ASpec: Unit Testing Using Aspect-Oriented Programming

Matúš TOMLEIN

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies
Ilkovičova 2, 842 16 Bratislava, Slovakia
`matus.tomlein@gmail.com`

Abstract.
We often need to create an artificial environment in order to run our tests in a kind of a sandbox. Many times this requires altering the system under test to make it more testable. That often leads to excessive abstraction in our classes and a long and unclean code. A better way to approach this is using a mocking framework, like Mockito in Java or RSpec in Ruby. Our goal is to deliver the features of a mocking framework, with some additional benefits, in a simpler way, using aspect-oriented programming. We introduce ASpec, an alternative approach that uses aspects to alter the system under test. ASpec enables stubbing methods, raising exceptions, tracking method invocations and their arguments and testing the Liskov Substitution Principle. In this paper, we describe ASpec and compare it to other solutions.

1 Introduction

One of the biggest struggles when writing unit tests is to ensure that our tests run isolated from the external environment, independently from each other and that we can control them.

To run our tests isolated from the external environment means to get the same result each time we run them, no matter what the configuration we run them under is. For instance, our tests should pass even if there is no internet connection available.

Tests should be independent of each other so that they give the same results unaffected by the order they are run in. This basically means that tests should not leave any lasting effect on the system and should not rely on any previous state to pass.

We also need to be able to configure the system under test to make it behave like we expect it to. For instance, we might want to make the system access a test database server instead of the development server without making a change to the system under test.

We can implement the mentioned requirements in several ways. Perhaps the trickiest one is to adapt the system under test ourselves so that we can test it appropriately [1]. That means creating interfaces or abstract classes that can be replaced by objects of mock classes when they are passed to the tested methods. The tricky part about this is that it can lead to excessive abstraction. Too many abstract classes and interfaces in our code can make it much longer and also less readable.

In some languages, we can also make use of metaprogramming to define additional methods or replace existing methods in the system under test. However, this approach is quite cumbersome and requires additional knowledge of metaprogramming.

A different approach is to use a mocking framework like Mockito[1] or RSpec[2] to do the mocking for us. The advantage of these frameworks is that we don't need to modify the system under test to make changes to its methods or classes. We can create mock objects with a modified behaviour directly in our tests and pass them to the tested methods.

The task of modifying the system under test can also be a perfect use for aspect-oriented programming (AOP). AOP addresses the problem of separation of concerns and enables modifying the standard behaviour of a program without making changes to it directly. We introduce a simple Ruby library, ASpec, that uses AOP to provide the features of a mocking framework with some additional benefits in a simpler way. Since the knowledge of AOP is not common among programmers, ASpec provides an API that doesn't require any knowledge of AOP to make use of its functions.

In this paper, we will describe the features of ASpec and compare it to other mocking frameworks. We will also describe several testing frameworks that are based on aspect-oriented programming.

2 Implementation and Use

ASpec is implemented in Ruby. The only requirement of ASpec is the Aquarium ruby gem, which is a toolkit for aspect-oriented programming.

To use ASpec, simply download and include the file *aspec.rb* from the ASpec source in a Ruby testing script. It is recommended to use ASpec in combination with a testing framework, e.g. RSpec.

3 Features of ASpec

In this section, we will describe features of the ASpec library. When explaining the features, we will work with the classes defined in listing 1.

In some code listings, we will use the *should* method from the RSpec unit testing framework. The *should* method is an assertion that passes if the following condition is true (e.g. it is used in listing 5).

Listing 1. Classes we will refer to in the examples in this section

```ruby
class Car
  def manufacturer
    'any'
  end
end
class BMW < Car
  def manufacturer
    'BMW'
  end
end
class Mercedes < Car
  def manufacturer
    'Mercedes'
  end
end
class Skoda < Car
  def manufacturer
    'Skoda'
  end
  def price(type, currency)
    10000
```

[1] https://code.google.com/p/mockito/
[2] http://rspec.info

```
  end
end
```

3.1 Execution Block

In ASpec, we separate the execution of the tested methods into an execution block (as shown on listing 2). The executon block is a parameter to the *execute* method. It is executed from within the *execute* method.

The ASpec configuration (e.g. stubbing or method call counts) has to be set before the execute method is called. Just before the execution of the block, ASpec creates the necessary aspects and metaprogramming definitions and removes them right after the execution.

The block can be executed once or multiple times, depending on the ASpec configuration. The values returned from the execution block can be accessed using the methods *return_value* for the value from the first execution and *return_values* for a list of values from all executions.

Listing 2. The execution block in this case executes the method SystemUnderTest.execute_the_tested_methods

```
ASpec.new.execute do
    SystemUnderTest.execute_the_tested_methods
end
```

3.2 Method Chaining

For convenience, most public methods of ASpec return the ASpec object, which enables method chaining. This means that the code snippets in listing 3 and 4 are equivalent.

Listing 3. Use of ASpec without method chaining

```
aspec = ASpec.new
aspec.stub(:BMW, :manufacturer) { 'Dacia' }
aspec.execute { BMW.new.manufacturer }
```

Listing 4. Use of ASpec with method chaining

```
ASpec.new.stub(:BMW, :manufacturer){ 'Dacia' }.execute{ BMW.new.manufacturer }
```

3.3 Method Stubbing

Method stubs are blocks of code that replace some methods in the system under test. A sample use of method stubbing is to replace a method that creates a connection to a development database with a method that creates a connection to a test database.

The *stub* method of the ASpec library accepts as arguments the name of the replaced method and its class and also a block of code that should replace it, as shown on a RSpec test case in listing 5. To replace the original methods, ASpec defines aspects that are executed around them and that call the blocks of code given to the *stub* method.

Listing 5. Stubbing the method manufacturer in the BMW class to return the string Dacia. The return value is tested using RSpec.

```
aspec = ASpec.new.stub(:BMW, :manufacturer) { 'Dacia' }
aspec.execute { BMW.new.manufacturer }.return_value.should == 'Dacia'
```

3.4 Raising Artificial Exceptions

In some cases, we might want to test how our code handles exceptions. For instance, if we want to test whether our code works without internet connection or if the database is not accessible.

To artificially raise an exception from a method in ASpec, we can use the method *raise_exception*, which takes the name of the method to be affected and its class as arguments. On listing 6, we can see a test case that passes if the exception is caught.

Listing 6. A test case that checks that an exception was raised from the set method.

```
aspec = ASpec.new.raise_exception(:Mercedes, :manufacturer)
aspec.execute do
    exception = false
    begin
        Mercedes.new.manufacturer
    rescue
        exception = true
    end
    exception
end.return_value.should == true
```

3.5 Counting Method Invocations

When debugging or testing algorithms, it can be useful to make sure that the right methods are called and that they are called the expected number of times.

ASpec provides the method *count_method_calls* to set the exact number of times a method should be called. For each of the methods with the defined number of calls, ASpec creates an aspect with a before setting. When the aspect is executed, it increments a counter for that method. After the execution, it is possible to test whether the method calls were as expected by comparing the actual numbers of method calls (method *method_call_counts*) and the expected (method *expected_method_call_counts*). It is also possible to debug the actual method call counts using the method *method_call_counts*. A sample test case for testing method call counts can be seen in listing 7.

Listing 7. A test case that passes if the number of method calls is as defined

```
aspec = ASpec.new
aspec.count_method_calls :Skoda, :manufacturer, 2
aspec.count_method_calls :Skoda, :price, 1
aspec.count_method_calls :Mercedes, :manufacturer, 1
aspec.count_method_calls :BMW, :manufacturer, 0
aspec.execute do
    skoda = Skoda.new
    skoda.manufacturer
    skoda.manufacturer
    skoda.price 'Octavia', 'EUR'
    Mercedes.new.manufacturer
end
aspec.method_call_counts.should == aspec.expected_method_call_counts
```

3.6 Testing Method Arguments

Similarly to testing method call invocations, we can test their arguments. The ASpec method *expect_method_arguments* defines the expected arguments passed in a list for each invocation of the given method.

Before execution of the tested block, ASpec creates an aspect for each of the defined methods. The aspect is executed before the execution of its method to save its arguments. If the output of the methods *method_call_arguments* and *expected_method_call_arguments* is the same, the functions were passed the same arguments as expected. This can be tested using a test case as shown in 8. It is also possible to debug the actual arguments using the method *method_call_arguments*.

Listing 8. A test that checks whether the arguments passed to certain methods were as expected

```
aspec = ASpec.new
aspec.expect_method_arguments :Skoda, :price,
    [['Octavia', 'EUR'], ['Rapid', 'EUR']]
aspec.expect_method_arguments :Skoda, :manufacturer, [[]]
aspec.execute do
    skoda = Skoda.new
    skoda.manufacturer
    skoda.price 'Octavia', 'EUR'
    skoda.price 'Rapid', 'EUR'
end
aspec.method_call_arguments.should ==
    aspec.expected_method_call_arguments
```

3.7 Testing the Liskov Substitution Principle

The Liskov Substitution Principle states that if S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program (source: Wikipedia[3]).

To simulate such situation over some given code, we can create an aspect for the join point of the constructor of T that would instead return a subtype S. This should be repeated for each of the subtypes S. This solution is similar to the Cuckoo's Egg pattern.

Since the Aquarium AOP framework doesn't provide any means to create a join point for a constructor (neither do other Ruby AOP frameworks), we were forced to use metaprogramming to implement this feature. Using metaprogramming we replace the constructor of type T with a method that returns a subtype S.

This feature can be used as shown in listing 9. In this case, we call the function *lsp* of ASpec to define that the allocation of an object of class *Car*, should be replaced with the allocations of objects of classes *BMW*, *Mercedes* and *Skoda*. The block in the *execute* method is executed for each of the given classes and the returned values from each execution are accessed using the *return_values* method.

Listing 9. Counting the method calls per class

```
ASpec.new.lsp(:Car, [:BMW, :Mercedes, :Skoda]).execute do
    car = Car.new
    car.manufacturer
end.return_values.should == ['any', BMW',␣'Mercedes',␣'Skoda']
```

4 Comparison With Other Mocking Frameworks

The Java mocking framework Mockito enables creating mock objects with a defined behaviour. A disadvantage of Mockito compared to ASpec is that each object in Mockito has to be created as a mock object explicitly. That means that objects created in the system under test, that are not accessible in the test, can't be affected. Using aspects we can control any object of a given class without explicitly mocking it. Mockito also supports counting method invocations and checking their arguments, however this is also possible only with mocked objects.

[3] http://en.wikipedia.org/wiki/Liskov_substitution_principle

RSpec, the Ruby testing and mocking framework, enables stubbing functions on any instance of a class. However, this framework does not support counting method invocations or tracking their arguments.

Neither Mockito nor Rspec have an explicit support for testing the Liskov Substitution Principle.

5 Related Work

The closest solution to ours that uses aspect-oriented programming for unit testing is FlexTest [2]. FlexTest is implemented in AspectJ and applies AOP to problems inherent in object-oriented programming. It enables testing encapsulated members, testing properties over a whole context and also testing the Liskov Substitution Principle. However, FlexTest only provides abstract aspects for testing and requires the tester to have a knowledge of AOP.

An alternative approach is described in [3], where the Aspect-Oriented Test Description Language (AOTDL) is introduced. It can be used to build top-level aspects for testing generic aspects.

There was also a research interest in using aspects to test aspects [1]. They also describe how aspects can help to build testing tools.

6 Conclusion and Future Work

We have shown that aspect-oriented programming can be used to solve issues we face when writing unit tests. Our library, ASpec, enables stubbing methods, raising artificial exceptions, counting method invocations, tracking their arguments and also testing the Liskov Substitution Principle. ASpec is very simple to use and doesn't require any knowledge of aspect-oriented programming.

We have compared ASpec to other mocking frameworks. We think that frameworks like Mockito, which is written in Java, could benefit from using AOP since, currently, they require mocking each object explicitly to modify its standard behaviour. In languages like Ruby, metaprogramming can be leveraged to build a library similar to ASpec. However, we think that using AOP is much simpler and more readable.

In future work, it would be interesting to see ASpec implemented in a statically typed language like Java. However, the implementation wouldn't be much different as long as aspects can be defined dynamically. There are also other ways to make use of aspect-oriented programming in order to help in unit testing, for instance by doing continual checks on some defined conditions accross test cases.

References

[1] Sokenou, D., Herrmann, S.: Aspects for testing aspects. In: *Workshop on Testing As-18*, Citeseer, 2005.

[2] Sokenou, D., Vösgen, M.: FlexTest: An Aspect-Oriented Framework for Unit Testing. In Reussner, R., Mayer, J., Stafford, J., Overhage, S., Becker, S., Schroeder, P., eds.: *Quality of Software Architectures and Software Quality*. Volume 3712 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 257–270.

[3] Xu, G., Yang, Z.: A novel approach to unit testing: The aspect-oriented way. In: *International Symposium on Future Software Technology*. Volume 10., 2004.