Interfaces and interactions

Perdita Stevens

School of Informatics
University of Edinburgh

Law of Demeter

(Recall from Inf2C-SE):

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

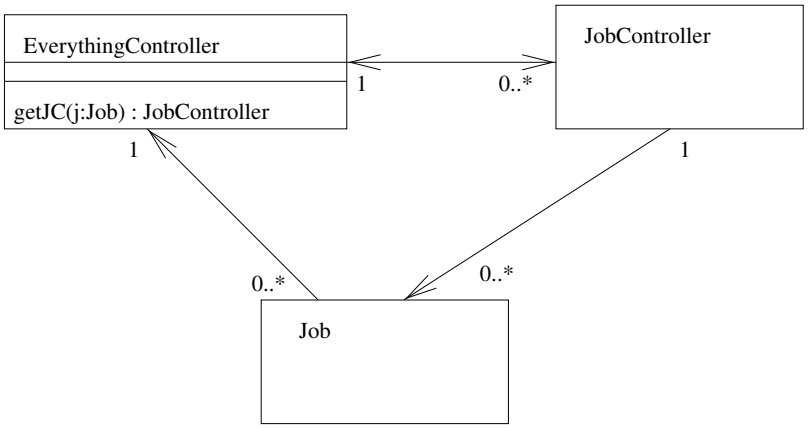
The Law of Demeter tries to avoid *indirect* dependencies of one class on another, which may be hard to spot from code or models.

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;}
}
```

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*.

Bertrand Meyer proposed the principle of Command Query Separation saying that any method should either change state, or return 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);
```

Alternative: `return this`

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, though it does violate CQS.

Fluent interfaces go further and often do violate LoD, in order to gain advantages of, well, fluency.

Optional exercise: google fluent interfaces and read up on this.

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.

We've seen:

- SOLID
- Many properties of APIs
- Law of Demeter
- Command Query Separation
- Many patterns