Dresden OCL Toolkit

Integrationsschnittstelle

bearbeitet von:

Mirko Stölzel

s2729561@mail.inf.tu-dresden.de

Technische Universität Dresden

Fakultät Informatik

Institut für Software- und Multimediatechnik

Lehrstuhl für Softwaretechnologie

1. Einleitung

Das Dresden OCL Toolkit [WWW1] wird seit 1999 am Lehrstuhl für Softwaretechnolgie der TU Dresden als Ergebnis der Arbeit von mehreren Beleg-, Diplomstudenten und wissenschaftlichen Mitarbeitern entwickelt. Derzeit wird an der Veröffentlichung der Version 2.0 gearbeitet, mit der es möglich ist OCL-Ausdrücke einzulesen bzw. als String zu übergeben, diese einer Konsistenzprüfung gegenüber dem zugehörigen Modell zu unterziehen und letztendlich den entsprechenden Java-Code zu erzeugen. Das Dresden OCL Toolkit soll dabei nicht als eigenständige Anwendung, sondern eher als eine Art Bibliothek fungieren.

Das Ziel dieser Arbeit ist es zukünftigen Beleg- und Diplomstudenten den Einstieg im Umgang mit dem Dresden OCL Toolkit zu erleichtern.

Die Integrationsschnittstelle ist als Ergebnis meines Großen Beleges zum Thema „Entwurf und Implementierung der Integration des Dresden OCL Toolkits in Fujaba“ [2] entstanden. Es handelt sich dabei um eine allgemeine Integrationsschnittstelle, die die Integration des Dresden OCL Toolkits in beliebige CASE-Tools ermöglicht.

In Kapitel 2 wird zunächst das Konzept, das hinter der Integrationsschnittstelle steht, vorgestellt. Darauf aufbauend wird dann in Kapitel 3 die Umsetzung dieses Konzeptes erläutert, bevor im abschließenden Kapitel 4 die notwendigen Schritte zur Anwendung der Integrationsschnittstelle aufgeführt werden.

Als Grundlage für diese Arbeit werden die Kenntnisse über die Technologien Metadata Repository und Java Metadata Interface vorausgesetzt, die in dem Dokument „Dresden OCL Toolkit: Metadata Repository und Java Metadata Interface“ [3] näher vorgestellt werden.

2. Vorbetrachtung

2.1 Ansatzpunkte für eine Integration

Bei der Integration des Dresden OCL Toolkits in ein CASE-Tool besteht das Hauptproblem darin, dass die Modellinformationen nicht im MDR des Toolkits vorhanden sind, sondern im Repository des jeweiligen CASE-Tools. Da für die Konsistenzprüfung eines OCL-Constraints durch den Parser des Dresden OCL Toolkits diese Informationen benötigt werden, musste eine Möglichkeit gefunden werden, dem Parser diese Informationen zugänglich zu machen.
Dazu wurde bei der Umsetzung der Integrationsschnittstelle für das Dresden OCL Toolkit eine Art Stellvertreterkonzept aufgegriffen, das schon in der Diplomarbeit von Stefan Ocke [1] erwähnt und anhand des folgenden Beispiels verdeutlicht wird.

2.2 Beispiel - Stellvertreterkonzept

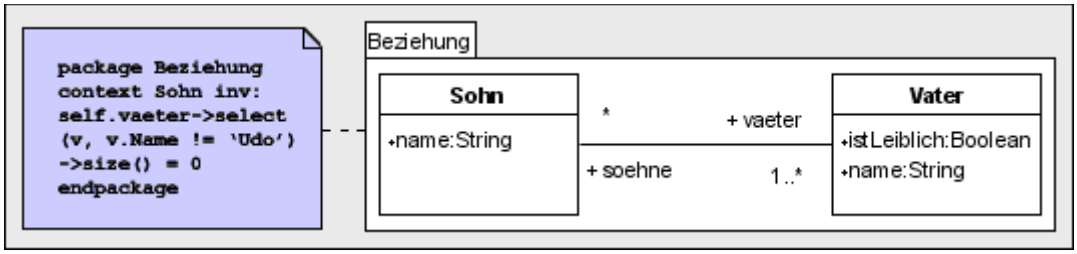

Abbildung 2.1: JMI-Beispiel: Metamodell

Zur Demonstration des Stellvertreterkonzeptes wird die Funktionsweise des Parser anhand der Beziehung zwischen Vätern und deren Söhnen näher betrachtet. In Abbildung 2.1 ist dazu das entsprechende Klassendiagramm dargestellt. Darin ist zu erkennen, dass jeder Sohn mehrere Väter haben kann, da Waisenkinder nach dem Tod ihrer Väter durch den neuen Mann ihrer Mutter adoptiert werden können. Dabei bleibt jedoch der Umstand bestehen, dass Söhne genau einen leiblichen Vater haben. Diese Eigenschaft wird durch das Attribut `istLeiblich` ausgedrückt. Damit zusätzlich dargestellt werden kann, dass jeder Sohn genau einen leiblichen Vater hat, muss ein entsprechender OCL-Ausdruck definiert und der Klasse `Sohn` zugewiesen werden
Bei der Konsistenzprüfung dieses Constraints durch den Parser wird dieser zunächst einer Syntaxüberprüfung entsprechend der konkreten Syntax der OCL-Spezifikation unterzogen. Sollten dabei keinerlei Fehler auftreten, wird anschließend der OCL-Ausdruck von konkreter in abstrakte Syntax, also in eine Instanz des UMLOCL Metamodells des Dresden OCL Toolkits, überführt. Da der OCL-Ausdruck verschiedene Modellelemente des zugrunde

liegenden UML-Modells referenziert (Paket `Beziehung` …), muss dabei überprüft werden, ob diese Modellelemente auch existieren. Dazu greift der Parser über die zusätzlichen Methoden des CommonOCL auf das zugrunde liegende Modell zu.
Beim Stellvertreterkonzept läuft dieser Vorgang völlig identisch ab. Der Hauptunterschied besteht darin, dass beim Zugriff auf das MDR die Modellinformationen nicht vorhanden sind, sondern indirekt durch Stellvertreter übertragen werden. Zu Beginn der Konsistenzprüfung befindet sich im MDR als Repräsentant unseres Modells lediglich eine Instanz des `RefOutermostPackage` des UML-Metamodells, und das TopPackage als eine Instanz der UML-Metaklasse `Model`. Bei dieser Instanz handelt es sich um einen Stellvertreter, der das zugehörige Modellelement auf CASE-Tool-Seite referenziert. In Abbildung 2.2 sind zur Verdeutlichung dieses Umstandes das MDR und das CASE-Tool-Repository abgebildet. Die Beziehung zwischen TopPackage-Stellvertreter und dem zugehörigen Modellelement auf CASE-Tool-Seite wird über die Assoziation dargestellt.

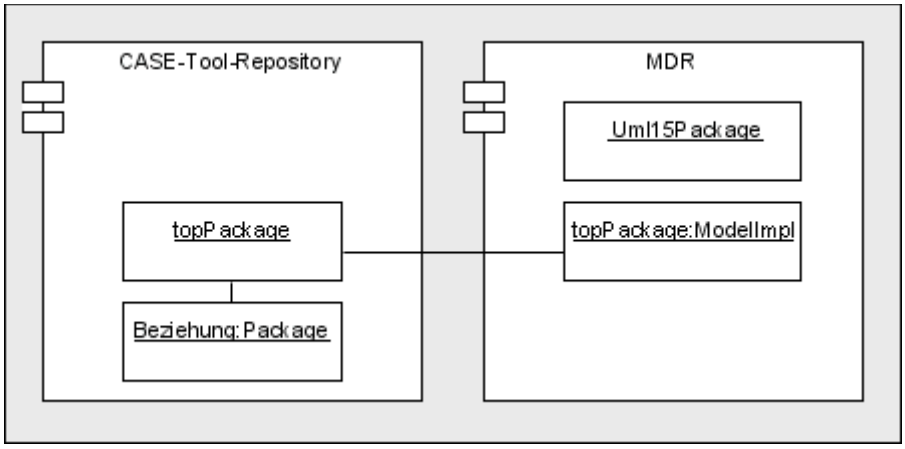

Abbildung 2.2: Das initiale UML-Modell im MDR

Bei der Konsistenzprüfung unseres OCL-Ausdrucks durch den Parser wird zunächst die Paketkontextdefinition des OCL-Ausdrucks überprüft. Aus diesem Grund wird zunächst versucht das Paket `Beziehung` zu finden. Zum Einstieg in das Modell wird das TopPackage genutzt, dessen Methode `findPackage()` aufgerufen wird. Dabei handelt es sich um eine Methode des CommonOCL, die auf Ebene des UMLOCL in der zugehörigen benutzerdefinierten JMI-Schnittstelle `PackageImpl` implementiert ist. In dieser Methode wird zum Auffinden des gesuchten Paketes die Methoden von UMLOCL genutzt, um innerhalb des UML-Modells zu navigieren. Somit würde die Methode `getOwnedElements`() aufgerufen werden, um sämtliche Elemente innerhalb des MDR zu erhalten, die sich im TopPackage befinden. Da jedoch das Dresden OCL Toolkit für unser Beispiel in ein CASE-Tool integriert ist, muss im Repository des CASE-Tools nach einem

Paket `Beziehung` gesucht werden. Dies geschieht über die Referenz des TopPackage-Stellvertreters auf das zugehörige Modellelement auf Seite des CASE-Tools. Im MDR des Dresden OCL Toolkits wird daraufhin ein entsprechender Stellvertreter erzeugt, der das gefundene Paket auf CASE-Tool-Seite referenziert. Das Ergebnis dieses Vorgangs ist in Abbildung 2.3 zu erkennen.

In Folge weiterer Typüberprüfungen wird mit der Klasse `Vater` und dem Attribut `istLeiblich` sowie mit der Klasse `Sohn` und dessen Assoziationsende `vaeter` analog verfahren. Es werden dabei lediglich andere Methoden des CommonOCL verwendet.

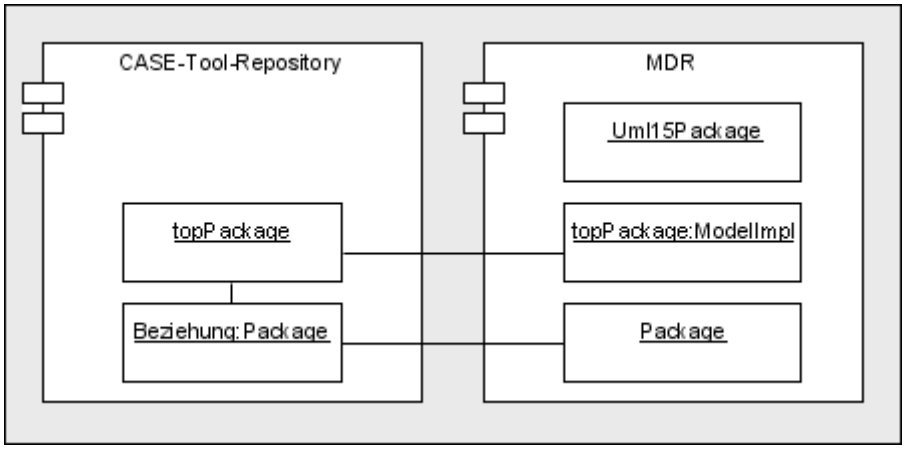

Abbildung 2.3: Das MDR nach Suche des Paketes „Beziehung“

3. Architektur der Integrationsschnittstelle

3.1 Anforderungen an die Integrationsschnittstelle

Anhand des Beispiels zum Stellvertreterkonzept im vorangegangenen Abschnitt haben wir sehen können, dass für die Umsetzung einer Integrationsschnittstelle verschiedene Anforderungen bestehen:

1. Bei der Durchführung einer Konsistenzprüfung durch den Parser muss im MDR des Dresden OCL Toolkits ein UML-Modell existieren, dessen TopPackage ein Stellvertreter zu einem zugehörigen Modellelement auf CASE-Tool-Seite ist.
2. Die Stellvertreterobjekte müssen die entsprechenden Modellelemente auf CASE-Tool-Seite referenzieren.
3. Beim Zugriff auf Eigenschaften eines Stellvertreterobjekts müssen über die Referenz auf das zugehörige Modellelement auf CASE-Tool-Seite, die entsprechenden Werte ermittelt werden. Dabei müssen existierende Stellvertreter herausgesucht bzw. neue Stellvertreter erzeugt und als Ergebnis zurückgegeben werden.

3.2 Architektur der Integrationsschnittstelle

Für die Umsetzung der aufgezählten Anforderungen existieren die beiden Klassen `OCLChecker` und `ModelFacade`.

Die Klasse `OCLChecker` erfüllt die Anforderung eins. Eine Instanz dieser Klasse dient dazu einen OCL-Ausdruck auf Konsistenz, also die korrekte Syntax und die Existenz sämtlicher referenzierter Modellelement, zu überprüfen. Um eine Instanz dieser Klasse zu erstellen, muss lediglich der Methode `getInstance()` das Modellelement auf CASE-Tool-Seite übergeben werden, das die Rolle des TopPackage einnimmt. Dadurch wird entweder die zu diesem Modellelement bereits existierende Instanz der Klasse `OCLChecker` herausgesucht oder aber eine neue Instanz erstellt. Dabei wird gleichzeitig innerhalb des MDR des Dresden OCL Toolkits das initiale UML-Modell erstellt.

Die abstrakte Klasse `ModelFacade` dient zur Erfüllung der aufgezählten Anforderungen zwei und drei. Sie verwaltet die Referenzen der Stellvertreter auf die zugehörigen Modellelemente auf CASE-Tool-Seite in einer Hashmap und ermittelt in Abhängigkeit von diesen die Eigenschaften der Stellvertreter.

In Abbildung 3.1 ist zur Verdeutlichung der Rolle der Klasse `ModelFacade` der Ablauf der Suche des Parsers nach dem Paket „Beziehung“ abgebildet. Aus dem zu erkennenden Objektdiagramm ist ersichtlich, dass der Parser beim Zugriff auf die Elemente des TopPackage-Stellvertreters auf die entsprechende Methode der Klasse `ModelFacade` umgeleitet wird. Mittels der übergebenen MOFId des TopPackage wird daraufhin das zugehörige Modellelement auf CASE-Tool-Seite herausgesucht, und das in diesem enthaltene Paket „Beziehung“ ermittelt. Das Paket „Beziehung“ bzw. ein Eintrag mit dem dieses Paket auf CASE-Tool-Seite identifiziert werden kann, wird dann an die Methode `getElement()` übergeben. Daraufhin wird entweder der schon existierende Stellvertreter des Paketes „Beziehung“ herausgesucht oder aber bei Nicht-Existenz ein neuer Stellvertreter im MDR des Dresden OCL Toolkit erzeugt. Dieser Stellvertreter wird dann als Ergebnis der `getOwnedElements()`-Methode zurückgegeben.

Damit man das Dresden OCL Toolkits nicht nur in ein CASE-Tool, sondern in beliebige CASE-Tools integrieren kann, sind die Methoden der Klasse `ModelFacade` abstrakt definiert und müssen für die Integration CASE-Tool spezifisch implementiert werden (siehe Kapitel 4.1).

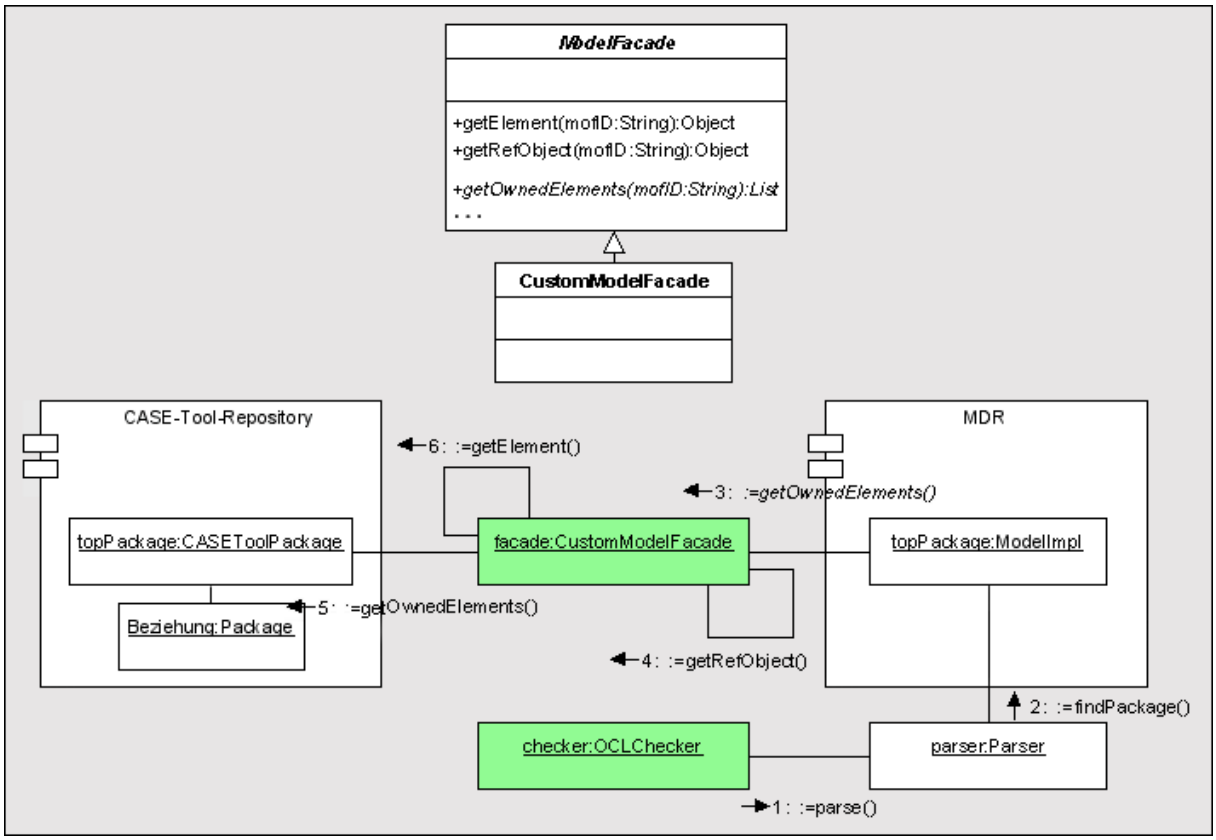

Abbildung 3.1: Suche nach dem Paket „Beziehung“ auf CASE-Tool mittels der Klasse ModelFacade

Die Verbindung zwischen einem UML-Modell im MDR des Dresden OCL Toolkits und der zugehörigen Implementierung der Klasse `ModelFacade` wird über die Methode `setModelFacade()` der Klasse `OCLChecker` hergestellt. Für eine Integration des Dresden OCL Toolkits muss der Methode `setModelFacade()` die Implementierung der Klasse `ModelFacade` übergeben werden, die dann dem zugehörigen initialen UML-Modell zugeordnet wird (siehe Kapitel 4.2). Die Implementierung kann auf diesem Weg auch nach Belieben ausgetauscht werden.

In Abbildung 3.2 ist abschließend die Einbindung der Klasse `ModelFacade` über die benutzerdefinierten Implementierungen der JMI-Schnittstellen anhand der Suche nach einem einer Classifier-Instanz in einem Sequenzdiagramms auszugsweise dargestellt. Darin ist zu erkennen, dass bei der Ermittlung der Features einer `Classifier`-Instanz zunächst überprüft wird, ob das Dresden OCL Toolkit in ein CASE-Tool integriert ist. In diesem Fall würde der Aufruf der Funktion `getInstance()` der Klasse `ModelFacade` die Instanz der zugehörigen Implementierung dieser Klasse als Ergebnis liefern. Sollte dabei keine Instanz gefunden werden, ist das Dresden OCL Toolkit in kein CASE-Tool integriert, und das Attribut wird im MDR des Toolkits über die Supermethode ermittelt. Somit wird sichergestellt, dass das Dresden OCL Toolkit sowohl im integrierten Modus, als auch im „Normalmodus“ arbeiten kann. Zusätzlich wird mittels der `isRepresentative()`-Methode überprüft, ob es sich bei dieser `Classifier`-Instanz um ein Stellvertreterobjekt handelt und ob somit die Features der `Classifier`-Instanz mittels der entsprechenden Methode der Implementierung der Klasse `ModelFacade` ermittelt werden sollen.

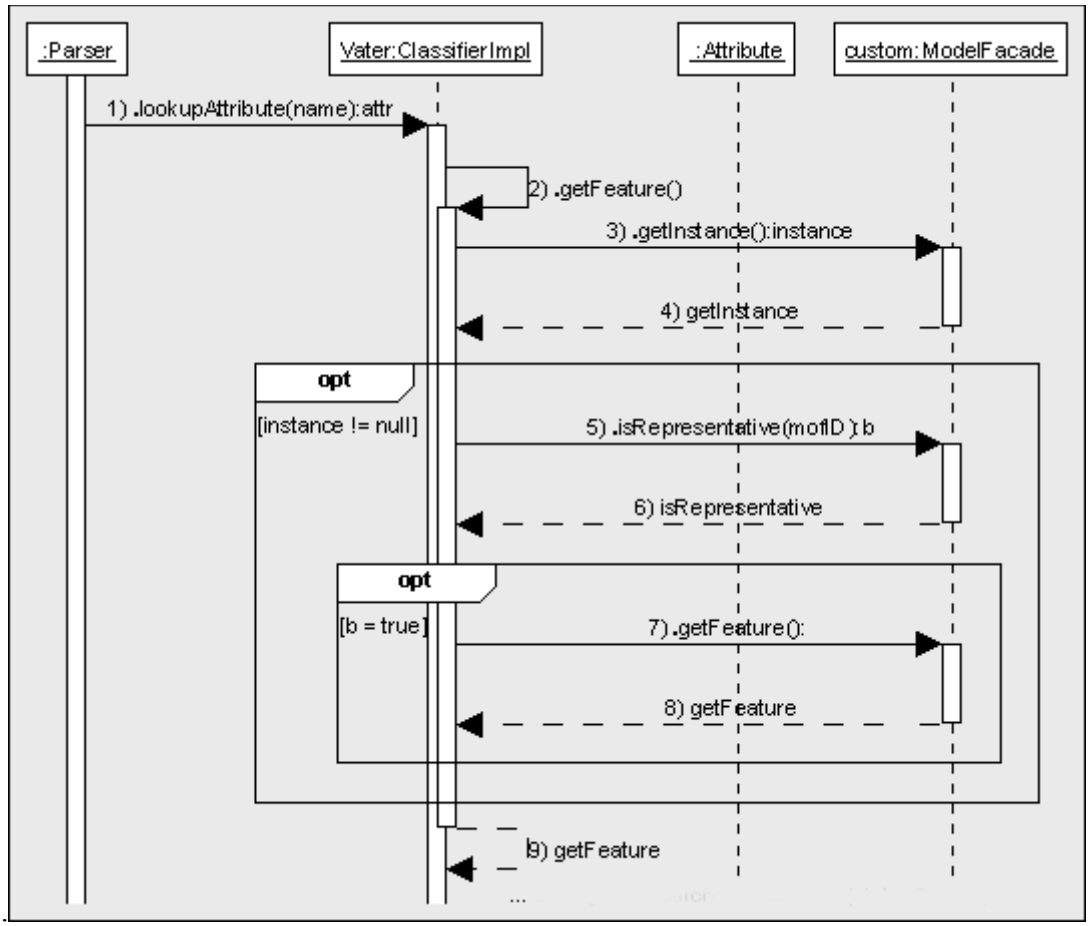

Abbildung 3.2: Suche nach Attribut „istLeiblich“

4. Einsatz der Integrationsschnittstelle

Zur Integration des Dresden OCL Toolkits sind lediglich zwei Schritte erforderlich:

1. CASE-Tool-spezifische Implementierung der abstrakten Klasse `ModelFacade`
2. Durchführung einer Konsistenzprüfung mittels einer Instanz der Klasse `OCLChecker`

In den folgenden zwei Abschnitten werden diese beiden Schritte zur Integration des Dresden OCL Toolkit näher erläutert.

4.1 Implementierung der Klasse ModelFacade

Die CASE-Tool-spezifische Implementierung der Klasse `ModelFacade` erfolgt für jede der abstrakten Methoden nach dem gleichen Prinzip. In Abbildung 4.1 ist zur Verdeutlichung dieses Vorgehens der Pseudocode einer Implementierung der abstrakten Methode `getFeatures()` abgebildet.

1. Zunächst muss das Modellelement auf CASE-Tool-Seite zu dem Stellvertreter herausgesucht werden, für den die jeweilige abstrakte Methode ausgeführt werden soll. Die dafür benötigten Informationen sind innerhalb der HashMap `refObjects` gespeichert und können mittels der bereits implementierten Methode `getRefObject()` der Klasse `ModelFacade` abgefragt werden. Dazu muss dieser lediglich die MOFId des Stellvertreters übergeben werden. Die MOFId des jeweiligen Stellvertreters wird jeder abstrakten Methode der Klasse `ModelFacade` als Parameter übergeben. (Abb. 4.1 Zeile 4)

   Mittels des so ermittelten Modellelements auf CASE-Tool-Seite können nun die Informationen herausgesucht werden, die über die jeweilige abstrakte Methode abgefragt werden sollen (Abb. 4.1 Zeile 6-12). Durch Aufruf der ebenfalls schon implementierten Methode `getElement()`(Abb. 4.1 Zeile 8, 12) können zu diesen Informationen bereits existierende Stellvertreter ermittelt bzw. neue Stellvertreter erzeugt werden. Dazu muss die Klasse des Stellvertreters mittels der Aufzählung `Classes` und das CASE-Tool-Modellelement bzw. ein Eintrag über den dieses identifiziert werden kann als Parameter an die Methode `getElement()` übergeben werden. Die jeweiligen neuen Einträge in der HashMap `refObjects` werden durch diese Methode ebenfalls automatisch erzeugt.

2. Aus diesen ermittelten Stellvertretern bzw. den neu erzeugten Stellvertretern kann dann die Rückgabe der jeweiligen abstrakten Methode generiert werden.
   (Abb. 4.1 Zeilen 8, 12 und 14)

```
 1 public List getFeature(String mofID)
 2 {
 3    ArrayList features = new ArrayList(); //Rückgabe-ArrayList
 4    CASEToolClass clazz = (CASEToolClass)getRefObject(mofID);
 5
 6    Iterator it = clazz.iteratorOfAttr();
 7    while (it.hasNext())
 8      features.add(getElement(Classes.attribute,it.next()));
 9
10    it = clazz.iteratorOfMethods();
11    while (it.hasNext())
12        features.add(getElement(Classes.method,it.next()));
13
14    return features;
15 }
```

Abbildung 4.1: CASE-Tool spezifische Impl. der Klasse ModelFacade (Bsp. getFeature())

4.2 Durchführung einer Konsistenzprüfung

Zur Durchführung einer Konsistenzprüfung sind drei Schritte erforderlich. Zur Verdeutlichung dieser Schritte ist in Abbildung 4.2 der Pseudocode abgebildet.

1. Erzeugung einer Instanz der Klasse `OCL-Checker`. Dazu muss der Methode `getInstance()` das Modellelement auf CASE-Tool-Seite (bzw. ein Eintrag über den dieses identifiziert werden kann) übergeben werden, das die Rolle des TopPackage einnimmt. Dadurch wird die zugehörige Instanz der Klasse `OCLChecker` ermittelt und als Ergebnis der Methode `getInstance()` zurückgegeben. (Abb. 4.2 Zeile 4)
2. Übergabe der CASE-Tool spezifischen Implementierung der abstrakten Klasse `ModelFacade` über die Methode `setModelFacade()`. Über diese Methode kann die CASE-Tool spezische Implementierung nach belieben ausgetauscht werden.
   (Abb. 4.2 Zeile 5, 6)
3. Aufruf der Methode `validate()` zur Konsistenzprüfung eines OCL-Ausdruckes. Dieser wird als Parameter übergeben.
   (Abb. 4.2 Zeile 9)

```
 1 public boolean validate(CASEToolPackage topPackage,
 2                         String constraint)
 3 {
 4   OCLChecker checker = OCLChecker.getInstance(topPackage);
 5   CustomModelFacade custom = new CustomModelFacade();
 6   checker.setModelFacade(custom);
 7   try
 8   {
 9       checker.validate(constraint);
10   }
11   catch (Exception e)
12   {
13       System.out.println("Fehler:" + e.message());
14   }
15 }
```

Abbildung 4.2: Durchführung einer Konsistenzprüfung

5. Zusammenfassung

In dieser Arbeit konnte man lesen, dass beim Entwurf und bei der Implementierung einer Integrationsschnittstelle für das Dresden OCL Toolkit eine Art Stellvertreter Konzept umgesetzt wurde. Dazu existieren die beiden Klassen `OCLChecker` und `ModelFacade`.

Für die Integration des Dresden OCL Toolkits in ein beliebiges CASE-Tool und zur Durchführung einer Konsistenzprüfung sind folgende Schritte erforderlich:

1. CASE-Tool spezifische Implementierung der Klasse `ModelFacade`
   -> siehe Kapitel 4.1
2. Erzeugen einer neuen Instanz der Klasse `OCLChecker` über die Methode `getInstance()`
   -> siehe Kapitel 4.2
3. Zuweisen der CASE-Tool spezifische Implementierung der Klasse `ModelFacade` über die Methode `setModelFacade()`
   -> siehe Kapitel 4.2
4. Aufruf der Methode `validate()` der Klasse `OCLChecker`
   -> siehe Kapitel 4.2

Literaturverzeichnis

[1] Stefan Ocke, *Entwurf und Implementation eines metamodellbasierten OCL-Compilers,* Diplomarbeit, TU-Dresden/Fakultät Informatik, 2003

[2] Mirko Stölzel, *Entwurf und Implementierung der Integration des Dresden OCL Toolkits in Fujaba,* Belegarbeit, TU-Dresden/Fakultät Informatik, 2005

[3] Mirko Stölzel, *Dresden OCL Toolkit: Metadata Repository und Java Metadata Interface,* TU-Dresden/Fakultät Informatik, 2005

[WWW1] TU-Dresden, Institut für Software- und Multimediatechnik, Dresden OCL Toolkit-Homepage, 29.10.2005, <http://dresden-ocl.sourceforge .net>