Addition by Abstraction

Jerren Every

jerren.r.every@huntington.com

Abstract

Many current automated test suites are brittle and not adept to handle change. This often occurs when people get pressed for time and begin to focus on short-term goals. This focus on the short-term can mean that your team will not be successful long-term as technology or your application goes through drastic changes.

The goal of a test development team should be to make a test suite that can stand the test of time. Common code development techniques such as Orthogonality, DRY(Don't Repeat Yourself), Metaprogramming, Refactoring, TDD (Test Driven Development), Dynamic Failures and Data Setup/Cleanup can help us create a more robust test suite that delivers value to the user.

Biography

Jerren Every is currently working as an Automation Developer Analyst-Senior on the Automation Architecture team at Huntington National Bank. His focus is on mastering the art of automation within design and deployment in a dynamic/fast paced agile environment. Jerren has received Professional Scrum Master, Professional Scrum Product Owner and Professional Scrum Developer certifications as well as the Certified Software Tester and Certified Software Quality Analyst certifications. With a strong knowledge of scrum ideology, QA ideology and automation ideology, Jerren is driven to help others embrace and understand these concepts. Jerren graduated from the Ohio State University with a degree in Psychology and enjoys bridging the gap in conversations between business minded individuals and technical minded individuals.

1 Introduction

This paper is an overview of the ways in which an automated test suite can be developed to be dynamic and robust by utilizing: Orthogonality, DRY(Don't Repeat Yourself), Metaprogramming, Refactoring, TDD (Test Driven Development), Dynamic Failures and Data Setup/Cleanup.

These ideologies are largely part of the standard body of knowledge for development for a variety of technologies. The source for definitions in this paper is the Pragmatic Programmer[1] which is consistently rated as one of the best software development books on the market. The challenge is that it is difficult to find examples on how to apply these ideologies within a testing framework.

With that being said, my goal in writing this paper is to start a conversation about how we can utilize these coding practices in a testing framework in order to develop more robust and dynamic tests that require minimal effort to maintain. The test frameworks in which I have utilized these ideologies include: database testing frameworks, mainframe testing frameworks, web testing frameworks and REST testing frameworks.

The structure of this paper will list first the ideology being discussed followed by a definition, then a section of examples of how to implement this ideology within a testing framework, and last any constraints that the ideology may have.

2 Making Code Orthogonal/Decoupling

2.1 Definition

Orthogonality is a term stolen from Geometry, two lines are orthogonal if they meet at right angles. In a programming sense parts of code are orthogonal if changes in one do not impact changes in the other (Hunt, 34).

Orthogonal code is beneficial as your team will not have to change existing code as new code is developed. (Hunt, 34).

Orthogonal code will reduce risk in the following ways (Hunt, 34).

A. Bad code is isolated.
B. Changes are isolated.
C. Easier to unit test.
D. Layer of abstraction between services and code.

2.2 Implementation

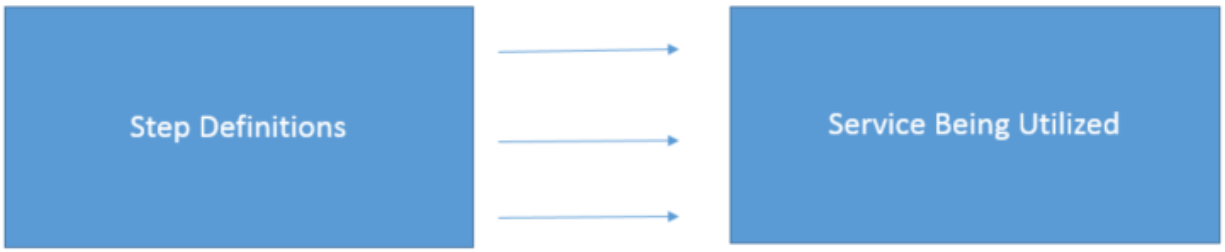
The main way decoupling is beneficial in an automated test suite is through providing an abstraction layer between your internal code and any external functionality. Some examples of external functionality are:

A. Excel
B. Watir (Web Application Testing In Ruby)
C. Databases
D. Rest Client

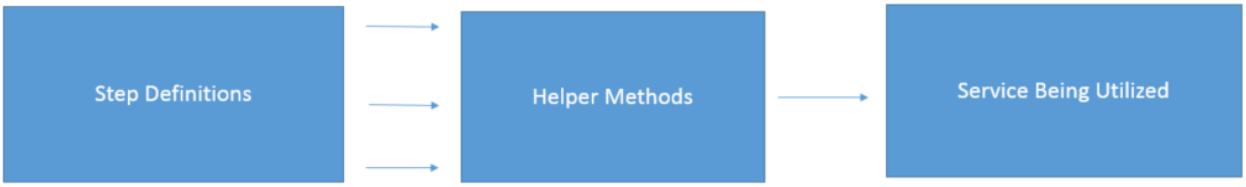
[1] Hunt, Thomas. 1999. *The Pragmatic Programmer: from journeyman to master*

E. Mainframes

Most often in test suites we see step definitions directly interacting with the services mentioned above. This non-orthogonal design will make it difficult to change and test your code.



Orthogonal Code – Step Definitions reference a helper method which interacts with the service being utilized. Changes to the service only impact the helper.



Another big benefit of utilizing these helper methods is that you now have a bit of functionality that is easy to unit test.

Unit testing is the act of testing a single section of functionality without outside interference. Within Ruby, RSPEC is typically utilized to unit test code.

2. 3 Constraints

There is a balance that needs struck between complexity and orthogonality.

3 DRY (Don't Repeat Yourself)

3. 1 Definition

In programming we want to be as lazy as possible, this means that we should take steps so that we do not repeat the same code. If the same logic is utilized twice this logic should be extracted into a method and stored so that it can be reused later on down the road.

3. 2 Implementation

The concept of DRY seems simple enough, most of the difficulty comes when you start making alterations that impact what others have developed. I would recommend first fostering a culture where no individual

owns portions of code but the whole team shares that responsibility. I would also recommend discussing the changes you are making with the team and showing the value that the changes are providing.

3. 3 Constraints

Making changes to code often times has unintended consequences. Ensuring that unit testing is performed on your code is a good way to ensure that your changes don't impact the external functionality of the code.

4 Metaprogramming

4. 1 Definition

The process of taking details out of the code. Developer should utilize metadata from the end user to set the arguments for the methods, the methods should be dynamic enough to allow for maximum reuse.

4. 2 Implementation

Within a Ruby/Cucumber test automation suite most of the details get into our code externally from the Gherkin level passing in arguments to the step definition. But what some teams miss is that while we are passing data from the Gherkin level to the step definitions, the data passed should only be enough to make the end user capable of understanding what the test is validating.

For example, it makes sense that the expected business logic that needs validated is housed at the Gherkin level like:

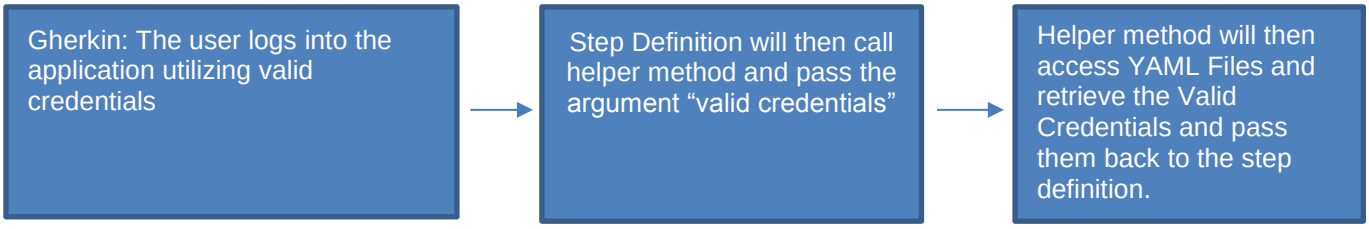
Given the user inputs 2 + 2

When the user selects calculate

Then the application returns four

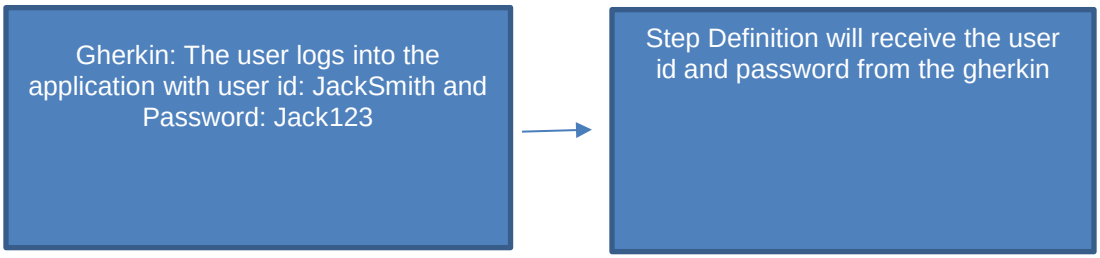
The more meaningful form and less evident way of metaprogramming is utilizing data keys where business logic is not concerned. In the example below the business can define what it means by valid credentials, and our code can go retrieve that test data and utilize it. This is beneficial as it prevents hard coding of values where it is not necessary and you can utilize the same data object in multiple flows.

Good Example:



We can contrast the above with the unnecessary hard coding seen below. In addition to being brittle this has little value to the end user. When they read this test they don't know what the JackSmith user is or what status his account is in.

Bad Example:



4. 3 Constraints

We should be careful only to take the details out of the code when it makes sense.

I would also caution that when utilizing metadata/metaprogramming we should write our test in a way that only valid data should pass the test. Otherwise we risk having false positives in our testing suite.

5 Refactoring

5. 1 Definition

Refactoring is the process of improving the structure of code without changing how the actual code functions.

5. 2 Implementation

In the previous three sections we have discussed orthogonality, metaprogramming and DRY ideologies. All of these ideologies are examples that overlap with the ideals of refactoring. Ruby is a text manipulation language. Most often in test automation we are either manipulating input text, or validating data returned to ensure it meets our expectation. How we structure the code that supports this input preparation and output validation is where refactoring becomes important and the three previous ideas lay a good groundwork to begin refactoring.

5. 3 Constraints

Refactoring is meant to be an organic process that happens as it provides value. Software is meant to be soft and change is supposed to be constant. One should not "own" code and one should not be defensive when code is critiqued.

Having a refactoring session is very beneficial in bringing about change to a test suite. These meetings should be discussions about finding improvements, not as a time to point blame.

6 Test Driven Development

6. 1 Definition

TDD- Test Driven Development is the process of developing code through testing. The benefit being that the software you develop is much more likely to be robust and less error prone. Additionally you have developed unit tests that can be utilized for the life of the functionality you have implemented.

Test Driven Development Steps:

A. A programmer writes a test for a bit of functionality that is not yet developed. The test is ran and it fails.
B. The programmer then writes the code that he thinks will satisfy the unit test
C. The test is ran and it passes or fails
D. If the test passes the programmer can then refactor the code or move on to additional functionality.
E. If the test fails the programmer can make modifications as needed.

6. 2 Implementation

Within Ruby, RSPEC is the most common framework for implementing unit tests that you can utilize for TDD. The Unit test developed provide continual benefit as you can test the validity of your code long after the code has been developed.

6. 3 Constraints

Most often the constraint with utilizing TDD come about through managerial concerns about implementing the new process.

7 Dynamic Failures/Timely Failures

7. 1 Definition

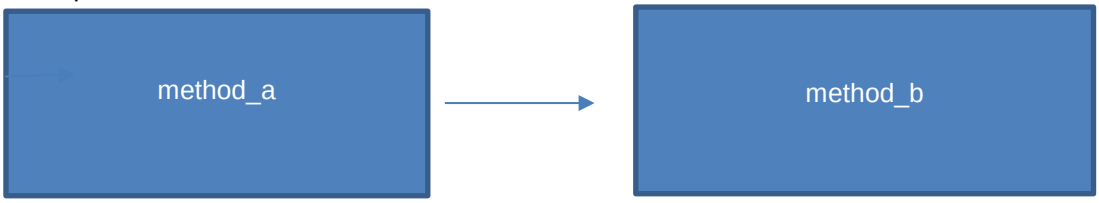
Test your code to fail as early as possible (RSPEC). Once a failure occurs have code to ensure that steps are completed to roll back any changes made to your environment.

Ensure that failures occur in a manner that will direct the developer to the root cause of the issue.

7. 2 Implementation

A programmer can implement simple checks on the input data that is provided to a method to ensure that the input data received is valid. This layer of checking when a method begins running provides an ideal separation of work for items.

Example:



In the above example if method_b always accepts what method_a returns, method_b can be blamed for method_a's failure.

If instead method_b looked at what method_a returned to make sure it was valid any failure within method_b is sure to be caused by the logic within that functionality.

Rescue clauses within Ruby provide an easy way to catch a failure and continue on an alternate flow that allows for the failure to be addressed through additional steps or cleanup if need be.

7. 3 Constraints

Rescue clauses should be used carefully. A true failure within the application that could be representative of a defect should not be caught by the rescue clause and should be passed to the tester as an issue.

Methods should not be so constrained by what they can receive as input that they lose a lot of flexibility.

8 Data Setup/Cleanup

8. 1 Definition

Data necessary for a test should (as often as possible) be created and destroyed within the context of a test. The benefits being that each test is receiving clean data, and the test leaves less of a footprint once complete.

8. 2 Implementation

In Ruby, hooks provide the ability to execute certain functionality before and after a test. For example; I worked on an account origination application and we had a tag invoked after hook that would remove account linking for a current user so that the test data would be able to utilize multiple times within the test suite. Without this after hook this test data would've been ruined after each test run.

8. 3 Constraints

This process of test data setup and tear down is the gold-standard but can become inefficient, this should only be used where it makes sense by improving the quality and efficiency of a given test.

9 Conclusion

In conclusion we have discussed:

- Orthogonality and decoupling as a way to provide a layer of abstraction between the step definitions that run your tests and the services that support those step definitions.
- DRY as a way to minimize duplication and the ideology that the team owns the code not the individual
- Metaprogramming as a way to remove details from our code that would otherwise make our code base brittle.
- Refactoring as a process combining a bunch of individual ideas to modify the structure in which our code operates.
- TDD as a way to minimize risk when creating and updating test automation software.
- Dynamic/Timely Failures as way to make sure that the developer can quickly identify problem code rather than going down a stack trace goose hunt.
- Date Setup/Cleanup as a way to maintain robust tests that require minimal preparation prior to execution.

All of these ideals combined provide the groundwork for a robust testing framework that can be fun to work within and easy to modify when necessary. One point to consider is that any programming standard has a point where the usage of that ideal can make a test suite optimal and a point where it can make a test suite inefficient. The conversation each team has to have is how they can use these ideals effectively in the environment in which they operate.

References

Hunt, Thomas. 1999. *The Pragmatic Programmer: from journeyman to master.*