Genetic Programming As If You Meant It

Bill Tozier

Abstract My title is a direct nod to Keith Braithwaite's exercise for software developers, "TDD as if you meant it". Like the original exercise, what I describe here for genetic programming may seem onerous and difficult to most experienced theoreticians or practitioners. I'll make the case that this approach suggests—and demonstrates—a broader systematic framework in which genetic programming is far more likely to *provide value to its users* than we find in a quarter-decade of exploration. Along the way, I'm forced to address deep philosophical problems in the program of genetic progamming research and practice more broadly: *viz.*, what it is "for" and when it is "done".

Key words: keywords to your chapter, these words should also be indexed

1 "TDD as if you meant it"

As far as I can tell, Keith Braithwaite first described his training exercise for software developers in 2009. The target of the exercise is "Pseudo-TDD": the noted habit among software developers who claim to "know" and "do" test-driven development as part of their daily work towards a sort of thoughtless approximation of the technique.

I should note that a number of agile software development practices share informative relations to genetic programming's dynamics[1], but in this work I'll focus

Name of First Author
Name, Address of Institute

[1] I imagine there is an Engineering Studies thesis in this for some aspiring graduate student: Genetic programming and agile development practices arose in the same period and more or less the same culture, and both informed by the same currents in complex systems and emergent approaches to problem-solving.

on those of TDD. In particular, test-driven development (or more accurately "test-driven design") *when done correctly* can break down the complex design space of a software project into a value-ordered set of incremental test cases, focus the developers' attention on those cases alone, inhibit unnecessary "code bloat" and feature creep, and produce low-complexity understandable and maintainable software.

TDD *as such* is a rigorous process, to the point where it can be described as "painful" (though also "useful") by experienced programmers. The steps are deceptively easy to trivialize and misunderstand, especially for those whose habits of thinking about code are ingrained:

1. Add a little test
2. Run all tests and fail
3. Make a little change
4. Run the tests and succeed
5. Refactor to remove duplication

Each stage offers a stumbling block for an experienced programmer, but the most salient for us now is the overarching flow of implementation (or "design") that it imposes: Each cycle, and also the overarching design process, begins with the choice of *which little test should next be added*; each cycle ends with a rigorous process of refactoring, not just of the new code but of the *entire cumulative codebase* produced over all iterations of this cycle. The middle three steps—implementing a *single* failing test and modifying the codebase *by just enough* so that all tests pass—feel when one is working through them as if they could be automated easily. The *mindfulness* of the process lives in the choice of next steps and (though somewhat less so) of standard refactoring operations.

Braithwaite's exercise does an interesting thing to surface the formal rigor of this approach. In it, the participants (willing, of course, because the exercise is a *kata* or "refresher" for experienced software developers to hone their skills) are asked to implement a nominally simple project like the game of Tic-Tac-Toe, given an *ordered* list of features to implement and the artificial restriction that they must go farther than normal TDD practice asks. Rather than producing a suite of tests and a self-contained codebase, they are forced to use *only* refactoring of code added to tests to produce their eventual "codebase". In other words, no code can be "produced" until *duplication in code added multiple passing tests* provides a warrant for refactoring it out. Further, a facilitator patrols ongoing work and deletes *any and all code not called for by a pre-existing failing test*.

Words like "irritating" and "annoying" crop up in participants' accounts of this onerous backtracking deletion the first few times it happens... as one might imagine. But as Gojko Adzik emphasizes in his descriptions of workshops he's run, the results of "design" of even simple algorithms in this artificial amplified setting seems much more *open-ended* than it would if the software were built within the typical framing of habits and assumptions that an experienced programmer carries along with her to any project.

A number of contextually positive benefits are attributed to agile software development practices, and to TDD within that suite of practices. But the one that brings us here today is that aspect surfaced particularly in Adzik's account of Tic-tac-toe:

> By the end of the exercise, almost half the teams were coding towards something that was not a 3×3 char/int grid. We did not have the time to finish the whole thing, but some interesting solutions in making were:
>
> - a bag of fields that are literally taken by players—field objects start in the collection belonging to the game and move to collections belonging to players, which simply avoids edge cases such as taking an already taken field and makes checking for game end criteria very easy.
> - fields that have logic whether they are taken or not and by whom
> - game with a current state field that was recalculated as the actions were performed on it and methods that could set this externally to make it easy to test

In other words: innovative approaches to the problem at hand began to arise, though there wasn't enough time to finish them in the time alloted for the exercise. The analogy to our collective experience to date with genetic programming should start to peek through at this point—though the thoughtful reader will hopefully wonder what utility there is in an analogy drawn between two similarly unsatisfying outcomes.