OCL Bad Smells and Refactoring Techniques

Formal Methods

Dr. Wafa Basit

Why Does This Matter?

- Object Constraint Language (OCL) is used to define **precise rules** in UML models.
- Clean OCL improves model **readability, maintainability, and correctness.**
- Well-written constraints reduce **ambiguity** and aid in **validation** and **verification**.

What is a Code Smell in OCL?

- A code smell is an indicator that something is wrong with the code, suggesting it might need refactoring or improvement to enhance its quality and maintainability.
- They are not bugs that cause immediate failures, but rather symptoms of underlying issues that can lead to problems down the line.
- Recognizing and refactoring smells helps you write professional, scalable specifications.

Common OCL Smells Overview

1. Implies Chain
2. ForAll Chain
3. Verbose Expression
4. Long Journey
5. Duplication
6. Downcasting
7. Type-Based Conditionals

Each has specific refactoring techniques for resolution.

Implies Chain

Nested implies expressions (e.g. A implies (B implies C))

Why it’s a problem: Hard to follow the logic flow. Deep nesting obscures intent.

Example:

Constraint : If the customer's age is greater than 18, then if they are eligible, they must have a valid ID.

context Customer

inv: self.age > 18 implies (self.isEligible = true implies self.hasValidId = true)

ForAll Chain

Two or more nested forAll quantifiers

Why it's a problem: It makes the expression long, complex, and harder to read.

Example:

Constraint : For every item the customer has rented, all items in that item's category must cost more than 100.

```
context Customer

    inv allPremiumExclusive:

    self.rentedItems->forAll(i | i.category.items->forAll(j | j.price > 100))
```

Verbose Expression

Unnecessarily long or redundant boolean logic

Why it’s a problem: Adds clutter Falls prey to typos or logic mistakes

Example:

Constraint : Unpaid orders are not valid.

context Order

inv validStatus:

if self.isPaid then true else false endif

Long Journey

Very deep navigation chains through many associations

Why it’s a problem: Fragile: any null link breaks it. Difficult to read and maintain

It’s like saying:

“My friend’s brother’s dog’s vet” — too long and messy.

Example: **Constraint**: The customer's country code on the invoice must be 'US'.

context Invoice

    inv deepCheck:

    self.customer.address.region.country.code = 'US'

Duplication

Same sub-expression repeated multiple times

Why it’s a problem: If you need to change the logic, you'll have to update it everywhere it’s repeated. If you forget to update one place, it could lead to errors or inconsistencies.

Example: **Constraint** : The product price must be greater than 0 and less than 500, excluding 100.

context Product

inv validDiscount:

    self.price > 0.0 and self.price < 500.0 and self.price <> 100.0

Downcasting

oclAsType() and oclIsTypeOf() to access subclass features

Why it’s a problem: Breaks abstraction. Fails if model evolves (new subclasses)

Example:

context Animal

inv validDogWeight:

  self.oclAsType(Dog).weight > 10

Type-Based Conditionals

if … oclIsTypeOf(…) then … else … endif blocks

Why it's a problem: Mixes logic for multiple types in one place Hard to extend when new types arrive

Example

Constraint : If the rental item is a movie, its rental fee must be greater than 5

context Person

inv taxRule: if self.oclIsTypeOf(Employee) then
self.oclAsType(Employee).taxClass = 'TC1' else true endif

Refactoring

Refactoring is the process of restructuring existing code or constraints without changing their external behavior.

Primary Goals:

- Improve Readability
- Enhance Maintainability
- Extensibility

Refactoring Implies Chain

Decompose conditional → combine antecedents into a single implication

inv: self.age > 18 implies

(self.isEligible = true implies
self.hasValidId = true)

self.age > 18 and

self.isEligible implies self.hasValidId

Refactoring ForAll Chain

Flatten navigation → single forAll on composed path

```
context Customer
inv allPremiumExclusive:
  self.rentedItems->forAll(i |
    i.category.items->forAll(j |
      j.price > 100))
```

```
context Customer
inv allPremiumExclusive:
  self.rentedItems.category.items
    ->forAll(item | item.price > 100)
```

Refactoring Verbose Expression

Flatten navigation → single forAll on composed path

```
context Order

    inv validStatus:

    if self.isPaid then true else false
    endif
```

```
context Order

inv validStatus:

  self.isPaid
```

Refactoring Long Journey

Decompose conditional → combine antecedents into a single implication

```
context Invoice

inv checkCountry:

self.customer.address.region.country.code = 'US'
```

```
context Invoice

inv checkCountry:

  let c = self.customer.address.region.country in c.code = 'US'

OR

context Customer

def: countryCode: String = self.address.region.country.code

context Invoice

inv checkCountry:

  self.customer.countryCode = 'US'
```

Refactoring Duplication

Extract repeated logic into let or helper operation

```
context Product
inv validDiscount:
  self.price > 0.0 and
  self.price <= 100.0 and
  self.price <> 50.0
```

```
context Product
inv validDiscount:
  let p = self.price in
    p > 0.0 and p <= 100.0 and p <> 50.0
```

Refactoring Downcasting

Move constraint to subclass → leverage context's type

```
context Animal
inv validDogWeight:

self.oclAsType(Dog
).weight > 10
```

Define a default operation on the superclass:

```
context Animal

def: effectiveWeight(): Real = 0.0
```

Override it in `Dog` to return its real weight:

```
context Dog

def: effectiveWeight(): Real =
self.weight
```

Replace the casted check by a single invariant:

```
context Animal

inv dogWeightAbove10:

  self.effectiveWeight() >
10
```

Refactoring Type-Based Conditionals

Replace with subclass invariants (polymorphism)

```
context Person

inv taxRule:

  if self.oclIsTypeOf(Employee) then

    self.oclAsType(Employee).taxClass =
'TC1'

  else

    true

  endif
```

```
context Employee

inv taxRule:

  self.taxClass = 'TC1'
```

Reference

Correa, A. L., Werner, C. M. L., & Barros, M. O. (2007). *An Empirical Study of the Impact of OCL Smells and Refactorings on the Understandability of OCL Specifications*. In *MoDELS 2007: 10th International Conference on Model Driven Engineering Languages and Systems* (pp. 76–90). Springer.

https://www.researchgate.net/publication/221223981_An_Empirical_Study_of_the_Impact_of_OCL_Smells_and_Refactorings_on_the_Understandability_of_OCL_Specifications