UML for Java Developers
Model Constraints & The Object Constraint Language

Jason Gorman
If you're reading this tutorial, you're probably either studying for or teaching a computing or software engineering academic qualification.

How do I know this?

Simple. In the real world, almost nobody uses OCL. And by "almost nobody", I mean maybe 1/100 software professionals may have learned it. And maybe 1/100 of them ever use it. Learning OCL very probably is not going to get you a job as anything other than someone who teaches OCL.

I believe it is a useful skill to have if you want to really get to grips with UML, but I can state categorically, with my hand on my heart, than I have in my entire career used OCL in anger maybe twice.

Knowing OCL will not make you a better software developer, and you are unlikely to work with other software developers who know OCL, rendering it useless as a communication tool.

You may have been told about Model-driven Architecture. Back in 2000, it was going to be the next big thing. It wasn't.

On 99.9% of professional software projects, we still type code in a third-generation language into a text editor. Occasionally we draw UML diagrams on whiteboards when we want to visualise a design or analysis concept. You will find that some UML notations are still in widespread use - especially class and sequence diagrams, and activity diagrams for workflow analysis.

Consider OCL as being a classical language, like Latin or Ancient Greek.

It's useful to know, as it can give you some general background when applying things like Design By Contract, or even for functional programming. But it is, too all intents and purposes, a dead language.

Trust me, almost nobody out here speaks it.

Having said that, I hope you find this tutorial useful in passing your exams. And I look forward to maybe teaching you some useful skills - like test-driven development or refactoring - when you graduate and join the community of professional software developers.

Best wishes,

Jason Gorman
UML Diagrams Don't Tell Us Everything
Constraints Make Models More Precise

The diagram shows a class diagram with a class `Person` having an association `children` with a multiplicity of `*`. There is also an association `parents` with a multiplicity of `0..2`. The instance `Bill : Person` has a self-loop on the `children` association with a note `{cannot be own descendant or ancestor}` and a dashed association connecting it back to the `Person` class, indicating that it is not a valid instance of `Person`.
What is the Object Constraint Language?

• A language for expressing necessary extra information about a model

• A precise and unambiguous language that can be read and understood by developers and customers

A language that is purely declarative, i.e., it has no side-effects (in other words it describes what rather than how)
What is an OCL Constraint?

• An OCL constraint is an OCL expression that evaluates to true or false (a Boolean OCL expression, in other words)
OCL Makes Constraints Unambiguous

{ancestors = parents->union(parents.ancestors->asSet())}
{descendants = children->union(children.descendants->asSet())}

{ancestors->excludes(self) and descendants->excludes(self)}
Introducing OCL  Constraints & Contexts

Q: To what type this constraint apply?
A: Person

context Person
inv: ancestors->excludes(self) and descendants->excludes(self)

Q: When does this constraint apply?
A: inv = invariant = always
Operations, Pre & Post-conditions

<table>
<thead>
<tr>
<th>&lt;&lt;enumeration&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sex</td>
</tr>
<tr>
<td>MALE = 1</td>
</tr>
<tr>
<td>FEMALE = 2</td>
</tr>
</tbody>
</table>

Optional constraint name applies to the marry() operation of the type Person

context Person::marry(p : Person)
pre cannot_marry_self: not (p = self)
pre not_same_sex: not (p.sex = self.sex)
-- neither person can be married already
pre not_already_married: self.spouse->size() = 0 and p.spouse->size() = 0

Comments start with --
Design By Contract :assert

class Sex
{
    static final int MALE = 1;
    static final int FEMALE = 2;
}

class Person
{
    public int sex;
    public Person spouse;

    public void marry(Person p)
    {
        assert p != this;
        assert p.sex != this.sex;
        assert this.spouse = null && p.spouse = null;
        this.spouse = p;
        p.spouse = this;
    }
}

context Person::marry(p : Person)
pre cannot_marry_self:  not (p = self)
pre not_same_sex: not (p.sex = self.sex)
-- neither person can be married already
pre not_already_married: self.spouse->size() = 0 and p.spouse->size() = 0
post: self.spouse = p and p.spouse = self

assert p != this;
assert p.sex != this.sex;
assert this.spouse = null && p.spouse = null;
this.spouse = p;
p.spouse = this;

self.spouse->size = 0
```java
public class Person {
    public int sex;
    public Person spouse;

    public void marry(Person p) throws ArgumentException {
        if (p == this) {
            throw new ArgumentException("cannot marry self");
        }
        if (p.sex == this.sex) {
            throw new ArgumentException("spouse is same sex");
        }
        if ((p.spouse != null || this.spouse != null) {
            throw new ArgumentException("already married");
        }
        this.spouse = p;
        p.spouse = this;
    }
}
```
Referring to previous values and operation return values

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance : Real = 0</td>
</tr>
<tr>
<td>deposit(amount : Real)</td>
</tr>
<tr>
<td>withdraw(amount : Real)</td>
</tr>
<tr>
<td>getBalance() : Real</td>
</tr>
</tbody>
</table>

context Account::withdraw (amount : Real)
pre: amount <= balance
post: balance = balance@pre - amount

context Account::getBalance() : Real
post: result = balance

balance before execution of operation

return value of operation
@pre and result in Java

```java
class Account {
    private float balance = 0;
    public void withdraw(float amount) {
        assert amount <= balance;
        balance = balance - amount;
    }
    public void deposit(float amount) {
        balance = balance + amount;
    }
    public float getBalance() { return balance; }
}
```

```java
class Account {
    private float balance = 0;
    public void withdraw(float amount) {
        assert amount <= balance;
        balance = balance - amount;
    }
    public void deposit(float amount) {
        balance = balance + amount;
    }
    public float getBalance() { return balance; }
}
```

```java
public void testWithdrawWithSufficientFunds() {
    Account account = new Account();
    account.deposit(500);
    float balanceAtPre = account.getBalance();
    float amount = 250;
    account.withdraw(amount);
    assertTrue(account.getBalance() == balanceAtPre - amount);
}
```

```
public void testWithdrawWithSufficientFunds() {
    Account account = new Account();
    account.deposit(500);
    float balanceAtPre = account.getBalance();
    float amount = 250;
    account.withdraw(amount);
    assertTrue(account.getBalance() == balanceAtPre - amount);
}
```
OCL Basic Value Types

- Integer: A whole number of any size
- Real: A decimal number of any size
- String: A string of characters
- Boolean: True/False

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance : Real = 0</td>
</tr>
<tr>
<td>name : String</td>
</tr>
<tr>
<td>id : Integer</td>
</tr>
<tr>
<td>isActive : Boolean</td>
</tr>
</tbody>
</table>

| deposit(amount : Real) |
| withdraw(amount : Real) |
# Operations on Real and Integer Types

<table>
<thead>
<tr>
<th>Operation</th>
<th>Notation</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals</td>
<td>a = b</td>
<td>Boolean</td>
</tr>
<tr>
<td>not equals</td>
<td>a &lt;&gt; b</td>
<td>Boolean</td>
</tr>
<tr>
<td>less</td>
<td>a &lt; b</td>
<td>Boolean</td>
</tr>
<tr>
<td>more</td>
<td>a &gt; b</td>
<td>Boolean</td>
</tr>
<tr>
<td>less or equal</td>
<td>a &lt;= b</td>
<td>Boolean</td>
</tr>
<tr>
<td>more or equal</td>
<td>a &gt;= b</td>
<td>Boolean</td>
</tr>
<tr>
<td>plus</td>
<td>a + b</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>minus</td>
<td>a - b</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>multiply</td>
<td>a * b</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>divide</td>
<td>a / b</td>
<td>Real</td>
</tr>
<tr>
<td>modulus</td>
<td>a.mod(b)</td>
<td>Integer</td>
</tr>
<tr>
<td>integer division</td>
<td>a.div(b)</td>
<td>Integer</td>
</tr>
<tr>
<td>absolute value</td>
<td>a.abs</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>maximum</td>
<td>a.max(b)</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>minimum</td>
<td>a.min(b)</td>
<td>Integer or Real</td>
</tr>
<tr>
<td>round</td>
<td>a.round</td>
<td>Integer</td>
</tr>
<tr>
<td>floor</td>
<td>a.floor</td>
<td>Integer</td>
</tr>
</tbody>
</table>

Eg, 6.7.floor() = 6
# Operations on String Type

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td>s.concat(string)</td>
<td>String</td>
</tr>
<tr>
<td>size</td>
<td>s.size</td>
<td>Integer</td>
</tr>
<tr>
<td>to lower case</td>
<td>s.toLower</td>
<td>String</td>
</tr>
<tr>
<td>to upper case</td>
<td>s.toUpper</td>
<td>String</td>
</tr>
<tr>
<td>substring</td>
<td>s.substring(int, int)</td>
<td>String</td>
</tr>
<tr>
<td>equals</td>
<td>s1 = s2</td>
<td>Boolean</td>
</tr>
<tr>
<td>not equals</td>
<td>s1 &lt;&gt; s2</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

Eg, `jason.concat(gorman) = jason gorman`
Eg, `jason.substring(1, 2) = ja`
# Operations on Boolean Type

<table>
<thead>
<tr>
<th>Operation</th>
<th>Notation</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>a or b</td>
<td>Boolean</td>
</tr>
<tr>
<td>and</td>
<td>a and b</td>
<td>Boolean</td>
</tr>
<tr>
<td>exclusive or</td>
<td>a xor b</td>
<td>Boolean</td>
</tr>
<tr>
<td>negation</td>
<td>not a</td>
<td>Boolean</td>
</tr>
<tr>
<td>equals</td>
<td>a = b</td>
<td>Boolean</td>
</tr>
<tr>
<td>not equals</td>
<td>a &lt;&gt; b</td>
<td>Boolean</td>
</tr>
<tr>
<td>implication</td>
<td>a implies b</td>
<td>Boolean</td>
</tr>
<tr>
<td>if then else</td>
<td>if a then b1 else b2 endif</td>
<td>type of b</td>
</tr>
</tbody>
</table>

Eg, true or false = true
Eg, true and false = false
Navigating in OCL Expressions

In OCL:

account.holder

Evaluates to a customer object who is in the role holder for that association

And:

customer.accounts

Evaluates to a collection of Account objects in the role accounts for that association

```java
Account account = new Account();
Customer customer = new Customer();
customer.accounts = new Account[] {account};
account.holder = customer;
```
Navigability in OCL Expressions

a.b is allowed

b.a is not allowed  it is not navigable
Calling class features

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>id : Integer</td>
</tr>
<tr>
<td>status : enum{active, frozen, closed}</td>
</tr>
<tr>
<td>balance : Real</td>
</tr>
<tr>
<td>nextId : Integer</td>
</tr>
<tr>
<td>deposit(amount : Real)</td>
</tr>
<tr>
<td>withdraw(amount : Real)</td>
</tr>
<tr>
<td>fetch(id : Integer) : Account</td>
</tr>
</tbody>
</table>

context Account::createNew() : Account
post: result.oclIsNew() and
result.id = Account.nextId@pre and
Account.nextId = result.id + 1
Enumerations in OCL

context Account::withdraw(amount : Real)
pre: amount <= balance
pre: status = AccountStatusKind.ACTIVE
post: balance = balance@pre - amount
Collections in OCL

customer.accounts.balance = 0 is not allowed

customer.accounts->select(id = 2324).balance = 0 is allowed
class Account
{
    public double balance;
    public int id;
}

class Customer
{
    Account[] accounts;

    public Account SelectAccount(int id)
    {
        Account selected = null;

        for(int i = 0; i < accounts.length; i++)
        {
            Account account = accounts[i];
            if(account.id == id)
            {
                selected = account;
            }
        }
        return selected;
    }
}
The OCL Collection Hierarchy

- **Collection**
  - **Set**
    - Elements can be included only once, and in no specific order
  - **Bag**
    - Elements can be included more than once, in no specific order
  - **Sequence**
    - Elements can be included more than once, but in a specific order
# Operations on All Collections

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>The number of elements in the collection</td>
</tr>
<tr>
<td>count(object)</td>
<td>The number of occurrences of object in the collection.</td>
</tr>
<tr>
<td>includes(object)</td>
<td>True if the object is an element of the collection.</td>
</tr>
<tr>
<td>includesAll(collection)</td>
<td>True if all elements of the parameter collection are present in the current collection.</td>
</tr>
<tr>
<td>isEmpty</td>
<td>True if the collection contains no elements.</td>
</tr>
<tr>
<td>notEmpty</td>
<td>True if the collection contains one or more elements.</td>
</tr>
<tr>
<td>iterate(expression)</td>
<td>Expression is evaluated for every element in the collection.</td>
</tr>
<tr>
<td>sum(collection)</td>
<td>The addition of all elements in the collection.</td>
</tr>
<tr>
<td>exists(expression)</td>
<td>True if expression is true for at least one element in the collection.</td>
</tr>
<tr>
<td>forAll(expression)</td>
<td>True if expression is true for all elements.</td>
</tr>
<tr>
<td>select(expression)</td>
<td>Returns the subset of elements that satisfy the expression</td>
</tr>
<tr>
<td>reject(expression)</td>
<td>Returns the subset of elements that do not satisfy the expression</td>
</tr>
<tr>
<td>collect(expression)</td>
<td>Collects all of the elements given by expression into a new collection</td>
</tr>
<tr>
<td>one(expression)</td>
<td>Returns true if exactly one element satisfies the expression</td>
</tr>
<tr>
<td>sortedBy(expression)</td>
<td>Returns a Sequence of all the elements in the collection in the order specified (expression must contain the &lt; operator)</td>
</tr>
</tbody>
</table>
Examples of Collection Operations

jason.accounts->forAll(a : Account | a.balance > 0) = true
jason.accounts->select(balance > 100) = {account1, account3}
jason.accounts->includes(account4) = true
jason.accounts->exists(a : account | a.id = 333) = false
jason.accounts->includesAll({account1, account2}) = true
jason.accounts.balance->sum() = 850

bool forAll = true;
foreach(Account a in accounts)
{
    if(!(a.balance > 0))
    {
        forAll = forAll && (a.balance > 0);
    }
}
Navigating Across & Flattening Collections

tsb.customers.accounts = {account1, account2, account3, account4}
tsb.customers.accounts.balance = {450, 100, 250, 50}
Specialized Collection Operations

<table>
<thead>
<tr>
<th>Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set</strong></td>
</tr>
<tr>
<td>minus(Set) : Set</td>
</tr>
<tr>
<td>union(Set) : Set</td>
</tr>
<tr>
<td>union(Bag) : Bag</td>
</tr>
<tr>
<td>symmetricDifference(Set) : Set</td>
</tr>
<tr>
<td>intersection(Set) : Set</td>
</tr>
<tr>
<td>intersection(Bag) : Set</td>
</tr>
<tr>
<td>including(OclAny) : Set</td>
</tr>
<tr>
<td>excluding(OclAny) : Set</td>
</tr>
<tr>
<td>asBag() : Bag</td>
</tr>
<tr>
<td>asSequence() : Sequence</td>
</tr>
</tbody>
</table>

| **Bag**    |
| union(Bag) : bag |
| union(Set) : bag |
| intersection(Set) : Set |
| intersection(Bag) : Bag |
| including(OclAny) : Bag |
| excluding(OclAny) : Bag |
| asSet() : Set |
| asSequence() : Sequence |

| **Sequence** |
| first() : OclAny |
| last() : OclAny |
| at(Integer) : OclAny |
| append(OclAny) |
| prepend(OclAny) |
| including(OclAny) : Sequence |
| excluding(OclAny) : Sequence |
| asBag() : Bag |
| asSet() : Set |

Eg, Set{4, 2, 3, 1}.minus(Set{2, 3}) = Set{4, 1}
Eg, Bag{1, 2, 3, 5}.including(6) = Bag{1, 2, 3, 5, 6}
Eg, Sequence{1, 2, 3, 4}.append(5) = Sequence{1, 2, 3, 4, 5}
Navigating across Qualified Associations

customer.account[3435]

Or

customer.account[id = 3435]
Navigating to Association Classes

context A inv: self.c
context B inv: self.c
context A inv: self.c[x]
context A inv: self.c[y]
Equivalents to Association Classes
Built-in OCL Types : OclType

Eg, Account.name() = Account
Eg, Account.attributes() = Set{ balance , id }
Eg, Customer.supertypes() = Set{Person}
Eg, Customer.allSupertypes() = Set{Person, Party}
Built-in OCL Types : OclAny

- `oclIsKindOf(OclType) : Boolean`
- `oclIsTypeOf(OclType) : Boolean`
- `oclAsType(OclType) : OclAny`
- `oclInState(OclState) : Boolean`
- `oclIsNew() : Boolean`
- `oclType() : OclType`

Eg, `jason.oclType() = Customer`

Eg, `jason.oclIsKindOf(Person) = true`

Eg, `jason.oclIsTypeOf(Person) = false`

Eg, `Account.allInstances() = Set{account1, account2, account3, account4}`
More on OCL

- OCL 1.5 Language Specification
- OCL Evaluator - a tool for editing, syntax checking & evaluating OCL
- Octopus OCL 2.0 Plug-in for Eclipse
www.parlezuml.com