SBES 2007
XXI Simpósio Brasileiro de Engenharia de Software

Enhancing the Understandability of OCL Specifications

Alexandre Correa[1], Cláudia Werner[2], Márcio Barros[3]

[1,2]COPPE/UFRJ – Universidade Federal do Rio de Janeiro (UFRJ)
Caixa Postal 68.511 – 21945-970 – Rio de Janeiro – RJ – Brazil

[3]UNIRIOTEC - Universidade Federal do Estado do Rio de Janeiro – (UNIRIO)
Av. Pasteur 458 – 22290-240 – Rio de Janeiro – RJ – Brazil

[1,2]{alexcorr, werner}@cos.ufrj.br, [3]marcio.barros@uniriotec.br

Resumo. *OCL (Object Constraint Language) é a linguagem padronizada pelo OMG para a especificação precisa de restrições associadas a modelos e meta-modelos compatíveis com o MOF. Embora a OCL tenha sido criada com a intenção de ser uma linguagem mais simples quando comparada com linguagens formais tradicionais, é comum encontrarmos especificações escritas em OCL contendo expressões difíceis de serem entendidas ou mantidas. Este artigo apresenta um conjunto de construções potencialmente problemáticas, frequentemente encontradas em especificações elaboradas em OCL, além de um conjunto de reestruturações que podem ser empregadas para removê-las. Este artigo descreve, ainda, um estudo experimental que foi realizado para avaliar os efeitos desses conceitos no entendimento de restrições especificadas em OCL.*

Abstract. *OCL (Object Constraint Language) is the OMG standard language for the precise specification of constraints associated to MOF-compliant models and meta-models. Although OCL has been created with the intention to be a simpler language when compared to traditional formal specification languages, it is common to find specifications containing OCL constraints that are difficult to understand and evolve. This paper presents a set of potential problematic constructions often found in specifications written in OCL and a set of refactorings that can be applied to remove them. We also present an experimental study that has been performed to evaluate the effects of applying those strategies on the understandability of OCL specifications.*

1. Introduction

The Object Constraint Language (OCL) is an OMG standard language [OMG 2003a] which supports the specification of elements that can not be graphically described on UML models and MOF-compliant meta-models. OCL can be used for many different purposes, such as: to specify constraint conditions that must hold for the system being modeled; to specify pre and post conditions on operations; to specify guard conditions and derivation rules; to specify well-formedness rules associated to meta-models; as a query language.

In the Model Driven Architecture (MDA) [OMG 2003b], a framework for model-based software development sponsored by the Object Management Group (OMG), OCL is also an important part of model transformation languages such as the

OMG QVT (Query, Views and Transformations) standard [OMG 2005a]. Since it is a precise specification language, OCL can help to remove potential ambiguity problems that might be present in natural language specifications [Berry and Kamsties 2004], as it happened with the first UML standard.

Instead of using formal mathematical notations, OCL is a textual language with syntactical elements similar to those present in object oriented programming languages. OCL was designed to be a language less intimidating than their traditional counterparts [Warmer and Kleppe 2003], such as Z [Woodcock and Davis 1996] and VDM-SL [Jones 1989]. However, this does not prevent specifications written in OCL from being hard to understand and maintain, especially in cases when they contain unnecessarily complex expressions.

In analogy to the term *Code Smells* [Fowler 1999], we defined the term *OCL smells* as constructions present in OCL expressions that might negatively affect the understandability or maintainability of OCL specifications [Correa and Werner 2004]. This paper presents a number of such constructions that are often found in OCL specifications. Refactoring is an important technique for handling software evolution that can be applied to remove *Code Smells* from a software implementation. Refactoring corresponds to changes made to the internal structure of a software, preserving its behavioral semantics, which aims at improving quality factors such as understandability, modularity and extensibility [Mens and Tourwe 2004]. In this paper, we present a set of *OCL Refactorings* that can be utilized to remove *OCL smells* from a specification.

This paper also reports on an experimental study that evaluated the usefulness of OCL refactorings on improving the understandability of OCL expressions. The study involved 23 software developers with industry experience in UML modeling. The results of this study indicate that the presence of OCL smells may have a negative impact in both the correctness and the time necessary to understand constraints written in OCL.

The rest of this paper is structured as follows: section 2 describes the main concepts of OCL. Section 3 presents some OCL smells and the OCL refactorings that can be used to remove them. Section 4 describes the objectives and design of the experimental study. Section 5 reports on the results obtained from the experimental study. Related works are discussed in section 6 and concluding remarks are drawn in section 7.

2. OCL

This section briefly describes the main concepts of OCL that are used in this paper. A more complete description of the language can be found in [Warmer and Kleppe 2003]. OCL is a declarative language that allows the specification of constraints on an underlying model. A model always provides the context for constraints. Figure 1 shows a small UML model used as context for the examples described in this section.

OCL can be used to specify constraints concerning the static structure and the behavior of a system. Invariants are static structure constraints defined in the context of a type. An invariant is defined by a boolean expression that must be true for all instances of that type. The invariant shown in Figure 2 requires all instances of type

Account to have a non negative balance. Since the invariant was defined in the context of the *Account* class, the expression *self.balance >= 0* must be true for all instances of that class. *Self* is a reserved keyword which refers to each instance of the contextual type (*Account*, in this example) used in the evaluation of the invariant.

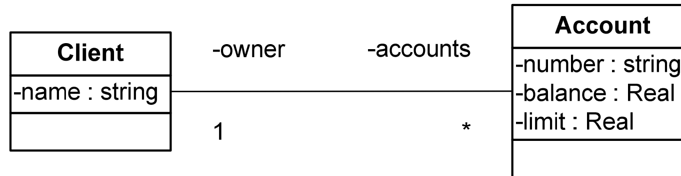


Figure 1. UML model example

```
context Account
        inv: self.balance >= 0
```

Figure 2. Example of a simple OCL invariant

More complex expressions can be built by navigating along the associations defined in the underlying model. Figure 3 shows a navigation expression (*self.accounts*) from a client object to their associated accounts. A navigation results in a collection of elements. In this example, *self.accounts* results in a set of instances of the *Account* class. A number of operations defined in the OCL standard library can be applied to a collection value by using the '->' operator. In this example, the *select* operation results in the subset of the accounts of a given client (*self*) having the *isGold* operation evaluated to true. The *size* operation gives the number of elements in the resulting subset.

```
context Client
def: goldAccountsQty : Integer =
    self.accounts->select(acc | acc.isGold())->size()
def: isGold() : Boolean =
    self.balance > 15000
```

Figure 3. Example of OCL navigation expression

Properties and operations can be added to the model by using the *def* keyword. In the example shown in Figure 3, the attribute *goldAccountsQty* and the operation *isGold* were added to the *Client* type. The value of *goldAccountsQty* is derived from its associated OCL integer expression, and *isGold* is a query operation whose body is defined by the boolean expression *self.balance > 15000.*

OCL is a typed language. Every expression has a type which defines the operations that can be applied to its result. Besides the primitive types (*Boolean*, *Integer*, *Real* and *String)*, all classifiers and enumerations defined on the underlying model are members of the set of available types. The OCL standard library defines several operations for each type. Logical operations (*and*, *or*, *xor*, *implies*, *if-then-else-endif*), collection operations (*size*, *includes*, *isEmpty*) and quantifiers (*forAll* and *exists*) are examples of such operations.

3. OCL Smells and Refactorings

Although OCL has been created with the intention to be a simpler language when compared to traditional formal specification languages, it is common to find specifications containing OCL constraints that are difficult to understand and evolve. Lack of experience in specifying and using OCL [Chiorean et al 2004] combined with poor tool support [Baar et al 2005] are possible explanations for this fact.

The constraints shown in Figure 4 and Figure 5 are well-formedness rules excerpted from the UML 1.3 specification [OMG 1999]. They are examples of overly complex expressions that are often found in OCL specifications. The presence of such complex expressions in the UML specification has lead to misconceptions about the OCL potential. In [Ambler 2002], the constraint shown in Figure 4 is used as an example that even simple rules may result in wordy and hard to read OCL expressions. This constraint, however, could have been defined with an expression as simple as: self.allConnections->isUnique(name).

```
context Association inv AllAssociationEndsHaveDistintctNames:
        self.AssociationEnds->forAll(element1, element2 |
          element1.name(self.AssociationEndNames) =
          element2.name(self.AssociationEndNames)
          implies element1 = element2))
```

Figure 4. UML well-formedness rule – Example 1

Figure 5 shows a constraint related to the Collaboration element defined in the UML metamodel, which refers to the classifier and association roles associated to that element. The operation *oclIsKindOf* returns true if the type of the object to which it is applied is either the same or a descendant of the type passed as argument. The operation *oclAsType* means casting the object to the type specified as argument. Those expressions are usually applied in downcast operations such as in the expression *q.oclAsType(ClassifierRole).base.* Since the type of *q* is *ModelElement* and *base* is defined as a property of *ClassifierRole*, a downcast is needed.

At a first glance, the complexity of the OCL constraint present in Figure 5 may be due to limitations of the specification language, which usually force the usage of several calls to *oclIsKindOf* and *oclAsType* operations [Vaziri and Jackson 2002]. A deeper analysis, however, reveals that the complexity of this constraint actually comes from the usage of inadequate OCL constructions combined with the absence of more general concepts in the underlying model and poor name choices in both the underlying model elements and the OCL expressions. The lack of a supertype for *ClassifierRole* and *AssociationRole* types resulted in complex and very similar expressions separated by the *and* operator.

Therefore, the constraints shown in Figure 4 and Figure 5 are examples of OCL smells. This section describes some OCL smells that are often found in specifications, particularly in those produced by OCL novices. They were collected not only from specifications produced by students and software practitioners, most of them with no previous experience with OCL, but also from the UML official specification [OMG 2005b], and from several papers published in software engineering conferences. Our definition of OCL smells does not include expressions containing errors due to

syntactical or type-checking issues.

```
self.allContents->forAll ( p |
  (p.oclIsKindOf (ClassifierRole) implies
      p.name = '' implies
          self.allContents->forAll ( q |
              q.oclIsKindOf(ClassifierRole) implies
                  (p.oclAsType(ClassifierRole).base =
                     q.oclAsType(ClassifierRole).base implies
                        p = q) ) )
  and
  (p.oclIsKindOf (AssociationRole) implies
      p.name = '' implies
          self.allContents->forAll ( q |
              q.oclIsKindOf(AssociationRole) implies
                  (p.oclAsType(AssociationRole).base =
                     q.oclAsType(AssociationRole).base implies
                        p = q) ) )
)
```

Figure 5. UML well-formedness rule – Example 2

A number of refactorings can be applied to OCL expressions and their underlying model in order to remove OCL smells from a specification. An important activity that precedes refactoring operations is the identification of the parts of a software artifact that should be refactored. In OCL specifications, we use a catalogue of OCL smells that we have elaborated to support the identification of potential targets for refactorings. Due to space constraints, we will only present the OCL smells and refactorings that were used in the experimental study described in section 4, since they correspond to the ones that we have found most often in OCL specifications.

3.1. OCL Smell: Implies Chain

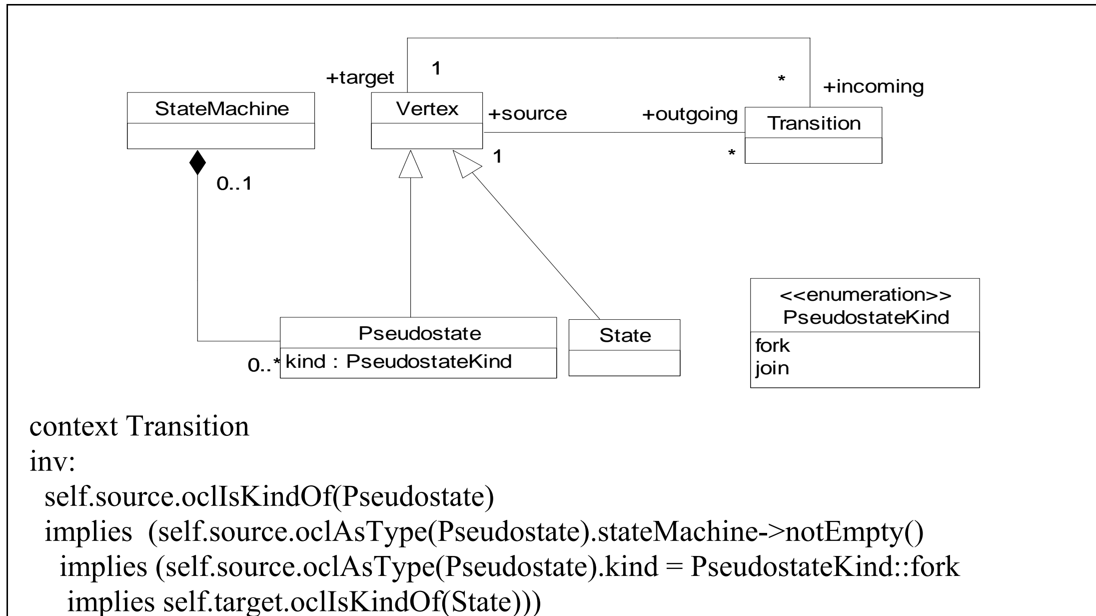
Implies chain corresponds to OCL expressions of the form *b1 implies (b2 implies b3)*, where *b1*, *b2* and *b3* are boolean expressions. Figure 6 shows an example of this smell extracted from the UML specification. This constraint states that the target of every transition from a *fork* pseudostate must be a *State*.

The refactoring *Replace Implies Chain by a Single Implication* replaces an *Implies Chain*, such as <<A implies (B implies C)>>, with an expression of the form <<(A and B) implies C>>. If the antecedent of the refactored expression results in a complex conjunction, the application of further refactorings should also be considered. For example, if the resulting expression is a complex conjunction of the form b_1 *and* b_2 *and* b_3 *... and* b_n, we should further refactor this expression by extracting the definition of auxiliary properties or operations from parts of that conjunction. The definition of auxiliary properties or operations from an OCL expression corresponds to the following refactorings:

- *Add Operation Definition* and *Replace Expression by Operation Call:* The main motivation for these two refactorings is to promote encapsulation and reuse across a specification. By using operation definitions, one can hide complex

expressions from other parts of a specification and avoid the duplication of expressions that are used in many parts of a specification;

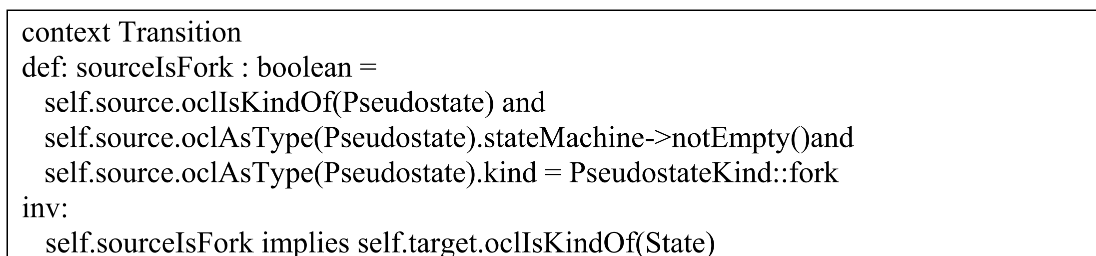
- *Add Property Definition* and *Replace Expression by Property Call* are analogous to the refactorings based on operation definitions previously described. Instead of creating and using OCL helper operations, they create and use OCL helper properties.




```
context Transition
inv:
  self.source.oclIsKindOf(Pseudostate)
  implies  (self.source.oclAsType(Pseudostate).stateMachine->notEmpty()
    implies (self.source.oclAsType(Pseudostate).kind = PseudostateKind::fork
     implies self.target.oclIsKindOf(State)))
```

Figure 6. Example of the *Implies Chain* OCL Smell

Figure 7 shows the refactored version of the constraint present in Figure 6. It is the result of the following refactorings: *Replace Implies Chain by a Single Implication* which replaced two implies by a conjunction, *Add Property Definition* which added the property *sourceIsFork* to the *Transition* class, and *Replace Expression by Property Call* which replaced the antecedent of the constraint by the expression *self.sourceIsFork*.



```
context Transition
def: sourceIsFork : boolean =
   self.source.oclIsKindOf(Pseudostate) and
   self.source.oclAsType(Pseudostate).stateMachine->notEmpty()and
   self.source.oclAsType(Pseudostate).kind = PseudostateKind::fork
inv:
   self.sourceIsFork implies self.target.oclIsKindOf(State)
```

Figure 7. Refactored Version of the *Implies Chain* OCL Smell

3.2. OCL Smell: Verbose Expression

Verbose Expression corresponds to OCL expressions that are bigger than necessary. Besides being easier to understand and maintain, a less verbose expression can usually be evaluated more efficiently. Two usual forms of this smell are:

a) *Expressions containing more operation calls than needed*: these expressions can be shortened by the *Simplify Operation Calls* refactoring. This refactoring consists of rewriting an OCL expression in a shorter form by using less operation calls. A common situation for this refactoring corresponds to expressions that can be simplified by using standard collection operations. Figure 8 illustrates some examples of verbose expressions (on the left column) and their corresponding shorter versions (on the right column). All examples are schematically represented: X denotes a collection and P(x) corresponds to a boolean expression that may use the iterator x on its definition.

<table>
<tr><th>Verbose expression</th><th>Shorter version</th></tr>
<tr><td>X->select(x | P(x))->size() > 0
X->select(x | P(x))->size() >= 1
X->select(x | P(x))->notEmpty()</td><td>X->exists(x | P(x))</td></tr>
<tr><td>X->select(x | P(x))->size() = 1</td><td>X->one(x | P(x))</td></tr>
<tr><td>X->select(x | P(x))->size() = 0
X->select(x | P(x))->isEmpty()</td><td>not X->exists(x | P(x))</td></tr>
<tr><td>X->select(x | P1(x))->select(y | P2(y))</td><td>X->select(x | P1(x) and P2(x))</td></tr>
<tr><td>X->forAll(x1, x2 | x1 <> x2 implies x1.p <> x2.p)</td><td>X->isUnique(p)</td></tr>
</table>

Figure 8. Example of *Verbose Expressions* and their Refactored Versions

b) *Invariants defined in the wrong context class*: since an invariant can be described in many ways depending on its context class, attaching an invariant to the wrong context usually makes it harder to specify and maintain [Warmer and Kleppe 2003]. The invariant shown in Figure 6 is an example of this situation. If *Pseudostate* were used as the context class, the result would be a simpler expression without downcastings, since the condition part of the invariant only references properties defined in this class. As a general guideline, the best context is the one that results in the easiest to read and write expression [Warmer and Kleppe 2003]. Therefore, it is a good exercise to describe the same invariant using different contexts. The refactoring *Change Context* consists of rewriting an invariant by using a different context class. In Figure 9, the invariant shown in Figure 6 was redefined using *Pseudostate* as the context class and two added properties (*isFork* and *allTargets*).

```
context Pseudostate
inv: self.isFork implies
    self.allTargets->forAll(t | t.oclIsKindOf(State))
def: isFork: Boolean = self.stateMachine->notEmpty() and
                  self.kind = PseudostateKind::fork
def: allTargets: Collection(Vertex) = self.outgoing.target
```

Figure 9. Example of the *Change Context* Refactoring

3.3. OCL Smell: Forall Chain

ForAll chain is a special case of the *Verbose Expression* smell that corresponds to expressions containing the following structure: *X ->forAll(a1 | a1.B->forAll(b1 | b1.C ->forAll(c1 | P(c1))))*. Suppose that X is a collection of elements of type A, and there is

an association between classes A and B, and between classes B and C. In such a situation, the expression denotes that some predicate P must be true for all instances of C indirectly associated to the instances of type A present in X. Notice that predicate P does not reference the iterators a1 and b1 defined in the outer *forAll* calls.

Expressions containing *forall chains* can be replaced by a single call to *forAll* operation applied to a navigation from A to C, i.e., X.*B.C ->forAll(c1 | P1(c1))*. Such change corresponds to the application of the *Replace ForAll Chain by Navigations* refactoring.

3.4. OCL Smell: Downcasting

Downcasting is a well-known smell in the object oriented programming community. In OCL, it corresponds to the use of expressions of the form *x.oclAsType(Y).z*, usually preceded by an expression of the form *x.oclIsKindOf(Y).* This is often an indication that some more abstract concepts are missing in the underlying model. The expression associated to the definition of the *sourceIsFork* variable in Figure 7 is an example of this OCL smell.

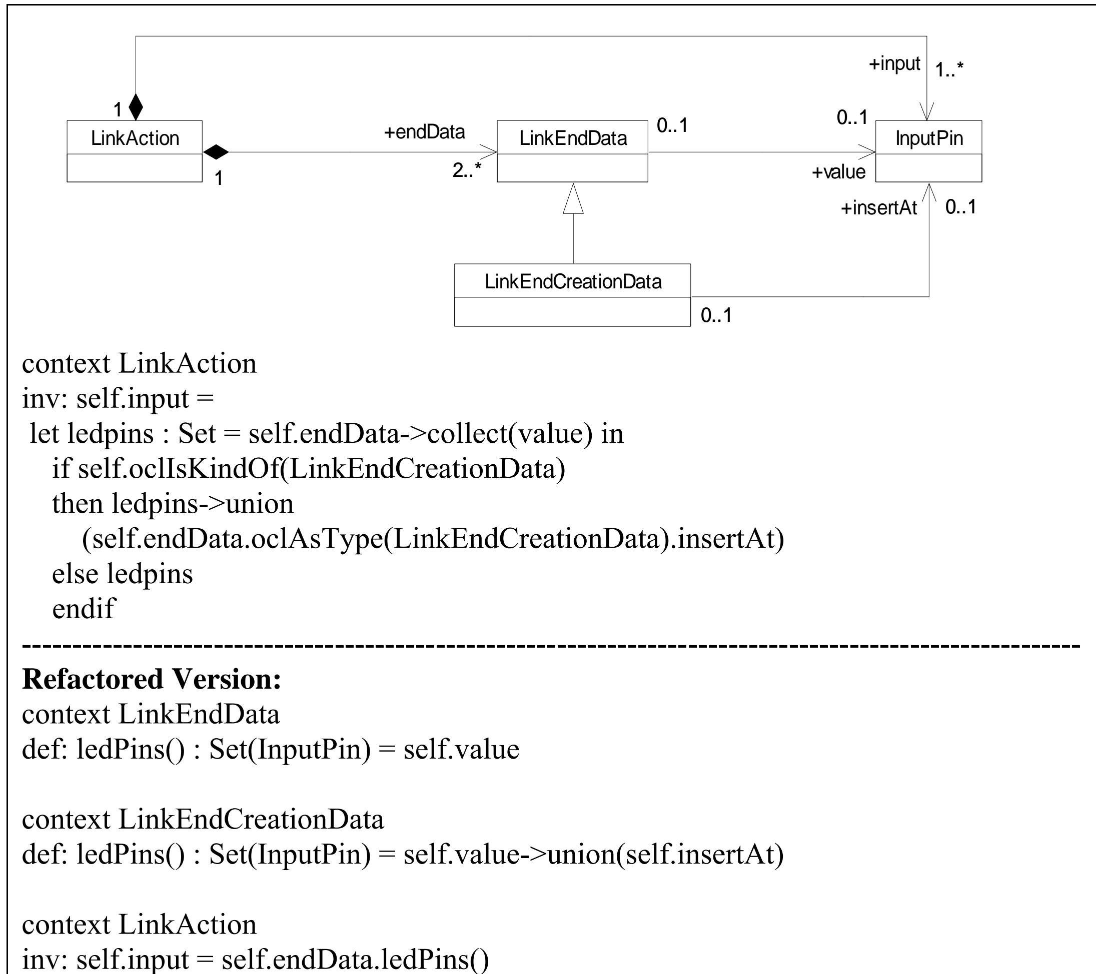
In most cases, this smell can be removed through one of the following refactorings: *Change Context* or *Introduce Polymorphism.* The *Introduce Polymorphism* refactoring replaces complex *if-then-else-endif* expressions that makes considerable use of operations such as *oclIsKindOf*, *oclIsTypeOf*, *oclAsType*, by a combination of generic and specific operations defined in a hierarchy of classes. This refactoring is also used in combination with other refactorings that are applied to the underlying model, such as *Add Class*, *Add Generalization*, *Pull Up*, *Push Down*, *Add Operation* among other possible model refactorings.

3.5. OCL Smell: Type Related Conditionals

This smell occurs in expressions of the form *if x.oclIsKindOf(A) then <exp1> else if x.oclIsKindOf(B) then <exp2> else ... endif*, i.e., the result of the expression depends on the type of a given object *x* which is obtained through calls to *oclIsKindOf* or *oclIsTypeOf* operations. This kind of structure results in less readable and less maintainable specifications. This smell can be removed by applying the *Introduce Polymorphism* refactoring described in section 3.4.

Figure 10 shows a constraint containing this smell, extracted from the UML 2.0 Superstructure Specification [OMG 2005b]. Besides presenting many syntactical and type checking errors, the invariant contains an *if-then-else-endif* structure based on the type of the *LinkEndData* objects associated to a *LinkAction* object that corresponds to the *Type Related Conditionals* smell.

The bottom part of Figure 10 shows the expressions that result from the application of the *Introduce Polymorphism* refactoring to this invariant. Since the *inputPins* associated to a *LinkEndData* object depends on its type (*value* for instances of *LinkEndData* and *value* + *insertAt* for instances of *LinkEndCreationData*), an operation *ledPins* was defined in both classes, so that the original invariant could be rewritten without using *oclIsKindOf* or *oclAsType* operations.




```
context LinkAction
inv: self.input =
 let ledpins : Set = self.endData->collect(value) in
   if self.oclIsKindOf(LinkEndCreationData)
   then ledpins->union
      (self.endData.oclAsType(LinkEndCreationData).insertAt)
   else ledpins
   endif
```

Refactored Version:

```
context LinkEndData
def: ledPins() : Set(InputPin) = self.value

context LinkEndCreationData
def: ledPins() : Set(InputPin) = self.value->union(self.insertAt)

context LinkAction
inv: self.input = self.endData.ledPins()
```

Figure 10. Example of the *Type Related Conditionals* Smell

4. An Experimental Study to Evaluate OCL Refactorings

Although there is anecdotal evidence on their usefulness, few quantitative evaluations of software refactorings have been published so far [Kataoka et al. 2002]. This section describes the planning of an experimental study that we have conducted in order to evaluate the impact of OCL smells and refactorings on the understandability of OCL specifications.

4.1. Definition

Using the structure defined in [Wohlin 2000], the study is defined as follows: analyze refactorings that can be applied to remove OCL smells from OCL constraints, for the purpose of evaluating the usefulness of the proposed refactorings, with respect to their benefits to the understandability of OCL constraints, from the point of view of the researcher, in the context of software developers reading and interpreting OCL constraints defined in a lab package.

4.2. Context and Material

The study procedure consists of reading and interpreting OCL constraints associated to elements defined in a UML model. The selected subjects were 23 graduated software developers who have attended to a 40-hour course in UML/OCL offered by Federal University of Rio de Janeiro. Their knowledge of OCL was restricted to basic OCL syntax and semantics, i.e., they were not aware of concepts such as OCL smells and OCL refactorings.

Each subject answered ten questions, each presenting an OCL constraint associated to an UML model and a small object diagram corresponding to a snapshot of objects of that model. A subject should answer whether and why the given snapshot violates the constraint. This UML model does not make use of elements with conflicting semantics such as composition or aggregation.

The subjects were divided in two groups (GI and GII) and each group answered a different set of questions (Set I or Set II). Each set was composed of five *S-Type* questions (interpretation of constraints containing OCL smells) and five *R-Type* questions (interpretation of refactored versions of the constraints present in the *S-type* questions answered by the other group). *R-type* questions of one set used the same object diagrams present in *S-type* questions of the other set.

By using two sets of questions with the proposed organization, we tried to expose all subjects to expressions containing the same number of OCL smells and at the same difficulty level. Table 1 presents the structure of each set of questions and the OCL smells present in each S-type question. The column *RQ (Related Question)* indicates the number of the question in the other set that contains the same OCL smell.

Table 1 – Composition of each set of questions

Set S1			Set S2		
Question	***OCL Smells***	***RQ***	***Question***	***OCL Smells***	***RQ***
S1	*Implies Chain*	10	R1	S1 – refactored	
S3	*Downcasting*	4	R3	S3 – refactored	
S5	*Forall Chain*	2	R5	S5 – refactored	
S7	*Verbose Expression*	8	R7	S7 – refactored	
S9	*Type Rel. Conditionals*	6	R9	S9 – refactored	
R2	S2 – refactored		S2	*Forall Chain*	5
R4	S4 – refactored		S4	*Downcasting*	3
R6	S6 – refactored		S6	*Type Rel. Contitionals*	9
R8	S8 – refactored		S8	*Verbose Expression*	7
R10	S10 – refactored		S10	*Implies Chain*	1

4.3. Hypothesis Formulation

The independent variable of this study is the type of the OCL expressions present in each answered question. This variable has two possible values: S (constraints with OCL

smells) and R (refactored constraints). We evaluated the impact of the independent variable on the following dependent variables:

a) Question Score (QS): the score of the subject on a specific question. QS is an integer variable, computed as follows: one point for the yes/no part of the answer and one point for the justification part. The variables SS and RS are defined for each subject as the sum of QS in questions of type S and R, respectively.
b) Question Time to Answer (QT): time in seconds spent by the subject to completely answer the question. The variables ST and RT are defined for each subject as the sum of QT in questions of type S and R, respectively.

Given these variables, we formulate the Null Hypothesis (H_0) as follows: "There is no difference in accuracy and time to answer interpretation questions on constraints containing OCL smells, when compared to the refactored versions of the same constraints". Therefore, H_O: $\mu SS = \mu RS$ and $\mu ST = \mu RT$.

The alternative hypothesis (H_1) is that OCL smells affect accuracy or time to answer those interpretation questions. To be more precise, H_1 should be one-tailed: we expect OCL smells to have one or both of the following effects: decreased accuracy or increased time to answer. Therefore, H_1: $\mu SS < \mu RS$ or $\mu ST > \mu RT$.

4.4. Study Design

The study was organized in the following phases:

a) ***Self-Study***: Each subject was given two weeks to study a written tutorial on OCL, elaborated by us.
b) ***OCL Assessment***: subjects were grouped in two blocks (higher scores and lower scores) according to the median of their scores on a ten question OCL test. Each group (GI and GII) was then randomly assigned eleven subjects from both blocks in nearly identical proportions. One subject was allocated to test the instrumentation.
c) ***Main session***: In the main session, each participant answered the set of ten questions assigned to his group (SI or SII). Each subject had to fully answer one question to proceed to the next one. They were not allowed to change answers of the preceding questions. This strategy allowed us to collect the time spent by the subjects on each question. No time limit was imposed to the subjects.
d) ***Subjective Evaluation***: After answering the set of questions, the participants were asked to classify each question according to two aspects: the difficulty level and the perceived quality of the OCL expressions. For the evaluation of the difficulty level of each question, we used a Likert scale from 1 to 5 (*1-very easy, 2-easy, 3-medium, 4-difficult, 5-very difficult*). The quality of the OCL expressions present in each question was evaluated according to the following nominal scale: *1- constraint is badly written; 2- not sure whether the constraint is well or badly written; 3-constraint is well written*.

5. Experimental Study: Results and Threats to Validity

5.1. Adequacy of Instruments

First, we analyzed the instruments used in the main session, since they were designed to provide a similar experience to all subjects. We applied an ANOVA test with 5% threshold (α-level) [Wohlin 2000] to compare the following data:

- mean score and time spent on each set of questions: we compared the average score and the average time to answer for groups I and II, taking into account both S-*type* and R-*type* questions;
- mean score and time spent on questions of the same type in each set of questions: we compared the average score and the average time to answer for groups I and II separately for each type of question;

The results showed no significant difference in scores and time to answer between the groups. Thus, there is no evidence that differences over the instruments should impose threats to further analyses.

5.2. Scores

Nine questions were correctly answered by all subjects (R1, R2, R3, R4, R7, R8, S2, S3 and S8). The bottom four scores correspond to *S-type* questions (S5, S10, S6 and S9, in ascending order). The total scores in all R-type questions were greater than or equal to their respective S-type questions, i.e., total score in question R_i >= total score in question S_i.

An ANOVA test (α-level = 0.05) was applied to the score of each type of question (µSS and µRS). The results shown in Table 2 rejected the null hypothesis H_0 in favor of the alternative hypothesis H_1: µSS < µRS, i.e., the mean score in *S-type* questions was lower than the mean score in *R-type* questions. Therefore, the results indicate that, at least in the sample analyzed in the study, the presence of OCL smells negatively impacts the understanding of OCL constraints.

Table 2 – ANOVA table: score in each type of question

Question Type	Size (N)	Sum of Squares	Mean Square	Individual Mean
S	110	339	278	7,95
R	110	402	378	9,27
F (ANOVA)	9,89	$\mathbf{F_{CRIT}}$	6,75	

5.3. Time to Answer

Results indicate that most subjects spent more time answering *S-type* questions than *R-type* questions. In 90% of the questions, the average time to answer *S-type* questions was higher than the average time to answer their respective *R-type* questions. Only in question 1, the time to answer the S-type question was slightly higher. In questions 3, 4, 7, 8 and 9, the average time spent on S-type questions was at least 100% higher than on their R-type counterparts.

An ANOVA test (α-level = 0.05) was applied to the time to answer each type of question (µST and µRT). The result shown in Table 3 rejected the null hypothesis H_0 in favor of the alternative hypothesis H_1: µST > µRT, i.e., the mean time to answer *S-type* questions was higher than the mean time to answer *R-type* questions. Therefore, the results indicate that, at least in the sample analyzed in the study, the presence of OCL smells can negatively impact the time needed to understand OCL constraints.

Table 3 – ANOVA table: time to answer each type of question

Question Type	Size (N)	Sum of Squares	Mean Square	Mean time spent in each question
S	105	13.236.000	10.047.146,67	05:14
R	103	8.525.400	5.942.403,88	04:00
F (ANOVA)	8,87	$\mathbf{F_{CRIT}}$	3,89	

5.4. Subjective Evaluation

Data collected from the subjective evaluation made by the subjects were analyzed in order to investigate whether they perceived some difference in the difficulty level of *S-type* questions compared to *R-type* questions. More than 60% of *R-type* questions were classified as easy or very easy, and only 10% of *R-type* questions were judged difficult or very difficult. On the other hand, less than 30% of *S-type* questions were classified as easy or very easy, while 30% of them were judged difficult or very difficult.

This evaluation also showed that there is a significant difference in the perceived quality of expressions present in *S-type* questions and *R-type* questions. While only 4% of the evaluations of *R-type* questions classified their expressions as of poor quality, that number raised to 36% in *S-type* questions. 80% of R-type expressions were evaluated as of good quality. However, a significant number of evaluations (44%) perceived expressions containing OCL smells (*S-type*) as of good quality.

5.5. Threats to Validity

This section discusses the different threats to the validity of results found in this study, in decreasing priority order: internal, external, construction, conclusion.

Internal validity is defined as the ability of a new study to replicate the observed behavior using the same subjects and instruments. We tried to minimize the threats to internal validity by submitting every subject to the same treatments and by alternating between *R-type* and *S-type* questions during the main session.

External validity reflects the ability to reproduce the same behavior in groups other than the ones that were analyzed. As in many academic studies, the issue of whether the subjects are representative of software professionals arises. We tried to involve subjects with different academic background and professional experience. We cannot state that the results of this study would occur in the same way using bigger and more complex models and OCL constraints. However, this issue is almost always present in experiments with industry professionals.

Construction validity refers to the relationship between the instruments / subjects and the theory under study. The study was carefully designed so that all subjects would have comparable and similar experiences. The results described in section 5.1 indicate that this goal was achieved. We followed an approach similar to those used in other empirical studies that evaluated some aspect related to program or specification understanding ([Briand et al 2005], [Finney et al 1999], [Snook and Harrison 2001]). The subjects were aware that we were attempting to evaluate some issues related to OCL, but they were not aware of the exact hypotheses we were testing or what results we were hoping to obtain.

Conclusion validity relates the treatments and the results, defining the ability of the study in generating some conclusion. We tried to obtain reliable results by using objective measures and statistical parametric tests. We also used subjective evaluations in order to support the quantitative results. Although the number of subjects could be considered low, we tried to increase the number of data points by submitting all subjects to both treatments.

6. Related Work

The basic ideas of model refactorings were presented in [Sunyé et al 2002], where the authors explored how the integrity of UML class diagrams and statecharts could be maintained after refactorings. Moreover, some refactorings related to statecharts were formally defined using OCL pre and postconditions. A detailed discussion about model refactorings can also be found in [Boger et al 2002], [Massoni et al 2005] and [Markovic and Baar 2005]. While those works are mostly related to refactorings applied to UML model elements that may take into account well written OCL expressions associated to them, our approach is focused on improving badly written OCL expressions associated to such elements.

A relevant issue regarding model transformations is to prove that a refactoring preserves the semantics previously described in the model. This requires a semantic interpretation of models that is amenable to formal analysis. Some results regarding this issue can be found in [Engels et al 2002] and [Mens et al 2002]. A rigorous approach for providing model refactorings is also described in [Gheyi et al 2005]. Our approach to support the proposed refactorings is based on manual and automated refactorings. Refactorings that can be precisely defined are implemented as update transformations on the set of instances representing the model. The implementation of each refactoring is written in OCL Action Language, an extension of OCL that can generate side effects. Manual refactorings are supported by the execution of tests initially developed to validate the semantics of a specification, but that can also be applied after performing a refactoring [Correa and Werner, 2006]. Although tests can not prove that a model is correct, consistent or complete, or that its semantics have been preserved after a transformation, we accept a lower level of assurance in return for more rapid feedback and a reduced reliance on formal proofs expertise.

Some OCL smells and refactorings described in this paper are closely related to the ones described in [Fowler 1999]. Although the smells described in that work are related to the implementation of methods, many of them can be adapted to OCL specifications. Some of the refactorings mentioned in section 2 (*Add Property Definition* and *Replace Expression by Property Call*) can be viewed as adaptations of the *Introduce Explaining Variable* refactoring described in [Fowler 1999].

7. Conclusions

Refactoring is considered an essential technique for handling software evolution. Since models are the central point in model driven approaches to software development, refactoring techniques and tools should also be developed at the model level. This paper showed how refactoring techniques can be applied to OCL expressions in order to remove bad constructions (OCL smells) that may negatively affects the understandability of OCL specifications.

The results of the empirical study described in this paper indicate that the presence of OCL smells in OCL expressions may have a negative impact in both the correctness and the time necessary to understand a constraint written in OCL. Subjects scored better and took less time to answer *R-type* questions. *S-type* questions were perceived as being more difficult. Moreover, we found a correlation of this level of difficulty perceived by the subjects and their performance. As it was somehow expected, the results confirmed the anecdotal evidence on the usefulness of refactorings in the understandability of OCL expressions.

The subjective evaluation of the perceived quality of the expressions reflects somehow the lack of experience of the subjects with OCL. Although a significant part was able to see that some expressions are more complex than necessary, few subjects were able to correctly explain how they could be made simpler. Besides the lack of experience with OCL, we believe that an additional reason for such results is that their knowledge is restricted to basic OCL syntax and semantics. Therefore, the results suggest that a catalogue of OCL smells and their respective refactorings could be an important asset that the software modeling community should consider to continually use and evolve in order to enhance the overall quality of OCL specifications.

So far, we have catalogued 15 OCL Smells and 25 Refactorings [Correa 2006]. As future work, we plan to expand those catalogues and also to develop tool support for both the automatic identification of OCL smells and the application of OCL and model refactorings.

References

Ambler, S.W. (2002), "Toward Executable UML", In: *Software Development Magazine*, January 2002 – http://www.sdmagazine.com.

Baar, T., Chiorean, D., Correa, A. et al (2005), "Tool Support for OCL and Related Formalisms - Needs and Trends". In: *Satellite Events at the MoDELS'2005 Conference - Selected Papers,* LNCS, v. 3844, pages 1-9, Montego Bay, Jamaica.

Berry, D. M., Kamsties, E. (2004), "Ambiguity in Requirements Specification". In Leite, J. C. S. P. and Doorn, J. H. (eds), *Perspectives On Software Requirements*, chapter 2, Kluwer Academic Publishers.

Boger, M., Sturm, T., Fragemann, P. (2002), "Refactoring Browser for UML". In: *International Conference on XP and Flexible Processes in Software Engineering*, pages 77-81, Alghero, Italy.

Briand, L.C., Labiche, Y., Penta, M. et al (2005), "An Experimental Investigation of Formality in UML-Based Development", In: *IEEE Transactions on Software Engineering* 31(10), pages 833-849.

Chiorean, D., Bortes, M., Corutiu, D., Sparleanu, R. (2004), "UML/OCL Tools – Objectives, Requirements, State of the Art: The OCLE Experience". In: Proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques", pages 163-180, Turku, Finland.

Correa, A. (2006), "Refactoring OCL Model Constraints Specifications", Doctoral Thesis – COPPE/UFRJ, In Portuguese.

Correa, A., Werner, C. (2004), "Applying Refactoring Techniques to UML/OCL Models". In: Baar, T., Moreira, A., Strohmeier, A., Mellor, S. (eds.) *<<UML>>*

2004 – The UML: Modeling Languages and Applications. 7th International Conference, LNCS, vol. 3273, Springer, pages 173-187, Lisbon, Portugal.

Correa, A., Werner, C. (2006), "Odyssey-PSW: Uma Ferramenta de Apoio à Verificação e Validação de Especificações de Restrições OCL". In: *XX Simpósio Brasileiro de Engenharia de Software – Tool Session*, Florianópolis, Brazil, In Portuguese.

Engels, G., Heckel, R., Küster, J.M., Groenewegen, L. (2002), "Consistency-preserving Model Evolution Through Transformations". In: *<<UML>> 2002 – The Unified Modeling Language: Model Engineering, Concepts and Tools. Proceedings of the 5th International Conference*, LNCS, vol. 2460, pages 212-226, Dresden, Germany.

Finney, K., Fenton, N., Fedorec, A. (1999), "Effects of structure on the comprehensibility of formal specifications", In: *IEE Proceedings – Software* 146(4), pages 193-202.

Fowler, M. (1999), "Refactoring – Improving the Design of Existing Code". Addison-Wesley.

Gheyi, R., Massoni, T., Borba, P. (2005), "A Rigorous Approach for Providing Model Refactorings". In: *20th IEEE/ACM International Conference on Automated Software Engineering (ASE),* pages 372-375, Long Beach, USA.

Jones, C. B. (1989), "Systematic Software Development Using VDM". Prentice-Hall.

Kataoka, Y., Imai, T., Andou, H., Fukaya, T. (2002), "A Quantitative Evaluation of Maintainability Enhancement by Refactoring". In: *Proceedings of the International Conference on Software Maintainance (ICSM)*, pages 576-585, Montreal, Canada.

Markovic, S. and Baar, T. (2005), "Refactoring OCL annotated UML class diagrams". In: *"Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005"*, LNCS, vol. 3713, pages 280-294, Montego Bay, Jamaica.

Massoni, T., Gheyi, R., Borba, P. (2005), "Formal Refactoring for UML Class Diagrams". In: *"XIX Simpósio Brasileiro de Engenharia de Software"*, pages 152-167, Uberlândia, Brazil.

Mens, T., Demeyer, S., Janssens, D. (2002), "Formalising behaviour preserving program transformations". In: "*Proceedings of the ICGT 2002 - First International Conference on Graph Transformation*", LNCS, vol. 2505, Barcelona, Spain.

Mens, T., Tourwe, T. (2004), "A Survey of Software Refactoring". In: *IEEE Transactions on Software Engineering*, 30(2), pages 126-139.

OMG – Object Management Group (2003a), "UML 2.0 OCL Specification", http://www.omg.org/cgi-bin/doc?ptc/2003-10-14.

OMG – Object Management Group (2003b), "Model Driven Architecture (MDA) Guide" - document number omg/2003-06-01.

OMG – Object Management Group (2005a), "MOF QVT Final Adopted Specification" - document ptc/05-11-01.

OMG – Object Management Group (2005b), "Object Management Group - Unified Modeling Language (UML) Superstructure Specification, version 2.0" - http://www.omg.org/cgibin/doc?formal/05-07-04.

OMG – Object Management Group (1999), "Object Management Group - Unified Modeling Language (UML) 1.3 specification" - http://www.omg.org/cgi-bin/doc?formal/00-03-01.

Snook, C.F., Harrison, R. (2001), "Experimental Comparison of the Comprehensibility of a Z Specification and its Implementation". In: *Proceedings of the Conference on Empirical Assessment in Software Engineering – EASE 01,* Keele University, England.

Sunyé, G., Pollet, D., Letraon, Y., et al (2001), "Refactoring UML models". In: *<<UML 2001>> The UML: Modeling Languages, Concepts and Tools. 4th International Conference,* LNCS, v. 2185, pages 134-148, Toronto, Canada.

Vaziri, M., Jackson, D. (2002), "Some Shortcomings of OCL, the Object Constraint Language of UML". In: *Proceedings of the Technology of Object Oriented Languages and Systems Conference (TOOLS USA 34)*, pages 555-562, Santa Barbara, California, USA.

Warmer, J., Kleppe, A. (2003), "The Object Constraint Language – Getting Your Models Ready for MDA". Addison-Wesley.

Wohlin, C., Runeson, P., Höst, M, et al (2000), "Experimentation in Software Engineering: An Introduction", Kluver Academic Publishers.

Woodcock, J., Davis, J. (1996), "Using Z. Specification, Refinement and Proof". Prentice Hall.