

Test Driven Development and the Scientific Method

Rick Mugridge
Department of Computer Science,
University of Auckland,
New Zealand
r.mugridge@auckland.ac.nz

Abstract

The scientific method serves as a good metaphor for several practices in Extreme Programming (XP). We explore the commonalities and differences and show that the scientific method, by analogy, can be used to better understand Test Driven Development (and vice versa).

1. Introduction

Naur argues that programming can be understood as theory building. The evolution of programs "... is essentially dependent on a certain kind of knowledge possessed by a group of programmers who are closely and continuously connected with them." [14]. He argues that programmers' knowledge of a system goes beyond the code and documentation in several ways:

- Explanation of "...how the solution relates to the affairs of the world..."
- Explanation of "...why each part of the program is what it is..."
- Ability to "... respond constructively to any demand for a modification of the program..."
- Ability to make judgements about the program, as "... simplicity and good structure can only be understood in terms of the theory of the program".

While "programming as theory building" may be applied to software development in general, we go further than Naur. We argue that understanding how such theories develop through the scientific method is particularly instructive in understanding Test Driven Development (TDD) and vice versa. The scientific method provides a rationale for TDD.

The main elements of the scientific method are as follows:

- Repeatable experiments are used to test predictions of the current theory. An experiment has an associated control.
- Theories evolve through small and large changes. Theories must remain internally consistent, as

well as consistent with known experimental results.

Tests in TDD take the role of experiments, while design takes the role of theory. Experimental reproducibility is managed through continued use of automated tests to ensure that the theory has not been broken. The program theory is driven through experimentation with tests. The theory is refined to fit the tests, with refactoring to ensure suitable generality.

The way in which XP itself has evolved seems to fit the scientific method, being driven by examples and abstracted from real experience by software practitioners with an interest in theory building. It restores the status of programming, which has been eroded by the manufacturing metaphor that has dominated traditional software engineering.

In the next section we introduce two levels of TDD in XP. In Section 3, we introduce the elements of the scientific method and relate each of them to Test Driven Development. In the process, we provide a rationale for the steps in TDD and discuss good practices. Section 4 concludes.

2. Test Driven Development

Test Driven Development [1] applies at two levels in XP [13]. The first level involves the evolving design of the whole system within an organisational context, with iterations and the planning game:

- A customer writes stories that will lead to the development of the system.
- The developers estimate the cost of developing the stories (which may require that a story is broken down into smaller, more manageable stories).
- The estimates help the customer to prioritise the stories, in order to choose which will fit into the next iteration.
- Customer tests are developed for the stories chosen for an iteration. They are used to determine whether a story is complete.
- Stories may be exploratory because no-one is yet clear as to what the system should do, or they may

be well understood by the customer and clearly part of the system. As the system evolves, the value of different possible aspects of the system will become clearer as it begins to be fitted into its organisational niche.

Often in software design there is not a clear boundary between requirements and design, because it can be very difficult to foresee the impact and value of speculations about what is needed and what problem is being solved with a computer system. XP is especially well suited when the scope and value of a computer system are not well understood, where the design of the system has to evolve along with the requirements. As a concrete system evolves, the customer is often better able to determine what has value.

The second level of TDD is as described in [2]. This is at the level of micro-iterations in completing a task (a subcomponent of a story). Each micro-iteration involves the following steps [12]:

- Choose an area of the design or task requirements to best drive the development.
- Design a concrete test that is as simple as possible while driving the development as required, and check that the test fails.
- Alter the system to satisfy the test and all other tests.
- Possibly refactor the system to remove redundancy, without breaking any tests [8].

The steps of TDD are illustrated in Fig. 1. At both levels of TDD, an important issue is choosing the next step(s) to provide the best learning during development.

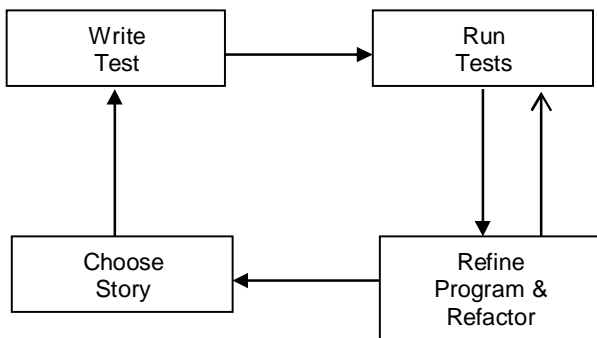


Figure 1. Model of TDD

3. Scientific Method and TDD

The scientific method is a model¹ of how theories of aspects of the world are developed through experimentation (or observation), as illustrated in Fig. 2.

¹ However, it is not universally accepted; there are other views of the philosophy of science

Given a theory of some phenomenon in the world, it is tested with experiments that check if the response of the world corresponds to the predictions of the current theory. Discrepancies lead to development of the theory and to subsequent experiments, providing a cycle of refinement and to the selections of theories.

We now consider elements of the Scientific Method and relate them to TDD.

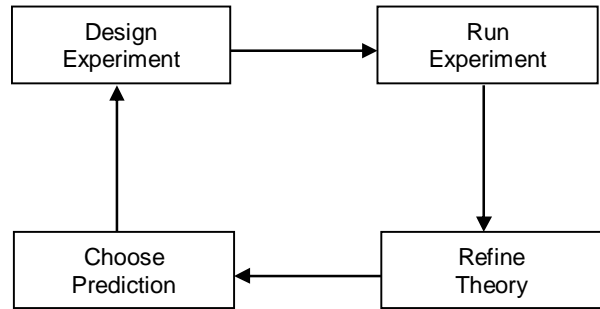


Figure 2. Model of the scientific method

3.1. Specifying a Theory

A theory needs to be specified in a form that makes clear the scope of the theory, that makes precise distinctions between elementary components, and which is specific enough to make predictions. Many scientific theories incorporate laws that are expressed mathematically. In general, a theory needs to be concise compared to the phenomena that it is attempting to explain; this means that it tends to ignore many properties to focus on a few.

When the definitions within a theory are sufficiently precise the theory may be considered as a "simulation" of the phenomena, where scientists use mathematics, computer models or imagination to predict emergent properties.

Creating a theory for something that exists in the world seems very different from designing and developing an artifact. However, to many people it is not clear whether scientists study something that really "exists". And on the other hand, software can be used to "create" new worlds through simulating them. As we see, it is useful to identify the commonalities between theory building and designing.

In software generally:

- The design is specified in a form that is precise, through program code (it may not be easy to understand, just as some theories are very technical).
- "Predictions" can be tested by executing the program with the tests.

In software, we are designing to satisfy perceived needs in a possible world. Our evolving system is just an element of some larger design (at the organisational level). So the parts of our system, as well as the whole system, have to be adaptable, to cope with change.

In TDD, the tests change as the aims of the software evolve.

But software is more than a theory, because it has to act in good time, with suitable levels of data reliability, security, and ease of use. Hence the simulation-nature of software is more grounded than that of theory.

3.2. Starting a Theory

Perhaps the most difficult part of creating a theory is in identifying a coherent aspect of the world that needs explanation. Often this arises from noticing a regularity, such as noticing that certain animals feed their young on milk. A theory may begin as a vague hypothesis or be developed as an alternative to a common sense model² of the world that is inadequate. Many common sense notions of the world are pervasive; it's easy to not question the views of the world you are brought up with, as they serve as fundamental assumptions.

To frame the very first version of a theory scientists must be clear about what their subject is. They may have to draw attention to something that has not been noticed before or that has been looked at in a different way. This may require a new name (eg gravity), which may in turn help or hinder an understanding of the phenomenon.

In TDD:

- The customer may not be sure what's really required, given that an unclear problem exists and needs to be clarified and solved.
- TDD encourages evolutionary design development while other approaches prefer design to be carried out before programming. The more unknowns there are about what is required, the more likely the design has to evolve as the needs of the customer are better understood.
- The first few tests may be difficult to choose, in order to get the process going, but after that it tends to be easier.

3.3. Selecting Experiments/Predictions

Going from an existing theory to an experiment is much more than a selection process. It's a matter of

identifying and choosing the best part of the theory to explore next, based on the potential value of the outcome.

This implies that the theory is sufficiently defined to be able to test predictions. An experiment may check that a concrete prediction from the theory itself is satisfied in the real world. The check may come from actively creating a controlled experiment to carry out the test or it may arise just from observation.

Experiments that are consistent with the current theory are likely to provide little information. So experiments are often designed to check the edge cases of the theory, either to push the bounds of the theory or to disprove it. For example, if two separate phenomena are seen to be examples of a more general notion in the theory, look for other things that are examples of that notion. Such experiments help to make distinctions between similar phenomena and help to find commonality and thus generality.

There is some cost associated with any experiment, so it is necessary to take account of the anticipated cost-benefit ratio of potential experiments. This can be difficult to assess, as it is not at all obvious where the theory may fail. For example, we cannot know whether an experiment that is focussing on smaller detail of a theory will lead to little or no change of the theory or whether it will lead to a major revision.

Many important developments in theory have arisen when people have decided to question their assumptions and consider the consequences of breaking them. Einstein is an obvious example.

Beginning with the prediction is analogous to de Bono's idea of learning things backwards, which "... may seem more complex but in practice it turns out to be easier and simpler. In this way you are always moving forward into an area you already know" [4].

In TDD:

- The planning game allows the customer to be agile with their design and to prioritise developments in order to learn the most, including how the software will fit into the larger context to provide value. "In general, most product failures are caused by market risk..." [16].
- TDD aims to drive the development through well-chosen tests (at customer and programmer levels). "... the purpose of a design process is to generate information" [16].
- Tests are designed to show an important inadequacy in the current design, so their selection is less hit and miss than the scientific method. They need to be chosen to best develop the design, often around the "edges", to provide good information: "...receiving failure

² The notion of "common sense" was based on a fallacy: that thinking occurs in the water-filled chambers of the brain

information early in the design process is much more valuable..." [16].

- The time scale of cycles of experimentation are much shorter with TDD, with micro-iterations being completed in minutes.
- Prediction is encoded as a test which can be tested automatically. The test is designed before the system is changed to satisfy it, working backwards.
- Designing a test, which acts as a mirror of the application, inevitably leads to design of the application (at least at the API level).

3.4. Carrying Out Experiments

In carrying out a scientific experiment, a control is used to ensure that extraneous factors do not lead to spurious results. A control is the same experiment step but without any change to the experimental variable of interest. A control serves the purpose of focussing on a single change. It is difficult to have a control when an experiment is pure observation (such as in cosmology and much of the social sciences).

Provision must be made for uncertainties that arise naturally from the experimental process. For example, measurements may not be precise, or there may be unnoticed extraneous environmental factors that affect the outcome.

Accuracy of measurements is an important consideration in the development of theories of the physical world; being sure that you're measuring what you think you're measuring is important in the social sciences.

Experiments may involve populations, with statistical results and a need to make the control and experimental populations as equal as possible.

In TDD:

- Small steps are used so that it is clear what caused the tests to fail, in the same way that a control tries to ensure a single change [7].
- A new test is first run to see that it fails before changing the software; this acts as a control on the test itself.
- Each test case needs to be independent of other tests, starting from a known state.
- While much of software testing is deterministic, uncertainties arise, analogous to imprecision in measurements. Non-determinism arises with concurrency, and measurement issues may be significant in non-functional requirements such as integrity.

- Statistical results arise from some sorts of tests, with multiple samples, such as testing for a required average response time.

Much of science has been able to develop through the design of equipment to carry out experiments [9]. Similarly, software development is constrained and invigorated by developments in hardware.

3.5. Reproducibility

Often there are competing theories, with their advocates trying to find success in the one they believe in and to find errors in those of their competitors. To avoid unnecessary arguments, experiments are designed to be reproducible, and the method and results are made publically available. Others can attempt to verify that the experimental results are correct. To be repeatable, the theory and the experimental design need to be clearly described - something happens or it doesn't; a measurement is more-or-less as expected.

In TDD:

- Test suites provide for reproducibility; they are rerun to ensure that we haven't broken emergent properties of our system that we wish to retain.

3.6. Experiments as Examples

In learning about a scientific area, exemplary experiments are used to both illustrate the experimental method, and to illustrate some aspect of nature in a concrete form. Such experiments serve to illuminate the area of interest.

In TDD:

- The test cases act as concrete examples that remain to help explain the ongoing development of the system.
- Keeping a history of significant customer and programmer tests is probably a good way to understand the system and how it has evolved.

3.7. Theory Simplicity

Where two theories explain the same phenomena, it is useful to have some approaches to choosing between them. Occam's Razor (a principle of William of Ockham) is one approach. It chooses a theory as more likely when it is simpler than another, all else being equal. For example, the theory that the earth was the centre of the universe lost to a competing theory, which more elegantly described the relationships between the members of our solar system.

As Albert Einstein said: "A scientific theory should be as simple as possible, but no simpler."

A theory may take considerable effort to develop but can end up appearing very simple once it has become widely accepted, being "obvious" in retrospect: "Because simplicity seems easy we believe it is easy to achieve. When it is not easy to achieve we give up too quickly" [4].

In TDD:

- "Occam's razor: choose the design with the fewer components, all things being equal" [3].
- "The simplest thing that could possibly work" [1].

3.8. Theory Generality

Some theories are more general than others, being able to explain a wider range of phenomena. Unifying multiple theories about related aspects of the world into a single theory is a favourite pastime of scientists.

As Richard Feynman says: "When you have put a lot of ideas together to make an elaborate theory, you want to make sure, when explaining what it fits, that those things it fits are not just the things that gave you the idea for the theory; but that the finished theory makes something else come out right, in addition." [6].

In TDD:

- Instead of aiming for the most general approach, TDD advocates keeping the design as simple as possible, and only generalising it as needed.
- Feynman's point makes it clear that we need Quality Assurance in addition to the tests that were used to drive the development of the system.

3.9. Theory: Evolution or Revolution?

A theory is found to be incorrect when its predictions differ from the results of experiments. The theory may be a little wrong, requiring some minor alterations to accommodate the experimental results, or it may need to be changed in more drastic ways.

A single theory may bud multiple paths of theories which compete for attention. Legacy theories, like legacy code, eventually die from messiness and overweight, as newer theories better explain the world. Others are rejected as experimental evidence builds against them. For example, evolutionary theories of mind better explain the experimental data about the brain than those that consider the brain to be a blank slate [15]. Some theories

end up being useful within certain constraints, such as Newton's laws.

Kuhn [11] argues that scientific theories go through revolutionary change, with an earlier theory being replaced by a new theory as it gains acceptance:

"... scientific revolutions are inaugurated by a growing sense ... that an existing paradigm has ceased to function adequately in the exploration of an aspect of nature to which that paradigm itself had previously led the way."

While some changes are incremental, Kuhn argues that "cumulative acquisition of novelty is not only rare in fact but improbable in principle" [11].

Hence theories have to undergo major and minor revamp to make them fit the experimental data. They are refactored over time to make them more general, sometimes through rejection of earlier ideas. As they are refactored or changed, they must remain consistent with the experimental data.

In TDD:

- A design emerges from evolutionary development of the software to satisfy the tests
- As the software evolves, we better understand what it's about (abstractions, naming, intent, types) and how the pieces interrelate (dependencies, coupling and cohesion).
- Some changes to a program simply add to the design, while others lead to fundamental changes. We do not usually distinguish between them, as refactorings occur repeatedly at multiple levels.
- Critics of XP argue that some design needs to be carried out upfront, due to the high future costs of the radical changes that may be needed, especially due to changes in "non-functional requirements". Advocates of XP argue that high-quality design allows changes to be made at reasonable cost when they are needed, and not before. Revolutions are to be expected!

3.10. Theory Building as a Social Process

It is not possible to prove that a theory is correct, only that it is (possibly) wrong if it cannot explain some experimental results. For a theory to become part of the body of science, it must survive public scrutiny and become generally accepted. Jardine discusses the social context of some 17th century developments of science [9].

In practice, science does not proceed in the tidy fashion we have presented so far. For example, Kaner et al [10] quote Rosenthal [17]: "... trained, conscientious,

intelligent experimenters unconsciously bias their tests, avoid running experiments that might cause trouble for their theories, misanalyze, misinterpret, and ignore test results that show their ideas are wrong".

In TDD:

- Team work and pairing practices in XP lead to programmers sharing a common view of the software.
- Programmers bring various skills to a team when working together, to build on each others' strengths (eg, brainstorm or criticize, abstraction skills or analysis/prediction skills, formalisation or an ability to find good examples, etc).
- TDD tries to avoid problems of wishful thinking about the quality of software by driving the development with sufficient tests. In other approaches to software development, programmers are less inclined to seriously test their code after they have written it as this means confronting the possibility that they may have made errors.

4. Conclusions

We have explored the value of the scientific method as a metaphor for the test driven development aspects of XP. It fits TDD rather well, especially when considering theory evolution and the role of reproducibility in scientific experimentation.

The scientific method has effectively provided centuries of guidance in finding ways of understanding the world. A lack of good science leads to speculative theories that are not tested by experiments, have little predictive power and are not to be trusted. The strong correspondence between TDD and the scientific method indicates that, compared to TDD, traditional approaches to software development are inadequate.

Acknowledgements

Thanks to Jackie Tyrrell for discussions about the scientific method. Thanks to the reviewers and John Hamer for helpful comments, and to Bryce Kampjes for pointing out the paper by Peter Naur.

References

- [1] K. Beck. *eXtreme Programming Explained*, Addison Wesley, 2000.
- [2] K. Beck. *Test Driven Development: By Example*, Addison Wesley, 2002.
- [3] Mike Champion, <http://lists.xml.org/archives/xml-dev/200203/msg00614.html>
- [4] E. de Bono. *Simplicity*, Penguin Books, 1998.
- [5] M. Feathers, *Working Effectively With Legacy Code*, April 2002, <http://www.objectmentor.com/resources/articles/WorkingEffectivelyWithLegacyCode.pdf>
- [6] Richard Feynman, *Cargo Cult Science*, quoted at <http://wwwcdf.pd.infn.it/~loreti/science.html>.
- [7] Phlip, extreme-programming yahoo email group, 16 Jan 2003.
- [8] M. Fowler. *Refactoring*, Addison Wesley, 1999.
- [9] L. Jardine, *Ingenious Pursuits*, Abacus, 1999.
- [10] C. Kaner, J. Falk, H. Q. Nguyen. *Testing Computer Software*, John Wiley, 1999.
- [11] T. Kuhn, *The Structure of Scientific Revolutions*, University of Chicago Press, 1962.
- [12] R. Mugridge, "Challenges in Teaching Test Driven Development", procs. *XP2003*, Genova, Italy, 2003.
- [13] R.Mugridge, "Test Driven Development as Micro-Iterations in XP", February 2003, www.cs.auckland.ac.nz/~rick/microIterations.pdf
- [14] P. Naur, "Programming as Theory Building", *Microprocessing and Microprogramming*, 15, pp253-261, 1985.
- [15] S. Pinker. *The Blank Slate*, Penguin Books, 2002.
- [16] D.G. Reinertsen. *Managing the Design Factory*, The Free Press, 1997.
- [17] R. Rosenthal. *Experimental Effects in Behavioral Research*, Appleton-Century Crofts, 1966.