

OOP Principles in Action: Tell, Don't Ask

Maximizing the Advantages of Encapsulation for Orthogonality

Greg Willits • Feb, 2007

http://www.ldml.org/articles/tell_dont_ask.lasso

Tell, Don't Ask is a paradigm or style of object oriented programming which prefers that objectA tells objectB to do something, rather than asking objectB about its state so that objectA can make a decision.

Learning the best practices of OOP often requires that we unlearn many things we were originally taught using procedural programming. Tell, Don't Ask ("TDA") is one of those areas.

In the procedural programming paradigm, code is written to do some stuff, and it fetches or looks at the state of some data (the actual values of a specific variable or place of memory), and then makes a decision or takes an action. You could say that procedural programming pulls data into the logic stream in order to make decisions and get things done.

In object oriented programming, we want to do the opposite. It turns out that better designs come from having objects push data out, and having other objects get things done. Of course the buck can only be passed so much, as somebody has to actually do something, and we'll look at where those lines are. It can be a bit of a mind bender to invert your way of thinking, and it can be difficult to recognize when your code is not living up to TDA if you've spent a lot of time doing procedural programming

This article attempts to explain how to write in the Tell, Don't Ask paradigm and identify the benefits of TDA with examples. I assume you're familiar with most OOP terms, and that you're familiar with the basic structure of Lasso's custom

types ("ctypes") which I'll often refer to as classes, and member tags as methods, which are the more common OOP terms for these things.

How to spot an ask

I've come to believe that to explain TDA well, a real-world code example is needed to really expose the pitfalls of not using it, and thus the advantages of using it. Before we dig into a complex code example though, I wanted to show something small that illustrates what TDA looks and feels like in code. To do that we want to contrast some procedural code and some OO code that do the same thing.

For our example, let's consider a shopping cart calculation that determines whether free shipping applies. We'll have some order processing logic which has to determine that if the line item totals of the cart add up to more than 100 sheckles, then shipping is free.

To start, let's see what a procedural programming example looks like. In a procedural environment, we'd have some order processing logic code, and we probably have a map of all the cart line items.

A procedural example

Code Fragment 1 shows code that might look like what we want. It's a classic procedural process where we have a logic stream which in order to make some decisions (determine if the cart items adds up to more than 100), and to get some things done (calculate an order total), the logic fetches

Code Fragment 1:

```
// these are code Fragments, you'll
// have to mentally fill in some blanks

1 var:'orderTotal' = 0;
2
3 iterate: $cart->find:'items', $thisItem;
4 #cartTotal += $thisItem->'lineTotal';
5 /iterate;
6
7 if: #cartTotal > $freeShippingMinimum;
8 $cart->'shippingCost' = 0.00;
9 /if;
10
11 $orderTotal += #cartTotal;
12 $orderTotal += $cart->'shippingCost';
```

data or looks at data from data structures. In a procedural paradigm, this code is fine.

Rewritten with OO syntax

Now let's rework that code into an object oriented syntax, but use the same basic code flow which will give us an "ask" style. Let's assume the logic is inside one class called an order, and the cart items are in another class called cart. Code Fragment 2 shows these classes (obviously in incomplete form, and obviously in a not so robust design, but just ignore all that).

The workflow of Code Fragment 2 is that an object \$order calculates its total price by reading data from the \$cart object in very much the same way as Code Fragment 1 does. The \$order object is asking for data from the \$cart object.

Many OOP practitioners would argue this is not even object oriented programming, but is actually just procedural programming with object oriented wrappers. In this code, several OOP principles are being broken.

Let's identify where the ask is happening, then look at why it's a problem. Technically, there's more than one, but let's start with the one at lines 16-20. This is a classic example where the first object (\$order) is asking a second object (\$cart) information about its internal details in order to make a decision and then set a value in the second object. It gets even worse in this example, but we'll get to that later.

Code Fragment 2:

```
// these are code Fragments, you'll
// have to mentally fill in some blanks

1 define_type:'order';
2
3 local:'orderTotal' = 0;
4 local:'cartTotal' = 0;
5
6 define_tag:'orderTotal',
7   -required = 'cart';
8
9 local:'thisItem' = null;
10
11 iterate: #cart->'items', #thisItem;
12 (self->'cartTotal')
13   += #thisItem->'lineTotal';
14 /iterate;
15
16 if: (#cart->'shippingCost') == null
17   && (self->'cartTotal')
18     > $freeShippingMinimum;
19   #cart->'shippingCost' = 0.00;
20 /if;
21
22 (self->'orderTotal')
23   += self->'cartTotal';
24 (self->'cartTotal')
25   += $handlingFees;
26 (self->'orderTotal')
27   += #cart->'shippingCost';
28
29 /define_tag;
30 /define_type;
31
32 //-----
33 define_type:'cart';
34
35 local:'items' = array; // of maps
36 local:'shippingCost' = null;
37
38 // cheesy tag just to show the
39 // data structure of the cart line item
40 define_tag:'addItem',
41   -required = 'qty', // etc
42
43 (self->'items')->(insert: (map:
44   'qty' = #qty,
45   'sku' = #sku,
46   'price' = #price,
47   'lineTotal' = #qty * #price));
48
49 /define_tag;
50 /define_type;
51
52 //-----
53 var:'order' = (order),
54 var:'cart' = (cart);
```

Achieving orthogonality

Now that we see what ask code looks like, what is wrong with it? If it is just fine for procedural code, why isn't it fine for OO code?

One of the goals of object oriented code is to maximize code orthogonality. Orthogonal is a geometry term to describe a change in position along one axis results in no change of position on another axis. Something can move freely on one axis with no impact to the other(s). In software, this means chunks of code should be written so that changes in one chunk has no impact on the other chunks. In OOP, those chunks are classes. Changes to code in one class should not require a change in code of another class in order for the second class to keep working.

To achieve orthogonality, object oriented code starts with an idea called encapsulation. Encapsulation is the combination of data and logic in a common wrapper. That wrapper is a class. The class defines boundaries around a data structure and the code that should be used to manipulate it. Adhering to rules which say that the modification of any data within a wrapper must be done through the logic contained in that same wrapper helps promote orthogonality.

If the internals of one class is changed, that should not affect how the internals of another class works. (You're probably already thinking of exceptions to that, but we'll get to that later).

Looking at Code Fragment 2 again, if we needed to change the data structure of the cart class and store shipping costs per line item, we might want to change the name of the instance variable `shippingCost` to `totalShippingCost` to better reflect what it really is. That change in the cart class would cause the order class to break. These two classes as written are not orthogonal.

Following through on this line of reasoning, we can conclude that in Code Fragment 2, the `$order` object has no business peering into the details of the `$cart` object let alone making decisions on what the internal data contents of `$cart` should be. Each object is supposed to do that for itself, and if `$cart` changes, `$order` breaks.

Still, we need `$order` and `$cart` to share information. Obviously all code chunks cannot exist in total isolation, so there has to be a way for one code chunk to communicate with another code chunk. In order for code in one class to cause

or potentially cause a change in the data of another class, the two classes have to have a conversation about it.

In object oriented code, this is called sending a message. It is not by accident that the term is "sending" a message—sounds like Tell, Don't Ask doesn't it? Sending a message is nothing more than using a method and parameters to send an object a command to do something along with some data to work with if needed.

Rewriting the ask into a tell

So how do we fix lines 16–20 to improve orthogonality and function as a tell? We want the cart class to be the one changing the value of the `shippingCost` instance variable, so we move that decision making logic into the cart class.

To do that, we need a method in the cart class which we'll call `adjustShippingCost`. And if that method will be doing the work, it will also need to know what the current order total is, so that has to be accepted as a parameter. The method ends up looking like Code Fragment 3.

We also need to update the the code in the order class to use this new method. So, lines 16–20 in Code Fragment 2 end up looking like Code Fragment 4.

Code Fragment 3:

```
define_tag: 'adjustShippingCost',
  -optional = 'orderTotal';

if: #orderTotal > $freeShippingMinimum;
  self->'shippingCost' = 0.00;
/if;

/define_tag;

// we'll assume $freeShippingMinimum
// remains an environment var for now
```

Code Fragment 4:

```
16 #cart->(adjustShippingCost:
17   -orderTotal = (self->'cartTotal')
18
19
20
```

Do you see how this now changes the *asking* of `$cart` about it's inner details to *telling* `$cart` to do something?

Now, technically there's some fishy things about `adjustShippingCost` too, but for now we'll accept that it demonstrates how to refactor (a fancy word for rewrite) code from an ask to a tell.

You should also be able to see that we could change the internal working of the method `adjustShippingCost` without breaking the order class. That method could have code added to deal with individual line items, but as long as we keep the name of the method the same, and continue to allow the passing of the `orderTotal` parameter, the order class will continue to work.

This change has not only converted our code to follow the Tell, Don't Ask principle, but it has made our code more orthogonal, and less prone to damage from changes. You gotta like that. Let's look at this orthogonal thing a little more.

Program to an interface, not to an implementation

We identified that a significant problem with the code in Code Fragment 2 is that breaks the intent behind encapsulation. Several authors have coined principles which all say the same thing: one object should not know about the internal details of another object. To cope with this, developers talk about programming to an interface not an implementation.

What's an interface? Technically an interface is simply where the boundaries of two discernible things meet and interact. I talked about classes as being wrappers for data and logic chunks. Where do the wrappers of two classes meet and interact? Methods. Depending on the language there can be more than just methods. In Lasso, we can also directly access a custom type's instance vars like was done in Code Fragment 2.

The *interface* of a class is defined by the access points we have to send or pull information from that class or its instances (objects).

The *implementation* of an object is the internal details of the instance variables, their data structure (i.e. map vs. array of pairs), and the algorithms used inside for data manipulation. As far as all other objects are concerned, these details should be unknown.

If we look at Code Fragment 2 lines 11–14, we're pulling data straight out of the `$cart` object's internal data structure. That's programming to an implementation. I, as the programmer, know how the internal data structure of the cart class was implemented, and I've infused that knowledge inside the order class.

So, now, if somebody were to change the cart class data structure, perhaps by renaming `lineTotal` to `lineItemTotal`, then the order class is going to fail because an implementation detail of another class that it depends on has changed.

Right about now, you might be thinking, "well, yeah, of course if I change something, the things that depend on that will have to change too. That's obvious." That's inevitable in procedural programming. It is *not* inevitable in object oriented programming when done "right."

We saw earlier how the code for lines 16–20 could be rewritten so that a change in cart did not cause order to break.

In the rewritten code, the interface would be defined to contain the `adjustShippingCost` method which is documented to accept an `orderTotal` parameter. This would be the *public* interface. Public meaning it is OK for other classes to know and use those details to interact with the cart class.

Regardless of whether we as programmers know the details of cart's innards or not, if we stick to using only `adjustShippingCost` to affect the shipping cost, then we are programming to the interface. Other programmers can change the internal implementation details, but as long as they keep that method with that parameter working consistently, we're happy.

Doesn't that just move the target though? All we've really done is moved what has to stay the same, yes? Instead of not allowing anyone to change the instance vars names, we say you're not allowed to change the method names. So, what's the difference?

In a way, all that might be true, but hopefully if you think it through, you'll see that now, the stuff we usually want to change (the details) can be changed without causing damage, and the stuff that's easier to keep the same (the interface) is what we're agreeing to keep the same. Yes, you could say all we have done is moved the target, but to stretch the metaphor further, we've moved it a lot closer and made it easier to hit.

TDA and reusability

I had intended the above order and cart to be just a quick example of what Tell, Don't Ask looks like, but it also covered a lot of why following Tell, Don't Ask is a good thing. It helps force code into a more orthogonal structure, and orthogonal code helps protect a system from widespread damage from changes in details.

I wanted to illustrate one more advantage to all this encapsulation and orthogonal stuff, and that advantage is code reusability.

If the detailed implementations of classes can change without changing interfaces, it stands to reason that multiple classes could be written with the same interfaces but have different implementations. I'll use the example of a user authentication system to demonstrate this.

In such a system we have users and their permissions. One of the problems with writing a user management system is the need for different permissions systems. Perhaps one application

needs only the simple permissions concepts used by the *nix file systems. Perhaps another application needs very granular controls over specific data and/or user interface elements. It would be advantageous if we could write a system where the permissions components could be swapped out to be best suited to the application at hand.

Thinking about orthogonality, we would want the user code to concern itself with attributes of the person (name, email address, password), and some other code to deal with the details of the permissions. We would need an interface that could be common to multiple permissions systems.

Code Fragments 5, 6, and 7 give us enough code to see how that might be done. We can use this code to illustrate where the concept of Tell, Don't Ask has been implemented to facilitate encapsulation, orthogonality, and reusability.

Looking at the user class in Fragment 5, we see that a user has certain properties — one of which is userPerms. This property is defined as an

Code Fragment 5:

```
1 define_type:'user', -prototype;
2
3 local:
4   'id'          = string,
5   'firstName'  = string,
6   'lastName'   = string,
7   'userPerms' = null;
8
9 //-----
10 define_tag:'onCreate',
11   -required = 'email',
12   -required = 'pswd',
13
14   self->'id'          = // field userID from a queryObject
15   self->'firstName'  = // field userNameF from a queryObject
16   self->'lastName'   = // field userNameL from a queryObject
17
18   (self->'userPerms') = (userPermissions: self->'id');
19
20 /define_tag;
21 //-----
22 define_tag:'isAuthorizedTo',
23   -required = 'permission';
24
25   return: (self->'userPerms')->(getAuthorizationStateFor:'permission');
26
27 /define_tag;
28 /define_type;
```

Code Fragment 6:

```
29 define_type:'userPermissions', -prototype;
30
31 local:
32   'userID'      = string,
33   'permissions' = map;
34
35 //-----
36 define_tag:'onCreate',
37   -required = 'userID';
38
39   self->'userID' = #userID;
40
41   // assume we did a db query to acquire data for a structure
42   // that after being transcribed ends up looking like this:
43
44   self->'permissions' = (map:
45     'news' = (map:
46       'add'   = true,
47       'update' = true,
48       'delete' = false));
49 /define_tag;
50 //-----
51 define_tag:'getAuthorizationStatusOf',
52   -required = 'permission';
53
54   local:
55     'module'   = (#permission->split->'_')->get:2;
56     'privilege' = (#permission->split->'_')->get:1;
57
58   return: ((self->'userPerms')->find:#module)->find:#privilege;
59
60 /define_tag;
61 /define_type;
```

Code Fragment 7:

```
// the instantiation of a user in a controller

62 var:'thisUser' = (user:
63   -email = $loginEmail,
64   -pswd  = $loginPswd);

// the use of the user's permissions in a view

65 if: $thisUser->isAuthorizedTo:'delete_news';
66   // show a delete button
67 /if;
```

instance of a class called `userPermissions` when a user object is created (line 18). For now we don't need to know the details. We further see in line 25 that when the user object is checked for a certain permission we defer to the `userPerms` object's

interface to find out the details. By using a method to access the data we want rather than accessing the permissions data structure directly, we have preserved encapsulation and hopefully some orthogonality too.

Code Fragment 6 shows us a little more detail about how we go about getting the actual permission value. You should be able to see that we could replace the entire internal data structure of `userPermissions` and the code of the method `getAuthorizationStatusOf` but as long as we leave the name of the method alone, and accept the name of the permission being sought, then the permissions system could be replaced with a number of different implementations.

What that does for us, is it let's the user class be reusable for a number of user management systems. In Code Fragment 7, our application-specific code in line 65 could be written for another permissions system like this:

```
if: $thisUser->isAuthorizedTo:'delete';
```

which would require a different implementation of the `userPermissions` class, but all the `$user` code remains the same.

Now, I don't want to give the impression that reusability is intended for big system modules like user management. Even the smallest of classes can be abstracted for reusability, and writing code for orthogonality using Tell, Don't Ask as a means to accomplish that can help even classes that perform small utilitarian functions.

Orthogonality at multiple levels

Now, after all this Tell, Don't Ask lecturing, I hope you've caught that the method `isAuthorizedTo` follows an Ask paradigm!

Asking if a user is authorized to do something seems like a very reasonable thing to do. The source code reads well, we aren't using that ask step to modify anything, so is there anything wrong with that?

Some practitioners say that we should avoid asking an object about it's state whenever possible. So, how can this functionality be written in a TDA manner? Code Fragment 8 gives a before and after version of rewriting the permission acquisition in a Tell paradigm.

Hmm. Technically, the wording follows a Tell vs an Ask paradigm, but it's a lot longer. Did we really accomplish anything meaningful? Either way we extracted a value from the user object's state. We changed the grammar, but did we

Code Fragment 8:

```
// instead of writing this

if: $user->isAuthorizedTo:'delete_news';
    // show the delete button
/if;

// it could be written like this:

local:'deleteNewsGranted' = boolean;
$user->(putAuthorizationOf:
    -permission = 'delete_news',
    -intoLocalVar = 'deleteNewsGranted');

if: #deleteNewsGranted;
    // show the delete button
/if;
```

change anything meaningful to the purposes of orthogonality? Yes!

Where? It's a trick question. Look at what gets done inside the if clause. I'm apparently modifying a view based on a conditional. If you're writing code to the MVC pattern, you know you want to keep all logic out of the view so that the logic and the view can be orthogonal to each other. To do that, the conditional logic itself should be kept out of the view code. In this case that would be the step of fetching the user state. If we use the first version, our view knows too much detail about the exact implementation about why a delete button should be drawn or not. If we change the user variable name, or any of the business logic in that if statement, the view is going to break. In other words, the interface between the controller and view break our desire to have code chunks orthogonal.

To make the view more orthogonal, we hide the implementation of the permission behind an abstraction which is the `#deleteNewsGranted` variable. Now it doesn't matter how we determine the value for that variable, the view code remains undamaged. Of course we could achieve the same thing with code like this:

```
if: $user->isAuthorizedTo:'delete_news';
    local:'deleteNewsGranted' = true;
else;
    local:'deleteNewsGranted' = false;
/if;
```

but that creates a problem. We already have a value in `$user`, so why not transfer it directly? The

above code interprets the value, and that interpretation could get us into trouble someday.

We could also do this:

```
local: 'deleteNewsGranted' =  
    $user->isAuthorizedTo: 'delete_news';
```

but that reads a little funny. It reads like the user is being told it is authorized to do something. Without the if keyword, it is not clear that this

assignment is based on a question. So, by rewording it to a Tell, the code reads better.

You may have noticed the style in which I named the method and parameters so they read like a sentence. Take all the syntax out, and this is what you have: putAuthorizationOf permission delete_news intoLocalVar deleteNewsGranted. Pretty easy to understand what's going on there. (I picked this style up from the Mac's Cocoa API code style).

Closing thoughts

One thing you may have noticed is that these ideas of Tell, Don't Ask, orthogonality, encapsulation, and resuability are all quite intertwined. And that's true for virtually all object oriented software principles. So many of them of are just different ways to look at the same thing.

If you study and really understand encapsulation, that can help you write more orthogonal code. If you really understand Tell, Don't Ask, that will help you understand the objectives of encapsulation.

As with so much in OOP, sometimes you have to study them all a little bit to have one of them suddenly jump out and makes sense. Then you go back to the others with fresh eyes.

One of the points I put off in the above text is exceptions. I wrote a lot of "should" statements like, "If the internals of one class is changed, that should not affect how the internals of another

class works." Will there be exceptions to this? Of course. There are no absolutes, least of all in software. The ideas are not about reaching some perfection of purism. They are guides that if followed as much as possible will lead to software that is more more manageable—less brittle in the face of changes and extensions, and easier to maintain and reuse. How will know when an exception is OK? When you've made a diligent effort to stick to the principles, and they just won't work to do what you need to do.

What happens when you screw up? Catholics have confession, software developers have refactoring. The more you learn about OOP, the more you'll discover code that you thought was great, but then see how it could be better. This is normal, and a good thing. It means somewhere along the road, you've learned something! 🗨