Mutation Driven Development: A Jackdaw, a Magpie, a Raven

Alex Groce
Northern Arizona University
Flagstaff, USA
agroce@gmail.com

Abstract

Test driven development (TDD) is a controversial and interesting approach to software development; while many think of “better tests” as a primary *purpose* of TDD, in practice the goal is as much to use tests to encourage continued progress in coding. That goal however rests on the notion that TDD ensures tests are good enough to let you implement small new features and refactor code without undue fear of mistakes. Unfortunately, TDD is not “self-enforcing” and standard TDD practice makes it easy to accidentally skip steps. By integrating a phase of focused mutation testing into TDD, however, developers applying TDD can be sure they are actually writing code that is supported by the scaffolding of tests, and so code in justified confidence. The incremental nature of TDD test and production code creation ensures that at no point will “fixing up” the mutants be likely to overwhelm the developer, and so the final result will be tests with excellent code coverage and mutation score, without a painful effort to “patch up” an inadequate testing effort, and a TDD approach that includes automated checks that the letter and spirit of TDD are truly being respected, with the benefits of TDD presumably following in due course.

CCS Concepts: • **Software and its engineering** → **Dynamic analysis**; **Software testing and debugging**.

Keywords: test driven development, mutation analysis

ACM Reference Format:
Alex Groce. 2024. Mutation Driven Development: A Jackdaw, a Magpie, a Raven. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! ’24), October 23–25, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3689492.3689810

1 Preface

What follows, contained between the *Preface* you are reading now and the *Postscript*, which I hope you will stick around to read (not due to any obligation, but because you have enjoyed what I have written) was submited to Onward! Essays 2025, and initially conditionally accepted. I then modified it, in response to reviewer requests, as I read them, thinking the requests were both beneficial to the essay and easy to implement; it was then, to my surprise, rejected, and I never did receive a second batch of reviews. I know that one reviewer vehemently disliked the original essay, viewing it as a research paper in shabby disguise, and stating it wasn’t really an Onward! Essay at all (I disagree, but I’ll grant the entire thing does reasonably call forth such a suspicion). I did receive some feedback via back channels on the reason for the rejection of the (fairly small) modification, which we’ll talk about after the main essay is done. I think this discussion will frame the whole essay in a way that does a much better job of satisfying what reviewers wanted from the version you read here; putting together this text (which I am told was viewed as enjoyable, by some reviewers, but as missing the point of the requested corrections) *and the Postscripts*, you get an essay that talks about a core issue that I think is important to talk about here, and that (more importantly, for publication purposes) the reviewers viewed as the essence of this essay.

2 Introduction

The first question I should answer: given the abstract, which sounds like a fairly standard software engineering research paper abstract, except for the missing part bragging about a positive experimental evaluation, why is this an Onward! Essay and not a conventional research paper?

The answer is personal: I thought of this idea a few years ago, and have thought about making it a full research project off and on since then, but I never managed to actually do so. A real evaluation of this idea would require a good graduate student’s devotion, and, more importantly, actual human-subjects experimentation work to arrive at even a speculative and provisional scientific evaluation of the idea’s efficacy. I can’t think of a useful way to make a purely “extract code from public repos and analyze it” investigation, or (better yet) a purely mechanical experiment. Perhaps in a few years it will be possible to just have LLMs do it all?

I (read: "my lab") don't really do human subjects; I don't ever want to write IRB paperwork. I do work with *other people* who do human subjects, from time to time; so I could potentially convince one of those wonderful folks that this is a good project, and thus avoid doing IRB paperwork and still make this a real research paper.

But I don't have a huge enough committment to Test Driven Development to get me excited enough to try to talk someone else into spending valuable research productivity time on this idea; I've never used TDD in writing code myself, and I'm not sure I even believe it's a good development practice! Nobody I know and trust in the real software industry seems to do things in a TDD way, which is worrying. So it just doesn't seem worth it to commit to a serious Research Project, likely spanning over a year, just to explore this idea (I have lots of other ideas that I think are equally or more interesting, that I also don't have time to make happen).

So why not do what we senior researchers usually do in this situation? Shelve the idea, perhaps mention it to one or two people, and let it basically die? I've done this before; there are a good half-dozen implementations of test generation ideas I think might be interesting or effective in my TSTL Python test generation tool [14], none of which ever became a paper. They live only as obscure command line options to a tool increasingly few people are likely to ever use. It's sad, but life is short, and I don't think any of those ideas are likely to be *really* important, vs. somewhat, occasionally, useful. Perhaps that's the right approach with Mutation Driven Development, too.

Still, I didn't *want* to do that; this idea has been bugging me for years, and I have wondered if TDD is unappealing precisely for the reasons that made me consider the idea of MDD, which are discussed below. Perhaps if TDD delivered more of what it promised, it would be better, and more people I know would make use of it or some variation of it? I don't particularly *believe* in TDD (but I also don't hate the idea; it has a certain formal elegance and logic), but I do believe in mutation testing! Furthermore, Dijkstra's ideas of proving small pieces of code correct are also seldom if ever practiced, and certainly not by anyone I know in industry; should we therefore throw them out? Or work to make them practical and useful? I wondered about these bigger-picture questions, and didn't throw the idea out of my head, even if I didn't make any progress on it.

2.1 Daniel Jackson Made Me Do It

I recently read Daniel Jackson's *The Essence of Software* [15], which is a very enjoyable book. The supposed topic of the book is an approach to design that I thought was interesting, but that also seems more appropriate to UI-heavy software than the kind of code analysis/test generation software I tend to write myself, or the kind of embedded or systems code (e.g. file systems and compilers) I tend to test. The real value of the book was, for me, twofold: first, the long long set of notes after the main text is a delightful random walk through software engineering. Second, Jackson's musings include a major theme: we've become too careful in the software engineering research community. Given the heroic (for lazy computer scientists, at least; obviously many a research project in SE is not *that* much work compared to the work of serious systems people, much less biologists or physicists) effort required for making Real Science that Has Six Good RQs and Makes it Past the ICSE Program Committee, we tend to drop marginal ideas, especially if we as researchers are not inclined to be managers for an army of graduate students, but prefer instead to be less-productive/more-hands-on (here's looking at myself in a mirror), and have many fewer meetings and Slack conversations.

There was a time when SE (or formal methods, where I was living at that time) was far too accepting of a paper based on application of an approach to a single toy problem, plus a good writeup. This was too lacking in rigor, and there was a very sensible effort to increase the level of experimental support for claims about software engineering. I don't have a strong opinion on how this worked out for more methdological aspects of SE, but for the most part the increased experimental validation (and, later, statistical sophistication, or at least competence) was necessary to make work in more technical parts of SE, such as software testing, trustworthy. In a minor way, I think I helped encourage this trend in software testing, especialy under the influence of an unusually statistics-savvy and very capable graduate student, Rahul Gopinath (now faculty at the University of Sydney).

However, this increased rigor was not without a cost: some of the most important and influential papers in SE, as Jackson points out, did not rest on a basis of experimental support, in part because they were more raw ideas than a narrower scientific claim.

Now, many researchers take ideas that don't satisfy the current rules for *proper* papers and turn them into blog posts or at least tweets. I've done this. But there's an argument that this limits the audience to people who already know who you are and watch what you say. If your ideas are relevant to another community, one you aren't part of, this is likely to mean they never reach that audience.

In the past I've written Onward! Essays when the topic was clearly not one suitable to a standard research paper. This time, I'm writing an essay because that seems more responsible than writing a blog post, and a research paper minus the experimental evidence is just an essay[1]. The reader is free to try the idea out, with no guarantee that it will work, but really — we know that's the case with most research,

[1]This is a joke! The related statement that an essay is just a research paper missing experimental evidence is, of course, even less true; an essay is a "jackdaw, a magpie, a raven" [10], and while some good research papers contain the germs of an essay, the two forms are generally distinct, not that you would be so sure of that from the example at hand, the essay you are reading.

as well, except in the most unusual of circumstances. So, onward to the idea.

3 What is Test Driven Development?

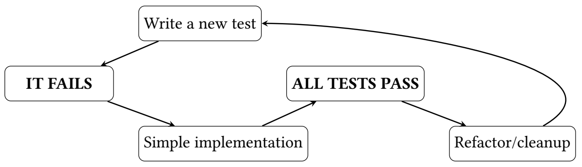

Figure 1. Test-Driven Development Loop

Test Driven Development (henceforth TDD) has been defined many times [1, 3, 9, 17], in many ways. Even if you're familliar with TDD, I would like to revisit the foundations for a moment, because the proposal made here is formulated with respect to a particular conception of TDD. Most definitions of TDD (including this one) properly present it as a *series of steps to be followed in development.*

1. Think of a behavior your software does not yet have, but that you want it to have.
2. Add a test that would pass if that behavior was present, but will not pass if it is not.
3. Run all your tests and see that the newly added test fails.
4. Make a *small* change that makes the test pass.
5. Run all the tests; make sure they all succeed now.
6. Refactor to remove duplication/generally clean the code from step 4 up so it doesn't make you ashamed of yourself.
7. Repeat (go to 1). Do not, generally, make changes other than by this process!

I added the first step, because you can't really write a test until you've thought of something to test. The basic process is as shown in Figure 1.

Why write software in this restrictive way? The benefits proposed vary, but I think can be summarized as:

- By making such small steps, you always have the confidence to move forwards. This may not be the first thing most people think of with TDD, but Beck's extremely influential book [3] really focuses on this. You never get stuck! You always have confidence! You are making progress at all times. Even if you can't actually implement the funcitonality at the moment (you haven't figured out how), TDD lets you write a *useful test* and gives you permission to produce a *passing, but fake, implementation* and thus have accomplished something. Beck thus offers strategies for getting from a fake (e.g., return a constant) implementation to a real implementation.
- Beck implies and everyone else seems to agree that this confidence isn't *just* about the small implementation steps; it's about the fact that those steps are accompanied by tests that you know will tell you if you mess things up. You can write code even when in doubt because not only are you tackling small pieces you should be able to get right, you have tests that you *know* will let you know if you don't get it right.
- More obviously, you should end up with a very good set of unit tests if there's nothing in the code you didn't write a test for, first. How could big holes arise in your testing? You should, for example, have very high code coverage (if you don't, what's going on?)
- Thinking about testing *first* should improve testability; you won't implement functionality in a hard to test way if you're forced to write the test before the implementation.
- Similarly, API users will be in your mind: you'll always use any APIs you make before you implement them. This is why in strict TDD you write the test in such a way that it *initially doesn't even compile* — you don't get to make the `.h` file then a test to fit it, you always see the signature/types/parameters "in action" before they are even defined. There's a strong expectation in the literature that this will produce better overall designs.

My personal experience with TDD consists of: (1) reading Beck's book as well as Grenning's excellent book on TDD for embedded systems [12], plus a few random papers on the topic (I have forgotten which ones), (2) trying it out for a few toy Python examples, and, (3) finally, requiring students taking CS 361 at Oregon State (the first of a two-term software engineering sequence) to use it to implement Parnas' KWIC (Key Word in Context) example [20]. The students hated it, and thought I was a maniac to insist on their following strict TDD discipline (and enforcing standard naming conventions to make the resulting code easy to analyze for coverage and (to get ahead of ourselves) mutation score)[2]. It's not a way I personally would write code, in part because of the odd, niche types of code I write (throwaway scripts for analyzing experimental results, and testing tools). On the other hand, the idea of TDD strikes me as attractive: I think most software would benefit from better (unit) tests, of course — I'm a "testing guy"; writing tests before implementation to ensure testability strikes me as a really good idea given all the problems code not designed to be testable can cause (for one thing, code not designed to be testable also, in my experience, has limited debuggability, extensibility, and maintainability).

[2]See the most negative comment here: https://www.ratemyprofessors.com/professor/1807710; that student was not unique, though I hope in the minority.

4 So What's Wrong With Test-Driven Development?

I think one thing is fairly clear: writing code according to faithful, careful, TDD practices will guarantee very high code coverage in the unit tests. The essence of TDD (as I see it) is that *you are not allowed to write a line of code unless you have a test that tests that line of code.* It is difficult to see how someone practicing TDD manages to produce code that is not covered by the tests. The refactoring step appears dangerous but if it is truly refactoring (even extending the idea to changes such as replacing a constant with the real computation) the semantics imposed by the tests should be preserved, and no new behaviors introduced. At most, very small gaps in coverage should be possible, and large gaps simply indicate "this was not really TDD."

From my perspective, the problem is that this *doesn't mean the code is extremely well tested*, which seems to be one of the core (at least implicit) claims of TDD. Software testing people know that while code coverage is generally *necessary* for good testing, it is far from *sufficient* [6, 13]. Admittedly, the *most* glaring risk of code coverage *is* obviated by TDD: it is very hard, when using TDD's rule that *first, you must see the test fail*, to write code that is run by tests, but not actually *checked* for any particular behavior [23]. For the most part, again, code is being written *so that a test will pass, which is currently failing*.

The problem is that checking one behavior of a small piece of code doesn't mean you are checking all the desired behaviors; TDD forces you to write *one* test that fails, that corresponds to every code-writing step. Even at the fine granularity of TDD, however, most code has more than one expected behavior. For instance, TDD will be naturally inclined to check that added code does something; it will be naturally disinclined (because of the nature of which unit tests are easiest to write, and the derivation of the original failing test from a desire to add some new functionality, in most cases) to check that it doesn't do anythng undesirable, and that it satisfies more esoteric conditions off the path of developing functionality: e.g., that it fails in the right way when called with invalid inputs. Later, we'll see that this corresponds to knowing not just the shape of your code, but the shape of the invisble river of program states that your code has to make its way through as it actually runs.

4.1 Enter the Mutants!

It seems plausible to reformulate one underlying goal of TDD as "ensure that after every step of coding, the functionality thus far implemented is tested solidly enough that changes can be made in the expectation that bugs will have a hard time getting past the tests." Code coverage is indeed guaranteed to be high, but the implied syllogism that *therefore* this goal will be met (which I think much TDD literature does indirectly propose to the developer) relies on code coverage being potent enough to, at least for small-scope code, guarantee highly effective testing. As we saw above, code coverage is important but too weak to make this kind of guarantee.

However, code coverage is not the only way to measure test effectiveness! A great many software testing researchers [6, 13, 19, 22], and a few high-profile software companies (e.g., Google, and Meta [4, 21]) believe that *mutation testing* [5, 8] can provide a more powerful guarantee of test quality. In mutation testing, the ability of tests to detect bugs is measured by injecting a large number of synthetic bugs, and seeing if they are detected. In mutation testing lingo, a detected bug is "killed" and an undetected bug is "not killed." Unkilled mutants are to mutation testing as uncovered lines of code are to code coverage: the things a developer aiming at good tests should examine and probably fix.

TDD by construction ensures high enough coverage that measuring coverage is likely to be fairly useless; the link betwen TDD and good mutation score, however, seems indirect enough that *mutation score can serve as a good way to inform TDD practitioners when their tests are not strong enough.* Because TDD ensures good coverage, and probably ensures at least *fairly good* mutation score, the primary human cost of mutation testing is reduced: well-done TDD should produce few unkilled mutants for developers to reivew. Add that TDD works by adding code in relatively small steps, and apply mutation only to the file(s) in which the newly added code appears, and the other persistent problem with mutation testing (lack of scalability when a very large number of mutants are produced) is also likely tamed: if you are doing TDD right, you should not be modifying many implementation files at once; in fact, you should probably only be modifying one at a time, most of the time.

Thus we have a new, very slightly modified, approach to TDD, shown in Figure 2 (I omit the step of coming up with a new behavior, because while this is essential it is *outside the loop*, and I want to emphasize the change in the *loop*).

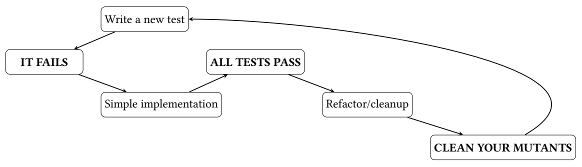

Figure 2. Test-Driven Development Loop

What does it mean to "clean your mutants?" It means that you 1) run mutation testing *on the file(s) you modified to produce the simple implementation* and 2) ensure that every mutant that is not killed has been inspected and rendered "clean." A killed mutant (one detected by your tests) is, by definition, clean.

Therefore, one easy way to make a mutant clean is to see that it indicates a weakness in your testing, and change the tests so it becomes a killed mutant. You'll get to see an inverted version of part of the TDD loop while doing this: the modified tests will start failing for the mutated code, once you add power to the tests, but they will still pass for the code you wrote for your actual implementation.

In some cases, you will see that the mutant is semantically equivalent to the correct code: no reasonable test will ever be able to detect this change. It's really just an alternative (perhaps ugly) implementation. In this case, you render the mutant clean by inspecting it. Once in a while, you might even refactor the code to remove the annoying mutant (perhaps an even cleaner implementation wouldn't have this variant? working at the very small scope of TDD does seem to give you the freedom to at least consider such detail-work). In other cases, you'll have to revisit these mutants every time you touch these files. That may be annoying, but it also means if you were wrong about the semantic equivalence, you get several chances to notice your mistake.

Calling this new version of TDD "mutation driven development" is, frankly, a bit much. The new step is intended to be fairly easy to apply, and to leave the basic structure of TDD unchanged. My defenses for the name are twofold: first, it sounds nice, and emphasizes the new idea. Second, I think there may be a real argument that "Test" in Test Driven Development is correct but not quite right; the real goal is to drive development by "small changes, likely to inflict only small, easily debugged and understood, semantic harm" — which sounds a lot like driving development by mutants. The tests are there to make sure the changes really are "mostly harmless" as Douglas Adams might put it; the tests are put in first because you always put in a safety net *before* you start walking the tightrope. But the small scope, easily debugged, carefully constrained, changes are the real focus, in this vision of TDD, which I think fits with Beck's original, essentially psychological, argument. I love tests! But tests are a means to an end, in TDD, though also extremely useful in the end for other reasons. MDD's purpose is to make sure the "mutants" to the code under development aren't simply "mostly" harmless but "almost guaranteed to be entirely harmless."

5 Is MDD A Good Idea?

I think every developer serious about avoiding bugs should be running mutation analysis on code as they develop and test it! So in that sense, I axiomatically think adding this to TDD would be great, if you're going to do TDD; you should do this in any case, for code where you want to make sure you have good tests.

Not everyone is going to be so inherently favorable to mutants, though, and as I see it the success or failure of the idea really rests on two empirical claims:

1. TDD ensures high enough code coverage and good enough basic checking of functionality that looking at unkilled mutants is not a major burden on developers.
2. TDD does not ensure such high quality tests that mutants never expose a weakness in testing.

As I said above, validating these assumptions empirically is not something I've set out to do. However, I don't want to leave you, the reader, without any evidence that this pair of hypotheses *might well* be true. Before I wrote this paper, I decided to take two examples of TDD-in-action and check if my claims held. Two is a tiny sample size, and I didn't randomly select examples, so this has *no scientific validity*. However, the results convinced me it was worth writing this paper.

I decided the best way to see if good TDD might benefit was to look at examples of people showing how TDD works. If TDD-produced code by people *showing off the process* had too many unkilled mutants for practical application, then MDD is probably overly burdensome (and, frankly, I wonder if TDD really works very well at the job of producing decent tests); if exemplary TDD on the other hand lacked interesting unkilled mutants, MDD is possibly useless, because the benefits will be limited. You'll have to clean many mutants before you find any that actually demonstrate weak testing.

I first looked for a blog post showing TDD in action for Python. The first one that came up in search results looked promising: https://pytest-with-eric.com/tdd/pytest-tdd/. I took the code, cleaned it up (the blog post has some code with typos) and followed along with the TDD example, applying mutation testing at each stopping point (just before a new test was added).

The versions I produced are available in github (https://github.com/agroce/mdd/tree/main/pytestwitheric). I used my own mutation testing tool, the Universal Mutator [7] (https://github.com/agroce/universalmutator, to generate mutants.

If you want to follow along at home, the easiest way to do so is to download the docker image in which I carried out my "experiments."

```
docker pull agroce/mdd
docker run -it agroce/mdd
cd ~/mdd/pytestwitheric/
```

The code is organized into versions I extracted from the blog post, numbered v1 through v6. It's a good idea to look at the code in each of these directories and match it up to the code from the blog post before proceeding; this will give you an idea of the lay of the land.

Then, to try out MDD, go into any directory, e.g. v3, and type:

```
cd v3
mutate string_manipulator.py
analyze_mutants string_manipulator.py "pytest"
```

You'll see the universal mutator tool producing a set of valid mutants of the version of the code, followed by actual

```
mutate src/LedDriver/LedDriver.c --cmd "gcc -c src/util/Utils.c src/LedDriver/LedDriver.c -I include/LedDriver/ -I include/util" --mutantDir LedDriver_mutants/
analyze_mutants src/LedDriver/LedDriver.c "make -f MakefileUnity.mk" --mutantDir LedDriver_mutants/
show_mutants notkilled.txt --sourceDir src/LedDriver/ --mutantDir LedDriver_ mutants | less
```

Figure 3. Commands for mutation testing of LedDriver.c

mutation testing. For the first version, v1, there are no mutants. Clean by the empty set! For version v2, there are two mutants, and both are detected. For version v3, there are six mutants, again all detected. For v4 the number of mutants grows to nine, still all detected. With v5 we get fifteen mutants (for about ten lines of code), still all detected. So far, we've gained nothing, but we've also done no real additional work (running the mutants takes less than 4 seconds, inspecting the result in case of 100% killed mutants is trivial).

And then with v6, we see that three mutants are not killed. Our score drops to 87.5% killed mutants. The three surviving mutants are all similar, so we can examine only the first one:

```
show_mutants notkilled.txt
MUTANT #1:
string_manipulator.mutant.18.py: ./string_manipulator.py:13
*** Original
--- Mutant
***************
*** 10,15 ****
          if not my_string:
              return "String is empty"
          if not isinstance(my_string, str):
!             return "Invalid input"
          new_string = my_string.replace(pattern, "")
          return new_string
--- 10,15 ----
          if not my_string:
              return "String is empty"
          if not isinstance(my_string, str):
!             return ""
          new_string = my_string.replace(pattern, "")
          return new_string
```

We wrote code for the case where an input to our function isn't a string, but we didn't write a test checking that this does what we expect when we pass a non-string. The test is easy to add:

```
def test_remove_pattern_type_int():
    sm = StringManipulator()
    res = sm.remove_pattern(3, "Eric")
    assert res == "Invalid input"
```

Once we add it (in my version v6.FIX), we're back to 100% killed mutants. Great!

This is simply a toy example from a blog post; what about more respectable demonstration of TDD?

Taking the first example from the excellent Grenning book [12], we can look at the final version of the code and tests for a simple LED driver.

```
cd ~/tddec-code/code
```

Producing mutants and analyzing them is now more complex (Figure 3). We kill almost 90% of the 98 generated mutants. But, importantly, we don't kill them all. The few mutants remaining show that, despite the tests including a number of checks for correct behavior when LED numbers are out of bounds, these tests *don't catch all the possible cases*: the ten mutants not killed show that some bounds checks are indeed tested, but others are not; changing the enum to incorrectly label the LEDs also can get past the tests (wrong numbering is a problem Grenning specifically calls out). The tests *mostly* catch these problems, but they don't catch *all* of them. And as far as I can tell, the mutants not killed all indicate these exact problems. Is looking at ten mutants a huge burden on a developer? I think not, at least in this case. As soon as you see one mutant showing bound checks aren't being fully probed, you probably will quit looking at mutants and proceed to strengthen the tests to cover bounds tests more effectively. After that minor effort, the likely outcome (for this code) is 100% mutation score.

Perhaps these examples aren't typical, but I think they are sufficiently compelling to make a case for trying out MDD, if you currently practice TDD. Let me know if it works for you (or if you are an eager graduate student dying to turn this sketchy idea into a real paper)!

6 The Mighty River: A Digression

This is an essay (thus a jackdaw, magpie, raven [10]), not a research paper, so we can turn our minds aside for a moment. The real reason I couldn't let MDD simply fall into the junkyard of never-pursued research ideas without making some effort to put it out in the world for serious consideration is that it fits into a larger vision I have of software engineering — the kind of vision you can only talk very very briefly about in a research paper in a short lyrical passage in the introduction that still annoys at least one ICSE reviewer.

Mark Twain's *Life on the Mississippi* is one of the best books ever written. And one thing it makes you *see*, that you may never have seen before, if you haven't read it, is how a river pilot had to know the river. Much of the first part of the book is taken up by the tale of young Clemens/Twain learning to be a river pilot; at first the task of knowing the ever-changing, infinitely complex, night and day and many-season Mississippi simply seems impossible. And every time young Twain thinks, cockily, that he *has* done this impossible thing, the irascible expert pilot Bixby tells him something else insanely difficult he has to know:

> "Then I've got to work and learn just as much more river as I already know."

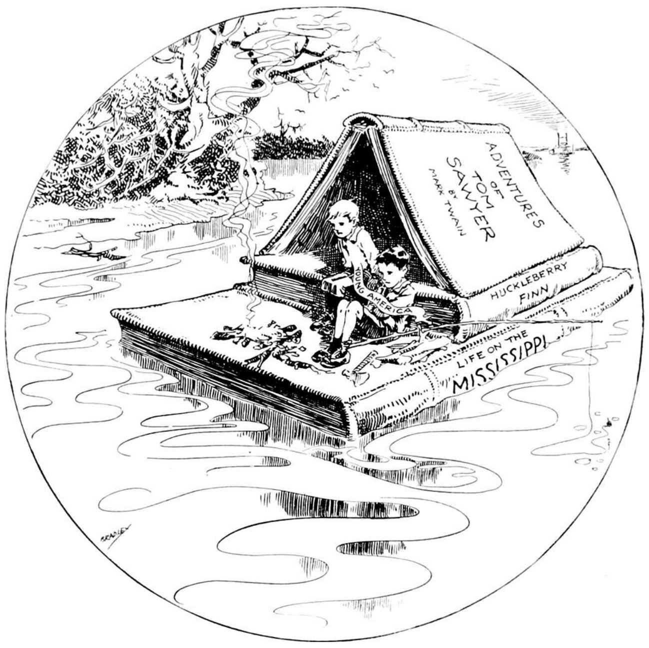

Figure 4. Cartoon by Bradley, cartoonist of the Chicago Daily News, 1917

"Just about twice as much more, as near as you can come at it."
"Well, one lives to find out. I think I was a fool when I went into this business."
"Yes, that is true. And you are yet. But you'll not be when you've learned it."
"Ah, I never can learn it."
"I will see that you DO."

This was written when the river had no lights or bouys, and we may think that it is fun to learn, but not very relevant to our work, not being river pilots, and the world having advanced to add lights and buoys by the time Twain wrote, much less by our far future day. But what Twain is really talking about is how well we have to know code, if we want it to be correct. You have to know your code like a pilot knows the river; and the river is ever-changing and our code, itself, and the context it lives in, is ever-changing.

I think the change in point of view from code coverage (handled by TDD) to mutation (added by MDD) is like one of those sudden enlargements of the notion of the river a pilot has to carry about all inside poor Mark Twain's young head: you thought you knew the river when you knew this (every bend and turn, every plantation and how it all looks at night when you can't really see it), but in fact you have to know *this* as well (how to read the river's dynamics from the height of every single bank you see). Figure 4 shows two intrepid young software engineers beginning their first journey on a small but real piece of code, perhaps binary search; larger rivers and larger boats await[3].

Coverage (TDD's forte; you not only have good coverage, but you have in your head the map of how you got that coverage, painstaking loop of failure, then new code, then pass, and once more failure to code to pass) is the river's basic shape; mutation (MDD's addition) shows you part of the changing shape of the river as it runs as well; mutation is a *text-driven* form of insight into the shape of the dynamic state space, not just the "textual space" of the code. The vision of TDD, as I see it, is to give a developer the confidence of an expert river pilot, not only the ability to avoid destroying a boat, but the bold power to still maintain speed while maintaining safety. The ideal of the process of TDD is to embody the teacher Bixby as a process to be followed; the promise of MDD is to make Bixby as demanding as the real Bixby Twain wrote about, and to make TDD form real river pilots. Does it work? I don't know; someone will have to try it on the actual river.

7 A Bigger Picture

If (it's a big if) MDD is a useful idea, I think it's a minor example of where we get many nice ideas: cross-overs. I don't do TDD and I think people who are interested in TDD in the research world are mostly (I would guess, I don't read much TDD literature) "process people" people. I think process people who care about TDD probably think a lot about testing, but don't think very deeply about code coverage metrics (beyond the basic "let's see what ran") or the oracle problem [2, 24] because these ideas are mostly siloed into the "tools and automated test generation" part of the larger SE world. I happened to start thinking about TDD because while I usually taught CS 362, the "testing and debugging" class at Oregon State, one semester I was asked to do CS 361, the "design and process" class. TDD seemed like the most "hardcore testing people" thing to learn for that class, and sounded interesting, even if I had no particular interest in agile/XP/any of that stuff. I didn't know at the time Grenning had a very good book applying all this to the embedded world, which I *did* care about.

Because I'm always thinking about mutation and mutants, I wondered if TDD gave rise to really great mutation scores, since it not only encourages coverage, but pushes for code to be written in tandem with good functionality probes. That mildly across disciplines thought, following an inspection of the mutation scores for 80+ students over their TDD versions of KWIC, gave rise to the notion of MDD. But TDD is in the "process" and "people" and "agile" boxes, not in any of my boxes ("test generation", "fuzzing", "test suite evaluation measures"), so I put it aside. And had I not read Jackson's

[3]One great advantage of this viewpoint is that it makes writing and even testing software an extraordinary adventure, not a boring slog we're frankly eager to turn over to coding assistants.

book, likely MDD's only existence would be as a thing I occasionally mentioned to people I knew.

In her fascinating new history of the Renaissance [18], Ada Palmer notes that "major discoveries often come from someone studying a different topic asking new questions" — in her epistemological (and thus extremely pleasing to a "testing guy") version of history, this is a key to understanding: you need to look at a "thing" from so many viewpoints that the connections between viewpoints become clear. This can even expose "new" facts (this event happened less than a week before this other similar event, and was done by the same guy, that means they are really *part of the same story*) even where technically, everyone has "known" about the new facts forever, it's just that nobody ever actually *looked* at those facts. I think Jackson's point about research that is just hard to evaluate, or where the mashup of fields involved may make a Busy Professional Professor Researcher reluctant to do anything with a basic partly-out-of-my-field idea, comes into play here: if the standards are always to be rooted in rigorous experimental work (RQs 1-6 evaluated using the right statistics and a compelling base of data from human subjects or mined from software repositories, pleasing to everyone on the ICSE PC!), we're going to miss those connections sometimes. That, in my opinion, even if MDD in particular isn't a very useful idea, is a shame. Even if I'm not, *the rest of you* are probably having lots of slightly out-of-band good ideas all the time, and I fear you're burying them because they don't quite fit the model of the way we do "research" now.

8 First Postscript

I should really have added a bit more evidence for MDD, as one reviewer requested; not first-class research paper level evidence, but something more to suggest that MDD might be interesting.

I half-joked above about doing experiments on MDD without humans via asking LLMs to write code using normal TDD and MDD, and seeing how the two compared. I think this isn't actually, at least right now, a good idea for a real paper on MDD, at least MDD for humans. The way LLMs write code is sufficiently alien that even if MDD helps an LLM, it's unclear this means it actually helps humans, or even clear that MDD itself is the key: simply mentioning mutants might shift the input's position in "token space" enough to produce better results.

Nonetheless, it's not the worst idea, and it's very easy to do a tiny experiment along these lines. In the agroce/mdd repository are the results of asking anonymous non-logged-in fresh ChatGPT instances to solve a simple problem (chosen truly at random by using the first programming task of a modest scope I heard anyone mention, over the next day) eight times. All eight times the prompt instructed ChatGPT to apply TDD, but four of the prompts asked ChatGPT to also apply MDD. I will note that ChatGPT's notion of "mutant" was sometimes not traditional mutation testing, but "small variants" of the code it came up with that invovled more than the usual single-token modification. I suspect this may be part of the future of mutation testing, and "check for a generated set of bugs" is the heart of MDD, not a particular set of mutation operators. I view "ask an LLM to inject a bug at random" as an expensive, untried, mutation operator but a perfectly reasonable one for many uses of mutation testing.

The resulting code was of mixed nature, thoroughness, and used a variety of names; the tests for the code also varied considerably. Interestingly, in two of the non-mutant cases, the code was also *wrong*. In one of those two cases, the code actually had a test that caught the error; in the other case, the pure-TDD tests passed, but the code was wrong[4]. All other implementations were, as far as I could determine, completely correct (the fairly thorough TDD test suites, all of which passed, helped check this, and I manually confirmed the duplicate bug found in the two pure-TDD versions was not present in any of the six other implementations). Is this remotely conclusive? Obviously not. For one thing, for a real experiment, you'd want to run this dozens of times, on a variety of LLM models and over a large number of programming problems of various sizes (this one is implementable by a single very simple function if you take the prompt narrowly, and can make use of an auxillary function or two if the LLM decides to be ambitious).

However, I think it's a small, independent, confirmation that MDD might be worth exploring. Who knows, perhaps it's useless for humans, but asking LLMs to consider program mutants when co-designing code and tests helps keep the LLM from missing simple bugs! I feel that further elaborating the experiment with more programming problems and more generations of each kind would undermine the point of this essay, however, and might lure me into just making this a workshop paper at Mutation 2026, so I stopped at four experiments per "treatment," while things are safely statistically meaningless.

9 (Concluding Unscientific) Postscript

Ok! You've read the rejected revision now. Why was it rejected? The reviewers fundamentally wanted me to do a few things (tone down a few bad jokes, which I did, fix some typos, which I did, and introduce some more experimental indication, however cursory, that MDD might be a good idea, which I don't really think I did very well, and which I have addressed in the first postscript), but mostly they wanted more on the problem of *going outside your research comfort zone.* Now, that I consider this work going outside my zone at

[4]If you look at the prompt in the repository or docker image, the issue is that when given a list with duplicate paths, the incorrect versions can return duplicate paths, instead of just returning one copy of each directory, `f(["A/B", "A/B"]) == ["A/B", "A/B"]`.

all says something disconcerting about either me or the field: this is a paper about a use of program mutants in testing. The two things I have worked on most over at least the last decade of my career are software testing (which has been my broad focus since at least 2008/9), and mutation testing (which has been a frequent theme since at least 2012 (as a heavy user of mutation testing in evaluating my testing work) and 2014 (when I started doing a lot of "inside-baseball" work on mutation testing methods themselves). The disconnect is really methodological (people experiments! uh oh!) more than subject matter.

The essay above talks about the reasons for either abandoning such potentially high-effort, low-payoff work in a scientific career. I think one way to look at is as a narrow, within-scientist, version of working inside a Kuhn-style "normal science" paradigm[5] or facing up to a crisis inducing revolutionary science [16]. This is a huge, complex topic, and I'm more of a Karl Popper man myself (working scientists may tend to be, since Popper does portray the working scientist as, ideally, a mix of Sherlock Holmes and Sextus Empiricus, heroic and stoic and majestic), but a point somewhat implicitly made in Peter Godfrey-Smith's excellent survey of the philosophy of science, *Theory and Reality* [11], is that besides the large-scale advantages of working in normal science (having a clue what to do next, having more guidance as to which experiments would be interesting vs. just doing a bunch of "let's collect facts and hope something comes of it" which even a Popperian framework can degenerate into with too much skepticism), there are probably *career benefits* to living as a scientist in a period with a dominant, useful paradigm: you can be fairly sure what you'll be doing tomorrow, and to a large extent how it will be received by the community. You know what to teach your graduate students how to do, and which general things are most important for them to know about, and which general things are less important for them to know about. You *can easily communicate with your colleagues because you share large, important, unspoken assumptions about what it is that you are about, and how you should go about it.*

Stepping outside your normal modes of science, whether diving into a people-centric research topic by yourself when you are normally a tools-and-full-automation researcher (only bringing people into the picture when working with a people-focused colleague on occasion) or writing an essay that tries to both describe a possible human-side research project and the problems in pursuing such a program when it lies outside your core interests, means among other things, risking a communication breakdown. I have had (especially counting journal papers) a large number of papers over the years that went through some form of conditional acceptance requiring me to satisfy explicitly stated requirements for the final submission.

In all previous cases, this process has been smooth and ultimately successful; while sometimes the conditions required substantial re-writing or major, time-pressured, experimental effort, it has always been very obvious what was to be done and how to do it. Moving outside research comfort zones risks making this process an unpleasant surprise, because it is harder to communicate desired changes when reviewers and author do not share context. In this case, the effect was probably more about my intentionally pushing the boundaries of what an Onward! Essay does than about software engineering subfield mismatches (though I note that the nature of Onward! Essays also means you are much more likely to have reviewers who do not live in your little, pastoral, clearly-fenced-in, mini-paradigm; it is possible this also contributed to my not understanding what was wanted, and going off about Mark Twain and the romance of riverboats, not about the problems of stepping outside one's scientific subfield). However, the similarity is that the fact of a mismatch (between reviewer expectations and the author's little world) makes it much easier to miscommunicate, frustrating the effort to get anything done. For me, with tenure many years ago, and few grand ambitions left other than to think about, teach about, and occasionally write about how to test software until I retire or die, perhaps funding some students on occasion, this isn't important at all. And, to a large extent, the friction of subfields having different modes and people wanting to cross boundaries needing to learn some of the cultural and linguistic patterns of their new homeland or vacation destination is probably impossible to get rid of, and so useful to have that it would, as Kuhn can be read as saying, be a disaster if we did get rid of it.

But the risk of miscommunication frustrating something valuable is greater the more every subfield mechanistically adopts a *template of research* that is often used not to guide reviewers in understanding a submission, and evaluating its true underlying value, but is more used as a *bureaucratic checklist* to quickly and painlessly eliminate "outside bounds" work. I talk above at length about why (other than, perhaps, laziness) I didn't turn the idea of MDD into a full-fledged research project, but perhaps the heart of the problem is: I didn't want to rope a people-centric colleague into joining in work in which I had more interest than faith; but I sure didn't want to try to write a people-paper, even granting I was willing to endure IRBs and rounding up undergraduates to try MDD, without a native speaker of the language of those software engineering people who do experiments that revolve around methodology and human behaviors. Some of this was wise fear of not doing a good job of evaluation, but an equal amount was fear of doing a perfectly reasonable evaluation but not putting it in the proper straitjacket of form

[5]Technically, the use I'm making here of Kuhn is more appropriately based on Lakatos' notion of "research programme" or Laudan's notion of a "research tradition," but everyone has heard of and has an idea of what Kuhn said, and few people have read Lakatos or Laudan, unless they have a specialist's interest in the philosophy of science [11], and for the metaphor the distinctions don't matter that much.

and terminology (which is a hazard even within a research comfort zone, depending on the set of reviewers assigned to a paper).

The problems that lead to the surprise rejection of this essay in 2025 are probably inevitable and certainly nobody's fault but mine; we need to think hard about whether our approach to research rigor has ossified to the point that it adds a set of arbitrary shibboleths to this necessary friction and discourages researchers from stepping outside their comfort zone when they think they have a relevant insight for another part of software engineering, not because speaking a foreign language is always hard, but because the natives are a bit xenophobic these days, and who needs the hassle? Daniel Jackson and Ada Palmer, among others, have convinced me that *software science* needs the hassle, even if we individual scientists, all too often, don't.

I call this concluding postscript unscientific because, beyond the narrow point about how to get good research, I suspect that even if the price for software science turns out to be small and acceptable, the price for scientists is, whether we know it or not, high. Nabokov talks in many places, most notably in some of his *Lectures on Literature*, about the "Precision of Poetry and the Excitement of Science" — and Nabokov, while of course mostly remembered as one of the greatest and most singular of novelists, was an accomplished lepidopterist (for a time the de factor curator of Harvard's Museum of Comparative Zoology, founded by Louis Agassiz, another scientist with poetic notions). What too-narrow siloing gains us in our careers, it may lose us by diminishing the excitement that is the best reason to try to understand anything, including software. This cost is hard to count, but I think it is important for the software research world to make some room for slightly awkward forays by generally respectable scientists, lest we lose track of the fact that this should, in addition to other concerns, remain *fun*.

The trick is to have fun, but not be *too unscientific*. Wild speculations and half-baked ideas have many venues now: blog posts, tweets, coffee breaks at conferences, (for those prominent and senior enough) sometimes keynote addresses, and, of course, Onward! Essays; and these are all good. But they also tend to require too *little* in the way of formality and experimental validation, and so may offer up too much that isn't so much half-baked as quarter-baked or raw, and on the other hand when the ideas presented *are* valuable and worth exploring, they may offer too little in the way of empirical, rather than rhetorical, argument, and so be unlikely to be adopted and further explored. To conclude: is there a place for the, let us say, three-quarters-baked idea, in software science, a place with enough rules to force some validation of an idea, but without the strictness of the most dyspeptic ICSE PC member[6]? And if not, how can we make one?

In short this essay argues:

- Early in the history of software engineering, there was a lack of rigor in requiring evidence for ideas, in some cases when the ideas clearly called for, and could be supported or refuted by, experimental or observational data.
- This resulted in a large number of weakly supported (and likely false claims) but also gave room for a nascent field to explore hard-to-evaluate ideas that have had a foundational impact on software engineering.
- With maturity, subfields, particularly those where fully automated experiments, larger-scale human studies, or evaluations based on mining large numbers of software repositories were possible, adopted a more rigorous stance with respect to both the kind of evidence required and the "standard template" for how research papers presenting such evidence would communicate both the hypothesis/idea and the evidence.
- This reduced the negative impact of ideas that could have been refuted by investigation, but also limited the field's ability to explore new concepts in respectable, highly visible venues, siloing work of a more exploratory nature.
- When a mismatch was present between a researcher's ideas and experimental (and communicatory) expertise, the idea was often simply abandoned. This may be a serious problem, since it is often the case that high-impact ideas arise at the borders between previously disconnected intellectual fields, in both science and art.
- Something should be done about this, if possible, without relaxing rigor to the point of again accepting too many papers arguing from extremely anecdotal evidence or featuring only toy problems.
- (Adding mutation testing as a rigorous, fully integrated, part of Test-Driven-Development *might* be an idea worth exploring.)

References

[1] Dave Astels. 2003. *Test driven development: A practical guide.* Prentice Hall Professional Technical Reference.

[2] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2015), 507–525.

[3] Kent Beck. 2002. *Test driven development: By example.* Addison-Wesley Professional.

[4] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry - A Study at Facebook. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE '18).* IEEE, 268–277. https://doi.org/10.1109/ICSE-SEIP52600.2021.00036

[6]You may answer, with Scrooge, "are there no workshops?" but in fact workshops and less-prestigious venues often adopt some of the same strictures as ICSE, at the whim of reviewers, and with much less reward for jumping through the hoops.

[5] Timothy Budd, Richard J. Lipton, Richard A DeMillo, and Frederick G Sayward. 1979. *Mutation analysis*. Yale University, Department of Computer Science.

[6] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 597–608. https://doi.org/10.1109/ICSE.2017.61

[7] Sourav Deb, Kush Jain, Rijnard van Tonder, Claire Le Goues, and Alex Groce. 2024. Syntax Is All You Need: A Universal-Language Approach to Mutant Generation. In *ACM International Conference on the Foundations of Software Engineering*.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11, 4 (Apr 1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[9] Chetan Desai, David Janzen, and Kyle Savage. 2008. A survey of evidence for test-driven development in academia. *SIGCSE Bull.* 40, 2 (June 2008), 97–101. https://doi.org/10.1145/1383602.1383644

[10] Brian Doyle. 2017. *Reading in Bed (Updated Edition): Brief Headlong Essays about Books and Writers and Reading and Readers*. ACTA Publications.

[11] Peter Godfrey-Smith. 2009. Theory and reality: An introduction to the philosophy of science. In *Theory and reality*. University of Chicago Press.

[12] James W. Grenning. 2011. *Test Driven Development for Embedded C*. Pragmatic Bookshelf.

[13] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. 2014. Coverage and Its Discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) *(Onward2014)*. Association for Computing Machinery, New York, NY, USA, 255–268. https://doi.org/10.1145/2661136.2661157

[14] Josie Holmes, Alex Groce, Jervis Pinto, Pranjal Mittal, Pooria Azimi, Kevi n Kellar, and James O'Brien. 2018. TSTL: the Template Scripting Testing Language. *International Journal on Software Tools for Technology Transfer* 20, 1 (2018), 57–78.

[15] Daniel Jackson. 2021. *The essence of software: Why concepts matter for great design*. Princeton University Press.

[16] Thomas S Kuhn. 1997. *The structure of scientific revolutions*. Vol. 962. University of Chicago press Chicago.

[17] E.M. Maximilien and L. Williams. 2003. Assessing test-driven development at IBM. In *25th International Conference on Software Engineering, 2003. Proceedings.* 564–569. https://doi.org/10.1109/ICSE.2003.1201238

[18] Ada Palmer. 2025. *Inventing the Renaissance*. University of Chicago Press.

[19] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[20] David Lorge Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (1972), 1053–1058.

[21] Goran Petrovic and Marko Ivankovic. 2018. State of Mutation Testing at Google. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE '18)*. 163–171. https://doi.org/10.1145/3183519.3183521

[22] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does mutation testing improve testing practices?. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 910–921. https://doi.org/10.1109/ICSE43902.2021.00087

[23] David Schuler and Andreas Zeller. 2011. Assessing Oracle Quality with Checked Coverage. *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation* (2011), 90–99. https://api.semanticscholar.org/CorpusID:12581175

[24] Matt Staats, Michael W Whalen, and Mats PE Heimdahl. 2011. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd international conference on software engineering*. 391–400.