A Scientific Approach to Developing Scientific Software

Gregory R. Watson
June 15, 2019

We recently decided to adopt a test driven development (TDD) approach for our development practices. Like many software developers, I've been writing unit tests for many years, but generally did so only once the primary development had been completed. As continuous integration tools have become more mainstream and easier to use, the importance and usefulness of good testing practices has become more apparent. However, switching to a TDD methodology, where tests are written a priori, has proved challenging. This is because I can no longer rely on my observation and knowledge of what the code should do in order to write the test. Instead, I have to reason about how a new requirement might be designed and try to conceive of a way to test that an implementation would be correct. It has become abundantly clear that this is something that requires considerable training and experience to become proficient at, while the old way of doing things is simply an extension of the design/development process that I've been using for years.

While thinking about this problem, it occurred to me that, at least in the case of scientific software, many of the practitioners who are developing the software are themselves scientists from other domains. There is little argument that the scientific method has been highly successful in driving scientific progress. What is it about the scientific method that makes it so successful? Although often presented as a number of steps or actions to carry out, the scientific method is actually a process to acquire knowledge about a particular subject using reproducibly testable and verifiable evidence. This process may include one or more of the following actions: observation; defining a question; research and planning; forming a hypothesis; predication from the hypothesis; experimentation; and evaluation. A key aspect of the method, and one that is often misunderstood, is the notion of falsifiability - that theories are never verifiable, but are only refutable - so the goal of the experimentation should be to refute the hypothesis.

Like most things, it turns out that others have thought about the relationship between the scientific method and TDD [1][2]. The approach generally taken is to consider a simple iterative model of the scientific method, and relate this to the generally accepted steps employed by TDD, as shown in Figure 1. The TDD process is typically described as: choose a story to work on; write a test case for that piece of software (the test will fail initially); write the simplest code possible to make the test pass; refactor and clean up code before moving on to the next story. This maps pretty well with the scientific method model, where choosing a story is analogous to "making an observation", writing a failing test is analogous to asking the question "can my application do this?" and the hypothesis "no, the application can't do this yet," writing code to satisfy the tests is analogous to "doing the experiment to refute the hypothesis", and finally, refactoring the code is analogous to "drawing conclusions and summarizing the results." A satisfying aspect of this comparison is that it matches the falsifiability criteria well. The scientific

method progresses by proving a hypothesis is incorrect and TDD progresses by creating code that falsifies the failing test. i.e. the test succeeds.

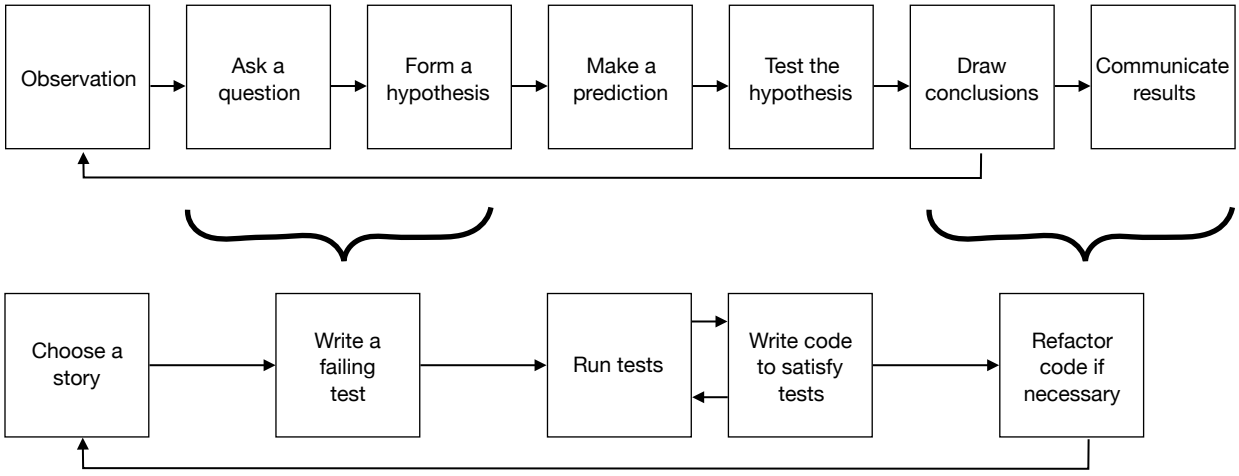

Figure 1: Scientific Method Model (top) compared to TDD (bottom)

Although this approach provides a good framework for approaching the TDD process, it still does not help much with the mechanics of actually implementing TDD. As Bill Caputo notes [3], there are still many questions to be answered, such as “what is a test,” “how much code should I write,” “what does refactoring mean,” “what is a Unit Test,” etc. Bill’s approach to this (given over 18 years of experience) is to consider TDD as a method (i.e. an empirical and iterative approach) rather than a mechanism (in the sense of an algorithm). The characteristics he outlines in his method are as follows (to paraphrase):

1. Think - sometimes for quite a while - about the problem being solved and how to solve it.
2. Discuss ideas with someone or otherwise seek the knowledge of others about the problem in question.
3. Try things out first, especially when starting a new task.
4. Ensure all existing assertions “fail to falsify” the system before changing anything.
5. Write some code that asserts something not currently true about the system (i.e. it falsifies the assertion)
   - Think a lot about how things would look once it’s completed.
   - Try to be as specific as possible.
   - Reference - as yet unwritten - code that would confirm the assertion if it existed.
   - Flesh out the context using fakes, spies, mocks, setting up state, etc until the assertion is in a valid environment.
6. Run this code and confirm the new assertion - and only the new assertion - successfully fails.
   - Iterate this and the previous step until you’re confident the environment is valid.
7. Add just enough code to confirm the assertion “fails to falsify” (i.e. passes).
8. Confirm that all the other assertions “fail to falsify."

9. Change names, extract functions and do other refactorings necessary prior to adding another failing assertion.
10. Keep doing this until the problem has been solved.

In summary, Bill's approach to TDD is the same that a scientist would take: namely "formulate a hypothesis about my delivery *method* and incrementally test those hypothesis against reality. If the resulting computational artifacts satisfy the demands placed on the system, they become my best model of the theory embodied in those demands."

Finally, I'd like to consider the syllogism created by Josh Peterson [4] who also provides a good introduction to syllogisms [5], which are essentially two or more statements connected by a logical argument, and which imply a conclusion.

> *The scientific method is the best way to advance technology,*
> *TDD is the application of the scientific method to software development,*
> *therefore TDD is the best way to develop software.*

He defines the terms in the syllogism as follows:

- *To advance technology* - The action of harnessing the natural world to perform tasks
- *Scientific method* - The iterative process of carrying out experiments
  - Making a hypothesis
  - Designing an experiment
  - Performing the experiment
  - Refining the hypothesis
- *Software development* - The action of instructing computers to perform tasks
- *Test Driven Design (TDD)* - The process of
  - Writing the minimum unit test code to cause a unit test to fail
  - Writing the minimum production code to cause the unit test to pass
  - Iteratively repeating the first two steps to generalize the code

These are hard to refute.

The syllogism comprises two premises, a major premise and a minor premise which must also be agreed on. The major premise is:

> *The scientific method is the best way to advance technology*

He argues that the rate of technological advancement since the widespread use of the scientific method has proven this premise to be true. The scientific method provides two concrete outcomes which have opened the door for technological advancement.

- The scientific method makes experimental results repeatable.
- The scientific method normalizes the process of performing experiments

The minor premise is:

> *TDD is the application of the scientific method to software development*

We have seen previously (based on Rich Mugridge's work) that there is a one-to-one correspondence between the steps of the scientific method model and TDD. This indicates that TDD is the application of the scientific method to software development.

The conclusion of the syllogism is the statement:

> *TDD is the best way to develop software*

In order for this to be true, the minor term ("TDD") must be the subject of the conclusion, the major term ("to advance technology") must be the object of the conclusion, and both statements connected by the middle term ("scientific method".) Therefore, the terms "to advance technology" and "to develop software" must be interchangeable. This holds if the tasks of software development are a proper subset of the tasks of technological advancement, which is also hard to refute.

References

[1] Rick Mugridge. 2003. Test Driven Development and the Scientific Method. In Proceedings of the Conference on Agile Development (ADC '03). IEEE Computer Society, Washington, DC, USA, 47-.
[2] John Prell. 2019. https://teamgaslight.com/blog/how-an-understanding-of-the-scientific-method-can-make-you-a-better-practitioner-of-agile-software-development
[3] William Caputo. 2017. https://logosity.net/notes.html#2017.02
[4] Josh Peterson. 2013. https://joshpeterson.github.io/the-scientific-method-and-programming
[5] Josh Peterson. 2013. https://joshpeterson.github.io/a-brief-introduction-to-syllogisms