FluentInterface

dsl **20 December 2005** Reactions

A few months ago I attended a workshop with Eric Evans, and he talked about a certain style of interface which we decided to name a fluent interface. It's not a common style, but one we think should be better known. Probably the best way to describe it is by example.

The simplest example is probably from Eric's timeAndMoney library. To make a time interval in the usual way we might see something like this:

```
TimePoint fiveOClock, sixOClock;
...
TimeInterval meetingTime = new TimeInterval(fiveOClock, sixOClock);
```

The timeAndMoney library user would do it this way:

```
   TimeInterval meetingTime = fiveOClock.until(sixOClock);
```

I'll continue with the common example of making out an order for a customer. The order has line-items, with quantities and products. A line item can be skippable, meaning I'd prefer to deliver without this line item rather than delay the whole order. I can give the whole order a rush status.

The most common way I see this kind of thing built up is like this:

```
    private void makeNormal(Customer customer) {
        Order o1 = new Order();
        customer.addOrder(o1);
        OrderLine line1 = new OrderLine(6, Product.find("TAL"));
        o1.addLine(line1);
        OrderLine line2 = new OrderLine(5, Product.find("HPK"));
        o1.addLine(line2);
        OrderLine line3 = new OrderLine(3, Product.find("LGV"));
        o1.addLine(line3);
        line2.setSkippable(true);
        o1.setRush(true);
    }
```

In essence we create the various objects and wire them up together. If we can't set up everything in the constructor, then we need to make temporary variables to help us complete the wiring - this is particularly the case where you're adding items into collections.

Here's the same assembly done in a fluent style:

```
   private void makeFluent(Customer customer) {
        customer.newOrder()
                .with(6, "TAL")
                .with(5, "HPK").skippable()
                .with(3, "LGV")
                .priorityRush();
    }
```

Probably the most important thing to notice about this style is that the intent is to do

Links
home
bliki
feed
Translations
Japanese
Spanish
Korean
Chinese
Thai
Categories
agile
design
dsl
leisure
refactoring
ruby
thoughtWorks
tools
uml
writing
Blog Roll
ThoughtBlogs
TW Alumni
Nicholas Carr
Steve Cook
Brian Foote
Simon Harris
Gregor Hohpe
Andy Hunt
Ralph Johnson
Patrick Logan
David Ing
Brian Marick
Jeremy Miller
Jimmy Nilsson
Samuel Pepys
Keith Ray
Johanna

something along the lines of an internal DomainSpecificLanguage. Indeed this is why we chose the term 'fluent' to describe it, in many ways the two terms are synonyms. The API is primarily designed to be readable and to flow. The price of this fluency is more effort, both in thinking and in the API construction itself. The simple API of constructor, setter, and addition methods is much easier to write. Coming up with a nice fluent API requires a good bit of thought.

Indeed one of the problems of this little example is that I just knocked it up in a Calgary coffee shop over breakfast. Good fluent APIs take a while to build. If you want a much more thought out example of a fluent API take a look at JMock. JMock, like any mocking library, needs to create complex specifications of behavior. There have been many mocking libraries built over the last few years, JMock's contains a very nice fluent API which flows very nicely. Here's an example expectation:

```
mock.expects(once()).method("m").with( or(stringContains("hello"),
                                          stringContains("howdy")) );
```

I saw Steve Freeman and Nat Price give an excellent talk at JAOO2005 on the evolution of the JMock API, they since wrote it up in an OOPSLA paper.

So far we've mostly seen fluent APIs to create configurations of objects, often involving value objects. I'm not sure if this is a defining characteristic, although I suspect there is something about them appearing in a declarative context. The key test of fluency, for us, is the Domain Specific Language quality. The more the use of the API has that language like flow, the more fluent it is.

Building a fluent API like this leads to some unusual API habits. One of the most obvious ones are setters that return a value. (In the order example `with` adds an order line to the order and returns the order.) The common convention in the curly brace world is that modifier methods are void, which I like because it follows the principle of CommandQuerySeparation. This convention does get in the way of a fluent interface, so I'm inclined to suspend the convention for this case.

You should choose your return type based on what you need to continue fluent action. JMock makes a big point of moving its types depending on what's likely to be needed next. One of the nice benefits of this style is that method completion (intellisense) helps tell you what to type next - rather like a wizard in the IDE. In general I find dynamic languages work better for DSLs since they tend to have a less cluttered syntax. Using method completion, however, is a plus for static languages.

One of the problems of methods in a fluent interface is that they don't make much sense on their own. Looking at a method browser of method by method documentation doesn't show much sense to `with`. Indeed sitting there on its own I'd argue that it's a badly named method that doesn't communicate its intent at all well. It's only in the context of the fluent action that it shows its strengths. One way around this may be to use builder objects that are only used in this context.

One thing that Eric mentioned was that so far he's used, and seen, fluent interfaces mostly around configurations of value objects. Value objects don't have domain-meaningful identity so you can make them and throw them away easily. So the fluency rides on making new values out of old values. In that sense the order example isn't that typical since it's an entity in the EvansClassification.

I haven't seen a lot of fluent interfaces out there yet, so I conclude that we don't know much about their strengths and weaknesses. So any exhortations to use them can only be preliminary - however I do think they are ripe for more experimentation.

There's a good follow up to this from Piers Cawley.

Update (23 June 2008). Since I wrote this post this term's been used rather widely, which gives me a nice feeling of tingly gratification. I've refined my ideas about fluent interfaces and internal DSLs in the book I've been working on. I've also noticed a common misconception - many people seem to equate fluent interfaces with Method Chaining. Certainly chaining is a common technique to use with fluent interfaces, but true fluency is much more than that.

The JMock example I show above uses method chaining, but also nested functions and object scoping

---