Technical Explorations

by Keith R. Bennett

Design by Contract, Ruby Style

Jun 15, 2011 • keithrbennett

I recently encountered a situation in which I was writing data to a key/value data store. There was a code layer that insulated the business logic layer from the data store internals. I wanted to be able to unit test this business logic without needing access to the data store.

I could mock the data access layer, but I wanted something more functional – something that would *behave* like the data store layer, and possibly even be used to test it. I decided to write something mimicking the production data store layer that used a Ruby Hash for storage. How could I use the language to help me know that I had faithfully reproduced all the functions in the original object?

Dependency Inversion

Dependency inversion is defined in Wikipedia as:

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions.

Using this principle, I rearranged the business layer so all it needed to know about its data access object was the method signatures and nothing else, not even its type. The data access object was passed into the business object’s constructor rather than the business object instantiating it directly (and thereby needing to know its type). That opened the door to alternate data access strategies, including a trivial implementation for unit testing.

Java Interfaces

If you’ve coded in Java, you’ll recognize that this principle is what Java interfaces are all about. Using Java interfaces involves the following:

1) defining the interface, that is, the signatures of the methods that need to be defined by implementations of that interface.

2) defining object references in the code that refer to the implementations as the interface type rather than the implementation type.

3) creating implementations of that interface.

Ruby’s Duck Typing

Ruby uses duck typing, which eliminates the need for #1. Unlike Java with its compile time checking, Ruby doesn’t verify the existence of the method before it’s actually called – it trusts the method to be there, and calls it, raising an error if it doesn’t exist, and is not handled by a method_missing function. In addition, in Ruby, references are not typed, eliminating the need for #2.

In this way, Ruby makes dependency inversion effortless and automatic.

Many Ruby developers believe the Java interface approach to be obtrusive and verbose. However, there can be issues with Ruby’s permissive approach as well. For example, while relying on duck typing is convenient, we want noncompliant implementations to fail quickly and gracefully. Using duck typing alone, the absence of a method will be detected only when it is called – and in the absence of adequate unit testing (which could be beyond one’s control), this could result in premature termination of a long job, or the detection of the error months after its introduction.

In addition, it would be helpful to show the user *all* the methods that are missing, not just the one that was called. Duck typing will not do that; it will throw a NoMethodError only for the method called.

There are other problems with relying exclusively on duck typing. Where can the author of a new implementation of the contract find all the methods that need to be implemented? Maybe one of the implementations? Which one? And can we trust that the author of that implementation made private all the methods that are not needed in the public interface?

Making the Promise in Executable Code

For the same reason that self documenting code is better than comments, these required methods should be expressed as executable code rather than included in some documentation file. Further, this promise is such an important part of the class’ existence, that it makes sense to include it somewhere inside the class definition. (The promise could live here even if we only validate the class’ adherence to the contract during unit testing.)

Sometimes this will be overkill – if we have complete control over our source code, then we can unit test to our heart’s delight, and guarantee that only thoroughly tested code makes it to production. However, there are some situations in which more verification would be helpful.

An Illustrative Example

Let’s say we’re writing a flexible data processing framework that allows users to plug in their own implementations of actions. As a trivial example, let’s say that one of the services of this framework is encrypting strings, and we want the users to be able to provide their own encryption strategies.

Let’s assume there are lots of these plugins used by the system, some provided by the software authors and others provided by customers and third parties. Let’s also assume that we want to verify that these plugins implement the required methods before starting the job. (The job may take a long time, use expensive resources, be especially critical, etc.)

What is the appropriate Ruby way to provide some kind of confidence in these plugins? Certainly, we should be providing implementation authors with comprehensive test suites that test much more than the presence of required methods. However, let’s go further than that. Let’s provide a mechanism to enable this verification at any time, not just in unit tests.

Here is a class that can be used to verify the presence of instance methods (also at https://gist.github.com/1034775):

```
# Shows a crude way to enforce that a class implements
# required methods.  Use it like this:
#
# class OkClass
#   def foo; end
#   def bar; end
#   def baz; end
# end
#
#
# class NotOkClass
#   def foo; end
# end
#
#
# verifier = InstanceMethodVerifier.new([:foo, :bar, :baz])
# verifier.verify('OkClass')
# verifier.verify('NotOkClass')
#
# $ ruby instance_method_verifier.rb
# instance_method_verifier.rb:22:in `verify': Class NotOkClass is missing the following required functions: bar, baz (RuntimeError)
#       from instance_method_verifier.rb:42

class InstanceMethodValidator

  attr_accessor :required_function_names


  def initialize(required_function_names)
    @required_function_names = required_function_names
  end


  # klass can be either a class name or a class object.
  def validate(klass)

    if klass.is_a? String
      klass = Kernel.const_get(klass)
    end

    missing_function_names = required_function_names - klass.public_instance_methods

    unless missing_function_names.empty?
      raise Error.new("Class #{klass} is missing the following required functions: #{missing_function_names.join(", ")}.")
    end
  end


  class Error < RuntimeError
  end
end
```

The contract below (also at https://gist.github.com/1034784) lists the required methods. It's also a good place to provide information about these methods.

```
>require 'instance_method_validator'

module EncryptionContract

  def required_function_names
      [
      # The purpose of this function is self-evident, but in other
      # cases the methods would benefit from some explanation
      # that would be helpful to authors of new implementations,
      # such as defining correct behavior, format and data type
      # of arguments and return values.
      #
      # encrypt a string
      #
      # arg: the string to encrypt
      # returns: the encrypted string
      'encrypt',

      # decrypts a string
      #
      # arg: the string to decrypt
      # returns: the decrypted string
      'decrypt',          # (encrpyted string)
    ]
  end


  def validate_contract(klass = self.class)
    InstanceMethodValidator.new(required_function_names).validate(klass)
  end
end
```

Now let's write a naive example implementation. Having the above code makes it trivial for the implementer to test that all required methods are implemented. Here is the rspec (also at https://gist.github.com/1034785):

```
>require 'rspec'
require 'reverse_encrypter'

describe ReverseEncrypter do

  it "should implement required methods" do
    lambda { subject.validate_contract }.should_not raise_error
  end
end
```

Here is the implementation itself (see also https://gist.github.com/1034788):

```
>require 'encryption_contract'

# Naive and totally lame encrypter that just reverses the string.
class ReverseEncrypter

 include EncryptionContract
```

```
  def encrypt(s)
    s.reverse
  end

  def decrypt(s)
    s.reverse
  end
end
```

In a pluggable framework as described above, it can be helpful to validate all “foreign” (that is, not provided by the framework authors) components as thoroughly as possible. The approach above enables the validation to be done at will, either in unit tests or at runtime. It’s a trivial function call and takes virtually no time to execute.

Another benefit of using this approach is that it naturally results in more thought given to the public interface. Do we really need all those methods? Do I need to change any method names so they don’t hint at implementation? Do I need to use a higher level object in my calling code to better insulate myself from the implementation?

Conclusion

This approach is no substitute for thorough unit testing. On the contrary, it is an *aid* to unit testing, and more.

Omitting a function is just one of an infinite number of ways to screw up an implementing class. This approach doesn’t verify that the arguments to the function are consistent across implementations, and that is very important too. Nor does it help test behavior. Nevertheless, given that limitation, it has the following benefits. It:

- formalizes the contract in executable code, where it is far more likely to be current
- provides an authoritative source of that contract
- promotes thinking more about the public interface
- aids in writing a new implementation class
- makes testing implementations DRYer; required method names are in one place rather than multiple unit tests, one for each implementation, so tests are more likely to be present, and more likely to be correct
- can be validated outside of unit tests

I think that’s pretty worthwhile. That said, I’d be interested in better ways. Feel free to comment on this article.

- Keith

Published with GitHub Pages