The Paperboy, The Wallet, and The Law Of Demeter

By David Bock
dbock@fgm.com

Introduction

When I was in college, I had a professor who stated that every programmer had a 'bag of tricks' – solutions that we tend to use over and over again because in our own personal experience, they work. He thought that his job was to 'put more tricks into our bags'. What this professor was talking about were things like design patterns and idioms.

This paper is going to talk about a particluar 'trick' I like, one that is probably better classified as an 'idiom' than a design pattern (although it is a component in many different design patterns). By being aware of this idiom and understanding how to apply it, I believe that your code will be better, less error prone, and more maintainable. I'm talking about a technique for reducing the 'coupling' between objects in your code. This technique is an easy concept identified by a fancy name: the Law Of Demeter.

Lets start by considering a trivial hypothetical example of a 'Paperboy', who has to get a payment from a 'Customer'. Lets define some code for a Customer, some mechanism for receiving payments from that customer, and a snippet of code that our paperboy will run.

Setting Up Our Example Code

OK, lets start with our Customer. A Customer object may have a first name, a last name, maybe an account number, a shipping address, some method of payment, and so on. For this example though, lets define a trivial customer, with just enough functionality to make our point:

Listing 1 – The Original Customer Class

```
public class Customer {
    private String firstName;
    private String lastName;
    private Wallet myWallet;

    public String getFirstName(){
        return firstName;
    }

    public String getLastName(){
        return lastName;
    }

    public Wallet getWallet(){
        return myWallet;
    }
}
```

We will leave the setters out of this example for the sake of length. By looking at this example, it becomes apparent that we should also define a Wallet object:

Listing 2 – The Simple Wallet Class

```
public class Wallet {
    private float value;

    public float getTotalMoney() {
        return value;
    }

    public void setTotalMoney(float newValue) {
        value = newValue;
    }

    public void addMoney(float deposit) {
        value += deposit;
    }

    public void subtractMoney(float debit) {
        value -= debit;
    }
}
```

Now, that's a pretty simplistic wallet... A future version may include a data structure we can hold things in, like Credit Cards, drivers license, different types of currency, etc. But that's a topic for another paper... For our example, this is sufficient.

So, now we know enough about the Customer and his wallet to start to do some useful stuff. Lets say our paperboy stops by, rings the doorbell, and demands his payment for a job well done. If we had to model this in code, there might be a code snippet that looks something like this:

Listing 3 – Paying the Paperboy

```
    // code from some method inside the Paperboy class...

    payment = 2.00;  // "I want my two dollars!"
    Wallet theWallet = myCustomer.getWallet();
    if (theWallet.getTotalMoney() > payment) {
        theWallet.subtractMoney(payment);
    } else {
        // come back later and get my money
    }
```

So there is a little code snippet that gets the wallet from the customer, checks to make sure he has the money, and transfers it from the customers wallet into the paperboy's wallet.

Why Is This Bad?

So, what's so bad about this code (besides being a horribly contrived example)? Well, lets translate what the code is actually doing into real-language:

Apparently, when the paperboy stops by and demands payment, the customer is just going to turn around, let the paperboy take the wallet out of his back pocket, and take out two bucks.

I don't know about you, but I rarely let someone handle my wallet. There are a number of 'real-world' problems with this, not to mention we are trusting the paperboy to be honest and just take out what he's owed. If our future Wallet object holds credit cards, the paperboy has access to those too... but the basic problem is that "the paperboy is being exposed to more information than he needs to be".

Thats an important concept... The 'Paperboy' class now 'knows' that the customer has a wallet, and can manipulate it. When we compile the Paperboy class, it will need the Customer class and the Wallet class. These three classes are now 'tightly coupled'. If we change the Wallet class, we may have to make changes to both of the other classes.

There is another classic problem that this can create... What happens if the Customer's wallet has been stolen? Perhaps last night a Thief picked the pocket of our example, and someone else on our software development team decided a good way to model this would be by setting the wallet to null, like this:

```
victim.setWallet(null);
```

This seems like a reasonable assumption... Neither the documentation nor the code enforces any mandatory value for wallet, and there is a certain 'elegance' about using null for this condition. But what happens to our Paperboy? Go look back at the code... The code assumes there will be a wallet! Our paperboy will get a runtime exception for calling a method on a null pointer.

We could fix this by checking for '`null`' on the wallet before we call any methods on it, but this starts to clutter the paperboy's code... our real-language description is becoming even worse... "If my customer has a wallet, then see how much he has... if he can pay me, take it". Obviously, we are getting convoluted here.

Improving The Original Code

The proper way to fix this is more like the 'real world' scenario; When the paperboy comes to the door, he's going to ask the customer for payment. He's not going to handle the customer's wallet... he doesn't even have to know the customer HAS a wallet.

Listing 4 – The New Customer Class

```
public class Customer {
    private String firstName;
    private String lastName;
    private Wallet myWallet;

    public String getFirstName(){
        return firstName;
```

```
    }

    public String getLastName(){
        return lastName;
    }

    public float getPayment(float bill) {
        if (myWallet != null) {
            if (myWallet.getTotalMoney() > bill) {
                theWallet.subtractMoney(payment);
                return payment;
            }
        }
    }
}
```

Notice that the Customer no longer has a '`getWallet()`' method, but it does have a getPayment method. Lets take a look at how the paperboy will use this:

Listing 5 – The New Paperboy Code

```
    // code from some method inside the Paperboy class...

    payment = 2.00;  // "I want my two dollars!"
    paidAmount = myCustomer.getPayment(payment);
    if (paidAmount == payment) {
        // say thank you and give customer a receipt
    } else {
        // come back later and get my money
    }
```

Why Is This Better?

That's a fair question... Someone who really liked the first bit of code could argue that we have just made the Customer a more complex object. That's a fair observation, and I'll address that when I talk about drawbacks, below.

The first reason that this is better is because it better models the real world scenario... The Paperboy code is now 'asking' the customer for a payment. The paperboy does not have direct access to the wallet.

The second reason that this is better is because the Wallet class can now change, and the paperboy is completely isolated from that change. If the interface to Wallet were to change, the Customer would have to be updated, but that's it... As long as the interface to Customer stays the same, none of the client's of Customer will care that he got a new Wallet. Code will be more maintainable, because changes will not 'ripple' through a large project.

The third, and probably most 'object-oriented' answer is that we are now free to change the implementation of '`getPayment()`'. In the first example, we assumed that the Customer would have a wallet. This led to the null pointer exception we talked about. In the real world though, when the paper boy comes to the door, our Customer may actually get the two bucks from a jar of change, search between the cushions of his couch, or borrow it from his roommate. All of this is 'Business Logic', and is of no concern to the paper boy... All this could be implemented inside of the `getPayment()` method, and could change in the future, without any modification to the paper boy code.

What Are The Drawbacks?

Above, I suggested that there are several drawbacks to this approach. Thats true in most things... when there are two ways of doing things, you can almost always find some reason for doing it either way. Lets talk about the potential drawbacks of this idiom, and see if they can be addressed.

The main one I already mentioned, the appearance that the 'Customer' object is getting more complex. This is partially true... Since we are not exposing the Wallet, we need a new method. If we were choosing to do this for some other type of object, there might be several methods we have to add to Customer, and Customer starts to become a class with many methods. We may potentially have to repeat a lot of method signatures that really belong to classes that we hold a reference to. This is another idiom called 'delegation'.

Is the Customer REALLY becoming more complex? In our first example, the paperboy code using the Customer also had to know about and manipulate the Wallet. This is no longer the case... If Customer has become MORE complex, then why are the clients now LESS complex? All the same complexity still exists, it is just better contained within the scope of each object, and exposed in a healthy way. By containing this complexity, we get all the benefits discussed above. So yes, we can conceed that Customer has gotten more complex The truth is, it's complexity that existed before in the code, and it may have existed in several places. Now it occurs once, and can be maintained in the same place that the changes that may break it occur.

There is another drawback with this idiom, and that is trying to apply it in every case that seems appropriate (the "I have a shiny new hammer so every problem looks like a nail" idiom). Lets assume our Wallet had been modified to contain a Driver's License object, and the person we are modelling had been pulled over by a police officer. Now, we don't want to hand him the Wallet, but we do want to hand them the object representing the Driver's License. The officer might not be happy just being able to ask "What is your driver's license number?". A '`getDriversLicense()`' method should return the real Driver's License object.

In Java, an AWT Component is a container for a LayoutManager. In this case, we would not want the component to 'hide' the LayoutManager from us... This is a 'strategy' pattern, and exposure of the object implementing the strategy is important. There are several other patterns where this is the case as well. We will see below some rules for determining when and how to apply this idiom.

You Too Can Sound Intelligent

I mentioned before that this concept has a name: This is called **The Law Of Demeter**.

Law Of Demeter

A method of an object should invoke only the methods of the following kinds of objects:

1. itself
2. its parameters
3. any objects it creates/instantiates
4. its direct component objects

So now, when you are talking to other people you work with, instead of saying 'You don't want the paperboy to know about your wallet', you can say 'See this code? This is a good opportunity to apply the Law of Demeter.' Then you get to explain it to them and sound intelligent. You might even get a pay raise or a consulting gig out of it.

While working on this paper, I was trying to determine 'why' it is called the Law of Demeter. I found several references, but none with a good explanation. The Law Of Demeter was first published in [Lieberherr +], and 'Demeter' is the name of a research project at Northeastern University [Demeter].

When and How to Apply The Law Of Demeter

So now you have a good understanding of the law and it's benefits, but we haven't yet discussed how to identify places in existing code where we can apply it (and just as important, where NOT to apply it...)

1) Chained 'get' Statements

The first, most obvious place to apply the Law Of Demeter is places of code that have repeated get() statements,

```
value = object.getX().getY().getTheValue();
```

as if when our canonical person for this example were pulled over by the cop, we might see:

```
license = person.getWallet().getDriversLicense();
```

2) lots of 'temporary' objects

The above license example would be no better if the code looked like,

```
Wallet tempWallet = person.getWallet();
license = tempWallet.getDriversLicense();
```

it is equivalent, but harder to detect.

3) Importing Many Classes

On the Java project I work on, we have a rule that we only import classes we actually use; you never see something like

```
import java.awt.*;
```

in our source code. With this rule in place, it is not uncommon to see a dozen or so import statements all coming from the same package. If this is happening in your code, it could be a good place to look for obscured examples of violations. If you need to import it, you are coupled to it. If it changes, you may have to as well. By explicitly importing the classes, you will begin to see how coupled your classes really are.

Law of Demeter and Design Patterns

As mentioned earlier, the Law of Demeter appears in many different design patterns. At its simplest, this is simply 'delegation'.

Consider the 'wrapper' patterns, Adapter, Proxy, and Decorator. Each of them hold onto an object, 'wrapping' it for some purpose. This is the Law of Demeter in the sense that it is 'hiding' the wrapped object, making the user use it through a different interface.

The 'Facade' pattern is another great example of the Law of Demeter at work. In the Facade pattern, several classes are hidden behind one API.

If you are working with Enterprise Javabeans, a common design pattern is to use Entity Beans for all your persistent storage, but control all access to these through Session Beans. That is, never let an EJB Client look up and reference the entity beans themselves. This is the Session Facade pattern.

So, the Law of Demeter is an important foundation concept. Not really a design pattern itself, but certainly an idiom with an intent that is often used in Patterns.

Sometimes Objects Are Just Containers

As in my examples above of the LayoutManager and the Driver's License, sometimes an object really is a container for another object. The Java Collections Framework and various Association patterns are good examples of this. Sometimes, you really DO want to give the officer your Driver's License. You may see examples that appear to break one of the rules above, but do so in a way you are comfortable with. Try to convert these to 'natural language' statements; this will help justify your design.

Conclusions:

I hope this paper has taught you something new. Hopefully, it almost seems like 'common sense'. If it does, that's good... But *understanding* it, giving it a name, and being able to communicate it to other people are all powerful.

I certainly didn't 'invent' this concept... I gave references earlier that you can find at the end of this paper if you want to read further on this subject. In particular, I would recommend [Hunt+Thomas], which is where I first learned the label for this concept. The Pragmatic authors talk about it in a much larger scope of 'Orthogonality' of code design.

References:

[Movie] *Better off Dead*
(Ok, maybe it's not a necessary reference for this paper, but if you've seen it, it sure would make the paperboy example a little more enjoyable...)

[Hunt+Thomas] Hunt, Andy and Thomas, Dave
The Pragmatic Programmer
Addison-Wesley Longman, Inc., 2000, p. 138-142

[Lieberherr +] Lieberherr, Karl. J. and Holland, I.
Assuring good style for object-oriented programs
IEEE Software, September 1989, pp 38-48

[Demeter] The Demeter Homepage:
http://www.ccs.neu.edu/research/demeter/