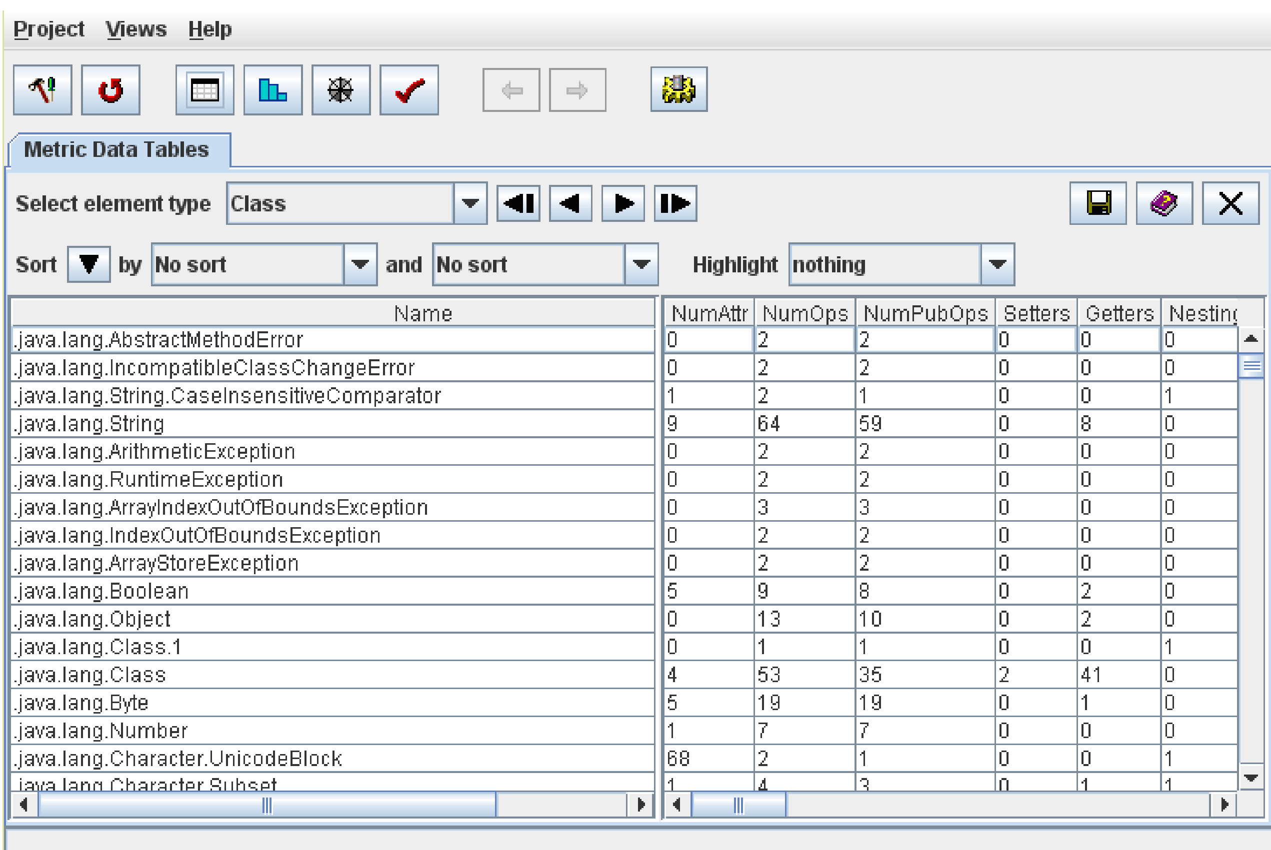


Figure 9: Table view of metrics

© 2002-2013. All rights reserved.

This view shows the metric data in a table. The left hand side of the table shows the names of the model elements that were analyzed, one element per row. The right hand side of the table shows the metric values for each model element, one metric per column. Via the control panel above the table you can select the model element type, sort the table and highlight outlying values.

For large tables, use the vertical and horizontal scroll bars of the metrics table to see more columns and rows. The left hand column showing the element names always remains visible in the table view. To adjust its width, drag the separator bar between the two table sides to a suitable position.

The context menu of the table cells of the right hand side table provides links to

- the measurement catalog to display the detailed definition of the selected metric (see Section 4.13 "The View 'Catalog'"),
- the histogram view showing the distribution of the selected metric (see Section 4.5 "The View 'Histograms'"),
- the Kiviat diagram view for the selected model element in (see Section 4.6 "The View 'Kiviat Diagrams'").

4.4.1 Highlighting Outliers

The table view supports highlighting of outliers based on one of two criteria: percentiles or distance from the mean.

Percentiles

This option highlights values above or below a given percentile. For instance, you may choose to highlight, in each column, the metric values equal or above the 95th percentile for the respective metric. This highlights the top 5% metric values in each column.

You pick the percentile you wish to highlight from the top of the highlight dropdown list in the control panel. If you choose a percentile above the median (50th percentile), all values above that percentile are highlighted. If you choose a percentile below the median, the values below that percentile are highlighted.

SDMetrics calculates percentiles using the empirical distribution function with averages. The list of percentiles is configurable, see Section 4.16.2 "Percentiles".

Distance from the mean

This option highlights metric values with a certain distance from the mean value of the metric. The distance is expressed in multiples of the metric's standard deviation. For example, you may choose to highlight metric values larger than the mean plus four times the standard deviation of the metric. Simply select the appropriate multiple from the highlight dropdown list.

"Distance from the mean" is meaningful for metrics defined on an interval or ratio scale. "Percentiles" are meaningful for all ordinal scale metrics.

© 2002-2013. All rights reserved.

To remove any highlighting from the table, choose the first entry "nothing" from the highlight dropdown list.

4.5 The View 'Histograms'

The histogram view provides a graphical representation of the distribution of a design metric.

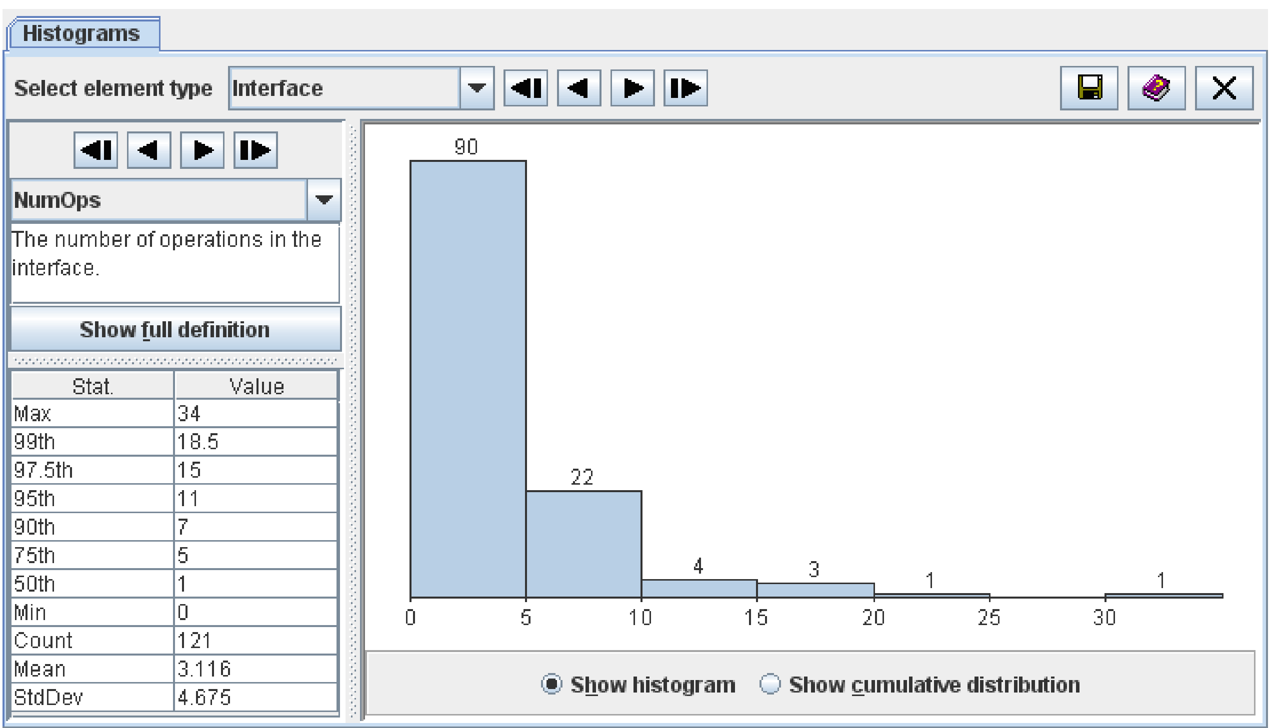


Figure 10: Histogram view

Select the metric to display from the dropdown list on the left control panel, or use the ◀▌◀ ▶ ▌▶ buttons to select the first/previous/next/last metric on the list, respectively. Press the "Show full definition" button to view the detailed definition of the metric in the measurement catalog (see Section 4.13 "The View 'Catalog'").

The table below the metric definition shows some descriptive statistics for the metric:

- Max - maximum value of the metric
- *x*th - the *x*th percentile of the metric, for various values of *x*
- Min - minimum value of the metric
- Count - the number of observations (UML model elements) for which the metric was calculated
- Mean - the average value of the metric distribution
- StdDev - the standard deviation of the metric distribution

The diagram on the right shows the distribution of the selected metric. You can choose between two types of diagrams with the radio buttons below the diagram:

© 2002-2013. All rights reserved.

- Show histogram
  The histogram is a type of bar graph that depicts the frequency of metric values in class intervals by the length of its bars. The scale of measurement on the horizontal axis is the range of the metric under consideration. It is subdivided into intervals of equal width. The plot points on the horizontal axis are the exact limits of the interval. The height of the bar for each interval is proportional to the number of values that fall into the interval. This number is also shown on top of each bar.
- Show cumulative distribution
  The cumulative distribution graph shows, for any value *x* in the range of the metric, the percentage of model elements for which the metric value is <=*x*. The scale of measurement on the horizontal axis is the range of the metric under consideration. The scale on the vertical axis is the percentage of elements below a given threshold *x* on the horizontal axis.

If you see a "Diagram not available" message instead of a graph, the measurement values of the selected metric are not numerical or do not vary at all, or both. No graph is shown for such metrics.

4.6 The View 'Kiviat Diagrams'

This view shows information for one UML design element at a time. On the control panel to the left, you can choose the element to display. The panel on the right hand side graphically displays the values of all metrics defined for the element.

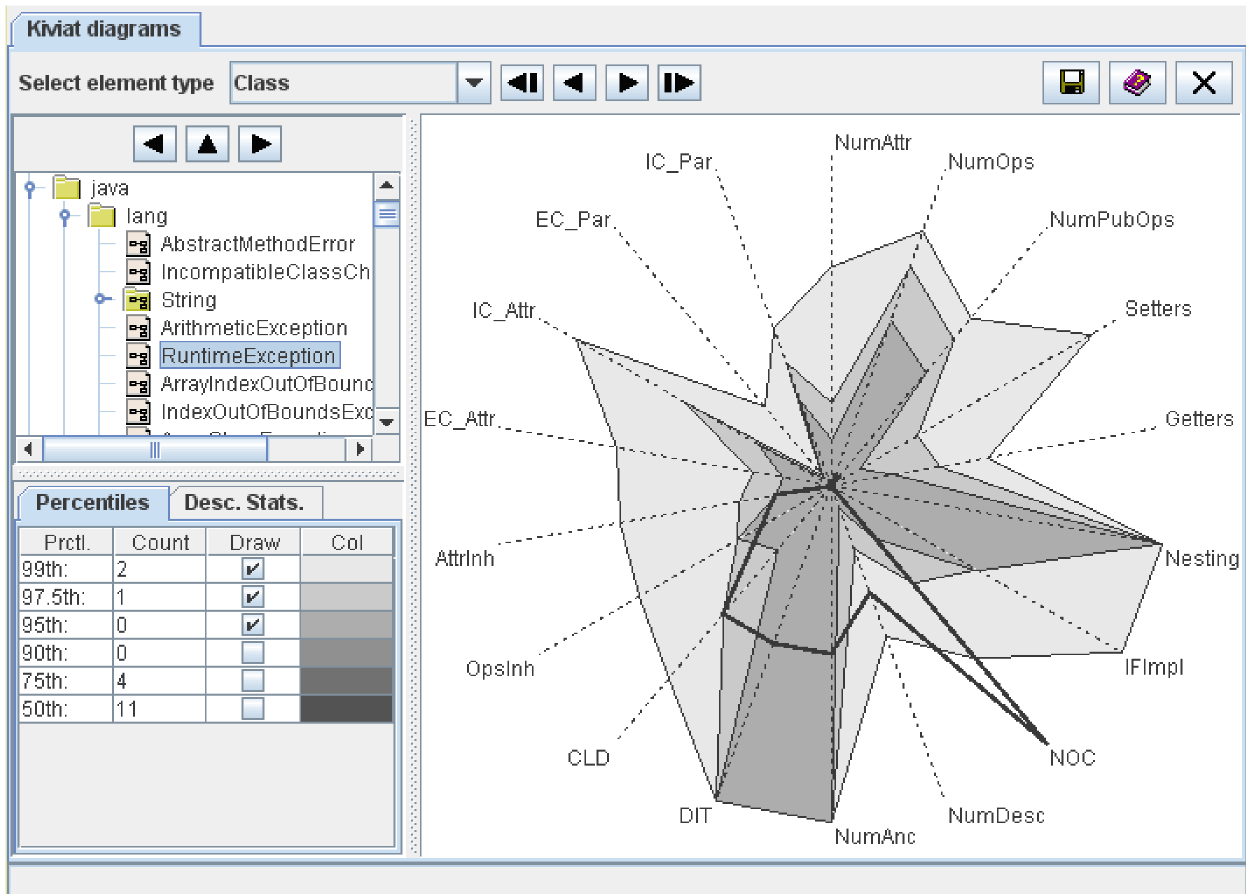


Figure 11: Kiviat Diagram View

© 2002-2013. All rights reserved.

Select the element to display from the tree at the top of the control panel. You can adjust the size of the tree with the separator bars below and to the right. Use the ◀▲▶ buttons or the cursor keys to navigate through the tree.

The graph on the right hand side is a so-called Kiviat diagram, showing the measurement values of all metrics for the selected element. Each axis (or ray) of the graph represents one metric, as labeled in the graph. The measurement scale of each axis is the range of the metric: the minimum value is located in the center, the maximum value at the outer end. The axes are linearly scaled.

The thick line connects the measurement values of the selected element for each metric on the axes. If the element has many relatively large values, the area enclosed by the thick line will be large. So the size of the enclosed area serves as an indicator of the criticality of the element.

Note that metrics with non-numerical values, or metrics that do not vary at all, are not suitable for the graph and therefore omitted. Also, the Kiviat graph can only be shown for elements with at least three suitable metrics. If you see a "Diagram not available" message instead of a graph, there are less than three suitable metrics for the selected element type.

Below the element tree you can choose to display information about metric percentiles of the selected model element, or to display the descriptive statistics for a metric shown in the graph.

Percentiles

The percentiles table shows, for various percentiles (column "Prctl."), the number of metrics for which the measurement values exceed the percentile for the selected element (column "Count"). Assuming we mostly deal with metrics where higher values indicate lower quality, a design element should be considered critical if a larger number of metric values for the element are in the upper percentiles (e.g., 90th, 95th).

The percentiles table also controls which percentiles of the metrics are displayed in the graph. Check the boxes in the column "Draw" for the percentiles to display. This gives an indication how the measurement values of the selected element compare to all other elements. Column "Color" indicates the color for each percentile on the graph.

Descriptive Statistics

If you click near one of the axis in the Kiviat diagram, a short definition of the metric and its descriptive statistics with minimum and maximum values will be displayed on this tab.

The context menu of the graph provides links the measurement catalog (see Section 4.13 "The View 'Catalog'") and histogram (see Section 4.5 "The View 'Histograms'") for the metric of the nearest axis.

4.7 The View 'Rule Checker'

Design rules and heuristics detect potential problems in your UML design, for example:

© 2002-2013. All rights reserved.

- incomplete design such as unnamed classes, states without transitions,
- incorrect design such as an interface having an association with navigability away from the interface,
- style issues such as circular dependencies among packages, a class referencing one of its subclasses,
- violation of naming conventions for classes, attributes, operations, packages,
- etc.

Appendix C: "List of Design Rules" contains the list of design rules that SDMetrics implements. Of course, you can customize this list to your needs, and define new design rules of your own. This is described in Section 8.3 "Definition of Design Rules".

The 'Rule Checker' view displays design rule violations in a table. Each row of the table represents a violation of a design rule by a design element.

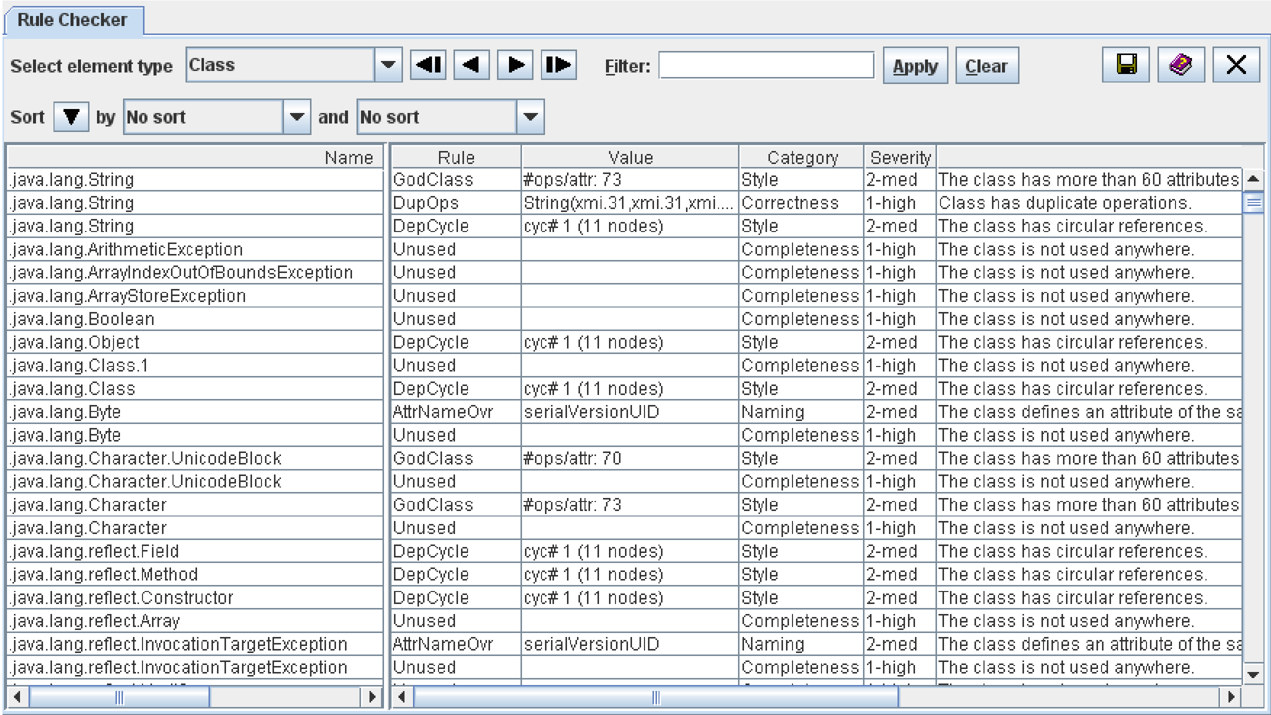


Figure 12: Design Rule Checker

The meaning of the columns is as follows:

- *Name*: Name of the design element that violates the rule.
- *Rule*: Name of the violated rule.
- *Value*: Some rules return a value that provides additional information about how the design element violates the rule.
- *Category*: The category of the rule describes the criteria the rule checks, for example naming, completeness, correctness, or style.
- *Severity*: The severity of the violated indicates how critical the violation of the rule is, and therefore how urgently it should be resolved.
- *Description*: A short description of the violated rule.

© 2002-2013. All rights reserved.

The context menu of right hand side table contains links to the measurement catalog showing the detailed definition of the violated rule in the selected row (see Section 4.13 "The View 'Catalog'"), and the Kiviat diagram for the model element in the selected row (when available).

Sort the table of design rule violations by columns, e.g., to quickly find all rule violations for a particular design element, all violations of a particular rule, or to sort the rule violations by their severity or category. Section 4.3.1 "Common controls in views" describes how to sort tables.

4.7.1 Filtering Design Rules

Some design rules only apply to certain types of UML models, e.g., models at a particular development phase (e.g., requirements, analysis, design), models of a particular application domain (e.g., embedded systems, real time systems), etc.

The measurement catalog indicates the applicable types of models for each rule. With the design rule filter, you can instruct SDMetrics to only check and report design rules that apply to your model at hand.

Writing Rule Filters

Table 3 shows some example filters and explains their function. For the example, we assume we have four application areas defined:

- `analysis` (rules for analysis models)
- `design` (rules for design models)
- `realtime` (rules for models of real-time systems)
- `pedantic` (rules about obsessive details)

© 2002-2013. All rights reserved.

Example Filter	Explanation
`design`	To only check the rules of one application area, simply specify the name of that application area. The example filter only checks rules that are applicable at the design phase.
`design&realtime`	To check rules applicable to all of several areas, combine the areas with the & operator. The example filter only checks rules that are applicable to real-time systems at the design phase.
`analysis|design`	To check rules applicable to at least one of several areas, combine the areas with the | operator. The example filter only checks rules that are applicable at the analysis phase or at the design phase.
`!analysis`	To check rules that do not apply to a particular area, precede the name of the area with the ! operator. The example filter only checks rules that do not apply to the analysis phase.
`'design'`	Some rules do not explicitly define an application area. Such rules are implied to apply to all areas. Therefore, the filter `design` will also check rules that do not specify any application area at all. To instruct SDMetrics to only check rules that *explicitly* are defined for an application area, put the application area in single quotes. The example filter only checks design rules which explicitly list "design" among their application areas.
`design&(!'realtime'|pedantic)`	Using parentheses, you can define arbitrarily complex rule filters. The example filter checks all rules applicable to the design phase that are also not explicitly defined for real-time systems or are pedantic.

Table 3: Rule filter examples.

Applying Rule Filters

To apply a filter, specify it in the "filter" text field at the top of the rule checker view, and press return or click the "Apply" button. To clear the filter, and show all design rule violations regardless of their application area, click the "Clear" button.

If your filter contains a syntax error (illegal operation, unmatched parentheses etc.) you will get an error message and the rules will not be checked. If your filter specifies an application area that is not defined explicitly by at least one rule, SDMetrics will warn you about this, but will check the rules anyway.

4.7.2 Accepting Design Rule Violations

Rules are there to be bent once in a while. That is, sometimes there is a justification why a particular model element should be allowed to violate a particular rule. Such approved rule violations should not be reported anymore for the model element. You achieve this with specific tags or comments that you add to the model element in your UML tool. The procedure differs for UML 1.x and UML 2.x models.

© 2002-2013. All rights reserved.

UML 1.x

To allow a model element *e* to violate a rule named `rulename`, define a tagged value for *e*, where the tag name is `violates_rulename` (please refer to the manual of your UML modeling tool for details on how to add tagged values). The value of the tagged value pair is ignored by SDMetrics. You can use the value for instance to document the reason why the model element is allowed to violate the rule.

For example, if after a design review a particular class is allowed to violate rules named "GodClass" and "MultipleInheritance", you add two tagged values to the class, one with tag `violates_GodClass` and the other with the tag `violates_MultipleInheritance`. From then on, violations of these two rules will no longer be reported for that class (and only these two rules, and only for that class).

UML 2.x

UML 2 no longer has tagged values as they are known in UML 1.x, but every model element can own comments. To exempt a model element from one or more rules, add one or more comments where the body text contains the string `violates_rulename` for each rule the model element is allowed to violate. Please refer to the manual of your UML modeling tool for details on how to add comments. An example comment body text could be:

```
"violates_GodClass, violates_MultipleInheritance: confirmed in
review"
```

The comment's body can contain additional text before, after, or between the `violates_rulename` tags. You can use this to document the reason why the model element is allowed to violate the rule.

Customizing rule exemption annotations

If the above methods to mark elements as exempt from certain rules do not suit you, the method is customizable to some degree, see Section 8.3.6 "Exempting Approved Rule Violations".

4.8 The View 'Descriptive Statistics'

The descriptive statistics view provides a tabular summary of the descriptive statistics for all metrics.

© 2002-2013. All rights reserved.

Descriptive Statistics

Select element type Package

Sort by No sort and No sort

Name	Sum	N	Mean	StdDev	Max	99th	97.5th	9:
NumCls	426	58	7.34482765	16.26809502	87	83	59.5	32.5
NumCls_tc	1044	58	18	58.90968323	382	308	165	70
NumOpsCls	4637	58	79.94827271	275.19827271	1776	1331	787	559.
NumInterf	121	58	2.08620691	4.60466766	27	21	15	11.5
R	2505	58	43.1896553	145.4302063	808	719	534.5	330
H	82.95030975	58	1.43017781	2.3387785	9.16666698	8.85256386	8.23491669	7.81
Ca	810	58	13.96551704	37.39170074	233	174	100.5	83
Ce	373	58	6.43103456	19.36637688	138	88.5	32.5	25.5
I	33.76302719	58	0.58212113	0.44554406	1	1	1	1
A	25.90454483	58	0.44663009	0.41998121	1	1	1	1
D	1.17914891	58	0.02033015	0.50327754	0.70710701	0.70710701	0.70710701	0.70
DN	33.03705597	58	0.5696044	0.42104185	1	1	1	1
Nesting	79	58	1.36206901	0.85220611	3	3	3	2.5
ConnComp	195	58	3.36206889	4.7928977	27	23.5	15	10
Dep_Out	0	58	0	0	0	0	0	0
Dep_In	0	58	0	0	0	0	0	0
DepPack	95	58	1.63793099	3.50787711	21	15	8.5	8

Figure 13: Descriptive statistics view

Each row of the table represents one design metric, the columns provide the descriptive statistics (minimum, maximum, mean, standard deviation, and several percentiles). The context menu of the table links to the histogram view for the selected metric (Section 4.5 "The View 'Histograms'"), and its full definition in the measurement catalog view (Section 4.13 "The View 'Catalog'").

4.9 The View 'Design Comparison'

The design comparison view allows you to compare the structural properties of two UML designs. This is useful in a number of situations.

Comparing subsequent versions of a design. Object-oriented systems are usually developed in an incremental, iterative fashion. After a design phase at the beginning of an iteration, you can compare the current design to the previous iteration:

- Size metrics deltas quantify the growth in size of the design. This serves as an input to the planning of the implementation and testing efforts for the remainder of the iteration.
- Identify parts of the system design that have undergone much change. Schedule these parts for review to ensure they are still in line with the original vision for the architecture, and if not, whether the deviation is justified.

Comparing design alternatives. If you have modeled alternative solutions to a design problem, the structural properties of the candidate designs can help support the decision which of the alternatives to choose for implementation.

© 2002-2013. All rights reserved.

4.9.1 Calculating and Viewing Metric Deltas

The steps to compare two designs with SDMetrics are as follows.

1. Calculate the metrics for the first (or older) design to compare as usual, as described in Section 4.3 "Calculating and Viewing Metric Data".
2. In the topmost text field of the design comparison view, specify the XMI file with the second (or newer) design to compare. If supported on your platform, you can also drag and drop the XMI file anywhere in the design comparison view.
3. Press the button next to the text field where you specified your second model. This will calculate the metrics for the second design, and the metric deltas to the first design.

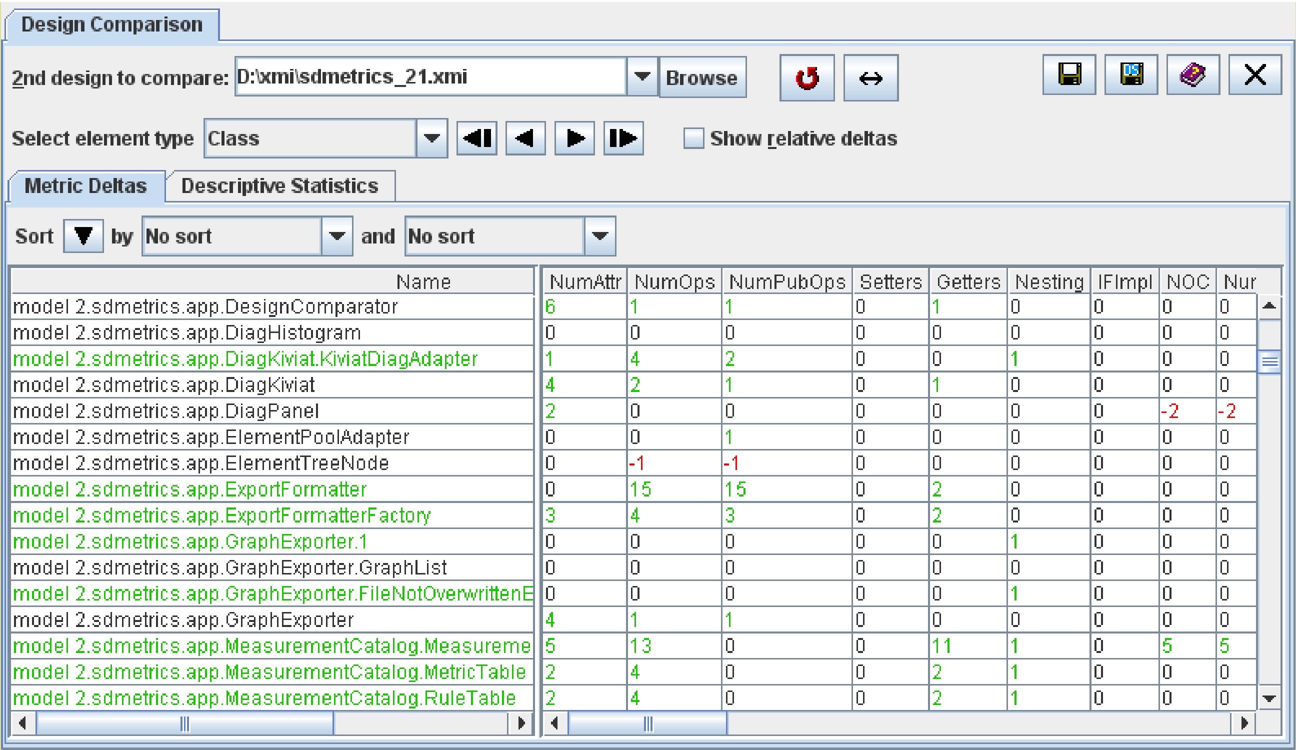


Figure 14: Design Comparison

The design comparison view provides two sets of tables: metric deltas and descriptive statistics. By default, the tables show the difference of measurement values between the first and second design (value in first design minus value in second design). To show the relative difference of the value in the second design as percentage of the value in the first design, select the "Show relative deltas" radio button next to the element type dropdown list.

The context menu of the right hand side table links to the detailed definition of the selected metric in the measurement catalog (see Section 4.13 "The View 'Catalog'").

4.9.2 Metric Deltas Table

This tab presents the metric deltas in a table. The table is organized in the same way as the metric data table view (see Section 4.4 "The View 'Metric Data Tables'"): design elements by row, metrics

© 2002-2013. All rights reserved.

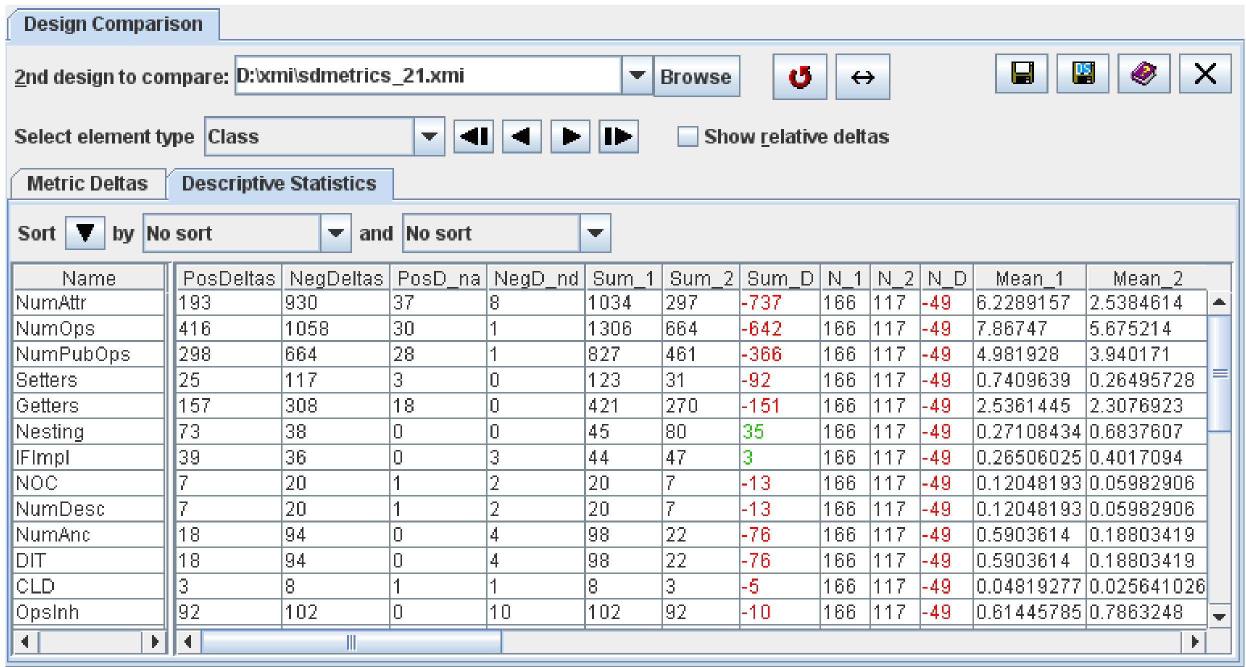
by column. In the metric columns, positive metric deltas indicate how much the metric value for the design element has increased in the second design, negative metric deltas indicate how much the metric value has decreased. Increased and decreased values are also highlighted by colors (green for increased values, red for decreased values). You can change these colors to your preferences, see Section 4.16.4 "Appearance".

For non-numerical metrics, the metric columns show the metric values for the second design. The value is preceded by a "=" if the metric value is unchanged from the first design, or a "#" if the metric value has changed. Changed values are also indicated by the color red.

When comparing designs, we will usually have elements in the first design that have been deleted in the second design, and elements added to second design that were not present in the first design. In the "Name" column, added and deleted elements are indicated by their color (green for added elements, red for deleted elements). The last column "AD_Status" also shows the status of each element: A for added, D for deleted, and empty for elements present in both designs. Use this column to sort elements by their status, and to identify the element status when exporting the table data to a file.

4.9.3 Comparative Descriptive Statistics Table

This tab sheet shows a side by side comparison of the descriptive statistics for both designs. The table shows the metrics by row.



Name	PosDeltas	NegDeltas	PosD_na	NegD_nd	Sum_1	Sum_2	Sum_D	N_1	N_2	N_D	Mean_1	Mean_2
NumAttr	193	930	37	8	1034	297	-737	166	117	-49	6.2289157	2.5384614
NumOps	416	1058	30	1	1306	664	-642	166	117	-49	7.86747	5.675214
NumPubOps	298	664	28	1	827	461	-366	166	117	-49	4.981928	3.940171
Setters	25	117	3	0	123	31	-92	166	117	-49	0.7409639	0.26495728
Getters	157	308	18	0	421	270	-151	166	117	-49	2.5361445	2.3076923
Nesting	73	38	0	0	45	80	35	166	117	-49	0.27108434	0.6837607
IFImpl	39	36	0	3	44	47	3	166	117	-49	0.26506025	0.4017094
NOC	7	20	1	2	20	7	-13	166	117	-49	0.12048193	0.05982906
NumDesc	7	20	1	2	20	7	-13	166	117	-49	0.12048193	0.05982906
NumAnc	18	94	0	4	98	22	-76	166	117	-49	0.5903614	0.18803419
DIT	18	94	0	4	98	22	-76	166	117	-49	0.5903614	0.18803419
CLD	3	8	1	1	8	3	-5	166	117	-49	0.04819277	0.025641026
OpsInh	92	102	0	10	102	92	-10	166	117	-49	0.61445785	0.7863248

Figure 15: Comparing descriptive statistics

The meaning of the first four columns is as follows:

- PosDeltas: the sum of all positive metric deltas for the metric.
  This indicates how much the metric values increased by adding new elements or modifying existing ones.

© 2002-2013. All rights reserved.

- NegDeltas: the sum of all negative metric deltas for the metric.
  This indicates how much the metric values decreased by deleting elements from the design, or modifying existing ones.
- PosD_na: the sum of all positive metric deltas for the metric, excluding added elements.
  This indicates how much the metric values have increased by modifying existing design elements.
- NegD_nd: the sum of all negative metric deltas for the metric, excluding deleted elements.
  This indicates how much the metric values have decreased by modifying existing design elements.

The remainder of the columns in the table shows the descriptive statistics including sum, mean, and percentiles for the first design, second design, and the difference for each descriptive statistic.

4.9.4 Mapping Design Elements

To calculate metric deltas, SDMetrics matches the design elements of the first design with the elements of the second design. Matches are based on the fully qualified names of the design elements (see Section 4.2.2.1 "Qualified Element Names"). This causes problems if a model element such as package has been renamed between designs. Because of the name change, the package and all of its contents can no longer be matched between designs. The package under its old name is considered to be deleted, and newly added under its new name.

To improve the element matching for renamed elements, you can define explicit element mappings. Click the ⇔ button to open the element mappings dialog.

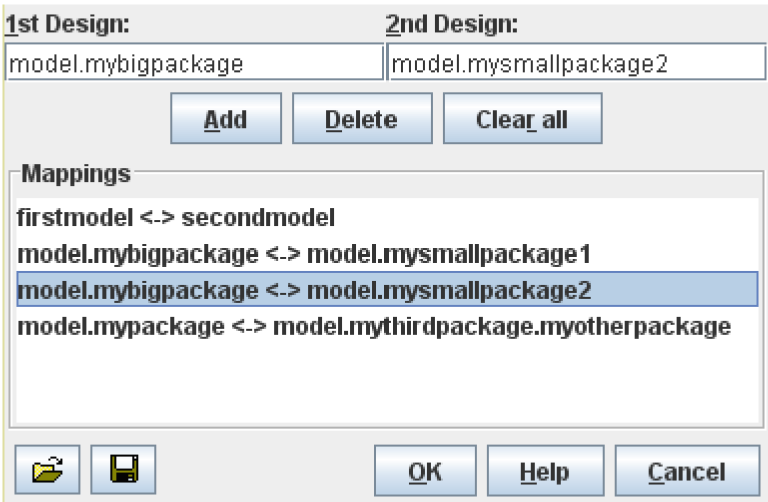


Figure 16: Element Mappings

Enter the old name of the renamed element in the first design in the upper left text field. Enter the new name of the element in the second design in the upper right text field. Press "Add" to add the mapping to the list of mappings below.

To remove a mapping from the list, select the mapping on the list and then press the "Delete" button. To remove all mappings, press the "Clear all" button. Click the button to save the current

© 2002-2013. All rights reserved.

list of mappings to a file, and the button to load a previously saved list. Or drag and drop the mapping file anywhere into the mappings dialog or the design comparison view.

Mapping Examples

- The model names of the designs to compare may differ.
  If the name of your first model is *firstmodel*, and the name of your second model is *secondmodel*, all qualified element names start with "firstmodel." and "secondmodel.", respectively.
  Define the mapping *firstmodel* <-> *secondmodel* so that the elements are properly matched.
- An element has been renamed and/or moved.
  A package *model.mypackage* of the first design has been renamed to *myotherpackage* and moved to be a subpackage of a third package *model.mythirdpackage* in the second design.
  Define the mapping *model.mypackage* <-> *model.mythirdpackage.myotherpackage* to match the package and all elements within.
- An element has been split, or joined with another element.
  A package *model.bigpackage* has been split into two packages *model.smallpackage1* and *model.smallpackage2*.
  Define two mappings: *model.bigpackage* <-> *model.smallpackage1* and *model.bigpackage* <-> *model.smallpackage2*. That way, elements in *model.smallpackage1* and *model.smallpackage2* will be mapped to elements in *model.bigpackage*.

4.9.5 Exporting Metric Deltas

The design comparison view provides two buttons for data export. To export metric deltas data to a file, click the button. To export the comparative descriptive statistics table, click the button. This will open an export dialog where you can specify file names, data format, and whether to export to single or separate files (see Section 4.15.1 "Exporting Data Tables").

4.10 The View 'Relation Matrices'

The relation matrices view shows relations between individual design elements. For example, the relation matrix in Figure 17 shows the inheritance relationships from child classes (rows) to parent classes (columns).

The rows of a relationship matrix show the source design elements from which the relationship originates, the columns contain the target design elements of the relation. The table cells indicate the presence or number of relationships between the respective source and target design element.

The dropdown list above the table shows the available relation matrices. Select the relation matrix for display from this list.

© 2002-2013. All rights reserved.

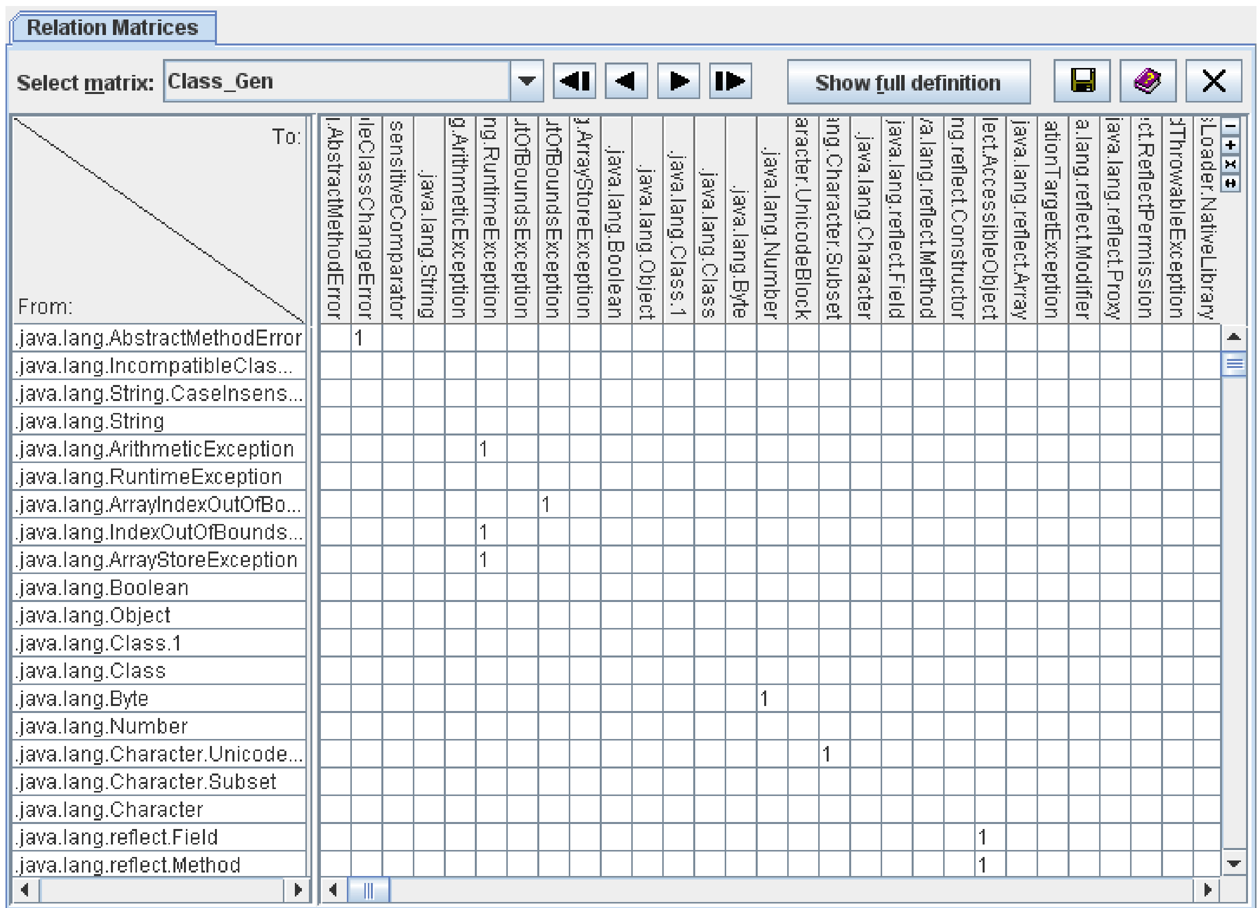


Figure 17: Relation matrix

Relation matrices can become quite large. Use the vertical and horizontal scroll bars of the table to see more columns and rows, as needed. The left hand column showing the source element names always remains visible. To adjust its width, drag the separator bar between the two table sides to a suitable position. The column header shows the names of the target elements. To increase or decrease the height of the column header, use the + and − buttons in the upper right corner of the table. You can adjust the width of the table columns with the ↔ and ›‹ buttons.

The "Show full definition" button above the matrix opens the measurement catalog with a detailed definition of the currently selected matrix (see Section 4.13 "The View 'Catalog'").

Appendix D: "List of Matrices" describes the relation matrices that SDMetrics calculates. You can define and display additional relation matrices of your own, the procedure is described in Section 8.4 "Definition of Relation Matrices".

4.11 The View 'Graph Structures'

SDMetrics features design metrics that count connected components in a graph, and design rules that check for cycles in a graph. The graph structures view shows you these connected components and cycles.

© 2002-2013. All rights reserved.

The view contains two register tabs, one for viewing cycles, the other for viewing connected components.

4.11.1 Viewing Cycles

Figure 18 shows the register tab for viewing cycles in graphs.

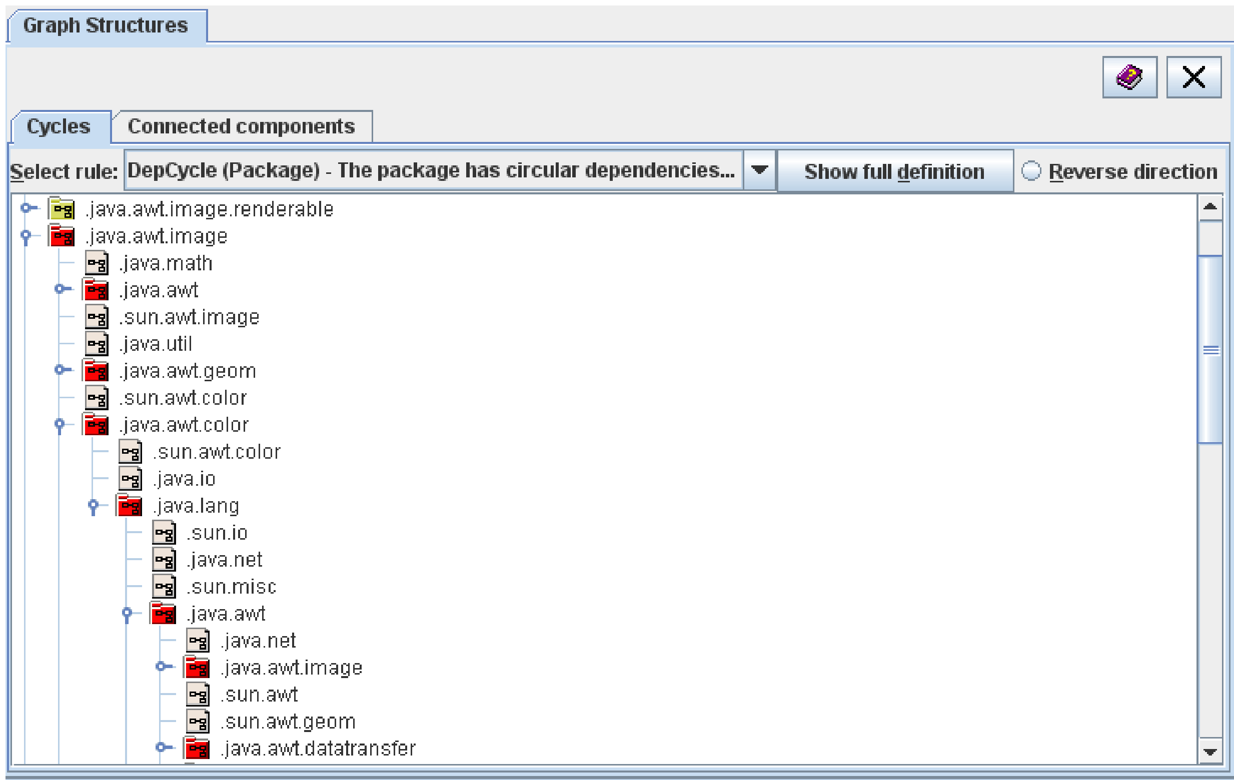


Figure 18: Graph Structures View - Cycles

There is one graph for each design rule that checks for cycles. From the dropdown list at the top, you can select the design rule for which you want to show the cycles. Click the "Show full definition" button to open the definition of the rule in the measurement catalog (Section 4.13 "The View 'Catalog'").

The tree below initially shows all model elements that depend on at least one other model element. The type of model elements and nature of the dependencies are defined by the selected design rule. Expand the model elements to the see on which other elements they depend. The icon next to each model element indicates its cycle status:

- Red icon: The element has dependencies and is part of one or more cycles.
- Yellow icon: The element has dependencies but is not part of any cycle.
- White icon: The element has no dependencies.

By expanding the elements with red icons, you can quickly trace the cycles present in the design. Note that the dependency tree is always shown, even if there are no cycles at all.

© 2002-2013. All rights reserved.

Use the "Reverse direction" button in the upper right corner to switch to a new tree that reverses the direction of the dependencies. Expanding a model element *e* shows all model elements that depend on *e*.

4.11.2 Viewing Connected Components

Figure 19 shows the register tab for viewing connected components.

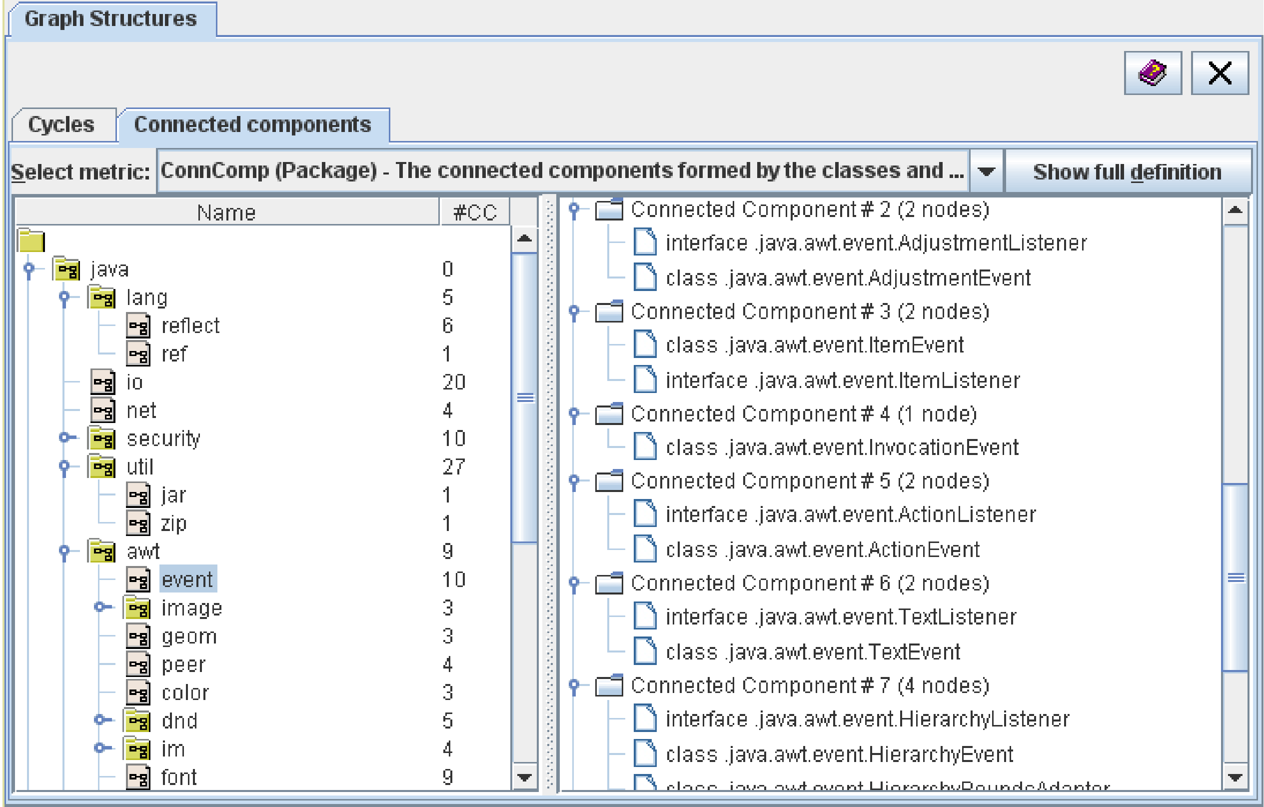


Figure 19: Graph Structures View - Connected Components

From the dropdown list at the top, you can select the metric for which you want to show the connected components. Click the "Show full definition" button to view the definition of the metric in the measurement catalog (Section 4.13 "The View 'Catalog'").

The tree on the left hand side shows the model elements for which the connected components are calculated. Column "#CC" indicates the number of connected components for each model element. If you select a model element in the tree, the connected components will be shown in a tree structure on the right hand side.

The top level nodes of the right hand side tree represent the connected components. The child nodes represent the model elements of each connected component.

© 2002-2013. All rights reserved.

4.12 The View 'Model'

This view shows the "raw" UML model information that SDMetrics extracted from your XMI files. For each element type, you obtain one table showing the model elements by row, element attributes by column.

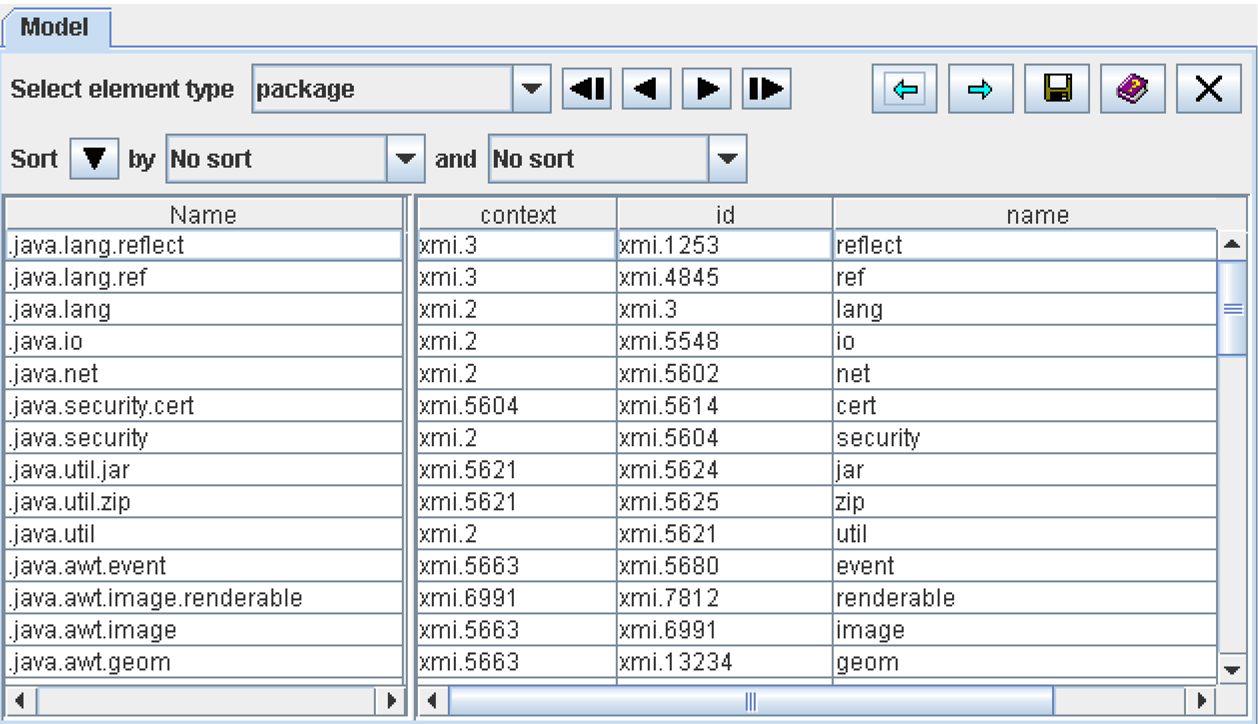


Figure 20: The View 'Model'

Figure 20 shows, for example, the table for metamodel element "package" with three of its attributes "id", "name", and "context" (see Section 7.1 "SDMetrics Metamodel" and Appendix A: "Metamodels"). The table provides the XMI id, name, and a reference to its owner (XMI id of the owner) for all packages of the UML model.

Likewise, you obtain tables for all other model element types with their respective attributes. As a result, you have a complete representation of your UML design in a table format (complete as far as SDMetrics' design measurement and rule checking requirements are concerned).

The context menu of the table provides quick access to the Kiviat diagram of the selected element, as well as simple navigation feature for cross-reference attributes. Via the context menu for cross-reference attributes you can find and show each model element referenced by the cross-reference attribute. With the ⇦ and ⇨ arrows in the upper right corner of the model view you can move back and forth within the previously selected model elements.

The most important feature of this view is the save button to export the tables to files (see Section 4.15.1 "Exporting Data Tables"). These files are easy to parse. This feature is useful if you want to build a custom application that performs operations on your UML designs, using the flexible XMI import capabilities of SDMetrics.

© 2002-2013. All rights reserved.

The model view also comes in handy when you create your own custom metamodels and XMI transformations (see Section 7 "SDMetrics Metamodel and XMI Transformation Files"). You can use this view to quickly verify if your XMI transformations work as intended.

4.13 The View 'Catalog'

The catalog view shows the definitions of the metrics, design rules, and relation matrices for the current data set, and provides literature references and a glossary for them.

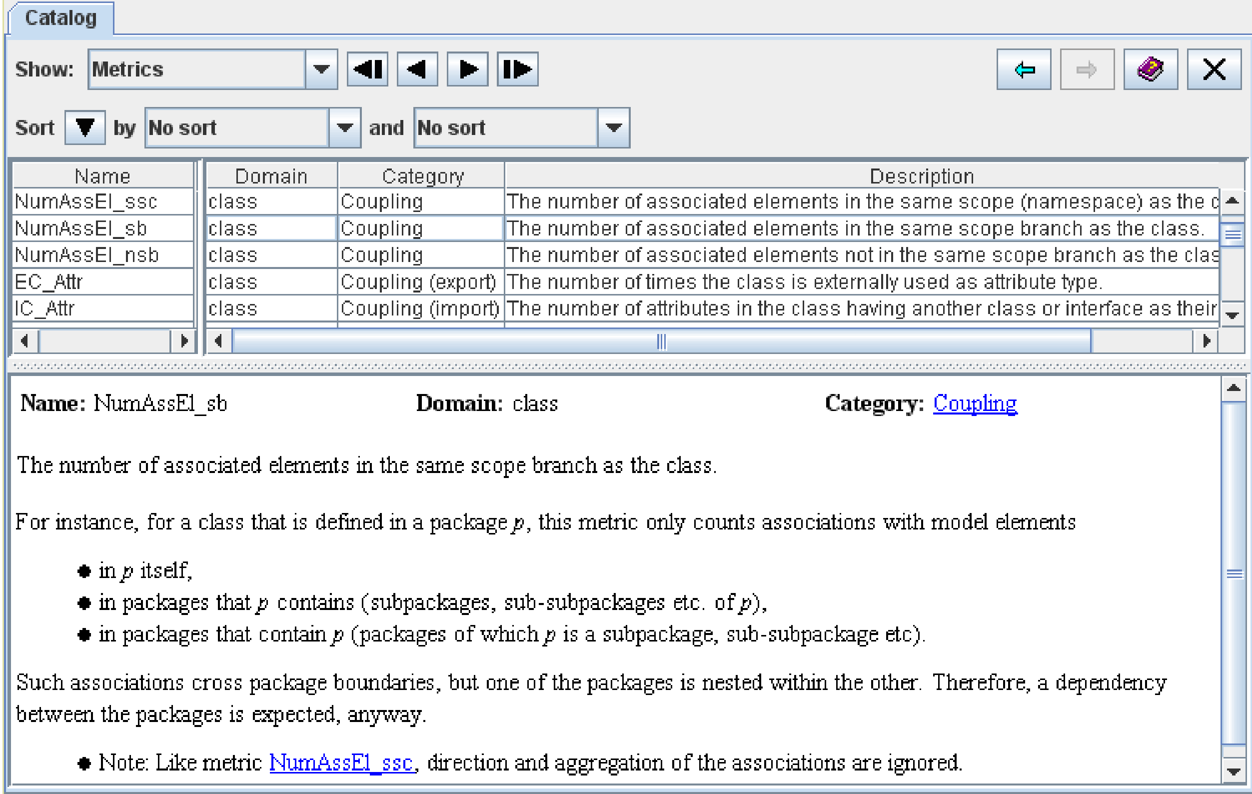


Figure 21: The View 'Catalog'

Selecting tables

From the dropdown list at the top of the catalog, you can select the following items to explore:

- Metrics - lists all design metrics in the table. For each metric, the table shows the name of the metric, its domain (the type of element it measures), its category (the property it measures), and a brief description.
- Rules - lists all design rules in the table. For each rule, the table shows the name of the rule, its domain (the element type it applies to), its category, its severity, its applicable areas, as well as a brief description of the rule.
- Matrices - lists the relation matrices in the table. For each matrix, the table shows the types of elements in the rows and columns, and a brief description of the matrix.
- References - shows a list of literature references and their bibliographic citations.
- Glossary - shows a list of measurement terms with definitions.

© 2002-2013. All rights reserved.

Showing detailed descriptions

Click any row in the table to see a detailed description of the selected item in the lower part of the window.

The detailed descriptions often contain cross-references to other metrics, rules, or matrices, include literature references, or reference terms from the glossary. These references are hyperlinks that take you to the full definition of the referenced item. Use the ⇦ and ⇨ arrows in the upper right corner of the catalog view to move back and forth within the previously visited definitions.

Sorting tables

You can sort each table by its columns, e.g., to quickly find all metrics for a particular domain, all rules of certain severity level, and so on. Section 4.3.1 "Common controls in views" describes how to sort tables.

4.14 The View 'Log'

The log view provides a log of the messages that SDMetrics generates on the progress bar during a calculation run.

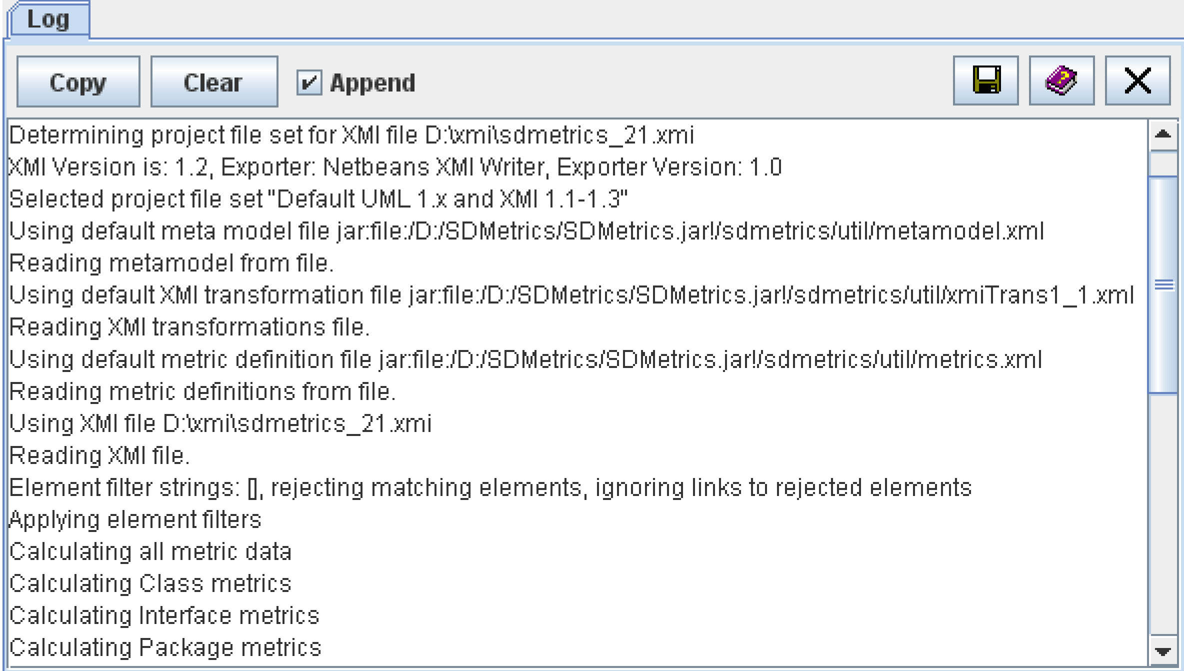


Figure 22: The 'Catalog' View

The log shows which project files have been selected by SDMetrics for processing and why, as well as the calculations that have been performed. You can use the output of the log to document calculation runs performed via the GUI.

The copy button copies the current contents of the log view to the clipboard. The clear buttons deletes all log entries from the view.

© 2002-2013. All rights reserved.

When the "Append" checkbox is selected, the log entries of subsequent calculation runs will be appended to the log. When the "Append" radio button is deselected, the log will be cleared automatically before each new calculation run.

Via the save button you can save the current log entries to a ".txt" or ".log" file.

4.15 Exporting Data

SDMetrics allows you to export data tables and graphs:

- You can export data tables from the metric data tables view, rule checker view, descriptive statistics view, design comparison view, relation matrices, and the UML model view.
- You can export graphs from the histogram and Kiviat diagram views.

SDMetrics provides separate export dialogs for data tables and graphs, which will be explained in the following sections. These dialogs allow you to customize output features that are likely to be changed frequently. Some additional output features can be set in the output preferences dialog, see Section 4.16.3 "Output".

4.15.1 Exporting Data Tables

Figure 23 shows the export dialog for data tables.

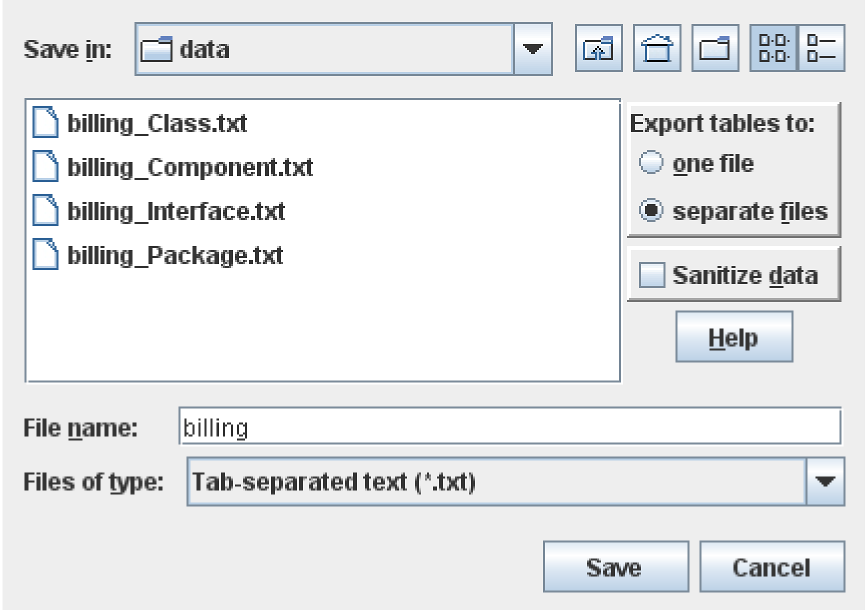


Figure 23: Data Export Dialog

To specify the export, you have a number of options:

Select export to one file or separate files. Usually, you will have several data tables, one for each element type. Here you specify if you want to store all tables in one file, or have each table written to a separate file.

© 2002-2013. All rights reserved.

Sanitize data. This option is only applicable to the data tables of the metric data tables view. The first table column of those tables shows the names of the design elements that were analyzed. Check this option to suppress the output of the element names. This can be useful if, for confidentiality reasons, you want to pass on or publish the metric data without disclosing the element names.

Select the file name. Enter the name of the file to write the data to. If you chose to export the data to separate files, the name you enter here will serve as a base name for the files created.
Example: You have tables for classes, packages, and interfaces, and enter "data" as file base name. This will create three files:

- `data_Class.txt`
- `data_Package.txt`
- `data_Interface.txt`

The file base name is extended with an underscore and the name of the model element type the file contains.

Select the file format (Files of type). The following formats are available from the "Files of type" dropdown list:

- Tab-separated text.
  Writes the data to text files with ".txt" file extension. Columns are separated by tabs, the first row contains the metric names. This format can be read by most spreadsheet software and statistical packages.
- Comma-separated vectors.
  Writes the data to text files with ".csv" file extension. Columns are separated by commas, unless you specified a different delimiter, see Section 4.16.3 "Output".
  Note: For CSV and text files you should prefer the "export to separate files" option, as these formats do not really support multiple tables in a single file.
- HTML.
  Writes the data to HTML files, using HTML tables. When you export to separate HTML files, SDMetrics will additionally create a HTML frame set for convenient viewing of all tables.
- OpenDocument Spreadsheet
  Writes the data to .ods files. This format is used by the various OpenOffice/LibreOffice incarnations, as well as many other office solutions.
- Excel XML File
  Writes the data to .xml files for Microsoft® Excel™ XP (2002) or later.
- OpenOffice.org Calc
  Writes the data to .sxc files for OpenOffice.org 1.0 Calc or later.

© 2002-2013. All rights reserved.

4.15.2 Exporting Graphs

Figure 24 shows the export dialog for graphs:

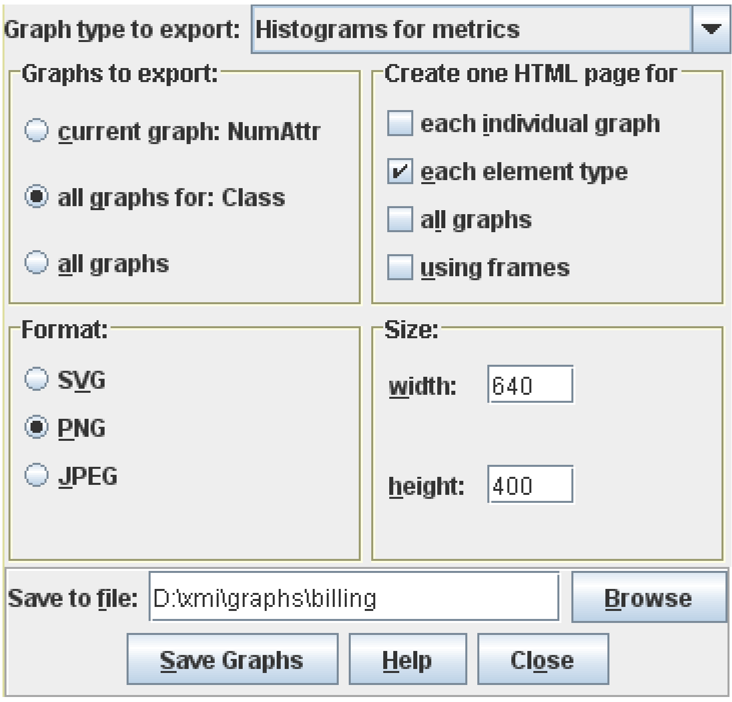


Figure 24: Export Graphs Dialog

The options for the graph export are:

Graph type to export You can export one of the three available graph types at a time:

- Histograms for metrics
- Cumulative distribution graphs for metrics
- Kiviat graphs for design elements

Graphs to export determines for which metrics or elements you wish to export graphs. You have the following choices:

- *Current graph: <name>* exports only the graph for the metric currently shown in the metric browser (or element in the element browser); the name of that metric or element is shown next to the option.
- *All graphs for: <type name>* exports the graphs for all metrics or elements of the currently selected element type; the element type name is shown next to the option.
- *All graphs* exports the graphs for all element types.

Create one HTML page for You can additionally create HTML pages that contain the exported graphs:

- *each individual graph* creates, for each exported graph, one HTML page that shows the graph.

© 2002-2013. All rights reserved.

- *each element type* creates for each element type one HTML page that shows all the graphs for that element type (e.g., a page with all class metric diagrams and so on).
- *all graphs* creates one HTML page that shows all the graphs
- *using frames* - Select this option if you want the HTML pages that show multiple graphs be organized in frames.

Format You can choose between the following file formats for the exported graphs:

- SVG - Scalable Vector Graphics
- PNG - Portable Network Graphics
- JPG images

Size Here you can specify the width and height of the graphs, in pixels.

Save to file: Specify a file base name for the graph and HTML files to be written. SDMetrics will extend the base name with element type names, metric names, and the proper file extension. For example, if you specify the base name "C:\graphs\model", exporting all graphs will generate files named "C:\graphs\model_Class_NumOps.svg", "C:\graphs\model_Package.html", and so on.

Finally, to export the graph(s) as specified above, press the "Save graphs" button. The "Close" button closes the dialog without exporting the graphs.

Note that exporting a large number of graphs in PNG or JPG format can take some time, you can monitor the progress on the status bar of the main window.

4.16 Setting Preferences

The preferences dialog is available via the menu bar "Project -> Preferences", or click the button on the tool bar. The preference options are organized in several tab sheets: Project file sets, percentiles, output, appearance, and behavior.

4.16.1 Project File Sets

SDMetrics maintains a list of project file sets (see Section 4.2.1 "Specifying Project Files"). Each project file set contains one XMI transformation file (for a specific XMI version and possibly a specific XMI exporter), one metamodel definition file, and one metric definition file.

When SDMetrics analyzes an XMI file, it retrieves the XMI version from the file, as well as the name and version of the XMI exporter that created the file. SDMetrics then selects from the list of available project file sets the one that best matches the specification of the XMI file at hand.

If you downloaded updated project files from the SDMetrics web site www.sdmetrics.com, or wrote your own project files, and want to use these files by default, you can define new project file sets or change the existing ones. That way, you do not have to specify your custom project files over and over again in the project settings dialog.

© 2002-2013. All rights reserved.

Open the preferences dialog and select the tab sheet "Project File Sets":

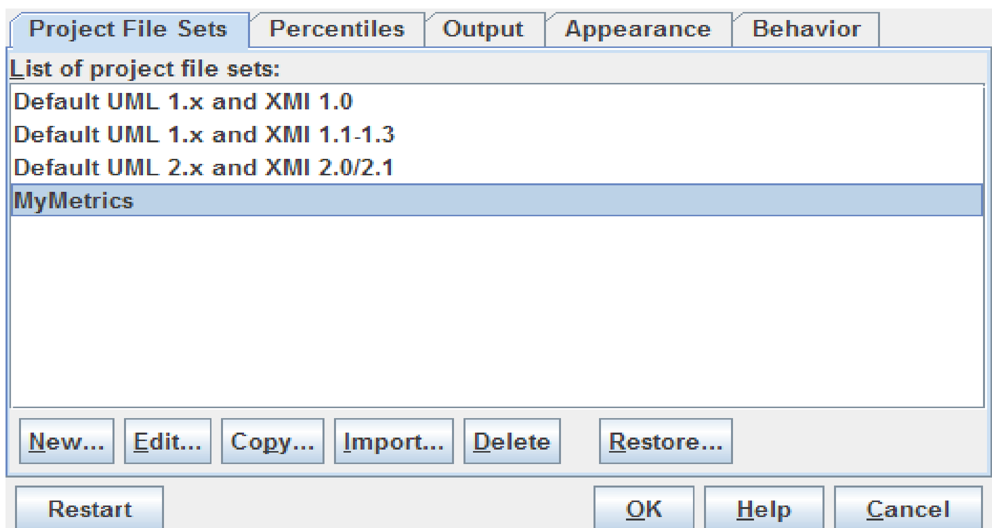


Figure 25: Project File Set Preferences

The tab sheet shows a list of all available project file sets. You can add, modify, copy, import, and delete entries from the list.

4.16.1.1 Adding New Project File Sets

To define a new project file set, click the "New..." button. This opens an editor where you can specify the parameters of the project file set:

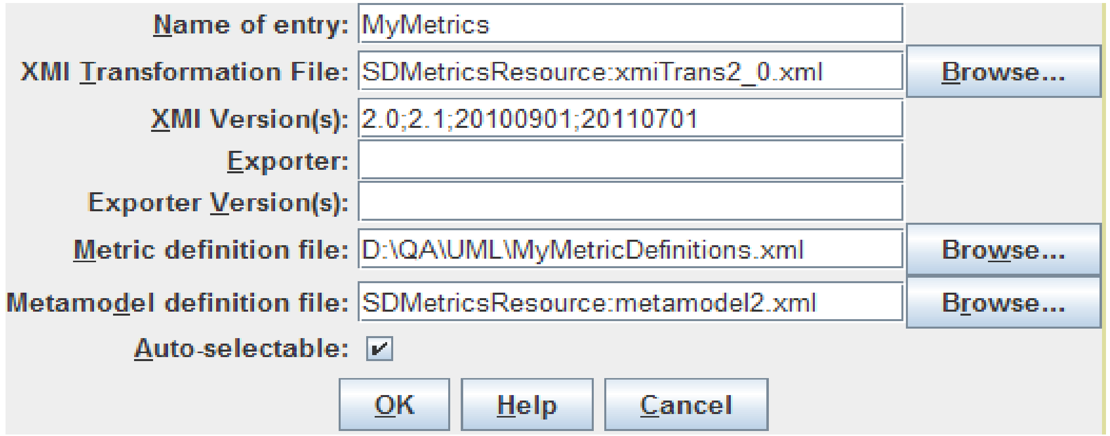


Figure 26: Project File Set Editor

The parameters of a project file set are as follows:

- **Name of entry**: The name of the project file set.
- **XMI Transformation File**: The location of the XMI transformation file of this project file set. Enter the file name in the text field, or use the "Browse..." button to locate the file via a file chooser dialog.
- **XMI version number(s)**: The XMI version numbers or version-namepaces that the XMI transformation file handles. For XMI versions 1.0 to 2.1, indicate the version number as

© 2002-2013. All rights reserved.

shown in the "version" attribute in the XMI file. For XMI version 2.4 and later, indicate the version-namespace, that is, the last part of the XMI namespace. For example, the XMI 2.4.1 namespace is "http://www.omg.org/spec/XMI/20110701", so the version-namespace is 20110701.
List all supported XMI version numbers and/or version namespaces, separated by semicolons, e.g. `2.0;2.1;20110701`.

- **Exporter**: If you created the XMI transformation file for a specific XMI exporter, enter the name of the exporter here. Use the exact name of the exporter as indicated in the `XMI.exporter` element in the documentation section of the XMI files produced by the exporter.
Leave this field empty if the XMI transformation file is not written for a specific exporter.
- **Exporter Version(s)**: If you created the XMI transformation file for a specific version of an exporter, specify the version number here. Use the exact version number as indicated in the `XMI.exporterVersion` element in the XMI files produced by the exporter. If the XMI transformation file is suitable for several exporter versions, list all supporter exporter version numbers, separated by semicolons.
Leave this field empty if the XMI transformation file is not written for a specific version of the exporter.
- **Metric definition file**: The location of the metric definition file of this project file set. Enter the file name in the text field, or use the "Browse..." button to locate the file via a file chooser dialog.
- **Metamodel definition file**: The location of the metamodel definition file of this project file set. Enter the file name in the text field, or use the "Browse..." button to locate the file via a file chooser dialog.
- **Auto-selectable**: Controls if the project file set is available for selection if you let SDMetrics automatically select the project file set to use for a given XMI input file. When disabled, the project file set can only be selected manually.

4.16.1.2 Importing Project File Sets

To import project file sets from the SDMetrics web site, download and save the Zip archive to a folder where you can store it permanently. Click the "Import..." button. This opens a file chooser dialog; select the Zip file and click "OK" to import the project file sets from the archive.

If the Zip archive contains project file sets you have already installed, you will be asked what to do with the duplicates. You can either

- **replace** your old project file sets with the new ones from the archive, or
- **keep** your old project file sets, the new ones from the archive will be added to the list and marked as updates.

If you intend to modify the project files in the Zip archive, extract the archive so you can edit the project files. In the file chooser dialog, select the file `contents.xml` that was extracted from the archive. This will import the project file set using the extracted files.

© 2002-2013. All rights reserved.

4.16.1.3 Editing, Copying, and Deleting Project File Sets

To **edit** an existing project file set, double-click its entry on the list, or select the entry and click the "Edit..." button. This opens the editor window where you can modify the details.

To **copy** the settings of an existing project file set, select its entry on the list and then click the "Copy" button. This opens the editor window where you can modify the details. When you close the editor window with the "OK" button, a new entry with the modified settings will be added to the list; the originally selected entry remains unchanged.

To **delete** an entry from the list, select the entry and click the "Delete" button.

4.16.1.4 The Default Project File Sets

SDMetrics ships with three default project file sets for the various versions of the UML and XMI:

- Default UML 1.x and XMI 1.0
- Default UML 1.x and XMI 1.1-1.3
- Default UML 2.x and XMI 2.x

These default project file sets cover all versions of the XMI currently in use. These project file sets cannot be renamed or deleted from the list, and you cannot modify their XMI version and XMI exporter name/version settings (the latter two being empty - no specific exporter). This is to ensure that SDMetrics always finds a suitable project file set. However, it is easy to override the default project file sets:

- When you specify a new project file set for any XMI version and no specific XMI exporter, SDMetrics will always select your project file set over one of its default project file sets.
- You can replace the XMI transformation, metric definition, and metamodel definition files associated with the default project file sets with your own.

Resetting default project file sets

The "Restore..." button resets the default project file sets to the "factory settings", that is, the default XMI transformations and associated metamodel and metric definition files that originally ship with SDMetrics.

Note that your custom project file sets will remain unaffected by this operation.

© 2002-2013. All rights reserved.

4.16.2 Percentiles

You can determine which percentiles are shown in the metric data views and descriptive statistics. Open the preferences dialog and select the tab sheet "Percentiles":

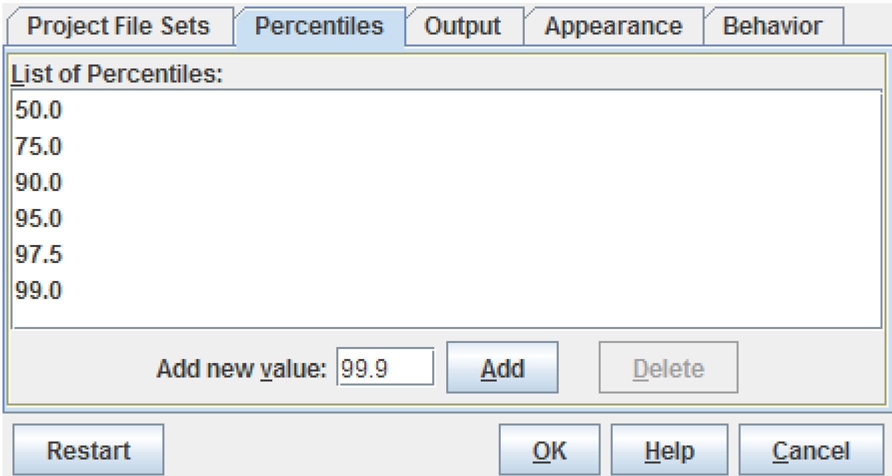


Figure 27: Percentiles Tab Sheet

The tab sheet shows the current list of percentiles. To add a percentile, type the percentile in the text field labeled "Add new value", and press the enter (or return) key, or click the "Add" button. The percentiles you enter must be valid floating-point numbers between 0 and 100. Always use a dot as the decimal point.

To delete percentiles, select the percentiles to delete on the list. You can select multiple percentiles by holding down the shift or control key on the keyboard while selecting list elements. Click the delete button to remove the selected percentiles from the list.

Note: the new percentiles settings become effective when the next set of metrics is calculated.

© 2002-2013. All rights reserved.

4.16.3 Output

The output preferences settings specify several aspects of SDMetrics' data export features. Open the preferences dialog and select the tab sheet "Output":

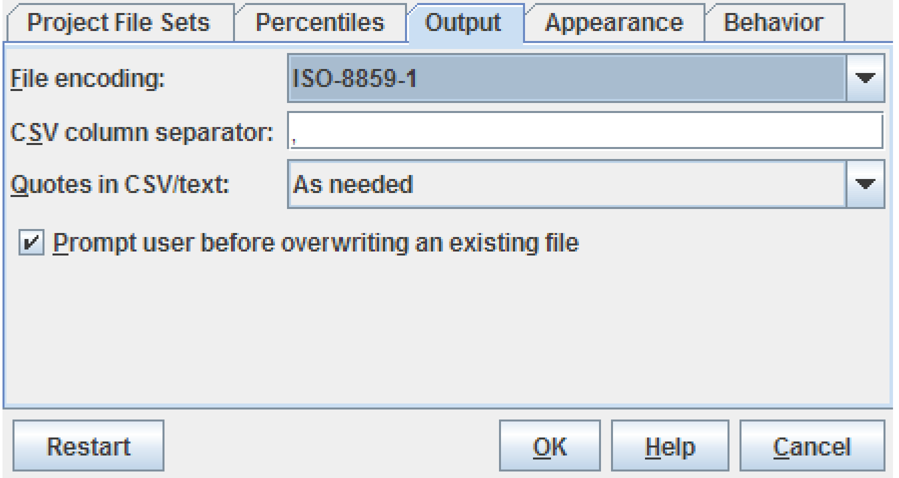


Figure 28: Output Settings

File encoding

Specifies the character encoding of all non-binary output file formats (TXT, CSV, HTML, ODS, XML, SXC and SVG). The dropdown list shows all available character encodings for your platform. Default is ISO-8859-1 (also known as Latin-1). Choose an alternative encoding if your UML model element names contain characters not available in ISO-8859-1.

CSV column separator

By default, the CSV (comma separated vector) format uses the comma to delimit the values in a row of data. If you require a different delimiter, you can enter it here.

Quotes in CSV/text

The TXT and CSV output formats use certain characters as column and line delimiter. This leads to a conflict if a value to be written itself contains one of these delimiters: writing the value "as is" destroys the overall table structure of the output file.

A standard solution to resolve this conflict is to put quotes (") around the values - characters between quotes will not be interpreted as column or line delimiters. If the value to be put in quotes itself contains a quote character, that quote character is doubled in the output file (the value 5"10 thus becomes "5""10").

SDMetrics offers three strategies to deal with conflicting values in CSV and TXT output formats:

- **Never** - values will not be set in quotes, even if they are conflicting
- **As needed** - only conflicting values will be set in quotes
- **Always** - all values will be set in quotes, conflicting or not.

© 2002-2013. All rights reserved.

Prompt user before overwriting an existing file

With this option enabled, SDMetrics will ask for your permission before overwriting existing files when you export data tables or graphs. This setting only affects the GUI, not the command line operation (which will always overwrite without asking).

4.16.4 Appearance

Here you can customize several aspects of the appearance of the SDMetrics user interface: font size, the look and feel and the design comparison colors.

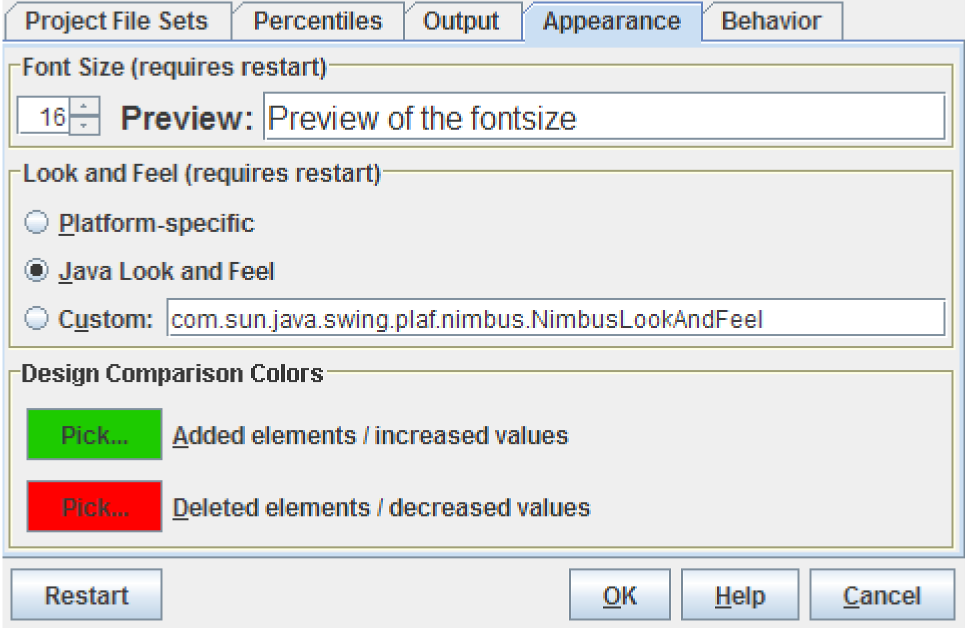


Figure 29: Appearance Dialog

Font size

Click the up and down arrows to increase or decrease the font size of SDMetrics' main window, menus, and dialogs. You can judge the resulting size by the preview fields next to the size adjustment controls.

Make sure to change the font size in small steps and "within reason". Too large or too small settings may render the application difficult or impossible to use.

Note: you need to restart SDMetrics for the new font size to become effective.

Look and Feel

To adjust the overall appearance - or look and feel - of SDMetrics, you have the following choices:

- **Platform-specific**: Uses the look and feel that best matches your operating system.
- **Java Look and Feel**: The cross-platform look and feel of Java (also known as "Metal" with "Ocean Theme").

© 2002-2013. All rights reserved.

- **Custom**: Here you can specify a custom look and feel, e.g., one provided by a third party. Refer to the documentation of the custom look and feel package for the class name to enter in the text field, and make sure the class is included in your CLASSPATH.
  For example, to use Java's Nimbus look and feel, enter:
  `com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel` (requires JRE6 update 10 or later).

Note: you need to restart SDMetrics for the new look and feel settings to become effective.

Design Comparison Colors

Here you can adjust the colors used to indicate added and deleted elements or increased or decreased measurement values for design comparisons (see Section 4.9 "The View 'Design Comparison'").

Click the button for the color you like to change. This opens a color chooser dialog where you can select the new color.

4.16.5 Behavior

On this tab sheet you can customize various aspects of SDMetrics' behavior.

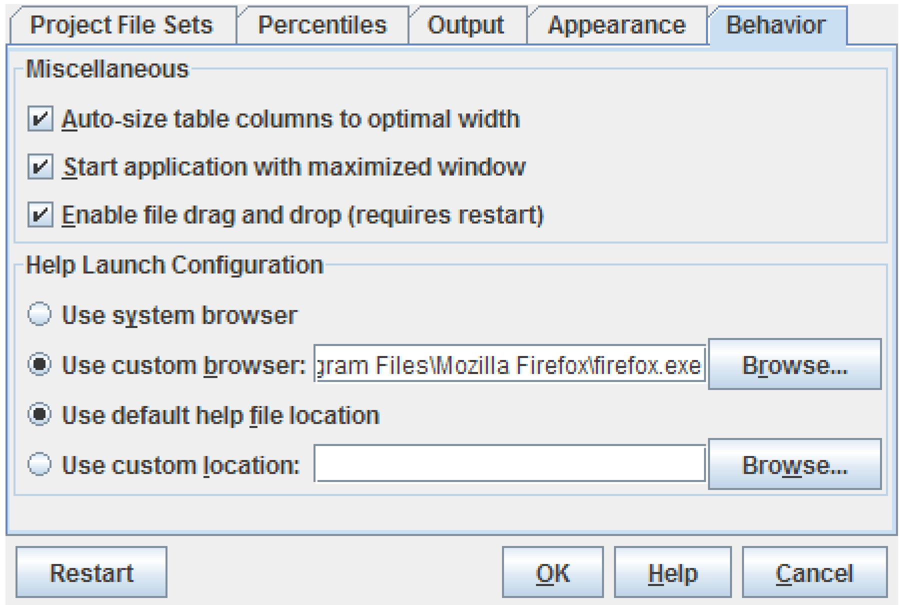


Figure 30: Behavior Dialog

Auto-size table columns to optimal width
When this option is enabled, the table columns in all data tables will individually be sized to minimal width. When this option is disabled, all table columns in a table initially have equal width.

Start application with maximized window
When this option is enabled, SDMetrics' main window will be fully maximized on startup. Deselect this option to launch SDMetrics in a smaller window.

© 2002-2013. All rights reserved.

Enable File Drag and Drop
File drag and drop is a convenient way to select an XMI or other project file (see Section 4.2.1 "Specifying Project Files") for analysis: simply drag the file from an external file system browser/explorer and drop it into SDMetrics' main window.

File drag and drop may not work on all combinations of operating system platforms and Java virtual machines. If file drag and drop causes problems on your system, you can disable this feature here. **Note:** You need to restart SDMetrics for changes to this setting to become effective.

Help Launch Configuration

The help launch configuration specifies the web browser to open the user manual with, and the location of the user manual.

- By default, SDMetrics tries to open the user manual with the web browser that the operating system associates with HTML files. If you wish to open the user manual in a different browser, select 'Use custom browser' and enter the name of the executable of the browser (complete with path information if the browser does not reside in the system path).
- SDMetrics expects the user manual to reside in the folder 'manual' of the installation directory. If for some reason you wish to open the user manual from a different location, select 'Use custom location' and specify an alternative directory that contains the SDMetrics user manual.

© 2002-2013. All rights reserved.

5 Running SDMetrics from the Command Line

You can run SDMetrics from the command line or shell scripts. This is useful to integrate SDMetrics in automated processes.

The SDMetrics command line syntax is:

```
java com.sdmetrics.SDMetrics  -xmi xmifile [-proj projfile]  [-customPF
pfsetname]
           [-meta mmfile] [-trans transformationfile] [-metrics metricsfile]
           [-filter filterstring]* [-nonmatching] [-ignore]
           [-compare 2ndxmifile] [-mapping mapfile] [-relative]
           [-gHisto] [-gCumDist] [-gKiviat] [-gFormat format]
             [-gHTMLPerDiag] [-gHTMLPerType] [-gHTMLForAll]
             [-gUseFrames] [-gWidth width] [-gHeight height]
           [-model] [-stats] [-relmat] [-rules [-rulefilter filter]]
           [-nometrics] [-s] [-one] [-f format] basename
```

The arguments in square brackets [] are optional. When followed by an asterisk (*), the bracket contents can be repeated an arbitrary number of times, otherwise, at most one occurrence is allowed. The meaning of the arguments is as follows.

Project file settings:

- `xmi xmifile`: XMI file for the design to be analyzed.
- `proj projfile`: name of a project settings file created via the GUI.
  SDMetrics reads the project settings file and processes the project files as specified in there.
- `customPF pfsetname`: specifies the name of the project files set to use for the analysis.
  If not specified, SDMetrics will automatically determine a project file set to use
- `meta mmfile`: file containing the SDMetrics metamodel.
  If you do not specify a metamodel definition file, the metamodel of the project file set is used.
- `trans transformationfile`: file containing the XMI transformations to be used to read the XMI file.
  If you do not specify a transformation file, the transformation file of the project file set is used.
- `metrics metricsfile`: file containing the definition of the metrics.
  If you do not specify a metrics definition file, the metrics definition of the project file set are used.

Filter Settings:

- `filter filterstring`: adds filter `filterstring` to the list of element filters to apply.
- `nonmatching`: When set, SDMetrics rejects elements that match none of the filters. The default (switch not set) is to reject elements that match at least one of the filter strings.

© 2002-2013. All rights reserved.

- `ignore`: When set, links to rejected elements are not considered during metric calculation. The default (switch not set) is to consider links to rejected elements.

Design comparison settings:

- `compare 2ndxmifile`: perform a design comparison with the design specified via the `-xmifile` or `-proj` switch, and output the metric deltas. Note that `2ndxmifile` contains the second or newer design.
- `mapping mapfile`: file with element mappings to use for the design comparison.
- `relative`: Show deltas as relative percentages in output.

Output file settings:

- `basename`: Base name of the file(s) the output will be written to.
- `model`: additionally create a set of tables containing the UML model. Files will have suffix "MODEL" appended to the base name.
- `stats`: additionally create a set of tables containing the descriptive statistics. Files will have suffix "SD" appended to the base name.
- `relmat`: additionally create a set of tables containing the relation matrices. Files will have suffix "RM" appended to the base name.
- `rules`: additionally create a set of tables containing the design rule violations. Files will have suffix "RULES" appended to the base name.
    - `rulefilter filter` an optional rule filter string specifying the rules to apply.
- `nometrics`: suppress export of metric data. Metric data tables are always created by default, unless you set this switch.
- `s`: sanitized output; when set, model element names are not written to the output files (metric data tables only).
- `one`: one file; when set, each set of tables is written to one file, otherwise each data table is written to a separate file.
- `f format`: format of the output file(s). Admissible formats:
  `txt`: Tab-separated text tables. This is the default if no format is specified.
  `csv`: Comma-separated text tables.
  `html`: HTML files containing the data in HTML tables.
  `ods`: OpenDocument spreadsheet files for OpenOffice/LibreOffice.
  `xml`: XML spreadsheet files for Microsoft Excel XP.
  `sxc`: Spreadsheet files for OpenOffice.org 1.0 and later.

Graph export settings:

- `gHisto`: additionally export metric histograms. Files will have suffix "HISTO" appended to the base name.
- `gCumDist`: additionally export cumulative distribution graphs for metrics. Files will have suffix "CUMDIST" appended to the base name.
- `gKiviat`: additionally export kiviat diagrams for model elements. Files will have suffix "KIVIAT" appended to the base name.
- `gFormat format`: format for the graphs to generate: SVG (default), PNG, or JPG
- `gHTMLPerDiag`: create one HTML page for each individual graph

© 2002-2013. All rights reserved.

- `gHTMLPerType`: create one HTML page for each element type
- `gHTMLForAll`: create one HTML page for all graphs
- `gUseFrames`: use frames for HTML graph pages
- `gWidth width`: width of the graphs in pixels (default: 640)
- `gHeight height` height of the graphs in pixels (default: 400)

Examples:

- `java com.sdmetrics.SDMetrics -proj mypsettings.txt -s -f html outfile`
  Reads the project files and filter settings from file `mypsettings.txt` and writes the sanitized output to separate HTML files.
- `java com.sdmetrics.SDMetrics -xmi myDesign.xmi data`
  Reads XMI file `myDesign.xmi` with standard XMI transformations, metamodel, and metrics, and writes the output in separate text files with tab-separated tables (output files start with name `data`).
- `java com.sdmetrics.SDMetrics -xmi myDesign.xmi -rules -rulefilter design -format html data`
  Reads XMI file `myDesign.xmi` with standard XMI transformations, metamodel, and metrics, writes the metric data to separate HTML files (output files start with name `data`), checks the design rules and writes the design rule violations to separate HTML files (output files start with name `dataRULES`).
- `java com.sdmetrics.SDMetrics -xmi myDesign.xmi -metrics myMetrics.xml -filter #.java -filter #.javax -nometrics -stats -one -f sxc dstats`
  Reads XMI file `myDesign.xmi` with standard XMI transformations and metamodel, calculates metrics defined in `myMetrics.xml`, filters elements defined in the java and javax packages, and writes the descriptive statistics for the metrics to an OpenOffice.org Calc workbook file named `dstats.sxc`.
  (Note: on some platforms, the # character may have a special meaning and needs to be protected from the shell, usually by preceding it with a backslash or enclosing the filter name in quotes).
- `java com.sdmetrics.SDMetrics -xmi myDesign.xmi -filter MyModel.MyPackage -nonmatching -model -one -f csv myDes`
  Reads XMI file `myDesign.xmi` with standard XMI transformation and metamodel, filters all elements outside package `MyPackage`, and writes the metric data to file `myDes.csv`, and dumps the model to file `myDesMODEL.csv`.
- `java com.sdmetrics.SDMetrics -xmi myOldDesign.xmi -compare myNewDesign.xmi -mappings MyMappings.txt -stats -one -f xml deltas`
  Reads XMI files `myOldDesign.xmi` and `myNewDesign.xmi`, calculates the metric deltas using the element mappings from file `MyMappings.txt`, and writes the metric deltas and comparison of descriptive statistics to files `deltas.xml` and `deltasDS.xml`, respectively.
- `java com.sdmetrics.SDMetrics -xmi myDesign.xmi -one -f xml -gHisto -CumDist -gFormat PNG -gHTMLforall -gUseFrames myDes`
  Reads XMI file `myDesign.xmi`, writes the metric data to file `myDes.xml`, creates a set of PNG files with histograms (myDesHISTO_*.png) and cumulative distributions (myDesCUMDIST_*.png) for all metrics, and creates a set of HTML pages with frames that

© 2002-2013. All rights reserved.

show the graphs (myDesHISTO_index.html, myDesCUMDIST_index.html plus additional HTML navigation pages).

© 2002-2013. All rights reserved.

6 Design Measurement

In this section, we describe the theory behind design measurement - what are the structural properties of a design and why can design metrics be considered quality indicators. We also discuss how design measurement data can be interpreted, how it should not be interpreted, and provide pointers to some useful data analysis techniques in the context of software design measurement.

6.1 Design Metrics and System Quality

A fundamental distinction for product quality attributes is between external and internal attributes [ISO9126,FP96].

- *External attributes* are properties or features of the product that are externally visible (hence the name), for example, reliability and maintainability. External attributes of products can only be measured with respect to how the product relates to its environment. Poor reliability, for instance, is visible to the user if the software system does not perform as expected. To measure external attributes directly requires additional information besides the product itself. For instance, reliability can be measured in terms of the mean time to failure of the operational product. Thus, external attributes can be measured directly only some time after the product is created.

- An *internal attribute* of a product can be measured in terms of the product itself. Examples of internal product attributes are structural properties such as size or coupling. All information that is needed to quantify the internal attribute is available from a representation of the product. Therefore, these attributes are measurable during and after creation of the product. Internal attributes, however, describe no externally visible quality of a product, and therefore have no inherent meaning in themselves. For example, during the operation of a system, the users will not notice whether the components of that system have loose coupling or not.

If structural design properties are not inherently meaningful, why should we care about them at all? Because we assume they have a causal impact on external quality. Undesirable structural properties such as high coupling or low cohesion indicate a - sometimes necessary - high *cognitive complexity*. Cognitive complexity is the mental burden put on the persons who have to deal with the design (developers, inspectors, testers, maintainers, etc.). The high cognitive complexity in turn leads to poor external quality, such as increased fault-proneness, or decreased maintainability and testability.

To summarize, external attributes are inherently relevant to the stakeholders in a software system, but can be measured directly only late in the development process. Internal attributes are early available but are not inherently meaningful. They become meaningful only when they are seen as indicators of (or in relation to) external attributes. Besides early availability, the advantages of design measurement are:

- Objectivity - the design assessment is repeatable and does not rely on subjective judgment.
- Automatable - with SDMetrics, measurement of even large designs can be performed quickly and at a low cost.

© 2002-2013. All rights reserved.

6.2 Structural Design Properties

This section discusses the pertinent structural properties that can be measured for UML designs. For each property we provide the following information:

- *Definition*: a short definition of the property.
- *Impact on quality*: the system quality attributes the property is hypothesized to impact, and why.
- *Empirical results*: qualitative summary of results from empirical studies investigating the usefulness of measures of the structural property. The summary is based on a literature survey and comprehensive empirical investigation of design metrics, see [BW02] for full details.

6.2.1 Size

Definition

Design size metrics measure the size of design elements, typically by counting the elements contained within. For example, the number of operations in a class, the number of classes in a package, and so on.

Impact on quality

Size metrics are good candidates for developing cost or effort estimates for implementation, review, testing, or maintenance activities. Such estimates are then used as input for project planning purposes and the allocation of personnel.

In addition, large sized design elements (e.g., big classes or packages) may suffer from poor design. In an iterative development process, more and more functionality is added to a class or package over time. The danger is that, eventually, many unrelated responsibilities are assigned to a design element. As a result, it has low functional cohesion. This in turn negatively impacts the understandability, reusability, and maintainability of the design element.

Therefore, interfaces and implementations of large classes or packages should be reviewed for functional cohesion. If there is no justification for the large size, the design element should be considered for refactoring, for instance, extract parts of the functionality to separate, more cohesive classes.

Empirical results

Empirical studies consistently confirm the importance of size as the main cost driver in a software project. Size metrics are also consistently good indicators of fault-proneness: large methods/classes/packages contain more faults. However, since size metrics systematically identify large design elements as fault-prone, these metrics alone are not suitable to find elements with high fault density.

© 2002-2013. All rights reserved.

6.2.2 Coupling

Definition

Coupling is the degree to which the elements in a design are connected.

Impact on quality

Coupling connections cause dependencies between design elements, which, in turn, have an impact on system qualities such as maintainability (a modification of a design element may require modifications to its connected elements) or testability (a fault in one design element may cause a failure in a completely different, connected element). Thus, a common design principle is to minimize coupling.

Most coupling dependencies are directed - the coupling usually defines a client-supplier relationship between the design elements. Therefore, it is useful to distinguish import coupling ("using", "fan-out") and export coupling ("used", "fan-in"), which we discuss in the following.

Import coupling

Import coupling measures the degree to which an element has knowledge of, uses, or depends on other design elements. High import coupling can have the following effects:

- Decreased maintainability: changes to the supplier may necessitate follow-up changes (ripple effects) to the client.
  The stability of the supplier is a factor to consider here. High coupling to elements that are not likely to change is less harmful than coupling to "hot spots".
- Decreased understandability, increased fault-proneness: elements with high import coupling operate in large context, developers need to know all the services the element relies on, and how to use them.
- Decreased reusability: To reuse a class or package with high import coupling in a new context, all the required services must also be made available in the new context.

Export coupling

Export coupling measures the degree to which an element is used by, depended upon, by other design elements. High export coupling is often observed for general utility classes (e.g., for string handling or logging services) that are used pervasively across all layers of the system. Thus, high export coupling is not necessarily indicative of bad design.

Again, an important issue to consider here is stability. High export coupling elements that are likely to change in the future can have a large impact on the system if the change affects the interface. Therefore, high export classes should be reviewed for anticipated changes, to ensure that these changes can implemented with minimal impact.

© 2002-2013. All rights reserved.

Empirical results

Coupling metrics have consistently been found to be good indicators of fault-proneness. It seems worthwhile to investigate different dimensions of coupling: import and export coupling, different coupling mechanisms, distinguishing coupling to COTS libraries and application-specific classes/packages. Coupling metrics are suitable to identify design elements with high fault density. Therefore, coupling metrics greatly help to identify small parts of a design that contain a large number of faults.

6.2.3 Inheritance

Definition

Inheritance-related metrics are concerned with aspects such as

- depth/width of the inheritance graph
- number of ancestors/descendents of a design element
- inherited size
- polymorphism, method overriding, etc.

Impact on quality

Deep inheritance structures are hypothesized to be more fault-prone. The information needed to fully understand a class situated deep in the inheritance tree is spread over several ancestor classes, thus more difficult to overview.

Similar to high export coupling, a modification to a design element with a large number of descendents can have a large effect on the system. Make sure the interface of the class is stable, or that anticipated modifications can be added without affecting the inheritance hierarchy at large.

Empirical results

Empirical studies show that effects of the use of inheritance on system qualities such as fault-proneness vary greatly. Depending on factors such as developer experience, system quality can benefit or suffer from the use of inheritance, or be unaffected by it.

Thus, inheritance metrics should not be relied on for decision making before their impact on system quality is not demonstrated in a given development environment. Extant inheritance metrics per se are not suitable to distinguish proper use of inheritance from improper use.

Also, inheritance is not very frequently used in designs. Typically, only a small percentage of the classes in a system will participate in inheritance relationships. As a consequence, inheritance-related metrics tend to have low variance and are difficult to use (see Section 6.3.1 "Descriptive Statistics").

© 2002-2013. All rights reserved.

6.2.4 Complexity

Definition

Complexity measures the degree of connectivity between elements of a design unit. Whereas size counts the elements in a design unit, and coupling the relationships/dependencies leaving the design unit boundary, complexity is concerned with the relationships/dependencies between the elements in the design unit. For instance, counting the number method invocations among the methods within one class can be considered a measure of class complexity, or the number of transitions between the states in a state diagram.

Impact on quality

High complexity of interactions between the elements of a design unit can lead to decreased understandability and therefore increased fault-proneness. Also, testing such design units is more difficult.

Empirical results

In practice, complexity metrics are often strongly correlated with size measures. Large design units that contain many design elements within are also more likely to have a large number of connections between the design elements.

Thus, while complexity metrics are good indicators of qualities such as fault-proneness, they provide no new insights in addition to size metrics.

6.2.5 Cohesion

Definition

Cohesion is the degree to which the elements in a design unit (package, class etc.) are logically related, or "belong together". As such, cohesion is a semantic concept.

Cohesion metrics have been proposed which attempt to approximate this semantic concept using syntactical criteria. Such metrics quantify the connectivity (coupling) between elements of the design unit: the higher the connectivity between elements, the higher the cohesion.

Cohesion metrics often are normalized to have a notion of minimum and maximum cohesion, usually expressed on a scale from 0 to 1. Minimum cohesion (0) is assumed when the elements are entirely unconnected, maximum cohesion (1) is assumed when each element is connected to every other element.

Not normalized metrics are based on counts of connections between design elements in a unit (e.g., method calls within a class). As such, not normalized metrics are conceptually similar to complexity metrics.

© 2002-2013. All rights reserved.

Impact on quality

A low cohesive design element has been assigned many unrelated responsibilities. Consequently, the design element is more difficult to understand and therefore also harder to maintain and reuse. Design elements with low cohesion should be considered for refactoring, for instance, by extracting parts of the functionality to separate classes with clearly defined responsibilities.

Empirical results

In practice, cohesion metrics are only of limited usefulness:

- Not normalized cohesion metrics often are strongly related to size metrics. This makes sense since, as discussed in the section on size metrics, large classes or packages may in fact suffer from low cohesion. Such cohesion metrics then are, of course, good quality indicators, but they are redundant with size metrics - they provide no new information about the design element.
- Normalized cohesion metrics do not consistently have a bearing on system qualities. I.e., we cannot conclude from a high or low cohesion value that a class is, e.g., more or less fault-prone. Either, the theoretical negative impact of low cohesion on system quality is not always that critical in practice, or, the cohesion metrics simply fail to identify design elements with unrelated responsibilities.

6.3 Data Analysis Techniques

In this section, we summarize a number of data analysis techniques that are useful in the context of design measurement, and provide a "roadmap" to their use.

- Before starting with design measurement, be aware of your measurement goals. You do not measure for the sake of measurement, but to help plan and control the development process, and/or to improve the quality of the product. Design measurement can help here by supporting effort estimation, monitoring progress, and identifying the areas in your design where improvements are likely to have a high payoff.

- Review the set of design metrics for completeness - are there structural features of your designs you deem important that SDMetrics does not yet cover? For instance, you may want to include metrics to count certain stereotyped elements that have a special meaning in your development environment (e.g., stereotypes to mark variation points in a reference architecture of a product line). Use SDMetrics' flexibility to define new metrics.

- Use descriptive statistics (Section 6.3.1 "Descriptive Statistics") and techniques such as PCA (Section 6.3.2 "Dimensional Analysis") to identify a minimal, nonredundant set of metrics that is meaningful for your design practices.

- This reduced set of metrics can be used for class/package rankings to identify areas with potential design problems (Section 6.3.3 "Rankings").

- Over time, you can build up a design measurement database and establish quality benchmarks (Section 6.3.4 "Quality Benchmarks").

© 2002-2013. All rights reserved.

- In more mature organizations that regularly perform process measurement (fault tracking, effort data), you can use design measurement for quality predictions (Section 6.3.5 "Prediction Models").

6.3.1 Descriptive Statistics

Descriptive statistics characterize the distribution of values for a design metric in terms of its mean and median value, interquartile ranges, and variance (or standard deviation).

The range and distribution of a metric determines the applicability of subsequent analysis techniques. Low variance metrics do not differentiate design elements very well and therefore are not likely to be useful predictors. Descriptive statistics allow us to determine if the data collected from two or more projects are comparable, stem from similar populations. If not, this information will likely be helpful to explain different findings across projects.

SDMetrics calculates and displays descriptive statistics for design metrics, see Section 4.5 "The View 'Histograms'" and Section 4.8 "The View 'Descriptive Statistics'".

6.3.2 Dimensional Analysis

In Section 6.2 "Structural Design Properties" we have noted that even though metrics have been defined to capture different aspects of a software design, in practice they often are correlated with each other. That is, they measure essentially the same thing - some of the metrics are redundant.

Redundant metrics provide no new design information. They can be discarded without loss of information, and should be discarded to facilitate the use of measurement data for decision making. Design measurement tools often come with a large set of metrics, and you should expect many of metrics to be redundant. SDMetrics is no exception here.

The difficulty is that, depending on design practices used in a development environment, two metrics may be redundant in one software system, but not in another. There is no such thing as a canonical set of non-redundant metrics that captures all important design properties and is valid for all systems. Therefore, SDMetrics opts for a rich set of metrics, to lower the risk of missing important design aspects, at the prize of some redundancy among the metrics.

Techniques such as principal component analysis (PCA) can be used to identify and eliminate redundant metrics. PCA is a standard technique to identify the underlying, orthogonal dimensions (which correspond to properties that are directly or indirectly measured) that explain relations between the variables in a data set. Examples of the application of PCA are demonstrated in [BWDP00,BWL01].

Using a technique such as PCA within a few projects, you can identify a reduced set of largely orthogonal metrics for your development environment.

6.3.3 Rankings

Most metrics have an underlying hypothesis of the form: the higher the measurement value (e.g., size, import coupling), the stronger the negative impact on quality (fault-proneness, effort,

© 2002-2013. All rights reserved.

maintainability). Therefore, when considered in isolation, design metrics only allow relative statements about the system quality, e.g. "package A displays stronger coupling than package B, therefore, package A is more likely to suffer from quality problems than package B."

An admissible interpretation is therefore to sort the model elements by a metric, and review the design of the model elements with the highest values: Are those high values justified, should the model element be considered critical?

There is no definitive answer to the question how many elements to choose for review from the top of the sorted list. One strategy could be to select so-called "outliers". Another strategy could be to select a certain number or percentage of model elements from the top, based on available resources for reviewing.

SDMetrics supports this kind of interpretation of measurement data. The histograms in the metric view allow you to visually identify outliers (Section 4.5 "The View 'Histograms'"). In the table view, you can sort the elements by a metric, and highlight elements in the upper percentiles for a metric (Section 4.4 "The View 'Metric Data Tables'").

What about thresholds?

One recurring suggested use of design metrics is that they can be used to build simple quality benchmarks based on thresholds. If a design metric exceeds a certain threshold, the design element is either rejected and must be redesigned, or at least flagged as "critical". It is difficult to imagine why a threshold effect would exist between, for example, size metrics and fault-proneness. This would imply a sudden, steep increase in fault-proneness in a certain size value range, something that would be difficult to explain. Also, empirical data does not support this idea [BEGR00].

6.3.4 Quality Benchmarks

The idea of benchmarks is to compare structural properties of a design with properties of previous system designs that are 'known to be good' and have stood the test of time. To this end, measurement values for selected design metrics obtained from previously developed, successful designs are stored in a database. If a new or modified design is to be evaluated, the same measurements are applied. Then, the distribution of each design metric is compared to the distribution of these metrics stored in the database.

As an example, Figure 31 shows a (fictitious) distribution of the number of operations invoked from within a class. The vertical axis indicates, for each value on the horizontal axis (number of invoked operations), the percentage of classes that have that particular value. In the example, the distribution of the new classes follows closely the distribution of the benchmark, except for values 7 and 8, which occur more frequently. Such deviations from expected distributions pinpoint potential risk areas. These areas could then be inspected to verify whether such a deviation is justified, or if a redesign of that part should be considered.

© 2002-2013. All rights reserved.

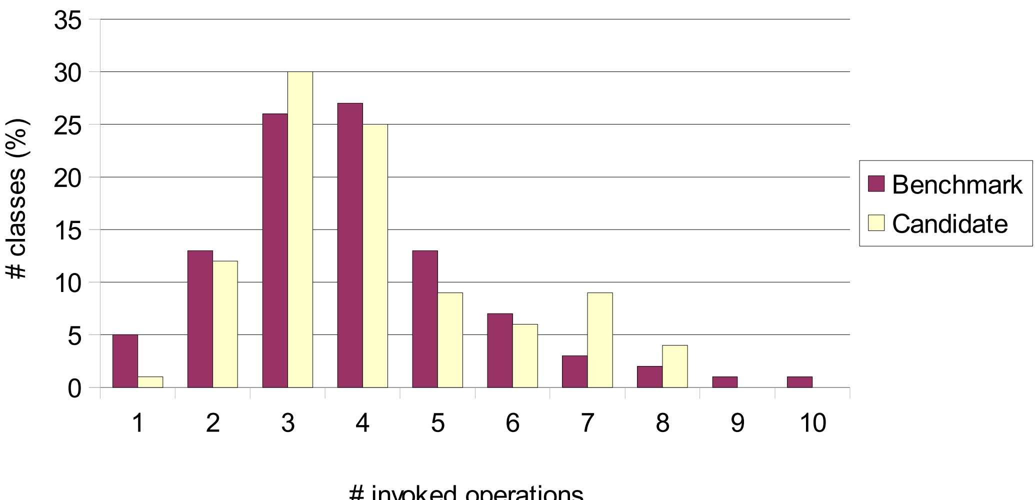


Figure 31: Benchmarking

The example also illustrates why the examination of distributions provides more information than simply defining thresholds for each metric that are not to be exceeded. In the example, the number of invoked operations is not exceedingly high for the candidate design - the benchmark suggests 10 as an upper value. A simple threshold test would have missed the potential design problem for the cluster of classes with 7 and 8 operation calls.

6.3.5 Prediction Models

Prediction models try to estimate the future quality of a system from internal quality attributes that are measurable at present. This is achieved by empirically exploring the relationships between internal and external quality from systems developed in the past, and applying these findings to new systems.

In the following, we describe how to build and use a prediction model for class fault-proneness from the structural properties of a class. Figure 32 depicts the steps involved in building the prediction model.

The starting point is a system design that has been created in the past. We apply SDMetrics to the design to obtain structural properties data for the classes in the design, collected from the various diagram types (class, object, collaboration, sequence, and state diagrams). In addition, fault data (e.g., from inspections, testing, or post-release faults) has to be collected and the faults per class recorded.

© 2002-2013. All rights reserved.

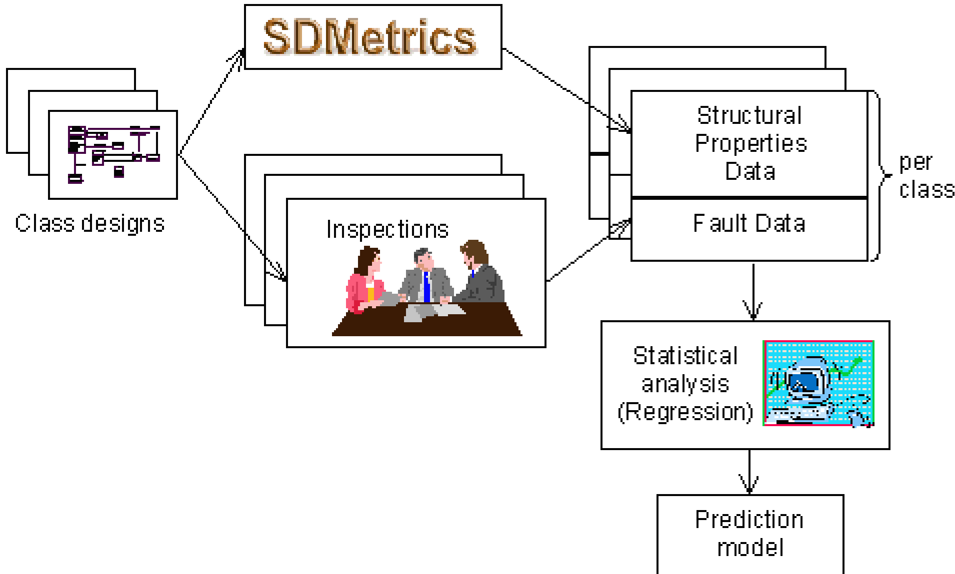


Figure 32: Building a Prediction Model

We now have a set of classes enriched with structural properties data and fault data. On this data set we perform a statistical analysis (e.g. classification or regression analysis) to identify relationships between fault data and structural properties. The result of this analysis is a prediction model, e.g., in the form of a regression equation. The prediction model computes a predicted fault-proneness or predicted number of faults from the structural properties of a class. This model can be used to make predictions for new classes, as depicted in Figure 33.

The starting point in applying the prediction model is a new design candidate. We apply SDMetrics to this design to again obtain the structural properties measurement data for the classes. This data is then fed into the prediction model. Using the now known relationship between the structural properties and faults, the prediction model calculates, for instance, for each class a probability that a fault will be found in the class upon inspection.

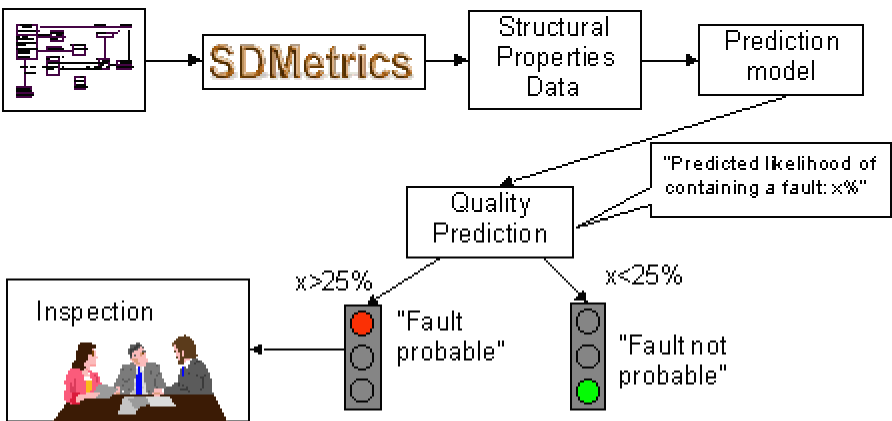


Figure 33: Using the Prediction Model

The output of the prediction model is useful for decision-making. For instance, we may decide that for classes with a high predicted fault-proneness, say, above 25%, the class design shall undergo

© 2002-2013. All rights reserved.

quality assurance (QA) activities such as inspections, extensive unit testing etc. Or, we may rank the classes by their predicted fault-proneness, and select the highly fault-prone classes from the top of the list for QA activities, until the allocated resources for QA are depleted.

Thus, the effort for QA activities can be focused on the classes that are more likely to contain faults. The benefits are manifold:

- effectiveness of QA increases,
- system quality increases as more faults are found,
- development cost decreases as faults are found earlier when they are cheaper to repair.

Note: what kind of prediction the model makes - e.g., predicted number of faults, or the likelihood a fault is found in a class during inspection, testing, or post-release - depends on the statistical analysis techniques used, and the type of fault data from which the model is built.

Prediction models for other system qualities can be built in the same way, for example, models to predict implementation and test effort from design size.

The advantage of using prediction models is that they provide a mapping from hard-to-interpret design measurement data ("size=12, coupling=7, ...") to easily interpreted external quality data ("predicted fault-proneness of class X: 78%", "predicted effort to implement package Y: 104 person hours"). The result is an absolute, quantitative statement (within certain error bars) about the external quality of a system, expressed in the same unit in which the external quality is measured.

Also, prediction models address the problem that a complex system quality attribute such as fault-proneness is influenced by many factors - various dimensions of size, coupling and so on. Approaches such as quality benchmarks, which investigate one design metric at a time to characterize fault-proneness, fail to take the combined effect of all factors into account. Prediction models provide a sound method to combine these multiple factors into one cohesive model.

Empirical evidence shows that highly accurate prediction models can be built from structural properties, and that they are beneficial in highlighting trouble areas, as well as in supporting project planning and steering [BWDP00, BWL01, BMW02, CK98, NP98, LH93].

© 2002-2013. All rights reserved.

7 SDMetrics Metamodel and XMI Transformation Files

To define and calculate design metrics, SDMetrics needs to know what types of design elements are present in a UML design, and what are their interrelationships. The UML, standardized by the Object Management Group (OMG), has a well-defined metamodel, based on the Meta Object Facility (MOF™), also a standard of the OMG.

XMI (the XML Metadata Interchange format), yet another standard of the OMG, is a mechanism to create a textual (XML) representation of models based on the MOF, such as the UML. XMI defines a set of production rules that prescribe how to serialize the elements of a model to XML, and how to generate a DTD or schema for the XML files thus generated. However, this standard is not immediately suitable for the definition of design metrics for a number of reasons:

- Different versions of the XMI exist, which yield different representations of UML models. At the time of this writing, XMI versions 1.2, 2.0, and 2.1 are widely used, and XMI 1.0 and 1.1 are still around, too, though less frequently used.
- The XMI DTDs or schemas for UML are huge (e.g., the XMI 1.2 DTD for UML1.4 has over 110 pages), which is too unwieldy for the purpose of defining metrics.
- In practice, tool vendors do not always fully adhere to the UML and XMI standards, they often deviate in minor (and sometimes not so minor) ways. This is one cause why, in practice, tool interoperability via XMI exchange does not always work as intended.

SDMetrics therefore uses a reduced and simplified 'metamodel for the UML'. This metamodel defines the UML elements that SDMetrics knows about, and contains all the relevant information to calculate metrics. This simplified metamodel also abstracts from the various versions of the XMI that are used to represent UML designs. To support a specific version of the XMI, we need to define a mapping of SDMetrics metamodel elements onto the XMI elements for the given version of the XMI. SDMetrics stores these mappings in "XMI transformation files".

When would I need to worry about all this?

Knowledge of the SDMetrics metamodel is needed if you want to define new metrics or rules of your own (see Section 8 "Defining Custom Design Metrics and Rules"). The metric and rule definitions make copious references to the elements and relationships defined in the metamodel.

In addition, knowledge of XMI transformation files is needed if

- an XMI exporter you use does not fully comply with the XMI standard or official UML metamodels, and you need to modify the XMI transformations files to account for this,
- you want to define new metrics that take proprietary and/or tool-specific XMI extensions into account. For instance, the representation of UML diagram layout information is not standardized in UML 1.x, and tool vendors usually define proprietary extensions to store this information. You can extend the SDMetrics metamodel and modify the XMI transformation files for such XMI extensions, and define metrics for them.

© 2002-2013. All rights reserved.

(Note: since XMI is not limited to the representation of UML models, but any models based on the MOF, it is possible to write SDMetrics metamodels and XMI transformation files for other MOF-based metamodels, and define metrics to perform measurements on these models.)

7.1 SDMetrics Metamodel

The SDMetrics metamodel defines which UML model elements types (e.g., classes, packages, associations, and so on) SDMetrics knows about, and what information is stored with each UML model element. This information provides the basis for the definition and calculation of design metrics.

The SDMetrics metamodel is defined in an XML file of the following structure (for a formal definition, see Appendix E: "Project File Format Definitions"):

```
<sdmetricsmetamodel version="2.0" >
  <modelelement name="element1">
    <attribute name="attr1" type="data" multiplicity="one" />
    <attribute name="attr2" type="ref" multiplicity="many" />
    ..
  </modelelement>
  <modelelement name="element2" parent="element1">
    <attribute name="attr3" type="extref" multiplicity="one" />
     ...
  </modelelement>
  ...
</sdmetricsmetamodel>
```

The metamodel definition file is a list of metamodel element definitions enclosed in `<sdmetricsmetamodel>` tags. The attribute "version" indicates the version number of the oldest version of SDMetrics with which the file can be used.

A metamodel element definition is enclosed in `<modelelement>` tags. The required attribute `name` specifies the name of the metamodel element (e.g., class, operation etc). The optional attribute `parent` specifies a parent metamodel element; the inheritance mechanism will be explained at the end of the section. Stored with each metamodel element is a set of metamodel attributes representing data fields and cross-references to other model elements. These are specified in a list of `<attribute>` definitions. We distinguish data attributes, cross-reference attributes, and extension references.

- Data attributes store a piece of data for a model element, e.g., the name of a class, the visibility of an operation.
- Cross-reference attributes store a reference to another model element, e.g., the type of an attribute, the recipients of a message.
- Extension references are a special type of cross-reference attributes used for UML extensions via profiles. Section 8.7 "Defining Metrics for Profiles" describes how to use extension references in detail.

© 2002-2013. All rights reserved.

We also distinguish single-valued and multi-valued attributes:

- Single-valued attributes only store a single data item or model element reference,
- Multi-valued attributes store a set of data items or model element references.

An `<attribute>` definition has three XML attributes:

- `name` (required): Indicates the name of the metamodel attribute.
- `type` (optional): Takes value "data" for data attributes, "ref" for cross-reference attributes, and "extref" for extension references. The default value is "data".
- `multiplicity` (optional): Takes value "one" for single-valued attributes, "many" for multi-valued attributes. The default value is "one".

For example, the definition of the metamodel element "operation" is as follows:

```
<modelelement name="operation">
  <attribute name="id" />
  <attribute name="name" />
  <attribute name="context" type="ref" />
  <attribute name="visibility" />
</modelelement>
```

An operation has four single-valued attributes: data attributes `id`, `name`, and `visibility`, and cross-reference attribute `context`. In the example, the meaning of the attributes is as follows:

- `name` is the name of the operation in the UML design,
- `id` is the XMI identifier of the operation, which is used in the XMI file to reference the operation,
- `visibility` indicates if the operation is public, private, etc.
- `context` is a reference to the owner of the operation (e.g., a class, an interface, or a use case).

These attribute meanings are not expressed in the metamodel definition. It is the job of an XMI transformation file (see Section 7.2 "XMI Transformation Files") to retrieve the intended information for each model element and attribute from an XMI file.

Note that extension reference attributes must be single-valued. A metamodel element can have at most one extension reference attribute.

Metamodel inheritance

To simplify the specification of metamodels, a metamodel element may inherit the attributes of another metamodel element. The parent metamodel element is specified via the `parent` attribute in the metamodel element definition:

```
<modelelement name="element2" parent="element1">
     <!-- additional attributes of element2 (optional) -->
</modelelement>
```

© 2002-2013. All rights reserved.

In the example, an instance of model element "element2" has all attributes of "element1", and possibly additional attributes.

By default, all metamodel elements inherit from a special metamodel element named `sdmetricsbase`, which must be defined explicitly in every metamodel definition file. In the default metamodel that is shipped with SDMetrics, the `sdmetricsbase` element defines the attributes "id", "name", and "context" which all elements must possess. See Appendix A: "Metamodels" for a list of all metamodel elements and a description of their attributes.

Note that parent model elements must be defined before any of their child elements in the metamodel definition file. Consequently, the "sdmetricsbase" model element must be the first one defined in the file.

If the parent model element defines an extension reference attribute, the child model elements inherit the extension reference and therefore cannot define new extension references of their own.

7.2 XMI Transformation Files

In Section 7.1 "SDMetrics Metamodel", we have seen how to specify SDMetrics metamodels. For each metamodel element, we can define data attributes and cross-reference attributes to other model elements. An XMI transformation file specifies how to retrieve the information pertaining to each SDMetrics metamodel element and its attributes from the XMI file.

7.2.1 XMI Transformation File Format

An XMI transformation file is an XML file of the following format (see Appendix E: "Project File Format Definitions" for a formal definition):

```
<xmitransformations version="2.0" >
  <xmitransformation ...xmitransformation attributes... />
    <trigger ...trigger attributes... />
    <trigger ...trigger attributes... />
    ...
  </xmitransformation>
  <xmitransformation ...xmitransformation attributes... />
   <trigger ...trigger attributes... />
   ...
  </xmitransformation>
  ...
</xmitransformations>
```

The transformation file defines a list of `xmitransformation` elements, each of which has a list of triggers. Each `xmitransformation` element provides XMI information for one SDMetrics metamodel element, each trigger provides XMI information for one attribute of the metamodel element.

The root element `xmitransformations` encloses the list of XMI transformations. The "version" attribute (required) indicates the version number of the oldest version of SDMetrics with which the XMI transformation file can be used.

© 2002-2013. All rights reserved.

7.2.2 XMI Transformations and Triggers

To illustrate how to define XMI transformations and triggers, we consider again the example of "operation" metamodel elements in Section 7.1 "SDMetrics Metamodel". An operation model element has four attributes: an `id`, a `name`, a `context` (operation owner), and a `visibility`.

Below is the representation of an operation as expressed in an XMI 1.0 file. The places that contain information we are interested in are set in boldface:

```
<Foundation.Core.Operation xmi.id="xmi.1632">
                          <!-- 1. operation id  -->
  <Foundation.Core.ModelElement.name>printStackTrace
                                     <!-- 2. operation name  -->
  </Foundation.Core.ModelElement.name>
  <Foundation.Core.ModelElement.visibility xmi.value="public"/>
                                     <!-- 3. operation visibility  -->
  <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
  <Foundation.Core.Feature.ownerScope xmi.value="instance"/>
  <Foundation.Core.BehavioralFeature.isQuery xmi.value="false"/>
  <Foundation.Core.Operation.isAbstract xmi.value="false"/>
  <Foundation.Core.Feature.owner>
    <Foundation.Core.Classifier xmi.idref="xmi.1605"/>
                            <!-- 4. operation context  -->
  </Foundation.Core.Feature.owner>
  ...
</Foundation.Core.Operation>
```

The following XMI transformation describes how the required information is retrieved from an XMI 1.0 document:

```
<xmitransformation modelelement="operation"
                   xmipattern="Foundation.Core.Operation" recurse="true">
  <trigger name="id" type="attrval" attr="xmi.id" />
  <trigger name="name" type="ctext" src="Foundation.Core.ModelElement.name" />
  <trigger name="visibility" type="cattrval"
           src="Foundation.Core.ModelElement.visibility" attr="xmi.value"/>
  <trigger name="context" type="gcattrval"
           src="Foundation.Core.Feature.owner" attr="xmi.idref"/>
</xmitransformation>
```

The XMI transformation is enclosed within the `xmitransformation` tags. The opening tag takes the following attributes:

- attribute `modelelement` (required): the SDMetrics metamodel element for which this transformation is defined. In our case, it's the operation.
- attribute `xmipattern` (required): the XMI element that represents our UML element. In XMI 1.0, operations are specified with the XMI element "Foundation.Core.Operation"
- attribute `recurse` (optional): is set to true if the model element can serve as context for subelements. In our case, operations can have parameters as subelements, so we set the attribute to true (the XMI transformation for parameters is taken care of in a separate XMI transformation).
- attribute `condition` (optional): specify a condition for the attributes of the XMI element that must be fulfilled for this XMI transformation to become effective. Conditional XMI transformations will be discussed in Section 7.2.3.6 "Conditional XMI Transformations".

© 2002-2013. All rights reserved.

Each trigger in our transformation describes how to retrieve the information for one SDMetrics metamodel attribute. Each trigger has two required attributes:

- attribute `type`: indicates how the trigger retrieves information from the XMI file. The `type` can be one of the following: attrval, ctext, cattrval, gcattrval, constant, ignore, and xmi2assoc.
- attribute `name`: the name of the SDMetrics metamodel attribute to which this trigger pertains. In the above example, each trigger refers to exactly one attribute of the operation metamodel element: id, name, visibility, context.

The meaning of the remaining attributes is dependent on the trigger type. We describe each trigger type in following.

7.2.2.1 Trigger Type "attrval"

This trigger instructs SDMetrics to retrieve the value from an XMI attribute of the XMI element that represents the metamodel element. In our example, the "id" information is stored in the attribute "xmi.id" of the Foundation.Core.Operation element. We retrieve its value with the trigger

```
<trigger name="id" type="attrval" attr="xmi.id" />
```

The "attr" attribute of the trigger indicates the XML attribute we are interested in. In most cases, XML attributes only store single values. Exceptions are element references. For example, a partition ("swimlane") in an UML1.x activity graph may be serialized in XMI1.2 as follows:

```
<UML:Partition name="mySwimlane" xmi.id="xmi12" contents="xmi35 xmi61 xmi115" />
```

The XML attribute "contents" contains the XMI IDs of the states of the partition, separated by spaces. In the SDMetrics metamodel, the model element "partition" has a *multi-valued* attribute "contents" (see Appendix A.1 "Metamodel for UML 1.3/1.4"). With the trigger

```
<trigger name="contents" type="attrval" attr="contents" />
```

SDMetrics will extract each part of the XML attribute, and store it as separate value in the multi-valued SDMetrics metamodel attribute.

7.2.2.2 Trigger Type "ctext"

This trigger retrieves the value from the text enclosed by a child element in the XMI tree.
In our example, the name of the operation is stored as text enclosed in the Foundation.Core.ModelElement.name element. With the trigger

```
<trigger name="name" type="ctext" src="Foundation.Core.ModelElement.name" />
```

we retrieve the value for attribute "name" from the text enclosed within the Foundation.Core.ModelElement.name tags. The child element is specified via the "src" attribute of the trigger.

© 2002-2013. All rights reserved.

7.2.2.3 Trigger Type "cattrval"

This trigger retrieves the value from an XMI attribute of a child element in the XMI tree.
In our example, the visibility information of the operation is stored in the attribute "xmi.value" of the child element "Foundation.Core.ModelElement.visibility". We retrieve its value with the trigger

```
<trigger name="visibility" type="cattrval"
         src="Foundation.Core.ModelElement.visibility" attr="xmi.value"/>
```

The "src" and "attr" attributes of the trigger specify the element/attribute we are interested in.

We can also use this trigger for multi-valued attributes. For example, a partition in a UML2 activity may be serialized in XMI2.0 as follows:

```
<group xmi:type='UML:ActivityPartition' name='mySwimlane' xmi:id='xmi12'>
  <containedNode xmi:idref='xmi35'/>
  <containedNode xmi:idref='xmi61'/>
  <containedNode xmi:idref='xmi115'/>
</group>
```

In the SDMetrics metamodel for UML2, the model element "activitygroup" that stores partitions has a multi-valued attribute "nodes" (see Appendix A.2 "Metamodel for UML 2.x"). With the trigger

```
<trigger name="nodes" type="cattrval" src="containedNode"
         attr="xmi:idref" />
```

SDMetrics will pick up each "containedNode" child XML element, and store the values of their "xmi:idref" attributes in the multi-valued SDMetrics metamodel attribute "nodes" of the activity partition.

7.2.2.4 Trigger Type "gcattrval"

This trigger retrieves the value from an XMI attribute of a child of a child ("grand child") in the XMI tree.
In our example, the owner of the operation is specified in the child element Foundation.Core.Feature.owner. This child element has another child element of its own, whose attribute "xmi.idref" holds the reference to the operation owner. We retrieve its value with the trigger

```
<trigger name="context" type="gcattrval"
         src="Foundation.Core.Feature.owner" attr="xmi.idref"/>
```

The "src" attribute of the trigger specifies the child element name, the "attr" attribute of the trigger specifies the attribute of the grandchild element we want to access. Note that we do not specify the element name of the grandchild. For single-valued attributes such as "context", the trigger always accesses the first child of the specified child element, regardless of its name.

We can also use the trigger for multi-valued attributes. Revisiting the example of UML1.x activity partitions from Section 7.2.2.1 "Trigger Type "attrval"", an XMI1.2 exporter can also serialize swimlanes as follows:

© 2002-2013. All rights reserved.

```
<UML:Partition' name='mySwimlane' xmi.id='xmi12'>
 <UML:Partition.contents>
  <UML:ModelElement xmi.idref='xmi35'/>
  <UML:ModelElement xmi.idref='xmi61'/>
  <UML:ModelElement xmi.idref='xmi115'/>
 </UML:Partition.contents>
</UML:Partition>
```

The trigger

```
<trigger name="contents" type="gcattrval"
         src="UML:Partition.contents" attr="xmi.idref"/>
```

will visit each child XML element of the `UML:Partition.contents` element, and store the values of their "xmi.idref" attributes in the multi-valued attribute "contents" of SDMetrics' "partition" metamodel element.

The "gcattrval" trigger has one more function that is not related to extracting values from XML attributes. If a model element is defined as child element of the XML element specified by the trigger's "src" attribute, the trigger will store a cross-reference to that model element with the attribute for which the trigger is defined. Consider the following example of entry and exit actions of a state in UML1.4:

```
<UML:SimpleState xmi.id = 'id127' name = 'full'>
  <UML:State.entry>
    <UML:CallAction xmi.id = 'id128' name = 'anon' isAsynchronous = 'false'>
       ...
    </UML:CallAction>
    ... (more entry actions)
  </UML:State.entry>
  <UML:State.exit>
    <UML:CallAction xmi.id = 'id129' name = 'anon' isAsynchronous = 'false'>
       ...
    </UML:CallAction>
  </UML:State.entry>
  ...
</UML:SimpleState>
```

The entry and exit actions are defined as child elements of the `UML:State.entry` and `UML:State.exit` elements. The "state" element of SDMetrics' metamodel for UML1.x has two multi-valued cross-reference attributes "entryaction" and "exitaction". For these, we define the following triggers:

```
<xmitransformation modelelement="state" xmipattern="UML:SimpleState"
    recurse="true">
  <trigger name="entryaction" type="gcattrval" src="UML:State.entry"
      attr="xmi.idref" />
  <trigger name="exitaction" type="gcattrval" src="UML:State.exit"
      attr="xmi.idref" />
...
</xmitransformation>
```

With these triggers, SDMetrics will store cross-references to the model elements defined as children of the `UML:State.entry` and `UML:State.exit` XML elements in the "entryaction" and

© 2002-2013. All rights reserved.

"exitaction" attributes of SDMetrics model element "state", respectively. Note that the values of the "attr" attributes of the triggers play no role in this.

7.2.2.5 Trigger Type "constant"

This trigger instructs SDMetrics to insert a constant value for an attribute. For example, the SDMetrics metamodel differentiates the types of states (simple, initial, final, etc.) in a state diagram with an attribute "kind" of the metamodel element "state". Consider the following excerpt of the XMI transformations for UML simple states and final states:

```
<xmitransformation modelelement="state"
   xmipattern="Behavioral_Elements.State_Machines.SimpleState">
   ...
   <trigger name="kind" type="constant" attr="simple"/>
</xmitransformation>

<xmitransformation modelelement="state"
   xmipattern="Behavioral_Elements.State_Machines.FinalState">
   ...
   <trigger name="kind" type="constant" attr="final"/>
</xmitransformation>
```

The "attr" attribute specifies the value to be inserted for the metamodel attribute ("simple" for SimpleState, "final" for FinalState).

7.2.2.6 Trigger Type "ignore"

This trigger leaves the attribute value empty. This is useful if you wish to override the triggers of an inherited attribute that is not meaningful for a particular metamodel element (Section 7.2.3.4 "Inherited Attributes and Triggers" describes the trigger inheritance mechanism). For example, every UML design has a root element "Model" that provides the context for all other design elements in the model. The "Model" element itself, however, has no owner or context. Hence, the XMI transformation for "Model" leaves the value for the required attribute "context" empty:

```
<xmitransformation modelelement="model" xmipattern="Model_Management.Model">
   ...
   <trigger name="context" type="ignore" />
</xmitransformation>
```

A trigger of type "ignore" requires no further attributes.

7.2.2.7 Trigger Type "xmi2assoc"

The "xmi2assoc" trigger is specifically defined to process serializations of associations and aggregations in XMI 2.x.

Composite aggregations in XMI 2.x

In XMI 2.x, the elements that play the parts in composite aggregations between metamodel elements are serialized in such a way that their types are not specified explicitly in the XMI file. For example, the type "Class" in the UML2 metamodel has a composite 1:n aggregation to the type "Property".

© 2002-2013. All rights reserved.

The role name for the parts is "ownedAttribute". A class named "aClass" with two attributes "attr1" and "attr2" is serialized as follows:

```
<uml:Model xmi:version="2.0" xmi:id="xmi.1" name='aModel'
  ...
  <ownedMember xmi:type='uml:Class' xmi:id='xmi.42' name='aClass'>
    <ownedAttribute xmi:id='xmi.43' name='attr1' type='xmi.2001'/>
    <ownedAttribute xmi:id='xmi.44' name='attr2' type='xmi.1138'/>
  </ownedMember>
  ...
</UML:Model>
```

The name of the XML elements defining the attributes "attr1" and "attr2" is the role name "ownedAttribute" of the aggregation in the metamodel. We need to provide SDMetrics with the information that "ownedAttribute" associates classes with elements of type "Property". This can be done with the "xmi2assoc" trigger as follows:

```
<xmitransformation modelelement="class" xmipattern="uml:Class">
   <trigger name="properties" type="xmi2assoc" attr="ownedAttribute"
            src="uml:Property"/>
   ...
</xmitransformation>

<xmitransformation modelelement="property" xmipattern="uml:Property">
  ...
</xmitransformation>
```

The attribute "attr" of the trigger specifies the role name of the aggregation that is also the name of the XML element defining the part. The attribute "src" specifies the XMI pattern of the XMI transformation that should be used to process the part definition. The attribute "name" specifies the name of the (multi-valued) SDMetrics metamodel cross-reference attribute that stores the reference(s) to the part(s).

If a part is a subtype of the associated class, the XMI exporter must indicate the type of the part with the "xmi:type" attribute. If such a type is specified, SDMetrics will of course use the XMI transformations defined for that type.

The "src" attribute is optional if the type of the parts in the UML2 metamodel is abstract. In that case, the concrete type of the part must always be indicated in the XMI file with the xmi:type attribute. In the example above, the UML2 metamodel class "Model" has a composite aggregation with the abstract class "PackageableElement"; the role name of the parts is "ownedMember". The following XMI transformation collects all members of the model:

```
<xmitransformation modelelement="model" xmipattern="uml:Model">
   <trigger name="members" type="xmi2assoc" attr="ownedMember" />
</xmitransformation>
```

Associations in XMI 2.x

The "xmi2assoc" trigger can also be used to capture plain associations that are not aggregations. For example, the UML2 metamodel class "InstanceSpecification" has an association with class "Classifier", the role name is "classifier". To serialize an instance specification with several classifiers, an XMI exporter has two options:

© 2002-2013. All rights reserved.

```
<!-- first option: -->
<ownedMember xmi:type='uml:InstanceSpecification' xmi:id='xmi.42'
             name='myInstance'>
  <classifier xmi:idref='xmi.43'/>
  <classifier xmi:idref='xmi.44'/>
</ownedMember>

<!-- second option: -->
<ownedMember xmi:type='uml:InstanceSpecification' xmi:id='xmi.42'
             name='myInstance' classifier='xmi.43 xmi.44'/>
```

The following XMI transformation will handle both options with one trigger:

```
<xmitransformation modelelement="instancespec"
      xmipattern="uml:InstanceSpecification">
   <trigger name="classifiers" type="xmi2assoc" attr="classifier" />
</xmitransformation>
```

Here we assume that the SDMetrics metamodel type "instancespec" has a multi-valued cross-reference attribute "classifiers" that stores the references to the classifiers of the instance specification.

7.2.3 Tips on Writing XMI Transformations

Now that we have explained XMI transformations and triggers, we conclude with some useful tips for writing XMI transformations.

7.2.3.1 The "linkbackattr" Trigger Attribute

Whenever a cross-reference is established from a model element *e1* to a model element *e2* via a cross-reference attribute, you can optionally store a cross-reference back from *e2* to *e1*.

If we take again the example of states in UML1.x activity partitions, we could add a cross-reference attribute "inPartition" to metamodel element "state", and populate that attribute with information in which partition a state lies as follows:

```
<xmitransformation modelelement="activitygraph" xmipattern="UML:ActivityGraph">
<trigger name="contents" type="attrval" attr="contents"
         linkbackattr="inPartition"/>
<trigger name="contents" type="gcattrval" src="UML:Partition.contents"
         attr="xmi.idref" linkbackattr="inPartition" />
</xmitransformation>
```

The semantic of the "linkbackattr" is as follows. When a trigger retrieves information for a cross-reference attribute (such as "contents" for partitions), we check if the referenced element has a cross-reference attribute of the name specified by the "linkbackattr" (i.e., attribute "inPartition" that we defined for states). If so, we set the value of that attribute to point back to the referencing model element (the partition containing the state).

© 2002-2013. All rights reserved.

7.2.3.2 Multiple Triggers per Attribute

It is possible to specify multiple triggers for one attribute or relation. For example, in XMI1.1 and XMI1.2, most model element information can be specified either via XML attributes or via XML elements. For instance, the name of a class can be specified like this:

```
<UML:Class xmi.id="cls1" name="Rectangle" />
```

or like this:

```
<UML:Class xmi.id="cls1">
  <UML:ModelElement.name>Rectangle</UML:ModelElement.name>
</UML:Class>
```

Therefore, the XMI transformations for these XMI versions have two triggers for most attributes, one trigger for each option. In the above example, the triggers to retrieve the class names are:

```
<xmitransformation modelelement="class" xmipattern="UML:Class">
   <trigger name="name" type="attrval" attr="name" />
   <trigger name="name" type="ctext" src="UML:ModelElement.name" />
   ...
</xmitransformation>
```

Multiple triggers are also useful to support modeling tools that deviate in minor ways from the UML standards. For example, some modeling tools denote the parent/child elements in a UML1.x generalization by "supertype" and "subtype", instead of the proper "parent" and "child". In many cases, you can simply add a trigger for the nonstandard way of representing the attribute, and a single XMI transformation file is thus capable of handling various XMI exporters.

7.2.3.3 "Context" Attribute Defaults

Some XMI exporters do not explicitly specify the owner of an element using the appropriate XMI element (e.g., a link from an operation back to the class to which it belongs). Instead, ownership is implied: the owner is the parent element in the XMI tree. Therefore, when the trigger(s) for attribute "context" retrieve no information for a model element, SDMetrics inserts the parent element in the XMI tree as the owner of the element.

7.2.3.4 Inherited Attributes and Triggers

A metamodel element can inherit attributes from a parent metamodel element. When defining XMI transformations, the child metamodel element automatically inherits the triggers that are defined for the parent's attributes. Therefore, you do not need to define triggers for inherited attributes, but only for the additional attributes of a metamodel element.

If you define a trigger for an inherited attribute, all inherited triggers for that attribute are overridden, and only the newly defined trigger(s) for that attribute are used for the child element.

Special care needs to be taken if there are multiple `xmitransformation`s defined for the parent metamodel element. In that case, one `xmitransformation` is chosen arbitrarily to determine the inherited triggers for the child metamodel element. Therefore, you have to make sure that any of the

© 2002-2013. All rights reserved.

possible `xmitransformations` is suitable to provide the inherited triggers for the child. Otherwise, explicitly provide the proper definitions of all triggers for the child metamodel element.

7.2.3.5 Optional XMI ID

When SDMetrics reads an XMI file, a UML model element is recognized only if its XML element in the XMI source file has an XMI ID, specified by the XML attribute "xmi.id" or "xmi:id". This should be the normal behavior. However, some XMI exporters do not endow all UML model elements with XMI IDs. For such cases, you can specify that an `xmitransformation` should be applied even if the XML element has no XMI ID:

```
<xmitransformation modelelement="taggedvalue" xmipattern="UML:TaggedValue"
      requirexmiid="false">
   <trigger name="tag" ...
   ...
</xmitransformation>
```

If attribute "requirexmiid" is set to "false", SDMetrics recognizes XML elements which have no XMI ID, as long as the XML element specifies no XMI IDREF cross reference (attribute "xmi.idref" or "xmi:idref") either. By default, "requirexmiid" is "true".

7.2.3.6 Conditional XMI Transformations

The XMI transformations we have discussed so far were unconditional: they are used whenever an XMI element matching the transformation's XMI pattern is encountered.

It is possible to make the use of an XMI transformation conditional. You may specify a condition for the attributes of the XMI element that must be fulfilled for a XMI transformation to become effective. With this feature, you can filter certain XMI elements, or you can conditionally map one XMI element onto different metamodel element types, depending on the values of its attributes.

For example, the MagicDraw™ UML modeling tool uses a proprietary XMI extension to encode diagram information as follows:

```
<mdElement elementClass = 'DiagramData' xmi.id = 'ID2546'>
  <parentID xmi.idref = 'ID0002' />
  <type>Class Diagram</type>
  <mdElement elementClass = 'DiagramView' xmi.id = 'ID2547'>
    <elementID xmi.idref = 'ID2546' />
    <zoomFactor xmi.value = '1.0' />
    <mdOwnedViews>
      <mdElement elementClass = 'ClassView' xmi.id = 'ID2548'>
       <elementID xmi.idref = 'ID00dd' />
       <geometry>165, 16, 358, 60</geometry>
      </mdElement>
      <mdElement elementClass = 'ClassView' xmi.id = 'ID2549'>
       <elementID xmi.idref = 'ID15f2' />
       <geometry>40, 170, 600, 111</geometry>
      </mdElement>
      ...
```

The XML element `mdElement` represents both diagrams and diagram elements, indicated by the value of attribute `elementClass`. The value "DiagramData" denotes a diagram (and we can then

© 2002-2013. All rights reserved.

extract additional information on diagram type and owner). A value other than "DiagramData" denotes a diagram element (which contains a cross-reference to the UML model element it represents). To define diagram-specific metrics, we must be able distinguish diagrams from diagram elements:

```
<xmitransformation modelelement="diagram" xmipattern="mdElement"
                   condition="elementClass='DiagramData'" recurse="true">
  <trigger name="style" type="ctext" src="type" />
  <trigger name="context" type="cattrval" src="parentID" attr="xmi.idref" />
</xmitransformation>

<xmitransformation modelelement="diagramelement" xmipattern="mdElement"
                   condition="elementClass!='DiagramData'">
  <trigger name="element" type="cattrval" src="elementID" attr="xmi.idref" />
</xmitransformation>
```

The first `xmitransformation` for model element "diagram" specifies a condition that the "mdElement" XML element must have an attribute elementClass of value "DiagramData". The second `xmitransformation` for model element "diagramelement" picks up all mdElement instances with an elementClass different from "DiagramData". As a result, "mdElement" DiagramData elements are stored as "diagram" metamodel elements, all others are stored as instances of "diagramelement".

SDMetrics uses the following strategy when searching a suitable XMI transformation for an XML element in the XMI source file:

- First, the conditional XMI transformations are tested against the current XML element. If a matching XMI transformation is found, it is used to process the XML element. The order in which the conditional XMI transformations are tested is undefined. Therefore, you should choose the conditions so that at most one condition can evaluate to true for any given XML element.
- If no conditional XMI transformation matches the current XML element, an unconditional XMI transformation is used, if one exists. Thus, the unconditional XMI transformation can serve as default, "catch-all" transformation. If no unconditional XMI transformation exists, the XML element is ignored.

The condition you specify is a Boolean expression using operands

- & (and),
- | (or),
- ! (not),
- = (equals),
- != (not equals),

as described in Section 8.5.4 "Condition Expressions". Identifiers (such as elementClass, see Section 8.5.1.2 "Identifiers") refer to XML attributes. Values of XML attributes can be compared to string constants, which are enclosed in single quotes (see Section 8.5.1.1 "Constants").

© 2002-2013. All rights reserved.

8 Defining Custom Design Metrics and Rules

With SDMetrics, you are not restricted to a fixed set of metrics. Using the *SDMetricsML*, the SDMetrics Markup Language, you can define and calculate your own metrics and design rules.

The SDMetricsML is based on the XML. The metric definition file is an XML file of the following format (for a formal definition of the file format, see Appendix E: "Project File Format Definitions"):

```
<sdmetrics version="2.3" ruleexemption="taggedvalue"
  exemptiontag="tagname" >
  <metric name="met1" ...>
   <'metric definition' ...>
  </metric>
  <set ... name="set1" ...>
   <'set definition' ...>
  </set>
  <metric name="met2" ...>
   <'metric definition' ... >
  </metric>
  <matrix name="matrix1" ...>
   <'matrix definition' ... >
  </matrix>
  <rule name="rule1" ...>
   <'rule definition' ... >
  </rule>
  <wordlist name="list1" ...>
   <'wordlist definition' ... >
  </wordlist>
  <reference tag="ref1">
   bibliographic citation #1
  </reference>
  <term name="term1">
   definition of term
  </term>

  ...
</sdmetrics>
```

The "version" attribute (required) of the root element `sdmetrics` indicates the version number of the oldest version of SDMetrics with which the metric definition file can be used. The remaining two attributes instruct the design rule checker how to access tagged values, this will be discussed later (Section 8.3.6 "Exempting Approved Rule Violations").

The file contains a list of definitions of metrics, as well as sets (sets of UML elements, sets of values), design rules and word lists (see Section 4.7 "The View 'Rule Checker'"), relation matrices (see Section 4.10 "The View 'Relation Matrices'"), literature reference and glossary terms. The following sections describe the definition of metrics, sets, rules, and relation matrices in detail.

© 2002-2013. All rights reserved.

8.1 Definition of Metrics

A metric is defined with an XML element as follows:

```
<metric name="metricname" domain="metricdomain" category="metriccategory"
  internal="true/false" inheritable="true/false">
  <description>Description of the metric.</description>
  <'metric definition' ...>
</metric>
```

The attributes in the opening "metric" tag are as follows:

- `name` (required): the name of the metric. The metric names serve as column names in the output tables, and are also used to reference the metrics.
- `domain` (required): the type of element for which the metric is defined (e.g., package, class, operation).
- `category` (optional): describes the structural property the metric measures, e.g., "size", "export coupling", "cohesion", etc. There is no preconceived classification of categories in SDMetrics, you can define any categories you deem fit. This attribute only serves documentation purposes and does not impact the calculation of the metric in any way.
- `internal` (optional): indicates if the metric is internal. An internal metric is not shown in the output tables. It is rather a "helper" metric that is used to define some other metrics, but by itself is probably not useful. Set the attribute value to "true" to mark a metric as internal. When omitted, the attribute defaults to "false".
- `inheritable` (optional): indicates if the metric should also be defined for all subtypes of the "domain" element type. When you set the attribute value to "true", all direct and indirect subtypes of the domain "inherit" the metric definition, and the metric will be calculated for all subtypes, too. When omitted or set to "false", the metric definition is not passed on to subtypes.

Note that metrics for one domain must have unique names (i.e., you cannot define two metrics named "NumOps" for classes). You can have metrics of the same name for different domains (e.g., a metric "NumOps" for classes, and a metric "NumOps" for interfaces).

Following the metric tag is an optional description of the metric, which will be shown in the measurement catalog (see Section 4.13 "The View 'Catalog'"). Section 8.6 "Writing Descriptions" explains how to write metric descriptions.

Following the description is an XML element that defines the calculation procedure for the metric. We describe each calculation procedure in detail in the following subsections.

8.1.1 Projection

The projection is your primary workhorse. Most of the metrics you will want to define can be expressed as projections. With twelve optional attributes that can be combined in multiple ways, the projection offers you a lot of flexibility to define metrics. We'll start the description of attributes with simple ones, and then advance towards more intricate attributes.

© 2002-2013. All rights reserved.

8.1.1.1 Attribute "relation"

In its simplest form, the projection gives you, for a certain element, a count of all elements having a certain relationship with that element. The attribute "relation" specifies the relationship. Here's an example of a simple projection:

```
<metric name="NumElements" domain="class">
  <description>The number of elements in the class.</description>
  <projection relation="context" />
</metric>
```

The metric NumElements is defined for elements of type "class". It counts, for a given class, all elements which have a reference attribute "context" that points to that class. Attribute "context" specifies the owner of a model element. So this metric counts all model elements of which the class is the owner, that is, the number of elements in the class. Those elements can be of any type, e.g., operations, attributes, or other classes (inner classes).

8.1.1.2 Attribute "relset"

Instead of relations, you can also feed element sets into the projection. For example, packages have a multi-valued attribute "ownedmembers" in the UML2 metamodel. We can count the number of elements in that set as follows:

```
<metric name="NumMembers" domain="package">
  <description>The number of owned members of the package.</description>
  <projection relset="ownedmembers" />
</metric>
```

Sources for sets can be

- multi-valued attributes of model elements,
- user-defined sets. These will be discussed in Section 8.2 "Definition of Sets".

Attribute "relset" defines a set expression (see Section 8.5.3 "Set Expressions") which is evaluated for the current element. The metric then counts the number of elements in that set.

For a projection, exactly one of the attributes "relation" or "relset" must be specified.

8.1.1.3 Filter Attribute "target"

To count only elements of a certain type in a relation or set, you can specify the optional "target" attribute:

```
<metric name="NumOps" domain="class">
  <description>The number of operations in the class.</description>
  <projection relation="context" target="operation" />
</metric>
```

This only counts elements of type "operation" whose "context" is the given class, i.e., the number of operations of the class.

© 2002-2013. All rights reserved.

Filtering for several types

If you want to count elements of several types, specify the additional types separated by a "|". For example, to count the classes, interfaces, and data types in a package, we write:

```
<metric name="NumEl" domain="package">
  <projection relation="context" target="class|interface|datatype" />
</metric>
```

Filtering for subtypes

In SDMetrics' metamodel for UML2, type "class" is the parent type of several subtypes such as "usecase", "actor", "component". The target filter `target="class"`, however, only accepts elements of type "class". Elements whose type is a subtype of type "class" will not be accepted. Thus, the following metric "NumCls" only counts the classes in the package, not including actors, use cases, and so forth:

```
<metric name="NumCls" domain="package">
  <projection relation="context" target="class" />
</metric>
```

To also accept elements of direct or indirect subtypes of the type, put a "+" prefix in front of the type name:

```
<metric name="NumClsTypeEl" domain="package">
  <projection relation="context" target="+class" />
</metric>
```

This will count all classes, actors, usecases, components, and other elements in the package whose type is a direct or indirect subtype of "class".

You can combine subtype filtering with filtering for several types. For example, `target="+class|interface|+datatype"` will accept classes and datatypes as well as their subtypes, and interfaces, but not any subtypes of "interface".

8.1.1.4 Filter Attributes "element" and "eltype"

In the UML, relationships such as generalizations or dependencies are represented by model elements of their own, which contain references to the elements that participate in the relationship. Figure 34 illustrates this situation:

© 2002-2013. All rights reserved.

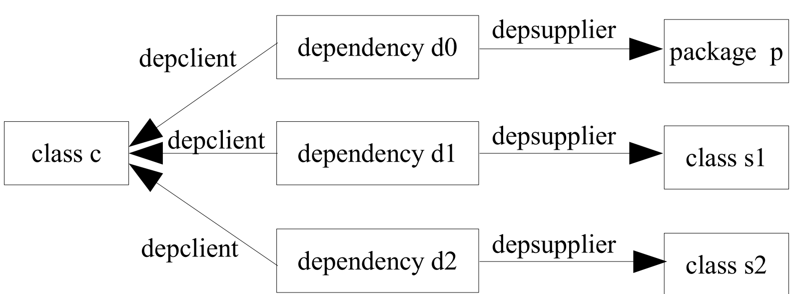


Figure 34: Example dependency links

Class c is the client in three dependency relationships with three suppliers: a package and two other classes. A UML 1.x dependency element links the client and supplier via its cross-reference attributes "depclient" and "depsupplier" (cf. metamodel element dependency in Appendix A.1 "Metamodel for UML 1.3/1.4"). We can specify a projection for the "depclient" relation for dependency elements:

```
<metric name="Dependencies" domain="class">
  <description>The number of dependencies in which the class
  participates as client.</description>
  <projection relation="depclient" target="dependency" />
</metric>
```

This projection retrieves the elements of type "dependency" where the given class is the client. For class c, this would get us d0, d1, and d2. So far so good, however, we are probably more interested in actual supplier elements. This is where attribute "element" comes in:

```
<metric name="SupplierElements" domain="class">
  <description>The supplier elements of which the class
  is a client.</description>
  <projection relation="depclient" target="dependency"
    element="depsupplier" />
</metric>
```

By specifying the attribute "element", the projection does not access the dependency element, but the element referenced via the "depsupplier" relation specified by the "element" attribute. In Figure 34 above, this gives us p, s1, and s2. In other words, the suppliers we want. To filter for suppliers of a certain type, we specify the additional attribute "eltype" that indicates the type of elements we are interested in:

```
<metric name="SupplierClasses" domain="class">
  <description>The supplier classes of which the class
  is a client in a dependency.</description>
  <projection relation="depclient" target="dependency"
    element="depsupplier" eltype="class" />
</metric>
```

This projection now only returns supplier classes (s1 and s2 for class c in the above example).

© 2002-2013. All rights reserved.

Note that the value of the "element" attribute is a metric expression. In addition to cross-reference attributes, you can specify arbitrary metric expressions that return model elements (see Section 8.5.2.2 "Special Operators").

As with the "target" attribute, you can filter for several element types and/or subtypes with the "eltype" attribute. Just separate the additional element types with a "|", and precede types whose subtypes should also be admitted with a "+", for example:
`eltype="+class|interface|+datatype"`.

8.1.1.5 Filter Attributes "condition" and "targetcondition"

Besides filtering elements of a certain type, you can filter for elements that satisfy a certain condition. Assume we have defined a multi-valued attribute 'ownedoperations' for classes, which contains the operations of the class.

```
<metric name="NumPublicOps" domain="class">
  <description>The number of public operations in the class.</description>
  <projection relset="ownedoperations" condition="visibility='public'" />
</metric>
```

The condition expression "visibility='public'" is evaluated for each element in the set, only operations that fulfill this condition are counted. The admissible operations for condition expressions are described in Section 8.5.4 "Condition Expressions".

If you combine the "condition" attribute with the "element" and "eltype" attributes, the expression will be evaluated for the model elements returned by those attributes, e.g.,

```
<metric name="AbstractDep" domain="package">
  <description>
     The number of abstract classes the class depends on.
  </description>
  <projection relation="depclient" target="dependency" element="depsupplier"
    eltype="class" condition="abstract='true'"/>
</metric>
```

With attribute "targetcondition" you can specify a condition that is always evaluated for the element returned by the "target" attribute, even if the "element" and "eltype" attributes are used. The following example assumes that "stereotypename" is an attribute or metric that yields the name of the stereotype of a model element:

```
<metric name="Realize" domain="package">
  <description>The number of classes the class realizes.</description>
  <projection relation="depclient" target="dependency"
    targetcondition="stereotypename='realize'" element="depsupplier"
    eltype="class" />
</metric>
```

This metric only counts dependencies with stereotype <<realize>> that have a class as supplier.

© 2002-2013. All rights reserved.

8.1.1.6 Filter Attribute "scope"

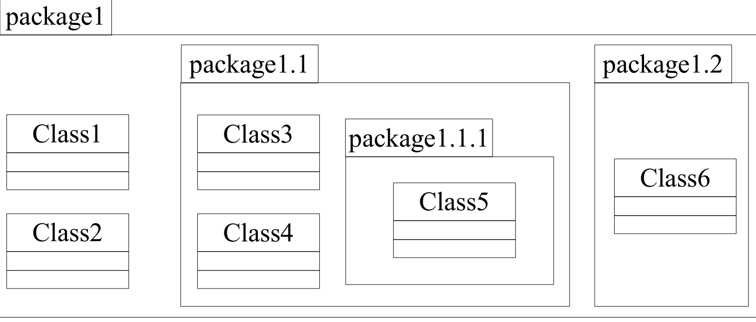
For a detailed analysis of the modularity of a system it is interesting to take the scope of elements into account. For instance, any sufficiently large design will usually be organized in a hierarchy of packages. Dependency relationships or associations between classes in the same package would appear less critical than links between classes from different packages.

With the "scope" attribute, you can filter for elements that fulfill a condition for the scope:

```
<metric name="SameScope_SupplierClasses" domain="class">
  <description>The number of supplier classes of a client
  class defined in the same scope.</description>
  <projection relation="depclient" target="dependency"
    element="depsupplier" eltype="class" scope="same"/>
</metric>
```

For a given client class, this projection only accesses supplier classes that are in the same scope as the current class. You can specify one of the following values for the "scope" attribute:

- same: the projected element is in the exact same scope. In Figure 35 below, Class1 and Class2 are in the same scope. So are Class3 and Class4.
- other: the projected element is not in the same scope. In Figure 35 below, this applies to all class pairs except Class1&Class2 and Class3&Class4.
- higher: the projected element is higher in the scope hierarchy. E.g., for Class3, only Class1 and Class2 are higher in the scope hierarchy. The same holds for Class4 and Class6. For Class 5, all classes except Class6 are higher in the hierarchy. Class6 is on a different branch, hence not higher or lower than Class5.
- lower: the projected element is lower in the scope hierarchy. For Class3, only Class5 is lower in the hierarchy. For Class1, all classes except Class2 are lower.
- nothigher: the inverse of "higher". For Class3, the following classes are not higher in the scope hierarchy: Class4 (because it's in the same scope), Class5 (because it's lower) and Class6 (because it's on a different branch).
- notlower: the inverse of "lower". For Class3, all classes but Class5 are not lower in the scope hierarchy.
- sameorhigher: the projected element is in the same scope or higher scope. E.g., for class4, that would be Class1, Class2, and Class3.
- sameorlower: the projected element is in the same scope or lower scope. Again, for Class4, that would be Class3 and Class5.
- samebranch: same or higher or lower scope. For Class1 and Class2, this includes all classes. For Class3, Class4, and Class5, this includes all classes but Class6.
- notsamebranch: the inverse of "samebranch". For Class6, this would be classes Class3, Class4, and Class5. All other class pairs are on the same branch.



© 2002-2013. All rights reserved.

Figure 35: Example package hierarchy

In the above cases, we compared the scope of two elements. There are four more values we can specify for scope: idem/notidem, and containedin/notcontainedin. These test if the element for which the metric is calculated is/is not the scope of the projected element.

For example, we want to define a package coupling metric that counts the number of stimuli sent to instances of classes in the package, from instances of classes outside the package (i.e., stimuli that cross the package boundary).

We assume we have already defined for packages the set "StimRecvSet" of stimuli received by object instances of classes in the package. We need to check, for each stimulus in that set, if type of the sender is a class that is not defined in the current package:

```
<metric name="StimRecv_ex" domain="package">
   <description>The number of stimuli received by object instances
                 of classes outside the package.</description>
   <projection relset="StimRecvSet" target="stimulus"
     element="stimsender.objtype.context" scope="notidem" />
</metric>
```

The `element` attribute accesses the package of the class of the instance that sent the stimuli. The `scope` attribute value "notidem" only accepts elements that are not identical to the current element for which the metric is calculated. So we only count stimuli where the context of the class of the sender is not the current package - i.e. stimuli sent from outside the package.

To further illustrate the new scope comparisons, consider again the example package hierarchy in Figure 35. The additional values for attribute scope are:

- idem: only return elements identical to the element for which the metric/set is calculated. Using "idem" as scope in metric "StimRecv_ex" above would count the stimuli sent between instances of classes of the package. For package1.1 in Figure 35, this would be stimuli sent within or between instances of Class3 and Class4.
- notidem: only return elements not identical to the element for which the metric/set is calculated. For package1.1, these are stimuli sent from Class1, Class2, Class5, and Class6.
- containedin: only returns elements that are identical to or contained in the element for which the metric/set is calculated. Using "containedin" as scope in metric "StimRecv_ex" above would count stimuli sent to instances of classes of the package from instances of classes of the package, or any of its subpackages. For package1.1, these are Class3, Class4, and Class5.
- notcontainedin: only return elements that are not contained in the element for which the metric/set is calculated. Using "notcontainedin" as scope in metric "StimRecv_ex" above would count stimuli sent to instances of classes of the package from instances of classes outside the package and any of its subpackages. For package1.1, these are stimuli sent from Class1, Class2, and Class6.

Note that in the definition of metric "StimRecv_ex" above, we could not use the scope value "other" or any of the values explained previously. To see why, take again the example of package1.1: the

© 2002-2013. All rights reserved.

scope of package1.1 is package1, and it is this package1 against which all comparisons are made! For example, the elements that are in the same scope as package1.1 are Class1 and Class2.

8.1.1.7 Attributes "sum" and "stat"

You are not restricted to just counting elements in a relation. You can evaluate a metric expression (see Section 8.5.2 "Metric Expressions") for the elements in a relation, and add up the values, or return the maximum or minimum value. For example:

```
<metric name="NumOps" domain="package">
  <projection relation="context" target="class" sum="NumOps" />
</metric>
```

The expression "NumOps" is evaluated for each class in the package. This metric returns the number of operations in the classes in a package.

To return the maximum or minimum of the values evaluated for the sum expression, specify the additional attribute "stat". The stat attribute can take values "max", "min" or "sum" (default value). For example:

```
<metric name="MaxNumOps" domain="package">
  <description>The maximum number of operations in a class
    of the package.</description>
  <projection relation="context" target="class"
              sum="NumOps" stat="max" />
</metric>
```

When you combine "sum" with the "element" and "eltype" attributes, the sum will be taken over the elements returned by those attributes. For example:

```
<metric name="SupplierOperations" domain="class">
  <description>The number of public operations in the supplier
  classes of a client class.</description>
  <projection relation="depclient" target="dependency"
    element="depsupplier" eltype="class" sum="NumPubOps"/>
</metric>
```

8.1.1.8 Attribute "recurse"

With the "recurse" attribute, you can define metrics that take the transitive closure of relations or sets into account. The recurse attribute takes the values "true" or "false" (default).

Setting the value of "recurse" to true changes the meaning of a metric *m* for elements of type *t* as follows: in addition to processing the projection attributes as usual, apply metric *m* to all "compatible" elements in the projection, and add the result to the return value of the metric. An element is "compatible" if its type is *t*, or if its type is a subtype or supertype of *t* and metric *m* is defined for the type.

```
<metric name="NumCls_tc" domain="package">
  <description>The number of classes in a package,
  its subpackages, and so on.</description>
  <projection relation="context" target="class" recurse="true"/>
</metric>
```

© 2002-2013. All rights reserved.

Without the "recurse" attribute being set to true, the above metric would just count the number of classes in a package. With recurse set to true, metric NumCls_tc is in addition recursively applied to all subpackages that are in the context of a package, and the values added up. As result, you not only obtain the number of classes in the package, but also in its subpackages, sub-subpackages, and so on. In Figure 35, the metric NumCls_tc yields the following values:

Package	NumCls_tc
package1	6
package1.1	3
package1.1.1	1
package1.2	1

Table 4: Example package metrics

As another example, the number of descendents of a class can be defined as follows:

```
<metric name="NumDesc" domain="class">
  <description>The number of descendents of a class.</description>
  <projection relation="genparent" target="generalization"
              element="genchild" eltype="class" recurse="true"/>
</metric>
```

Without the "recurse" attribute, this metric would count the number of children of a class. With "recurse" being set to true, the metric NumDesc is recursively evaluated for each child class, and added to the total.

Care must be taken when the "recurse" attribute is combined with the filter attributes (target, targetcondition, eltype, element, condition, scope). The filter attributes are NOT evaluated for the selection of elements on which to apply the metric recursively. The metric is always recursively applied to all elements in the unfiltered projection that are compatible with the element for which the metric is calculated. Consider this example:

```
<metric name="NumDesc_SameScope" domain="class">
  <projection relation="genparent" target="generalization"
              element="genchild" eltype="class"
              scope="same" recurse="true"/>
</metric>
```

This will not work. While only child classes in the same scope are counted, the metric is recursively applied to all child classes (regardless of their scope), and the number of their "same scope descendents" is added to the total. To define the intended metric, you would first define the set (see Section 8.5.3 "Set Expressions") of descendents of the class, and then define a projection to filter the classes with the desired scope in that set.

8.1.1.9 Attribute "nesting"

Like the "recurse" attribute, the "nesting" attribute takes the transitive closure of relations or sets into account. However, it has a very different meaning; with the "nesting" attribute, you determine the length of the longest path in the transitive graph of a relation.

© 2002-2013. All rights reserved.

The "nesting" attribute takes the values "true" or "false" (default). When "nesting" is set to true for a project metric *m* for type *t*, the metric is recursively applied to all "compatible" elements in the projection. An element is "compatible" if its type is *t*, or if its type is a subtype or supertype of *t* and metric *m* is also defined for the type. The maximum value of all values returned, increased by one, is the value of the metric. Example:

```
<metric name="DIT" domain="class">
  <description>Depth of the class in the inheritance tree.</description>
  <projection relation="genchild" target="generalization"
              element="genparent" eltype="class" nesting="true"/>
</metric>
```

Without the `nesting="true"` attribute, the above metric would simply count the number of parent classes of a class. With "nesting" set to true, the meaning of the metric changes as follows: calculate the DIT for all parents of the class, take the maximum DIT over all parent classes, increase it by one. As a result, this metric calculates the longest path from the class to the root in the inheritance graph.

The "nesting" attribute cannot be combined with the "sum" attribute or the "recurse" attribute. It can, however, be combined with all other attributes, including the filter attributes (target, targetcondition, element, eltype, scope, condition). Recursive calls are not applied to elements that do not fulfill the filter criteria.

8.1.1.10 Summary of Projection Attributes

The admissible attributes for a projection definition are

- `relation`: operate on elements of 'incoming' relations
- `relset`: operate on elements of 'outgoing' relations or pre-calculated sets
- `target`: element type(s) of related elements
- `element`: for indirect relationships, relation link to follow
- `eltype`: for indirect relationships, type(s) of related element
- `condition` and `targetcondition`: discard elements which do not fulfill the specified expression
- `scope` discard elements which do not fulfill the scope condition
- `sum`: expression to evaluate and add up for each related element
- `stat`: specify "min" or "max" to obtain the minimum or maximum value of all "sum" metric expressions evaluated.
- `recurse`: calculate and add up same metric for compatible elements (default false)
- `nesting`: calculate nesting level for specified relation (default false)

The algorithm to evaluate a projection is presented here in pseudo-code.

```
function projection(modelelement me) is:
  // init result value
  result := 0
  // retrieve elements specified by attribute "relation"
  if(relation specified) rel := all elements in relation with me
  else rel := result of set expression applied to me
  while(still elements in rel):
    el := next element in rel
```

© 2002-2013. All rights reserved.

```
    // filter "target" attribute and condition
    if(target specified and el is not one of the types) goto invalid
    if(targetcondition specified and el does not fulfill it) goto invalid
    // handle indirect relations (attributes "element" and "eltype")
    if(element specified)
       el := el.element
       if(eltype specified and el is not one of the types) goto invalid
    endif
    // filter "condition" attribute
    if(condition specified and el does not fulfill it) goto invalid
    // filter "scope" attribute
    if(scope specified and el does not fulfill scope) goto invalid
    if(nesting)
      if(el is compatible with me)
        result:=max(result, projection(el)+1)
        continue // next iteration of the while loop
    else
      if(sum specified)
        if(stat="max")
          result:=max(result,sum(el))
        else if(stat="min")
          result:=min(result,sum(el))
        else
          result:=result+sum(el)  // default no attribute "stat"
      else
        result:=result+1
invalid:
    if(recurse)
      if(el is compatible with me)
        result:=result+projection(el)
  endwhile
  return result
endfunction
```

8.1.2 Compound Metrics

A compound metric is a metric that is defined in terms of other metrics. As an example for a compound metric, assume we have two metrics defined for packages

- "NumCls", the number of classes in the package,
- "NumOps", the number operations in the classes in the package.

We can then define a metric for the average number of operations in the classes of the package as follows:

© 2002-2013. All rights reserved.

```
<metric name="AvgOps" domain="package">
  <description>The average number of operations in the
  classes of the package.</description>
  <compoundmetric term="NumOps/NumCls" fallback="0"/>
</metric>
```

The `term` attribute defines the metric expression to be evaluated, which is the return value of the metric. The optional `fallback` expression is used if the metric expression returns a "NaN" (not a number) or infinity. In that case, the `fallback` expression is evaluated and returned as metric result. This is useful to catch, e.g., divisions by zero and still have a well-defined metric result.

A second version of the compound metric allows the conditional evaluation of expressions:

```
<metric name="accessedOps" domain="class">
  <description>The number of public operations of the class,
    if it has any clients, or 0 else.</description>
  <compoundmetric condition="NumClients>0"
                  term="NumPubOps" alt="0" />
</metric>
```

For the conditional version of the compound metric, a conditional expression (`condition`) and an alternative metric expression (`alt`) is specified in addition to the `term` attribute. Their meaning is as follows. First, the `condition` expression is evaluated. If it evaluates to "true", the `term` expression is evaluated and returned as value of the metric (value 1 is returned if no `term` expression is specified). If the `condition` expression is "false", the `alt` metric expression is evaluated and returned as result of the metric (value 0 is returned if no `alt` expression is specified).

8.1.3 Attribute Value

You can define a metric which returns the value of a SDMetrics metamodel attribute of an element or a related element. This is useful if you want attribute values to appear in the output table, and it can help simplify the definition of other metrics. For example, to show the "visibility" attribute for elements of type "operation" in the output tables:

```
<metric name="visibility" domain="class">
  <description>The visibility of the operation.</description>
  <attributevalue attr="visibility"/>
</metric>
```

An `attributevalue` element has three attributes:

- `attr` (required) specifies the metamodel attribute to return.
- `element` and `eltype` (both optional) if you want to access the attribute of a related element, not the model element itself; we know these attributes from projections, see Section 8.1.1.4 "Filter Attributes "element" and "eltype"".

The below examples demonstrates the "element" attribute. The metric shows the name of a parameter's type.

© 2002-2013. All rights reserved.

```
<metric name="typename" domain="parameter">
  <description>The parameter type name.</description>
  <attributevalue element="parametertype" attr="name"/>
</metric>
```

8.1.4 Nesting

The `nesting` procedure can be used to calculate the level of recursive nesting of elements. For instance, packages can be nested within other packages, classes can contain inner classes, states can be contained in (composite) states, and so on. With the `nesting` procedure, you can count the levels of nesting.

The `nesting` procedure has one required attribute, `relation`, specifying the relation which establishes the nesting relationship. Usually, this will be the "context" relation. For example:

```
<metric name="NestingLevel" domain="package">
  <nesting relation="context" />
</metric>
```

The calculation procedure is as follows: retrieve the element that is referenced by the specified relation (here: context). If this element is "compatible", recursively calculate the metric for that element, increase by one, and return that value. Else (context element is not of the same type, e.g., top level packages defined in the namespace of the model), the metric yields 0.

An element is "compatible" with nesting metric *m* of domain type *t* if the type of the element is *t*, or if its type is a subtype or supertype of *t* and metric *m* is defined for the type.

For a typical package hierarchy, the results are (column NestingLevel):

Package	NestingLevel	LNest
javax	0	2
javax.swing	1	1
javax.swing.event	2	0

Table 5: Package nesting metrics example

How does the `nesting` procedure is differ from projections using the `nesting` attribute? Projections navigate the specified relation in the opposite direction. A projection operates on all elements where the specified relation points to the current element for which the projection is calculated (navigation to the current element). The `nesting` procedure, on the other hand, simply navigates from the current element to the element referenced by the specified relation (navigation away from the current element). Consequently, a metric LNest defined as a projection as follows would count the levels of nesting within a package (as shown in column "LNest" of Table 5):

```
<metric name="LNest" domain="package">
  <description>Levels of package nesting in this package.</description>
  <projection relation="context" target="package" nesting="true" />
</metric>
```

© 2002-2013. All rights reserved.

8.1.5 Signature

The `signature` procedure creates signature strings such as the name and parameter list of an operation.

```
<metric name="Signature" domain="operation" >
  <description>The signature of an operation.</description>
  <signature set="OPParameters" element="parametertype" />
</metric>
```

The elements that constitute the parameter list are specified by attribute `set`. Each element in the set can be subjected to the usual filters (target, element, eltype, condition, targetcondition, scope). In the above example, set OPParameters is defined to be the set of parameters of the operation, and we include the type of each parameter in the signature instead of the parameter itself.

The result is a string that contains the name of the operation, and the IDs of the types of the parameters in the order the parameters are defined in the XMI source file, for example 'getBalance(xmi4893,xmi238011)'. With these strings you can build, for instance, signature sets of the operations of a class and define metrics dealing with method overriding.

To obtain more human readable signature strings, you can further define the presentation of the elements on the signature list. The following definition generates signature strings such as 'getBalance[account:int; time:TimeStamp]':

```
<metric name="Signature2" domain="operation" >
  <description>Operation signature with brackets and parameter
     names/types in the clear.</description>
  <signature prologue="name+'['" set="OPParameters"
     value="name+':'+parametertype.name" separator="'; '" epilogue="']'" />
</metric>
```

The attributes to control the presentation are:

- `prologue`: defines the prologue of the signature string, and usually includes the name of the element and some version of an opening parenthesis. The expression is evaluated for the owner of the signature list. The default value is `name+'('`.
- `value`: defines the presentation of the elements on the signature list. The expression is evaluated for each list element. By default, this is the XMI ID of the element.
- `separator`: defines how the elements on the signature list are separated. The default value is a single comma.
- `epilogue`: defines the closing of the signature string, usually some version of a closing parenthesis. The expression is again evaluated for the owner of the signature list. The default value is `')'`.

8.1.6 Connected Components

The `connectedcomponents` procedure is used to count the connected components in a graph. The procedure uses element sets to obtain information about nodes and edges of the graph.

© 2002-2013. All rights reserved.

The procedure has four attributes:

- set (required): the element set that constitutes the nodes of the graph.
- nodes (required): a set expression that returns, for a node, the set of connected nodes to which there is an (outgoing) edge.
- undirected (optional): takes values "true" or "false". The default value is "false" and the procedure calculates the strongly connected components of the directed graph. When set to "true", the connected components of the underlying undirected graph will be computed (i.e., the direction of edges is ignored, an edge can always be followed in both ways).
- minnodes (optional): only counts connected components exceeding a certain size. For example, if you specify minnodes="3", only connected components that have at least three nodes will be reported.

For example, to calculate the number of inheritance hierarchies among the classes in a package, we define three element sets (see Section 8.2 "Definition of Sets"):

- set Classes: the set of classes in a package
- set Parents: the set of parent classes of a class
- set Children: the set of children classes of a class

We can then calculate the number of connected components in the inheritance hierarchy as follows:

```
<metric name="InhHierarchies" domain="package">
  <description>The number of inheritance hierarchies.</description>
  <connectedcomponents set="Classes" nodes="Parents+Children" />
</metric>
```

The following example uses an undirected graph and only counts connected components with at least two nodes (i.e., ignoring isolated classes that do not participate in any inheritance relationship):

```
<metric name="InhHierarchies" domain="package">
  <description>The number of non-trivial inheritance hierarchies.</description>
  <connectedcomponents set="Classes" nodes="Parents" undirected="true"
     minnodes="2" />
</metric>
```

8.1.7 Value Filter

The "filtervalue" procedure evaluates a metric expression for the first element in a relation or set, and returns the result of this metric expression.

The following example uses the "filtervalue" procedure to access the data value of a particular tagged value pair. We assume elements of type "taggedvalue" have two metrics "TagName" and "TagData", which yield the name and data value of the tagged value, respectively.

```
<metric name="Author" domain="class">
  <description>Author of the class.</description>
  <filtervalue relation="context" target="taggedvalue"
               condition="tagName='Author'" value="tagData"/>
</metric>
```

© 2002-2013. All rights reserved.

The above metric takes the first tagged value with tag name 'Author' that it finds, evaluates the expression specified by the "value" attribute for that tagged value, and returns the result of that expression as the result of the metric.

The "filtervalue" procedure has the following attributes:

- `relation` (optional): name of the relation that contains the elements to check.
- `relset` (optional): an element set that contains the elements to check. Exactly one of the attributes `relation` or `relset` must be specified.
- the usual optional filter attributes: target, targetcondition, element, eltype, scope, condition (see Section 8.1.1 "Projection").
- `value` (optional): the metric expression to evaluate for the first matching element. When omitted, the element itself is returned.

8.1.8 Subelements

The `subelements` procedure allows for a simple way to count all elements, or all elements selected types, that a given model element owns directly or indirectly.

For example, to count for a package the number of operations that are defined in interfaces, classes (including inner classes), subsystems etc. of the package and all of its subpackages and subsystems, sub-subpackages and so on, at any level, we can simply write:

```
<metric name="NumOps_tc" domain="package">
  <description>The number of operations defined in the package, its subpackages,
      etc.</description>
  <subelements target="operation"/>
</set>
```

One could achieve the same result with projections, however, it would require the definition of several helper metrics to gather operations in interfaces, classes, etc. separately and take the sum of these metrics.

The `subelements` procedure takes the following, optional, attributes:

- the filter attributes target, targetcondition, element, eltype, condition, and scope (see Section 8.1.1 "Projection").
- the summation attributes sum and stat (see Section 8.1.1.7 "Attributes "sum" and "stat"").

8.1.9 Substring

UML tools sometimes mangle various bits of tool-specific information into a single string in the XMI file. For example, the coordinates of an element in a diagram could be specified as follows:

- `...<geometry>206, 320, 69, 13</geometry>...`, or
- `... location="Left=620;Top=743;Right=732;Bottom=783" ...`

© 2002-2013. All rights reserved.

With the `substring` procedure we can extract parts of such strings. Substring extraction is based on separator strings. In its simplest form, we have something like a comma-separated list of values such as "206, 320, 69, 13". The following definition extracts the third value, "69", from the list:

```
<metric name="width_string" domain="diagramelement">
  <substring source="geometry" separator="','" position="2"/>
</metric>
```

Attribute `source` specifies a metric expression that yields the string we want to dissect. Attribute `separator` specifies the separator string, which can be of any length. This will usually be just a string constant, so it needs to be enclosed in single quotes (see Section 8.5.1.1 "Constants"). In our example, the separator is a single comma, which will split the string "206, 320, 69, 13" into four parts: "206", "320", "69", and "13". Leading or trailing whitespaces are automatically trimmed from each substring.

With the `position` parameter, we tell SDMetrics which substring we want; the first substring is at position 0, position 2 yields the third value, "69". With negative positions -1, -2, etc. we can access the last substring, last but one substring, and so on, without having to know how many substrings there are. The default value for attribute `position` is -1 (return the last substring).

Often we do not want to use the substring as is, but process it further, e.g., retrieve its numerical value. We can do this directly with the optional attribute `result`. The following metric extracts the first value of a comma-separated list and converts it to a number:

```
<metric name="left" domain="diagramelement">
  <substring source="geometry" separator="','" position="0"
    result="parsenumber(_value)"/>
</metric>
```

Attribute `result` specifies a metric expression, variable `_value` contains the value of the substring we extracted.

In a source string such as "Left=620 ; Top=743 ; Right=732 ;", the beginning and end of the interesting substrings are delimited by different separators. The optional argument `endseparator` deals with this situation:

```
<metric name="top" domain="diagramelement">
  <substring source="location" separator="'Top='" endseparator="';'"
    result="parsenumber(_value)"/>
</metric>
```

The separator 'Top=' cuts up the source string into two substrings: "Left=620 ;" and "743 ; Right=732 ;". Of these, we use the second (last) substring, because the `position` argument is missing. The attribute `endseparator="';'"` instructs SDMetrics to cut off the substring at the first semicolon. After automatic trimming of whitespace, this leaves us with the string "743", which then gets converted to the number 743.

The "substring" procedure has the following attributes:

- `source` (required): Metric expression that yields the source string.

© 2002-2013. All rights reserved.

- `separator` (required): Metric expression that yields the separator string.
- `position` (optional): Metric expression that yields the index of the substring to use. Uses the last substring when omitted.
- `endseparator` (optional): Metric expression that yields a separator string to further delimit the substring to the right. Uses the entire substring when omitted.
- `limit` (optional): The maximum number of substrings returned. The last substring contains the remainder of the source string, even if there are further occurences of the separator string. When omitted, the number of substrings is determined by the number of occurences of the separator string.
- `result` (optional): A metric expression to be applied to the resulting substring. The variable "_value" contains the value of the substring. When omitted, the substring itself is returned.

8.2 Definition of Sets

For a UML model element, you can define any number of sets. Sets are useful to simplify the definition of metrics and rules (as we already have seen in the previous section), and to define metrics with set semantics. For instance, there can be multiple associations between two classes. If you want a metric not to account for each association separately, but just the fact that there is at least one association, you can accomplish this with sets.

We distinguish two types of sets:

- *element sets*, which are sets of UML model elements, for example, the set of descendent classes of a class,
- *value sets*, which are sets of metric values.

We also distinguish between multisets and regular sets. In a multiset, an element can occur multiple times, whereas in a regular set, an element can occur only once. Multisets are also known as "bags".

The distinctions element/value set and multi/regular set are orthogonal. You can have regular element or value sets as well as multisets of elements and multisets of values.

A set is defined with an XML element as follows:

```
<set name="setname" domain="setdomain" multiset="false|true"
       inheritable="true/false">
  <description>Description of the set</description>
  <'set definition' ...>
</set>
```

The attributes in the opening "set" tag are:

- `name` (required): the name of the set, used to reference the set by other metrics in the metrics definition file.
- `domain` (required): the type of element for which the set is defined (e.g., package, class, operation).

© 2002-2013. All rights reserved.

- `multiset` (optional): indicates if this set is a multiset (`multiset="true"`) or a regular set (`multiset="false"`). When omitted, this attribute defaults to "false".
- `inheritable` (optional): indicates if the set should also be defined for all subtypes of the "domain" element type. When you set the attribute value to "true", all direct and indirect subtypes of the domain "inherit" the set definition, and the set can be calculated for all subtypes, too. When omitted or set to "false", the set definition is not passed on to subtypes.

Like metrics, the sets for one domain must have unique names. Also, within one domain you cannot have a set and a metric of the same name. For example, if you defined a set "Ancestors" for classes, you cannot also define a metric "Ancestors" for classes.

Following the "set" tag is an optional description of the set, enclosed in `<description>` tags. Set descriptions are currently not shown to the end user, and mostly serve as comments for the maintainer of the metric definition file. See Section 8.6 "Writing Descriptions" how to write (set) descriptions.

Following the description is an XML element that defines the calculation procedure for the set. We describe these calculation procedures in detail in the following subsections.

8.2.1 Projection

We have already seen how projections work for the definition of metrics (Section 8.1.1 "Projection"). The definition of sets via projections works almost the same way. For example, an element set containing the child classes of a class can be defined as follows:

```
<set name="Children" domain="class">
  <description>The set of children of the class.</description>
  <projection relation="genparent" target="generalization"
              element="genchild" eltype="class"/>
</set>
```

The following example defines a multiset containing the supplier classes to which a class has a dependency link. If a class has multiple dependency links to the same supplier class, that supplier class will occur multiple times in the resulting set.

```
<set name="SuppClasses" domain="class" multiset="true">
  <description>The supplier classes of which the class
  is a client in a dependency.</description>
  <projection relation="depclient" target="dependency"
    element="depsupplier" eltype="class" />
</set>
```

The following attributes we know from projections for metrics (see Section 8.1.1 "Projection") are also available for set projections:

- `relation`: name of the relation that contains the elements to include in the set
- `relset`: a set expression defining the elements of the set. Exactly one of the attributes relation or relset must be specified.
- `target`: element type(s) of related elements
- `element`: for relations with an indirection, relation links to follow

© 2002-2013. All rights reserved.

- `eltype`: for relations with an indirection, type(s) of indirectly related element
- `targetcondition` and `condition`: reduce set to elements which fulfill the specified expression
- `scope`: reduce set to elements which fulfill a scope condition in relation to the element for which the set is defined

In the remainder of this section, we describe set projection attributes that are new or have changed meanings from the metric projection attributes.

8.2.1.1 Attribute "recurse"

The `recurse` attribute known from metrics projections is also available for sets. By setting `recurse="true"`, SDMetrics recursively calculates and takes the union of sets for compatible elements in the relation. The element is compatible if it is of the same type as the domain of the metric, or a subtype or supertype and the set is also defined for the type.

For example, we can easily change the definition of the set of children classes from above to yield the element set of descendents classes:

```
<set name="Descendents" domain="class">
  <description>The set of descendents of a class.</description>
  <projection relation="genparent" target="generalization"
              element="genchild" eltype="class" recurse="true"/>
</set>
```

For each child class, the set "Descendents" is evaluated, and the union of all these sets is taken. The child classes themselves also are included in the set. This effectively produces the set of descendents for a class.

8.2.1.2 Attribute "set"

The `set` attribute is the set counterpart of the `sum` attribute for metrics. If you specify a set expression using the `set` attribute, the set expression is evaluated for each element in the projection, and the union over all these sets is taken.

In the following example, we assume we have defined two sets for classes:

- "AssCls" - the set of classes that have an association with the given class
- "DepCls" - the set of class that have a dependency (client or supplier) with the given class

We can then define, for a package, the set of classes that have an association or dependency link with a class in the package:

```
<set name="AssCls" domain="package">
  <projection relation="context" target="class"
              set="AssCls+DepCls" />
</set>
```

© 2002-2013. All rights reserved.

The sets referenced in the `set` expression may be either all element sets, or all value sets (see also Section 8.2.1.4 "Attribute "valueset""). The resulting set will be of the same type as the constituent sets.

8.2.1.3 Attribute "exclude_self"

The attribute `exclude_self` can take the values "true" or "false" (default).

When `exclude_self` is set to true, the element for which the set is calculated is not included in the result set. For example, assume we want to determine, for a state in a state diagram, the set of states that can be reached from that state:

```
<set name="Reachable_States" domain="state">
  <description>The set of reachable states.</description>
  <projection relation="transsource" target="transition"
              element="transtarget" eltype="state"
              recurse="true" exclude_self="true" />
</set>
```

By setting `exclude_self` to true, the state for which the set is calculated is not included in the result set, even if the state-transition graph has loops back to the state.

8.2.1.4 Attribute "valueset"

So far, the examples shown were element sets, containing UML model elements. With the `valueset` attribute, you can define value sets containing the values of metric expressions.

For example, if a metric "Signature" for operations contains the signature string of the operation, you can define the set of signatures for the operations of a class as follows:

```
<set name="Sigs" domain="class">
  <description>The set of signatures of the operations in the
     class.</description>
  <projection relation="context" target="operation"
              valueset="Signature" />
</set>
```

The metric expression "Signature" is evaluated for each operation; the resulting values are stored in the set.

Attribute `valueset` may be used in combination with all other attributes except `set` or `exclude_self`.

8.2.2 Subelements

Similar to the `subelements` procedure for metrics (see Section 8.1.8 "Subelements"), the `subelements` procedure for sets allows for a simple way to define a set of all elements that a given model element contains.

For example, to define the set of all actors in a model, we simply write:

© 2002-2013. All rights reserved.

```
<set name="ActorSet" domain="model">
  <description>The set of actors defined in the model.</description>
  <subelements target="actor"/>
</set>
```

The `subelements` procedure for sets takes the following, optional, attributes:

- the filter attributes target, targetcondition, element, eltype, condition, and scope (see Section 8.2.1 "Projection"),
- summation attributes set (see Section 8.2.1.2 "Attribute "set""), exclude_self (see Section 8.2.1.3 "Attribute "exclude_self""), and valueset (see Section 8.2.1.4 "Attribute "valueset"").

8.3 Definition of Design Rules

SDMetrics' design rules and heuristics (see Section 4.7 "The View 'Rule Checker'") are defined in the metric definition file, making it easy for you to modify existing rules and add new rules of your own.

The SDMetricsML defines a design rule with an XML element like follows:

```
<rule name="rulename" domain="ruledomain"
      category="rulecategory" severity="ruleseverity"
      applies_to="ruleapplication" disabled="false|true"
      inheritable="true/false" >
  <description>Description of the rule.</description>
  <'rule definition'" />
</rule>
```

The attributes of the enclosing "rule" tag are:

- `name` (required). The name of the rule, used to identify the rule in the output.
- `domain` (required). The element type for which the rule is defined.
- `category` (optional). The name of the category to which this rule belongs, e.g., "Incomplete Design", "Naming", "Modularity", etc.
  Note that there is no preconceived categorization of design rules to adhere to in SDMetrics - you can define any categories you deem fit.
- `severity` (optional). The severity of the rule indicates how critical a violation of the rule is, e.g., "very severe", "moderate", etc.
  Again, there is no preconceived severity scale in SDMetrics, you can define any severity levels you like. However, you should consider that users will sort the list of reported rule violations by the severity of rules. The sorting is done lexicographically. Therefore, you should define the severity levels in a manner that they can be meaningfully sorted. For example, instead of "high", "medium", "low", define the levels as "1-high", "2-med", "3-low" to preserve the scale when sorting.
- `applies_to` (optional). Defines when the rule is applicable. For instance, some rules may only be useful at a particular design phase (requirements, analysis, design), or for a particular type of system (say, real time systems). The user can then decide to only check rules that are applicable to the model at hand, and ignore the remainder of the rules (see Section 4.7.1 "Filtering Design Rules").

© 2002-2013. All rights reserved.

To use this feature, you first identify the application areas you wish to distinguish, and define a label for each area. The label must be a sequence of characters, without whitespaces and punctuation, for example "analysis", "design", or "realtime". Once more, there is no preconceived set of application areas, you can define whatever areas you find useful.
Once you have identified your application areas and their labels, specify for each rule the label(s) of the area(s) to which the rule applies. If several areas apply, separate the labels with commas. For example `applies_to="design"` or `applies_to="analysis,realtime"`.
If you do not specify the `applies_to` attribute for a rule, SDMetrics will consider the rule to be applicable to all application areas.

- `disabled` (optional). With this attribute you can disable a rule. The attribute takes values `true` (rule will not be checked), or `false` (the default, rule will be checked).
- `inheritable` (optional). Indicates if the rule should also apply to subtypes of the "domain" element type. When you set the attribute value to "true", all direct and indirect subtypes of the domain "inherit" the rule definition, and the rule will be checked for all subtypes, too. When omitted or set to "false", the rule definition is not passed on to subtypes.

Next is an optional description of the rule, enclosed in `description` tags. The description will be shown in the measurement catalog (see Section 4.13 "The View 'Catalog'"). Section 8.6 "Writing Descriptions" explains how to write rule descriptions.

Following the rule description is an XML element that defines the check the rule performs. There are a number of procedures to choose from:

- `violation` - rules defining a condition expression that must not be violated.
- `cycle` - rules that check for the presence of cycles in a directed graph.
- `projection` - rules that report elements in a projection.
- `valueset` - rules that reports values occurring in a value set.

8.3.1 Violation

The "violation" procedure defines a condition expression for a model element. If the condition evaluates to true, the model element violates the rule and is reported.

For example, consider a rule stating that an abstract class should have child classes. The abstract class itself cannot be instantiated, so child classes are required for the class to become useful in the system. That rule can be defined as follows:

```
<rule name="NoSpec" domain="class" category="Completeness" severity="2-med">
  <description>Abstract class has no child classes, must be
              specialized to be useful.</description>
  <violation condition="abstract='true' and NOC=0" />
</rule>
```

The condition expression `condition` checks if the class is abstract, and the number of children of the class (metric NOC) is zero.

The "violation" procedure has two attributes;

© 2002-2013. All rights reserved.

- `condition` (required). Defines a condition expression (see Section 8.5.4 "Condition Expressions"); the rule is violated if the condition expression is "true".
- `value` (optional). Specifies a metric expression that is evaluated for the violating model element. The value is displayed to the user and can be used to provide additional details how the model element violates the rule.

8.3.2 Cycle

The cycle procedure detects cycles in a directed graph. It does so by calculating the strongly connected components of the graph. Within a strongly connected component (SCC), there exists a path from each node to every other node of the SCC, hence a cycle. A graph is acyclic if and only if it has no SCCs.

You can use this procedure to check for cyclic dependencies between model elements. For example, the following rule checks for cycles in the inheritance graph for classes:

```
<rule name="CyclicInheritance" domain="class" category="Inheritance"
      severity="1-high">
  <description>Class inherits from itself!</description>
  <cycle nodes="Parents" />
</rule>
```

The domain of the rule is "class", so the classes of the model constitute the nodes of the graph. The required set expression `nodes` yields, for each node, the set of connected nodes to which there is an edge. In our example, set "Parents" is the set of parents of the class.

To report the SCCs found, each model element in a SCC receives a label of the form "cyc# c (n nodes)", where *c* is the number of the SCC, and *n* is the number of nodes in the SCC.

The rule reports a violation for each model element in each SCC. The value of the rule shown to the user is the label of the model element. That way, users can tell which elements belong to which connected component.

In some cases, you may only want to report SCCs of a certain size, for instance, SCCs with three or more nodes. To this end, the cycle procedure has a second, optional attribute `minnodes`, which specifies the minimum number of nodes an SCC must have to be reported. The default is 1 (nodes having an edge back to themselves).

8.3.3 Projection for Rules

The projection procedure for rules calculates a set projection for a model element (see Section 8.2.1 "Projection"), and reports any elements contained in there as violation of the rule. Consider this example:

© 2002-2013. All rights reserved.

```
<rule name="MissingGuard" domain="state" category="Correctness"
      severity="1-high">
  <description>If there are two or more transitions from a choice
      state, they all must have guards.</description>
  <projection precondition="kind='choice' and Outgoing>1"
      relation="transsource" target="transition" condition="Guards=0"
</rule>
```

Attributes `relation`, `target`, and `condition` define the projection, and are processed exactly the same way as we know them from set projections. In the above example, the projection yields, for a state, the set of outgoing transitions that have no guard.

In the context of a rule, a projection can have three additional attributes:

- `precondition`: A condition expression that is evaluated for the model element about to be checked. If the condition is false, the rule will not be checked. In the example above, the rule is only applicable if the state is a choice state (`kind='choice'`) and has at least two outgoing transitions (`Outgoing>1`).
- `value`: defines the value of the rule that is displayed to the user. This is a metric expression, and is evaluated for the model elements returned by the projection. If the projection returns a valueset, no value needs to be specified - the value of the rule is the value returned by the projection.
- `mincnt`: defines the minimum number of times an element must be included in the resulting multiset to be reported. You can use this feature to find and report duplicates, as shown in the next example. The default value is 1 (report all elements).

```
<rule name="DupName" domain="state" category="Correctness" severity="1-high">
  <description>The compound state has two or more states of
                 the same name.</description>
  <projection relation="context" target="state" condition="name!=''"
              valueset="name" mincnt="2" />
</rule>
```

For rule "DupName", the projection defines the value set of names of the states of a compound state (omitting anonymous states). By setting `mincnt="2"`, only names that occur two times or more will be reported.

Note the following restrictions that apply in the context of a rule projection:

- The resulting set is always a multiset.
- Attribute "recurse" is not allowed for rules. If you need the rule to operate on a recursively defined set, define the set using a regular set projection (Section 8.2.1 "Projection"), and feed that set into the rule projection with the "relset" attribute.

8.3.4 Valueset for Rules

The valueset procedure for rules calculates a value set for a model element (see Section 8.2.1.4 "Attribute "valueset""), and reports any elements contained in there.

For example, assume we have a value set "AttrNameSet" defined for classes, which is a multiset of the names of the attributes of the class. We can then define a rule to check for duplicate names:

```
<rule name="DupAttrNames" domain="class" category="Correctness"
   severity="1-high">
  <description>The class has two or more attributes with
     identical names.</description>
  <valueset set="AttrNameSet" mincnt="2" />
</rule>
```

The procedure evaluates the `set` expression for the class, and reports all elements that occur at least twice. The value of the rule is the value of the reported element in the value set; in our example this is the duplicate attribute name. In addition to the `set` attribute, the valueset procedure accepts the attributes `mincnt` and `precondition`, which have the same meaning as they have for rule projections (see Section 8.3.3 "Projection for Rules").

8.3.5 Word lists

One aspect of design rule checking involves the use of reserved keywords of the UML or programming languages as names of model elements.

To check model element names for keywords, we define a list of keywords in the metric definition file, and add a rule that checks for names on the list:

```
<wordlist name="CPP" ignorecase="false" >
  <entry word="auto" />
  <entry word="bool" />
  <entry word="break" />
  ...
</wordlist>

...

<rule name="Keyword" domain="class" category="Naming" severity="1-high">
  <description>Class name is a reserved C++ keyword.</description>
  <violation condition="name onlist CPP" value="name" />
</rule>
```

The word list is enclosed in `wordlist` tags. Attribute `name` defines a name for the keyword list, which is used later to reference the list. Attribute `ignorecase` controls if uppercase and lowercase letters should be distinguished when searching for a word on the list. For programming languages that are case-insensitive, set `ignorecase` to "true". By default, `ignorecase` is "false" (i.e., case-sensitive search).

The word list contains a list of `entry` elements. An `entry` has one required attribute `word` that contains one word of the list.

To check if a word is on a list, there is a special operator `onlist` which you can use in condition expressions (see Section 8.5.4 "Condition Expressions"). In the condition expression of rule "Keyword" above, the left hand operator yields the name of the class, the right hand operator is the name of the word list to check.

© 2002-2013. All rights reserved.

Note that while the main purpose of the word list feature is to define design rules checking for keywords, there are other possible uses:

- Your development standards may forbid certain classes or data types to be used as types of attributes, parameters, return types, etc. To define a rule that checks adherence to this standard, create a word list with the names of the forbidden types, and then create rules to check if the name of an attribute type etc. appears on the list.
- Word lists can also be used to simplify condition expressions of set or metric definitions. A condition of the form `val='string1'|val='string2'|...|val='stringn')` can be rewritten: define a word list containing the strings `string1` to `stringn`, and reduce the condition expression to `val onlist wordlist`.

8.3.6 Exempting Approved Rule Violations

As described in Section 4.7.2 "Accepting Design Rule Violations", the SDMetrics rule checker uses tagged values or comments to exempt a particular model element from a particular rule, so that such an approved violation is no longer reported. Therefore, the rule checker needs to know which model element type stores the tagged values or comments in the SDMetrics metamodel, and how to access their contents.

This information is specified in two attributes of the XML root element of the metric definition file, for example:

```
<sdmetrics version="2.1" ruleexemption="taggedvalue"
  exemptiontag="tagname" >
  ...
```

- *ruleexemption* specifies the model element type that stores the tagged values or comments
- *exemptiontag* is a metric expression that can be evaluated for the model elements specified by the *ruleexemption* attribute. This expression should yield the text to be searched for occurrences of strings of the form `violates_rulename` (where `rulename` is the name of a design rule). For tagged values, this should be the tag, for comments, this should be the body text of the comment.

The above example defines that tagged values are contained in meta model element "taggedvalue", which has an attribute or metric "tagname" that stores the tag.

8.4 Definition of Relation Matrices

The metric definition file also hosts the definition of SDMetrics' relation matrices (see Section 4.10 "The View 'Relation Matrices'"). The SDMetricsML defines a relation matrix with an XML element as follows:

```
<matrix name="matrixname" from_row_type="rowtype" to_col_type="columtype"
        row_condition="..." col_condition="..." >
  <description>Description of the matrix.</description>
  <'matrix definition' ...>
</matrix>
```

© 2002-2013. All rights reserved.

The attributes in the opening "matrix" tag are:

- `name` (required): the name of the matrix, used in the dropdown list of the relation matrix view, and as filename when saving the relation matrix to files.
- `from_row_type` (required): the type of the source elements that will make up the rows of the matrix (e.g., package, class, operation).
- `to_col_type` (required): the type of target elements that will make up the columns of the matrix. This can be the same type as the row elements, or a different type.
- `row_condition` (optional): a condition expression (see Section 8.5.4 "Condition Expressions") that must be true for a source element to be included in the matrix rows. If the attribute is not specified, all source elements will be included.
- `col_condition` (optional): a condition expression that must be true for a target element to be included in the matrix columns. If the attribute is not specified, all target elements will be included.

Following the matrix tag is an optional description of the matrix, enclosed in `<description>` tags. The description is shown in the measurement catalog (see Section 4.13 "The View 'Catalog'"). Section 8.6 "Writing Descriptions" explains how to write descriptions for matrices.

The matrix definition is an XML element that defines the calculation procedure establishing the relationship from source elements to target elements in the matrix. The available procedures are the same ones that are available for set definitions (see Section 8.2 "Definition of Sets").

The definition below calculates a relation matrix showing UML abstractions for classes, i.e., which classes implement which interfaces:

```
<matrix name="UMLAbstr" from_row_type="class" to_col_type="interface">
  <description>UML abstractions: implementation of interfaces.</description>
  <projection relation="client" target="abstraction" element="supplier" />
</matrix>
```

The evaluation procedure for this matrix definition is as follows. For a source element in a given matrix row, the projection is evaluated, as described in Section 8.2.1 "Projection". The number of occurrences of each target element in that projection is counted, and entered in the respective column of the relation matrix. This procedure is repeated for all rows of the matrix. Thus, we obtain in every table cell, a count of the number of relationship links from the source element to the target element.

In the context of a relation matrix definition, the set projection attribute `recurse` is not allowed.

As another example, assume we have defined, for a class, the set "AssEl_out" of elements associated with the class by bidirectional or navigable outgoing associations. We can then simply create a relation matrix showing these associations as follows:

```
<matrix name="ICAssoc_Cls" from_row_type="class" to_col_type="class">
  <description>Import coupling via associations between classes.</description>
  <projection relset="AssEl_out" />
</matrix>
```

© 2002-2013. All rights reserved.

The evaluation procedure for this matrix definition is straightforward. For a source element in a given matrix row, the specified set expression "relset" is evaluated, as described in Section 8.2.1 "Projection". For each target element, the value of the respective column of the relation matrix is the cardinality of the target element in the set. This procedure is repeated for all rows of the matrix.

Continuing the association matrix example from above, we may want to reduce the size of the matrix by dropping empty rows (classes without outgoing associations), and empty columns (classes without incoming associations). Assume we have defined two helper metrics "Num_AssEl_out" and "Num_AssEl_in", to count for a class the number of outgoing and incoming associations, respectively. We add row and column conditions to the matrix definition as follows:

```
<matrix name="ICAssoc_Cls" from_row_type="class" to_col_type="class"
        row_condition="Num_AssEl_out>0" col_condition="Num_AssEl_in>0">
  <description>Import coupling via associations between classes.</description>
  <projection relset="AssEl_out" />
</matrix>
```

Then, only classes with one ore more outgoing associations (`Num_AssEl_out>0`) will be included in the rows of the matrix, which eliminates empty rows. Likewise, the column condition `Num_AssEl_in>0` eliminates empty columns.

© 2002-2013. All rights reserved.

8.5 Expression Terms

The entries in the metric definition file use three types of expression terms:

- metric expressions, which return a number or string value, or reference a model element
- set expressions, which return a set or multiset of model elements or metric values,
- condition expressions, which return a Boolean value (true or false).

The following subsections describe each of these expression types in an informal manner. A formal definition is given in Appendix E: "Project File Format Definitions".

8.5.1 Constants and Identifiers

8.5.1.1 Constants

Expression terms frequently contain string and number constants.

A string constant is a sequence of characters, enclosed in single quotes, for example: `'public'`. A string constant may contain any sequence of digits (0-9), letters (a-z, A-Z), whitespace, and the underscore (_).

Number constants (integer or floating point) are denoted in the usual way:

- preceded by an optional sign (+ or -)
- always use a dot as decimal point
- if you use scientific notation, separate the exponent from the mantissa with a 'e' or 'E', like so: -0.234e-5 or 1.2E34

Examples of valid numbers are: 2, -1.3, 1e-10.

8.5.1.2 Identifiers

Identifiers denote the name of attributes of metamodel elements, the name of metrics, the name of sets, or variables (which we will discuss in more depth in a moment).

An identifier is a letter or underscore (a-z, A-Z, _), followed by any sequence of digits (0-9), letters, and underscores. Examples of valid identifiers are: i, visibility, NumOps, set2, Metric1_a, _principal.

By convention, attribute names start with a lowercase letter, metric and set names with an uppercase letter, and variables with an underscore. This avoids name conflicts between metrics or sets, attributes, and variables.

© 2002-2013. All rights reserved.

Variables

Metric, set, and rule procedures can define variables that hold values with special meanings for the calculation. We access those values in metric, set, and condition expressions. SDMetrics defines two standard variables `_self` and `_principal`.

The variable **`_self`** yields the model element for which the expression is evaluated. The variable **`_principal`** yields the model element for which the current metric, set, or rule is evaluated. Variables `_principal` and `_self` need not be the same element. In fact, the variable `_principal` should only be used if it is not the same element as `_self`.

Consider the following example. The metric counts, for a class `A` the supplier classes on which `A` depends that have more operations than `A`:

```
<metric name="SuppliersWithMoreOperations" domain="class">
  <description>The number of classes this class depends on that have
     more operations.</description>
  <projection relation="depclient" target="dependency" element="depsupplier"
    eltype="class" condition="_self.NumOps > _principal.NumOps"/>
</metric>
```

The `condition` expression is evaluated, in turn, for each supplier class of class `A`. The variable `_self` therefore refers to the supplier class. The variable `_principal` always refers to class `A` for which the metric is calculated.

Note: the condition expression in the above example could have also been written as `"NumOps > _principal.NumOps"`, because the identifier `NumOps` will be interpreted as a metric of the model element for which the expression is calculated. In fact, variable `_self` is rarely needed in practice. Useful applications include

- for clarity/readability, as in the above example to make it clear which model elements are being compared.
- to pass the element for which an expression is evaluated into a function, for example in `qualifiedname(_self)`.

8.5.2 Metric Expressions

A metric expression is an expression that returns a number, a string, or a model element that can be used as the value of a metric. A metric expression is made up of constants, identifiers, and operators. These are described in the following.

8.5.2.1 Mathematical Operators and Functions

The operators allowed in a metric expression are:

- a+b (a plus b, or, if a is a string, the string concatenation of a and b)
- a-b (a minus b)
- a*b (a multiplied by b)
- a/b (a divided by b)

© 2002-2013. All rights reserved.

- a^b (a to the power of b)
- a->b or a in b (the number of times a is contained in (multi-)set b; the alternate notation `in` can be used to avoid conflicts with the special XML character >, see Section 8.5.5 "Expression Terms and XML")

The operators have the following precedence:

1. ^ and -> (or in)
2. the unary - and +
3. * and /
4. + and -

In addition, a number of math and special purpose functions are available.

Math functions

- `ln(x)`: Returns the natural logarithm of x.
- `exp(x)`: Returns Euler's number e raised to the power of x.
- `sqrt(x)`: Returns the square root of x.
- `abs(x)`: Returns the absolute value of x.
- `ceil(x)`: Returns the smallest integer greater or equal to x.
- `floor(x)`: Returns the largest integer smaller or equal to x.
- `round(x)`: Returns the integer closest to x (`round(x)=floor(x+0.5)`).

Special purpose functions

- `length(s)`: Returns the length of the string s.
- `size(s)`: Returns the number of elements in set s (respecting cardinality if s is a multiset).
- `flatsize(s)`: Returns the number of different elements in set s. For regular sets, this is the same as `size`, for multisets, the cardinality of elements is disregarded.
- `parsenumber(s)`: Returns the numerical value of a string that represents a number, e.g. '3' or '-5.15'. The formatting for number constants applies (see Section 8.5.1.1 "Constants").
- `tolowercase(s)`: Returns a string with all upper case letters in s changed to lower case.
- `typeof(e)`: Returns the type of the model element e, as a string.
- `qualifiedname(e)`: Returns the fully qualified name (see Section 4.2.2.1 "Qualified Element Names") of the model element e.

Parentheses can be used as usual to override the default precedence. An example of a valid metric expression is `-0.5*NumOps^2+(self->AssocCls+ln(NumOps))*NOC/(1+NOD)`.

8.5.2.2 Special Operators

The following special operators help the navigation between model elements.

1. The "dot" operator

In an expression term, it is sometimes useful to refer to attribute or metric values of related elements. This can be accomplished with the dot operator.

© 2002-2013. All rights reserved.

For example, to access the name of an operation's owner, we simply write "context.name". The left hand side expression "context" refers to the owner of the operation, which has to be some model element (e.g., a class). For that model element, the right hand side expression "name" is evaluated, which yields the value of its attribute "name".

In general, evaluating expression `a.b` for a model element *e* works as follows:

- Evaluate metric expression `a` for model element *e*. This yields a second model element, *e'*.
- Evaluate metric expression `b` for model element *e'*. This is the result of `a.b`.
- If an error occurs during evaluation of expression `a` or `b`, or if expression `a` does not yield a model element, no error is reported. Instead, an empty string is returned as value of `a.b`.

2. The "upto" operator

If we want to determine the package in which a class resides, we usually access it via its "context" attribute. However, the owner of a class need not be a package. For an inner class, for example, the "context" attribute yields its containing outer class.

The "upto" operator in the expression "context upto (typeof(self)='package')" follows the "context" attribute, across several model elements if necessary, until it encounters a model element of type package.

In general, evaluation of the expression `a upto b` for a model element *e* works as follows:

1. Evaluate metric expression `a` for model element *e*. Let *e'* denote the result of this evaluation.
2. If *e'* is not a model element, an empty string is returned as value of `a upto b`.
3. Evaluate condition expression `b` for model element *e'*.
4. If the result of `b` is true, then *e'* is returned as result of the metric expression `a upto b`.
5. Otherwise, set *e*=*e'* and repeat from step 1.

3. The "topmost" operator

Similar to the "upto" operator, the "topmost" operator successively evaluates a condition expression for a chain of model elements. Whereas the "upto" operator returns the first element that fulfills the condition expression, the "topmost" operator returns the last element in the chain that fulfills the condition expression.

The "topmost" operator in the expression "context topmost (typeof(self)='package')" yields the top-level package that contains the element.

In general, evaluation of the expression `a topmost b` for a model element *e* works as follows:

1. Set *result* to be the empty string.
2. Evaluate metric expression `a` for model element *e*. Let *e'* denote the result of this evaluation.
3. If *e'* is not a model element, return *result* as value of `a upto b`.
4. Evaluate condition expression `b` for model element *e'*.
5. If the result of `b` is true, set *result*=*e'*.
6. Set *e*=*e'* and repeat from step 2.

© 2002-2013. All rights reserved.

8.5.3 Set Expressions

A set expression is an expression that returns a set. It is made up of identifiers (names of sets), and set operations. Identifiers have been described in Section 8.5.1.2 "Identifiers". The admissible set operations are:

- A+B: the union of A and B.
  If both A and B are regular sets, the resulting set will be a regular set containing the elements that occur in A or in B or both.
  If A or B is a multiset, the resulting set will be a multiset, respecting the cardinality of elements. For example, if element *e* occurs five times in multiset A, and twice in multiset B, its cardinality in the union is seven. If B is a regular set containing element *e*, the cardinality of *e* in the union is six.
- A*B: the intersection of A and B.
  If A or B is a regular set, the resulting set will be a regular set containing the elements that occur both in A and in B.
  If both A and B are multisets, the resulting set will be a multiset, respecting the cardinality of elements. For example, if element *e* occurs five times in multiset A, and twice in multiset B, its cardinality in the intersection is two.
- A-B: A without B.
  If A is a regular set, the resulting set will be a regular set containing all elements that occur in A but not in B.
  If A is a multiset, the resulting set will be a multiset, respecting the cardinality of elements. For example, if element *e* occurs five times in multiset A, and twice in multiset B, its cardinality in the resulting set is three. If B is a regular set containing element *e*, the cardinality of *e* in the resulting set is four.
- `a`.B: set B of model element `a`.
  The dot operator (see Section 8.5.2.2 "Special Operators") is also available in set expressions. The difference is that the right hand side, B, must be a set expression.
  Evaluating expression `a`.B for model element *e* thus works as follows:
  - Evaluate metric expression `a` for model element *e*. This yields a second model element, *e'*.
  - Evaluate set expression B for model element *e'*. This is the result of `a`.B.
  - If an error occurs during evaluation of expression `a` or B, or if expression `a` does not yield a model element, no error is reported. Instead, a regular empty set is returned as value of `a.B`.

The dot operator has precedence over the * operator, which has precedence over the + and - operators. A+B-C is equivalent to (A+B)-C, not A+(B-C). Use parenthesis to enforce the intended precedence.

An example for a valid set expression is ((A+B)-context.C)*D.

8.5.4 Condition Expressions

A condition expression is an expression that returns a Boolean value (true or false). Condition expressions are made up of relational operators comparing metric expressions, Boolean functions, and logical operators combining condition expressions.

© 2002-2013. All rights reserved.

8.5.4.1 Relational Operators

The relational operators to compare metric expressions (as in Section 8.5.2 "Metric Expressions") are:

- `a=b` (equals)
- `a!=b` (not equals)
- `a<b` or `a lt b` (less than; the alternate notation `lt` can be used to avoid conflicts with the special XML character <, see Section 8.5.5 "Expression Terms and XML")
- `a<=b` or `a le b` (less or equal)
- `a>b` or `a gt b` (greater than)
- `a>=b` or `a ge b` (greater or equal)

These operators apply to numbers (numerical ordering) and string values (lexicographical ordering).

In addition, you can use the following operators on the indicated types:

- `a startswith b` (true if string `a` starts with string `b`)
- `a endswith b` (true if string `a` ends with string `b`)
- `a onlist b` (true if string `a` is on word list `b`)
- `a->b` or `a in b` (true if element or value `a` is contained in set `b`)

Examples of condition expressions are:

- visibility!='public'
- (NumOps+NumAttr)<=40
- (NumOps+NumAtrt) le 40
- name startswith 'get'
- 'getID' in OperationNameSet

8.5.4.2 Boolean Functions

Boolean functions operating on strings:

- `startswithcapital(s)` - true if string s starts with a capital letter or is empty
- `startswithlowercase(s)` - true if string s starts with a lower case letter or is empty
- `islowercase(s)` - true if all letters in string s are lower case or if s is empty

Boolean functions operating on sets:

- `isunique(s)` - true if s is a regular set or if s is a multiset that contains no element more than once.

Boolean functions operating on model elements:

- `instanceof(e,s)` - true if the type of model element e is s or a subtype of s. For example, instanceof(context, 'package') is true if the owner of the element is a package or a subtype of package.

© 2002-2013. All rights reserved.

8.5.4.3 Logical Operators

Logical operators can be used to combine condition expressions. The logical operators allowed are

- A & B (or A `and` B to avoid conflict with the XML special character &, see Section 8.5.5 "Expression Terms and XML") - true if both condition expressions A and B are true, else false.
- A | B - true if at least one condition expression A or B is true, else false.
- !A - true if A is not true, else false.

As usual, ! has precedence over & which has precedence over |.

An example of a condition expression with logical operators is: (NumOps>5) & (!(NumAttr=3 | islowercase(name))

8.5.5 Expression Terms and XML

Expression terms are always defined as values of XML attributes. Expression terms use characters that have special meanings in the XML:

- Single quotes for string constants.
  In the XML, attribute values can be delimited by double quotes (") or single quotes ('). If an expression contains string constants, it must be delimited by double quotes in the metrics definition file (e.g. `condition="visibility='public'"`, not `condition='visibility='public''`). Otherwise, XML parsers will report an error.

- operators &, <, and >.
  The and operator & is an escape character in the XML. Likewise, the operators < and > are used in the XML to delimit XML tags. Some XML parsers will report an error if these characters are used in XML attributes. To remedy this, you have two options:
  - Denote the characters by their escape sequence, if your XML editor does not automatically translate the characters for you:
    & - &
    < - <
    > - >
  - Use the alternative notations:
    `and` instead of &
    `lt` and `le` instead of < and <=
    `gt` and `ge` instead of > and >=
    `in` instead of ->

8.6 Writing Descriptions

The measurement catalog (see Section 4.13 "The View 'Catalog'") provides detailed descriptions of the design metrics, rules, and relation matrices, complete with literature references and a glossary. These descriptions are provided by the definitions in the metric definition file.

© 2002-2013. All rights reserved.

Below is an example of a full-fledged metric description. Descriptions of rules and matrices follow the exact same scheme.

```
<metric name="DIT" domain="class">
<description>The depth of the class in the inheritance tree.((p))
This is calculated as the longest glossary://Path/path/ from the class to
the root of the inheritance tree. The DIT for a class that has no
parents is 0.((p))
((ul))
((li))Defined in ref://CK94/.
((li))See also metric://class/CLD/.
((/ul))
</description>
<projection relation="genchild" target="generalization" element="genparent"
 eltype="class" nesting="true"/>
</metric>

<reference tag="CK94">S. Chidamber, C. Kemerer, "A Metrics Suite
  For Object Oriented Design", IEEE Trans. Software Eng., vol. 20, no. 6,
  pp. 476-493, 1994.
</reference>

<term name="Path">A sequence of adjacent nodes in a graph. The
  sequence of nodes n(1), n(2), ..., n(m) is a path if there is an
  edge from n(i) to n(i+1) for all i from 1 to m-1.
<term>
```

Below we explain the various features available for writing descriptions.

HTML markup

The detailed description is meant to be rendered in HTML. Therefore, you can use HTML markups to format your text. However, because the description is embedded in an XML file, you cannot use the plain HTML tags. They would interfere with the XML structure. Instead of enclosing your HTML tags with the usual < and > brackets, use two subsequent opening and closing parentheses, like so: `Text in ((i))italics((/i)) or ((b))boldface((/b)).`

The example for metric DIT above uses paragraphs and bulleted lists to format the text.

Brief description and full description

The first sentence of the description should be a short definition that states the basic idea what the metric (or rule or matrix) is about. It should be brief enough to fit in a single line (about 80 characters, but that is no hard limit).

This first sentence must be terminated by a period, and must not use any HTML markup. The brief descriptions are used in various windows of SDMetrics' GUI, to provide short explanations of metrics, rules, and matrices. The description then continues with additional explanations, and notes about the metric. Together with the brief description, this constitutes the full description shown in the measurement catalog.

Cross-references

© 2002-2013. All rights reserved.

You can reference other metrics, rules, or matrices in the description.

- To cross-reference a metric, use a "metric locator" of the form `metric://<domain>/<name>/`. For example, to cross-reference a metric `NumOps` for elements of type `class`, write `metric://class/NumOps/`. Note the terminating slash at the end of the locator.
- Likewise, to cross-reference a rule, use a locator of the form `rule://<domain>/<name>/`. For example, rule `CyclicInheritance` for classes is referenced by `rule://class/CyclicInheritance/`.
- To cross-reference a relation matrix, use a locator of the form `matrix://<matrix>/`. A matrix named "Associations" would be referenced by `matrix://Associations/`.

In the measurement catalog display, the locator will be replaced by an HTML hyperlink that takes the user to the specified target.

Literature references

To provide literature references, you define a bibliographic citation as in the above example for metric DIT. The citation is enclosed in a `reference` XML element. The element has one required attribute, `tag`, which provides a handle for the citation.

You can then reference the citation in your metric, rule, and matrix descriptions with a locator of the form `ref://<reference_tag>/`, for example `((li))Suggested in ref://CK94/. ((/li))`. In the measurement catalog display, the locator will be replaced by a hyperlink that takes the user to the bibliographic citation of the specified reference.

Glossary terms

If your description uses any terms that you feel need additional explanation, you can define these terms in a glossary. The above example for metric DIT shows a definition for the term "Path".

The term definition is enclosed in a `term` XML element. The element has one required attribute, `name`, which is the term that is defined. You can then use the term in your description with a locator as follows:

```
glossary://<Nameofterm>/<hyperlinktext>/
```

In the full description as shown to the user, the locator will be replaced by a hyperlink that takes the user to the definition of the term. If the name of the glossary term and the hyperlinked text to be shown are identical, you can leave the hyperlinked text empty, like so: `This is a glossary://WFR// of the UML` (note the two slashes after "WFR").

© 2002-2013. All rights reserved.

8.7 Defining Metrics for Profiles

8.7.1 Profiles in UML 2

UML2 profiles provide a mechanism to extend and adapt the UML metamodel for particular domains, platforms, or methods. A UML profile defines a set of stereotypes, which extend existing UML meta-classes by adding new properties (via "tagged values"), imposing additional constraints, and providing alternative graphical representations for the extended elements.

The profile mechanism is a lightweight extension mechanism; it does not allow for modifying existing meta-classes or creating new, "first-class citizen" meta-classes in the UML metamodel. UML profiles can, in theory, be easily imported and exchanged between UML tools, and dynamically applied, combined, and retracted from a UML model.

The most prominent example of a UML profile is probably the Systems Modeling Language (SysML), maintained by the Object Management Group [OMG10]. SysML is an extension of the UML to better support systems engineering. Other profiles maintained by the OMG include MARTE (Modeling and Analysis of Real-time and Embedded Systems), UTP (UML Testing Profile), and SoaML (Service oriented architecture Modeling Language).

How are UML profiles relevant to design quality measurement?

If you extensively use profiles in your models, it is useful to define design metrics and rules that take profile extensions into account. A SysML model, for example, contains elements such "blocks", "requirements", or "test cases", and relationships between them such as "allocate", "refine", or "verify", all defined by the SysML profile. As a SysML user, you may therefore be interested in the number of blocks and requirements in the packages, measure the size of blocks in terms of their part-, reference-, and value-properties, define rules such as "requirements that are not further decomposed or refined must be verified by a test case", or create a relation matrix showing the allocation of activities to blocks.

8.7.2 Profiles in SDMetrics

SDMetrics uses a simplified version of the UML metamodel, and is not a full-fledged MOF implementation. SDMetrics therefore cannot simply import any existing profiles and apply them to the models. Unlike the UML's lightweight approach to metamodel extensions, SDMetrics takes a heavyweight approach of creating new meta-classes in the metamodel. To account for a profile extension requires

- an extension of the SDMetrics metamodel
- XMI transformations for the metamodel extensions
- definition of custom metrics and rules that take the extensions into account.

Creating these definitions is a manual process, but it only has to be done once for each profile. After that, you can analyze models using the profile just like regular models. In the remainder of this section we'll see how profile extensions are serialized to XMI, and discuss the options for developing profile extensions for SDMetrics. Using the example of SysML requirements, we show

© 2002-2013. All rights reserved.

how to add a "requirement" class to SDMetrics' metamodel, define an XMI transformation for it, and create metrics for requirements.

8.7.3 XMI Serialization of Profile Extensions

To facilitate the tracing of system requirements, the SysML defines requirement diagrams to represent the textual requirements of a system. Formally, a SysML requirement is a stereotype that extends meta-class "class", and defines two new properties "Id" and "Text" to capture a human-readable ID and summary text of the requirement.

The following extract of an XMI file shows a SysML model with a package named "Specifications". The package contains a requirement "XYZ" with Id "R1" and text "The system shall do XYZ":

```
<xmi:XMI xmi:version="2.1" ...>
  <uml:Model name = "Sample" xmi:id = "xmi1">
    <packagedElement xmi:type="uml:Package" xmi:id="xmi2" name="Specifications">
      <packagedElement xmi:type="uml:Class" xmi:id="xmi3" name="XYZ" />
    </packagedElement>
    ...
  </uml:Model>
  <sysml:Requirement base_Class="xmi3" xmi:id="xmi7"
    Text="The system shall do XYZ" Id="R1" />
</xmi:XMI>
```

As a consequence of the UML's lightweight extension approach, two XML elements are needed to represent the requirement:

- A regular UML Class with the name "XYZ". This element represents the requirement in the model.
- A SysML Requirement that references class "XYZ" via the attribute "base_Class", and provides the values for the new properties "Id" and "Text". This element represents the application of the "Requirement" stereotype to class "XYZ". Note that the element is outside the model.

SDMetrics offers three options to deal with such XMI files:

- profile extensions with regular model elements
- extension references without inheritance
- extension references with inheritance

8.7.4 Profile Extensions with Regular Model Elements

In our first and simplest approximation, we just define a new model element type "requirement" in our SDMetrics metamodel (see Section 7.1 "SDMetrics Metamodel") to store the information from the `<sysml:Requirement>` XML element:

```
<modelelement name="requirement">
  <attribute name="base" type="ref">The extended class.</attribute>
  <attribute name="text">Text of the requirement.</attribute>
  <attribute name="reqid">ID of the requirement in the model.</attribute>
</modelelement>
```

© 2002-2013. All rights reserved.

The XMI transformation to process the `<sysml:Requirement>` XML element looks like this:

```
<xmitransformation modelelement="requirement" xmipattern="sysml:Requirement">
  <trigger name="base" type="attrval" attr="base_Class" />
  <trigger name="text" type="attrval" attr="Text" />
  <trigger name="reqid" type="attrval" attr="Id" />
</xmitransformation>
```

With these definitions in place, the SysML requirement of the SysML model is represented by two elements in SDMetrics: one "class" element and one "requirement" element.

How do we know if a class in the model represents a SysML requirement? The class represents a SysML requirement if there is a "requirement" element in the model whose reference attribute "base" points to the class. Using the "filtervalue" procedure (Section 8.1.7 "Value Filter"), we can define a "helper" metric that yields the requirement element extending the class, if one exists:

```
<metric name="Reqmt" domain="class" internal="true">
  <filtervalue relation="base" target="requirement"  />
</metric>
```

We can then use this metric to identify classes that represent requirements. For example, to count the number of requirements in a package, we count all classes where metric "Reqmt" is not empty:

```
<metric name="ReqCount" domain="package">
  <projection relation="context" target="class" condition="Reqmt!=''" />
</metric>
```

8.7.5 Extension References without Inheritance

SysML requirements are often decomposed into sub-requirements that partition the containing requirement. The number of sub-requirements of a requirement is therefore a potentially useful size measure.

The following XMI serialization of a SysML model contains a requirement "ABC" with Id "R2". The requirement is decomposed into two sub-requirements "A" and "B" with Ids "R2.1" and "R2.2", respectively:

© 2002-2013. All rights reserved.

```
<xmi:XMI xmi:version="2.1" ...>
  <uml:Model name = "Sample" xmi:id = "xmi1">
    <packagedElement xmi:type="uml:Package" xmi:id="xmi2" name="Specifications">
      <packagedElement xmi:type="uml:Class" xmi:id="xmi3" name="XYZ" />
      <packagedElement xmi:type="uml:Class" xmi:id="xmi4" name="ABC">
        <nestedClassifier xmi:type="uml:Class" xmi:id="xmi5" name="A" />
        <nestedClassifier xmi:type="uml:Class" xmi:id="xmi6" name="B" />
      </packagedElement>
    </packagedElement>
    ...
  </uml:Model>
  <sysml:Requirement base_Class="xmi3" xmi:id="xmi7"
    Text="The system shall do XYZ" Id="R1" />
  <sysml:Requirement base_Class="xmi4" xmi:id="xmi8"
    Text="The system shall do such and such" Id="R2" />
  <sysml:Requirement base_Class="xmi5" xmi:id="xmi9"
    Text="The system shall do A" Id="R2.1" />
  <sysml:Requirement base_Class="xmi6" xmi:id="xmi10"
    Text="The system shall do B" Id="R2.2" />
</xmi:XMI>
```

From the XMI serialization we can see that the SysML uses class nesting to model requirement decomposition. The class to represent the decomposed requirement owns the classes representing the sub-requirements as nested classifiers. We could therefore define a metric counting the sub-requirements like so:

```
<metric name="SubRequirements" domain="class">
  <projection relset="nestedclassifiers" target="class" condition="Reqmt!=''" />
</metric>
```

Defining the metric with the domain "class" has the disadvantage that the metric is calculated for all classes, even those that do not represent requirements at all. A better solution is to define the metric for the domain "requirement":

```
<metric name="SubReqCount" domain="requirement">
  <projection relset="base.nestedclassifiers" target="class"
              condition="Reqmt!=''" />
</metric>
```

The output of this metric in the metric data tables is not very satisfactory, though:

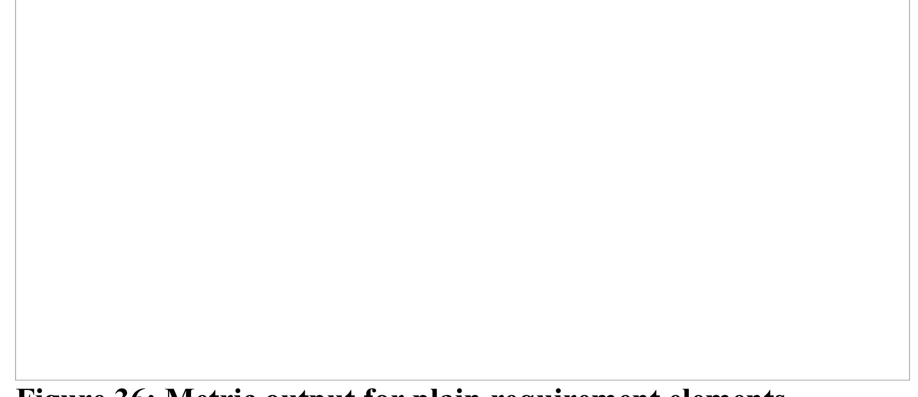


Figure 36: Metric output for plain requirement elements

© 2002-2013. All rights reserved.

Because the requirement elements in the model are unnamed and outside the scope of the UML model, we cannot tell from the metric output to which requirement the data belongs.

To improve this situation, SDMetrics' metamodel facility provides *extension references*. If the extending model element contains an extension reference, SDMetrics moves the extending model element into the same namespace as the extended model element, and copies the name from the extended element.

To use this feature for our SysML requirements, we change the type of cross-reference "base" from "ref" to "extref", thus declaring it an extension reference:

```
<modelelement name="requirement">
  <attribute name="base" type="extref" />
  <attribute name="text" />
  <attribute name="reqid" />
</modelelement>
```

This instructs SDMetrics to move the requirement model elements into the same namespace as the class they extend, and copy the name of the extended class. Each requirement element thus has the same owner and the same name as the class that represents the requirement. With this modification, the extending elements are easier to handle.

One immediate benefit of extension references is that the elements are easier to trace in the metric data tables:

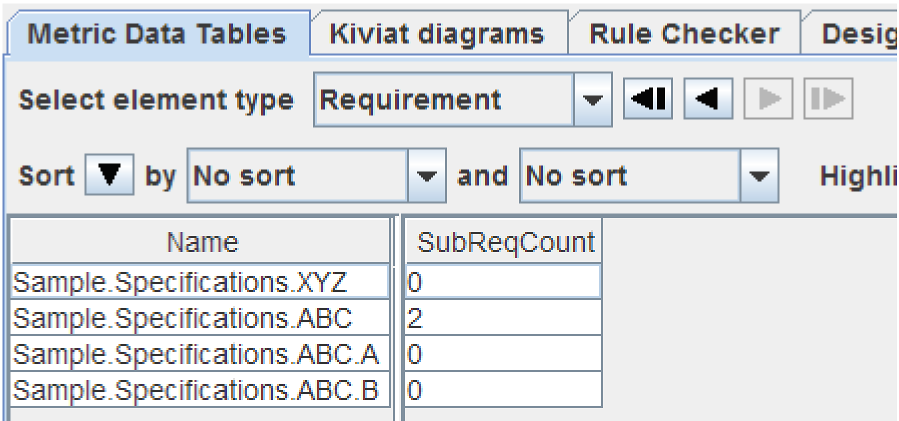


Figure 37: Metric output for requirements with extension reference

Another benefit is that we can simplify some metric and rule definitions. Take for example the metric "ReqCount" defined earlier to count the number of requirements in a package. Because the packages now own the requirement elements, we can define the metric in a more straightforward manner without resorting to 'helper' metrics:

```
<metric name="ReqCount" domain="package">
  <projection relation="context" target="requirement" />
</metric>
```

© 2002-2013. All rights reserved.

8.7.6 Extension References with Inheritance

For the purpose of defining design metrics and rules, it is still cumbersome to have two model elements in the model to logically represent a single element. Consider for example metric "NumCls" for packages to count the number of classes in the package. This is the definition of the metric that ships with SDMetrics ever since its first release in 2002:

```
<metric name="NumCls" domain="package">
  <projection relation="context" target="class" />
</metric>
```

Applied to our sample package with four requirements, this metric will include the class elements to represent the requirements in its class count. Depending on our measurement goals, this may not be what we want. We could of course adjust the definition of the metric to only count classes that do not represent requirements:

```
<metric name="NumClsExceptRequirements" domain="package">
  <projection relation="context" target="class" condition="Reqmt=''"/>
</metric>
```

This approach is possible, but ugly. If we add support for SysML blocks or other types that also extend "class", we would need to adjust the definition of metric again to exclude classes representing blocks, etc.

To better deal with these issues, SDMetrics provides a mechanism to merge the extended element with its extending element to form a single model element that combines all the information from both elements. To enable this mechanism, we have to define the type of the extending element as a subtype of the extended element.

In our SysML requirements example, we therefore change the definition of type "requirement" by declaring its parent to be type "class":

```
<modelelement name="requirement" parent="class">
  <attribute name="base" type="extref" />
  <attribute name="text" />
  <attribute name="reqid" />
</modelelement>
```

If SDMetrics finds a model element of type *t* ("requirement" in our example) with an extension reference, and the type of the extended element is a supertype of *t* ("class" in our example), then SDMetrics will merge the two elements into one element of type *t*. The merged element copies all attribute values of the extended element, and the extended element is removed from the model. We are left with one element of type *t* that assumes the place of the extended element.

Applied to our sample model and XMI file, we end up with a model that contains four requirement elements, no classes, and maintains the expected containment hierarchy:

© 2002-2013. All rights reserved.

```
Model "Sample"
  Package "Specifications"
    Requirement "XZY" with Id "R1"
    Requirement "ABC" with Id "R2"
      Requirement "A" with Id "R2.1"
      Requirement "B" with Id "R2.2"
```

The package "Specification" no longer contains elements of type "class", and metric NumCls will yield 0. Model element type "requirement" has become a full, first-class citizen. We can identify requirements simply by the type of the model element, the need for helper metrics is eliminated.

Using reference extensions with inheritance, it is very easy to define metrics that take more complex relationships into account. For example, the SysML defines a relationship "satisfy" to identify the model elements that fulfill a given requirement. The more elements are needed to satisfy a requirement, the more difficult the requirement will be to change. We are therefore going to define a metric that counts the number of elements that satisfy a requirement.

Technically, the SysML defines the "satisfy" relationship as a stereotype that extends the meta-class "abstraction". We therefore create a new model element type "satisfy" with an extension reference as a subtype of "abstraction" (see Appendix A.2 "Metamodel for UML 2.x"):

```
<modelelement name="satisfy" parent="abstraction">
  <attribute name="base" type="extref" />
</modelelement>
```

The type "satisfy" inherits the multi-valued reference attributes "client" and "supplier" from "abstraction". For the XMI transformation, we only have to provide a trigger for the new extension reference "base":

```
<xmitransformation modelelement="satisfy" xmipattern="sysml:Satisfy">
  <trigger name="base" type="attrval" attr="base_Abstraction" />
</xmitransformation>
```

We can then define our metric to count the number of elements that satisfy a requirement:

```
<metric name="Satisfiers" domain="requirement" category="Coupling (export)">
  <description>The number of elements that satisfy this
requirement.</description>
  <projection relation="supplier" target="satisfy" sum="size(client)"/>
</metric>
```

8.7.7 Tips on Creating Metrics and Rules for Profile Extensions

We have discussed three options to deal with stereotypes in UML profiles:

- option 1: using regular model elements
- option 2: using extension references without inheritance
- option 3: using extension references with inheritance

Here are some guidelines to decide which option to use for a given stereotype.

- How often do you need to distinguish elements to which the stereotype is applied from elements of the same type to which the stereotype is not applied?

  If, for example, the distinction between SysML FlowPorts and plain UML ports is only relevant for a few of your metrics and rules, and you can treat them equally as plain UML ports most of the time, option 1 or 2 is appropriate.

- Do you need to identify the extension elements in the metric, rule, or relation matrix data tables?

  If so, you have to choose option 2 or 3.

- How much does the stereotype application change the "character" or "nature" of the extended model element?

  In case of SysML requirements, for example, the characteristic of the "class" element changes considerably. Unlike regular classes, requirements can't have operations or attributes, do not participate in associations, can't be generalized, and are usually not instantiated. They are actually not much like classes at all.

  In such situations, option 3 is appropriate, because it eliminates the "class" element representing the requirement from the model, and existing metrics and rules that involve classes will ignore the extended element.

- How important or central is the concept of the extension in the profile?

  Like requirements, a SysML block extends UML meta-class "class". Unlike requirements, however, SysML blocks maintain most of the class features from UML. Yet, as blocks are the central modular structure units of the SysML, the set of metrics and rules to calculate for blocks will probably be quite different from the set of class metrics and rules. Option 3 is therefore the best choice for the central concepts of the profile.

- How many meta-classes does the stereotype extend?

  If the stereotype extends two or more meta-classes (for example, SysML TestCase can extend "operation" or "behavior"), you can use option 3 for at most one of the extension, and must use option 1 or 2 for all others. This is because SDMetrics' metamodeling facility does not support multiple inheritance.

Using extension references with inheritance, the inheritance mechanism plays a more important role in the SDMetrics metamodel. With the support for profiles, a number of features were introduced to provide more flexibility dealing with inheritance:

- inheritance of metric, set, and rule definitions to pass selected metrics, sets, and rules on to descended types (see e.g. Section 8.1 "Definition of Metrics" for metric inheritance)
- metrics or sets involving recursion or nesting extend their recursion to compatible super- and subclasses (see e.g. Section 8.1.1.8 "Attribute "recurse"")

© 2002-2013. All rights reserved.

- filtering for subtypes with the "target" and "eltype" filter attributes (see Section 8.1.1.3 "Filter Attribute "target"").
- new function "instanceof" (see Section 8.5.4.2 "Boolean Functions")

Where to go from here

On the SDMetrics web site, you can download a set of project files for SysML 1.2, with metrics and rules that cover block definition diagrams, internal block diagrams, parametric diagrams, and requirement diagrams.

© 2002-2013. All rights reserved.

9 Extending the Metrics and Rule Engine

The previous section showed how to use the XML-based SDMetrics Markup Language (SDMetricsML) to calculate new metrics and rules of your own. The SDMetricsML is flexible enough to cover most measurement needs. Defining new metrics is quick and easy to do, and only requires a simple text editor.

The SDMetricsML definitions are read by the "metrics engine", a subsystem of SDMetrics that interprets the metric definitions and calculates the metrics for a UML model accordingly. Likewise, the "rule engine" checks the design rules defined through SDMetricsML for a UML model.

Occasionally, the capabilities of the SDMetricsML/metrics/rule engine may be insufficient or inefficient for your measurement needs. For these situations, the metrics engine and rule engine provide a plugin mechanism to extend their calculation capabilities and the SDMetricsML. The plugins add

- new metric, set, or rule checking procedures that implement new or special purpose algorithms,
- new scalar, set, or Boolean functions that implement new operations on numbers, strings, or sets.

Users can then write SDMetricsML definitions as usual, using the new procedures and functions.

This section describes how to develop such plugins for the metrics and rule engine. The plugins are implemented in the Java programming language. Writing a plugin thus requires Java skills and a Java compiler.

9.1 Metric Procedures

9.1.1 Conception of a New Metric Procedure

Assume we wish to calculate a "lack of cohesion" style metric similar to LCOM proposed by Chidamber and Kemerer in [CK94]. These metrics count pairs of elements that have or have not something in common, for instance, the number of pairs of methods in a class that use no common attributes, or, with regards to applicability to UML designs, the number of pairs of operations of a class that do not have at least one parameter type in common. In the latter case, just obtaining the set of parameter types of an operation is easy enough:

```
<set name="ParaTypeSet" domain="operation">
<description>The set of parameter types of an operation.</description>
<projection relation="context" target="parameter" element="parametertype" />
</set>
```

What we would need then is a metric that takes each pair of operations of a class, and counts those pairs that have a parameter type in common. None of the existing metric procedures in SDMetrics accomplish this, but if we had such a procedure, the definition of the metric could look something like this:

© 2002-2013. All rights reserved.

```
<metric name="LCOM_Parametertypes" domain="class">
<description>Lack of cohesion in operations based on operation
   parameter types.</description>
<pairwise relation="context" target="operation"
   paircondition="size(_first.ParaTypeSet * _second.ParaTypeSet)=0" />
</metric>
```

The idea is that the metric obtains the set of operations of a class (as we would in a standard projection), and then evaluate the "paircondition" expression for each operation pair. The variables "_first" and "_second" in the pair condition refer to the first and second operation of the pair. The given expression evaluates to true if the intersection of the parameter types of the operations at hand is empty.

To make the procedure more flexible, we could have it support a few more attributes:

- Users should be able to specify the set to operate on as they are used to from the "projection" procedure (Section 8.1.1 "Projection"), using either the "relation" or "relset" attribute, and the standard filter attributes "target", "element", etc.
- The procedure should provide an option to not only yield pairs of elements of a set, but tuples. For pairs, only one combination of elements `A` and `B` will be reported as either `{A,B}` or `{B,A}`, the order of elements carrying no semantics. For tuples, both combinations `(A,B)` and `(B,A)` will be reported, distinguishing the order of elements. We therefore define an optional attribute "tuples" that takes values "true" or "false" to indicate if we want to access pairs (value "false", also the default) or tuples (value "true").
- The procedure should optionally report the combination of an element with itself. We therefore define an optional attribute "withself" that takes values "true" or "false". When set to "true", the procedure additionally reports one combination `{A,A}` for each model element.
- In addition to just counting the number of pairs/tuples that satisfy the pair condition, the procedure should support the "sum" and "stat" attributes as we know them from the "projection" and "subelements" procedure (see Section 8.1.1.7 "Attributes "sum" and "stat""). When "sum" is set, the specified expression will be evaluated for each reported pair, and the metric will report the sum of all values (or the maximum/minimum value when attribute "stat" is set accordingly).
- The "paircondition" attribute should be optional. If not set, all reported combinations will be processed.

9.1.2 Implementation of the Metric Procedure

The following listing shows the complete implementation of the new metric procedure outlined in the previous section. Because we heavily utilize the API of the metrics engine, which already provides much of the functionality, the implementation of the procedure is quite compact, and essentially contains a nested loop to generate the element pairs.

```
packacke com.acme;
import java.util.Collection;
import java.util.Comparator;
import com.sdmetrics.math.ExpressionNode;
import com.sdmetrics.metrics.*;
import com.sdmetrics.model.ModelElement;
```

© 2002-2013. All rights reserved.

```
01 public class MetricProcedurePairwise extends MetricProcedure {

   @Override
02 public Object calculate(ModelElement element, Metric metric)
         throws SDMetricsException {
03    ProcedureAttributes attributes = metric.getAttributes();

04    Variables vars = new Variables(element);
05    Collection<ModelElement> set =
         getRelationOrSet(element, attributes, vars);
06    if (set == null)
07       return Integer.valueOf(0);

08    ExpressionNode pairCondition = attributes.getExpression("paircondition");
09    boolean allTuples = attributes.getBooleanValue("tuples", false);
10    boolean withSelf = attributes.getBooleanValue("withself", allTuples);

11    FilterAttributeProcessor fap = getFilterAttributeProcessor(attributes);
12    SummationHelper sum = new SummationHelper(getMetricsEngine(), attributes);
13    Comparator<ModelElement> comparator = ModelElement.getComparator();

14    for (ModelElement first : fap.validIteration(set, vars)) {
15       vars.setVariable("_first", first);
16       for (ModelElement second : fap.validIteration(set, vars)) {
17          int comp = comparator.compare(first, second);
18          if (comp == 0 && !withSelf)
19             continue;
20          if (comp > 0 && !allTuples)
21             continue;

22          vars.setVariable("_second", second);
23          if (pairCondition == null
                || evalBooleanExpression(element, pairCondition, vars)) {
24             sum.add(second, vars);
            }
         }
      }
25    return sum.getTotal();
   }
}
```

Let's go through the salient points of this implementation line by line:

- 01: All metric procedure classes must have public visibility, a default or no-argument constructor, and extend the abstract class `com.sdmetrics.metrics.MetricProcedure`.
- 02: The base class defines the abstract method `calculate` which we must override. Input parameters are the model element and the metric to be calculated.
- 03: Class `ProcedureAttributes` provides access to the attribute values in the metric definition.
- 04: Class `Variables` contains the values of variables to be used in metric expressions. In the constructor, we specify the principal model element for which the metric is calculated.
- 05: Method `getRelationOrSet` is a helper method provided by the base class to evaluate the "relation" or "relset" attributes (whichever was specified) as we know them from metric and set projections.

© 2002-2013. All rights reserved.

- 08: Class `ExpressionNode` represents metric, set, or condition expressions (see Section 8.5 "Expression Terms"). Here, we obtain the condition expression of the "paircondition" that we defined.
- 09-10: Here we obtain the values of attributes "tuples" and "withself", providing default values when the attributes are not set.
- 11: Class `FilterAttributeProcessor` is a helper class to process the standard filter attributes ("target", "targetcondition", "element", "eltype", "condition", and "scope"). The method `getFilterAttributeProcessor()` yields an instance of this class to apply the filter attributes for the metric at hand.
- 12: Class `SummationHelper` is a helper class that processes the "sum" and "stat" attributes.
- 14: We use the filter attribute processor from line 11 to iterate over all elements specified via the "relation" or "relset" attribute from line 5. The filter attribute processor automatically discards elements which should be ignored as per the element filter settings (see Section 4.2.2 "Specifying Filters") for us, applies the filter attributes "target", "element" etc., and returns an iteration over the resulting elements.
- 15: We define a variable "_first" with the first element of the pair/tuple as value.
- 16-21: In a nested loop, we iterate again over all elements. Depending on the values of attributes "tuples" and "withself", we skip unwanted pairs/tuples.
- 22: At this point we have identified a relevant pair or tuple, and we define a variable "_second" with the second element of the pair/tuple as value.
- 23: We evaluate the "paircondition" (if set), making sure to pass the values of the "_first" and "_second" variables into the evaluation. Method `evalBooleanExpression` evaluates condition expressions for model elements and returns a simple `boolean` value with the result.
- 24: When the "paircondition" yields true (or was not set), we use the summation helper from line 12 to either increase the result value by one or process the "sum" and "stat" attributes if they were specified. Again, we pass the "_first" and "_second" variables to the summation helper so that the values are available for the evaluation of the "sum" attribute.
- 25: We return the total as value of the metric.

9.1.3 Using the New Metric Procedure

Before we can use our new metric procedure, we must first register it with the metrics engine. We add the following XML element to our metric definition file:

```
<metricprocedure name="pairwise"
        class="com.acme.MetricProcedurePairwise" />
```

Attribute `name` defines the name to denote the metric procedure in subsequent metric definitions, attribute `class` defines the fully qualified name of the metric procedure class.

We can then define metrics using our newly created metric procedure. Here's another example of a LCOM-style cohesion metric:

© 2002-2013. All rights reserved.

```
<metric name="LCOC_AttributeTypes" domain="package">
<description>The maximum number of attribute types shared by a pair
   of classes in a package.</description>
<pairwise relation="context" target="class"
   sum="size(_first.AttrTypeSet * _second.AttrTypeSet)" stat="max" />
</metric>
```

For SDMetrics to find our new metric procedure, we have to ensure that the class is included in the class path when we run SDMetrics. The simplest way to achieve this is by putting the class files in the "bin" folder of the SDMetrics installation directory. SDMetrics searches this directory for class files.

Within the "bin" directory, we have to create a structure of subfolders that reflects the packages of the classes we provide. The class file for class `com.acme.MetricProcedurePairwise` therefore will be located at the path `bin/com/acme/MetricProcedurePairwise.class`.

9.2 Set Procedures

Defining set procedures is very similar to defining metric procedures. In the following example, we will implement a "conditional" set procedure that yields one of two possible sets based on a condition. The set procedure takes three required attributes "condition", "set", and "alt". If the "condition" expression evaluates to true, the set procedure returns the result of the "set" expression, otherwise the value of the "alt" expression.

```
   packacke com.acme;
   import java.util.Collection;
   import com.sdmetrics.math.ExpressionNode;
   import com.sdmetrics.metrics.*;
   import com.sdmetrics.model.ModelElement;

01 public class SetProcedureCondition extends SetProcedure {

    @Override
02  public Collection<?> calculate(ModelElement element, Set set)
         throws SDMetricsException {

03    ProcedureAttributes attributes = set.getAttributes();
04    ExpressionNode condexp = attributes.getRequiredExpression("condition");
05    ExpressionNode setExpr = attributes.getRequiredExpression("set");
06    ExpressionNode altExpr = attributes.getRequiredExpression("alt");

07    Variables vars = new Variables(element);
08    boolean condition = evalBooleanExpression(element, condexp, vars);
09    if (condition)
         return evalSetExpression(element, setExpr, vars);

10    return evalSetExpression(element, altExpr, vars);
    }
}
```

Some of the metrics engine API used here we already know from (Section 9.1.2 "Implementation of the Metric Procedure"), so we focus the discussion of this implementation on new features:

- 01: The class must have public visibility, a standard constructor, and extend the abstract class `com.sdmetrics.metrics.SetProcedure`.

© 2002-2013. All rights reserved.

- 02: The base class defines one method `calculate`. Input parameters are the model element and the definition of the set to be calculated (an instance of class `com.sdmetrics.metrics.Set`).
- 04-06: Method `getRequiredExpressionNode` returns the expression of the indicated attribute, or throws an `SDMetricsException` if the attribute was not defined. The message text of the exception is suitable to be read and understood by end users.
- 09-10: Method `evalSetExpression` evaluates set expressions for model elements, and returns a `java.util.Collection` with the set elements.

The return value of a set procedure can be a regular set (instance of `java.util.HashSet`), or a multiset (instance of `com.sdmetrics.math.HashMultiSet`). An "element set" (see Section 8.2 "Definition of Sets") will contain only instances of `ModelElement`. A "value set" will contain instances of `java.lang.Number` (`Integer` or `Float`), or strings. This set procedure can return either value sets or element sets, depending on the types of sets specified via the "set" and "alt" attributes.

For further examples on how to programmatically manipulate regular and multisets, see Section 9.6 "Set Functions".

To use our new set procedure, we register it with the metrics engine as follows:

```
<setprocedure name="conditionalset"
   class="com.acme.SetProcedureCondition" />
```

Again, we deploy the class file of the procedure class in the "bin" folder of our SDMetrics installation (path `com/acme/SetProcedureCondition.class`). After that, we can write set definitions using the new set procedure. For example:

```
<set name="InterestingObjects" domain="class">
<conditionalset condition="'DataStore' in StereoTypeNames" set="AttributeSet"
    alt="OperationSet" />
</set>
```

Note: every set procedure can also be used to define relation matrices (cf. Section 8.4 "Definition of Relation Matrices"). This includes custom set procedures you defined yourself.

9.3 Rule Procedures

To illustrate the definition of custom rule procedures, consider the following scenario. We wish to implement a rule that checks adherence to a particular naming convention. The naming convention states that the names of certain elements (such as actions) must be constructed according to the pattern "<verb> <object>", for example "create account" or "update customer". Both verbs and objects must be chosen from a predefined list (maybe "create, read, update, delete, validate" for verbs, and "account, customer, credit, debit" for objects). The rule should check that the name of each action adheres to this pattern and uses only verbs and objects from the approved list.

To define the contents of the approved list of verbs and objects, we will use word lists (see Section 8.3.5 "Word lists"). The definition of the rule including the word lists could look something like this:

© 2002-2013. All rights reserved.

```
<wordlist name="VerbList">
        <entry word="create"/>
        <entry word="read"/>
        <entry word="update"/>
        <entry word="delete"/>
</wordlist>

<wordlist name="ObjectList">
        <entry word="bank account"/>
        <entry word="customer"/>
        <entry word="credit"/>
</wordlist>

<rule name="ActionNames" domain="action">
   <description>Checks that action names contain only approved
      verbs and objects.</description>
   <verbobject
      term="name"
      condition="!(_verb onlist VerbList) or !(_object onlist ObjectList)"
      value="'Illegal name: '+name" />
</rule>
```

Attribute `term` is a metric expression that yields the string to be checked. The attribute can be optional, when omitted, the procedure will check the name of the model element.

The procedure then has to extract the verb and object part from the result of the `term` expression, and pass the values as variables `_verb` and `_object` to the `condition` expression. If the expression evaluates to true, the procedure reports a violation, the value of the violation, as usual, given by the `value` expression.

Here is a possible implementation of such a rule procedure:

```
import com.sdmetrics.math.ExpressionNode;
import com.sdmetrics.metrics.*;
import com.sdmetrics.model.ModelElement;

01 public class RuleProcedureVerbObject extends RuleProcedure {

   @Override
02 public void checkRule(ModelElement element, Rule rule) throws
SDMetricsException {

03    String name = element.getName();

04    ProcedureAttributes attributes = rule.getAttributes();
05    ExpressionNode term = attributes.getExpression("term");
06    Variables vars = new Variables(element);
07    if (term != null)
08       name = evalExpression(element, term, vars).toString();

09    int boundary = name.indexOf(' ');
10    if (boundary < 0) {
11       reportViolation(element, rule, "Element does not specify an object.");
12       return;
      }
```

© 2002-2013. All rights reserved.

```
13    String verb = name.substring(0, boundary);
14    String object = name.substring(boundary + 1);
15    vars.setVariable("_verb", verb);
16    vars.setVariable("_object", object);

17    ExpressionNode condition = attributes.getRequiredExpression("condition");
18    if (evalBooleanExpression(element, condition, vars)) {
19       Object value = getRuleValue(element, attributes, vars);
20       reportViolation(element, rule, value);
      }
   }
}
```

The example illustrates a number of new features of the metrics and rule engine API:

- 01: All rule procedure classes must have public visibility, a default or no-argument constructor, and extend the abstract class `com.sdmetrics.metrics.RuleProcedure`.
- 02: The base class defines the abstract method `checkRule` which we must override. Input parameters are the model element and the rule to check.
- 08: Here we evaluate the "term" expression, if defined. Method `evalExpression` evaluates a metric expression for a model element and returns its value. Metric expressions usually produce numbers (Float or Integer), strings, or model elements. In our example, the "term" expression should return a string.
- 11: We use method `reportViolation` when we detect the violation of a rule. We report the violated element, the violated rule, and a value describing the nature of the violation.
- 13-16: Here we extract the "verb" part and the "object" part of the term, and store them as variables "_verb" and "_object" for the evaluation of the condition expression in line 18.
- 19: Method `getRuleValue` evaluates the attribute "value" of the rule definition. We use this value to report the rule violation in the following statement.

To register the rule procedure with the rule engine, we deploy the class file of the procedure class in the "bin" folder of our SDMetrics installation (path `com/acme/RuleProcedureVerbObject.class`), and add the following definition to our metric definition file:

```
<ruleprocedure name="verbobject"
        class="com.acme.RuleProcedureVerbObject" />
```

After that, we can define rules using our new rule procedure, such as rule "ActionNames" from above.

9.4 Boolean Functions

9.4.1 Conception of a New Boolean Function

Boolean functions occur in condition expressions (see Section 8.5 "Expression Terms") and yield Boolean (yes/no) values as results of conditions.

© 2002-2013. All rights reserved.

For example, let's assume we have a modeling process that defines a stereotype 'foobar' for classes. Classes of this stereotype must either contain operations, or contain attributes, or specialize some other class. We wish to define a rule that checks this constraint for 'foobar' classes.

Checking each condition individually is simple. We can use SDMetrics' standard class metrics NumOps, NumAttr, and DIT (depth of inheritance tree), and compare their values to 0. However, the condition expressions to assert that exactly one of the conditions holds is cumbersome:

```
(NumOps!=0 and NumAttr=0 and DIT=0) or
(NumOps=0 and NumAttr!=0 and DIT=0) or
(NumOps=0 and NumAttr=0 and DIT!=0)
```

The length of the condition expressions grows with the square of the number of individual conditions to check. The efficiency of evaluating this condition expressions is suboptimal, as each individual condition may be evaluated several times.

A Boolean function that calculates the "exclusive or" (XOR) for any number of conditions would be convenient in this situation. The condition expression could then be boiled down to

```
xor(NumOps!=0, NumAttr!=0, DIT!=0)
```

With this new XOR function, we could define our rule as follows:

```
<rule name="FooBarCondition" domain="class">
<description>Classes with stereotype foobar must either define operations,
  attributes, or specialize another class.</description>
<violation condition=
  "('foobar' in StereotypeNames) and !xor(NumOps!=0, NumAttr!=0, DIT!=0)" />
</rule>
```

The condition term of the rule first checks if the class is of stereotype 'foobar' (we assume value set `StereotypeNames` was defined elsewhere to contain the names of the stereotypes of the class). If the stereotype condition holds, and the XOR function returns `false`, the condition term is `true` and the rule violation will be reported.

9.4.2 Implementation of the Boolean Function

The following listing shows the complete implementation of the XOR function outlined in the previous section. The function successively evaluates the condition expressions passed as arguments to the XOR function. The result of the function is `true` if exactly one of the individual condition expressions is `true`:

```
package com.acme;

import com.sdmetrics.math.ExpressionNode;
import com.sdmetrics.metrics.BooleanOperation;
import com.sdmetrics.metrics.SDMetricsException;
import com.sdmetrics.metrics.Variables;
import com.sdmetrics.model.ModelElement;
```

© 2002-2013. All rights reserved.

```
01 public class BooleanOperationXOR extends BooleanOperation {

   @Override
02 public boolean calculateValue(ModelElement element,
03                               ExpressionNode node,
04                               Variables vars)
05                               throws SDMetricsException {

06   int trueConditions = 0;
07   int index = 0;
08   while (index < node.getOperandCount() && trueConditions <= 1) {
09     if (evalBooleanExpression(element, node.getOperand(index), vars)) {
10       trueConditions++;
       }
11     index++;
     }
12   return trueConditions == 1;
   }
}
```

Let's go through the salient points of this implementation line by line:

- 01: Boolean function classes must have public visibility, a default or no-argument constructor, and extend the abstract class `com.sdmetrics.metrics.BooleanOperation`.
- 02: The base class defines the abstract method `calculateValue` which we must override. Parameter `element` is the model element for which to evaluate the function.
- 03: Parameter `node` contains the operator tree for the function call. The operands of the node represent the arguments passed into the function.
- 04: Parameter `vars` contains the values of the variables defined for the evaluation of the arguments to the function.
- 08: The XOR function in this example has an arbitrary number of arguments. Method `getOperandCount()` obtains the actual number of arguments passed to the function. Other functions may have a fixed number of arguments, in which case we would not need to determine the argument count.
- 09: We use method `evalBooleanExpression` provided by the base class to evaluate the condition expressions that are passed as arguments to the XOR function, one by one. Method `node.getOperand(index)` accesses the arguments to the XOR function by their index. The first argument has index 0, the second argument has index 1, and so on.
- 12: Boolean functions must return a value of type `boolean`.

9.4.3 Using the New Boolean Function

To use our new Boolean function, we must register it with the metrics engine. We add the following XML element to our metric definition file:

```
<booleanoperationdefinition name="xor"
        class="com.acme.BooleanOperationXOR" />
```

Attribute `name` defines the name of the function to be used in condition expressions, attribute `class` defines the fully qualified name of the Boolean function class.

© 2002-2013. All rights reserved.

At runtime, the Boolean function class must be included in the class path of SDMetrics. The simplest way to achieve this is by putting the class files in the "bin" folder of the SDMetrics installation directory. SDMetrics searches this directory for class files.

Within the "bin" directory, we have to create a structure of subfolders that reflects the packages of the classes we provide. The class file for class `com.acme.BooleanOperationXOR` therefore will be located at the path `bin/com/acme/BooleanOperationXOR.class`.

9.5 Scalar Functions

A scalar function is used in metric expressions (see Section 8.5 "Expression Terms") and yields a numerical value, a string value, or a model element. In the following example, we will implement a function that yields the maximum value from a list of numerical values, e.g., in a metric expression like this:

```
max(NumClasses, NumInterfaces, NumUseCases)
```

The following implementation of the MAX function calculates the value of each argument provided to the function, and returns the maximum value:

```
   packacke com.acme;
   import com.sdmetrics.math.ExpressionNode;
   import com.sdmetrics.metrics.MetricTools;
   import com.sdmetrics.metrics.SDMetricsException;
   import com.sdmetrics.metrics.ScalarOperation;
   import com.sdmetrics.metrics.Variables;
   import com.sdmetrics.model.ModelElement;

01 public class ScalarOperationMax extends ScalarOperation {

   @Override
02 public Number calculateValue(ModelElement element, ExpressionNode node,
                         Variables vars) throws SDMetricsException {

03    float result = Float.NEGATIVE_INFINITY;
04    for (int index = 0; index < node.getOperandCount(); index++) {
05      float value = ((Number) evalExpression(element,
          node.getOperand(index), vars)).floatValue();
06      result = Math.max(result, value);
      }
07    return MetricTools.getNumber(result);
   }
}
```

Again, we discuss the implementation line by line:

- 01: The class must have public visibility, a standard constructor, and extend the abstract class `com.sdmetrics.metrics.ScalarOperation`.
- 02: Like Boolean functions (Section 9.4.2 "Implementation of the Boolean Function"), the base class defines an abstract method `calculateValue`, with identical input parameters. The return value of the method is of type `Object`, and provides the value of the scalar function. Since our MAX function deals with numerical values, we have narrowed the return

© 2002-2013. All rights reserved.

type to `Number`. Other acceptable return types are `Integer`, `Float`, `String`, or `ModelElement`.

- 05: Method `evalExpression` evaluates metric expressions. We use it to calculate the values of the arguments passed into the MAX function, one by one. Method `evalExpression` returns instances of `Object`. Since we expect the arguments to our MAX function to be numerical, we cast the result to `Number` and take its float value.
- 07: Static method `MetricTools.getNumber(float)` wraps the given float value into an `Integer` if it is small enough and has no fractional part, or else into a `Float`.

To use the scalar function, we register it with the metrics engine as follows:

```
<scalaroperationdefinition name="max"
   class="com.acme.ScalarOperationMax" />
```

As with Boolean functions, we deploy the class file of the class in the "bin" folder of our SDMetrics installation (path `com/acme/ScalarOperationMax.class`). After that, we can write metric expressions using the new function. For example:

```
<metric name="DominantElementTypeSize" domain="package">
<description>Size of the package in terms of the number elements
  of the most dominant type in the package.</description>
<compoundmetric
  term="max(NumClasses, NumInterfaces, NumDataTypes, NumUseCases)" />
</metric>
```

9.6 Set Functions

A set function is used in set expressions (see Section 8.5 "Expression Terms") and yields an element set or a value set. To illustrate the implementation of set functions, we'll define one to calculate the symmetric difference of two sets.

The symmetric difference of two sets `A` and `B` is the set of elements contained in either `A` or `B`, but not both. For regular sets, we could express the symmetric difference in terms of the existing set operations as `(A+B)-(A*B)` (i.e., the union of the sets without the intersection of the sets, see Section 8.5.3 "Set Expressions"). For multisets, however, we must take the cardinality of elements into account: the cardinality of an element in the symmetric difference is the absolute difference of the cardinality of the element in sets `A` and `B`. For example, if the cardinality of element `e` is five in set `A` and three in set `B`, the cardinality of element `e` in the symmetric difference is two. The formula `(A+B)-(A*B)` would yield cardinality (5+3)-3=5 for element `e` and therefore cannot be used for multisets.

The following implementation handles both regular and multisets.

© 2002-2013. All rights reserved.

```
   packacke com.acme;
   import java.util.Collection;
   import java.util.Iterator;

   import com.sdmetrics.math.ExpressionNode;
   import com.sdmetrics.metrics.MetricTools;
   import com.sdmetrics.metrics.SDMetricsException;
   import com.sdmetrics.metrics.SetOperation;
   import com.sdmetrics.metrics.Variables;
   import com.sdmetrics.model.ModelElement;

01 public class SetOperationSymmDiff extends SetOperation {

   @Override
02 public Collection<?> calculateValue(ModelElement element,
      ExpressionNode node, Variables vars) throws SDMetricsException {

03   Collection<?> left = evalSetExpression(element, node.getOperand(0),
         vars);
04   Collection<?> right = evalSetExpression(element, node.getOperand(1),
         vars);

05   boolean isMultiSet = MetricTools.isMultiSet(right)
         || MetricTools.isMultiSet(left);
06   Collection<?> result = MetricTools.createHashSet(isMultiSet);

     // process elements from the first set
07   Iterator<?> it = MetricTools.getFlatIterator(left);
08   while (it.hasNext()) {
09     processElement(it.next(), result, left, right);
     }

     // process additional elements from the second set
10   it = MetricTools.getFlatIterator(right);
11   while (it.hasNext()) {
12     Object o = it.next();
13     if (!left.contains(o)) {
14       processElement(o, result, left, right);
       }
     }
15   return result;
   }

   @SuppressWarnings({ "unchecked", "rawtypes" })
16 private void processElement(Object o, Collection col,
      Collection<?> left, Collection<?> right) {
17   int leftCount = MetricTools.elementCount(left, o);
18   int rightCount = MetricTools.elementCount(right, o);
19   int count = Math.abs(leftCount - rightCount);
20   for (int i = 0; i < count; i++)
21     col.add(o);
    }
}
```

Once more, we discuss the salient features of this implementation, line by line.

- 01: The class must have public visibility, a standard constructor, and extend the abstract class `com.sdmetrics.metrics.SetOperation`.
- 02: Like Boolean and scalar functions (cf. Section 9.4.2 "Implementation of the Boolean Function"), the base class defines an abstract method `calculateValue`, with identical

© 2002-2013. All rights reserved.

input parameters. The return value of the method is of type `java.util.Collection` and provides the result set of the function. Regular sets must be represented by instances of `java.util.HashSet`, multiset must be instances of `com.sdmetrics.math.HashMultiSet`. An "element set" (see Section 8.2 "Definition of Sets") contains instances of `ModelElement`. A "value set" contains instances of `java.lang.Number` (`Integer` or `Float`), or strings.

- 03-04: Method `evalSetExpression` evaluates set expressions. We use it to calculate the two input sets passed as arguments into the function.
- 05: Class `MetricTools` contains a number of static methods that are useful when dealing with sets that may be either regular or multisets. Method `isMultiSet` checks if a set is a multiset or a regular set.
- 06: Method `MetricTools.createHashSet(boolean)` creates a new, empty regular set or multiset. The Boolean parameter determines the type of set created. In our example, we create a multiset if at least one of the input sets is a multiset.
- 07: Method `MetricTools.getFlatIterator(Collection)` obtains an iterator over the elements in a set that returns each element in the set exactly once, even if the set is a multiset and the cardinality of the element is greater than one.
- 17: Method `MetricTools.elementCount(Collection, Object)` determines the cardinality of an element in a set. For regular sets, the method returns 1 if the element is contained in the set, else 0.
- 20-21: The element is added to the result set, multiple times if necessary to get the cardinality of the element right. This implementation does not check the types of the elements added to the result set. That way, the types of the elements in the input sets determines if the result set is an element set or a value set.

To use the set function, we register it with the metrics engine as follows:

```
<setoperationdefinition name="symmdiff"
   class="com.acme.SetOperationSymmDiff" />
```

Again, we deploy the class file of the class in the "bin" folder of our SDMetrics installation (path `com/acme/SetOperationSymmDiff.class`). After that, we can write set expressions using the new function. For example:

```
<metric name="FooBar" domain="package">
<compoundmetric term="size(symmdiff(FooBarClassesSet, FooBazClassesSet))" />
</metric>
```

9.7 Metrics Engine Extension Guidelines

Before you embark on creating new custom procedures and functions, consider the following guidelines.

General guidelines

- The procedure and function classes must have public visibility, and they must have a public constructor without arguments (usually the default constructor).
- The classes must be stateless. Do not define or use instance fields in these classes. The method implementations must be reentrant.

© 2002-2013. All rights reserved.

- Avoid side effects in the method implementations. For instance, do not modify the intermediate sets obtained from calls to method `evalSetExpression`, as these sets may be cached by the metrics engine and used elsewhere.

Guidelines for metric, set, and rule procedures

- Defining new procedures should rarely be necessary. The primary means of defining new metrics and rules is the SDMetricsML, which is faster to do and only requires a text editor for tool support. Only when the capabilities of the SDMetricsML are absolutely insufficient should you consider defining new procedures.
- A metric, set, or rule procedure does not define a concrete metric, set, or rule. It rather provides a template and algorithm for the definition of a whole family of different metrics/sets/rules. Think of the many different metrics you can define with just the "projection" procedure. Therefore, aim to define the attributes of your procedures to provide as much flexibility as possible. Make sure that all inputs to the procedure can be specified as procedure attributes, and only hardcode default values for optional attributes in the implementation of the procedure.
- When you define the attributes for your new procedure, use established groups of attributes such as the standard filter attributes supported by class `FilterAttributeProcessor`, or the "sum/stat" attributes of class `SummationHelper`. Your new procedures will better blend in and form a cohesive whole with the standard procedures, and the implementation of your procedure will be easier.

Guidelines for boolean, scalar, and set functions

- As with procedures, defining new functions should rarely be necessary.
- The return type of the function determines where the function can be used:
    - Boolean functions return Boolean values and are used in condition expressions.
    - Scalar functions return numbers, strings, or a model elements and are used in metric expressions
    - Set functions return sets (regular or multisets, element or value sets) and are used in set expressions.
- The types of the arguments to the function are independent from the return type of the function. Each function type can have condition expressions, metric expressions, and/or set expressions as arguments. You can mix and match the argument types as needed. The methods `evalExpression`, `evalBooleanExpression`, and `evalSetExpression` are equally provided by all of the function base classes.

Where to go from here

The above examples of custom procedures and functions did not cover the entire metrics and rule engine API. On the SDMetrics web site, you can

- view the JavaDocs of the complete API,
- download the SDMetrics Open Core distribution, which contains the implementations of all the standard metric, set, and rule procedures described in Section 8 "Defining Custom Design Metrics and Rules", and all the functions and operations described in Section 8.5

© 2002-2013. All rights reserved.

"Expression Terms". These implementations provide further usage examples for every feature of the API.

© 2002-2013. All rights reserved.

A: Metamodels

SDMetrics ships with default metamodels for UML1.3/1.4 and UML2.x (see Section 7.1 "SDMetrics Metamodel"). This appendix lists all metamodel elements and a description of their attributes.

A.1 Metamodel for UML 1.3/1.4

All metamodel elements inherit by default from a special metamodel element "`sdmetricsbase`", which defines, amongst others, the attributes "id", "name", and "context" that all model elements must possess (see Section 7.1 "SDMetrics Metamodel"). Table 6 shows these default attributes.

Column "Attributes" provides the name of the attributes, column "Type" indicates if the attribute is a data or cross-reference attribute, column "Mult." indicates the multiplicity of the attribute ("one" for single-valued, "many" for multi-valued attributes).

Attribute	Type	Multi.	Description
context	ref	one	Owner of the element in the UML model.
id	data	one	Unique identifier of the model element.
name	data	one	Name of the element in UML model.
stereotypes	ref	many	The stereotypes of the model element.

Table 6: Default attributes for the UML1.x metamodel

In Table 7, column "Model Element" lists all elements of the UML1.4 metamodel. The remaining columns provide the details about all further attributes the metamodel elements define in addition to the inherited attributes.

Model Element	Attribute	Type	Multi.	Description
class	visibility	data	one	Visibility of the class (public, private, protected, package).
	abstract	data	one	Boolean indicating if class is abstract.
	leaf	data	one	Boolean indicating if a class must not have subclasses.
interface	---	---	---	
datatype	---	---	---	
attribute	visibility	data	one	Visibility of the attribute (public, private, protected, package).
	attributetype	ref	one	Reference to the attribute type (class, data type etc).
	changeability	data	one	Changeability of the attribute (changeable or none, frozen, addOnly).

© 2002-2013. All rights reserved.

Model Element	Attribute	Type	Multi.	Description
operation	visibility	data	one	Visibility of the operation (public, private, protected, package).
	abstract	data	one	Boolean indicating if operation is abstract.
	isquery	data	one	Boolean indicating if operation is a query that does not change the classifier's state.
parameter	kind	data	one	The kind of parameter (in, out, inout, return).
	parametertype	ref	one	Reference to the parameter type (class, data type etc).
method	---	---	---	
model	---	---	---	
package	---	---	---	
subsystem	---	---	---	
association	---	---	---	
associationclass	---	---	---	
associationend	navigable	data	one	Boolean indicating if the association end is navigable.
	aggregation	data	one	Indicates if the association end is an aggregation or composite.
	associationendtype	ref	one	A link to the element attached to the association end.
generalization	genchild	ref	one	Link to the child in the generalization.
	genparent	ref	one	Link to the parent in the generalization.
abstraction	abssupplier	ref	one	Link to the interface.
	absclient	ref	one	Link to the element implementing the interface
dependency	depsupplier	ref	one	Link to the supplier of the dependency.
	depclient	ref	one	Link to the client of the dependency.
usage	depsupplier	ref	one	Link to the supplier of the usage.
	depclient	ref	one	Link to the client of the usage.
object	objtype	ref	one	Link to the element of which this object is an instance.
stimulus	stimsender	ref	one	Sender of the stimulus.
	stimreceiver	ref	one	Receiver of the stimulus.
	stimaction	ref	one	The action that the stimulus dispatches.
link	---	---	---	
linkend	linkendtype	ref	one	The element that is attached to the link end.

© 2002-2013. All rights reserved.

Model Element	Attribute	Type	Multi.	Description
collaboration	---	---	---	
classifierrole	classifierbase	ref	one	The classifier which the classifier role is a view of.
interaction	---	---	---	
message	messagesender	ref	one	The sender of the message.
	messagereceiver	ref	one	The receiver of the message.
	messageaction	ref	one	The action that the message dispatches.
statemachine	---	---	---	
state	entryaction	ref	many	The entry actions of the state.
	exitaction	ref	many	The exit actions of the state.
	doactivity	ref	many	The do actions of the state.
	internaltrans	ref	many	The internal transitions of the state.
	kind	data	one	The kind of state (simple, composite, submachine, stub, synch, final; initial, deepHistory, shallowHistory, join, fork, choice or branch, junction; action, objectflow, call, subactivity).
	isconcurrent	data	one	Indicates if a composite state is concurrent.
transition	transsource	ref	one	The source state of the transition.
	transtarget	ref	one	The target state of the transition.
	trigger	ref	one	The event that triggers the transition.
event	kind	data	one	The kind of event (signal, call, change, time).
	linkedeventelement	ref	one	The element (signal, operation, boolean expression, or deadline) that raised the event.
action	kind	data	one	The kind of action (send, return, create, destroy, call, terminate, uninterpreted).
	linkedactionelement	ref	one	Depending on the kind of action, the operation or signal that is invoked when the action is executed, or the classifier of which an instance is created.
guard	---	---	---	
partition	contents	ref	many	The elements contained in the swimlane.
activitygraph	---	---	---	
signal	---	---	---	
usecase	---	---	---	
actor	---	---	---	

© 2002-2013. All rights reserved.

Model Element	Attribute	Type	Multi.	Description
usecaseextend	usecaseextbase	ref	one	The use case that is extended.
	usecaseextension	ref	one	The use case that is the extension.
	usecaseextensionpoint	ref	one	The extension point in the extended use case.
usecaseinclude	usecaseincbase	ref	one	The including use case.
	usecaseaddition	ref	one	The included use case.
extensionpoint	---	---	---	
component	deploymentlocation	ref	one	The node on which the component is deployed.
componentinstance	componenttype	ref	one	The component that is instantiated.
node	---	---	---	
nodeinstance	nodetype	ref	one	The node that is instantiated.
stereotype	extendedelements	ref	many	The set of elements that the stereotype extends.
taggedvalue	tag	data	one	The tag of the tagged value (up to UML 1.3)
	definition	ref	one	Definition of the tagged value (since UML 1.4)
	value	data	one	The value of the tagged value.
tagdefinition	tagtype	data	one	The tag name of a tagged value's tag.
diagram	type	data	one	The type of diagram (class diagram, sequence diagram, etc).
diagramelement	element	ref	one	The element that is shown on the diagram.

Table 7: SDMetrics metamodel elements and attributes for UML1.x models

A.2 Metamodel for UML 2.x

As with the UML1.x metamodel, we first describe the default attributes defined by the "`sdmetricsbase`" model element.

Attribute	Type	Multi.	Description
context	ref	one	Owner of the element in the UML model.
id	data	one	Unique identifier of the model element.
name	data	one	Name of the element in UML model.
comments	ref	many	The comments for the model element.

Table 8: Default attributes for the UML2.x metamodel

Table 9 lists all elements of the UML2.x metamodel, and the attributes they define in addition to those inherited from the parent.

© 2002-2013. All rights reserved.

Model Element	Attribute	Type	Multi.	Description
class	visibility	data	one	Visibility of the element (public, protected, package, private).
	abstract	data	one	Boolean indicating if the element is abstract.
	leaf	data	one	Boolean indicating if the element can have specializations.
	ownedattributes	ref	many	The attributes of the element.
	ownedoperations	ref	many	The operations of the element.
	nestedclassifiers	ref	many	The nested classifiers of the element.
	generalizations	ref	many	The generalizations owned by the element.
	interfacerealizations	ref	many	The interface realizations owned by the element.
	connectors	ref	many	The connectors owned by the element.
interface	ownedattributes	ref	many	The attributes of the interface.
	ownedoperations	ref	many	The operations of the interface.
	nestedclassifiers	ref	many	The nested classifiers of the interface.
	generalizations	ref	many	The generalizations owned by the interface.
datatype	ownedattributes	ref	many	The attributes of the element.
	ownedoperations	ref	many	The operations of the element.
	generalizations	ref	many	The generalizations owned by the element.
enumeration (extends datatype)	ownedliterals	ref	many	The literals of the enumeration.
enumerationliteral	---	---	---	
primitivetype (extends datatype)	---	---	---	
connector	ends	ref	many	The connector ends of the connector.
connectorend	role	ref	one	The element that is attached at this connector end.

© 2002-2013. All rights reserved.

Model Element	Attribute	Type	Multi.	Description
property	visibility	data	one	Visibility of the feature (public, private, protected, package).
	propertytype	ref	one	Reference to the feature type (class, data type etc).
	isreadonly	data	one	Changeability of the feature. Values: true, false (default if not specified).
	association	ref	one	Reference to association if this is an association end.
	aggregation	data	one	The aggregation kind of the property: shared, composite, or none (default).
	qualifiers	ref	many	The qualifier attributes of the property.
port (extends property)	---	---	---	
operation	visibility	data	one	Visibility of the operation (public, private, protected, package).
	abstract	data	one	Boolean indicating if operation is abstract.
	isquery	data	one	Boolean indicating if operation is a query that does not change the classifier's state.
	ownedparameters	ref	many	The parameters of the operation.
parameter	kind	data	one	The direction of the parameter: in (default), out, inout, return.
	parametertype	ref	one	Reference to the parameter type (class, data type etc).
method	---	---	---	
package	ownedmembers	ref	many	The owned member elements.
model (extends package)	---	---	---	
association	memberends	ref	many	The member ends of the association.
	ownedends	ref	many	The owned ends of the association (not navigable association ends).
	generalizations	ref	many	The generalizations owned by the association.
associationclass (extends association)	---	---	---	
generalization	general	ref	one	Link to the parent in the generalization.
interfacerealization	contract	ref	one	Link to the realized interface.

© 2002-2013. All rights reserved.

Model Element	Attribute	Type	Multi.	Description
dependency	supplier	ref	many	Links to the supplier elements of the relationship.
	client	ref	many	Links to the client elements of the relationship.
abstraction (extends dependency)	---	---	---	
realization (extends abstraction)	---	---	---	
substitution (extends realization)	---	---	---	
usage (extends dependency)	---	---	---	
collaboration (extends class)	ownedbehaviors	ref	many	Behavior specifications owned by the collaboration.
interaction (extends class)	lifelines	ref	many	The lifelines involved in the interaction.
	messages	ref	many	The messages sent within the interaction.
	fragments	ref	many	The message occurrence specifications and combined fragments of the interaction.
instancespecification	classifier	ref	many	Links to the classifiers that this entity is an instance of.
	deployments	ref	many	The deployments where this instance specification is the target.
lifeline	represents	ref	one	The interaction participant that this lifeline represents.
message	receiveevent	ref	one	Occurrence specification for the message reception.
	sendevent	ref	one	Occurrence specification for the message sending.
	sort	data	one	The sort of communication (synchCall, asynchCall, asynchSignal, etc.).
occurrencespec	covered	ref	one	The lifeline on which this occurrence specification appears.
	kind	data	one	The type of this occurrence specification (execution, message, destruction) (since UML2.4).
	event	ref	one	The occurring event (up to UML2.3).

© 2002-2013. All rights reserved.

Model Element	Attribute	Type	Multi.	Description
combinedfragment	operator	data	one	The interaction operator (alt, loop, break, opt, par, ref, etc.)
	covered	ref	many	The lifelines covered by this combined fragment.
	operands	ref	many	The interaction operands of this combined fragment.
interactionoperand	fragments	ref	many	The message occurrence specifications and combined fragments of the interaction operand.
actor (extends class)	---	---	---	
usecase (extends class)	includes	ref	many	The include relationships owned by the use case.
	extends	ref	many	The extend relationships owned by the use case.
	extensionpoints	ref	many	The extension points owned by the use case.
extensionpoint	---	---	---	
usecaseextend	extendedcase	ref	one	The use case that is extended.
	usecaseextensionpoint	ref	one	The extension point in the extended use case.
usecaseinclude	usecaseaddition	ref	one	The included use case.
statemachine	regions	ref	many	The top-level regions owned by the state machine.
	isprotocol	data	one	Boolean indicating if this is a protocol state machine
	connectionpoints	ref	many	The connection points of the state machine.
region	subvertices	ref	many	The states contained in the region.
	transitions	ref	many	The transitions contained in the region.
state	kind	data	one	The kind of state: '' (empty), final, initial, deepHistory, shallowHistory, join, fork, choice, junction, entryPoint, exitPoint, terminate.
	regions	ref	many	The regions of the composite or concurrent state.
	entry	ref	many	The entry activities of the state.
	exit	ref	many	The exit activities of the state.
	doactivity	ref	many	The do-activities of the state.
	connectionpoints	ref	many	The connection points of the state.

© 2002-2013. All rights reserved.

Model Element	Attribute	Type	Multi.	Description
transition	kind	data	one	The kind of transition: external, internal, or local.
	isprotocol	data	one	Boolean indicating if this is a protocol transition.
	transsource	ref	one	The source state of the transition.
	transtarget	ref	one	The target state of the transition.
	triggers	ref	many	The triggers that may fire the transition.
	guard	ref	one	The guard of the transition.
	effect	ref	one	The activity to be performed when the transition fires.
constraint	---	---	---	
trigger	event	ref	one	The event causing the trigger.
event	kind	data	one	The kind of event (signal, call, change, time, anyreceive).
	linkedeventelement	ref	one	The element (signal, operation, boolean expression, or deadline) that raised the event, if any.
activity	nodes	ref	many	The action, control, and object nodes of the activity.
	edges	ref	many	The control and object flows of the activity.
	groups	ref	many	The groups of the activity.
activitygroup	edges	ref	many	The contained edges of the group.
	nodes	ref	many	The contained nodes of the group.
	groups	ref	many	The subgroups of the group.
	handlers	ref	many	The exception handlers of the structured node group.
	pins	ref	many	Pins owned by the activity group.
	kind	data	one	The kind of group: partition, interruptible, expansion, structured, conditional, loop, sequence.
action	kind	data	one	The kind of action: send, return, create, etc (there are many).
	inputs	ref	many	The input pins of the action.
	outputs	ref	many	The output pins of the action.
	handlers	ref	many	The exception handlers owned by the action.
controlnode	kind	data	one	The kind of control node: initial, activityfinal, flowfinal, fork, join, merge, or decision.

© 2002-2013. All rights reserved.

Model Element	Attribute	Type	Multi.	Description
objectnode	kind	data	one	The kind of object node: centralbuffer, datastore, activityparameter, or expansion.
pin	kind	data	one	The kind of pin (input, output, value, actioninput).
	type	ref	one	The type of the pin.
controlflow	source	ref	one	Source of the control flow.
	target	ref	one	Target of the control flow.
	guard	ref	one	Guard of the control flow.
objectflow	source	ref	one	Source of the object flow.
	target	ref	one	Target of the object flow.
	guard	ref	one	Guard of the control flow.
signal	---	---	---	
exceptionhandler	handlerbody	ref	one	The node that handles the exception.
	exceptioninput	ref	one	The input node of the handler body that received the exception token.
reception	signal	ref	one	Signal handled by this reception.
expression	---	---	---	
opaqueexpression	---	---	---	
instancevalue	---	---	---	
literalboolean	---	---	---	
literalinteger	---	---	---	
literalreal	---	---	---	
literalstring	---	---	---	
literalunlimitednatural	---	---	---	
literalnull	---	---	---	
component (extends class)	realizations	ref	many	The realizations owned by the component.
	members	ref	many	Other members owned by the component.
node (extends class)	nestednodes	ref	many	The subnodes located on the node.
	deployments	ref	many	The deployment links owned by the node (=deployment location).
	kind	data	one	The type of node (regular, executionenvironment, device)
artifact (extends class)	nestedartifacts	ref	many	The artifacts that are defined (nested) within the artifact.
	manifestations	ref	many	The manifestation links to the model elements that are manifested in the artifact.

© 2002-2013. All rights reserved.

Model Element	Attribute	Type	Multi.	Description
deploymentspec (extends artifact)	---	---	---	
deployment (extends dependency)	configurations	ref	many	The deployment specifications that parameterize the deployment.
manifestation (extends abstraction)	---	---	---	
stereotype	---	---	---	
comment	body	data	one	The comment text.
diagram	type	data	one	The type of diagram (class diagram, sequence diagram, etc).
diagramelement	element	ref	one	The element that is shown on the diagram.

Table 9: SDMetrics metamodel elements and attributes for UML2.x models

© 2002-2013. All rights reserved.

B: List of Design Metrics

This appendix provides the detailed definitions of the design metrics that ship with SDMetrics. For space reasons, we only list the metrics for the UML2.x metamodel. The metrics for the UML1.x metamodel are largely the same, or have equivalent counterparts, where applicable. You can browse the list of UML1.x metrics in the measurement catalog (see Section 4.13 "The View 'Catalog'") when analyzing UML1.x models.

B.1 Class Metrics

Metric: **NumAttr** Category: Size

The number of attributes in the class.

The metric counts all properties regardless of their type (data type, class or interface), visibility, changeability (read only or not), and owner scope (class-scope, i.e. static, or instance attribute).

Not counted are inherited properties, and properties that are members of an association, i.e., that represent navigable association ends.

- Also known as: NV (Number of Variables per class) [LK94].

Metric: **NumOps** Category: Size

The number of operations in a class.

Includes all operations in the class that are explicitly modeled (overriding operations, constructors, destructors), regardless of their visibility, owner scope (class-scope, i.e., static), or whether they are abstract or not. Inherited operations are not counted.

- Also known as: WMC (Weighted method complexity) where each operation is assigned unity complexity [CK94].
- Also known as: NM (Number of Methods) [LK94].

Metric: **NumPubOps** Category: Size

The number of public operations in a class.

Same as metric NumOps, but only counts operations with public visibility. Measures the size of the class in terms of its public interface.

- Also known as: NPM (Number of Public Methods) [LK94].

© 2002-2013. All rights reserved.

Metric: **Setters** Category: Size

The number of operations with a name starting with 'set'.

Note that this metric does not always yield accurate results. For example, an operation `settleAccount` will be counted as setter method.

- See also: Getters.

Metric: **Getters** Category: Size

The number of operations with a name starting with 'get', 'is', or 'has'.

Note that this metric does not always yield accurate results. For example, an operation `isolateNode` will be counted as getter method.

- See also: Setters.

Metric: **Nesting** Category: not specified

The nesting level of the class (for inner classes).

Measures how deeply a class is nested within other classes. Classes not defined in the context of another class have nesting level 0, their inner classes have nesting level 1, etc. Nesting levels deeper than 1 are unusual; an excessive nesting structure is difficult to understand, and should be revised.

Metric: **IFImpl** Category: Inheritance

The number of interfaces the class implements.

This only counts direct interface realization links from the class to the interface. For example, if a class *C* implements an interface *I*, which extends some other interfaces, only interface *I* will be counted, but not the interfaces that *I* extends (even though class *c* implements those interfaces, too).

Metric: **NOC** Category: Inheritance

The number of children of the class (UML Generalization).

Similar to export coupling, NOC indicates the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class. A large number of child classes may indicate improper abstraction of the parent class.

- Defined in [CK94].

© 2002-2013. All rights reserved.

Metric: **NumDesc** Category: Inheritance

The number of descendents of the class (UML Generalization).

Counts the number of children of the class, their children, and so on.

- See also: NOC (Number of Children).
- Suggested in [LC94] and [TSM92].

Metric: **NumAnc** Category: Inheritance

The number of ancestors of the class.

Counts the number of parents of the class, their parents, and so on. If multiple inheritance is not used, the metric yields the same values as DIT.

- Suggested in [LC94] and [TSM92].

Metric: **DIT** Category: Inheritance

The depth of the class in the inheritance hierarchy.

This is calculated as the longest path from the class to the root of the inheritance tree. The DIT for a class that has no parents is 0.

Classes with high DIT inherit from many classes and thus are more difficult to understand. Also, classes with high DIT may not be proper specializations of all of their ancestor classes.

- Defined in [CK94].

Metric: **CLD** Category: Inheritance

Class to leaf depth. The longest path from the class to a leaf node in the inheritance hierarchy below the class.

- Defined in [TSM92].
- See also: NOC (Number of Children).

Metric: **OpsInh** Category: Inheritance

The number of inherited operations.

This is calculated as the sum of metric NumOps taken over all ancestor classes of the class.

- Also known as NMI [LK94].
- See also: DIT.

© 2002-2013. All rights reserved.

Metric: **AttrInh** Category: Inheritance
The number of inherited attributes.

This is calculated as the sum of metric NumAttr taken over all ancestor classes of the class.

- Loosely based on AIF in [AGE95].

Metric: **Dep_Out** Category: Coupling (import)
The number of elements on which this class depends.

This metric counts outgoing plain UML dependencies and usage dependencies (shown as dashed arrows in class diagrams, usage with 'use' stereotype).

Metric: **Dep_In** Category: Coupling (export)
The number of elements that depend on this class.

This metric counts incoming plain UML dependencies and usage dependencies (shown as dashed arrows in class diagrams, usage with 'use' stereotype).

Metric: **NumAssEl_ssc** Category: Coupling
The number of associated elements in the same scope (namespace) as the class.

For instance, for a class that is defined in a package *p*, this counts only associations with classes, interfaces, etc. in the same package *p*. Such associations are encouraged, because they do not cross the package boundary, and contribute to the cohesion of the package.

This metric does not distinguish plain, aggregate, or composite associations, as well as incoming, outgoing, or bidirectional associations; all such associations are counted.

Metric: **NumAssEl_sb** Category: Coupling
The number of associated elements in the same scope branch as the class.

For instance, for a class that is defined in a package *p*, this metric only counts associations with model elements

- in *p* itself,
- in packages that *p* contains (subpackages, sub-subpackages etc. of *p*),
- in packages that contain *p* (packages of which *p* is a subpackage, sub-subpackage etc).

Such associations cross package boundaries, but one of the packages is nested within the other. Therefore, a dependency between the packages is expected, anyway.

- Note: Like metric NumAssEl_ssc, direction and aggregation of the associations are ignored.

© 2002-2013. All rights reserved.

Metric: **NumAssEl_nsb** Category: Coupling
The number of associated elements not in the same scope branch as the class.

For a class that is defined in a package *p*, this counts only associations with model elements in packages that neither contain *p* nor are contained by *p*. These are the least desirable associations, because they couple otherwise unrelated packages. Such associations cannot be avoided, but their use should be minimized.

- Note: Like metric NumAssEl_ssc, direction and aggregation of the associations are ignored.

Metric: **EC_Attr** Category: Coupling (export)
The number of times the class is externally used as attribute type.

This is the number of attributes in other classes that have this class as their type.

- Version of: OAEC+AAEC in [BDM97].

Metric: **IC_Attr** Category: Coupling (import)
The number of attributes in the class having another class or interface as their type.

- Version of: OAIC+AAIC in [BDM97].
- Also known as: DAC (data abstraction coupling) [LH93].

Metric: **EC_Par** Category: Coupling (export)
The number of times the class is externally used as parameter type.

This is the number of parameters defined outside this class, that have this class as their type.

- Version of: OMEC+AMEC in [BDM97].

Metric: **IC_Par** Category: Coupling (import)
The number of parameters in the class having another class or interface as their type.

- Version of: OMIC+AMIC in [BDM97].

Metric: **Connectors** Category: Complexity
The number of connectors owned by the class.

© 2002-2013. All rights reserved.

Metric: **InstSpec** Category: not specified
The number of instance specification where the class is a classifier.

Similar to export coupling, the more instances of the class there are, the larger the role of the class in the system.

Metric: **LLInst** Category: not specified
The number of lifelines that represent a property of which this class is the type.

Similar to export coupling, the more lifelines there are of the class, the larger the role of the class in the system.

Metric: **MsgSent** Category: Coupling (import)
The number of messages sent.

Counts the number of messages that instances of this class send to instances of other classes, or unclassified instances.

- Version of OMMIC+AMMIC in [BDM97].

Metric: **MsgRecv** Category: Coupling (export)
The number of messages received.

Counts the number of messages that instances of this class receive from instances of other classes or unclassified instances.

- Version of OMMEC+AMMEC in [BDM97].

Metric: **MsgSelf** Category: Complexity
The number of messages sent to instances of the same class.

Counts the number of messages that instances of this class send to themselves or to other instances of the same class.

- Version of ICH in [LLW95].

Metric: **Diags** Category: Diagram
The number of times the class appears on a diagram.

© 2002-2013. All rights reserved.

B.2 Interface Metrics

Metric: **NumOps** Category: Size
The number of operations in the interface.

- See also: NumOps for classes.

Metric: **EC_Attr** Category: Coupling (export)
The number of times the interface is used as attribute type.

- See also: EC_Attr for classes.

Metric: **EC_Par** Category: Coupling (export)
The number of times the interface is used as parameter type.

- See also: EC_Par for classes.

Metric: **IC_Par** Category: Coupling (import)
The number of parameters in the interface having an interface or class as their type.

- See also: IC_Par for classes.

Metric: **Assoc** Category: Coupling
The number of elements the interface has an association with.

The metric counts incoming, outgoing, or bidirectional associations, aggregations and compositions to all kinds of elements.

In practice, there should mostly be incoming associations (see AttrOnIF), so the metric has export coupling characteristics.

Metric: **NumDirClients** Category: not specified
The number of elements directly implementing the interface.

This is the number of UML abstractions where this interface is the target.

- See also NumIndClients.

© 2002-2013. All rights reserved.

Metric: **NumIndClients** Category: not specified
The number of elements implementing a descendent of the interface.

An element implementing an interface is also an implementation of every ancestor of that interface. This metrics counts how many classes, components, etc., indirectly implement the interface via a descendent. Together with metric NumDirClients, this indicates the total number of realizations of the interface.

Metric: **NumAnc** Category: Inheritance
The number of ancestors of the interface.

- See also: NumAnc for classes.

Metric: **NumDesc** Category: Inheritance
The number of descendents of the interface.

- See also: NumDesc for classes.

Metric: **Diags** Category: Diagram
The number of times the interface appears on a diagram.

B.3 Package Metrics

Metric: **NumCls** Category: Size
The number of classes in the package.

Counts all classes, regardless of their visibility (public, protected, private, or package), or abstractness.

Metric: **NumCls_tc** Category: Size
The number of classes in the package, its subpackages, and so on.

This is the sum of metric NumCls for this package, and all its direct and indirect subpackages.

Metric: **NumOpsCls** Category: Size
The number of operations in the classes of the package.

This is the sum of metric NumOps, taken over all classes in this package, and more fine-grained measure of the size of the package.

© 2002-2013. All rights reserved.

Metric: **NumInterf** Category: Size

The number of interfaces in the package.

Like metric NumCls, this counts all interfaces, regardless of their visibility.

Metric: **R** Category: Complexity

The number of relationships between classes and interfaces in the package. There is a dependency from class or interface *C* to class or interface *D* if

- *C* has an attribute of type *D*
- *C* has an operation with a parameter of type *D*
- *C* has an association, aggregation, or composition with navigability to *D*
- *C* has a UML dependency or usage dependency to *D*
  UML dependencies are shown as dashed arrows in the diagrams (usage with stereotype 'use').
- *C* is a child of *D*
- *C* implements interface *D*

The metric counts all such dependencies between classes and interfaces in the package. Bidirectional associations are counted twice, because *C* knows *D* and vice versa. By convention, associations that indicate no navigability at either end are considered to be bidirectional.

- Suggested in [Mar03].

Metric: **H** Category: Cohesion

Relational cohesion.

This is the average number of internal relationships per class/interface, and is calculated as the ratio of R+1 to the number of classes and interfaces in the package.

- Suggested in [Mar03].

Metric: **Ca** Category: Coupling (export)

Afferent coupling.

The number of elements outside this package that depend on classes or interfaces in this package. The dependencies considered are the same ones listed with metric R.

- Suggested in [Mar03].

© 2002-2013. All rights reserved.

Metric: **Ce** Category: Coupling (import)
Efferent coupling.

The number of elements outside this package that classes or interfaces in this package depend on. The dependencies considered are the same ones listed with metric R.

- Suggested in [Mar03].

Metric: **I** Category: not specified
Instability or ease of change.

This is the ratio of efferent coupling (metric Ce) to total coupling (Ce+Ce). Values of metric I range between 0 and 1.

A value close to 0 indicates a package that does not rely much on other packages, but is heavily relied upon by other packages. Such a package should be stable, because it is hard to change: changes to the package potentially have a large impact on the model ("ripple effects").

A value close to 1 indicates a package that mostly relies on other packages, but that itself is not much relied upon. Such a package can be unstable, because it is easy to change: changes to the package are not likely to have a large impact on the model.

- Suggested in [Mar03].
- See also: rule SDP1.

Metric: **A** Category: not specified
Abstractness (or generality) of the package.

This is the ratio of abstract classes and interfaces in the package to the total number of interfaces and classes in the package.

Values range from 0 to 1. Zero indicates packages without interfaces or abstract classes, 1 indicates a package consisting only interfaces and abstract classes.

- Suggested in [Mar03].

© 2002-2013. All rights reserved.

Metric: **D** Category: not specified

Distance from the main sequence. Package design should aim to strike a balance between instability and abstractness of the packages. A stable package should be abstract, so that changes to the package are merely extensions that do not affect existing clients of the package. An unstable package is easy to change and can therefore be concrete.

The Stable-Abstractions-Principle (SAP) says that a package should be as abstract as it is stable. With abstractness measured by metric A, and stability measured by metric I, the SAP demands that A+I be close to 1.

If you plot A vs. I in a graph, the "main sequence" is the theoretical optimal line where A+I=1. Metric D is the distance of the package from the main sequence, and is calculated as (A+I-1)*sqrt(2).

Values range from -sqrt(2) to +sqrt(2). Values close to zero indicate packages that adhere to the SAP. A large negative value indicates a package that is concrete and stable (A and I close to 0). Such a package can be "painful" because it is not extensible and prone to change. A large positive value indicates a package that is abstract and unstable. Such a package is extensible but has few dependents, and is therefore useless.

- Suggested in [Mar03].

Metric: **DN** Category: not specified

Normalized distance D' from the main sequence.

This is a variation of metric D that has been normalized to range between 0 and 1. It is calculated as |A+I-1|. Values close to zero indicate packages that adhere to the SAP (see metric D).

- Suggested in [Mar03].

Metric: **Nesting** Category: Nesting

Nesting level of the package in the package hierarchy.

Top level packages have nesting level 0, their subpackages are at level 1, and so on.

© 2002-2013. All rights reserved.

Metric: **ConnComp** Category: Cohesion
The connected components formed by the classes and interfaces of the package.

The classes and interfaces of a package, and their dependencies, form a graph. This metric counts the number of connected components of that graph.

Ideally, all classes and interfaces of the package should be related directly or indirectly, so that there is only one connected component. If there are two or more connected components, you may consider moving some classes or interfaces to other packages, or splitting up the package.

- See metric R for the list of dependencies considered between the classes and interfaces.
- The graph considered is an undirected graph, directions of dependencies are ignored.
- You can view the connected components in the Graph Structures View.

Metric: **Dep_Out** Category: Coupling (import)
The number of UML dependencies where the package is the client.

See also Dep_Out for classes.

Metric: **Dep_In** Category: Coupling (export)
The number of UML dependencies where the package is the supplier.

See also Dep_In for classes.

Metric: **DepPack** Category: Coupling (import)
The number of packages on which classes and interfaces of this package depend.

A package *P* depends on a package *P'* if

- a class or interface in *P* depends on a class or interface in *P'* (see metric R for a description of these dependencies).
- there is a UML dependency (dashed arrow) from *P* to *P'*.

The more packages *P* depends on, the more difficult it is to reuse *P* in a different context.

This metric is similar to metric Ce, the difference is that Ce counts the individual classes or interfaces that are depended upon.

© 2002-2013. All rights reserved.

Metric: **MsgSent_Outside** Category: Coupling (import)
The number of messages sent to instances of classes outside the package.

The metric counts, for instances of classes of this package, the messages they send to instances of classes from other packages. That is, outgoing messages that cross the package boundary. Note that messages to unclassified instances are not counted here.

Metric: **MsgRecv_Outside** Category: Coupling (export)
The number of messages received by classifier instances of classes outside the package.

The metric counts, for instances of classes of this package, the messages they receive from instances of classes from other packages. That is, incoming messages that cross the package boundary. Note that messages from unclassified instances are not counted here.

Metric: **MsgSent_within** Category: Complexity
The number of messages sent between classifier instances of classes in the package.

The metric counts, for instances of classes of this package, the number of messages they send to themselves or other instances of classes from this package. Note that messages to unclassified instances are not counted here.

Metric: **Diags** Category: Diagram
The number of times the package appears on a diagram.

B.4 Interaction Metrics

Metric: **LifeLines** Category: Size
The number of lifelines participating in the interaction.

Metric: **Messages** Category: Complexity
The number of messages sent within the interaction.

Metric: **SelfMessages** Category: not specified
The number of messages that objects in the interaction send to themselves.

The focus of sequence diagrams should be on object interactions. A sequence diagram with a large number of self messages may indicate that the modeler attempted to model object internal algorithms.

© 2002-2013. All rights reserved.

Metric: **CombinedFragments** Category: Size
The number of combined fragments in the interaction.

Metric: **Operands** Category: Size
The number of interaction operands of the combined fragments in the interaction.

Metric: **Height** Category: not specified
The maximum number of messages on any of the lifelines of interaction.

Very long and busy lifelines with lots of messages attached to them may indicate objects that have too many responsibilities.

Metric: **CombinedFragmentNesting** Category: not specified
The maximum nesting level of combined fragments in the interaction.

Excessive nesting of combined fragments makes sequence diagrams harder to read and understand. Consider extracting such combined fragments into sequence diagrams of their own.

B.5 Usecase Metrics

Metric: **NumAss** Category: not specified
The number of associations the use case participates in.

Counts incoming, outgoing, and bidirectional associations.

Metric: **ExtPts** Category: not specified
The number of extension points of the use case.

Metric: **Including** Category: Coupling (import)
The number of use cases which this one includes.

Metric: **Included** Category: Coupling (export)
The number of use cases which include this one.

Metric: **Extended** Category: not specified
The number of use cases which extend this one.

© 2002-2013. All rights reserved.

Metric: **Extending** Category: not specified
The number of use cases which this one extends.

Metric: **Diags** Category: Diagram
The number of times the use case appears on a diagram.

B.6 Statemachine Metrics

Metric: **Trans** Category: Complexity
The number of transitions in the state machine.

Internal transitions are not included in this count.

- Version of NT in [MGP03].

Metric: **TEffects** Category: Complexity
The number of transitions with an effect in the state machine.

Metric: **TGuard** Category: Complexity
The number of transitions with a guard in the state machine.

- Also known as NG in [MGP03].

Metric: **TTrigger** Category: Complexity
The number of triggers of the transitions of the state machine.

- Also known as NE in [MGP03].

© 2002-2013. All rights reserved.

Metric: **States** Category: Size
The number of states in the state machine.

This includes pseudo states, as well as composite and concurrent states of the statemachine, and recursively the states they contain, at all levels of nesting. Submachine states count as "one", the states in statemachines they reference are not included.

- Corresponds to NSS+NCS in [MGP03].

Metric: **SActivity** Category: Size
The number of activities defined for the states of the state machine.

This counts entry, exit, and do activities (or interactions or statemachines) defined for the states. The states considered are those counted by metric States.

- corresponds to NEntry+NExit+NA in [MGP03].

Metric: **CC** Category: Complexity
The cyclomatic complexity of the state-transition graph.

This is calculated as Trans-States+2.

- Suggested in [MGP03].

B.7 Activity Metrics

Metric: **Actions** Category: Size
The number of actions of the activity.

Includes actions in all activity groups (partitions, interruptible regions, expansion regions, structured activities including conditional, loop, and sequence nodes), and their subgroups, sub-subgroups, etc.

Metric: **ObjectNodes** Category: Size
The number of object nodes of the activity.

Counts data store, central buffer, and activity parameter nodes in all activity groups and their subgroups etc. (see Actions).

© 2002-2013. All rights reserved.

Metric: **Pins** Category: Size
The number of pins on nodes of the activity.

Counts all input, output, and value pins on all nodes and groups of the activity.

Metric: **ControlNodes** Category: Size
The number of control nodes of the activity.

Control nodes are initial, activity final, flow final, join, fork, decision, and merge nodes. The metric also counts control nodes in all activity groups and their subgroups etc. (see Actions).

Metric: **Partitions** Category: Size
The number of activity partitions in the activity.

Metric: **Groups** Category: Size
The number of activity groups or regions of the activity.

Counts interruptible and expansion regions, structured activities, conditional, loop, and sequence nodes, at all levels of nesting.

Metric: **ControlFlows** Category: Complexity
The number of control flows of the activity.

Includes contained edges in all activity groups and their subgroups etc. (see Actions)

Metric: **ObjectFlows** Category: Complexity
The number of object flows of the activity.

Includes contained edges in all activity groups and their subgroups etc. (see Actions)

Metric: **Guards** Category: Complexity
The number of guards defined on object and control flows of the activity.

Includes contained edges in all activity groups and their subgroups etc. (see Actions).

© 2002-2013. All rights reserved.

Metric: **ExcHandlers** Category: Complexity
The number of exception handlers of the activity.

Includes exception handlers for all nodes in all activity groups and their subgroups etc. (see Actions).

B.8 Component Metrics

Metric: **NumOps** Category: Size
The number of operations of the component.

- See also metric NumOps for classes

Metric: **NumComp** Category: Size
The number of subcomponents of the component.

Counts components directly owned by this component; sub-sub-components etc. are not included in this count.

Metric: **NumPack** Category: Size
The number of packages of the component.

This only counts packages directly owned by the component; any sub-packages etc. of these packages are not included in this count.

Metric: **NumCls** Category: Size
The number of classes of the component.

This only counts classes directly owned by the component; nested classes, classes in packages, subcomponents, etc. are not included in this count.

- See also metric NumCls for packages.

Metric: **NumInterf** Category: Size
The number of interfaces of the component.

This only counts interfaces directly owned by the component; interfaces in packages, sub-components, etc. are not included in this count.

- See also metric NumInterf for packages.

© 2002-2013. All rights reserved.

Metric: **NumManifest** Category: not specified
The number of artifacts of which this component is a manifestation.

Metric: **Connectors** Category: Complexity
The number of connectors owned by the component.

Metric: **ProvidedIF** Category: Inheritance
The number of interfaces the component provides.

A component provides an interface if there is a plain or interface realization to the interface, or if a port of the component provides the interface.

- See also metric IFImpl for classes.

Metric: **RequiredIF** Category: not specified
The number of interfaces the component requires.

The component requires an interface if there is a dependency, usage dependency, or an association to the interface, or if a port of the component requires the interface.

Metric: **Dep_Out** Category: Coupling (import)
The number of outgoing UML dependencies (component is the client).

See also Dep_Out for classes.

Metric: **Dep_In** Category: Coupling (export)
The number of incoming UML dependencies (component is the supplier).

See also Dep_In for classes.

Metric: **Assoc_Out** Category: Coupling (import)
The number of associated elements via outgoing associations.

Takes associations, aggregations, and compositions with navigability away from the component into account, i.e., elements the component knows.

© 2002-2013. All rights reserved.

Metric: **Assoc_In** Category: Coupling (import)
The number of associated elements via incoming associations.

Takes associations, aggregations, and compositions with navigability to the component into account, i.e., elements that know the component.

Metric: **Diags** Category: Diagram
The number of times the component appears on a diagram.

B.9 Node Metrics

Metric: **Type** Category: not specified
The type of node (regular, execution environment, or device).

Metric: **NumOps** Category: Size
The number of operations of the node.

- See also metric NumOps for classes.

Metric: **NumComp** Category: Size
The number of components located on the node.

Counts components directly owned by this node; sub-components etc. are not included in this count.

Metric: **NumNodes** Category: Size
The number of subnodes of the node.

Counts nodes directly located on this node; sub-sub-nodes etc. are not included in this count.

Metric: **NumArt** Category: Size
The number of artifacts deployed on the node.

This metric counts

- artifacts directly owned by this node; sub-artifacts etc. are not included,
- artifacts with a deployment dependency to the node.

© 2002-2013. All rights reserved.

Metric: **NumPack** Category: Size
The number of packages of the node.

This only counts packages directly owned by the node; any sub-packages etc. of these packages are not included in this count.

Metric: **AssEl** Category: Coupling
The number of elements the node is associated with.

Takes incoming, outgoing, and bidirectional communication paths, associations, aggregations, and compositions into account.

Metric: **Diags** Category: Diagram
The number of times the node appears on a diagram.

B.10 Diagram Metrics

Metric: **Type** Category: not specified
The type of diagram (class diagram, sequence diagram, etc.).

Metric: **Elements** Category: Size
The total number of design elements on the diagram. See diagram metrics.

Metric: **Classes** Category: Size
The number of classes on the diagram. See diagram metrics.

Metric: **Interfc** Category: Size
The number of interfaces on the diagram. See diagram metrics.

Metric: **Packages** Category: Size
The number of packages on the diagram. See diagram metrics.

Metric: **Assoc** Category: Complexity
The number of associations on the diagram. See diagram metrics.

Metric: **Genrs** Category: Complexity
The number of generalizations on the diagram. See diagram metrics.

© 2002-2013. All rights reserved.

Metric: **Deps** Category: Complexity
The number of UML dependencies and UML usage dependencies on the diagram. See diagram metrics.

Metric: **IfRealize** Category: Complexity
The number of interface realizations on the diagram. See diagram metrics.

Metric: **InstSpec** Category: Size
The number of instance specifications on the diagram. See diagram metrics.

Metric: **Lifelines** Category: Size
The number of lifelines on the diagram. See diagram metrics.

Metric: **Connectors** Category: Complexity
The number of connectors on the diagram. See diagram metrics.

Metric: **Messages** Category: Complexity
The number of messages on the diagram. See diagram metrics.

Metric: **Actors** Category: Size
The number of actors on the diagram. See diagram metrics.

Metric: **UseCase** Category: Size
The number of use cases on the diagram. See diagram metrics.

Metric: **ExtPts** Category: Size
The number of extension points on the diagram. See diagram metrics.

Metric: **Extends** Category: Complexity
The number of use case extensions on the diagram. See diagram metrics.

Metric: **Includes** Category: Complexity
The number of use case includes on the diagram. See diagram metrics.

© 2002-2013. All rights reserved.

C: List of Design Rules

This appendix provides the detailed definitions of the design rules that ship with SDMetrics. For space reasons, we only list the rules for the UML2.x metamodel. The rules for the UML1.x metamodel are largely the same, or have equivalent counterparts, where applicable. You can browse the list of UML1.x rules in the measurement catalog (see Section 4.13 "The View 'Catalog'") when analyzing UML1.x models.

C.1 Class Rules

Rule: **Unnamed**	Category: Completeness
Severity: 1-high	Applies to: all areas

Class has no name.

Give the class a descriptive name that reflects its purpose. Unnamed classes will cause problems during code generation.

- Suggested in [RVR04].

Rule: **Unused**	Category: Completeness
Severity: 1-high	Applies to: all areas

The class is not used anywhere.

The class has no child classes, dependencies, or associations, and it is not used as parameter or property type. You'll probably still need to model the clients of the class, or else consider deleting the class from the model.

Note: for models that were reverse-engineered from source code, this rule may falsely report many classes as "unused". This happens for classes that are only referenced in method implementations, e.g., via local variables.

- Suggested in [Rie96] (heuristic #3.7 eliminate irrelevant classes).

Rule: **NotCapitalized**	Category: Naming
Severity: 3-low	Applies to: all areas

Class names should start with a capital letter.

This naming convention is a recommended style guideline in the UML standards [OMG03], [OMG05].

© 2002-2013. All rights reserved.

Rule: **GodClass** Category: Style
Severity: 2-med Applies to: all areas
The class has more than 60 attributes and operations.

Also known as blob classes, large classes are likely maintenance bottlenecks, sources of unreliability, and indicate a lack of (object-oriented) architecture and architecture enforcement.

Consider refactoring the class to split it up into smaller classes.

- Threshold of 60 cited in [BMM98].
- See also metrics NumOps and NumAttr
- Also known as "Large Class" code smell [Fow99].
- Value reported: number of operations and attributes.

Rule: **Keyword** Category: Naming
Severity: 2-med Applies to: design
Class name is a Java or C++ keyword. Using programming language keywords for class names will cause problems during code generation. Find another name for the class. Capitalizing the name will also help, see rule NotCapitalized.

- Suggested in [RVR04].

Rule: **MultipleInheritance** Category: Style
Severity: 3-low Applies to: all areas
Use of multiple inheritance - class has more than one parent.

The use of multiple inheritance is controversial. Some OO programming languages do not support multiple inheritance. Review the class design to confirm that the use of multiple inheritance is justified.

- Suggested in [Rie96].

Rule: **SpecLeafClass** Category: Correctness
Severity: 1-high Applies to: all areas
Class is marked as leaf, but it has child classes.

Leaf classes cannot have any child classes. This is a WFR of the UML.

© 2002-2013. All rights reserved.

Rule: **NoSpec** Category: Completeness
Severity: 2-med Applies to: all areas
Abstract class has no child classes.

Abstract classes cannot be instantiated. Without specializations that can be instantiated, the abstract class is useless.

- Suggested in [Rie96].
- Violations of this rule would be justified if the class is part of a framework or library, and is meant to be extended by users of the framework/library.

Rule: **CyclicInheritance** Category: Inheritance
Severity: 1-high Applies to: all areas
Class inherits from itself directly or indirectly.

The inheritance graph must be a tree, no cycles are allowed.

- This is a WFR of the UML.
- You can view the inheritance graph in the graph structures dialog.
- Value returned: number of classes in the cycle.

Rule: **ConcreteSuper** Category: Style
Severity: 1-high Applies to: all areas
The abstract class has a parent class that is not abstract.

This is bad design. A child class should be substitutable for the parent class. Since the parent class can be instantiated, but not the child class, substitution is not possible anymore.

- Suggested in [Lan03].
- Value returned: name of the concrete parent class.

Rule: **DupOps** Category: Correctness
Severity: 1-high Applies to: all areas
Class has duplicate operations.

There are two or more operations with identical signatures (i.e., operation name and list of parameter types). Operation signatures must be unique within the class.

- This is a WFR of the UML.
- Value reported: name of the duplicate operation.

© 2002-2013. All rights reserved.

Rule: **DupAttrNames** Category: Correctness
Severity: 1-high Applies to: all areas
The class has two or more properties with identical names.

Attribute names must be unique within the class.

- This is a WFR of the UML.
- Value reported: name of the duplicate attribute.

Rule: **AttrNameOvr** Category: Naming
Severity: 2-med Applies to: all areas
The class defines a property of the same name as an inherited attribute.

During code generation, this may inadvertently hide the attribute of the parent class. Consider changing the name of the attribute in the child class.

- Suggested in [RVR04].

Rule: **DescendentRef** Category: Style
Severity: 1-high Applies to: all areas
The class references a descendent class via associations, UML dependencies, attribute or parameter types.

This is poor design. A class *c* should be oblivious of its descendent classes. The reference to the descendent class and the inheritance links back to class *c* effectively form a dependency cycle between these classes.

Redesign this to eliminate the need for the reference to the descendent class.

- Suggested in [RVR04], [Rie96].
- Value reported: name of the referenced descendent class.

© 2002-2013. All rights reserved.

Rule: **DepCycle** Category: Style
Severity: 2-med Applies to: all areas
The class has circular references.

Circular dependencies should be avoided. The classes participating in the cycle cannot be tested and reused independently. The more classes participate in the cycle, the worse the problem is, especially if the classes reside in different packages (see also rule DepCycle for packages).

Consider revising the design to eliminate the cycle.

- See also: Dependency Inversion Principle [Mar03].
- You can view the class dependency graph and its cycles in the Graph Structures View.
- Value reported: number of classes in the cycle.

C.2 Interface Rules

Rule: **Unnamed** Category: Completeness
Severity: 1-high Applies to: all areas
Interface has no name.

- See rule Unnamed for classes.

Rule: **Unused** Category: Completeness
Severity: 1-high Applies to: all areas
The interface is not used anywhere.

The interface is not implemented anywhere, has no associations, and is not used as parameter or attribute type.

- See also rule Unused for classes.

Rule: **NotCapitalized** Category: Naming
Severity: 3-low Applies to: all areas
Interface names should start with a capital letter.

- See rule NotCapitalized for classes.

© 2002-2013. All rights reserved.

Rule: **Keyword** Category: Naming
Severity: 2-med Applies to: design
Interface name is a Java or C++ keyword; find another name for it.

- See rule Keyword for classes.

Rule: **PubOpsOnly** Category: Correctness
Severity: 1-high Applies to: all areas
The interface has operations that are not public.

All operations in interfaces must have public visibility.

- This is a WFR of the UML.
- Value returned: number of non-public operations.

Rule: **PubAttrOnly** Category: Correctness
Severity: 1-high Applies to: all areas
The interface has attributes that are not public.

All attributes in interfaces must have public visibility.

- This is a WFR of the UML2.
- Value returned: number of non-public attributes.

Rule: **AttrOnIF** Category: Style
Severity: 3-low Applies to: all areas
The interface has attributes or outgoing associations.

Interfaces can have attributes and outgoing associations since UML2.0. This rather appears to be a concession to certain component technologies, and should otherwise be avoided.

- Suggested in [Oes04].
- Value returned: number of attributes of the interface.

C.3 Datatype Rules

Rule: **Unnamed** Category: Completeness
Severity: 1-high Applies to: all areas
The data type has no name.

- See rule Unnamed for classes.

© 2002-2013. All rights reserved.

Rule: **Keyword** Category: Naming
Severity: 2-med Applies to: design
Data type name is a Java or C++ keyword; find another name for it.

- See rule Keyword for classes.

Rule: **NoQuery** Category: Correctness
Severity: 1-high Applies to: all areas
The data type has an operation that is not marked as a query.

All operations of a data type must be queries.

- This is a WFR of the UML.
- Value returned: name of the operation that should be a query.

C.4 Property Rules

Rule: **Unnamed** Category: Completeness
Severity: 1-high Applies to: all areas
The attribute has no name.

- See rule Unnamed for classes.

Rule: **Capitalized** Category: Naming
Severity: 3-low Applies to: all areas
Attribute names should start with a lowercase letter.

This is a recommended style guideline in the UML standards [OMG03], [OMG05].

- It is common practice in many programming languages to capitalize constant identifiers, including attributes. Therefore, the rule does not report read-only attributes that start with an uppercase letter.

Rule: **Keyword** Category: Naming
Severity: 2-med Applies to: design
Attribute name is a Java or C++ keyword.

- See rule Keyword for operations.

© 2002-2013. All rights reserved.

Rule: **PublicAttr** Category: Style
Severity: 2-med Applies to: all areas
Non-constant attribute is public.

External read/write access to attributes violates the information hiding principle. Allowing external entities to directly modify the state of an object is dangerous. State changes should only occur through the protocol defined by the interfaces of the object. Make the attribute private or protected.

- Suggested in [Rie96].

Rule: **NoType** Category: Completeness
Severity: 2-med Applies to: design
The attribute has no specified type.

Without a type, the attribute has no meaning in design, and code generation will not work. Specify a type for the attribute.

- Suggested in [Fra03].

C.5 Operation Rules

Rule: **Unnamed** Category: Completeness
Severity: 1-high Applies to: all areas
Operation has no name.

- See also rule Unnamed for classes.

Rule: **Capitalized** Category: Naming
Severity: 3-low Applies to: all areas
Operation names should start with a lower case letter. This is a recommended style guideline in the UML standards [OMG03], [OMG05].

In many programming languages, constructors have the same name as their class, thus starting with upper case letters. Therefore, operations with the same name as their class are not reported.

© 2002-2013. All rights reserved.

Rule: **Keyword** Category: Naming
Severity: 2-med Applies to: design
Operation name is a Java or C++ keyword.

Using programming language keywords as operation names will cause problems during code generation. Find another name for the operation.

- Suggested in [RVR04].

Rule: **AbstractOp** Category: Correctness
Severity: 1-high Applies to: all areas
Operation is abstract, but its owner class is not abstract.

In many programming languages, a class is abstract if at least one of its operations is abstract. Either make the owner class abstract, or provide an implementation for the operation.

Rule: **LongParList** Category: Style
Severity: 2-med Applies to: all areas
The operation has a long parameter list with five or more parameters.

Long parameter lists are difficult to use, and likely to change more frequently. Change the design to pass one or more objects to the operation that encapsulate the required parameters. Note: the rule only considers in, out, and inout parameters; the return parameter (if any) is not counted.

- Suggested in [Fow99].
- Value returned: the number of parameters of the operation.

Rule: **MulReturn** Category: Correctness
Severity: 1-high Applies to: all areas
The operation has more than one return parameter.

Many programming languages only support one return parameter per operation. Change some return parameters to out or inout parameters, or return one object that encapsulates all return parameters.

- This is a WFR of the UML2.
- Value returned: number of return parameters of the operation.

© 2002-2013. All rights reserved.

Rule: **DupName** Category: Naming
Severity: 1-high Applies to: all areas
The operation has two or more parameters with identical names.

Parameters must have unique names to distinguish them. This is a WFR of the UML.

- Value returned: the name of the duplicate parameters.

Rule: **Query** Category: Style
Severity: 2-med Applies to: all areas
The operation name indicates a query, but it is not marked as a query.

The operation name suggests this is a getter (see Getters). Mark the operation as query to indicate that it does not change the owner's state.

C.6 Parameter Rules

Rule: **Unnamed** Category: Completeness
Severity: 1-high Applies to: all areas
Parameter has no name.

Note that this rule does not check return parameters, as they are unnamed in most programming languages.

- See also rule Unnamed for classes.

Rule: **NoType** Category: Completeness
Severity: 2-med Applies to: all areas
The parameter has no specified type.

- See also rule NoType for attributes.

Rule: **Keyword** Category: Naming
Severity: 2-med Applies to: design
Parameter name is a Java or C++ keyword.

Return parameters are not checked, as they are unnamed in many programming languages. Also, some modeling tools assign the name 'return' as default name to return parameters.

- See rule Keyword for operations.

© 2002-2013. All rights reserved.

C.7 Package Rules

Rule: **Unnamed** Category: Completeness
Severity: 1-high Applies to: all areas
Package has no name.

- See rule Unnamed for classes.

Rule: **Capitalization** Category: Naming
Severity: 3-low Applies to: all areas
Package name has upper case letters. A common naming convention is that package names use all lower case letters.

- Suggested in [RVR04].

Rule: **Keyword** Category: Naming
Severity: 2-med Applies to: design
Package name is a Java or C++ keyword.

- See rule Keyword for operations.

Rule: **EmptyPackage** Category: Completeness
Severity: 2-med Applies to: all areas
The package has no contents.

Add model elements to the package, or delete it from the design.

Rule: **DupClsName** Category: Naming
Severity: 1-high Applies to: all areas
The package has two or more classes or interfaces with identical names.

This will cause problems during code generation. Rename the classes or interfaces so that the names are unique.

- Value returned: name of the duplicate class/interface.

© 2002-2013. All rights reserved.

Rule: **DepCycle** Category: Style
Severity: 1-high Applies to: all areas
The package has circular dependencies to other packages.

Cycles in the package dependency graph should be avoided. The packages participating in the cycle cannot be tested, reused, or released independently. The more packages participate in the cycle, the worse the problem is. Other design guidelines such as the Stable-Dependencies Principle (see rule SDP1) are also invariably violated.

Revise the design to eliminate the cycle.

- See also: Dependency Inversion Principle [Mar03].
- See DepPack for what constitutes dependencies between packages.
- You can view the package dependency graph and its cycles in the Graph Structures View.
- Value returned: number of packages in the cycle.

Rule: **SDP1** Category: Style
Severity: 2-med Applies to: all areas
Package violates the Stable-Dependencies Principle (SDP).

The package depends on another package, *P* that is less stable than itself (as measured by metric I). Package *P* is less stable and therefore more liable to change than this package. A change to *P* may ripple to this package. This is undesirable because this package is more stable and therefore harder to change.

Therefore, the Stable-Dependencies Principle says that dependencies should run in the direction of stabilities.

- Suggested in [Mar03].
- See DepPack for what constitutes dependencies between packages.
- Value returned: name of the depended package that is less stable.

Rule: **SDP2** Category: Style
Severity: 2-med Applies to: all areas
Package violates the Stable-Dependencies Principle (SDP).

The package depends on another package that is less abstract than itself.

The Stable-Abstractions-Principle (SAP) says that a package should be as abstract (as measured by metric A) as it is stable (as measured by metric I). The Stable-Dependencies Principle (SDP) says that dependencies should run in the direction of stabilities.

Therefore, dependencies should run in the direction of abstraction: a package should be more abstract than the packages it depends on.

© 2002-2013. All rights reserved.

- Suggested in [Mar03].
- See DepPack for what constitutes dependencies between packages.
- Value returned: name of the depended package that is less abstract.

C.8 Association Rules

Rule: **AggEnds** Category: Correctness
Severity: 1-high Applies to: all areas
The binary association has two composite or shared aggregation ends.

A binary association may have at most one shared (hollow diamond) or composite (filled diamond) aggregation end. This is a WFR of the UML.

Rule: **NaryAggEnds** Category: Correctness
Severity: 1-high Applies to: all areas
The n-ary association has a composite or shared aggregation end.

Three (or more)- way associations must not indicate shared or composite aggregation.

- This is a WFR of the UML.
- Value returned: the number of shared/composite aggregation ends.

Rule: **NaryNavEnds** Category: Correctness
Severity: 1-high Applies to: all areas
The n-ary association indicates a navigable association end.

Three (or more)- way associations must not indicate navigability at any of the association ends.

- This is a WFR of the UML.
- Value returned: the number of navigable association ends.

Rule: **LooseEnd** Category: Completeness
Severity: 1-high Applies to: all areas
The association has one or more ends not connected to a model element.

Check the ends of the association, and attach the loose end(s) to the proper model element(s), or remove the association from the model.

© 2002-2013. All rights reserved.

Rule: **NaryAgg** Category: Style
Severity: 3-low Applies to: design
The association has three or more association ends.

People are often confused by the semantics of n-ary associations. N-ary associations have no representation in common programming languages. The suggestion is therefore to remodel the n-ary association using several plain associations.

- Suggested in [Fra03] and [Oes04].

Rule: **SpecAgg** Category: Style
Severity: 3-low Applies to: all areas
The association is a specialization of another association.

People are often confused by the semantics of specialized associations. The suggestion is therefore to model any restrictions on the parent association using constraints.

- Suggested in [Oes04].
- Value returned: name of the parent association.

C.9 Associationclass Rules

Rule: **AssocClass** Category: Style
Severity: 3-low Applies to: design
Avoid association classes.

Association classes have no representation in common programming languages. They defer the decision which class(es) eventually will be responsible to manage the association attributes.

The recommendation is to remodel the association class to only use regular classes and binary associations.

- Suggested in [Fra03] and [Oes04].

C.10 Generalization Rules

Rule: **NoChild** Category: Completeness
Severity: 1-high Applies to: all areas
The child end of the generalization is not connected to a model element.

Check the generalization and attach the child end to the proper model element, or remove the generalization from the model.

© 2002-2013. All rights reserved.

Rule: **NoParent** Category: Completeness
Severity: 1-high Applies to: all areas
The parent end of the generalization is not connected to a model element.

Check the generalization and attach the parent end to the proper model element, or remove the generalization from the model.

Rule: **TypeMismatch** Category: Correctness
Severity: 1-high Applies to: all areas
Parent and child of the generalization are not of the same type.

The child must be of the same type as the parent. This is a WFR of the UML.

- This is a WFR of the UML.
- Value returned: types of the parent and child elements.

C.11 Interfacerealization Rules

Rule: **NoSupplier** Category: Completeness
Severity: 1-high Applies to: all areas
The supplier end of the interface realization is not connected to an interface.

For a loose supplier end, either attach the end to the proper interface or delete the interface realization from the model.

If the supplier end is connected to anything else but an interface, connect it to the proper interface, or consider replacing the interface realization with a plain UML generalization, realization, or dependency.

Rule: **NoInterface** Category: Completeness
Severity: 1-high Applies to: all areas
The client end of the interface realization is not connected to a model element.

Check the interface realization and attach the client end to the proper model element, or remove the interface realization from the model.

© 2002-2013. All rights reserved.

C.12 Dependency Rules

Rule: **NoSupplier** Category: Completeness
Severity: 1-high Applies to: all areas
The supplier end of the dependency link is not connected to a model element.

Check the dependency and attach the supplier end to the proper model element, or remove the dependency link from the model.

Rule: **NoClient** Category: Completeness
Severity: 1-high Applies to: all areas
The client end of the dependency link is not connected to any model element.

Check the dependency and attach the client end to the proper model element, or remove the dependency link from the model.

C.13 Interaction Rules

Rule: **Alternatives** Category: Style
Severity: 3-low Applies to: all areas
The interaction models alternative sequences.

The interaction contains a combined fragment with 'alt', 'opt', or 'break' operator to model alternative execution sequences. The purpose of a sequence diagram is to show one scenario, not a set of different possible sequences. Activity diagrams are better suited for that purpose.

Consider using several sequence diagrams showing one scenario each, or use an activity diagram.

- Suggested in [Oes04].
- Value returned: name of the operator of the combined fragment.

C.14 Actor Rules

Rule: **NoAssoc** Category: Completeness
Severity: 1-high Applies to: all areas
The actor is not associated with any use cases, classes, or subsystems.

Without such associations, the actor is useless. Associate the actor with one or more use cases, or delete it from the model.

- Suggested in [Amb03].

© 2002-2013. All rights reserved.

Rule: **NaryAssoc** Category: Correctness
Severity: 2-med Applies to: all areas
The actor participates in an n-ary association.

An actor can only participate in binary associations. Replace the n-ary association with several binary associations.

- This is a WFR of the UML.

Rule: **Unnamed** Category: Correctness
Severity: 1-high Applies to: all areas
The actor has no name.

- See rule Unnamed for classes.

C.15 Usecase Rules

Rule: **Unused** Category: Completeness
Severity: 1-high Applies to: all areas
The use case is not used.

The use case is not associated with any actors, or included in or extending other use cases. Such a use case is useless. Associate it with an actor, attach it to another use case, or delete it from the model.

- Suggested in [Amb03].

Rule: **DupExPoint** Category: Correctness
Severity: 1-high Applies to: all areas
The use case has two or more extension points of the same name.

Rename the extension points so that they all have unique names.

- This is a WFR of the UML.
- Value returned: the name of the duplicate extension points.

© 2002-2013. All rights reserved.

Rule: **NoName** Category: Completeness
Severity: 1-high Applies to: all areas
The use case has an extension point without a name.

Check the extension points of the use case and make sure they all have a name.

- This is a WFR of the UML.

Rule: **NaryAssoc** Category: Correctness
Severity: 2-med Applies to: all areas
The use case participates in an n-ary association.

A use case can only participate in binary associations. Replace the n-ary association with several binary associations.

- This is a WFR of the UML.

Rule: **Unnamed** Category: Correctness
Severity: 1-high Applies to: all areas
The use case has no name.

- See rule Unnamed for classes.

Rule: **CyclicIncludes** Category: Correctness
Severity: 1-high Applies to: all areas
Use case directly or indirectly includes itself.

A use case cannot include use cases that directly or indirectly include it. Remove some include links to break the cycle.

- This is a WFR of the UML.
- You can view the graph of 'use case includes' in the graph structures window.
- Value returned: number of use cases participating in the cycle.

© 2002-2013. All rights reserved.

Rule: **FunctionalDecomp** Category: Style
Severity: 2-med Applies to: all areas
Use case both includes and is included in other use cases.

Several levels of include relations between use cases indicate a functional decomposition, which should not be part of requirements analysis.

- Suggested in [Amb03] to avoid more than two levels of include relations; this rule flags more than one level.

Rule: **Extends** Category: Style
Severity: 3-low Applies to: all areas
The use case is extending another use case.

The semantics of the extend relationship between use cases are often misunderstood, and there are no definite criteria when to use "extend" and when to use "include" relationships. The suggestion is to avoid using "extend" relationships in favor of the more intuitive "include".

- Suggested in [Amb03], [Oes04].

C.16 Statemachine Rules

Rule: **RegularTransition** Category: Correctness
Severity: 1-high Applies to: all areas
The protocol state machine contains a transition that is not a protocol transition.

Protocol state machines can only contain protocol transitions. Check the transitions and make sure they are all protocol transitions.

- This is a WFR of the UML.
- Value returned: name of the source and target states of the transition.

Rule: **ProtocolTransition** Category: Correctness
Severity: 1-high Applies to: all areas
The state machine contains a protocol transition.

Protocol transitions can only occur in protocol state machines. Check the transitions and make sure they are all regular transitions.

- This is a WFR of the UML.
- Value returned: name of the source and target states of the transition.

© 2002-2013. All rights reserved.

Rule: **HistoryState** Category: Correctness
Severity: 1-high Applies to: all areas
The protocol state machine contains a history state.

Protocol state machine cannot have deep or shallow history states. Check the state machine and remove all history states.

- This is a WFR of the UML.
- Value returned: name of the history state.

Rule: **StatesWithActivities** Category: Correctness
Severity: 1-high Applies to: all areas
The protocol state machine contains states with entry/exit/doactivities.

States in protocol state machines must not have any entry, exit, or do activities. Check the states and remove the activities.

- This is a WFR of the UML.
- Value returned: the name of the state with activities.

Rule: **TransWithEffects** Category: Correctness
Severity: 1-high Applies to: all areas
The protocol state machine contains transitions with effects.

Transitions in protocol state machines must not have any effects. Check the transitions and remove the effects.

- This is a WFR of the UML.
- Value returned: name of the source and target states of the transition.

C.17 Region Rules

Rule: **TooManyInitialStates** Category: Correctness
Severity: 1-high Applies to: all areas
The region has two or more initial states.

A region can have at most one initial state. Check the region and remove the surplus initial states.

- This is a WFR of the UML.
- Value returned: the number of initial states of the region.

© 2002-2013. All rights reserved.

Rule: **InitialAndFinalStates** Category: Style
Severity: 2-med Applies to: all areas
There is no initial or final state for the state machine.

The top-level region of a state machine should have one initial state and at least one final state so that the state machine has a well-defined beginning and end.

- Suggested in [JRH04].

Rule: **DupName** Category: Correctness
Severity: 1-high Applies to: all areas
The region has two or more states of the same name.

Distinctive states should have distinctive names. Duplicate names can also cause problems during code generation.

- Suggested in [RVR04].
- Value reported: name of the duplicate state.

Rule: **EmptyRegion** Category: Completeness
Severity: 1-high Applies to: all areas
The region has no states.

Add states to the region, or remove the region from the model.

Rule: **DeepHistory** Category: Correctness
Severity: 1-high Applies to: all areas
The region has two or more deep history states.

A region can have at most one deep history state. Check the region and delete the surplus history states.

- This is a WFR of the UML.
- Value returned: the number of deep history states of the region.

© 2002-2013. All rights reserved.

Rule: **ShallowHistory** Category: Correctness
Severity: 1-high Applies to: all areas
The region has two or more shallow history states.

A region can have at most one shallow history state. Check the region and delete the surplus history states.

- This is a WFR of the UML.
- Value returned: the number of shallow history states of the region.

C.18 State Rules

Rule: **NoIncoming** Category: Completeness
Severity: 1-high Applies to: all areas
State has no incoming transitions.

Without incoming transitions, the state can never be reached. Add one or more transitions to the state.

- Suggested in [Amb03].

Rule: **NoOutgoing** Category: Completeness
Severity: 1-high Applies to: all areas
State has no outgoing transitions.

Without outgoing transitions, the state can never be left. Check if this is merely an oversight or the actually intended behavior. In the former case, add the missing outgoing transition(s). In the latter case, consider adding an outgoing transition to a final state.

- Suggested in [Amb03].

Rule: **IllegalJoin** Category: Correctness
Severity: 1-high Applies to: all areas
Join states must have two or more incoming and exactly one outgoing transition.

- This is a WFR of the UML.
- Value returned: number of incoming and outgoing transitions of the join state.

© 2002-2013. All rights reserved.

Rule: **IllegalFork** Category: Correctness
Severity: 1-high Applies to: all areas
Fork states must have exactly one incoming and two or more outgoing transitions.

- This is a WFR of the UML.
- Value returned: number of incoming and outgoing transitions of the fork state.

Rule: **IllegalChoice** Category: Correctness
Severity: 1-high Applies to: all areas
A choice or junction state must have at least one incoming and one outgoing transition.

- This is a WFR of the UML.
- Value returned: number of incoming and outgoing transitions of the choice state.

Rule: **MissingGuard** Category: Correctness
Severity: 1-high Applies to: all areas
If there are two or more transitions from a choice state, they all must have guards.

A choice state realizes a dynamic conditional branch; the guards are required to evaluate the branch conditions. Check the outgoing transitions and add the missing guard(s).

- Suggested in [Amb03].
- Value returned: name of the target state of the transition without guard.

Rule: **IllegalInitial** Category: Correctness
Severity: 1-high Applies to: all areas
An initial state must have no incoming and exactly one outgoing transition.

- This is a WFR of the UML.
- Value returned: number of incoming and outgoing transitions of the initial state.

Rule: **IllegalFinal1** Category: Correctness
Severity: 1-high Applies to: all areas
A final state cannot have any outgoing transitions. Remove the outgoing transitions from the model.

- This is a WFR of the UML.
- Value returned: number of outgoing transitions of the final state.

© 2002-2013. All rights reserved.

Rule: **IllegalFinal2** Category: Correctness
Severity: 1-high Applies to: all areas
A final state cannot have any regions or entry/exit/state behavior.

Check the state and remove all regions, entry, exit, and do activities.

- This is a WFR of the UML.
- Value returned: number of regions and activities of the final state.

Rule: **IllegalEntryExit** Category: Correctness
Severity: 1-high Applies to: all areas
Entry or exit point is not owned by a top-level region. Move the entry or exit point to the top-level region of the state machine.

- This is a WFR of the UML.

Rule: **IllegalHistory** Category: Correctness
Severity: 1-high Applies to: all areas
The history state has two or more outgoing transitions.

A history state can have at most one outgoing transition. Check the history state and remove the surplus outgoing transitions.

- This is a WFR of the UML.
- Value returned: the number of outgoing transitions of the history state.

Rule: **Unnamed** Category: Completeness
Severity: 3-low Applies to: all areas
State has no name.

While the UML allows for anonymous states, adding a descriptive name to the state increases the readability and understandability of the diagram.

- Suggested in [Amb03].
- This rule does not check pseudo and final states. Their function is obvious, so they can be left unnamed.

© 2002-2013. All rights reserved.

Rule: **BadForkOutgoing** Category: Correctness
Severity: 1-high Applies to: all areas
Transitions from fork states must not have a guard or triggers. Check the outgoing transitions and remove the guards and triggers.

- This is a WFR of the UML.

Rule: **ForkTargetStates1** Category: Correctness
Severity: 1-high Applies to: all areas
Fork state has transitions to states in identical regions.

The transitions from a fork state must target states of different regions of a concurrent state. Check the outgoing transitions and make sure they all target different regions.

- This is a WFR of the UML.

Rule: **ForkTargetStates2** Category: Correctness
Severity: 1-high Applies to: all areas
Fork state has transitions to states in different concurrent states.

The transitions from a fork state in a state machine must target regions of the same concurrent state. Check the outgoing transitions and make sure they all target the same concurrent state.

- This is a WFR of the UML.

Rule: **JoinSourceStates1** Category: Correctness
Severity: 1-high Applies to: all areas
Join state has transitions from states in identical regions.

The transitions to a join state must originate from states of different regions of a concurrent state. Check the incoming transitions and make sure they all come from different regions.

- This is a WFR of the UML.

Rule: **JoinSourceStates2** Category: Correctness
Severity: 1-high Applies to: all areas
Join state has transitions from states in different concurrent states.

The transitions to a join state in a state machine must originate from regions of the same concurrent state. Check the incoming transitions and make sure they all come from the same concurrent state.

- This is a WFR of the UML.

© 2002-2013. All rights reserved.

Rule: **BadForkTarget** Category: Correctness
Severity: 1-high Applies to: all areas
Transitions from fork states in a state machine must not target a pseudo state. Check the outgoing transitions and make sure they do not point to any pseudo states.

- This is a WFR of the UML.

Rule: **BadJoinSource** Category: Correctness
Severity: 1-high Applies to: all areas
Transitions to join states in a state machine must not originate from a pseudo state. Check the incoming transitions and make sure they do not come from pseudo states.

- This is a WFR of the UML.

Rule: **BadIncoming** Category: Correctness
Severity: 1-high Applies to: all areas
Transitions to join states must not have triggers or guards. Check the incoming transitions and remove all triggers and guards.

- This is a WFR of the UML.

Rule: **BadOutgoing** Category: Correctness
Severity: 1-high Applies to: all areas
Transitions from pseudo states must not have any triggers. Check the outgoing transitions and remove all triggers and guards.

- This is a WFR of the UML.

Rule: **MissingTarget** Category: Correctness
Severity: 1-high Applies to: all areas
State has an outgoing transition not attached to a target state. Check the outgoing transitions of the state and attach any loose ends to the proper target states.

- Suggested in [JRH04].

© 2002-2013. All rights reserved.

Rule: **MissingSource** Category: Correctness
Severity: 1-high Applies to: all areas
State has an incoming transition not attached to a source state. Check the incoming transitions of the state and attach any loose ends to the proper source states.

- Suggested in [JRH04].

C.19 Activitygroup Rules

Rule: **EmptyGroup** Category: Completeness
Severity: 2-med Applies to: all areas
The activity group does not contain any nodes or subgroups.

Add nodes to the activity group, or delete it from the model.

Rule: **NoExpansionNode** Category: Completeness
Severity: 1-high Applies to: all areas
The expansion region has no expansion node.

An expansion region must have at least one input expansion node. This is a WFR of the UML.

C.20 Action Rules

Rule: **Unnamed** Category: Completeness
Severity: 3-low Applies to: all areas
Action has no name.

Give the action a descriptive name that describes its purpose.

Rule: **IsolatedAction** Category: Completeness
Severity: 1-high Applies to: all areas
The action has neither incoming nor outgoing edges.

If nothing goes in or comes out of the action, the action is useless. Add the missing edges, or delete the action.

© 2002-2013. All rights reserved.

C.21 Controlnode Rules

Rule: **IllegalInitial** Category: Correctness
Severity: 1-high Applies to: all areas
The initial node has incoming edges our outgoing object flows.

Check the initial node to make sure it has no incoming edges, and that all outgoing edges are control flows.

- This is a WFR of the UML.
- Value returned: the number of incoming edges and outgoing object flows.

Rule: **IllegalFinal** Category: Correctness
Severity: 1-high Applies to: all areas
Final nodes must not have any outgoing edges. Check the node and remove the outgoing edges.

- This is a WFR of the UML.
- Value returned: the number of outgoing edges of the final node.

Rule: **IllegalJoin1** Category: Correctness
Severity: 1-high Applies to: all areas
Join nodes must have exactly one outgoing edge.

- This is a WFR of the UML.
- Value returned: the number of outgoing edges of the join node.

Rule: **IllegalJoin2** Category: Correctness
Severity: 1-high Applies to: all areas
Outgoing edge of join is the wrong type.

If all of the join node's incoming edges are control flows, the outgoing edge must be a control flow. If there is at least one incoming object flow, the outgoing edge must be an object flow.

- This is a WFR of the UML.

Rule: **IllegalFork1** Category: Correctness
Severity: 1-high Applies to: all areas
Fork nodes must have exactly one incoming edge.

- This is a WFR of the UML.
- Value returned: the number of incoming edges of the fork node.

© 2002-2013. All rights reserved.

Rule: **MixedEdgeTypes** Category: Correctness
Severity: 1-high Applies to: all areas
Edges from or to decision, merge, or fork nodes must be of the same type.

The incoming and outgoing edges of a decision, merge, or fork node must either be all control flows or all object flows. A mixture of control and object flows attached to one such node is not allowed.

- This is a WFR of the UML.

Rule: **ForkOut** Category: Completeness
Severity: 2-med Applies to: all areas
Fork nodes should have two or more outgoing edges. Otherwise, there is no fork. Check the node and add the missing outgoing edges.

- Value returned: the number of outgoing edges of the fork node.

Rule: **IllegalDecision** Category: Correctness
Severity: 1-high Applies to: all areas
Decision nodes must have exactly one incoming edge.

- This is a WFR of the UML.
- Value returned: the number of incoming edges of the decision node.

Rule: **DecisionOut** Category: Completeness
Severity: 2-med Applies to: all areas
Decision nodes should have two or more outgoing edges, each with a guard.

Otherwise, there is no decision. Check the node and add the missing edges and/or guards.

- This is a WFR of the UML.
- Value returned: the number of outgoing edges of the node, and the number of guards on these edges.

Rule: **IllegalMerge** Category: Correctness
Severity: 1-high Applies to: all areas
Merge nodes must have exactly one outgoing edge.

- This is a WFR of the UML.
- Value returned: the number of outgoing edges of the merge node.

© 2002-2013. All rights reserved.

Rule: **NumIncoming** Category: Completeness
Severity: 2-med Applies to: all areas
Merge and join nodes should have two or more incoming edges. Otherwise, there is nothing to merge/join. Check the node and add the missing incoming edges.

- This is a WFR of the UML.
- Value returned: the number of incoming edges of the node.

Rule: **IsolatedNode** Category: Completeness
Severity: 1-high Applies to: all areas
The control node has neither incoming nor outgoing edges.

- See rule IsolatedAction for actions.

C.22 Objectnode Rules

Rule: **Unnamed** Category: Completeness
Severity: 3-low Applies to: all areas
The object node has no name.

- See rule Unnamed for actions.

Rule: **InOrOut** Category: Correctness
Severity: 1-high Applies to: all areas
The activity parameter node has both incoming and outgoing flows.

Activity parameter nodes must have either only incoming or only outgoing edges, but not both at the same time. Check the edges on the node and make sure they're all pointing in the same direction.

- This is a WFR of the UML.

Rule: **IsolatedNode** Category: Completeness
Severity: 1-high Applies to: all areas
The object node has neither incoming nor outgoing edges.

- See rule IsolatedAction for actions.

© 2002-2013. All rights reserved.

C.23 Pin Rules

Rule: **NoIncoming** Category: Completeness
Severity: 1-high Applies to: all areas
The input pin has no incoming edges.

Input pins require incoming edges to provide values to their actions. Add an incoming edge to the pin.

Rule: **NoOutgoing** Category: Completeness
Severity: 1-high Applies to: all areas
The output pin has no outgoing edges.

Output pins require outgoing edges to deliver values provided by their actions. Add in outgoing edge to the pin.

Rule: **IllegalInputPin** Category: Correctness
Severity: 1-high Applies to: all areas
The input pin has outgoing edges.

Only input pins on structured activity nodes can have outgoing edges. Check the pin and remove any outgoing edges.

- This is a WFR of the UML.
- Value returned: number of outgoing edges of the pin.

Rule: **IllegalOutputPin** Category: Correctness
Severity: 1-high Applies to: all areas
The output pin has incoming edges.

Only output pins on structured activity nodes can have incoming edges. Check the pin and remove any incoming edges.

- This is a WFR of the UML.
- Value returned: number of incoming edges of the pin.

© 2002-2013. All rights reserved.

Rule: **IllegalValuePin** Category: Correctness
Severity: 1-high Applies to: all areas
The value pin has an incoming edge.

Value pins provide their actions with constant values and thus require no incoming edges. Check the pin and remove any incoming edges.

- This is a WFR of the UML.
- Value returned: number of incoming edges of the pin.

C.24 Controlflow Rules

Rule: **DanglingCtrlFlow** Category: Completeness
Severity: 1-high Applies to: all areas
The control flow has no source or target node, or both.

Check the control flow and attach the proper source and target node, or delete the edge from the model.

C.25 Objectflow Rules

Rule: **DanglingObjectFlow** Category: Completeness
Severity: 1-high Applies to: all areas
The object flow has no source or target node, or both.

Check the object flow and attach the proper source and target node, or delete the edge from the model.

© 2002-2013. All rights reserved.

D: List of Matrices

This appendix lists the relation matrices that are available for UML2.x models. The matrices for the UML1.x metamodel are largely the same, or have equivalent counterparts, you can browse their definitions in the measurement catalog (see Section 4.13 "The View 'Catalog'") when analyzing UML1.x models.

Matrix: **Actor-Usecase** Row type: actor Column type: usecase
Association relationships between actors and use cases.

Shows which actor participates in which use case. A "1" indicates the actor in that row participates in the use case in that column.

Matrix: **Class_Gen** Row type: class Column type: class
Generalization relationships between classes (from child to parent).

Shows which class a given class inherits from. A "1" indicates the class in that row is a child of the class in that column.

Matrix: **Class_Assoc** Row type: class Column type: class
Association relationships between classes.

The numbers indicate how many associations the class in a row has with classes in the column. All associations (or aggregations or compositions) with navigability from the row class to the column class are counted.

Matrix: **Package_Dependencies** Row type: package Column type: package
Dependencies due to class/interface usage between packages.

Shows on which packages a given package depends on. A "1" indicates the package in that row has a class or interface that uses a class or interface of the package in that column.

- See package metric R for what constitutes usage between classes and/or interfaces.

Matrix: **Messages_Sent** Row type: lifeline Column type: lifeline
Messages sent between lifelines.

The numbers indicate how many messages the lifeline in a row sends to the lifelines in the columns.

© 2002-2013. All rights reserved.

E: Project File Format Definitions

This appendix contains the definitions of the project file formats for the metamodel definition file, the XMI transformation file, and the metrics definition files. Being XML files, the definition is presented in the form of a DTD (document type definition).

Metamodel definition file format

```
<!ELEMENT sdmetricsmetamodel ( modelelement+ ) >
<!ATTLIST sdmetricsmetamodel
         version CDATA #REQUIRED>

<!ELEMENT modelelement (#PCDATA|attribute)*>
<!ATTLIST modelelement
         name CDATA #REQUIRED
         parent CDATA "sdmetricsbase">

<!ELEMENT attribute (#PCDATA)>
<!ATTLIST attribute
         name CDATA #REQUIRED
         type (ref|data) "data"
         multiplicity (one|many) "one">
```

XMI transformation file format

```
<!ELEMENT xmitransformations ( xmitransformation+ ) >
<!ATTLIST xmitransformations
         version CDATA #REQUIRED
         requirexmiid (true|false) "true">

<!ELEMENT xmitransformation (trigger*) >
<!ATTLIST xmitransformation
         modelelement CDATA #REQUIRED
         xmipattern CDATA #REQUIRED
         recurse (true|false) "false"
         requirexmiid (true|false) #IMPLIED
         condition CDATA #IMPLIED>

<!ELEMENT trigger EMPTY>
<!ATTLIST trigger
         name CDATA #REQUIRED
         type (attrval|cattrval|gcattrval|ctext|reflist|constant|ignore|
xmi2assoc) #REQUIRED
         attr CDATA #IMPLIED
         src CDATA #IMPLIED
         linkbackattr CDATA #IMPLIED>
```

Metric definition file format

```
<!ELEMENT sdmetrics (metric|set|matrix|rule|wordlist|reference|term|
                     metricprocedure|setprocedure|ruleprocedure)+ >
<!ATTLIST sdmetrics
         version CDATA #REQUIRED
         ruleexemption CDATA #IMPLIED
         exemptiontag CDATA #IMPLIED
         exemptionvalue CDATA #IMPLIED>
```

© 2002-2013. All rights reserved.

```
<!ELEMENT metric (description?,(projection|setoperation|compoundmetric|
        attributevalue|nesting|signature|connectedcomponents|count|compare|
        subelements|filtervalue|substring))>
<!ATTLIST metric
        name CDATA #REQUIRED
        domain CDATA #REQUIRED
        inheritable (true|false) "false"
        category CDATA #IMPLIED
        internal (true|false) "false">

<!ELEMENT set (description?,(projection|compoundset|subelements|compare))>
<!ATTLIST set
        name CDATA #REQUIRED
        domain CDATA #REQUIRED
        inheritable (true|false) "false"
        multiset (true|false) "false">


<!ELEMENT matrix (description?,(projection|setoperation|compoundset))>
<!ATTLIST matrix
        name CDATA #REQUIRED
        from_row_type CDATA #REQUIRED
        to_col_type CDATA #REQUIRED>

<!ELEMENT rule (description?,(violation|cycle|projection|compoundset|
        valueset|compare))>
<!ATTLIST rule
        name CDATA #REQUIRED
        domain CDATA #REQUIRED
        inheritable (true|false) "false"
        category CDATA #IMPLIED
        severity CDATA #IMPLIED
        applies_to CDATA #IMPLIED
        disabled (true|false) "false">

<!ELEMENT wordlist (entry*)>
<!ATTLIST wordlist
        name CDATA #REQUIRED
        ignorecase (true|false) "false">

<!ELEMENT entry EMPTY>
<!ATTLIST entry
        word CDATA #REQUIRED>

<!ELEMENT description (#PCDATA)>

<!ENTITY % scopes "same|other|higher|lower|sameorhigher|sameorlower|samebranch|
        notsamebranch|containedin|notcontainedin|idem|notidem" >

<!ENTITY % filteratts
        'target CDATA #IMPLIED
        targetcondition CDATA #IMPLIED
        element CDATA #IMPLIED
        eltype CDATA #IMPLIED
        scope (%scopes;) #IMPLIED
        condition CDATA #IMPLIED'
>

<!ENTITY % sumatts
 'sum CDATA #IMPLIED
        stat (sum|min|max) "sum"'
>
```

© 2002-2013. All rights reserved.

```
<!ENTITY % ruleatts
 'precondition CDATA #IMPLIED
        mincnt CDATA #IMPLIED
        value CDATA #IMPLIED'
>

<!ELEMENT projection EMPTY>
<!ATTLIST projection
        relation CDATA #IMPLIED
        relset CDATA #IMPLIED
        %sumatts;
        set CDATA #IMPLIED
        valueset CDATA #IMPLIED
        exclude_self (true|false) "false"
        recurse (true|false) "false"
        nesting (true|false) "false"
        %ruleatts;
        %filteratts;>

<!ELEMENT compoundmetric EMPTY>
<!ATTLIST compoundmetric
        term CDATA #IMPLIED
        condition CDATA #IMPLIED
        alt CDATA #IMPLIED
        fallback CDATA #IMPLIED>

<!ELEMENT attributevalue EMPTY>
<!ATTLIST attributevalue
        attr CDATA #REQUIRED
        element CDATA #IMPLIED>

<!ELEMENT signature EMPTY>
<!ATTLIST signature
        set CDATA #REQUIRED
        name CDATA #IMPLIED
        prologue CDATA #IMPLIED
        value CDATA #IMPLIED
        separator CDATA #IMPLIED
        epilogue CDATA #IMPLIED
        %filteratts;>

<!ELEMENT connectedcomponents EMPTY>
<!ATTLIST connectedcomponents
        set CDATA #REQUIRED
        nodes CDATA #REQUIRED>

<!ELEMENT valuesetcount EMPTY>
<!ATTLIST valuesetcount
        set CDATA #REQUIRED>

<!ELEMENT count EMPTY>
<!ATTLIST count
        term CDATA #REQUIRED
        set CDATA #REQUIRED
        relset CDATA #REQUIRED
        %filteratts;>
```

© 2002-2013. All rights reserved.

```
<!ELEMENT compare EMPTY>
<!ATTLIST compare
        term CDATA #IMPLIED
        set CDATA #IMPLIED
        with CDATA #IMPLIED
        comp CDATA #IMPLIED
        return_element (true|false) "false"
        exclude_self (true|false) "true"
        %ruleatts;
        %filteratts;>

<!ELEMENT setoperation EMPTY>
<!ATTLIST setoperation
        set CDATA #REQUIRED
        %sumatts;
        %filteratts;>

<!ELEMENT nesting EMPTY>
<!ATTLIST nesting
        relation CDATA #REQUIRED>

<!ELEMENT subelements EMPTY>
<!ATTLIST subelements
        set CDATA #IMPLIED
        exclude_self (true|false) "true"
        valueset CDATA #IMPLIED
        %filteratts;
        %sumatts;>

<!ELEMENT filtervalue EMPTY>
<!ATTLIST filtervalue
        relation CDATA #IMPLIED
        relset CDATA #IMPLIED
        value CDATA #IMPLIED
        %filteratts;>

<!ELEMENT substring EMPTY>
<!ATTLIST substring
        source CDATA #REQUIRED
        separator CDATA #REQUIRED
        position CDATA #IMPLIED
        endseparator CDATA #IMPLIED
        limit CDATA #IMPLIED
        result CDATA #IMPLIED>

<!ELEMENT compoundset EMPTY>
<!ATTLIST compoundset
        set CDATA #REQUIRED
        cum CDATA #IMPLIED
        valueset CDATA #IMPLIED
        exclude_self (true|false) "false"
        %ruleatts;
        %filteratts;>

<!ELEMENT violation EMPTY>
<!ATTLIST violation
        condition CDATA #REQUIRED
        value CDATA #IMPLIED>
```

© 2002-2013. All rights reserved.

```
<!ELEMENT valueset EMPTY>
<!ATTLIST valueset
        set CDATA #REQUIRED
        precondition CDATA #IMPLIED
        mincnt CDATA #IMPLIED>

<!ELEMENT cycle EMPTY>
<!ATTLIST cycle
        nodes CDATA #REQUIRED
        minnodes CDATA #IMPLIED>

<!ELEMENT reference (#PCDATA|description)*>
<!ATTLIST reference
        tag CDATA #REQUIRED>

<!ELEMENT term (#PCDATA|description)*>
<!ATTLIST term
        name CDATA #REQUIRED>

<!ELEMENT metricprocedure EMPTY>
<!ATTLIST metricprocedure
        name CDATA #REQUIRED
        class CDATA #REQUIRED>
<!ELEMENT setprocedure EMPTY>
<!ATTLIST setprocedure
        name CDATA #REQUIRED
        class CDATA #REQUIRED>

<!ELEMENT ruleprocedure EMPTY>
<!ATTLIST ruleprocedure
        name CDATA #REQUIRED
        class CDATA #REQUIRED>
```

Grammar of expressions in metric definitions

The grammar for expressions in metric and set definitions is presented here using an extended Backus-Naur Formalism (EBNF). Note that the production rule FCT_LITERAL for function names recognized by the expression parser only lists the default functions provided by SDMetrics.

```
Expression             = UptoExpression.

UptoExpression         = OrExpression {"upto" OrExpression | "topmost"
OrExpression}.

OrExpression           = AndExpression {"|" AndExpression | "or" AndExpression}.

AndExpression          = EqualExpression {"&" EqualExpression |
                                          "and" EqualExpression}.

EqualExpression        = RelationalExpression
                           ["!=" RelationalExpression |
                             "=" RelationalExpression].
```

© 2002-2013. All rights reserved.

```
RelationalExpression  = AdditiveExpression [ "<" AdditiveExpression |
                        ">" AdditiveExpression | "<=" AdditiveExpression |
                        ">=" AdditiveExpression |"lt" AdditiveExpression |
                        "le" AdditiveExpression | "gt" AdditiveExpression |
                        "ge" AdditiveExpression | "->" AdditiveExpression |
                        "in" AdditiveExpression |
                        "startswith" AdditiveExpression |
                        "endswith" AdditiveExpression |
                        "onlist" AdditiveExpression].

AdditiveExpression    = MultiplicativeExpression {
                        "+" MultiplicativeExpression |
                        "-" MultiplicativeExpression}.

MultiplicativeExpression
                      = PowerExpression {
                        "*" PowerExpression | "/" PowerExpression }

PowerExpression       = DotExpression {"^" DotExpression |
                        "->" DotExpression |"in" DotExpression }.

DotExpression         = UnaryExpression {"." UnaryExpression}.

UnaryExpression       = "+" UnaryExpression | "-" UnaryExpression |
                        "!" UnaryExpression | "not" UnaryExpression |
                        FCT_LITERAL UnaryExpression | "self" |
                        "(" Expression ")".

Constant              = STRING_LITERAL | IDENTIFIER | INTEGER_LITERAL
                        | FLOATING_POINT_LITERAL.

FCT_LITERAL           = "ln" | "exp" | "sqrt" | "abs" |
                        "ceil" | "floor" | "round" |
                        "startswithcapital" | "startswithlowercase" |
                        "islowercase" | "tolowercase" |
                        "length" | "size" | "flatsize" |
                        "isunique" | "typeof" | "qualifiedname".

INTEGER_LITERAL       = DECIMAL_LITERAL.

DECIMAL_LITERAL       = DIGIT {DIGIT}.

FLOATING_POINT_LITERAL= ({DIGIT} "." DIGIT {DIGIT} [EXPONENT])
                        | DIGIT {DIGIT} [EXPONENT].

EXPONENT              = ("e"|"E") ["+"|"-"] DIGIT {DIGIT}.

STRING_LITERAL        = "'" {LETTER|DIGIT} "'".

IDENTIFIER            = LETTER {LETTER|DIGIT}.

LETTER                =  "_" | "A".."Z" | "a".."z".

DIGIT                 = "0" | "1" | "2" | "3" | "4" | "5" |
                        "6" | "7" | "8" | "9".
```

© 2002-2013. All rights reserved.

F: Glossary

Completeness
: Design rules of the "Completeness" category raise issues that hint at incomplete design. This highlights model elements that still need some work. For example, empty packages, unused classes, unnamed attributes, parameters without a type, etc.

Correctness
: Design rules of the "Correctness" category raise issues that constitute illegal design. For example, violation of well-formedness rules (WFR) of the UML.

Diagram
: Diagram metrics pertain to the diagrams of the UML model. There are two types of diagram metrics:

1. Diagram metrics that count how often a class, package, etc. appears on the diagrams in the model. Similar to export coupling, the more often a model element appears on diagrams, the more important is the role of the model element.
   You may also look out for elements which do not appear on any diagrams. This could indicate that the diagrams are not complete, or they may be the result of an incomplete delete operation.
2. Metrics that count the number of model elements on a diagram. These measure the size of the diagram. You may consider reorganizing large diagrams into several smaller ones, e.g., based on the 7 +/-2 rule of the amount of information that people can deal with at a time [Amb03].

On a technical note, many UML modeling tools use a proprietary solution to store diagram layout information in the XMI file. If you do not obtain diagram metrics for your model, you need a special XMI transformation file. Check the SDMetrics website if there is one available for your UML tool.

Naming
: Design rules of the "Naming" category raise issues concerning the names assigned to model elements. For example, adherence to naming conventions for capitalization, use of keywords in element names.

Severity
: The severity of a design rule indicates how critical a violation is. The rules distinguish three levels of severity:

- 1-high - violation of the rule constitutes illegal design, or poor design practices with a strong negative impact on system quality. The issue should be resolved under all circumstances.
- 2-med - violation of the rule may negatively impact system quality. The issue should be resolved if there is no justification for the violation.
- 3-low - violation of the rule is not likely to have severe consequences, but perfectionists will still want to address the issue.

Style
: Design rules of the "Style" category raise design issues that are considered bad practice. While these issues do not indicate illegal design, they may be detrimental to system quality in the long run. For example, circular dependencies among packages, a class referencing one of its subclasses, long parameter lists, etc.

© 2002-2013. All rights reserved.

WFR

Well-formedness rules. The UML standards [OMG03] [OMG05] define a set of well-formedness rules, or constraints, that any valid UML model must comply with.

© 2002-2013. All rights reserved.

G: References

[AGE95]
F. Abreu, M. Goulao, R. Esteves, "Toward the Design Quality Evaluation of Object-Oriented Software Systems", 5th International Conference on Software Quality, Austin, Texas, October 1995.
[Amb03]
S. Ambler, "The Elements of UML Style", Cambridge University Press, 2003.
A comprehensive collection of style guidelines for the UML. Also available online at www.agilemodeling.com/style
[BEGR00]
S, Benlarbi, K. El Emam, N. Goel, S. Rai, "Thresholds for Object-Oriented Measures", Proceedings of ISSRE2000, 24-37, 2000.
[BDM97]
Briand, Devanbu, Melo, "An Investigation into coupling measures for object-oriented designs", Proceedings of the 19th International Conference on Software Engineering, ICSE '97, Boston, 412-421, 1997.
[BMM98]
W. Brown, R. Malveau, H, McCormick, T Mowbray, "AntiPatterns: Refactoring Software, Architectures, and Projects in Crises", Wiley, 1998.
[BMW02]
L. Briand, W. Melo, J. Wuest, "Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects", IEEE Transactions on Software Engineering, 28 (7), 706-720, 2002.
Also available from http://www.sdmetrics.com/Refs.html
[BW02]
L. Briand, J. Wuest, "Empirical Studies of Quality Models in Object-Oriented Systems", Advances in Computers Vol. 59, 97-166, 2002.
Also available from http://www.sdmetrics.com/Refs.html
[BWDP00]
L. Briand, J. Wuest, J. Daly, V. Porter, "A Comprehensive Empirical Validation of Product Measures for Object-Oriented Systems". Journal of Systems and Software 51, p. 245-273, 2000.
Also available from http://www.sdmetrics.com/Refs.html
[BWL01]
L. Briand, J. Wuest, H. Lounis, "Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs, Empirical Software Engineering: An International Journal, Vol 6, No 1, 11-58, 2001.
Also available from http://www.sdmetrics.com/Refs.html
[CK94]
S. Chidamber, C. Kemerer, "A Metrics Suite for Object-Oriented Design", IEEE Transactions on Software Engineering, 20 (6), 476-493, 1994.
[CK98]
S. Chidamber, D. Darcy, C. Kemerer, "Managerial use of Metrics for Object-Oriented Software: An Exploratory Analysis", IEEE Transactions on Software Engineering, 24 (8), 629-639, 1998.
[Fow99]

© 2002-2013. All rights reserved.

M. Fowler, "Refactoring: Improving the Design of Existing Code", Addison Wesley, 1999.
[FP96]
N. Fenton, S. Pfleeger, "Software Metrics: A Practical and Rigorous Approach". International Thompson Computer Press, 1996.
[Fra03]
D. Frankel, "Model Driven Architecture: Applying MDA to Enterprise Computing", Wiley, 2003.
[ISO9126]
ISO/IEC FCD 9126-1.2, "Information Technology - Software Product. Quality- Part 1: Quality Model", 1998.
[JRH04]
M. Jeckle, C. Rupp, J. Hahn, B. Zengler, S. Queins, "UML 2 glasklar", Carl Hanser Verlag, 2004.
[Lan03]
C. Lange, "Empirical Investigations in Software Architecture Completeness", Master's Thesis, Department of Mathematics and Computing Science, Technical University Eindhoven, 2003.
[LH93]
W. Li, S. Henry, "Object-Oriented Metrics that Predict Maintainability", J. Systems and Software, 23 (2), 111-122, 1993.
[LC94]
A. Lake, C. Cook, "Use of factor analysis to develop OOP software complexity metrics", Proc. 6th Annual Oregon Workshop on Software Metrics, Silver Falls, Oregon, April 1994.
[LK94]
M. Lorenz, J. Kidd, "Object-oriented Software Metrics", Prentice Hall, 1994.
[LLW95]
Y. Lee, B. Liang, S. Wu, F. Wang, "Measuring Coupling and Cohesion of an Object-Oriented Program Based On Information Flow", Proc. International Conference on Software Quality (ICSQ '95), 81-90, 1995.
[Mar03]
R. Martin, "Agile Software Development: Principles, Patterns, and Practices", Prentice Hall, 2003.
[McC76]
T. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, 2 (12), 308-320, 1976.
[MGP03]
D. Miranda, M. Genero, M. Piattini, "Empirical validation of metrics for UML statechart diagrams", 5th Internation Conference on Enterprise Informations Systems (ICEIS03), 1, p. 87-95, 2003.
[NP98]
P. Nesi, T. Querci, "Effort estimation and prediction of object-oriented systems", Journal of Systems and Software 42, p. 89-102, 1998.
[Oes04]
Bernd Oesterreich, "Die UML 2.0 Kurzreferenz fuer die Praxis", Oldenbourg Verlag, 2004.
[OMG03]
Object Management Group, "OMG Unified Modeling Language Specification", Version 1.5, OMG Adopted Formal Specification formal/03-03-01, 2003.
[OMG05]
Object Management Group, "UML 2.0 Superstructure Specification", OMG Adopted Formal

© 2002-2013. All rights reserved.

Specification formal/05-07-04, 2005.
[OMG10]
Object Management Group, "OMG Systems Modeling Language", Version 1.2, OMG Document Number formal/2010-06-02, 2010.
[Rie96]
A. Riel, "Object-Oriented Design Heuristics", Addison Wesley, 1996.
[RVR04]
A. Ramirez, P. Vanpeperstraete, A. Rueckert, K. Odutola, J. Bennett, L. Tolke, M. van der Wulp, "ArgoUML User Manual v0.16", 2004.
Available from http://argouml.tigris.org.
[TSM92]
D. Tegarden, S. Sheetz, D. Monarchi, "The Effectiveness of Traditional Software Metrics for Object-Oriented Systems", in: J. Nunamaker Jr, R. Sprague (eds.), Proceedings of the 25th Hawaii International Conference on Systems Sciences, Vol. IV, IEEE Computer Society Press, 359-368, Jan. 1992.

© 2002-2013. All rights reserved.