Contracts and the Object Constraint Language

Perdita Stevens

School of Informatics
University of Edinburgh

Plan

1. Introducing the use of contracts in development
2. Contracts in English
3. Contracts in the Object Constraint Language (part of UML)

Contracts

In ordinary life, a contract lays out an agreement between two (or more) parties: what are each party's obligations and rights?

Two benefits:

- Avoid misunderstanding: making obligations explicit increases chance that they will be met;
- Assign blame: if an obligation is not met, the terms of the contract should make it clear who is at fault.

Contracts in software design

Term Design by Contract introduced (and trademarked!) by Bertrand Meyer.

By making explicit the contract between supplier of a service and the client, D by C

- contributes to avoiding misunderstandings and hard-to-track bugs;
- supports clear documentation of a module – clients should not feel the need to read the code!
- supports defensive programming;
- allows avoidance of double testing.

Software contracts may be tool-supported and checked, or purely for human reading.

Design by contract

says: there should be explicit contracts attached to the responsibilities a class exists to fulfill.

A method has:

- a precondition – this must be true when the method is invoked, or all bets are off
- a postcondition – the method promises to ensure this, provided its precondition was met

A class has:

an invariant, which it must maintain.

Subcontracting

When a subclass reimplements an operation it must fulfill the contract entered into by its base class – for substitutivity. A client must not get a nasty surprise because in fact a subclass did the job.

Rule of subcontracting:

Demand no more: promise no less

It's OK for a subclass to weaken the precondition, i.e. to work correctly in more situations... but not OK for it to strengthen it.

It's OK for a subclass to strengthen the postcondition, i.e. to promise more stringent conditions... but not OK for it to weaken it.

Understanding the role of LSP

Remember, the Liskov Substitution Principle was an attempt to say when subclassing was safe *without* having explicit contracts.

Effectively, to obey the LSP, a subclass must obey whatever contract for the superclass is in the client's mind – any property of the superclass must also hold of the subclass. This is why LSP is too strong to expect it to hold in all good designs (but the best you can do, without making contracts explicit).

Constraints in a UML model

Constraints allow you to give more information about what will be considered a correct implementation of a system described in UML.

Specifically, they constrain one or more model elements, by giving conditions which they must satisfy.

They are written in an appropriate language, enclosed in set brackets $\{...\}$ and attached to the model in some visually clear way.

Constraining implementation of a class (1)

A class invariant restricts the legal objects by specifying a relationship between the attributes and/or the attributes of associated classes.

Simple example: the class invariant

{name is no longer than 32 characters}

could be applied to a class Student which has an attribute

name : String

to forbid certain values of that attribute.

Implementors of the class must ensure that the invariant is satisfied (when?)

Clients of the class may assume it.

Constraining implementation of a class (2)

Suppose our class Student is associated with classes DirectorOfStudies and also with Lecturer by tutor – a student has a DoS and a tutor.

Suppose it is forbidden for the student's DoS and tutor to be the same person.

We can represent this by a class invariant on Student, say

{ student's tutor and DoS are different }

(Is this sufficiently unambiguous?)

Constraining implementation of an operation

We can constrain the behaviour of operations using pre and post conditions.

A pre condition must be true before the operation is invoked – it is the client's responsibility to ensure this.

A post condition must be true after the operation has been carried out – it is the class's implementor's responsibility to ensure this.

E.g.

`context Module::register(s : Student)`
`pre:` s is not registered for the module
`post:` the set of students registered for the module is whatever it was before plus student `s`.

What contracts are good for

At the beginning we claimed that the use of contracts:

- contributes to avoiding misunderstandings and hard-to-track bugs; because assumptions and promises are explicit: if all contracts are explicit and dovetail nicely, the bug is in the code that doesn't fulfil its contract
- supports clear documentation of a module – clients should not feel the need to read the code! reading the contract should be enough
- supports defensive programming; e.g., when you implement an operation, verify that the postcondition holds before returning; o/w fail gracefully and report a bug
- allows avoidance of double testing. e.g., when you implement an operation, you need not test that your preconditions are satisfied: that's the job of the client to ensure. (Defensively, you may wish to anyway: defensiveness/performance trade-off.)

Languages for contracts

Writing contracts in English can be

- ambiguous
- long-winded
- hard to support with tools

– but nevertheless, careful English is very often the best language to use! Using it certainly beats using a formal language that people who need to read it can't read!

Formal languages for contracts

If English is not good enough, you could consider:

- the chosen programming language – e.g. write a Boolean expression in Java that should evaluate to true.
  + can be pasted into the implementation and checked at runtime
  − may be too low level, e.g. lack quantifiers
- "plain" mathematics and/or logic
  + don't need any special knowledge or facilities
  − can end up being the worst of all worlds, e.g. unfamiliar, lacking UML integration
- a formal specification language e.g. Z or VDM
  + can be truly unambiguous, some tool support exists
  − unfamiliar to most people, may be non-ASCII, need special UML-integrated dialect
- The Object Constraint Language, OCL

About OCL

OCL aimed for the sweet spot between formal specification languages and use of English. It tries to be formal but easy to learn and use.

Extensively used in the documentation of the UML language itself, and related standards.

Written in plain text (no funny symbols).

Had serious semantic problems, but these seem to be solved now.

Some tool support.

Worth knowing a bit about.

OCL for SEOC

The following slides *summarise* what you will need to know, but are not complete notes.

Required reading from the OCL standard will give you the details.

Bottom line: you should be able to read and write straightforward OCL constraints. But you are not expected e.g. to memorise the list of reserved words!

Examples here; tutorial coming up.

OCL basic types

- Boolean
- String
- Integer
- Real
- (UnlimitedNatural)

With all the operations you'd expect. (Know the ones in Table 7.2 of spec.)

Integer is considered a subtype of Real.

(Remark: OCL uses the terms class and type interchangeably, which is just about OK in this context, though normally a big mistake.)

Example: class invariant

```
context Company inv:
  self.numberOfEmployees > 50
```

Note that declaring the context to be `Company` means that `self` refers to an instance of class `Company`.

Specifying that this is a class invariant (`inv`) means that the constraint has to be true of every instance of class `Company`.

Example: pre and post conditions

```
context Stove::open()
  pre: status = OVENSTATUS::off
  post: status = OVENSTATUS::off and isOpen
```

Here `status` and `isOpen` are attribute names of `Stove`. We could have written `self.status` etc.

`off` is a member of the enum type OVENSTATUS.

More features useful for pre and post conditions

Arguments and return type could be specified in the context:

```
context MyClass::foo(i:Integer):String
```

and then `i` can be referred to in this context.

Reserved word `result` can be used in the postcondition.

You can refer to the old value of an attribute using `pre`, e.g.

```
context MyClass::incrementCount()
post:count = count@pre + 1
```

OCL collection types

- Collection
- Set
- Bag
- Sequence
- (Tuple)
- (OrderedSet)

Set, Bag, Sequence are kinds of Collection: more specifically, Set(S) conforms to Collection(T) iff S conforms to T, etc.

Reasonable facilities for manipulating collections. E.g.

```
context Company inv:
self.employee->select(age > 50)->notEmpty()
```

Note the use of the arrow to access properties of collections...

Collections operations returning collections

(NB All these have variants that allow you to name the collection element.)

```
collection->select(boolean_expression)
collection->reject(boolean_expression)
```

(you might recognise this as filter in FP?)

```
collection->collect(expression)
```

(like map in FP) NB collect on a Set gives you a Bag.

Conversion operations, especially:

```
collection->asSet()
```

Collections operations returning boolean

Emptiness checking:

```
collection->isEmpty()
collection->notEmpty()
```

Quantifiers:

```
collection->forAll(boolean_expression)
collection->exists(boolean_expression)
```

Convenient variant:

```
self.employee->forAll( e1, e2 : Person
  | e1 <> e2 implies e1.forename <> e2.forename)
```

Collections operations returning numbers

```
collection->sum() -- type depends on element type
collection->size()
```

Navigation

As we've seen, an OCL expression in the context of one class A may refer to an associated class B.

Single (? - 1) association: straightforward, since any object of class A determines just one object of class B:

- If there's a rolename use it, e.g. `self.DoS.name`
- If not may just use classname, e.g. `self.DirectorOfStudies.name`

More navigation

What if the association is not (? - 1)? E.g. consider the same association from the point of view of the DirectorOfStudies – a DoS may direct many Students.

For each DirectorOfStudies the rolename directee refers to a *set* of Students. Use OCL collection operations, e.g.

```
self.directee->forAll (regNo <= 200000)
self.directee->notEmpty()
```

(If you use a collection operation on something that isn't a collection it gets interpreted as a set containing one element!)

Two-stage navigation

What happens if we take more than one “hop” round the class diagram?

e.g. what is `self.student.module`?

It’s deemed to be short for

```
self.student->collect(module)
```

which is a *Bag* (not a Set) of all the modules taken by students linked to self.

Notice that putting such a constraint into a UML model creates a dependency of self on module, if there wasn’t one already.

Using operations in OCL

Consider an operation register(s:Student) of Module. Should we be able to refer to this operation in an OCL expression?

Problem: it does something – alters the state of the Module. When should this happen, if at all?

Only good way round this is to allow in OCL *only* operations that guarantee not to alter the state of any object.

Such operations are known as queries – in UML an operation has an attribute isQuery which must be true for the operation to be legal in OCL.

“Control” structures

Naturally we don’t need much in the way of control structures: OCL is a constraint language, used for defining expressions (not commands, not functions).

- `if ... then ... else ... endif`
- `let v : Sometype = someExpression in ...`
  (NB there’s no endlet! Typical use: let begins the whole constraint, and its scope extends to the end.)