TDD can be somewhat misunderstood because it is not a well-defined process. “What comes next?” was the initial question that this research was seeking to answer. In seeking to more clearly define and measure TDD, a number of different approaches have been explored. One approach was to consider a hypothesis set forward by Robert Martin called the Transformation Priority Principle. Robert Martin (colloquially known as “Uncle Bob”) is a programmer and author, particularly in the realm of Agile Planning and Programming. He is a staunch proponent of the practice of Test Driven Development, going so far as to say that “it could be considered *unprofessional* not to use it.” (The Clean Coder, p. 83).

To understand this premise, the words Transformation and Priority must be examined in the context. Martin draws a parallel between Transformations and Refactorings. In the red/green/refactor cycle of TDD, a red condition indicates a failing test case, a green condition indicates all test cases currently pass, and refactoring is a situation where the structure of the code is modified or “cleaned up” without changing the behavior of the code. This “clean up” is designed to make the code more readable and/or maintainable. Martin Fowler referred to removing “bad smells” in the code. (Refactoring, Martin Fowler, p.75)

Transformations are used to bring the production code from a red condition to a green condition. They change the behavior of the code without significantly changing the structure of the code. Martin uses a TDD philosophy “As the tests get more specific, the code gets more generic. (<https://sites.google.com/site/unclebobconsultingllc/home/articles/as-the-tests-get-more-specific-the-code-gets-more-generic>.) He sees the Transformations as a way to *generalize* behavior of the production code without changing the structure. His initial list of Transformations was expressed this way (<http://blog.8thlight.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html>) :

* **({}–>nil)** no code at all->code that employs nil
* **(nil->constant)**
* **(constant->constant+)** a simple constant to a more complex constant
* **(constant->scalar)** replacing a constant with a variable or an argument
* **(statement->statements)** adding more unconditional statements.
* **(unconditional->if)** splitting the execution path
* **(scalar->array)**
* **(array->container)**
* **(statement->recursion)**
* **(if->while)**
* **(expression->function)** replacing an expression with a function or algorithm
* **(variable->assignment)** replacing the value of a variable.

In a later video, the list was simplified to this (*The Transformation Priority Premise* video, Bob Martin, January 31, 2014):

1. Null
2. Null to Constant
3. Constant to Variable
4. Add Computation
5. Split Flow
6. Variable to Array
7. Array to Container
8. If to While
9. Recurse
10. Iterate
11. Assign
12. Add Case

In both lists, the ordering of the Transformations is important. As you traverse the list from top to bottom, the transformations progress from lesser complexity to greater complexity. So to a certain degree it addresses the question of “What next?”

Micah Martin writes the following in his blog: “As I pondered these transformations, I found it simpler to think about them in terms of the resulting code, and that led me to the short list below.

1. constant : a value
2. scalar : a local binding, or variable
3. invocation : calling a function/method
4. conditional : if/switch/case/cond
5. while loop : applies to for loops as well
6. assignment : replacing the value of a variable”

(http://blog.8thlight.com/micah-martin/2012/11/17/transformation-priority-premise-applied.html)

Regardless of which list is used, Martin’s Priority Premise is that programmers should opt to perform transformations to their code that are higher up on the list. When following this order, Martin hypothesizes that “better” algorithms will emerge than if transformations are selected from further down the list. In other words, given the option to use a simpler transformation or a more complex transformation, the programmer should select the simpler transformation.

This Priority Premise has implications both with transforming the code to achieve a green light state, but also with the subsequent tests to be written. As the production code is transformed to make the tests pass using higher priority transformations, so the next test written should lead the developer to choose a higher priority transformation. Martin believes that this should lead to simpler tests which should lead to simpler code.

The idea was to test this hypothesis using the Software Process class (CS 6700). After being instructed on TDD (without the additional Transformation description and priority list), the class was to use TDD to develop a series of homework assignments. The assignments would increase in complexity over the course of the semester. After 3 homework assignments, the class would receive further TDD instruction including the Transformation List and be instructed to use the list as a guideline for moving from red to green.

The normal development platform for Software Process is the IDE Eclipse along with the Python scripting language. Students used the version control software git, integrated into Eclipse, to create a local repository of their Python code. They were instructed to commit their code after every TDD red and green light. An Eclipse plug-in was written that provided buttons the students pressed that would run their code, and also automatically submit their code to the git repository with a comment indicating whether the commit was for a Red or Green Light. The students initially submitted the entire git repository, and in later semesters pushed their code to Github.

Custom code was written in Python to provide automated examination of the git logs. The initial approach was to evaluate the transformations that occurred in each commit. Key words and symbols were used to search the git log to analyze for transformations. Git logs contain only the incremental change from one commit to the next. Lines that were added in the commit have a leading plus sign in front of them, deleted lines lead with a minus sign. Below is a table showing how the various keywords and symbols were interpreted to determine the specific transformation:

|  |  |
| --- | --- |
| **Transformation** | **Conditions that indicate the transformation has occurred** |
| Null | + pass  + return None  + return |
| Null to Constant | - pass  - return None  - return  + return with a number, or a string literal, or empty list |
| Constant to Variable | - return with a number, or string literal, or empty list  + return with variable name |
| Add Computation | + string containing either +,-,\*,/,%,math. |
| Split Flow | + if |
| Variable to Array |  |
| Array to Container |  |
| If to While | - if (record conditional values)  + while (if it contains the same conditional values as if) |
| Recurse | + Method name is called within the method of that name |
| Iterate | + for |
| Assign | + Parameter is assigned a new value inside the method’s code. |
| Add Case | + else or elif |

In the process of evaluating the transformations, any patterns opposed to the TPP were also noted. The following Anti-Transformations were recorded:

|  |  |
| --- | --- |
| **Anti-Transformation** | **Conditions that indicate the anti-transformation has occurred** |
| Constant Only | + return with a number, or a string literal, or empty list with no corresponding deleted return or return None |
| Straight to Variable | + return with variable name with no corresponding deletes consistent with Constant or Null |
| While with no If | + while (with no corresponding if statement containing the same conditional values) |

A text-based report is generated for each student indicating the transformations that were detected in each of their commits.

(Thoughts: as I looked over my notes when we started looking at TPP, there was the thought of evaluating TDD as it is normally described, and then later adding the instruction about TPP. Then we would compare whether the code developed with a generic TDD instruction vs. code developed with a TPP mindset was better/worse/the same quality. The thought was

Higher risk == No TDD

Less risk == TDD

Even lower risk == TDD using TPP

At some point, it feels like we moved away from proving anything about the accuracy/validity of TPP. Maybe in made more sense in class to teach TPP in conjunction with TDD? Don’t recall for sure.)

A summary report was designed to indicate compliance with general TDD recommendations. At first, this report showed number of commits, red light commits, green light commits, average lines of code and transformations per commit, add and deleted/modified lines of both production and test code, and the ratio of production to test code.

The summary report led to additional questions about the students’ TDD performance. Did the students alternate between Red and Green light commits? If they were not alternating, why not? Were their tests failing in unexpected ways on the Red Light commit, resulting in consecutive Red lights? Was their production code not passing on their Green Light commit, resulting in a number of consecutive Green lights? Were they adding too much code at one time, violating the TDD principle that you should only write enough code to make a single test pass? Or in TPP terms, were they making more than one Transformation per Red Light/Green Light cycle?

A TDD grading criteria was developed by looking at some of these data points. A grade was assigned per commit by examining each Red and Green Light commit and whether they appropriately created production or test code in it, by looking at how many transformations they performed, and by how large the commit was. The following is a chart shows the breakdown:

|  |  |  |
| --- | --- | --- |
| **Criteria (per commit)** | **Condition** | **Grade** |
| If Red Light commit, does it contain Production Code | 0 | 100 |
|  | 1 and above | Deduct 10 points for every prod file in RL |
| If Green Light commit, does it contain Test Code | 0 | 100 |
|  | 1 and above | Deduct 10 points for every test file in GL |
| If Green Light commit, Is there more than 1 Transformation | 1 | 100 |
|  | 2 and above | Deduct 5 points for every additional transformation |
| Large Commits | Less than 50 LOC | 100 |
|  | Increments of 10 | Deduct 5 points for each increment of 10 LOC above 50 |

The three elements (there would either be a Red or Green Light per commit) were averaged together to produce a TDD grade for each commit.

If a student was performing TDD per the recommendations, then there should have been many commits. An overall assignment grade was calculated incorporating all the TDD Commit averages and examining the phenomenon of consecutive red or green lights. If a person is performing TDD correctly, then they should be cycling back and forth between red and green lights. So consecutive red and green light commits indicated either a difficulty at some point or a misunderstanding of the TDD process. The following table indicates the grading criteria for an overall assignment:

|  |  |  |
| --- | --- | --- |
| **Criteria (per assignment)** | **Condition** | **Grade** |
| Average length of Consecutive Red Lights | 1 | 100 |
|  | 2 and above | Deduct 5 points for every number above average length of 1 |
| Average length of Consecutive Green Lights | 1 | 100 |
| (The criteria for Green Lights was more lenient because they were given instructions to hit the Green Light button until the test passed) | 5 and above | Deduct 5 points for every number above average length of 5 |

The three elements were averaged together to produce a TDD grade for each assignment.

If the reader is familiar with TDD, he may be asking himself, “Where does Refactoring fit into the overall analysis?” Refactoring is a key component of the TDD Cycle, but as noted above, it is used to clean up the code without changing its behavior. In this analysis, the focus was placed on the transformations that actually implement the behavior between the Red Light and Green Light step. Students also had the opportunity to press a Refactor button. The data for the Refactor commits

Transformation grade should only count for Green Light commits.

By examining the amount and types of changes to the code between commits (hopefully with assistance from automated examination of the git logs), we can evaluate their TDD compliance. Using black box tests that provide, at a minimum, boundary value analysis we can evaluate the code quality. With those evaluations in hand, we can draw some conclusions and examine the implications. Using the implications, we can provide feedback to the students regarding how their TDD compliance impacts their code quality.

Potential quote to address from Kent Beck:

I get paid for code that works, not for tests, so my philosophy is to test as little as possible to reach a given level of confidence (I suspect this level of confidence is high compared to industry standards, but that could just be hubris). If I don't typically make a kind of mistake (like setting the wrong variables in a constructor), I don't test for it. I do tend to make sense of test errors, so I'm extra careful when I have logic with complicated conditionals. When coding on a team, I modify my strategy to carefully test code that we, collectively, tend to get wrong.

Different people will have different testing strategies based on this philosophy, but that seems reasonable to me given the immature state of understanding of how tests can best fit into the inner loop of coding. Ten or twenty years from now we'll likely have a more universal theory of which tests to write, which tests not to write, and how to tell the difference. In the meantime, experimentation seems in order. (Kent Beck, posted 09/30/2008, read on 6/25/2018, https://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/153565#153565)