Test Driven Development In Rosie Pattern Language

By Onorio Catenacci
http://onor.io
https://twitter.com/OldDutchCap

This article first appeared in the May 2019 edition of Prose Garden (https://theprosegarden.com/contents-of-recent-issues/#06-19). It is posted here with the permission of the author.

Given the erudite nature of the people who read Prose Garden I am confident that the fastest way I could discover a plethora of alternatives to regular expressions would be to claim that there are no good alternatives to regular expressions. So I will instead confine myself to saying that I am aware of one excellent alternative to regular expressions and that's the Rosie Pattern Language. One of the features that I find most impressive is that they've built in the ability to test expressions in the code. That is what I will set out to explain.

But First . . .

Of course I feel quite confident in guessing that most readers haven't heard of the Rosie Pattern Language so I think a few words of introduction are in order. Rosie Pattern Language (or RPL for short) was created and open-sourced by IBM.

They had the problem of trying to find data in literal Petabytes of data: security logs, application logs and the like. They tried creating regular expressions(RE) but they quickly ran into the multitude of drawbacks which RE's present. They went back to the drawing board and invented something to remedy the shortcomings of RE; hence RPL was born.

There are several advantages of RPL over RE's and I won't describe them all here. The primary advantage I'll concern myself with (for purposes of this article) is the fact that the architects of RPL considered testing a first class need for a language and so they built it in.

By the way if you care to follow along with this, it's quite easy to get Rosie. You can get the source from the RPL repository on GitLab: https://gitlab.com/rosie-pattern-language/rosie. There's all sorts of good information on RPL and the motivation behind it at the RPL language site as well: https://rosie-lang.org/

TDD and Rosie

So, of course when we practice the discipline of TDD, we first decide what we'll test, then we write the test and last we write the code to pass it.

For sake of illustration let's say I've been tasked with finding all references to US Zip Codes in a large text file. I am not aware of a rule for zip codes beyond the fact that old zipcodes in the US were five digits (after 1963) and since 1983 "Zip Plus Four" has been optional to allow for faster routing; hence four more optional digits. I rely on the readers of this fine journal to contact me and let me know where my assumption has gone astray but for now I'll work from that assumption.

So first things first; first I need to write a test that my expression will accept a five digit number. Since there are literally 100,000 five digit numbers (if I count 00000), I'm going to test edge cases. I fire up my handy text editor and I write this:

```
-- test base_zip accepts "00000"
```

Let me take this apart for you. The double dash is Rosie's comment character. The test keyword tells Rosie that this is an embedded test. Base_zip is the name of the expression to be tested (we'll write this in just a moment). The accepts keyword indicates that Rosie should match the expression on an input of 00000.

Next I will try to write code to fail the test so that I can be assured my test is actually running. I add the following immediately above the test line:

```
base_zip = [0-9]{4}
```

This says match on 4 (and only 4) occurrences of any digit from 0 to 9.

Now I will save this text into a plain text file which I will call *zipcode_ut.rpl* rpl being the default extension for RPL source code files.

Next I run my test and ensure it fails as expected by doing this:

```
rosie test zipcode_ut.rpl
```

And, just as expected I get the following output:

```
rosie test zipcode_ut.rpl
zipcode_ut.rpl
    FAILED: base_zip did not accept 00000
    Tests: 1  Errors: 0  Failures: 1  Blocked: 0  Passed: 0
```

So now I need to add code to fix this error. I will modify this expression to match five digits instead of four—like so:

```
base_zip = [0-9]{5}
-- test base_zip accepts "00000"
```

This passes as we would expect.

```
rosie test zipcode_ut.rpl
zipcode_ut.rpl
    All 1 tests passed
```

Now we want to test the other end of our range. So we can simply modify our test to include it like so:

```
base_zip = [0-9]{5}
-- test base_zip accepts "00000",”99999”
```

We run this and we get two passing tests. So now let's add some tests to ensure we don't accidentally get 5 digit strings which aren't valid. This time rather than asserting that the expression will accept some input, now we assert that the expression will reject specific inputs. Our code now looks like this:

```
base_zip = [0-9]{5}
-- test base_zip accepts "00000",”99999”
-- test base_zip rejects “000000”, “99,999”, “-00000”
```

I now have five passing tests. By the way, quick note; I have my unit tests immediately below the expression they're testing. That's not necessary; it's just something I do to make it more convenient for others who may need to maintain my code.

Next I'll work on testing the zip plus four portion of my match expression. This is one of the strengths of RPL; I can build an entirely different matching expression and then combine multiple expressions into a signle expression to match. This allows me to focus on insuring I get the smaller parts right so that when I combine them it's far more likely my combined expression will work correctly.

```
zip_plus_four = [0-9]{4}
--test zip_plus_four accepts "0000","9999"
--test zip_plus_four rejects "00000", "9,999", "-0000"
```

Now I have 10 passing tests. Now comes the last step—combining base_zip and zip_plus_four into my full match expression. I'll simply show you the expression and the tests.

```
zipcode = {base_zip ("-" zip_plus_4)?}
-- test zipcode accepts "01111","91111","01111-1111","91111-9111"
-- test zipcode rejects "000000-0000", "-0000"
```

Notice that I retest to ensure five digits still passes. I also, of course, test to ensure that if a zip_plus_four is specified that works too.

Why Is This Good?

Besides the obvious benefit of TDD, this approach adds a few other nice benefits to the package.

First, it puts the tests and the expressions in the same place. This gives me a built-in sample if I have a question as to what the expression should or should not match.

Secondly, I can run my tests from a Command Line Interface (CLI) as was demonstrated above. This means I can easily integrate tests into a CI/CD workflow and force the tests to run every time I build.

Third, although I hadn't mentioned it before, RPL has bindings for Python, Go, and Haskell and the RPL community is working to build native interfaces for other languages as well.

Now there are some other wrinkles that RPL allows me to do to make pattern matching even easier but I'll save that conversation for another article. RPL is truly a step forward in using regular expressions and I hope I have whetted your appetite to learn more about it!