Extracted from:

Domain-Driven Design

Using Naked Objects

The Pragmatic Programmers

Domain-Driven Design Using Naked Objects

Dan Haywood

Edited by Susannah Davidson Pfalzer

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-44-1

ISBN-13: 978-1-934356-44-9

Printed on acid-free paper.

P1.0 printing, December 2009

Version: 2010-1-16

Run your application to make sure it all works as expected.

So much for validation. However, implementing such rules still allows our user to *attempt* to modify the property or collection or to *attempt* invoke the action. What if we said that once it's set, you shouldn't be able to modify the Car's RegistrationNumber *at all*? For this, we need to make the property read-only. In Naked Objects parlance, we *disable* it.

6.2 Disabling Class Members

Disabling a class member is a stronger constraint than validation; it prevents the property or collection from being modified or an action from being invoked. . . period. Typically this is because using the class member doesn't make sense given the current state of the domain object. For example, if one action is called "go" and the other "stop," then presumably only one is active at a time.

In terms of the user interface, you can think of a disabled class member as being grayed out, and you might want to describe it in these terms when demonstrating and discussing the domain model with your domain experts. Indeed, in non–Naked Objects applications you have built, you've almost certainly implemented this responsibility within the presentation layer. But this is an area where Naked Objects has strong opinions: such responsibilities should reside in the domain layer, not the presentation layer. In any case, the discussion is moot; we implement the rule on the domain object because in Naked Objects there is literally nowhere else to put it!

As for all the business rules, we can disable class members either declaratively or imperatively. Let's use CarServ to look at each.

Disabling Declaratively

Preventing our user from changing the Car's RegistrationNumber property declaratively really couldn't be much simpler:

Download chapter06/Car-RegistrationNumber-disabled.java

```
@Disabled
public String getRegistrationNumber() { ... }
```

Make this change, and then run the application. As shown in Figure 6.1, on the next page, you shouldn't be able to modify the property.

The @Disabled annotation can also be applied to collections and to actions. Indeed, it is one of the most commonly used annotations, and

Figure 6.1: Disabled properties cannot be edited.

there a couple of places in CarServ where we ought to use it. Cars are now created only by Customers (as opposed to using the CarRepository). However:

- Currently we *can* remove a Car from a Customer's Cars() collection, leaving an orphaned Car with no owner. We can fix this by annotating the collection as @Disabled.
- For the other side of this relationship, Car's OwningCustomer property, we should also annotate this property as @Disabled (and remove the redundant @Optional annotation).

Similarly, we should make the Car-Service bidirectional relationship read-only, by annotating both Car's Services collection and Service's Car property as @Disabled.

In a similar vein, it doesn't really make sense to change the Model of a Car once it has been created. So, also add the annotation to the Car's Model property.

Go ahead and apply all these changes and check that this works. Then we'll move onto the imperative approach.

Disabling Imperatively

To disable imperatively, we write a disableXxx() supporting method, analogous to the validateXxx() methods we saw for validation. The framework looks for the presence of this method and, if it exists, will call it first to determine whether to make the property or collection modifiable/action invokable.

To demonstrate this, let's consider the Customer's deleteCar() action. It doesn't make much sense to try to invoke this for a Customer that has no Cars, so we should disable it in these cases.

Figure 6.2: The object's state may disable members.

That's easily done:

Download chapter06/Customer-deleteCar-disable.java

```
public String disableDeleteCar() {
    return getCars().size() == 0? "No cars to delete": null;
}
```

Add this code, and try your revised application (use the nopdis template). As shown in Figure 6.2, the action should be disabled.

As I already mentioned, we can also use a disableXxx() method for properties and collections. To see this in action, let's add a rule to prevent the Customer's Notes property from being updated unless the Customer has at least one Car. It's a bit contrived as an example but easy enough to implement:

Download chapter06/Customer-Notes-disable.java

```
public String disableNotes() {
    return getCars().size() == 0?
        "Can only add notes for customers with cars":null;
}
```

Disabling is stricter than validation because although validation *might* let a change through (so long as the value you provide is valid), disabling will *never* do so. Our final category of business rule is even stricter—not being able to see the class member in the first place.

Factor Out Ruthlessly

Both our deleteCar() action and our Notes property now have a rule that has to do with there being no Cars in the Cars collection. To better express intent, factor this out:

Download chapter06/Customer-Notes-disable-refactored.java

```
public String disableDeleteCar() {
    return doesntOwnAnyCars()?  "No cars to delete": null;
}
...
public String disableNotes() {
    return doesntOwnAnyCars()?
        "Can only add notes for customers with cars":null;
}
private boolean doesntOwnAnyCars() {
    return getCars().size() == 0;
}
```

It's a simple change but a great improvement!

6.3 Hiding Class Members

The strongest of our three rules is the first one that the framework checks: should the class member even be visible?

For disabling class members, I made the observation that you may be more accustomed to implementing that type of rule in the presentation layer. If that's the case, then you almost definitely will have implemented this rule in the presentation layer too. Even so, it too is fundamentally a domain responsibility. For example, when an object transitions between two states, some of its members might be relevant in only one of those states. If the user selects to pay by credit card, then the properties for capturing the credit card details are relevant (and so should be shown); if they pay by cash, then these same properties are irrelevant (and so should be hidden).

Having said that, the most common reason for hiding a class member is because it is really part of the "inner workings" of the object, not to be exposed in a Naked Objects viewer. This occurs with methods that are intended to be called programmatically but that—for whatever implementation reason—happen to have **public** visibility.

As for the other two rule types, we can hide the class member either declaratively or imperatively. Let's look at the declarative case first.

Hiding Declaratively

In Section 5.4, *Adding Finders to Repositories*, on page 105, we implemented a version of the findByName() action on the CustomerRepository. Let's take a look at this code again:

Download chapter06/CustomerRepository-findByName.java

```
public List<Customer> findByName(
        @Optional
        @Named("Last Name")
        final String lastName,
        @Optional
        @Named("First Name")
        final String firstName) {
    return allMatches(Customer.class, new Filter<Customer>() {
        public boolean accept(final Customer customer) {
            return matches(customer, firstName, lastName);
        }});
}
// ...
private static boolean matches(
       final Customer customer,
       final String firstName, final String lastName) {
    return nullSafeEquals(customer.getFirstName(), firstName) ||
           nullSafeEquals(customer.getLastName(), lastName);
}
private static <T> boolean nullSafeEquals(final T s1, final T s2) {
    return s1 == null && s2 == null ||
           s1 != null && s2 != null && s1.equals(s2);
}
```

That matches() method doesn't look right on CustomerRepository. Far nicer would be for Customer to do the matching itself. Let's change CustomerRepository first:

Download chapter06/CustomerRepository-findByName-refactored.java

```
public List<Customer> findByName(
        @Optional
        @Named("Last Name")
        final String lastName,
        @Optional
        @Named("First Name")
        final String firstName) {
    return allMatches(Customer.class, new Filter<Customer>() {
        public boolean accept(final Customer customer) {
            return customer.matches(firstName, lastName);
        }});
}
```

Subtractive Programming

One of the main responsibilities of a business analyst is to identify business rules, documenting them in specifications documents or in UML, or even semiformally using the Object Constraint Language (OCL). Meanwhile, the developer's responsibility is to implement the functionality up to the point where the constraints are... but no further!

If there's a gap between what the application *can* do and what the spec says it *mustn't* do, then we're left wondering: is this an omission in the application or an omission in the spec?

With Naked Objects this problem doesn't arise. We start off with an application that has all degrees of freedom, just like a UML diagram with no constraints. Then, as we analyze and explore our domain and identify the constraints, we can write code just as we might have once added an OCL constraint.

I call this *subtractive programming*: adding constraints subtracts functionality. Putting aside the fact that Naked Objects is a highly productive development environment, this is also a much more *honest* way of developing software.

And now let's move the matches() method to Customer:

Download chapter06/Customer-matches.java

```
@Hidden
public boolean matches(final String firstName, final String lastName) {
    return nullSafeEquals(this.getFirstName(), firstName) ||
           nullSafeEquals(this.getLastName(), lastName);
}
private static <T> boolean nullSafeEquals(final T s1, final T s2) {
    return s1 == null && s2 == null ||
           s1 != null && s2 != null && s1.equals(s2);
}
```

To ensure that this new **public** method of Customer doesn't appear as an action in the user interface, we annotate it as @Hidden. Run your application to make sure. To double-check, temporarily comment out the @Hidden annotation and see the action appear.

Let's now look at the imperative method of hiding members.

Hiding Imperatively

Suppose we'd like to capture feedback from our most valuable Customers, which we'll (slightly naively) define as those that own two or more Cars. To do this, let's define a new (multiline, optional) Feedback property on Customer, using the nop template. You should end up with the following methods:

Download chapter06/Customer-Feedback.java

```
private String feedback;
@MultiLine(numberOfLines = 5, preventWrapping = false)
public String getFeedback() { ...  }
public void setFeedback(final String feedback) { ... }
```

Now let's implement the business rule. Since there's no point in displaying the Feedback property for Customers that don't qualify as being valuable, we'll just hide it (use the nophid template):

Download chapter06/Customer-Feedback-hide.java

```
public boolean hideFeedback() {
    return !isValuableCustomer();
}
private boolean isValuableCustomer() {
    return getCars().size() >= 2;
}
```

Note that hideFeedback() must be **public** for the framework to call; on the other hand, because isValuableCustomer() is **private**, it won't appear in the UI. If we wanted to (and there's probably a good argument for this because it does sound like it is part of the *ubiquitous language*), we could make the latter **public** too; it would then appear as a derived read-only property.

Try adding this code and then adding and removing Cars to your Customers. You should find that when they have two or more Cars, then the Feedback property magically appears; otherwise, it will be hidden, as illustrated in Figure 6.3, on the following page.

Note that whereas the disableXxx() and validateXxx() supporting methods return the reason as a String, the hideXxx() method simply returns a boolean. All that the framework needs to know is, should the class member be displayed or not?

That takes us through the three main categories of business rules that Naked Objects supports. However, the declarative forms of disabling and hiding (@Disabled and @Hidden) are a little more powerful than I let on. Let's look at that now.

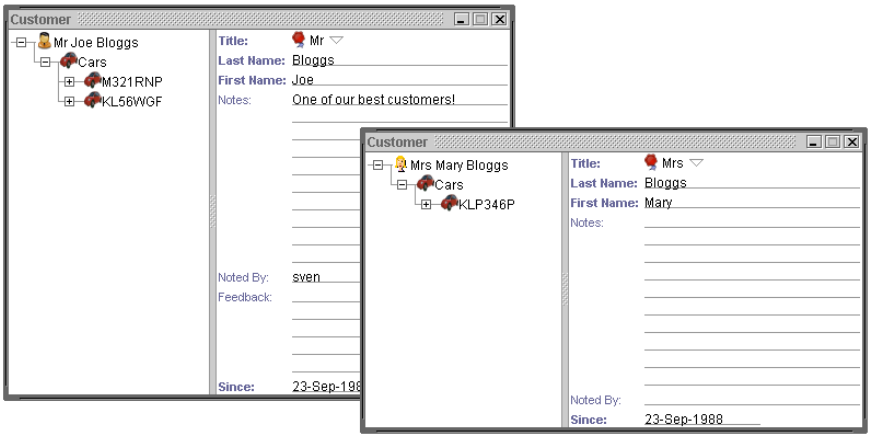

Figure 6.3: The object's state may hide members.

6.4 Declarative Rules and the Object Life Cycle

Very often class members can be used or are visible dependent on the object's state only, in particular whether the object is persistent. A property may be disabled if the object is still unsaved; conversely, an action might be visible only when the object has been saved.

Because this is a common requirement, the @Disabled and @Hidden annotations both provide support for this. Each optionally takes an attribute—an instance of the When enumerated type (also in the applib). The default value is When.ALWAYS, so if omitted, we are stating that the class member should be disabled or hidden at all times. The other values of the When enum, though, allow us to qualify when these annotations apply, based on whether the object is persistent or not.

For example, imagine we were building a security management system where we capture Users as domain objects. When first created, the administrator might select the username and enter an initial password; to do that, they will obviously need a field in the UI to fill in. However, once the new User object has been persisted, we almost certainly don't want the password visible, not even to the administrator. To capture this rule, we would use an @Hidden(When.ONCE_PERSISTED). Conversely, an action to *change* the password would be annotated as @Hidden(When.UNTIL_PERSISTED).

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Domain-Driven Design using Naked Objects Home Page
http://pragprog.com/titles/dhnako
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/dhnako.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)