SiDiff 2.0 – Compare-Functions (EMF)

Pit Pietsch, Timo Kehrer

8. März 2009

Inhaltsverzeichnis

Document Changes

Date	Changes
17.02.09	Initial Version
02.03.09	Beschreibung zur Erstellung von Testmodellen hinzugefügt (initial)
08.03.09	New todos and comments added

1 todo + comments

Einige Inhalte dieses Dokuments beziehen sich auf ALLE Compare-Functionen, nicht nur auf EMF-spezifische. Sollen wir in dieses Dokument somit auch alle CFs aufnehmen.!? (tk)

Strengere Konventionen für die Bezeichnung von CompareFunctions?
Bsp.: ParentsEqualType und ChildrenEqualViewingTypes. Beide Compare-Functionen meinen doch dasselbe, oder? Entweder mit oder ohne Viewing (ganz generell). (tk)

ValueAdmeasue aufräumen (pp)

warum existiert compareStringViewingSimilarityUsingLCSIgnoringCase/...ConsideringCase aber keine vergleichbare trennung bei IndexOf? (pp)

wir sollten spezifizieren und überprüfen (und dokumentieren) für welche eTypes eine comparefunction anwendbar ist (daraus ergeben sich ja auch die aequivalenzklassen zum testen). die NumericAttributEquals...-funktionen haben einen derartigen mechanismus bereits hartverdrahtet implementiert. (pp)
tk kümmert sich darum (tk)
Zusätzliche Anmerkung (tk): Eine, wie ursprünglich angedacht, einfache Matrix (CompareFunktion X AttributTyp) ist unter Umständen nicht ausreichend. Im Prinzip müssen hier auch noch die Parameter als 3.te Dimension aufgenommen werden. Oder sollen wir eben eine solche Kompatibilitäts-Matrix auch noch für Parameter erstellen?
Was passiert bei fehlerhaften regulaeren Ausdruecken? Gibt es die ueberhaupt, oder heisst falsch lediglich, dass der RegEx nicht der Intention des Benutzers entspricht? (tk)

macht es sinn neben den aequvialenzklassen die im moment die ausgangsituation für die testergebnisse festlegt auch ergebnissorientierte aequivalenzklassen zu definieren? beispielsweise: erzeuge eine similarity von 0.5 mit vergleichsfunktion x,y? (pp)

ExplicitMatch/MaximumSimilarity: brauchen wir beide, oder kann explicitMatch nicht einfach auch weg? (pp)
ExplicitMatch wird rausgenommen. Evtl späters name-refactoring (pp/tk)

-erledigt
Vorschlag für Name-Refactoring: Maximum-Similarity und MinimumSimilarity evtl. nicht sofort intuitiv einleuchtend? Wie wäre es mit ExplicitMatch und ExplicitDismatch? (tk)

Vergleich von Attributen: Wird hier schon unterstellt, dass die beiden zu vergleichenden Elemente vom selben Typ sind? Haette Auswirkungen auf die Aequivalenzklassen...(tk)
typgleichheit ist über die dedicatedClass sichergestellt.(pp)
-erledigt

equalID kann als comparefunction doch eigentlich entfallen: spezialfall von compareAttributeUsingEquals? (pp)
equalID wird rausgenommen (pp/tk)
-erledigt

parameter: der inhalt eines parameter-strings ist abhängig von dem typ des zu vergleichenden attributs. die verarbeitung des parameters wird in der abstracten typ oberklasse definiert.

beispiel: StringAttributeUsingEquals erbt das wissen um die Verarbeitung von abstractStringAttribute, wo festgelegt wird, wie der Parameter auszulesen ist. dieser mechanismus steht ja bei unser idee so nicht mehr zur verfügung... gleiches gilt für den boolean parameter sensitive.

evtl. auslagern der parameterfunktionalität? eine utilklasse der man parameter + eType übergibt, die ihn entsprechend zerlegt und das Ergebniss zurückliefert? (pp)

lösung: AbstractAttribute zerlegt parameter und behandelt alle parameter die auf dieser ebene relevant sind. weitere parameter werden in einer liste gespeichert und können in der implementierenden klasse abgefragt werden (pp/tk)
–erledigt

2 Testdatenerstellung

Im Hinblick auf die Testdaten zum Testen der Compare-Funktionen sind zwei Kategorien von Testdaten zu unterscheiden: Testmodelle und Parameter der jeweiligen Compare-Funktion. Die in diesem Abschnitt beschriebene Erstellung von Testdaten bezieht sich dabei auf die Erstellung von Testmodellen.

2.1 Testmetamodell

Für die Erstellung von Testmodellen wurde ein eigenes Metamodell, im Folgenden als Testmetamodell bezeichnet, entworfen (Bundle `org.sidiff.common.testmetamodel`). Das Testmetamodell ist dabei möglichst schlank gehalten, beinhaltet aber dennoch die für reale Metamodelle typischen Strukturen: Komponenten (Blöcke) und gerichtete Beziehungen (Linien) zwischen den Komponenten. Zudem wurde darauf geachtet, alle primitiven EMF-Datentypen zu verwenden. Abbildung 2.1 zeigt das Testmetamodell in der bekannten Ecore-Diagramm-Notation.

2.2 Erstellung von Testmetamodell-Instanzen

Für die Erstellung von Testmetamodell-Instanzen exisitieren verschiedene Möglichkeiten, von denen hier zwei näher betrachtet werden sollen:

1. Mittels des durch EMF bereit gestellten, generischen Ecore-Editors. Hat den Nachteil, dass es teilweise mühsam sein kann, die Graph-Struktur der Modelle zu erfassen. Das gilt sowohl für die Erstellung der Testdaten, als auch für die Nachvollziehbarkeit der einzelnen Testfälle.

2. Instanzen werden in UML-Notation mittels eines UML-Tools spezifiziert und über einen Konverter nach EMF transformiert.

Aus Gründen der Nachvollziehbarkeit und der leichteren Erstellung von Testmodellen werden wir die 2.te Variante anwenden. Als UML-Werkzeug ist der IBM Rational Software Modeler (RSM) in der Version 7.5 zu benutzen. Im Folgenden werden die einzelnen Schritte zur Erstellung von Testmodellen beschrieben.

2.2.1 Modellerstellung in RSM

Alle Testmodelle sind im RSM zu erstellen. Hierzu existieren zwei RSM-Projekte, TestmetamodelProfile und TestModels, welche im Verzeichnis `rsm/workspace` des Bundles `org.sidiff.common.testmodels` zu finden sind. Das Projekt TestmetamodelProfile definiert einige zur Identifikation von Komponenten benötigte Stereotypen (s. unten). Die eigentlichen Testmodelle werden im Projekt TestModels spezifiziert.

Pakethierarchie Testmodelle werden in einer Pakethierarchie organisiert, welche folgenden Konventionen genügt (s. Abbildung 2.2.1):

- Auf oberster Ebene befindet sich das Paket, welches die JUnit-Testfälle beinhaltet. Das Paket heißt wie das jeweilige OSGI-Bundle (z.B. org.sidiff.compare.comparefunctions.emf.test)

- Auf der nächsttieferen Ebene befindet sich das Paket, welche alle Testdaten für eine spezielle Compare-Funktion Test-Suite (z.B. CompareAttributeUsingLCS-Test) beinhaltet.

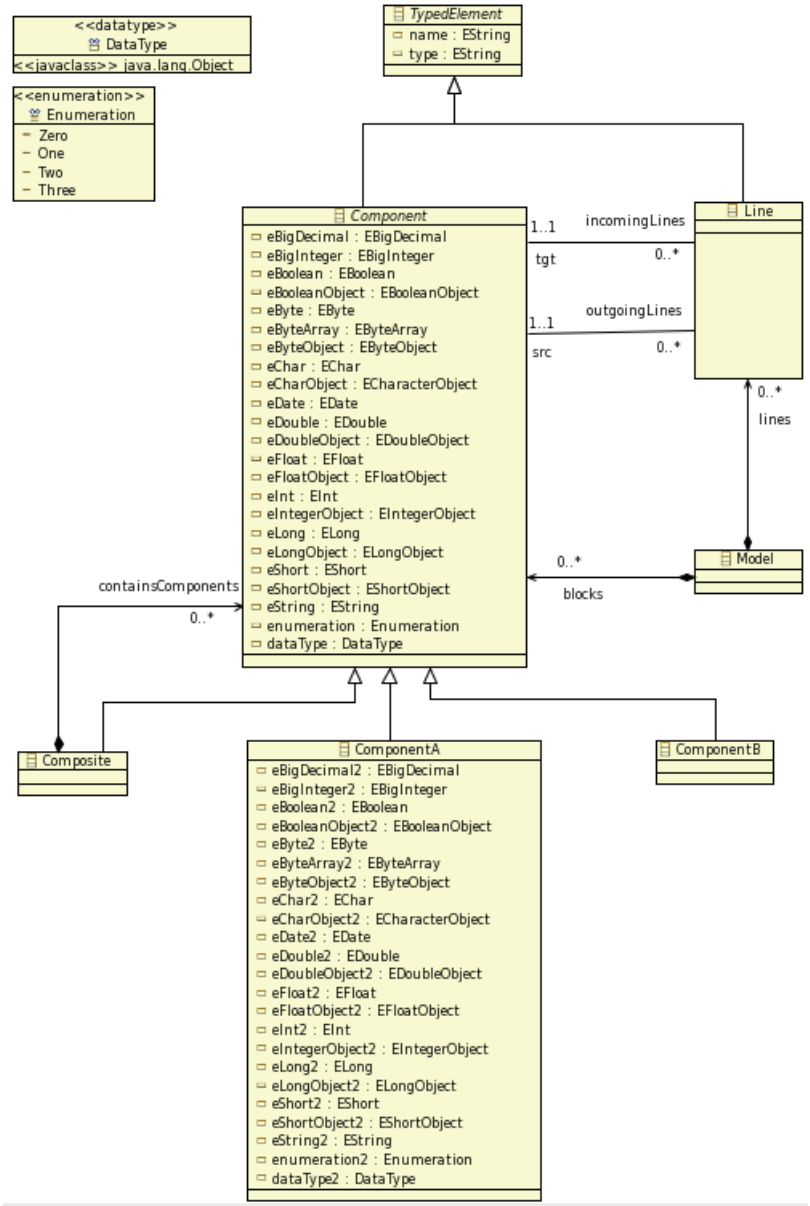

Abbildung 1: Testmetamodell

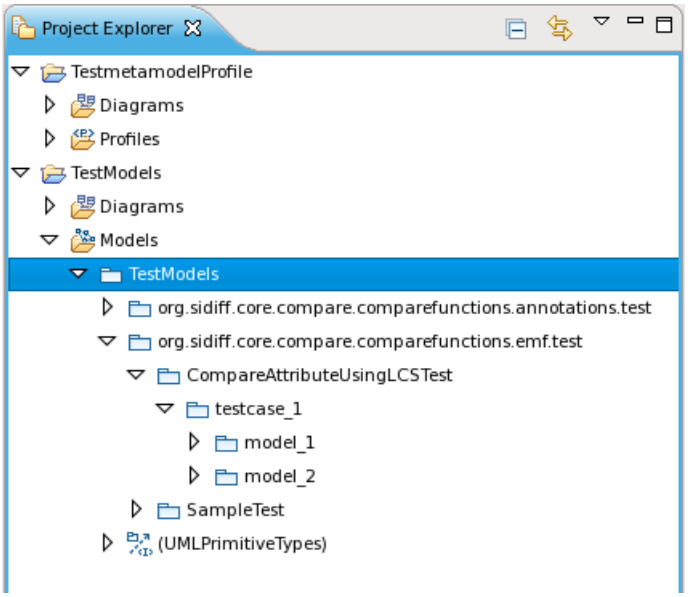

Abbildung 2: Exemplarische Paket-Hierarchie

- Auf der nächsten Hierarchie-Ebene befinden sich Pakete für alle Testfälle. Diese Pakete werden entsprechend dem Namensmuster $testcase_{<}NR. >$ benannt, wobei $< NR. >$ einer sequenziell hochgezählten Nummer entspricht.
- Für jeden Testfall werden schließlich die Eingabemodelle (in der Regel zwei) spezifiziert, welche sich in Unterpakten model-1 bzw. model-2 befinden.

Spezifikation der eigentlichen Testmodelle Die eigentlichen Testmodelle werden in den oben beschriebenen Paketen model-1 bzw. model-2 spezifiziert. Zur grafischen Visualisierung ist ein Klassendiagramm zu verwenden. Abbildung 2.2.1 zeigt ein Exemplarisches Testmodell in Klassendiagramm-Notation, Abbildung 2.2.1 die dazugehörige Modellstruktur, wie sie im RSM im Projekt-Explorer dargestellt wird.

Die im Testmetamodell definierten Metaklassen werden durch folgende UML-Konstrukte umgesetzt:

- ComponentA: Entspricht einer Klasse mit dem Stereotyp ComponentA (welcher in TestmetamodelProfile als Erweiterung der UML-Metaklasse Class definiert ist).
- ComponentB: Entspricht einer Klasse mit dem Stereotyp ComponentB (welcher in TestmetamodelProfile als Erweiterung der UML-Metaklasse Class definiert ist).

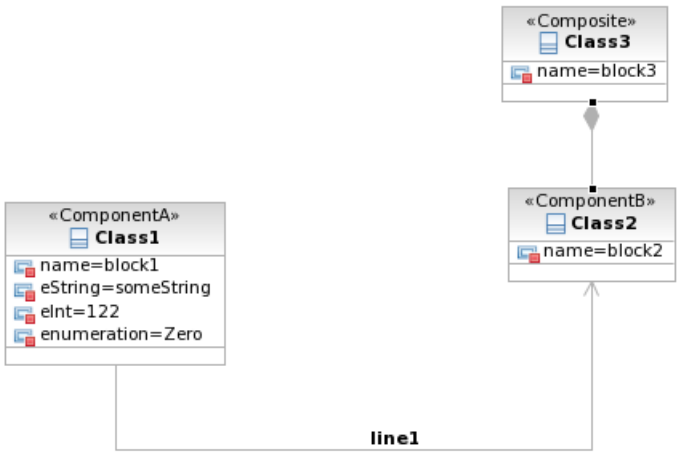

Abbildung 3: Exemplarisches Testmodell in UML-Klassendiagramm Notation

- Composite: Entspricht einer Klasse mit dem Stereotyp Composite (welcher in TestmetamodelProfile als Erweiterung der UML-Metaklasse Class definiert ist). Die Beziehung containsComponent ist durch eine Kompositionsbeziehung umzusetzen.
- Line: Lines werden durch gerichtete Assoziationen abgebildet. Rollennamen und Kardinalitäten brauchen nicht spezifiziert zu werden.

Einschränkungen und Konventionen Folgende Einschränkungen und Konventionen sind zu beachten:

- Attribute: Die Zuordnung von durch das Testmetamodell definierten Attributen und Attributwerten geschieht über die Benennung eines UML-Attributs nach folgendem Muster: $< attr - name >=< attr - value >$, wobei $< attr - name >$ dem Namen eines Attributs im Testmetamodell und $< attr - value >$ dem dazugehörigen Attributwert entspricht. Für einen Testfall nicht relevante Attribute müssen nicht spezifiziert zu werden.
- Eindeutige Identifizierer: Modellelemente müssen eindeutig identifizierbar sein. Für alle Komponenten (ComponentA, ComponentB, Composite) ist daher das Attribut name unbedingt anzugeben. Auch Lines müssen identifizierbar sein. Als eindeutiger Identifizierer dient hier der Assoziationsname.

2.2.2 Modellexport und Generierung von EMF-Modellen

Um aus den im RSM erstellten UML-Modellen EMF-Modelle als die eigentlichen Testdaten der JUnit-Testfälle zu erhalten, sind diese zunächst im Eclipse UML2-Format

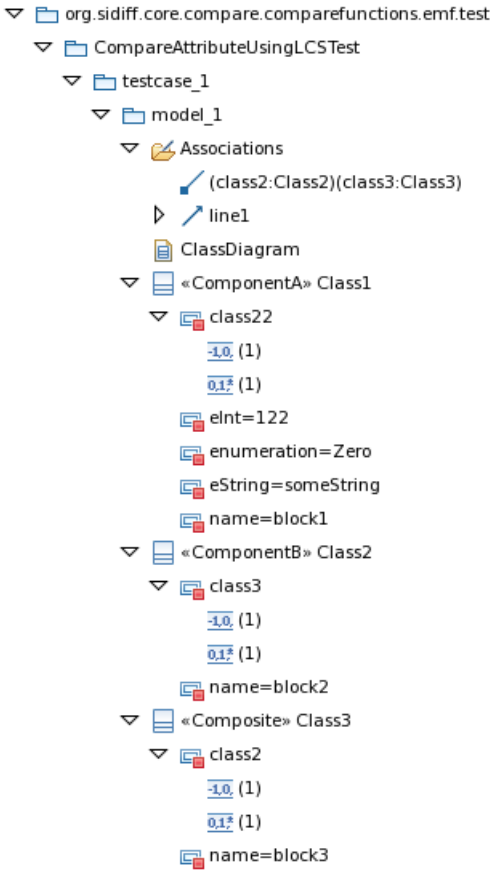

Abbildung 4: Exemplarische Modellstruktur

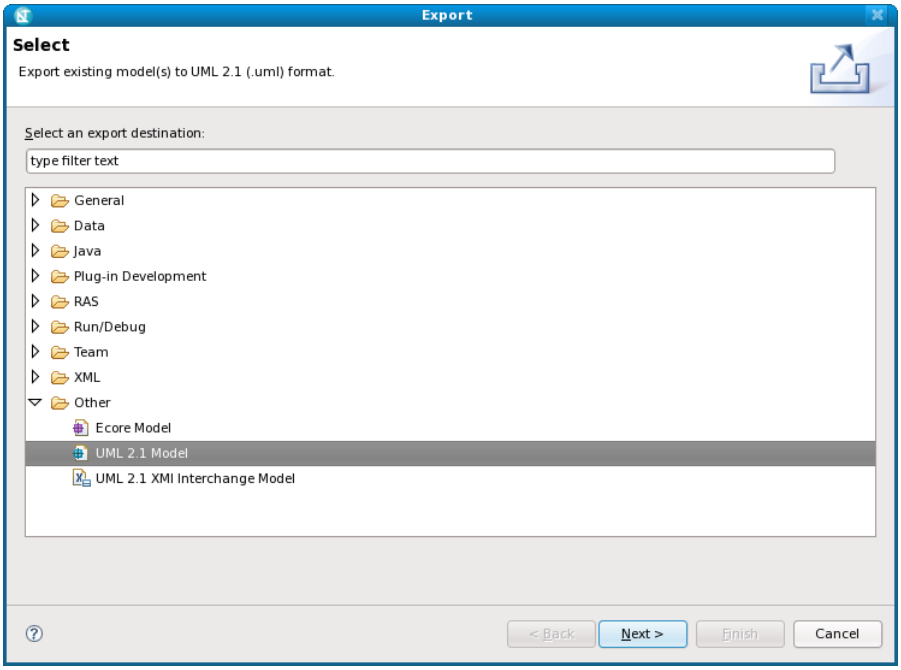

Abbildung 5: Export Wizard (1)

zu exportieren und anschließend durch einen Konverter nach EMF zu transformieren. Beide Schritte werden im Folgenden beschrieben.

Modellexport Der Modellexport untergliedert sich in die folgenden Schritte:

Schritt 1: Rechtsklick auf das RSM-Projekt TestModels > Export.

Schritt 2: Auswahl der Kategorie Other > UML 2.1 Model (s. Abb. 2.2.2).

Schritt 3: Auswahl des zu exportierenden Modells und des Zielverzeichnisses, für welches das Verzeichnis `rsm/export` im Bundle `org.sidiff.common.testmodels` zu wählen ist. **Wichtig:** Unbedingt die Option Export applied profiles aktivieren! (s. Abb. 2.2.2).

Generierung von EMF-Modellen Um EMF-Modelle zu generieren, muss eine OSGI-Konsole mit dem Bundle org.sidiff.common.testmodels gestartet werden. Dieses nimmt ein Kommando generate $< WORKSPACE{-}URI >$ entgegen, wobei $< WORKSPACE{-}URI >$ den Betriebssystem-absoluten Pfad zu dem für die SiDiff-Entwicklung genutzten Eclipse-Workspace darstellt. Um die ständige Eingabe dieses Parameters zu

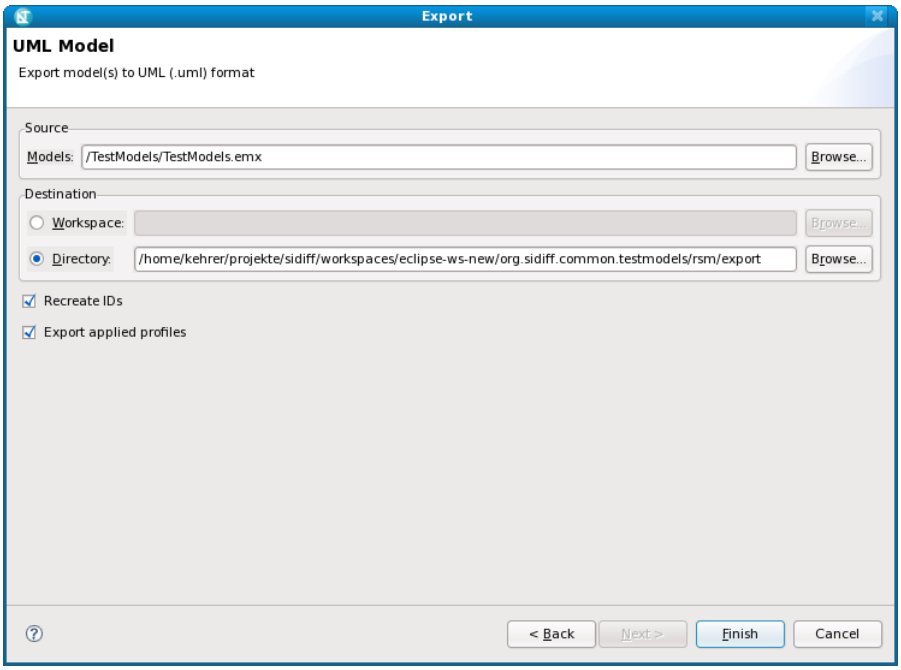

Abbildung 6: Export Wizard (2)

vermeiden, kann in der Klasse `TestdataGeneratorCommandProvider` im Bundle und gleichnamigen Paket `org.sidiff.common.testmodels` ein neues Kommando spezifiziert werden, welches die private Methode generate(String workspaceUri) aufruft. Beispiel:

```
public void _generateTK(CommandInterpreter commandInterpreter) throws Exception {
  generate("/home/kehrer/projekte/sidiff/workspaces/eclipse-ws-new");
}
```

Die generierten EMF-Modelle werden in den jeweiligen Bundles zum Testen der Compare-Funktionen abgelegt.

3 Einrichten und Ausführen der Unit-Tests

Checkout der relevanten Bundles. Für die Unit-Tests der EMF Funktionen sind dies beispielsweise

- org.sidiff.common
- org.sidiff.common.emf
- org.sidiff.common.junit
- org.sidiff.common.services
- org.sidiff.common.testmetamodel
- org.sidiff.core.compare.comparefunctions
- org.sidiff.core.compare.comparefunctions.emf
- org.sidiff.core.compare.comparefunctions.emf.test

Starten des Activators in org.sidiff.common.junit als neue OSGi Framework Run-Konfiguration.

Reiter Bundles: Auswahl der benötigten bundles (i.d.R. Workspace + required Bundles)

Reiter Arguments: Wichtig! Das Working Directory auf den lokalen Workspace setzen. Dies ist nötig damit die zu testenden Models später über relative Pfade gefunden werden können!

Einstellungen übernehmen und starten. In der OSGi Konsole sampleTest zum Starten des Beispiels tippen und Return drücken...

4 Parameters and Regular Expressions

4.1 Parameter Values

- ; : parameter-seperator
- - : indicates a negative regular expression
- + : indicates a positve regular expression
- cs : case-sensitive
- ci : case-insensitive
- auto : viewing order depending on the metamodel
- io: ignore order

4.2 Using RegularExpressions to specifiy Attributes

If the parameter starts with + or -, it indicates that the usage of regular expressions. The first character indicates the expected result of the regular expression which starts from the second character.

Examples:

- To compare the name attribute the parameter is name
- To compare all attributes the parameter is $+.*$
- To compare the attributes name and visibility use $+name|visibility$
- To compare all attributes except name and visibility use $+.*[^(name|visibility)].*$ or $-name|visibility$
- To compare all attributes starting with a, b or c use $+[abc].*$

5 Testaequivalenzklassen

5.1 General Compare Issues

Allgemeine Test-Aequivalenzklassen (Typ-unabhängig):

1. Es werden keine Attribute mit dem gegebenen Namen gefunden
2. Es wird genau ein Attribut mit dem gegebenen Namen gefunden
3. Es werden mehrere durch einen positiven regulaeren Ausdruck spezifizierte Namen gefunden
4. Es werden mehrere durch einen negativen regulaeren Ausdruck spezifizierte Namen gefunden

5.2 Comparing String Values

String Values: eString

Test-Aequivalenzklassen:

1. Zeichenketten ohne gemeinsame Zeichen [aBBa-cddc]
2. Zeichenketten mit ausschliesslich case-sensitiv gemeinsamen Zeichen: [aBBa-aBcBac]
3. Zeichenketten mit ausschliesslich case-insensitiv gemeinsamen Zeichen: [aBBa-AbcbAc]
4. Zeichenketten mit sowohl case-sensitiven, als auch case-insensitiv gemeinsamen Zeichen [aBBa-abcbac]
5. identische Strings (CI) [aBBa-AbbA]
6. identische Strings (CS) [aBBa-aBBa]
7. NULL-Werte

5.3 Comparing Numerical Values

Numerical Values: eBigDecimal, eBigInteger, eByte, eByteObject, eDouble, eDoubleObject, eFloat, eFloatObject, eInt, eIntegerObject, eLong, eLongObject, eShort, eShortObject

Test-Aequivalenzklassen:

1. ungleiche numerische werte [1-2]
2. identische numerische werte [1-1]
3. NULL-Werte

6 Compare Functions

6.1 EqualID

Precondition

Requires that the ID attribute exists.

Semantics

Compares two nodes based on their ID.

Return Value

In case the IDs are equal the similarity is 1, otherwise it is 0.

Parameters

-

Exceptions

Tests

FiXme Note: *Was passiert, wenn ID-Attribut nicht existiert?*

6.2 ExplicitMatch

Precondition

-

Semantics

This compare function returns a similarity value of 1 for any two given nodes.

Return Value

Always 1

Parameters

-

Exceptions

-

Tests

6.3 MaximumSimilarity

FiXme Note: *Unterschied zu ExplicitMatch?*

Precondition

-

Semantics

This compare function returns a similarity value of 1 for any two given nodes;

Return Value

Always 1

Parameters

-

Exceptions

-

Tests

successful test (sim = 1)
NodeA
NodeB

6.4 NoSimilarity

Precondition

-

Semantics

This compare function returns a similarity value of 0 for any two given nodes;

Return Value

Always 0

Parameters

-

Exceptions

-

Tests

successful test (sim = 0)
  NodeA
  NodeB

6.5 CompareAttributeUsingEquals

Allowed eTypes

numerical Values, eString, eDate, eChar, eCharObject, eBoolean, eBooleanObject, Enumeration, datatype, eByteArray

Precondition

-

Semantics

Compares two nodes based on the string values of their attributes.

Return Value

In case the values of the attributes are equal sim = 1 is returned, else sim = 0.

Parameters

For eStrings: The name of the attributes and the case-sensitive value For other attributes: Only the attribute to be compared by name

Exceptions

Tests

Erste Iteration:
Tests für eString, eInteger, eDouble, eFloat, eBoolean
Identische Tests für eIntegerObject, eDouble, eFloat, eBoolean

Zweite Iteration:
Tests für weitere eTypes

compareAttributeUsingEqualsTest1

Different Integers (Sim = 0)
Model 1: eInteger = 1
Model 2: eInteger = 15

compareAttributeUsingEqualsTest2

Identical Integers (Sim = 1)
Model 1: eInteger = 7

Model 2: eInteger = 7

compareAttributeUsingEqualsTest3

Different Doubles (Sim = 0)
Model 1: eDouble = 1.2d
Model 2: eDouble = 7.3d

compareAttributeUsingEqualsTest4

Same Doubles (Sim = 1)
Model 1: eDouble = 1.6d
Model 2: eDouble = 1.6d

compareAttributeUsingEqualsTest5

Different Floats (Sim = 0)
Model 1: eFloat = 1.2f
Model 2: eFloat = 7.3f

compareAttributeUsingEqualsTest6

Same Floats (Sim = 1)
Model 1: eFloat = 1.6f
Model 2: eFloat = 1.6f

compareAttributeUsingEqualsTest7

Same Strings (case-sensitive) (sim = 1)
case-insensitive test
Model 1: eString = aaBBaa
Model 2: eString = aaBBaa

compareAttributeUsingEqualsTest8

Same Strings (case-sensitive) (sim = 1)
case-sensitive test
Model 1: eString = aaBBaa
Model 2: eString = aaBBaa

compareAttributeUsingEqualsTest9

Same Strings (case-insensitive) (sim = 1)
case-insensitive test
Model 1: eString = AAbbAA
Model 2: eString = aaBBaa

compareAttributeUsingEqualsTest10

Same Strings (case-insensitive) (sim = 0)
case-sensitive test
Model 1: eString = AAbbAA
Model 2: eString = aaBBaa

compareAttributeUsingEqualsTest11

Same Strings (case-insensitive + case-sensitive) (sim = 1)
case-insensitive test
Model 1: eString = AaBbaA
Model 2: eString = aaBBaa

compareAttributeUsingEqualsTest12

Same Strings (case-insensitive + case-sensitive) (sim = 0)
case-sensitive test
Model 1: eString = AaBbaA
Model 2: eString = aaBBaa

compareAttributeUsingEqualsTest13

Same Chars (case-sensitive) (sim = 1)
case-sensitive test
Model 1: eChar = A
Model 2: eChar = A

compareAttributeUsingEqualsTest14

Same Chars (case-sensitive) (sim = 1)
case-insensitive test

Model 1: eChar = A
Model 2: eChar = A

compareAttributeUsingEqualsTest15

Same Chars (case-insensitive) (sim = 0)
case-sensitive test
Model 1: eChar = A
Model 2: eChar = a

compareAttributeUsingEqualsTest14

Same Chars (case-insensitive) (sim = 1)
case-insensitive test
Model 1: eChar = A
Model 2: eChar = 0

compareAttributeUsingEqualsTest16

Different Chars (sim = 0)
case-sensitive test
Model 1: eChar = A
Model 2: eChar = C

compareAttributeUsingEqualsTest17

Different Chars (sim = 0)
case-insensitive test
Model 1: eChar = A
Model 2: eChar = C

compareAttributeUsingEqualsTest18

Different Boolean (sim = 0)
Model 1: eBoolean = true
Model 2: eBoolean = false

compareAttributeUsingEqualsTest19

Same Boolean (sim = 1)
Model 1: eBoolean = true
Model 2: eBoolean = true

6.6 CompareAttributeUsingGauss

Allowed eTypes

numerical Values

Precondition

-

Semantics

Compares a specific attribute of two nodes based on the numeric similarity of their values viewed as double with the help of the gauss-distribution. It needs two additional parameters: the name of the attribute to be compared and a scaling-value.

Return Value

The similarity value is calculated as follows. $exp(-((value1 - value2)^2)/scale))$

Parameters

The attribute to be compared by name The scale (as double) for the gaussian distribution.

Exceptions

Tests

Allgemeine Testaquivalenzklasse

compareAttributeGaussTest1-15

identical values: one test for every numerical eType (sim = 1)
Model1: numValue = identValue
Model2: numValue = identValue

compareAttributeGaussTest16-30

different values: one test for every numerical eType (sim = 0)
Model1: numValue = ValueA
Model2: numValue = ValueB

compareAttributeGauss30-45

test auf null-werte für jeden numerical eType
Model1: numValue = null
Model2: numValue = ValueB

6.7 CompareAttributeUsingLcs

Allowed eTypes

eString

Precondition

-

Semantics

Compares two nodes based on the string values of their attributes.

Return Value

This compare functions the string-values of two attributes based on their Longest Common Subsequence (LCS). The similarity is the ratio of the LCS and the length of the longer attribute.

Parameters

The Attribute to be compared The case-sensitive value

Exceptions

Tests

Allgemeine Testaquivalenzklasse
  compareAttributeUsingLcsTest1

Zeichenketten ohne gemeinsame Zeichen
case-insensitive test (sim = 0)
Model1: eString = aabbaa
Model2: eString = ccddcc

  compareAttributeUsingLcsTest2

Zeichenketten ohne gemeinsame Zeichen
case-sensitive test (sim = 0)
Model1: eString = aabbaa
Model2: eString = ccddcc

  compareAttributeUsingLcsTest3

Zeichenketten mit case-sensitive gemeinsamen Zeichen
case-insensitive test (sim = 0.75)
Model1: eString = aaBBaa
Model2: eString = aacBBcaa

compareAttributeUsingLcsTest4

Zeichenketten mit case-sensitive gemeinsamen Zeichen
case-sensitive test (sim = 0.75)
Model1: eString = aadBBaa
Model2: eString = aacBBaad

compareAttributeUsingLcsTest5

Zeichenketten mit case-insensitive gemeinsamen Zeichen
case-insensitive test (sim = 0.75)
Model1: eString = AAbbAA
Model2: eString = aacBBcaa

compareAttributeUsingLcsTest6

Zeichenketten mit case-insensitive gemeinsamen Zeichen
case-sensitive test (sim = 0)
Model1: eString = AAbbAA
Model2: eString = aacBBcaa

compareAttributeUsingLcsTest7

Zeichenketten mit case-insensitive + case sensitive gemeinsamen Zeichen
case-insensitive test (sim = 0.75)
Model1: eString = aAbbAa
Model2: eString = aacBBcaa

compareAttributeUsingLcsTest8

Zeichenketten mit case-insensitive + case sensitive gemeinsamen Zeichen
case-sensitive test (sim = 0.25)
Model1: eString = aAbbAa
Model2: eString = aacBBcaa

compareAttributeUsingLcsTest9

identische Zeichenketten case-sensitive
case-sensitive test (sim = 1)
Model1: eString = aAABA
Model2: eString = aAABA

compareAttributeUsingLcsTest10

identische Zeichenketten case-sensitive
case-insensitive test (sim = 1)
Model1: eString = aAABA
Model2: eString = aAABA

compareAttributeUsingLcsTest11

identische Zeichenketten case-insensitive
case-insensitive test (sim = 1)
Model1: eString = Aaaba
Model2: eString = aAABA

compareAttributeUsingLcsTest12

identische Zeichenketten case-insensitive
case-insensitive test (sim = 0)
Model1: eString = Aaaba
Model2: eString = aAABA

compareAttributeUsingLcsTest13

null-Werte
case-insensitive test
Model1: eString = null
Model2: eString = aAABA

compareAttributeUsingLcsTest14

null-Werte
case-sensitive test

Model1: eString = null
Model2: eString = aAABA

6.8 CompareAttributeUsingIndexOf

Allowed eTypes

eString

Precondition

-

Semantics

Compares two nodes based on the string values of their attributes.

Return Value

This compare functions test if the strings of the attribute are contained by one another. E.g when the attribute value of Node B is contained in the attribute value of Node A the similarity is the ratio of the shorter attribute value of Node B and the attribute value of Node A.

Parameters

The Attribute to be compared The case-sensitive value

Exceptions

Tests

Allgemeine Testaquivalenzklasse

compareAttributeUsingIndexOfTest1

Different Strings
case-insensitive test (sim = 0)
Model1: eString = aabbaa
Model2: eString = ccddcc

compareAttributeUsingIndexOfTest2

Different Strings
case-sensitive test (sim = 0)
Model1: eString = aabbaa
Model2: eString = ccddcc

compareAttributeUsingIndexOfTest3

Zeichenketten mit case-sensitive gemeinsame Zeichen
case-sensitive test (sim = 0.75)
Model1: eString = AABBAA
Model2: eString = acAABBAA

compareAttributeUsingIndexOfTest4

Zeichenketten mit case-sensitive gemeinsame Zeichen
case-insensitive test (sim = 0.75)
Model1: eString = AABBAA
Model2: eString = acAABBAA

compareAttributeUsingIndexOfTest5

Zeichenketten mit case-insensitive gemeinsame Zeichen
case-sensitive test (sim = 0)
Model1: eString = aabbaa
Model2: eString = acAABBAA

compareAttributeUsingIndexOfTest6

Zeichenketten mit case-sensitive gemeinsame Zeichen
case-insensitive test (sim = 0.75)
Model1: eString = aabbaa
Model2: eString = acAABBAA

compareAttributeUsingIndexOfTest7

Zeichenketten mit case-sensitiven und case insensitiven gemeinsamen Zeichen
case-sensitive test (sim = 0)
Model1: eString = aabBaA
Model2: eString = acaABBAa

compareAttributeUsingIndexOfTest8

Zeichenketten mit case-sensitiven und case insensitiven gemeinsamen Zeichen
case-insensitive test (sim = 0.75)
Model1: eString = aabBaA

Model2: eString = acaABBAa

compareAttributeUsingIndexOfTest9

Identische Strings case-sensitive
case-sensitive test (sim = 1)
Model1: eString = aaBBaa
Model2: eString = aaBBaa

compareAttributeUsingIndexOfTest10

Identische Strings case-sensitive
case-insensitive test (sim = 1)
Model1: eString = aaBBaa
Model2: eString = aaBBaa

compareAttributeUsingIndexOfTest11

Identische Strings case-insensitive
case-sensitive test (sim = 0)
Model1: eString = aaBBaa
Model2: eString = aABBAa

compareAttributeUsingIndexOfTest12

Identische Strings case-insensitive
case-sensitive test (sim = 1)
Model1: eString = aaBBaa
Model2: eString = aABBAa

compareAttributeUsingIndexOfTest14

null-werte
case-sensitive test
Model1: eString = null
Model2: eString = aaBBaa

compareAttributeUsingIndexOfTest15

null-werte
case-insensitive test
Model1: eString = null
Model2: eString = aaBBaa