Contracts and Interaction Design

Perdita Stevens

School of Informatics
University of Edinburgh

Plan

1. Contracts and Design by Contract
2. Command-Query Separation
3. Law of Demeter
4. Fluent Interfaces

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:

$\boxed{\text{Demand no more: promise no less}}$

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, which you have met.

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.

Command-Query Separation

Term also coined by Bertrand Meyer (like DbC) and goes well with it.

Separate:

- commands, which may change an object's state
- queries, which return a value.

Then queries can be freely used in contracts – running them does not change anything! In UML operations can be given a {`query`} property when they guarantee not to change anything, and this is what allows you to use them in constraints.

Can be bent (think about it!) – the value comes from queries *not* changing state, rather than from commands not returning values. Advantage of following it strictly: it's easy to do! Disadvantage: coming up after...

Law of Demeter

in response to a message *m*, an object *o* should send messages *only* to the following objects:

1. *o* itself
2. objects which are sent as arguments to the message *m*
3. objects which *o* creates as part of its reaction to *m*
4. objects which are directly accessible from *o*, that is, using values of attributes of *o*.

Law of Demeter

in response to a message *m*, an object *o* should send messages *only* to the following objects:

1. *o* itself
2. objects which are sent as arguments to the message *m*
3. objects which *o* creates as part of its reaction to *m*
4. objects which are directly accessible from *o*, that is, using values of attributes of *o*.

In particular *o* should not send a message to an object which is acquired by sending another message e.g.

```
myP.getThing().doSomething(); //violates LoD
```

Why is the LoD not “the one dot rule”?

You can see the attraction: LoD tries to rule out code like:

```
myP.getThing().doSomething();
```

Some code violates LoD without having more than one dot on a line:

```
Thing t = myP.getThing();

t.doSomething();
```

More interestingly some has more than one dot on a line and does not violate it: e.g. if an object returned from a message was already accessible.

Let’s look at the rationale behind LoD, rather than the mechanics.

Rationale

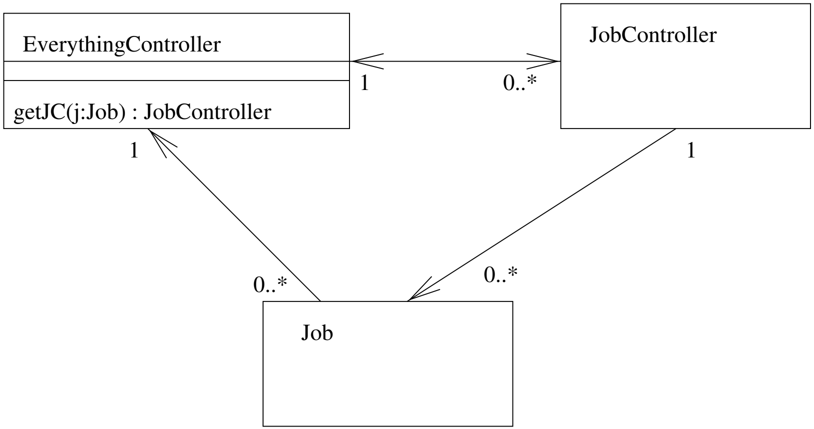
The Law of Demeter tries to avoid *indirect* dependencies of one class on another, which may be hard to spot from code or models.

E.g. if class `OClass` has an attribute `myP` of class `P`, it is clear from the source of `OClass` that it depends on `P`. If `P` changes, we will easily discover that we have to check whether `OClass` needs to change.

But if `P` has a method `getThing()` returning an object of class `Thing` and *o* calls this and sends the resulting `Thing` a message, now `OClass` depends on `Thing`.

This may not be readily apparent from `OClass`'s code or a corresponding UML diagram. That's the problem.

Setting where LoD helps avoid design problem



(names slightly changed to protect the guilty)

But the LoD must not be followed slavishly...

There are several situations where even a good design will disobey the LoD, and we detect them by understanding the rationale for it.

Suppose my code goes:

```
myP.getThing().doSomething();
```

First: if I already depend on the class providing `doSomething()`, no harm is done.

Second: if I can't modify `myP`'s class (to make its API more complete and offer me the service I'm accessing this way), I may have no good alternative.

Indeed, the purer the OO language the more likely it is that there's no point in a method returning something if you can't subsequently sent it a message!

E.g. where a method returns a String.

Example

Suppose we are responsible for classes O, P and Thing.

```
public class OClass {
   private P myP;
   public void m(String s) {
      Thing th = myP.getThing(); //ok, attribute
      P p = new P();
      Thing newth = p.getThing(); //ok, object created here
      int sl = s.length(); //ok, argument
      String t = this.n(); //ok, object itself
      int tl = t.length(); //technically not ok
      int tl2 = this.n().length(); //technically not ok
      myP.getThing().doSomething(); //really not ok
   }
   public String n() {...return someString;}
}
```

Example

Suppose we are responsible for classes O, P and Thing.

```
public class OClass {
   private P myP;
   public void m(String s) {
      Thing th = myP.getThing(); //ok, attribute
      P p = new P();
      Thing newth = p.getThing(); //ok, object created here
      int sl = s.length(); //ok, argument
      String t = this.n(); //ok, object itself
      int tl = t.length(); //technically not ok
      int tl2 = this.n().length(); //technically not ok
      myP.getThing().doSomething(); //really not ok
   }
   public String n() {...return someString;}
}
```

Example

Suppose we are responsible for classes O, P and Thing.

```
public class OClass {
   private P myP;
   public void m(String s) {
      Thing th = myP.getThing(); //ok, attribute
      P p = new P();
      Thing newth = p.getThing(); //ok, object created here
      int sl = s.length(); //ok, argument
      String t = this.n(); //ok, object itself
      int tl = t.length(); //technically not ok
      int tl2 = this.n().length(); //technically not ok
      myP.getThing().doSomething(); //really not ok
   }
   public String n() {...return someString;}
}
```

Example

Suppose we are responsible for classes O, P and Thing.

```
public class OClass {
   private P myP;
   public void m(String s) {
      Thing th = myP.getThing(); //ok, attribute
      P p = new P();
      Thing newth = p.getThing(); //ok, object created here
      int sl = s.length(); //ok, argument
      String t = this.n(); //ok, object itself
      int tl = t.length(); //technically not ok
      int tl2 = this.n().length(); //technically not ok
      myP.getThing().doSomething(); //really not ok
   }
   public String n() {...return someString;}
}
```

Example

Suppose we are responsible for classes O, P and Thing.

```
public class OClass {
   private P myP;
   public void m(String s) {
      Thing th = myP.getThing(); //ok, attribute
      P p = new P();
      Thing newth = p.getThing(); //ok, object created here
      int sl = s.length(); //ok, argument
      String t = this.n(); //ok, object itself
      int tl = t.length(); //technically not ok
      int tl2 = this.n().length(); //technically not ok
      myP.getThing().doSomething(); //really not ok
   }
   public String n() {...return someString;}
}
```

Example

Suppose we are responsible for classes O, P and Thing.

```
public class OClass {
   private P myP;
   public void m(String s) {
      Thing th = myP.getThing(); //ok, attribute
      P p = new P();
      Thing newth = p.getThing(); //ok, object created here
      int sl = s.length(); //ok, argument
      String t = this.n(); //ok, object itself
      int tl = t.length(); //technically not ok
      int tl2 = this.n().length(); //technically not ok
      myP.getThing().doSomething(); //really not ok
   }
   public String n() {...return someString;}
}
```

Example

Suppose we are responsible for classes O, P and Thing.

```
public class OClass {
   private P myP;
   public void m(String s) {
      Thing th = myP.getThing(); //ok, attribute
      P p = new P();
      Thing newth = p.getThing(); //ok, object created here
      int sl = s.length(); //ok, argument
      String t = this.n(); //ok, object itself
      int tl = t.length(); //technically not ok
      int tl2 = this.n().length(); //technically not ok
      myP.getThing().doSomething(); //really not ok
   }
   public String n() {...return someString;}
}
```

Example

Suppose we are responsible for classes O, P and Thing.

```
public class OClass {
   private P myP;
   public void m(String s) {
      Thing th = myP.getThing(); //ok, attribute
      P p = new P();
      Thing newth = p.getThing(); //ok, object created here
      int sl = s.length(); //ok, argument
      String t = this.n(); //ok, object itself
      int tl = t.length(); //technically not ok
      int tl2 = this.n().length(); //technically not ok
      myP.getThing().doSomething(); //really not ok
   }
   public String n() {...return someString;}
}
```

What should a method return?

Conventionally, many OO methods return `void`. Their job is to change some state, not to compute a result. They are *commands*, e.g. *modifiers*.

In strict Command Query Separation any method either changes state, or returns a value, but not both. ("Asking a question should not change the answer.")

Advantages include: then all non-void-returning methods can be used in OCL constraints, because they're all queries!

But there are disadvantages to this separation, not least that it can lead to repetitious code:

```
customer.setFirstName(``John'');
customer.setLastName(``Bloggs'');
customer.setAge(32);
```

Method chaining

If modifiers return themselves – their code ends with `return this;` – we can write instead:

```
customer.setFirstName(``John'')
        .setLastName(``Bloggs'')
        .setAge(32);
```

and sometimes this is a win.

In the pure form this does *not* violate LoD (why?)

Fluent interfaces go further and often do violate LoD, in order to gain advantages of, well, fluency. Moving towards the design of an *internal domain-specific language*

Conclusion

The great thing about design principles and patterns is that there are so many to choose from.

You cannot, and should not, try to follow them all at all times.

Try to be aware of what underlies them, and use them as a guide where appropriate. Often reading discussion of them, and of how they reinforce or conflict with, one another, is illuminating.