Object Constraint Language

- The OCL is a language of typed expressions.
- An OCL expression is valid if it is written according to the rules (formal grammar) of OCL.
- A constraint is a valid OCL expression of type Boolean.
- A constraint is a restriction on one or more values of (part of) an object-oriented model or system.

Constraint examples

```
self.transaction -> forAll(t:Transaction | t.value > 100)
```

```
Account[1..]
Transaction
SuperSaver
Account
self.balance > 0
```

Design by contract

```
put (element: T, key: STRING) is
  -- insert element with given key
  require
  count < capacity
  do
    .. insertion algorithm ...
  ensure
    count <= capacity;
    item (key) = element;
    count = old count + 1
  end
end --put
```

Obligations Benefits

<table>
<thead>
<tr>
<th>Client</th>
<th>Contractor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call put only on a non-full table</td>
<td>Insert x so that it may be retrieved through key</td>
</tr>
<tr>
<td>Get modified table in which x is associated with key</td>
<td>No need to deal with the case in which the table is full before insertion</td>
</tr>
</tbody>
</table>

The requirements of OCL

1. The OCL must enable us to express extra, necessary, information on object models.
2. The OCL must be a precise and unambiguous language that can be easily read by developers and customers.
3. The OCL must be a declarative language, its expressions can have no side-effects.
4. OCL must be a typed language so that OCL expressions can be type checked for correctness.

Types in OCL

The types in OCL are as follows:

- Predefined types
  - Basic types - Integer, Real, String and Boolean
  - Collection types - Collection, Set, Bag, Sequence
- Meta types
  - OclAny, OclExpression, OclType
- User-defined model types
  - Enumeration and all classes, types and interfaces

The context of an OCL expression

```
self.hintColor <= 5 and self.hintColor >= 0
```
### Real and Integer

<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>modulo</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>

### Multiplicity constraints

**Vocabulary**

- self.hint -> size >= 0 and self.hint -> size <= 5

**VocabElement**

- self.vocabulary -> size = 1

**Hint**

- self.vocabElement -> size = 1

### String

<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>

### Using Enumerations

**Customer**

- gender: enum{male, female}
- name: String
- title: String
- dateOfBirth: Date

```
gender = #male implies title = 'Mr. '
```

### Boolean

<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>

### Simple boolean constraints

**Customer**

- name: String
- title: String
- age: Integer
- isMale: Boolean

```
title = if isMale then 'Mr.' else 'Ms.' endif
age >= 18 and age < 66
name.size < 100
```
The implication operation

Probably the most counter-intuitive logical operator.

\[ \begin{array}{ccc}
A & B & A \implies B \\
T & T & T \\
T & F & F \\
F & ? & T \\
\end{array} \]

Torches with broken bulbs?

Operations on 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>
</tbody>
</table>

Navigating Associations

- **Account**
  - self.transaction returns a set of transactions

- **Book**
  - self.borrower returns a set of members

Navigating to collections

- **Customer**
  - self.account produces a set of Accounts

- **Customer**
  - self.account.transaction produces a bag of transactions

If we want to use this as a set we have to do the following:

- self.account.transaction -> asSet

The subset constraint

- **Flight**
  - self.crew -> includes( self.pilot )
  - self.crew -> includesAll( self.flightAttendants )
### Iteration over collections

```ocl
Account
self.transactions -> select( value > 500 )
```

```ocl
Account
self.transactions -> reject( not(value > 500 ) )
```

```ocl
Account
self.transactions -> sum(collect ( value ))
```

### Abstract classes

The fact that the Hint class is abstract can be expressed this way.

```ocl
Hint
Hint.allInstances -> select(oclType = Hint) -> isEmpty
```

This constraint uses some of the meta facilities which I have mostly glossed over so far.

### OCL references

Warmer, J. Kleppe, A. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley 1999

*Object Constraint Language Specification*, version 1.1

OMG document ad970808

The OCL parser can be found at http://www.software.ibm.com/ad/ocl

The Klasse Objecten web site is http://www.klasse.nl/Engels/ocl.htm