PHILIPPS-UNIVERSITÄT MARBURG

Erweiterung von Werkzeugen für die Qualitätssicherung von Software-Modellen

Im Wintersemester 2014/2015

Von Dominik Jung
Betreuer: Dr. Thorsten Arendt

Inhalt:

Aufgabenstellung

Das Ziel in diesem Fortgeschrittenenpraktikum war es, das Eclipse-Plugin EMF Refactor um weitere Werkzeuge zur Qualitätssicherung für andere Modellarten zu erweitern.

In diesem Praktikum haben wir als Gruppe Techniken (Metriken, Smells, Refactorings) der OCL und von Zustands- und Aktivitätendiagrammen implementiert.

Meine Aufgabe war die Implementierung von Smells und einem Refactoring für die OCL.

Problemanalyse

EMF Refactor ist eine Erweiterung für Eclipse, welche in diesem Fortgeschrittenen-Praktikum erweitert werden soll. Dabei bietet EMF Refactor verschiedene Funktionen an, um UML-Modelle auf schlechte Struktur zu überprüfen und entsprechend zu bearbeiten, also zu refactorn.

UML-Modelle sind Diagramme, die zur besseren Verständlichkeit eines Projekts beitragen können. Häufig werden UML-Diagramme während der Planungsphase des Projekts erstellt, bevor die eigentliche Implementierungsphase beginnt. Das bekannteste Beispiel für ein UML-Diagramm ist das Klassendiagramm, welches die im Projekt verwendeten Klassen zusammen mit ihren Attributen modelliert und Beziehungen unter den Klassen wie beispielsweise Vererbung darstellt. Es zählt daher zu den Strukturdiagrammen. Allerdings gibt es auch viele andere UML-Modelle wie beispielsweise Aktivitätsdiagramme oder Anwendungsfalldiagramme, die das Verhalten eines Programms und nicht dessen Struktur modellieren.

Refactoring bezeichnet die manuelle oder automatische Verbesserung von bestehendem Code oder von bestehenden Modellen, wobei die eigentlichen Funktionalitäten dieses Codes oder dieser Modelle beibehalten werden. Dabei werden insbesondere die Verständlichkeit und Wartbarkeit der Modelle, die bearbeitet wurden, verbessert. In einigen Fällen kann man jedoch durch Refactoring auch eine Leistungssteigerung erreichen, weshalb sich Refactoring für alle größeren Projekte lohnen kann. Man benutzt Smells und Metriken, um festzustellen, ob ein Projekt durch Refactoring verbessert werden kann.

Smells sind eigentlich richtige Codestellen oder Strukturen in UML-Modellen, die ihre angedachte Funktion wie geplant erfüllen, aber nicht optimal entwickelt wurden und somit möglicherweise für andere Entwickler schwer verständlich oder umständlich zu ändern sind. Ein häufig auftretender Smell ist z.B. redundanter Code, also Programmcode, der an mehreren Stellen in einem Programm vorkommt. Ein Beispiel hierfür wäre Code, der eine Temperaturangabe von Celsius nach Fahrenheit umrechnet und umgekehrt. Dieser Smell kann verbessert werden, indem dieser Code in eine Funktion ausgelagert wird, welche an den entsprechenden Stellen im Programm aufgerufen wird.

Bei UML-Modellen, besonders Klassendiagrammen, gibt es häufig redundante Attribute von einzelnen Klassen wie z.B. das Attribut Geschwindigkeit bei den Klassen "Auto", "LKW" und "Motorrad". Durch diese Redundanz muss jede Klasse einzeln geändert werden, wenn etwas an dem Attribut Geschwindigkeit verändert werden soll. Eine geschicktere Lösung dafür wäre es, eine Oberklasse "Fahrzeug" einzuführen, die das Attribut Geschwindigkeit enthält. Von dieser Oberklasse Fahrzeug erben die Klassen "Auto", "LKW" und "Motorrad" ihre Attribute, wodurch

im Falle einer nötigen Veränderung das Modell nur an einer Stelle, nämlich in der Oberklasse "Fahrzeug", angepasst werden muss.

Metriken, auch Softwaremetriken, sind (mathematische) Funktionen, die Maßzahlen berechnen, mit denen festgestellt werden kann, ob sich ein Refactoring für ein gegebenes Projekt lohnt. Eine sehr bekannte Metrik ist LoC, Lines of Code, die die Gesamtanzahl der Codezeilen in einem Projekt beschreibt. Üblicherweise kann mit 10 - 50 Codezeilen pro Programmierer pro Tag gerechnet werden. Dabei wird unter anderem auch zwischen LoC (alle Zeilen) und SLoC (nur Programmcode ohne Leerzeilen oder Kommentare) unterschieden. Liegt die berechnete Maßzahl für LoC weit über diesem Mittelwert, ist es wahrscheinlich, dass entweder redundanter Code benutzt wurde oder dieser zu ausführlich mit Kommentaren dokumentiert wurde. Falls die berechnete Maßzahl für LoC deutlich unter diesem Mittelwert liegt, wurde der Code wahrscheinlich nicht ausreichend dokumentiert. In beiden Fällen wäre es gut, über ein Refactoring zur Verbesserung der Lesbarkeit des Codes nachzudenken, um den Einstieg in das Projekt für neue Programmierer zu erleichtern.

Metriken können auch für UML-Modelle berechnet werden. Betrachtet man z.B. das eben genannte Fahrzeug-Beispiel kann man die Metrik NATP, also die Anzahl der Attribute der Klassen in einem Paket, berechnen. So kann man einen Überblick erhalten, ob redundante Attribute wahrscheinlich sind. Diese Zahl kann man senken, indem man wie in dem eben genannten Fahrzeug-Beispiel eine Oberklasse erstellt, welche die gemeinsamen Attribute der Klassen enthält und an die entsprechenden Klassen vererbt.

Bei UML-Modellen gibt es aber nicht nur Klassendiagramme zu beachten, für die bereits sehr viele Smells und Metriken in EMF Refactor implementiert sind. Es gibt auch andere wichtige Diagrammarten wie Anwendungsfalldiagramme, Aktivitätsdiagramme oder Zustandsdiagramme, bei denen jeweils eigene Smells und Metriken beachtet oder zumindest anders behandelt werden müssen, als bei einem Klassendiagramm. Diese Diagrammarten sind noch nicht vollständig in EMF Refactor enthalten und müssen daher noch implementiert werden.

Ist eine Diagrammart zusammen mit den jeweiligen Metriken und Smells fertig in EMF Refactor implementiert, können diese Metriken berechnet und Smells gefunden werden, die dann durch automatisches Refactoring verbessert werden können.

Fachkonzept

OCL:

Die Object Constraint Language, kurz OCL, ist eine an Smalltalk angelehnte Sprache, mit der Einschränkungen (Constraints) für UML-Modelle beschrieben werden können. Dabei unterscheidet man verschiedene Arten von Constraints, darunter Invariants, Preconditions, Postconditions und Guards. Dabei müssen Preconditions beim Start einer Operation erfüllt sein, während Postconditions am Ende einer Operation erfüllt sein müssen. Invarianten müssen jederzeit gelten, und Guards müssen am Anfang eines Zustandsüberganges gültig sein. Die OCL wird hauptsächlich bei Klassendiagrammen verwendet, um sinnvolle Einschränkungen für die Attribute der im Diagramm modellierten Klassen zu definieren, wie beispielsweise die Einschränkung, dass das Alter einer Person nicht negativ sein darf. Aufgrund dieser Eigenschaften wird die OCL nicht als eigenständige Sprache, sondern mehr als Ergänzung zu UML verwendet, da die durch die OCL definierten Einschränkungen nicht oder nur schwer in UML realisiert werden können. Für die OCL existieren natürlich auch Smells sowie Refactorings, welche diese verbessern können.

OCL Smells:

Implies Chain: Eine Verkettung von mindestens zwei implies-Operationen. Diese kann durch das dazugehörige Refactoring "Replace Implies Chain by a Single Implication" verbessert werden.

ForAll Chain: Eine Verkettung von mindestens zwei ForAll-Operationen. Diese kann durch das Refactoring "Replace ForAll-Chain by Navigations" verbessert werden.

Duplication: Doppelte und somit redundante Operationen, die häufig durch copy/paste entsteht. Dies können beispielsweise Berechnungen sein, die mehrmals ausgeführt werden. Kann mit "Add Operation Definition" und "Replace Expression by Operation Call" behoben werden.

Magic Literal: Magic Literal beschreibt einen im Code auftauchenden festen Wert, der nicht weiter erklärt wird. Dieser Smell kann durch die Refactorings "Add Variable Definition" und "Replace Expression by Variable" verbessert werden.

Verbose Expression: Dieser Smell besagt, dass ein längerer Ausdruck gefunden wurde, der einfacher und somit kürzer geschrieben werden kann. Refactorings, die diesen Smell beheben, sind "Change Context", "Simplify Operation Calls" und "Change Initial Navigation".

Refactorings:

Replace Implies Chain by a Single Implication: Bei diesem Refactoring wird eine Verkettung von implies-Operationen durch eine einzige, gleichwertige implication ersetzt.

Replace ForAll-Chain by Navigation: Bei diesem Refactoring wird eine Verkettung von ForAll-Operationen durch neue Navigationen ersetzt, bis am Ende nur noch ein ForAll-Operator benutzt wird.

Add Operation Definition: Eine neue Methode wird erstellt. Diese wird häufig aus bestehendem Code extrahiert, um die Redundanz zu verringern.

Replace Expression by Operation Call: Redundante Ausdrücke im Code werden durch einen Methodenaufruf ersetzt, der die gleiche Funktionalität erfüllt. Dieses Refactoring wird häufig zusammen mit "Add Operation Definition" benutzt, um Redundanz zu verringern.

Add Variable Definition: Bei diesem Refactoring werden im Code auftauchende, nicht erklärte Zahlenwerte in Variablen gespeichert.

Replace Expression by Variable: Ein im Code auftauchender, nicht erklärter Zahlenwert wird durch eine Variable mit dem gleichen Wert ersetzt, um die Lesbarkeit und Wartbarkeit des Codes zu verbessern.

Remove/Add Redundant Brackets: Bei diesen Refactorings werden Klammern um Operationen entfernt, falls diese nicht benötigt werden, und hinzugefügt, falls sie die Lesbarkeit verbessern, da sie die Präzedenzen besser darstellen.

Change Context: Bei diesem Refactoring benutzt man einen anderen Context, um einen längeren Ausdruck etwas abkürzen zu können.

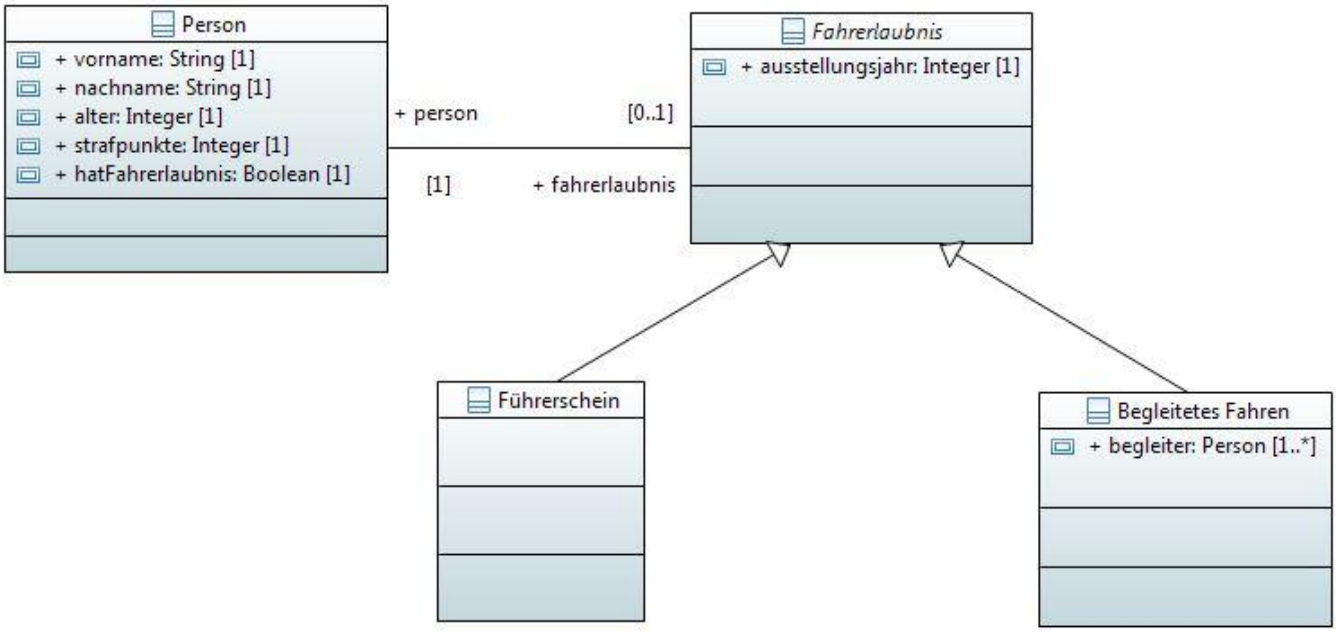
Simplify Operation Calls: Längere Ausdrücke, die aus mehreren, nicht zwingend benötigten Methodenaufrufen bestehen, werden durch weniger Methodenaufrufe kürzer geschrieben.

Change Initial Navigation: Ein längerer Ausdruck wird verkürzt, indem man statt der ursprünglichen Navigation eine andere Navigation verwendet, für die der entsprechende Ausdruck kürzer ist.

Smells und Refactorings an einem Beispiel erklärt:

Beispiel

Zur Demonstration der verschiedenen Smells und Refactorings, die ich ausgewählt habe, habe ich das folgende Beispiel gewählt:



Hierbei handelt es sich um eine vereinfachte Modellierung des Führerscheinsystems, die nur den normalen Führerschein ab 18 Jahren und das "Begleitete Fahren" ab 17 Jahren betrachtet. Durch diese Unterscheidung muss überprüft werden, ob ein Inhaber eines Führerscheins für" Begleitetes Fahren" 17 Jahre alt ist, da ab 18 Jahren der reguläre Führerschein gilt. Außerdem muss überprüft werden, ob die eingetragenen Begleitpersonen die folgenden Bedingungen für die Aufsicht beim "Begleiteten Fahren" erfüllen: Sie müssen mindestens 30 Jahre alt sein, seit mindestens 5 Jahren den Führerschein besitzen und sie dürfen nicht mehr als einen Strafpunkt haben. Ebenfalls überprüft werden muss das Ausstellungsdatum des Führerscheins, da dieses unmöglich in der Zukunft liegen kann, sowie die Anzahl der Strafpunkte, die ein Besitzer einer Fahrerlaubnis angesammelt hat. Sollte die Anzahl der Strafpunkte 8 erreichen oder übersteigen, wird der Führerschein entzogen, weswegen eine Person mit aktivem Führerschein nur weniger als 8 Punkte haben darf. Eine Person hat im Modell einen Vornamen, einen Nachnamen ,ein Alter und die Anzahl der Strafpunkte sowie die Boolean-Variable "hatFahrerlaubnis", die eine Antwort auf die Frage liefert, ob diese Person eine Fahrerlaubnis hat. Eine Fahrerlaubnis ist eine abstrakte Klasse, die ihr Attribut Ausstellungsjahr an die Klassen "Führerschein" und "Begleitetes Fahren" vererbt. Während "Führerschein" keine eigenen Attribute und Funktionen

besitzt, wird "Begleitetes Fahren" um die Begleiter ergänzt. Dabei muss mindestens ein Begleiter eingetragen sein, allerdings gibt es keine Obergrenze für die Anzahl der Begleiter.

Code

```
context Person
inv Fahrer:
  self.hatFahrerlaubnis
    implies (self.alter>=17)
    implies (self.fahrerlaubnis.ausstellungsdatum<=2015)
    implies (self.strafpunkte>=0)
    implies (self.strafpunkte<8)
```

Smells

Folgende Smells habe ich eingebaut:

Magic Literal: Alle Zahlen in der Invariante Fahrer sind "Magic Literals", da sie nicht an eine Variable gebunden sind und somit an jeder Stelle, an der sie auftreten, geändert werden müssten, falls sich der Wert dieser Zahl ändert. Sollte also zum Beispiel die Anzahl der erlaubten Strafpunkte für einen Führerscheinbesitzer von 8 auf 12 erhöht werden, müsste die 8, die vorher zur Überprüfung der Punktzahl benutzt wurde, an jeder Stelle geändert werden.

Implies Chain: Aus dem Wert der Variable "hatFahrerlaubnis" entstehen mehrere Implikationen, die zusammen den Smell "Implies Chain" bilden.

Refactoring

Der "Implies Chain" Smell, der in der Invariante "Fahrer" vorhanden ist, kann in diesem Fall einfach aufgelöst werden, indem sämtliche Implikationen der Implikationen-Kette mit Ausnahme der letzten Implikation durch eine AND-Bedingung ersetzt werden, die die selbe Aussage hat, wie alle ersetzten Implikationen.

Der Code sieht nach diesem Refactoring wie folgt aus:

```
context Person
inv Fahrer:
  (self.hatFahrerlaubnis and
    self.alter>=17 and
    self.strafpunkte>=0 and
    self.strafpunkte<8)
    implies (self.fahrerlaubnis.ausstellungsjahr<=2015)
```

Danach können alle "Magic Literals" der Invariante "Fahrer" durch passende Variablen ersetzt werden, indem für jede Zahl eine Variable erstellt wird und diese Zahl an jeder Stelle, an der sie auftritt, durch die passende Variable ersetzt wird.

Danach sieht der Code beispielsweise so aus:

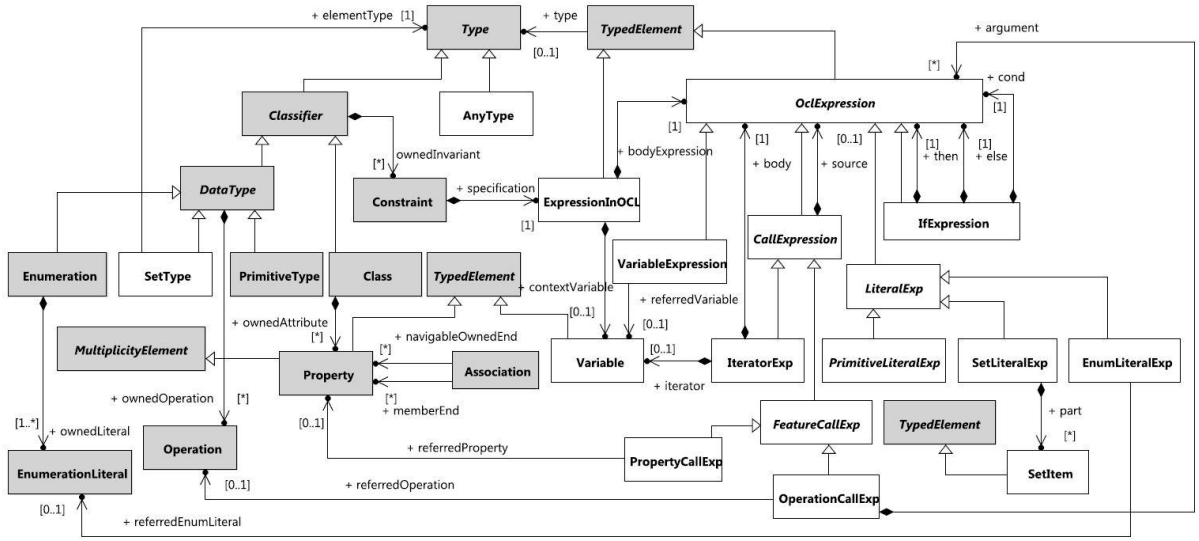
```
def: MindestAlter  : Integer = 17
def: PunkteMinimum : Integer = 0
def: PunkteMaximum : Integer = 8
def: AktuellesJahr : Integer = 2015

context Person

inv Fahrer:
  (self.hatFahrerlaubnis and
    self.alter>=MindestAlter and
    self.strafpunkte>=PunkteMinimum and
    self.strafpunkte<PunkteMaximum)
    implies (self.fahrerlaubnis.ausstellungsjahr<=AktuellesJahr)
```

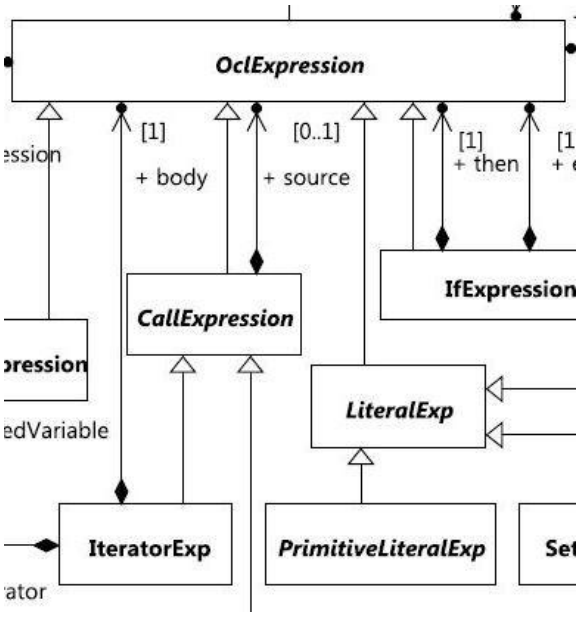
Erläuterung zur Implementierung der OCL Smells

Aufbau der OCL:



Dieses Schaubild zeigt die abstrakte Syntax der OCL. Dabei stammen die grau hinterlegten Elemente aus der UML, die Elemente mit weißem Hintergrund aus der OCL. Die OCL ist eine streng getypte Sprache, was daran erkennbar ist, dass jedes Element der OCL von "Type" oder "TypedElement" (über "OclExpression") erbt.

Magic Literal:



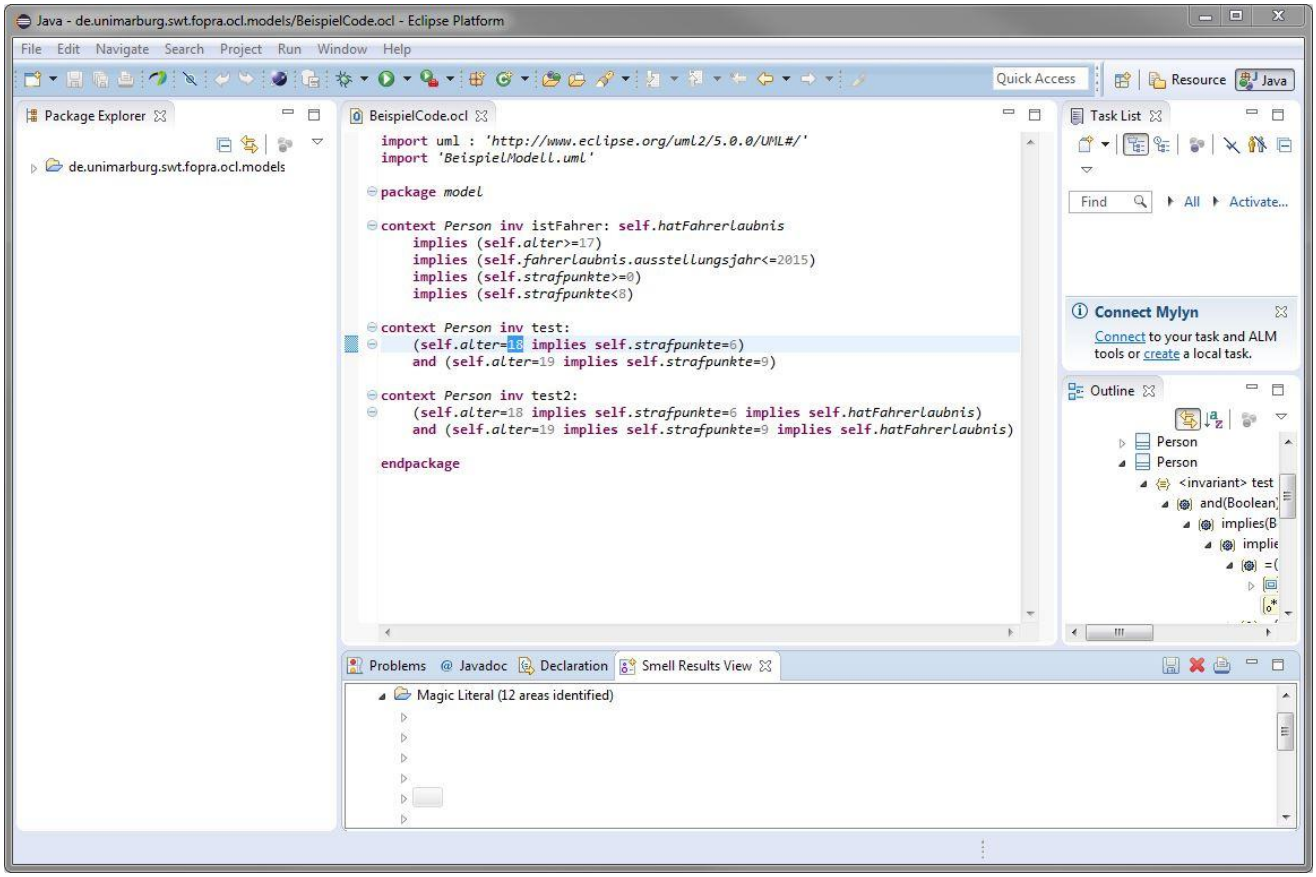
Der "Magic Literal" Smell beschreibt Werte im OCL-Code, die nicht an Variablen gebunden sind. Werte sollten an Variablen gebunden sein, da der entsprechende Wert im Falle einer Änderung an jeder Stelle, an der er verwendet wird, angepasst werden muss. In dem Bildausschnitt zum Aufbau der OCL kann man erkennen, dass dafür nach einer "LiteralExpression", genauer nach einer "PrimitiveLiteralExpression" gesucht werden muss.

Meine Implementierung dieses Smells:

```
1.  Root oclRoot = OCLUtil.getOCLPivotRoot(root);
2.    if (oclRoot != null) {
3.      EList<EObject> allOCLElements = OCLUtil.getAllOCLElementsFromRoot(oclRoot);
4.    for (EObject elem : allOCLElements) {
5.      if (elem instanceof PrimitiveLiteralExp) {
6.        OCLUtil.addModelSmell(results, elem, root);
7.        }
8.      }
9.    }
```

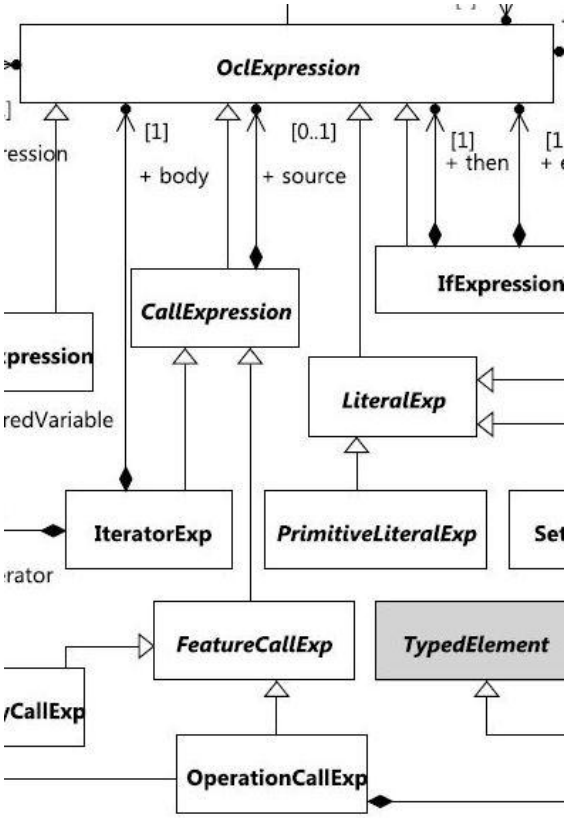
Die Idee dieser Implementierung ist es, der Reihe nach alle Elemente des vorliegenden OCL-Codes zu untersuchen und darin vorkommende, nicht an Variablen gebundene Werte zu finden. Dies wird erreicht, indem eine Liste aller Elemente in diesem Code erstellt wird (Z. 3). Danach wird über alle Elemente dieser Liste iteriert (ab Z. 4). Dabei wird bei jedem Element überprüft, ob dieses vom Typ "PrimitiveLiteralExp " ist (Z. 5), wobei es sich um eine "PrimitiveLiteralExpression" handelt. Dies ist der allgemeine Typ eines Wertes, der an keine Variable gebunden ist. Wird ein Element dieses Typs gefunden, wird es in die Liste "results" eingefügt (Z. 6), welche alle gefundenen Vorkommen des "Magic Literal" Smells beinhalten wird und als Ergebnis der Methode dient.

Beispiel der Anwendung:

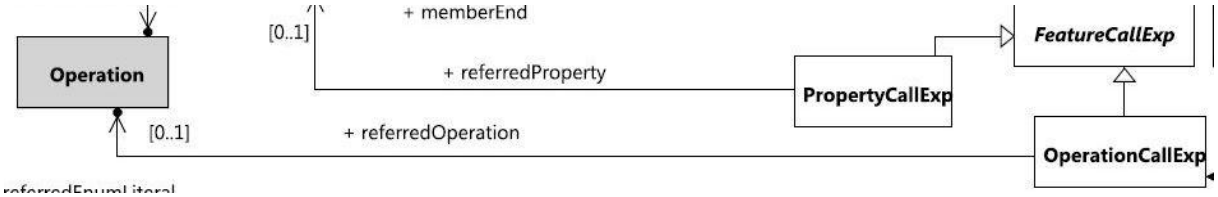


Wie man oben sehen kann, wurden in diesem Beispiel 12 Vorkommen des "Magic Literal" Smells gefunden. Der jeweilige Smell wird im Code markiert, wenn er in der "Smell Results View" ausgewählt wird. In diesem Beispiel ist das der Wert 18, also der fünfte gefundene "Magic Literal" Smell. Die entsprechenden Elemente werden in der "Smell Results View" nicht richtig angezeigt, da es hier zu einem Casting-Fehler (BigInteger zu String) kommt.

Implies Chain:



Der "Implies Chain" Smell beschreibt Verkettungen von mehreren "implies"-Operationen, wobei diese aufeinander folgen müssen und nicht durch andere Operationen wie bspw. "and" getrennt sein dürfen. Wie der Bildausschnitt zeigt, muss dafür nach einer "CallExpression" gesucht werden. Verfolgt man weiter den Aufbau der OCL, kann man erkennen, dass es sich bei dieser "CallExpression" um eine "FeatureCallExpression", genauer eine "OperationCallExpression" handeln muss. Über "referredOperation" erreicht man von dort die entsprechende Operation.



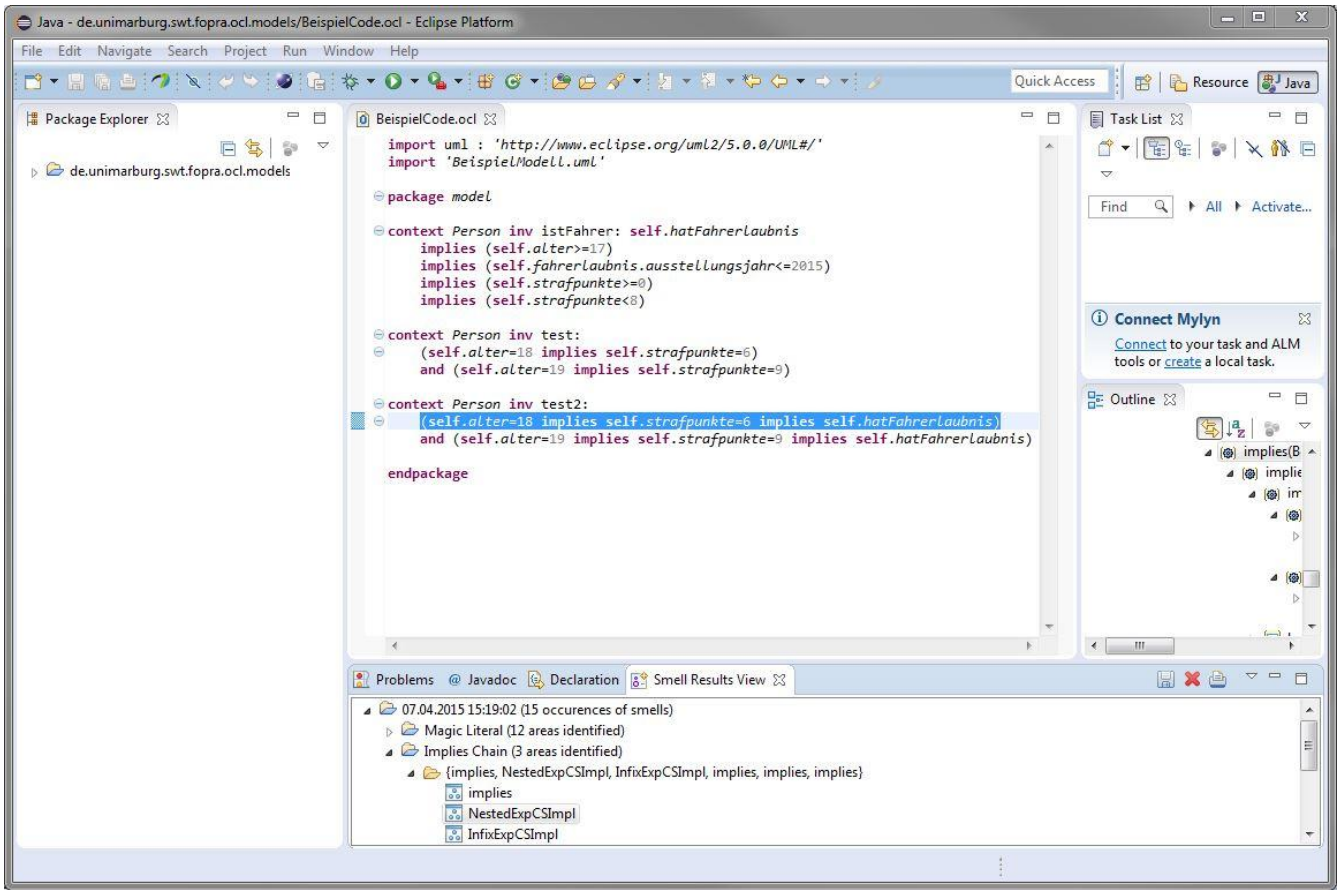
Meine Implementierung dieses Smells:

```
1.  Root oclRoot = OCLUtil.getOCLPivotRoot(root);
2.  if (oclRoot != null) {
3.    EList<EObject> allOCLElements = OCLUtil.getAllOCLElementsFromRoot(oclRoot);
4.    for (EObject elem : allOCLElements) {
5.      if (elem instanceof ExpressionInOCL) {
6.        ExpressionInOCL inv = (ExpressionInOCL) elem;
7.        EList<EObject> usages = new BasicEList<EObject>();
8.        EList<EObject> allInvariantElements =
9.          OCLUtil.getAllOCLElementsFromInvariant(inv);
10.       for (EObject invElem : allInvariantElements) {
11.         if (invElem instanceof OperationCallExp) {
12.           OperationCallExp OpCExp = (OperationCallExp) invElem;
13.           if (OpCExp.getReferredOperation().getName().equals("implies")) {
14.             if (OpCExp.getSource() instanceof OperationCallExp) {
15.               OperationCallExp OpCExpSource = (OperationCallExp) OpCExp.getSource();
16.               if (OpCExpSource.getReferredOperation().getName().equals("implies")) {
17.                 if(!usages.contains(OpCExp)){
18.                   usages.add(OpCExp);
19.                 }
20.                 usages.add(OpCExpSource);
21.               }
22.               else {
23.                 if (usages.size() > 1) {
24.                   OCLUtil.addModelSmell(results, usages, root);
25.                 }
26.                 usages.clear();
27.               }
28.             }
29.             else {
30.               if (usages.size() > 1) {
31.                 OCLUtil.addModelSmell(results, usages, root);
32.               }
33.               usages.clear();
34.             }
35.           }
36.         }
37.       }
38.     }
39.   }
40. }
```

Auch in dieser Implementierung werden der Reihe nach alle Elemente des OCL-Codes nach den richtigen Elementen, hier "implies"-Operationen, durchsucht. Der Unterschied hier ist allerdings, dass es für die richtige Erkennung des "Implies Chain" Smells nicht ausreicht, nur alle Vorkommen von "implies" zu finden. Da "Implies Chain" nur durch die Verkettung mehrerer "implies"-Operationen hintereinander in einer Invariante entsteht, muss jeweils auch das Source-Element der gefundenen Operationen überprüft werden. Dazu wird erst eine Liste mit allen Elementen des Codes erstellt (Z. 3). Danach werden aus dieser Liste neue Listen erstellt, die jeweils alle Elemente einer einzelnen Invariante enthalten (Z.5-8). So wird sichergestellt, dass der Smell wirklich in einer Invariante auftritt und sich nicht über mehrere Invarianten erstreckt. Um jetzt alle "implies"-Operationen in einer Invariante zu finden, wird über alle Elemente der Liste iteriert und überprüft, ob es sich bei dem aktuellen Element um eine Operation, also eine "OperationCallExp" handelt (Z. 10). Ist dies der Fall, wird abgefragt, ob diese Operation "implies" ist (Z. 12-13). Sofern diese Bedingung auch erfüllt ist, wird das

Source-Element dieser Operation auf die gleiche Art überprüft (Z. 14-16). Erst wenn auch dieses Element eine "implies"-Operation ist, ist ein "Implies Chain" Smell gefunden, und beide Elemente werden in die Liste "usages" eingefügt (Z. 17-20). Dabei wird überprüft, ob das erste Element möglicherweise bereits in der Liste enthalten ist (Z. 17), da durch die oben beschriebene Überprüfung des Source-Elements immer zwei Elemente in die Liste eingefügt werden. Ist das Source-Element keine "implies"-Operation, wird der aktuelle Smell als vollständig gefunden angesehen und in die Ergebnisliste "results" eingefügt sowie die Liste "usages" geleert (Z. 22-26). Diese Überprüfung findet danach auch noch für Elemente, die normal über den Iterator aufgerufen werden, statt (Z.29-33). Wenn alle Vorkommen des Smells in einer Invariante gefunden sind, beginnt die Untersuchung der nächsten Invariante. Durch die Untersuchung aller Invarianten nach diesem Prinzip werden alle Vorkommen des "Implies Chain" Smells im OCL-Code gefunden.

Beispiel der Anwendung:



Oben im Bild kann man erkennen, dass im gewählten Beispielcode nur ein "Implies Chain" Smell gefunden wurde. Nur die Invariante "istFahrer" enthält drei gültige "Implies Chain" Smells, da dort die "implies"-Operationen direkt aufeinander folgen. In der Invariante "test" wird demnach kein "Implies Chain" Smell gefunden, da die beiden "implies"-Operationen dort nicht aufeinander folgen, sondern von einer "and"-Operation getrennt sind.
In der Invariante "test2" werden zwei Vorkommen des Smells gefunden, die durch ein "and" getrennt werden. In diesem Beispiel wird der erste Smell in "test2" ausgewählt und damit im Code markiert.

Erläuterung zur Implementierung des Refactorings "Replace Primitive Literal Expression by Variable"

Erklärung des Refactorings:

Bei diesem Refactoring wird ein Wert des Typs "PrimitiveLiteralExpression", der an keine Variable gebunden ist, durch eine Variable mit gleichem Wert ersetzt. Dabei ist es wichtig, dass nicht nur eine Variable mit diesem Wert angelegt wird, sondern dass auch eine Operation, die diesen Wert nutzt, an die neue Variable angepasst wird.

Meine Implementierung dieses Refactorings:

```
135             public void run() {
136                 org.eclipse.ocl.examples.xtext.completeocl.completeoclcs.ClassifierContextDeclCS selectedEObject =
137                     (org.eclipse.ocl.examples.xtext.completeocl.completeoclcs.ClassifierContextDeclCS) dataManagement.
138                             getInPortByName(dataManagement.SELECTEDEOBJECT).getValue();
139                 String variableName =
140                     (String) dataManagement.getInPortByName("variableName").getValue();
141                 Constraint constraint = getConstraint(selectedEObject);
142                 PivotFactory factory = PivotFactory.eINSTANCE;
143                 LetExp letExp = factory.createLetExp();
144                 EList<EObject> allElements = getAllElements(constraint);
145                 Variable var = factory.createVariable();
146                 var.setName(variableName);
147                 VariableExp varExp = factory.createVariableExp();
148                 varExp.setReferredVariable(var);
149                 ExpressionInOCL ExpInOCL = getExpressionInOCL(allElements);
150                 OCLExpression BodyExp = ExpInOCL.getBodyExpression();
151                 ExpInOCL.setBodyExpression(letExp);
152                 letExp.setIn(BodyExp);
153                 PrimitiveLiteralExp PLExp = getPrimitiveLiteralExp(allElements);
154                 OperationCallExp OpCExp = getOperationCallExp(allElements, PLExp);
155                 if(OpCExp.getSource() instanceof PrimitiveLiteralExp){
156                     PrimitiveLiteralExp PLExp2 = (PrimitiveLiteralExp) OpCExp.getSource();
157                     if(PLExp2.equals(PLExp)){
158                         OpCExp.setSource(varExp);
159                     }
160                     else{
161                         List<OCLExpression> arguments = OpCExp.getArgument();
162                         arguments = replaceArgument(arguments, PLExp, varExp);
163                     }
164                 }
165                 else{
166                     List<OCLExpression> arguments = OpCExp.getArgument();
167                     arguments = replaceArgument(arguments, PLExp, varExp);
168                 }
169                 var.setInitExpression(PLExp);
170                 letExp.setVariable(var);
171                 System.out.println(constraint.toString());
172             }
```

Zuerst wird eine Liste aller Elemente der ausgewählten "Constraint" erstellt (Z. 136-141, 144). Danach wird eine "Factory" erstellt (Z. 142), mit deren Hilfe die neuen OCL-Elemente "letExp", "var" (Variable) und "varExp" (VariableExp) generiert werden können (Z. 143, 145, 147). Über diese neuen Elemente wird eine neue Variable in die "Constraint" eingebunden. Dies wird erreicht, indem den neuen Objekten die jeweils benötigten Werte zugewiesen werden. Die Variable erhält so einen vorher vom Nutzer gewählten Namen (Z. 136-140, 146), und "varExp"

erhält "var" als "referredVariable" (Z. 148). Danach wird über den Aufruf der Hilfsmethode "getExpressionInOCL(allElements)" das Element des Typs "ExpressionInOCL" gesucht (Z. 149), welches die für das Refactoring wichtige "BodyExpression" enthält. Anschließend wird die "BodyExpression" der "ExpressionInOCL" durch die neu generierte "LetExp" ersetzt, welche die alte "BodyExpression" als Wert für "in" übergeben bekommt (Z. 150-152). Danach wird über den Aufruf der Hilfsmethode "getPrimitiveLiteralExp(allElements)" der durch die neue Variable zu ersetzende Wert "PLExp" vom Typ "PrimitiveLiteralExp" gesucht (Z. 153). Im nächsten Schritt wird über den Aufruf der Hilfsmethode "getOperationCallExpression(allElements, PLExp)" die zugehörige Operation gesucht, die "PLExp" verwendet (Z. 154). Jetzt muss die zuvor verwendete "PrimitiveLiteralExp" durch die neue "varExp" ersetzt werden. Dazu wird überprüft, ob "PLExp" das Source-Element von "OpCExp" ist oder in deren Argumenten enthalten ist (Z. 155-168). Abhängig davon, wo die "PrimitiveLiteralExp" verwendet wurde, wird diese entweder direkt durch "varExp" ersetzt (Z. 155-162)oder mit Hilfe der Hilfsmethode "replaceArgument(arguments, PLExp, varExp)" in der Argumenten-Liste ersetzt(Z. 165-167). Am Ende erhält die Variable "var" den Wert von "PLExp" als "InitExpression" (Z. 169), während "letExp" die Variable "var" über "setVariable(var)" erhält (Z. 170). Damit ist die neue Variable in den OCL-Code integriert und das Refactoring abgeschlossen, was die Textausgabe der "Constraint" zeigt (Z. 171).

Benutzte Hilfsmethoden:

getExpressionInOCL und getPrimitiveLiteralExp:

```
194⊖        private ExpressionInOCL getExpressionInOCL(EList<EObject> list) {
195             for (EObject elem : list) {
196                 if (elem instanceof ExpressionInOCL) {
197                     return (ExpressionInOCL) elem;
198                 }
199             }
200             return null;
201         }

203⊖        private PrimitiveLiteralExp getPrimitiveLiteralExp(EList<EObject> list) {
204             for (EObject elem : list) {
205                 if (elem instanceof PrimitiveLiteralExp) {
206                     return (PrimitiveLiteralExp) elem;
207                 }
208             }
209             return null;
210         }
```

Beide Hilfsmethoden arbeiten nach dem gleichen Prinzip: Der Methode wird eine Liste übergeben, deren Elemente der Reihe nach durchlaufen werden (Z. 195, 204). Das erste Element vom gesuchten Typ wird als Ergebnis geliefert (Z. 196-197, 205-206).

getOperationCallExp:

```
212         private OperationCallExp getOperationCallExp(EList<EObject> list, PrimitiveLiteralExp PLExp) {
213             for (EObject elem : list) {
214                 if (elem instanceof OperationCallExp) {
215                     OperationCallExp OpCExp = (OperationCallExp) elem;
216                     if(OpCExp.getSource() instanceof PrimitiveLiteralExp){
217                         PrimitiveLiteralExp PLExp2 = (PrimitiveLiteralExp) OpCExp.getSource();
218                         if(PLExp2.equals(PLExp)){
219                             return OpCExp;
220                         }
221                     }
222                     List<OCLExpression> arguments = OpCExp.getArgument();
223                     for(OCLExpression arg : arguments){
224                         if(arg instanceof PrimitiveLiteralExp){
225                             PrimitiveLiteralExp PLExp2 = (PrimitiveLiteralExp) arg;
226                             if(PLExp2.equals(PLExp)){
227                                 return OpCExp;
228                             }
229                         }
230                     }
231                 }
232             }
233             return null;
234         }
```

Diese Hilfsmethode bekommt nicht nur eine Liste, sondern auch ein Element "PLExp" des Typs "PrimitiveLiteralExp" übergeben. Die Liste wird, ähnlich wie bei den vorherigen Hilfsmethoden, nach einer "OperationCallExp" durchsucht (Z. 213-214). Wird ein entsprechendes Element gefunden, wird überprüft, ob diese "OperationCallExp" das Element "PLExp" entweder als Source-Element (Z. 216-219) oder als Argument benutzt (Z. 222-227). Die erste "OperationCallExp", die diese Bedingung erfüllt, wird als Ergebnis zurückgegeben (Z. 219, 227).

replaceArgument:

```
236         private List<OCLExpression> replaceArgument(List<OCLExpression> list, PrimitiveLiteralExp PLExp, VariableExp varExp) {
237             int count = 0;
238             for(OCLExpression elem : list){
239                 if(elem instanceof PrimitiveLiteralExp){
240                     PrimitiveLiteralExp PLExp2 = (PrimitiveLiteralExp) elem;
241                     if(PLExp2.equals(PLExp)){
242                         list.add(count, varExp);
243                         list.remove(count + 1);
244                     }
245                 }
246                 count++;
247             }
248             return list;
249         }
```

Diese Hilfsmethode bekommt eine Liste, eine "PrimitiveLiteralExp" "PLExp" und eine "VariableExp" "varExp" übergeben. Die Liste wird anfangs nach dem Element "PLExp" durchsucht, wobei über die Variable "count" die Position in der Liste gespeichert wird (Z. 237, 246). Das entsprechende Element wird ermittelt, indem jedes gefundene Element des Typs "PrimitiveLiteralExp" mit "PLExp" verglichen wird (Z. 239-241). Wurde die Position von "PLExp" ermittelt, wird zuerst an dieser Position das Element "varExp" eingefügt (Z. 242). Danach wird das Element an der nächsten Position gelöscht (Z. 243), wobei es sich um das Element "PLExp"

handelt, welches durch die vorherige Operation verschoben wurde. Damit wurde "PLExp" in der Liste durch "varExp" ersetzt, wobei die resultierende Liste als Ergebnis geliefert wird (Z. 248).

Beispiel der Anwendung:

Code vor dem Refactoring:

```
context Person invariant istFahrer:
self.hatFahrerlaubnis.implies(self.alter.>=(17)).implies(self.fahrerlaubnis.ausstellu
ngsjahr.<=(2015)).implies(self.strafpunkte.>=(0)).implies(self.strafpunkte.<(8))
```

Code nach dem Refactoring:

```
context Person invariant istFahrer: let test = 17 in
self.hatFahrerlaubnis.implies(self.alter.>=(test)).implies(self.fahrerlaubnis.ausstel
lungsjahr.<=(2015)).implies(self.strafpunkte.>=(0)).implies(self.strafpunkte.<(8))
```

Durch das Refactoring wird die Variable "test" erstellt, wodurch der Ausdruck "let test = 17 in" vor dem ursprünglichen Code eingefügt wird. Im ursprünglichen Code wird der zu ersetzende Wert 17 durch die neue Variable "test" ersetzt.

Getestet wurde das Refactoring manuell an einem OCL-Dokument.

Benutzer-Dokumentation

Um die von mir implementierten Smells und das Refactoring nutzen zu können, folgt man zuerst der Installation von EMF Refactor, wie sie auf https://www.eclipse.org/emf-refactor/install.php beschrieben wird.

Danach müssen die Projekte für Smells und Refactorings als Eclipse-Erweiterung installiert werden.

Um den Menüpunkt zu erreichen, von dem aus man OCL-Smells finden oder das Refactoring durchführen kann, muss in einem OCL-Dokument in Eclipse ein Rechtsklick auf ein OCL-Element (z.B. context) ausgeführt und im daraufhin erscheinenden Dropdown-Menü der entsprechende Punkt ausgewählt werden.

Literaturverzeichnis

„Refactoring object constraint language specifications“; Correa & Werner ; 2007

„Refactoring to improve the understandability of specifications written in object constraint language“; Correa, Werner & Barros; 2009

„Tool Supported OCL Refactoring Catalogue“; Reimann, Wilke, Demuth, Muck & Aßmann; 2012