Building Models with OCL

- Introduction
- Completing UML Diagrams
- Modeling Tips and Hints
- Summary

What Is a Model?

Simply put, a **model** is a high level system description.

It consists of a consistent, coherent set of **model elements**, i.e., artifacts written in a well-defined modeling language such as UML and OCL.

**Question:** Why do we want to have a **model**?
View Consistency

An UML model consists of several diagrams, each of which presents one view of the model.

Multiple views must be consistent. For example, objects in the interaction diagram must be instances of classes present in the class diagram.

UML and OCL (1)

Both UML diagrams and OCL expressions are part of a model. They are connected by the context definition.

Note that OCL expressions can be written in a textual format or attached to the UML diagrams directly.
**Example**

```plaintext
LoyaltyAccount
points: Integer
number: Integer
earn (i: Integer)
burn (i: Integer)
isEmpty(): Boolean

Transaction
points: Integer
date: Date
amount: Real
program(): LoyaltyProgram

inv: transactions.card.owner -> size() = 1
pre: i > 0
init: 0
body: points = 0
```

---

**Adding Extra Info**

OCL expressions can add extra info. to UML diagrams in the following situations:

- Elements might be under-specified.
- Business rules need to be incorporated.
- Interfaces need to be precisely defined.
- Ambiguity must be removed.
Building Models with OCL

- Introduction
- Completing UML Diagrams
- Modeling Tips and Hints
- Summary

Discovering Constraints

OCL expressions can be used to specify constraints on a class diagram, but where do they come from?

- Uniqueness Constraints
- Optional & Dynamic Multiplicities
- Inheritance
- Association Cycles
Uniqueness Constraints (1)

Recall the following uniqueness constraint:

```
context Person
inv: Person::allInstances() -> isUnique (socSecNr)
```

**Question:** How to determine if the value of a attribute needs to be unique?

Uniqueness Constraints (2)

- In general, we should ask the question: "Should equality imply identity?"
  - If two people have the same name, does it mean they are the same person?
  - If two people have the same social security number, does it mean they are the same person?

- One way to identify uniqueness constraints is looking for "sharing"
  - If an instance can be shared, then it is not unique.
**Optional Multiplicity (1)**

An optional multiplicity indicates that an associated object may or may not exist.

The existence of an associated object often depends on the state of the objects involved.

In this case, it is necessary to use OCL constraints to describe precisely when the optional association may be empty.

**Optional Multiplicity (2)**

```
<table>
<thead>
<tr>
<th>House</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>value : Money</td>
<td>socSecNr : Integer</td>
</tr>
<tr>
<td></td>
<td>salary : Money</td>
</tr>
<tr>
<td>security : 1</td>
<td>getMortage (sum : Money, security : House)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mortgage</th>
<th>borrower</th>
</tr>
</thead>
<tbody>
<tr>
<td>principal : Money</td>
<td>0 .. *</td>
</tr>
<tr>
<td>monthlyPayment : Money</td>
<td>mortgages</td>
</tr>
<tr>
<td>startDate : Date</td>
<td>mortgages</td>
</tr>
<tr>
<td>endDate : Date</td>
<td></td>
</tr>
</tbody>
</table>
```

```
**Dynamic Multiplicity (1)**

A *dynamic multiplicity* is one that needs to be determined based on another value in the system. In this case, OCL expressions can be used to precisely specify how to determine such a multiplicity.

**Dynamic Multiplicity (2)**

```plaintext
Flight
- Flightnr: Integer
- availableSeats(): Integer
  - flights 0 .. *
  - passengers 0 .. *

Airplane
- numOfSeats: Integer

Person
- name: String
```
Inheritance (1)

Inheritance (2)
**Association Cycles (1)**

An association cycle consists of a set of classes and associations such that one can start from one class, navigate through the associations, and return to the same class.

Cycles are often the source of ambiguity. The general rule is that cycles should be checked carefully, especially when the multiplicities are higher than one.

**Association Cycles (2)**
**Undefined Value**

Some OCL expressions will have an undefined value when evaluated.

OCL provides a predefined operation, `isUndefined`, that returns true if the argument is undefined, and false otherwise.

An expression is undefined if any sub-expression is undefined, except for the following: (1) True or undefined = True; (2) False and undefined = False; (3) False implies undefined = True.
Association Cycles (5)

From Drs. Dwyer and Hatcliff

Association Cycles (7)

From Drs. Dwyer and Hatcliff
Design by Contract

A contract is a precise specification of mutual obligations and benefits between the client and the supplier of a service.

Design by contract is a method that emphasizes the precise description of interface semantics. An object is responsible for executing a service if and only if certain conditions are fulfilled.
An Example Contract

Client (Traveler)

- **Obligation**
  - check in 10 minutes before boarding
  - < 3 small carry-ons
  - buy ticket
- **Benefit**
  - reach Boston

Supplier (Airline)

- **Obligation**
  - take traveler to Boston
- **Benefit**
  - don’t need to wait for late travelers
  - don’t need to store arbitrary amounts of luggage

Why Contract?

- **Facilitates design**
  - establishes the responsibilities of each object clearly and precisely.

- **Simplifies implementation**
  - allows certain assumptions to be made.

- **Helps debugging**
  - clearly identifies the responsible party in case of a failure

- **Enables automatic reasoning**
  - tools can be developed to automatic checking the conformance of a contract
**Contract in OCL**

In OCL, a **contract** is specified using **pre-/post-conditions**.

```
Flight
  takeoff: Time
  ticketed: Set (Person)
  destination: Destination
```

```
Destination takeFlight (Person p, Luggage l)
pre: System::currentTime () < takeoff - 10
pre: l.number < 3
pre: ticketed -> includes (p)
post: result = destination
```

---

**Building Models with OCL**

- Introduction
- Completing UML Diagrams
- Modeling Tips and Hints
- Summary
**Navigation**

- Avoid complex navigation expressions
  - makes them easier to read and understand
  - is consistent with the encapsulation principle

  ```
  context Membership
  inv: programs.partners.deliveredServices ->
  forAll (pointsEarned = 0) implies account -> isEmpty()
  ```

  ```
  context LoyaltyProgram
def: isSaving : Boolean =
  partners.deliveredServices -> forAll (pointsEarned = 0)
  ```

  ```
  context Membership
  inv: programs.isSaving implies account -> isEmpty()
  ```

---

**Choose Context Wisely (1)**

- If the invariant restricts the value of an attribute of one class, the class containing the attribute is a clear candidate.

- If the invariant restricts the values of attributes of more than one class, the class containing any of the attributes are candidates.

- If a class can be appointed the responsibility for maintaining the constraint, the class should be the context.

- Any invariant should navigate through the smallest possible number of associations.
Choose Context Wisely (2)

Consider how to write the constraint that two persons who are married to each other are not allowed to work at the same company.

Split and Constraints

- A large invariant can often be split into smaller ones at the and operations
  - easier to read and write
  - localizes the problem in case of a broken invariant
  - easier to maintain

```plaintext
context LoyaltyProgram
inv: participants -> forAll (age() > 0)
   and
   partners.deliveredServices -> forAll (pointsEarned = 0)

context LoyaltyProgram
inv: participants -> forAll (age() > 0)
inv: partners.deliveredServices -> forAll (pointsEarned = 0)
```
Use the \textit{collect} shorthand

- Use the \textit{collect} shorthand whenever possible
  - the shorthand is easy to read and understand

\begin{verbatim}
context Person
inv: parents -> collect (brothers) -> collect (children) -> notEmpty ()
\end{verbatim}

\begin{verbatim}
context Person
inv: parents.brothers.children -> notEmpty ()
\end{verbatim}

Always Name Association Ends

- Naming association ends is a good practice
  - indicates the purpose of the associated object
  - suggests the attribute name to maintain the association
Building Models with OCL

- Introduction
- Completing UML Diagrams
- Modeling Tips and Hints
- Summary

Summary

- A model allows one to focus on high level design decisions, rather than low level details.
- One way to identify constraints is to look for optional/dynamic multiplicities and cycles.
- OCL expressions can also be used to simplify a class diagram.
- Design by contract allows the interface of an object to be precisely specified.
- The rule of thumb for writing a good OCL spec is to keep the expressions as simple as possible.