Assertions, Constraints and OCL

Mechanisms and Methodology

Department of Computer Science,
The University of York, Heslington, York YO10 5DD, England
http://www.cs.york.ac.uk/~paige

Context of this work

- The present courseware has been elaborated in the context of the MODELWARE European IST FP6 project (http://www.modelware-ist.org/).
- Co-funded by the European Commission, the MODELWARE project involves 19 partners from 8 European countries. MODELWARE aims to improve software productivity by capitalizing on techniques known as Model-Driven Development (MDD).
- To achieve the goal of large-scale adoption of these MDD techniques, MODELWARE promotes the idea of a collaborative development of courseware dedicated to this domain.
- The MDD courseware provided here with the status of open source software is produced under the EPL 1.0 license.

Assertions

- An **assertion** is a boolean expression or predicate that evaluates to true or false in every state.
- In the context of a program, assertions express constraints on program state that must be true at a specified point during execution.
  - In a model/diagram, they document what must be true of an implementation of the modelling element.
- Assertions are typically associated with methods, classes, and even individual program statements.
- They are useful for:
  - helping to write correct software, since they specify what is expected behaviour.
  - documentation of interfaces, and for debugging.
  - to improve fault tolerance.

Pre- and Postconditions

- Assertions associated with methods of a class.
- Precondition: properties that must be true when the method is called.
- Postcondition: properties that must be true when the method returns safely.
- Can both be optional (true).
  - An optional precondition: "there are no constraints on calling this method".
  - An optional postcondition: "the method body can do anything (but it must terminate)".

Example: Assertions for a Stack

```
public class IntStack {
        private int[] contents;
        public int capacity, count;

        //**
           * @pre not empty();
        */
        public int top();

        public boolean empty(){
          ...
        }

        public boolean full(){
          ...
        }

...
```

Stack Assertions, Continued...

```
    //**
       * @pre not empty();
       * @post count==count@pre-1;
    */
    public void pop();

    //**
       * @pre not full();
       * @post top()==x;
       * @post count==count@pre+1;
       * @post contents[count]==x;
    */
  public void push(int x);
}
```

Assertions in Java

- Java (1.4+) supports very primitive assertions.
- The Java iContract package is a preprocessor that provides very substantial support.
  - `www.reliable-systems.com`
- Supports three main types of assertions:
  - **@pre, @post, @invariant**
- **@pre** expr: specifies preconditions
- **@post** expr: specifies postconditions. Can refer to value of an expression when method was called.
- expr**@pre** refers to the value of expr when a method was called (can only be used in postconditions)
- **@invariant** (more on this soon)

Design by Contract

- A pre- and postcondition should be viewed as a contract that binds a method and its callers.

"If you promise to call me with the precondition **satisfied, then I guarantee to deliver a final state in which the** postcondition **holds."**

- The caller thus knows nothing about how the final state is produced, only that they can depend on delivery.
- The method and its implementer need only worry about cases where the precondition is true (and none other).

Obligations and Gains

	Obligation	Gain
CLIENT	Call Push(x) when the Stack isn't Full()	Gets x added to the top of the Stack
CLASS AUTHOR	Make sure that x is on the top() of the Stack	Need not treat the case when the Stack is full().

What if the Precondition Fails?

- That is, what should happen if a client foolishly calls a method when the precondition is false?

  A precondition violation is the client's fault!

- Thus, a false precondition is the result of an error in the caller.
  - Extremely helpful in tracking down errors.
  - Failed precondition -> ignore supplier code.
- Formally, the contract says nothing about what should be done if the precondition fails, so any behaviour (infinite loop, exception handler, return error message) is acceptable.

Class Invariants

- Classes may have global properties, preserved by "all" methods.
- This is captured in the **class invariant**.

```
class IntStack {
        private int[] contents;
        public int count, capacity;

        /**
          * @invariant count >= 0;
          * @invariant count <= capacity;
          * @invariant capacity==contents.capacity;
          * @invariant empty()==(count==0);
          * @invariant count>0 implies contents[count]==top();
        */
      }
```

Who Must Satisfy the Invariant?

- Despite its name, the invariant need not always be true.
- The constructors must leave the object in a state satisfying the invariant; it cannot assume the invariant.
- Any legal call made by a client must start from a state satisfying the invariant, and must end up in such a state.
- "Private" methods can do what they like.
  - But if they terminate in a state where the invariant is false, clients cannot use the object!
  - This is sometimes too inflexible.
  - Ongoing work at Microsoft Research is suggesting that explicit permission for various forms of "rollback" are needed for different methods or clients.

Example - Bank Account

```
/**
  * @invariant balance >= 0;
  */
class Bank_Account {
     public int balance;
     public Bank_Account() { balance=0; }

     /**
       * @pre x>0;
       * @post balance==balance@pre+x;
      */
     public void deposit(int x);

     /**
       * @pre x<= balance;
       * @post balance==balance@pre-x;
      */
     void withdraw(int x);
   }
```

Assertions at Run-Time

- The effect of an assertion at run-time should be under the control of the developer.
  - Checking assertions takes time (esp. invariants and postconditions).
- For testing and debugging, checking all assertions is very important.
- For production releases, we may want to turn assertion checking off
- iContract offers no fine-grained control.
- iContract plus iControl gives complete control over which assertions are checked (without modifying source code).
- iControl: `icplus.sourceforge.net/iControl.html`

Inheritance and Assertions

- Assume that we want to build a graphics library containing points, segments, vectors, circles, polygons, triangles, etc.
- Use an inheritance hierarchy.
  - At some point in the hierarchy there will be the general notion of a Polygon.
  - Specialized polygons will descend from it: rectangle, triangle, etc.
- Polygon class may introduce assertions.
- What happens to these assertions under inheritance?

General Polygon Class

```
/**
       * @invariant count==vertices.size();
       * @invariant count>=3;
       * @invariant forall Object v in vertices |
       *               (v instanceof Point);
*/
     class Polygon {
        ...
        public int count;
        public double perimeter() { ... }
        public void display() { ... }
        public void rotate(Point c,double angle){...}
        public void translate(double a,b){...}

        private LinkedList vertices;
     }
```

Assertions are Inherited

- In general, assertions are inherited with the features or class to which they apply.

```
/**
  * @invariant count==4;
  */
  class Rectangle extends Polygon {
    public double side1,side2,diagonal;
    public double perimeter(){...}
  }
```

- The invariant of Polygon is and-ed to that of Rectangle.
- **Question:** what if `count==2` was added to the invariant instead of `count==4`?

Assertions and Overriding

- Child classes can override method implementations from a parent.
  - But what if the parent method has a contract (that is inherited)?
- Two alternatives:

1. **Replace inherited implementation but keep the contract.**
   - e.g., provide a more efficient implementation
   - e.g., provide a new implementation that does **more** than the original.
2. **Modify the contract and the implementation.**
   - ... because the contract may not say exactly what you want.
   - complications will arise with substitution (example).

Simple Example

```
// in Polygon, inherited by Rectangle
/** @pre d>0 */
public void rotate(double d);

Polygon p; Rectangle r;
if(d>0) p.rotate(d);  // Polygon's rotate
p=r;
if(d>0) p.rotate(d);  // Rectangle's rotate
                       // Check Polygon's pre
```

➢ Even though **p** is a Rectangle, we must check the precondition of *Polygon*'s version of rotate.

Assertions and Overriding

- Contracts cannot be invalidated or broken by overriding, otherwise clients cannot rely on the methods' results.
- Thus, the following change would be illegal.

```
// in parent
/**
  * @pre x>=0;
  * @post return>=0;
 */
double sqrt(double x);
```

```
// in child
/**
  * @pre x>=0;
        * @post return<=0;
 */
double sqrt(double x);
```

➢ Thus, you must do at *least* what the original contract said, but you can also do more.

Assertions and Overriding Rules

1. The inherited contract cannot be broken.
2. The inherited contract may be kept unchanged; this guarantees that it is not broken.
   - **Usage**: changing implementation details (method body).
3. The **@pre** may be replaced by a weaker one.
4. The **@post** may be replaced by a stronger one.

- Rules 3/4 imply that if you want to change the contract under inheritance, you can replace it with a subcontract: every behaviour of the new contract satisfies the original.
- Example.

Class IntSet

```
public void addZero()
/** @pre this is not empty
  * @post add 0 to this
  */
```

- In a child of IntSet, e.g., Child_IntSet

```
public void addZero()
/** @pre true
  * @post add 0 to this and add 1 to the
          number_of_elements
  */
```

Why This Works

```
IntSet i = new IntSet(…);
Child_IntSet c = new …;
i = c;
if(                ) { c.addZero(); }
if(                ) { i.addZero(); }
```

- Checking the precondition for IntSet's version is sufficient to guarantee that the precondition for the child is also true.
- What should go into the conditions for the ifs?

Example - Matrix Inversion Routine

```
/** Version in parent class
  * @pre epsilon>= pow(10.0,-6);
  * @post det(sub(mult(this,inverse),identity))<=epsilon;
  */
double invert(double epsilon);

 ===================================================

/** Version in subclass; weaker @pre, stronger @post.
  * @pre epsilon >= pow(10.0,-20.0);
  * @post det(sub(mult(this,inverse),identity))<=epsilon/2;
  */
double invert(double epsilon);
```

Using Subcontracting Efficiently

- A compiler has to check that preconditions are weakened, postconditions strengthened.
- Inefficient for real programs
- Efficient implementation: use a low-tech language convention based on the observations that for assertions α, β:
  - α **implies** (α **or** γ)
  - (β **and** γ) **implies** β
- i.e., accept weaker preconditions only in form (α **or** γ)
- Accept stronger postconditions only in form (β **and** γ)

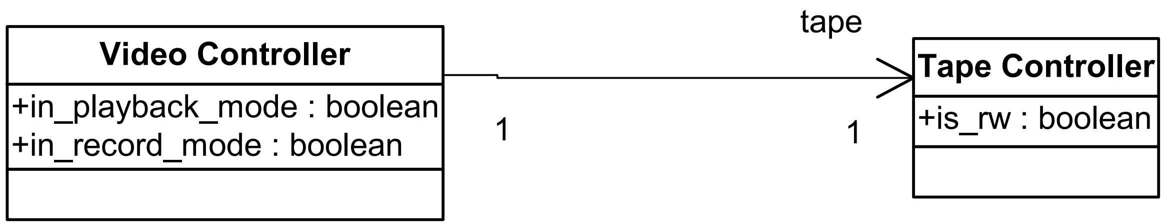
Language Support

- In iContract-annotated Java this convention is implemented as follows:
  - In overridden method, only specify the **new** clauses.
  - New **@pre** clauses are automatically **or**-ed with any inherited precondition clauses.
  - New **@post** clauses are automatically **and**-ed with any inherited postcondition clauses.
- Overridden contract is automatically a subcontract, and no theorem proving needs to be done.

Design by Contract in UML

- UML is primarily a graphical notation.
  - Uses text for labels and names.
  - Text is also used for writing constraints.
    - **Constraint** = a restriction on state values.
    - Types are a form of constraint.
  - Constraints will be how we support design by contract in UML.
- Constraints are also useful for resolving limitations with UML's expressiveness.

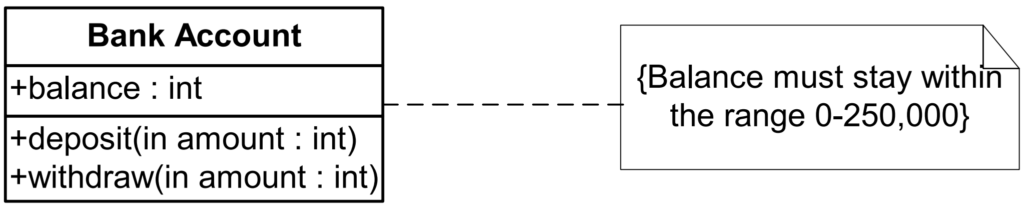
A Motivation for Constraints



- The Video Controller cannot be simultaneously in playback mode and record mode.
- If it is in playback mode, the Tape Controller is in read/write mode.
- Cannot capture this easily and formally in pure UML.

Notes and Informal Constraints

- UML supports an informal notion of constraint, called a **note**: any piece of information.
- A note is a string attached to a model element.



➢ There are no restrictions on what can go in a note.

➢ To write formal (machine-checkable) constraints, use the standard constraint language OCL.

OCL (Object Constraint Language)

- The constraint/assertion language for UML.
- Used for writing general constraints on models, and also for design-by-contract.
- Can be applied to any modelling element, not just classes.
- Issues with OCL:
  - programming language-like syntax (similar to C++).
  - "easy to use" by non-formalists.
  - cumbersome in places, can be difficult to support (type check, model check) using tools.
- Used widely in the UML metamodel.

OCL - Essential Capabilities

1. Specifying which model element is to be constrained (the **context**).
2. Navigating through models to identify objects that are relevant to a constraint (**navigation expressions**).
   - Can constrain values of attributes and related objects.
3. Asserting properties about relationships between objects (**expressions**).

OCL Context

- OCL constraints are usually written textually, separate from the graphical UML model.
- Consider a Bank Account class with an integer **`balance`**.
- OCL constraint:

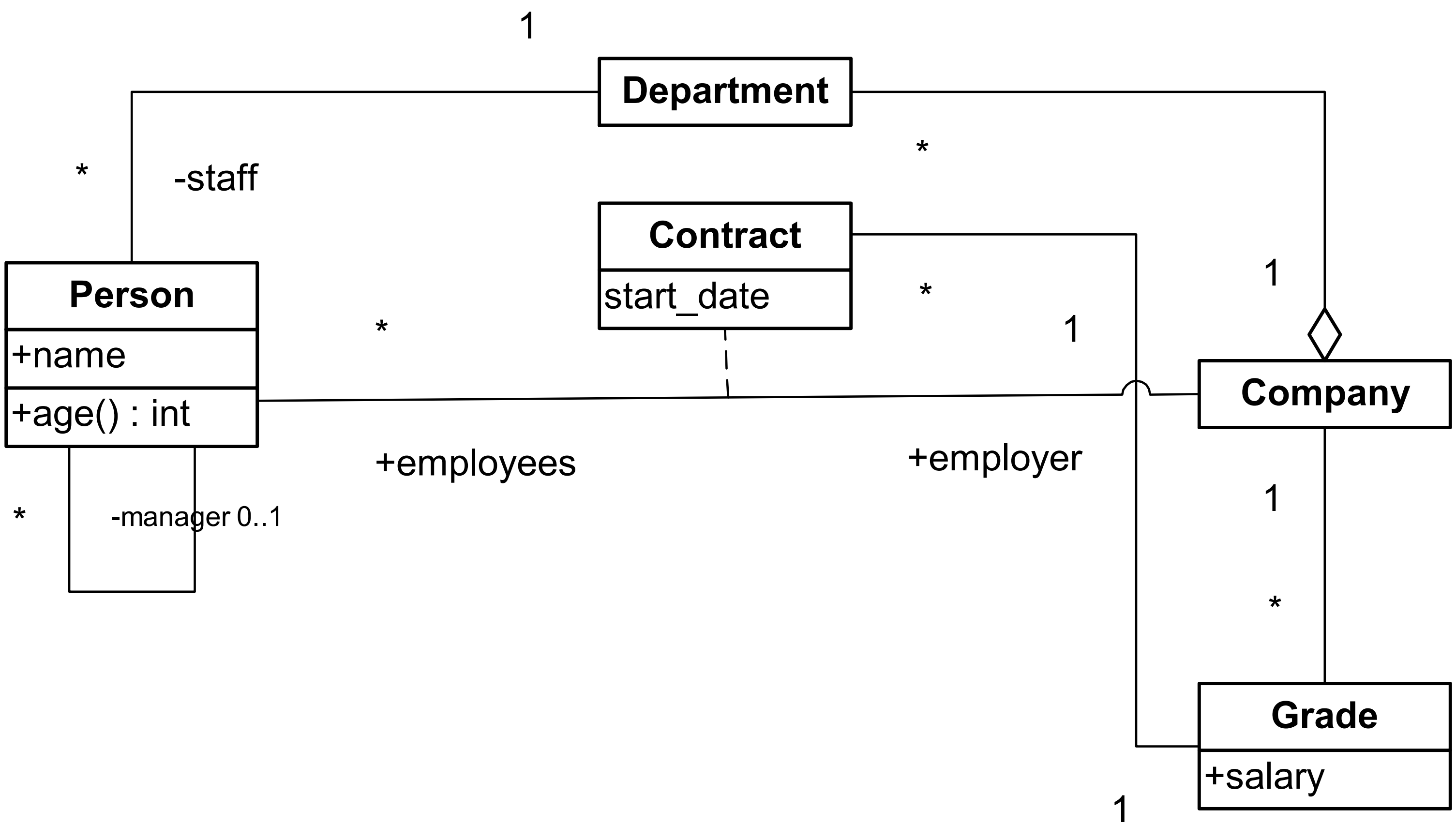
```
context Bank Account inv:
self.balance>=0 and self.balance<=250,000
```

- The constraint applies in the context of the class Bank Account.

Navigation Expressions

- OCL provides the means for referring to objects that are linked to a specified context object.
- Linked objects are obtained starting from the context object.
  - Links are followed to gain access to other objects of interest (similar to references in Java/C++/Eiffel).
  - Complexity arises when collections (e.g., sets, sequences, bags, etc.) are navigated.
- Personnel system running example...

Example - Personnel System



Following Links

- Basic form of navigation involves following links from one object to another.
- Specify by giving names of associations to be traversed.
- Denote set of employees working in department.

```
context Department
self.staff
```

- If association has no role name, use name of class at far end of association (unless it is ambiguous)

```
context Company
self.department
```

Names and Constraints

- An OCL invariant need not be applicable to any instance (i.e., self).

```
context bart : Person inv:
bart.age == 10
```

- An OCL invariant may optionally be given a name; the name can be used by simulators, debuggers, and in documentation.

```
context bart : Person inv how_old:
bart.age == 10
```

Basic Values and Types

- OCL supports a number of basic types and typical operations upon them.
  - e.g., Boolean, Integer, Real, String
- Collection, Set, Bag, Sequence, and Tuple are basic types as well.
- Iterative operations and operators will be defined shortly for making use of these.
- Every class appearing in a UML model can also be used as an OCL type.
- Type conformance rules are as in UML, i.e.,
  - types conform to supertypes
  - Set, Bag, and Sequence conform to Collection

Predefined Operations

- Available on all objects:
  - **oclIsTypeOf(t:OclType):** true if self and t are the same
  - **oclInState(s:OclState):** true if self is in the state specified by s. s is the name of some state in the statechart for the class.
  - **oclIsNew():** true if used in a postcondition and the object is created by the operation.
  - casting (next slide)

Casting

- For better or for worse, casting (re-typing) is permitted in OCL.
- An object may be cast to one of its subtypes if it is known that the object's dynamic type is that of a subtype.
- Use the oclAsType(OclType) operation.
- Example:

```
homer.oclAsType(Employee)
```

treats `homer` as an Employee, not a Person.

Three-Valued Logic

- The logic for OCL is actually 3-valued.
- An expression can evaluate to true, false, or undefined.
- Examples:
  - an illegal cast returns undefined.
  - taking the first() element of an empty sequence is undefined.
- How does this affect the truth tables for expressions?
- An expression is undefined if one of its arguments is undefined, except:
  - true OR anything is true
  - false AND anything is false
  - false IMPLIES anything is true.

Collections

- A navigation expression denotes the objects retrieved by following links.
- Depending on multiplicities of associations, the number of objects retrieved may vary.

```
context Person
self.department
```

- This retrieves one object.
- A navigation expression that can return more than one object returns a **collection** (a set or a bag).
  - Iterated traversals produce bags; single traversals produce sets.
- Collections may be nested (earlier versions of OCL flattened all collections, e.g., OCL 1.x)

Iterated Traversal

- Navigations can be composed; more complicated paths through a diagram can be denoted
- All the people who work for a company

```
context Company
self.department.staff
```

- Evaluate in a step-by-step manner:
  - `self.department` gives a set of departments.
  - `staff` is then applied to each element of the set, producing a bag of people.

Operations on Objects and Collections

- Operations and attributes defined for classes in a model can be used in OCL expressions, e.g.,

```
context Person
self.age()
self.contract.grade.salary
```

- Collections come with some built-in operations, accessed using ->, typically producing bags.
  - sum(): applicable to numeric collections

    ```
    context Department inv:
    self.staff.contract.grade.salary->sum()
    ```

  - asSet(): converts bags to sets, removing duplicates

    ```
    self.staff.contract.grade->asSet()->size()
    ```

'Select' on Collections

- A built-in operation for picking out specific objects from larger collections (a quantifier).
- Example: all employees with a salary greater than £50,000.

```
context Company inv:
employees->select(p:Person|p.contract.grade.salary>50000)
```

- Can define further navigations on the result of the `select`.
- Think of OCL quantifiers as iterators over data structures (ie., operationally).

'Collect' on Collections

- The collect operation takes an expression and returns a bag containing all values of expression.

```
context Department inv:
staff->collect(p:Person | p.age())
```

- Returns ages of all employees in a department.
- Avoid dropping name and type of bound variables - can easily lead to ambiguity.
  - OCL guide is careless on this!

Basic Constraints

- OCL can be used to write arbitrarily complex constraints.
- Some invariant examples in context **`Person`**
  - `self.employer = self.department.company`
  - `employer.grade->includes(contract.grade)`
  - `age()>50 implies contract.grade.salary>25000`
- Some invariant examples in context **`Company`**
  - `employees->select(age()<18)->isEmpty()`
  - `grade->forAll(g:Grade|not g.contract->isEmpty())`
- Correspond to class invariants.

Iterative Constraints

- Select is an iterative constraint defined on a collection.
- Others include:
  - **`forAll`**: return true if every member of collection satisfies the boolean expression
  - **`exists`**: true if there is one member of the collection satisfying the boolean expression
  - **`allInstances`**: returns all instances of a type, e.g.,

```
Grade.allInstances->forAll(g:Grade | g.salary>20000)
```

- Use allInstances very carefully! It's not always necessary to use it (e.g., as in above example), and it is often difficult to implement it.

Using 'forAll' and 'exists'

- At least one employee appointed at every grade in the company.

```
context Company inv:
grade->forAll(g:Grade|not g.contract->isEmpty())
```

- Every department has a head (i.e., an employee with no manager).

```
context Department inv:
staff->exists(e:Employee | e.manager->isEmpty())
```

Using 'allInstances'

- It is not necessary to use allInstances in the previous example.
  - The constraint expresses that salary must be at least 20000 in all grades. So put the constraint with the appropriate class!

```
context Grade inv:
self.salary > 20000
```

- When might allInstances be useful?
  - Specifying a change in the objects known to a system (e.g., the result of new or malloc, or the addition of a new object from outside the system).

Pre/Postconditions

- Preconditions are standard OCL constraints that can refer to arguments and the prestate of an operation.
- Postconditions can refer to arguments, the prestate, the result of a function, and the poststate of an operation.
  - **@pre**: value of an expression evaluated in a prestate
  - **result**: the value returned by a function call
- Example:

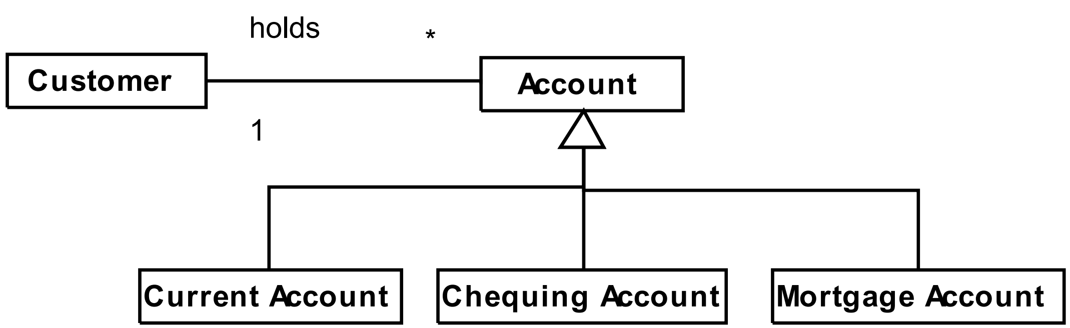
```
context Savings_Account::withdraw(int amount)
pre: amount <= balance
post: balance = balance@pre-amount
```

Design by Contract

- The usual laws of design by contract should be followed with UML and OCL:
  - preconditions may be **weakened** ('or' new clauses)
  - postconditions may be **strengthened** ('and' new clauses)
  - new clauses may be **and-ed** to the invariant
- OCL does not require that you follow these rules, but it is strongly recommended.
- Modified contracts must maintain consistency!

Generalization and OCL

- Generalization relationships are not navigable - they do not usually feature in writing constraints.
- But generalizations may be constrained.
- **Example**: a Customer must hold at least one Current account.



Messages and Signals

- OCL 2.0 now supports messages.
- Messages can be sent to objects, and correspond either to operation calls or signals in a UML model.
- Signals are the simplest form of inter-object communication in UML.
- Think of a signal as a class (with attributes, operations, etc) that represents communication.
- A signal will be instantiated (with values for attributes) and can generate state transitions in its recipient.
- In a UML model, a signal is drawn as a class with a <<signal>> stereotype.
- Examples: InputEvent, MouseButtonUp, MouseButtonDown, etc.

Sending Signals

- To specify that communication has taken place, use the hasSent operator ^.
- Example:

```
context Subject::hasChanged()
post: observer^update(12,14)
```

- The hasSent results in true if an update message with the specified arguments was sent to observer during execution of the operation.
- Arguments can be omitted (replace with ?)