CIS 771: Software Specifications

Lecture 16:
Pre/Post-conditions In OCL

Outline

- Syntax for pre/post-conditions
- Example pre/post-conditions
- Checking pre/post-conditions in USE

...with the Academia model as the running example.

Pre/Post-condition Syntax

Name of class to which operation belongs

Multiple named preconditions (bool expressions). Each of these may use the parameter 'param1' and the name 'self ' can be used to refer to the receiver object.

```
context Typename::operationName(param1 : Type1, …): ReturnType
  pre precondname1 :  …param1…  …self…
  pre precondname2 :  …param1…  …self…

     …

  post postcondname1 : …result… ...param1... …self…
  post postcondname2 : …result… ...param1... …self…
```

Multiple named postconditions (bool expressions). Each of these may use the OCL reserved word result to denote the return value of the operation (if any).

Note: Frame conditions are a special form of post-condition so they are written using 'post'.

Example

Student::newID(n : Integer)

```
context Student::newId(n: Integer)
  pre  GE100:      n >= 100
  post NewId:      id.oclIsNew
  post IdNumber:   id.number = n
```

true, if Id object bound to id attribute of student did not exist in the pre-state.

Example

Student::dropCourse(c: Course)

```
context Student::dropCourse(c: Course)
  pre  NowTaking:   taking->includes(c)
  post NotTaking:   taking->excludes(c)
```

...a first attempt

Assessment

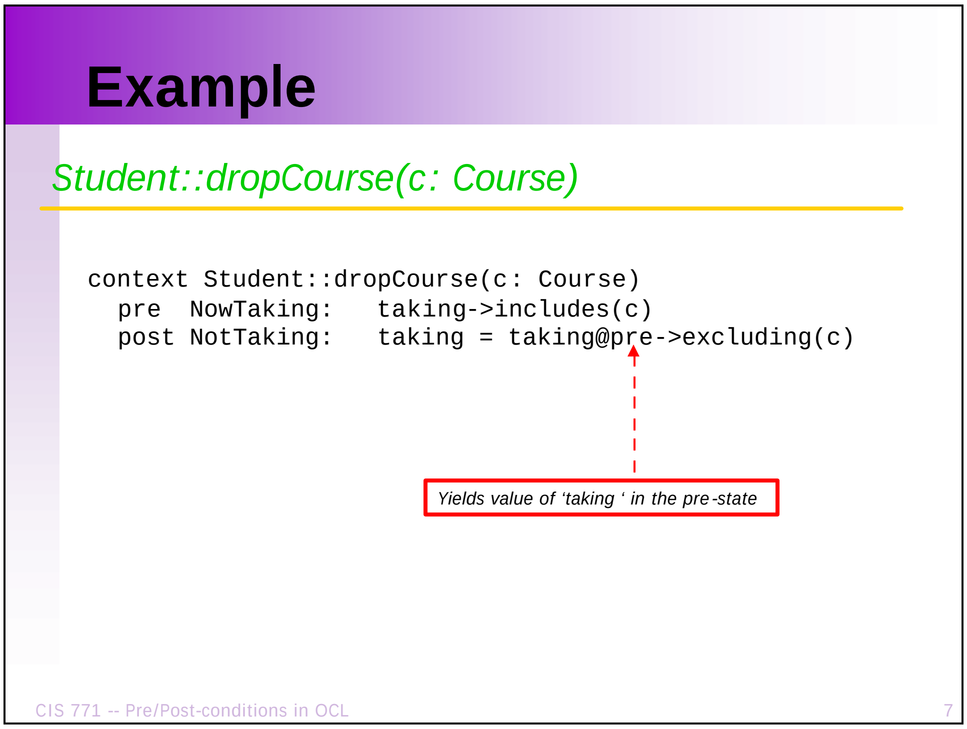
What pre/post states satisfy the specification when c = C3?

```
context Student::dropCourse(c: Course)
  pre  NowTaking:   taking->includes(c)
  post NotTaking:   taking->excludes(c)
```

Example 1:	pre-state : self.taking = {C1,C2,C3}
	post-state : self.taking = {C1,C2}
		✓

Example 2:	pre-state : self.taking = {C1,C2,C3}
	post-state : self.taking = {}
		✓

...but this is not what we want

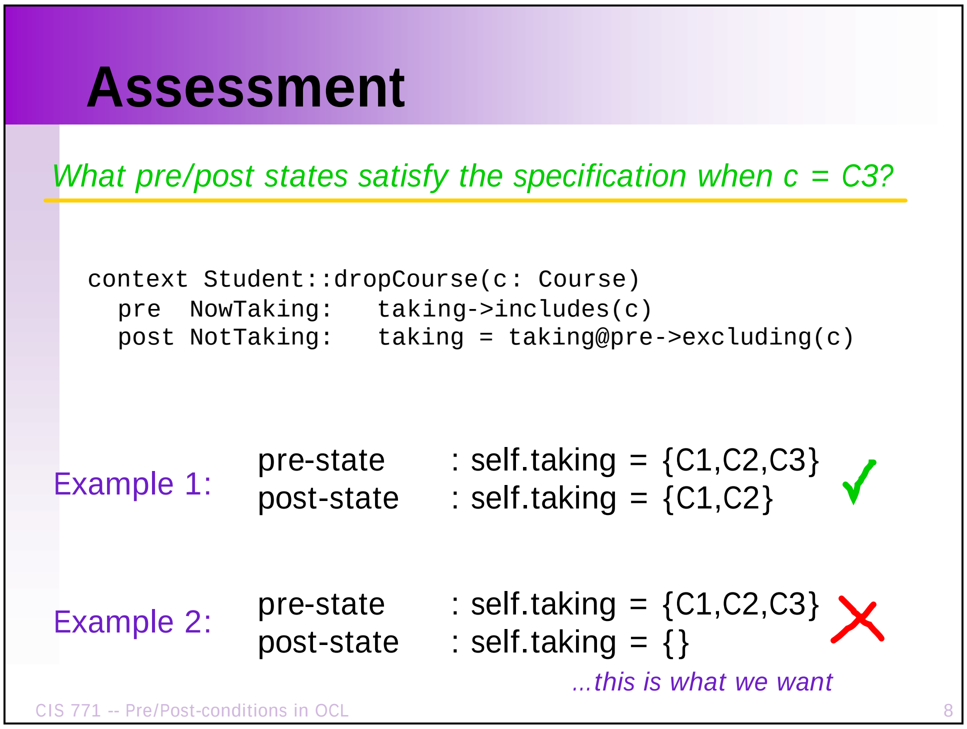


Example

Student::dropCourse(c: Course)

```
context Student::dropCourse(c: Course)
  pre  NowTaking:   taking->includes(c)
  post NotTaking:   taking = taking@pre->excluding(c)
```

Yields value of 'taking' in the pre-state



Assessment

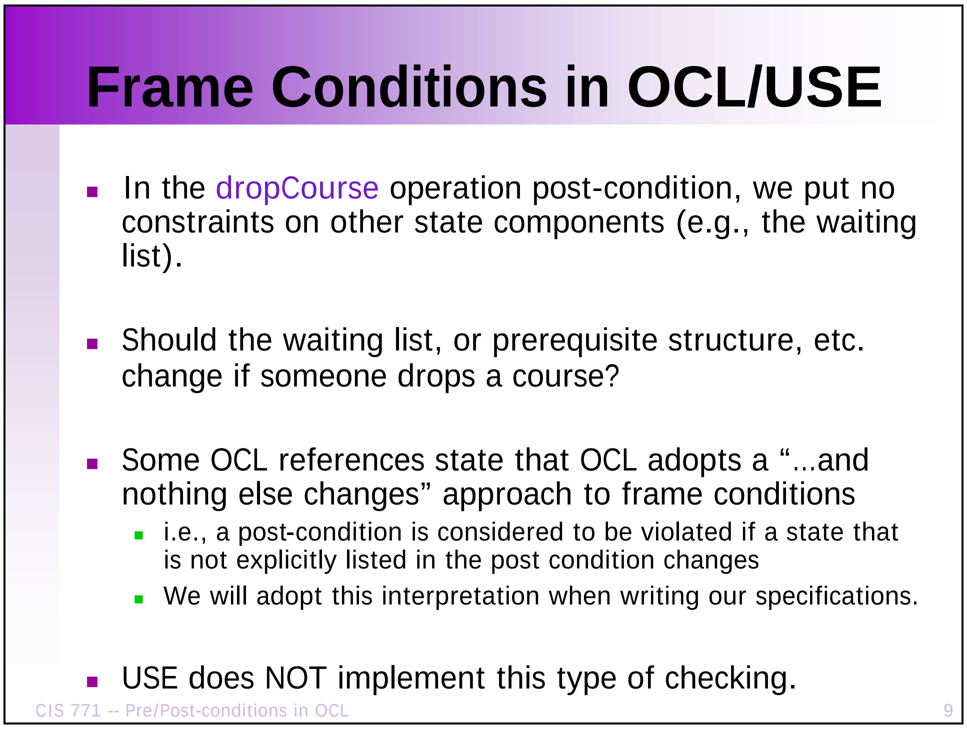
What pre/post states satisfy the specification when c = C3?

```
context Student::dropCourse(c: Course)
  pre  NowTaking:   taking->includes(c)
  post NotTaking:   taking = taking@pre->excluding(c)
```

Example 1:	pre-state
	post-state
		: self.taking = {C1,C2,C3}
		: self.taking = {C1,C2}
			✓

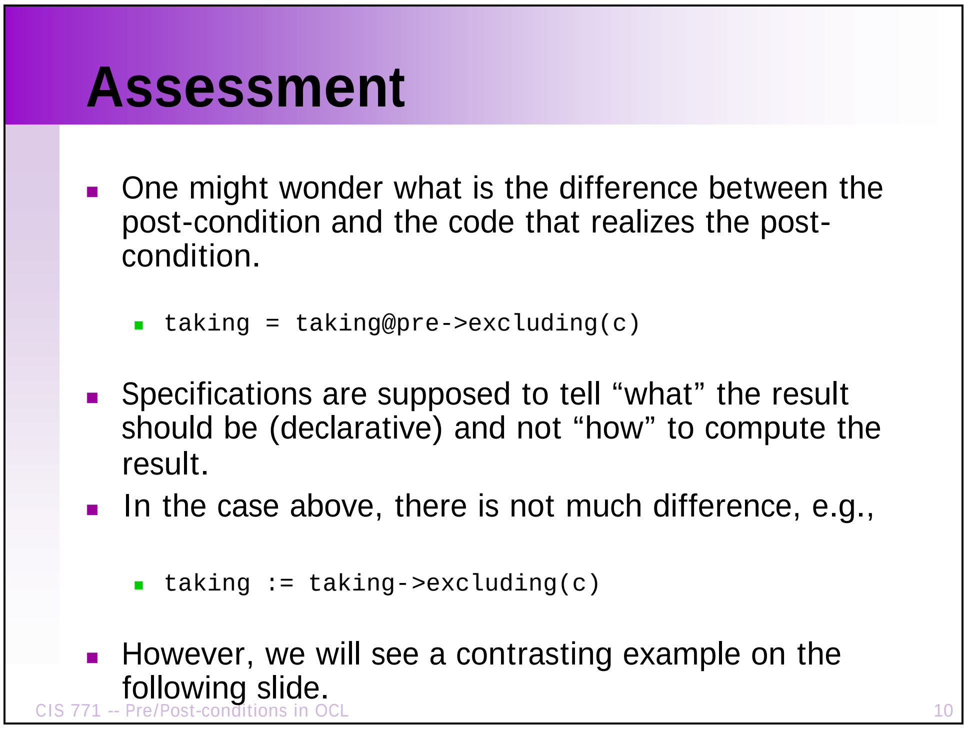
Example 2:	pre-state
	post-state
		: self.taking = {C1,C2,C3}
		: self.taking = {}
			✗

...this is what we want



Frame Conditions in OCL/USE

- In the dropCourse operation post-condition, we put no constraints on other state components (e.g., the waiting list).
- Should the waiting list, or prerequisite structure, etc. change if someone drops a course?
- Some OCL references state that OCL adopts a "...and nothing else changes" approach to frame conditions
  - i.e., a post-condition is considered to be violated if a state that is not explicitly listed in the post condition changes
  - We will adopt this interpretation when writing our specifications.
- USE does NOT implement this type of checking.



Assessment

- One might wonder what is the difference between the post-condition and the code that realizes the post-condition.
  - `taking = taking@pre->excluding(c)`
- Specifications are supposed to tell "what" the result should be (declarative) and not "how" to compute the result.
- In the case above, there is not much difference, e.g.,
  - `taking := taking->excluding(c)`
- However, we will see a contrasting example on the following slide.

Example

Sorting sequences of integers

```
context A::sort(s : Sequence(Integer)) : Sequence(Integer)

  post SameSize:
    result->size = s->size

  post SameElements:
    result->forAll(i | result->count(i) = s->count(i))

  post IsSorted:
    Sequence{1..(result->size-1)}->
      forAll(i | result.at(i) <= result.at(i+1))
```

Correctness vs. Algorithm

- We specify only the correctness criteria, not how the results are computed
- The particular algorithm (e.g., quicksort, heapsort) can be chosen at implementation time
- We need not define a unique result

Academia Operations

- Student::addCourse(c: Course)
  - add c to set of courses that self is taking
  - preconditions
    - self is not already taking c
    - self has already taken the prerequisites of c
  - postconditions
    - c is added to the set of course that self is taking (and the set of courses being taken by self is otherwise unchanged)

Academia Operations

- Course::addPreReq(c: Course)
  - make c a prerequisite of self
  - preconditions
    - c is not already a prereq of self
    - self is not transitively a prereq of c (I.e., this will ensure that there are no cycles in the prereq structure – an invariant that the operation should preserve)
  - postconditions
    - c is added to the set of self's prerequisites (and self's prerequisites are otherwise unchanged)

Academia Operations

- Student::completeCourse(c: Course, g: Grade)
  - self completes course c and earns grade g
    - c is removed from the set of courses that self is taking
    - c is added to self's transcript with grade g
  - preconditions
    - *...for you to do...*
  - postconditions
    - *...for you to do...*

For You To Do...

- Pause the lecture...
- Starting with the file `academia-9.use`, construct the pre/post-conditions for the *Academia* operations listed on the following slide.

Checking Pre/Post-Conditions in USE

Structure of a USE script for checking pre/post-conditions

```
...
use commands to generate some object
instances
...
!openter <source-expr> <operation-name> ( [<argument-expr-list>] )
...
use commands to carry out the effects
(state changes) of the operation
...
!opexit
...
use commands that rely on the
effects/values produced by the operation
...
```

Checking Pre/Post-Conditions in USE

Example using the dropCourse operation

```
-- create instances suitable for the academia-9 model
-- (use the script academia-extension4-instantiation.cmd)
read academia-extension4-instantiation.cmd

-- should succeed because Oksana is currently taking cis775
!openter oksana dropCourse(cis775)

-- effect of dropCourse
!delete (oksana,cis775) from Taking

-- exit operation, check postconditions with state saved at operation
-- entry time and current state (should succeed)
!opexit
```

Details of openter

1. Source expression is evaluated to obtain the receiver object.

2. The argument expressions are evaluated.

!openter oksana dropCourse(cis775)

Bindings:
self = student instance named by oksana
c = course instance named by cis775

3. self is bound to object to which oksana evaluates, and formal parameter c is bound to object to which cis775 evaluates.

4. All pre-conditions of the operation are evaluated using old variables bindings plus the new bindings to self and formal parameters.

5. If check of pre-conditions succeeds, operation call with new bindings are pushed on operation stack and the pre-state is saved (to be accessed if the post-condition refers to it using the '@pre' construct)

Details of openter

1. The source expression is evaluated to determine the receiver object.

2. The argument expressions are evaluated.

3. The variable self is bound to the receiver object and the argument values are bound to the formal parameters of the operation. These bindings determine the local scope of the operation.

4. All preconditions specified for the operation are evaluated.

5. If all preconditions are satisfied, the current system state is saved and the operation call is saved on a call stack.

Using openter

Example using the dropCourse operation

```
-- create instances suitable for the academia-9 model
-- (use the script academia-extension4-instantiation.cmd)
read academia-extension4-instantiation.cmd

-- should succeed because Oksana is currently taking cis775
!openter oksana dropCourse(cis775)

info vars
...
c : Course = @cis775
self : Grad = @oksana

info opstack
1. Student::dropCourse(c : Course) | oksana.dropCourse(@cis775)
```

Details of opexit

1. The currently active operation is popped from the call stack.
2. If an optional result value is given, it is bound to the special OCL variable "result".
3. All postconditions specified for the operation are evaluated in context of the current system state and the pre-state saved at operation entry time.
4. All local variable bindings are removed.

Assessment

- The previous example script for dropCourse only tests the operation specification in one context (academia-extension4-instantiation.cmd) and for one set of argument values.
- The user is responsible for creating enough tests to reveal any potential flaws in the operation specification.
- The operation actions are abstractly simulated by the USE command-line steps (e.g., deleting a pair from an association).

Methodology

- Write an operation specification
- Come up with the USE command-line steps that capture what you intend to actually code in the body of the operation
- Write a number of test contexts and operation calls to test your operation’s functionality (as specified by the USE command-line steps) against the specified pre/post-conditions
- Once you are satisfied that your specification and abstract implementation are correct, you can code the operation by providing a concrete implementation for your command-line steps

For You To Do…

- Pause the lecture…
- If you have not already done so, read through the tutorial on pre/post-conditions in USE (using the Employee model) that comes with the USE distribution.
- Code the three operation specifications in USE that you wrote earlier in the lecture, and develop scripts to test your specifications.
  - Make sure that your scripts include multiple test cases – cases that cause the pre/post-conditions to fail as well as cases that cause the conditions to succeed.

Acknowledgements

- Material for this lecture is based on the following sources
  - Chapter 7 (the OCL chapter) of the OMG-UML specification (version 1.3 – March 2000)
  - The documentation from the USE distribution (in particular, the documentation on pre/post-conditions that uses the Employee model)