Unified Model Language (UML)

Dr. Jeff Huang

Adapted from materials developed by Dr. Charles Zhang (HKUST)
Unified Model Language (UML)

- The UML is a graphical language for
  - specifying
  - visualizing
  - constructing
  - documenting
  the artifacts of software systems
- Created by unifying the Booch, OMT, and Objectory modeling languages
- Specified and managed by OMG in November 1997 as UML 1.1
- Most recent revision is UML 2.5
UML 2.0: Complete specification

- **Structure diagrams.**
  - 1. Class diagram
  - 2. Composite structure diagram
  - 3. Component diagram
  - 4. Deployment diagram
  - 5. Object diagram
  - 6. Package diagram + Profile diagram
  - 7. Use-case diagram

- **Behavior diagrams**
  - 8. State machine diagram
  - 9. Activity diagram
  - 10. Sequence diagram
  - 11. Communication diagram
  - 12. Interaction overview diagram
  - 13. Timing diagram
UML: Structural modeling

- Structural model: a view of a system that emphasizes the structure of the objects, including their classifiers, relationships, attributes and operations.
- Show the static structure of the model
  - the entities that exist (e.g., classes, interfaces, components, nodes)
  - internal structure
  - relationship to other entities
- Do not show
  - temporal information
- Kinds
  - static structural diagrams
    - class diagram
    - object diagram
  - implementation diagrams
    - component diagram
    - deployment diagram
A class describes a group of objects having common:

- semantics
- state
- behaviour
- relationships

A class provides a template to create objects (i.e., it is a “factory” for objects).

⇒ In the UML a class is a classifier; an object is an instance.

A good class should capture one and only one abstraction.

⇒ It should have one major theme.

A class should be named using the vocabulary of the problem domain.

⇒ So that it is meaningful and traceable from the real world to a model.
An attribute describes the data values held by objects in a class.

- Each attribute has a:
  - *name*: unique within a class, but not across classes
  - *type*: specifies the domain of values — string, integer, money, etc.
    - continuous (e.g., salary, name)
    - discrete (e.g., sex, colour)
  - *visibility*: specifies who can access the attribute’s values
    - public (+)
    - private (–)
    - protected (#)
    - package (~)
  - *initial value* [optional]: specifies the attribute’s initial value
  - *multiplicity* [optional]: specifies the number of simultaneous values
  - *changeability*: which specifies whether the value can be changed
    - no constraint (default)
    - frozen
    - addOnly

- An attribute can be either a **base attribute** or a **derived attribute**
  - e.g., birthdate versus age
ATTRIBUTES: UML TEXTUAL NOTATION

«stereotype» visibility name [multiplicity]: typeExpression = initialValue
{propertyString}

propertyString → a comma separated list of properties or constraints

➤ Only “name” is mandatory.

Examples

+ size: area = (100,000) {frozen}
name: string
telephone[0..2]: string
telephone[1, 3..4]: string
– salary: money {>0, <1,000,000}
An operation is a function or transformation that may be applied to or by objects in a class.

Company → hire, fire, pay-dividends, …
Course → register, waive-prerequisite, …

- An operation is invoked through the object’s interface and has the following properties:
  - **operation signature:** name of the operation (called the selector) names and types of the arguments type of the result value
  - **visibility:** public (+), private (−), protected (#), package (~)

- An operation is said to have side effects if its execution changes the state of an object (a query operation has no side effects).
CLASS: METHOD

A method is the implementation (code) of an operation.

- **An operation** is visible in the interface; the method is hidden.
  - In the UML an operation is a classifier; a method is an instance.

- **Polymorphic operation** – An operation that can have several different methods.
  
<table>
<thead>
<tr>
<th>class</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>payInterest</td>
</tr>
<tr>
<td>Savings</td>
<td>calculate interest on savings accounts</td>
</tr>
<tr>
<td>Checking</td>
<td>calculate interest on checking accounts</td>
</tr>
</tbody>
</table>
  
  ➔ Useful for overriding methods in subclasses (see generalization).

- **Dynamic binding** – Choosing the method to execute for an operation based on the object’s class.
**OPERATION: UML TEXTUAL NOTATION**

```
«stereotype» visibility name (parameterList): returnType {propertyString}
```

- **parameterList** → kind name: typeExpression = defaultValue
  - **kind** → in - pass by value
  - out - pass by reference (no input value; output value only)
  - inout - pass by reference (input and output value)

- **propertyString** → a comma separated list of properties or constraints
  - **isQuery** → true or false  (if true → no side effects)
  - **isPolymorphic** → true or false
  - **concurrency** → sequential - callers must coordinate to ensure only one call to an object may execute at one time
  - guarded - multiple calls to an object may occur simultaneously, but only one is allowed to execute at a time; other calls are blocked
  - concurrent - multiple calls may occur simultaneously to an object and all execute concurrently

- **operation signature** → name (parameterList): returnType  (mandatory)
### Class: `UML Graphical Notation`

The UML textual notation for attributes and operations is used within the compartments of a class.

<table>
<thead>
<tr>
<th>Class name</th>
<th>BankAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute compartment</td>
<td>–accoutNumber : int +amount : money –count : int = 0</td>
</tr>
<tr>
<td>operation compartment</td>
<td>+create(aNumber : int) –incrementCount( ) +getCount( ) : int +getNumber( ) : int +balance( ) : money +deposit(amount) +withdraw(amount) –payInterest( )</td>
</tr>
<tr>
<td>extra compartment</td>
<td>The UML textual notation for attributes and operations is used within the compartments of a class.</td>
</tr>
</tbody>
</table>

- **Visibility**: 
  - + public
  - – private
  - # protected
  - ~ package
A **stereotype** is a new class of modeling element, which is a subclass of an existing modeling element.

- Allows the object model to be **dynamically extended**.

**Example:** We can define different kinds of classes that are useful for modeling.

- **PenTracker**
- **OrderForm**
- **BankAccount**

**Special icons** can be used for each stereotype that improve modeling clarity.
An association describes a group of links with common semantics.

- An association is a classifier; a link is an instance.
- Conceptually, associations are inherently bi-directional.
There can be several associations between the same two classes.

Or even with the same class.
ASSOCIATION: DEGREE

- **unary** (reflexive) relates a class to itself
  
- **binary** relates two classes
  
- **ternary** relates three classes

- **n-ary** relates any number of classes

In practice, the vast **majority of associations are binary**!
Generalization

Separate Target Style

Shared Target Style
Interfaces

String

isEqual(String):Boolean
hash():Integer

Comparable

HashTable

Hashable

Comparable «interface»

isEqual(String):Boolean
hash():Integer

«use»
Composition

Fig. 3-36, *UML Notation Guide*
Notice that the links are directional. Point does not know the existence of Polygon.
## Structural Modeling:
### Core Elements

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>a description of a set of objects that share the same attributes, operations, methods, relationships and semantics.</td>
<td></td>
</tr>
<tr>
<td>interface</td>
<td>a named set of operations that characterize the behavior of an element.</td>
<td><img src="image" alt="interface" /></td>
</tr>
<tr>
<td>component</td>
<td>a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces.</td>
<td><img src="image" alt="component" /></td>
</tr>
<tr>
<td>node</td>
<td>a run-time physical object that represents a computational resource.</td>
<td><img src="image" alt="node" /></td>
</tr>
<tr>
<td>constraints</td>
<td>a semantic condition or restriction</td>
<td><img src="image" alt="constraint" /></td>
</tr>
</tbody>
</table>
## Structural Modeling:
### Core Relationships

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>association</td>
<td>a relationship between two or more classifiers that involves connections among their instances.</td>
<td></td>
</tr>
<tr>
<td>aggregation</td>
<td>A special form of association that specifies a whole-part relationship between the aggregate (whole) and the component part.</td>
<td></td>
</tr>
<tr>
<td>generalization</td>
<td>a taxonomic relationship between a more general and a more specific element.</td>
<td></td>
</tr>
<tr>
<td>dependency</td>
<td>a relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).</td>
<td></td>
</tr>
<tr>
<td>Realization</td>
<td>a relationship between a specification and its implementation.</td>
<td></td>
</tr>
</tbody>
</table>
Reverse Engineering Example

class Clutch {
    Gear [] gears;
    GearBox box;
    public void calibrate (Calibrator c) {
        ....
    }
}
**ASSOCIATION: MULTIPLICITY**

*Multiplicity* specifies a restriction on the number of objects in a class that may be related to an object in another class.

For a given instructor, how many courses can he teach?

- An instructor does not have to teach any course but may teach more than one course in a given semester.

For a given course, how many instructors can teach it?

- Each course must be taught by one instructor but may have many.

Multiplicity is a *real-world constraint*!
ASSOCIATION: MULTIPLICITY (cont’d)

**minimum cardinality (min-card)**

min-card(C₁,A): the minimum number of links in which each object of C₁ can participate in association A

\[
\text{min-card}(C_1, A) = 0 \rightarrow \text{optional participation (may not be related)} \\
\text{min-card}(C_1, A) > 0 \rightarrow \text{mandatory participation (must be related)}
\]

**maximum cardinality (max-card)**

max-card(C₁,A): the maximum number of links in which each object of C₁ can participate in association A
special cardinalities:

- $\text{max-card} = * \rightarrow \text{an unlimited upper bound (}\infty\text{)}$
- $\text{min-card} = 1$ and $\text{max-card} = 1 \rightarrow \text{can use 1 by itself}$
- $\text{min-card} = 0$ and $\text{max-card} = * \rightarrow \text{can use * by itself}$
max-card(C1,A) = 1 and max-card(C2,A) = 1 ➞ one-to-one association (1:1)

Country 0..1 CapitalOf 1..1 City

Country
- China
- Canada
- USA

CapitalOf

City
- Beijing
- Ottawa
- New York
- Washington
max-card(C1,A) = 1 and max-card(C2,A) = *

➨ one-to-many association (1:N)

Employee 0..* AssignedTo 1..1 Department

Employee: John, James, Alan, Bill, Larry

Department: Sales, Production, Marketing
ASSOCIATION: MULTIPLICITY (cont’d)

max-card(C1,A) = * and max-card(C2,A) = *

many to many association (N:M)

![Diagram showing a many-to-many association between Supplier, Supplies, and Part with specific instances of companies and parts.]

- Supplier: E-com, Vtech, SGH, Mach, Hi-tech, Lansing
- Supplies: 85213, 28175, 33472, 90231
- Part: 85213, 28175, 33472, 90231
Given a ternary association among classes (A, B, C), the multiplicity of the C end states how many C objects may appear in association with a particular pair of (A, B) objects.

- A student will not take the same course from more than one professor, but a student may take more than one course from one professor and a professor may teach more than one course.
- For each (professor, course) \(\rightarrow\) many students
Behavior modeling

- Sequence diagram
- State machine diagram
UML state machine diagram

- The output behavior is not only a direct consequence of the current input, but of some past history of its inputs.
- Characterized by an internal state which represents this past experience.
- Behavior description mechanism.
- Describes the behavior for:
  - System
  - Class
  - Operation
State machine concepts

- State – stores information of the system (encodes the past)
  - Particular states
    - Initial state
    - Final state

- Transition – describes a state change
  - Can be triggered by an event
  - Can be guarded by a condition

- Actions – behavior performed at a given moment
  - Transition action: action performed at transition time
  - Entry action: action performed when entering a state
  - Exit Action: action performed when exiting a state
  - Do Action: action performed while staying in a state
Simple state machine example

Start state

Off

event

action

switch/nb = nb + 1

Guarded Transition (no event)

End state

[nb > 250000]

On
Telephone: Entry/Exit actions

- Standby
  - Entry/ start dial tone
  - Exit /end dial tone

- Dialing
  - Entry/ number.append(n)
  - [ number.is_valid]

- Call

- Pick up
- Hang up
Hierarchical states diagrams

- All states are at the same level => the design does not capture the commonality that exists among states
- Solution: Hierarchical states – described by sub-state machine(s)

- Two kinds of hierarchical states:
  - And-states (the contained sub-states execute in parallel)
  - Or-states (the contained sub-states execute sequentially)
Hierarchical OR-state machine
Hierarchical AND-state machine
Sequence Diagram

- Shows a concrete execution scenario, involving: objects, actors, generic system
- Highlights the lifelines of the participating instances
- Focuses on interaction, exchanged messages and their ordering
- Can address various levels of abstraction:
  - System level
  - Object sets level
  - Object level
  - Method level
Structure of Sequence Diagram

object symbol

lifeline

activation

name : Class

other

name (...)

new (...)

: Class

stimulus

return

create

delete
Common roles of sequence diagrams

- **Actor**: Users of the system
- **Boundary**: UI or I/O interface
- **Control**: Active object
- **Entity**: Data
Message Types

Procedure call or other kind of nested flow of control

Flat flow of control

Explicit asynchronous flow of control

Return

Nested Flow

Flat Flow

Asynchronous Flow
Self-message and Recursion
Sequence diagram: example

1. Student indicates wish to enroll
2. Student inputs name and number
3. System verifies student
4. System displays seminar list
5. Students picks seminar

Copyright 2002-2006 Scott W. Ambler
A combined fragment groups sets of messages together to show conditional flow within a region of a sequence diagram.

- Some commonly used combined fragments:
  - **alt** - (alternative) models if-then-else logic.
  - **break** - models an alternative sequence of events that is processed instead of the whole of the rest of the diagram.
  - **loop** - models logic which will be potentially repeated several times.
  - **opt** - (optional) models switch logic.
  - **par** - (parallel) models concurrent processing.
EXAMPLES OF COMBINED FRAGMENTS: ALT

- An alt combined fragment has two or more subfragments.

- Exactly one of the subfragments will execute based on the guard condition, unless no guard is true; the condition [else] is true if no other guard is true.

- If more than one of the guards is true, the choice may be nondeterministic.
EXAMPLES OF COMBINED FRAGMENTS:

**BREAK**

- A **break** combined fragment has one subfragment with a guard condition.

- The subfragment is executed if the guard is true and the remainder of the enclosing interaction fragment is not executed; otherwise execution continues normally.
EXAMPLES OF COMBINED FRAGMENTS: LOOP

- A **loop** combined fragment has one subfragment with a guard, which may have a minimum and maximum count as well as a Boolean condition.

- The subfragment is executed as long as the guard condition is true, but it is executed at least the minimum count and no more than the maximum count.

- If the guard is absent, it is treated as true and the loop depends on the repetition count.
EXAMPLES OF COMBINED FRAGMENTS: OPT

- An opt combined fragment has one subfragment with a guard condition.
- The subfragment is executed if the guard is true and not executed otherwise.
- The opt construct is equivalent to an alt construct with a single guarded subfragment and an empty [else].
EXAMPLES OF COMBINED FRAGMENTS: PAR

- A parallel combined fragment has two or more subfragments that are executed concurrently.

- The execution order of individual elements in parallel subfragments may interleave in any possible order (unless prohibited by a critical construct).

- The concurrency is logical and need not be physical; the concurrent executions may be interleaved on a single execution path.
A constraint is an assertion about properties of model elements that must be satisfied by the system.

**Example** A preferential savings account whose balance always must be greater than $100,000.

```
PreferredSavings
accountNumber
amount
balance() {balance ≥ 100000}
deposit(amount)
withdraw(amount)
```

```
PreferredSavings
accountNumber
amount
balance()
deposit(amount)
withdraw(amount)
```

A constraint is a statement that can be tested (true/false) and should be enforced by the system implementation.
COMMON CONSTRAINTS ON ASSOCIATIONS

**ordering**

- Country \(\rightarrow\) Has \(\rightarrow\) President
  - \(1\) \(\rightarrow\) \(*\) (ordered)

**subset**

- Person \(\rightarrow\) Committee
  - \(1..*\) \(\rightarrow\) \(*\)
  - \(\{\text{subset}\}\)

**xor**

- Account \(\rightarrow\) HasPersonal \(\rightarrow\) Person
  - \(1..*\) \(\rightarrow\) \(1\)
  - \(\{\text{xor}\}\)
  - An account can be related to either a Person or a company, but not to both.
COMMON CONSTRAINTS: ASSOCIATION (cont’d)

ordering

- States that the set of objects at one end of an association is in an explicit order (normally, they are unordered).

subset

- States that the set of links which are instances of one association must be a subset of those of another association.

xor

- States that *instances of the common class* can participate in only one of the constrained associations at any given time.
Object Constraint Language (OCL)

- Aims to fill the gap between mathematical rigor and business modeling
- Used to specify:
  - Invariants on classes and types in the class model
  - Pre and post conditions on operations
  - Boolean guards and constraints
  - Defining initial and derived values of features
OBJECT CONSTRAINT LANGUAGE (OCL)

context [contextName] : {className inv | operationName {pre | post}}

constraintName : constraintSpecification

An optional alternative name for the constraint. A Boolean expression that evaluates to True or False.

inv accountMinimum: acc.balance ≥ 100000

A constraint specification may contain:

1. navigation expressions, which identify other objects that may be relevant to the constraint.

2. assertions about the context object or the relationships between the context object and any objects retrieved by the navigation expressions.
OBJECT CONSTRAINT LANGUAGE
(OCL)

Example Class Diagram
Example Objects
OCL: NAVIGATION EXPRESSIONS

Starting from a context object and, using dot notation and class/role names, we navigate (possibly using links) to a:

- **local attribute** *(does not navigate any association)*.
  
  *Example:* The salary for every post is greater than $10,000.
  
  context Post
  
  inv: salary > 10000 (or self.salary > 10000)

- **directly related class** *(navigates a single association)*.

- **indirectly related class** *(navigates a series of associations)*.
  
  *Example:* The department an employee works for should be part of the same division the employee works for.
  
  context Person
  
  inv: employer = Department.Division
  
  directly related class
  
  indirectly related class
### OCL: SOME STANDARD OPERATIONS

#### Boolean operations
- **or**: $a \lor b$
- **and**: $a \land b$
- **exclusive or**: $a \oplus b$
- **negation**: $\neg a$
- **equals**: $a == b$
- **not equals**: $a \neq b$
- **implies**: $a \implies b$

#### String operations
- **concatenation**: `string.concat(string)`
- **size**: `string.size()`
- **to lower case**: `string.toLowerCase()`
- **to upper case**: `string.toUpperCase()`
- **substring**: `string.substring(int, int)`
- **equals**: `string1 == string2`
- **not equals**: `string1 != string2`

#### Integer and real operations
- **equals**: $a == b$
- **not equals**: $a \neq b$
- **less**: $a < b$
- **more**: $a > b$
- **less or equal**: $a \leq b$
- **more or equal**: $a \geq b$
- **plus**: $a + b$
- **minus**: $a - b$
- **multiplication**: $a \times b$
- **division**: $a / b$
- **modulus**: $a \mod b$
- **integer division**: $a \div b$
- **absolute value**: $a.abs()$
- **maximum of $a$ and $b$**: $a.max(b)$
- **minimum of $a$ and $b$**: $a.min(b)$
- **round**: $a.round()$
- **floor**: $a.floor()$
A navigation expression that returns more than one object returns a collection of which there are the following types:

- **set** → result of navigating a single association.
- **ordered set** → result of navigating a single ordered association.
- **bag** → result of navigating a series of at least two associations with one-to-many or many-to-many multiplicity (e.g., Person→Division→Post).
- **sequence** → result of navigating a series of at least two associations with one-to-many or many-to-many multiplicity and one of the associations is ordered.

**Notes:** Bags are multisets and so can contain the same object multiple times (bags can be converted to sets if necessary).

The operator “→” indicates a collection operation is being applied.
OCL: SOME COLLECTION OPERATIONS

- `allInstances` returns the collection consisting of all the instances of a type
- `asSet(collection)` returns a set containing each element of `collection`
- `collect(expression)` returns a bag consisting of the values of `expression` for each object in the original collection
- `excludes(object)/includes(object)` returns True if `object` is not/is in the collection
- `excludesAll(collection)/includesAll(collection)` returns True if the original collection does not/does contain all of the elements in `collection`
- `intersection(collection)` returns a collection that contains only elements that are part of both the original collection and `collection`
- `isEmpty/notEmpty` returns True if the collection has no/has at least one element
- `select(expression)` returns a collection containing only the elements of the original collection for which `expression` is True
- `size` returns the number of elements in the collection
- `sum` returns the total of all the elements in the collection
- `union(collection)` returns a collection containing elements from both the original collection and `collection`
The number of employees in every division must equal the number of employees in all of its departments.

**context** Division

**inv:** employee \(\rightarrow\) size() = Department.staff \(\rightarrow\) size()
Every employee’s post is one of the set of posts of that employee’s division.

**context** Person

**inv:** employer.Post → includes(Contract.Post)
OCL: EXAMPLE CONSTRAINTS (cont’d)

The staff of a department are all employees of the division the department belongs to.

context Department

department as Division

inv: Division.employee → includesAll(staff)

Person

contract as Contract

Manager

Clerk

Agent

Advertising

Sales

PartOf

Division

Marketing

Production

Post

Has

IsFor

employee

WorksIn

staff

inv: Division.employee → includesAll(staff)
OCL: EXAMPLE CONSTRAINTS (cont’d)

Every employee over the age of 50 must receive a salary of at least $25,000.

**context** Person

**inv**: age() > 50 implies Contract.Post.salary ≥ 25000
OCL: ITERATIVE CONSTRAINTS

- Support for universal and existential quantifiers
- Iterative constraints iterate over each instance of a collection testing a condition and returning either True or False.

**forAll** – Returns True if the specified Boolean expression is true for all elements in the collection.

**exists** – Returns True if the Boolean expression is true for at least one of the elements in the collection.
No two posts have the same value for their salary attribute.

context Post
inv: Post.allInstances -> forAll(g: Post | g <> self implies g.salary <> self.salary)
OCL: EXISTS ITERATIVE CONSTRAINT EXAMPLE

Every division has a head (i.e., there is an employee—the head—who does not have a manager).

context Division
inv: employee → exists(e: Person | e.manager → isEmpty())
OCL: OPERATION CONSTRAINTS

precondition

- Must be true *just before* an operation is called.
- It usually relates the *attributes of a class instance* and the *parameters of an operation*.

Example: The withdrawal amount must be less than the balance.

context SavingsAccount::withdraw(amount : Real)
    pre: amount < balance

postcondition

- Must be true *just after* an operation has completed.
- We can compare *values before* (using “@pre”) and *after* operation execution.

Example: The new balance must be equal to the old balance minus the withdrawal amount.

context SavingsAccount::withdraw(amount : Real)
    post: balance = balance@pre - amount
EXAMPLE: MORTGAGE CONSTRAINTS

Consider the class diagram representing mortgages that people have on houses. A person owns a house, which is paid for by taking out a mortgage. The mortgage takes as security the house that is owned by the person.

Express the following constraints using OCL:

(a) The end date for any mortgage must be at least one year after the start date.

(b) If a person has a mortgage, he or she must own a house.
EXAMPLE: MORTGAGE CONSTRAINTS

- **House**
  - value: Money
  - security: 1

- **Mortgage**
  - principle: Money
  - monthlyPayment: Money
  - startDate: Date
  - endDate: Date

- **Person**
  - hkid: String
  - monthlySalary: Money

- **Owns**
  - owner: 1

- **Holds**
  - borrower: 1
EXAMPLE: MORTGAGE CONSTRAINTS — SOLUTION

(a) The end date for any mortgage must be at least one year after the start date.

**Context** Mortgage

inv: endDate - startDate \(\geq\) 1 year \(\text{(can also use 365 days)}\)

**Context** Mortgage

inv: startDate + 1 year \(\geq\) endDate \(\text{(can also use 365 days)}\)

Since we are writing a constraint *not* a program, there is no need to specify conversion of dates to years or days. It is only necessary to convey the *intent* of the constraint.
(b) If a person has a mortgage, he or she must own a house.

**Context** Person

**inv**: Mortgage $\rightarrow$ notEmpty() implies House $\rightarrow$ notEmpty()

For a person, we check whether he/she is related to any mortgages (via the Holds association) and if this collection is not empty, then the person must also be related to houses (i.e., have a non empty collection of House instance).

**Context** Mortgage

**inv**: borrower.House $\rightarrow$ notEmpty()

For a mortgage, we navigate to its related borrower (Person) instance and then to the borrower’s related House instances, which must be non empty.
OCL Examples

- **Invariants:**
  - *No grandchild may not have more than 2 pet dogs:*
    
    ```
    context Person inv:
    self.child.child.pet -> size() < 2
    ```

- **Pre/Post conditions**
  - ```
    context Automobile::fillTank (in volume:real):real
    pre: volume > 0
    pre: tankLoad + volume < maxLoad
    post: tankLoad = tankLoad@pre + volume
    ```

  @pre→ before the method body executes
Initial and derived values

- **Context** Person::income:Integer
  - **init**: parents.income->sum*1%
  - **derive**: if underAge
    - then parents.income->sum()*1%
    - else job.salary
    - endif

Let expression: aliasing conditions

- **Context** Person inv:
  - let income:Integer =
    - self.job.salary->sum() in
  - if isUnemployed then
    - income < 100
  - else
    - income >=100
  - endif