GP As If You Meant It: Real and Imaginary User Experience

William A. Tozier

Abstract Each chapter should be preceded by an abstract (10–15 lines long) that summarizes the content. The abstract will appear *online* at `www.SpringerLink.com` and be available with unrestricted access. This allows unregistered users to read the abstract as a teaser for the complete chapter. As a general rule the abstracts will not appear in the printed version of your book unless it is the style of your particular book or that of the series to which your book belongs.

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

1 Pickering's "Mangle of Practice"

Genetic Programming (and the broader body of work to which it belongs, which I prefer to call "generative processing", also abbreviated "GP") embodies a very particular *stance* towards the scientific and engineering work of modeling, design, analysis and—least of all—optimization. A case could be made that the resistance we have all recounted towards GP from the technical and lay audience has little to do with the technical results so amply demonstrated over the last quarter-century, but rather from discomfort among that audience in GP's particular "way of working" on problems. There is a tacit assumption, even among GP theorists and practitioners, that science and engineering are "rigorous", or even successful, only when they proceed through ordered phases of

1. conceptualization, the "vision thing"
2. planning
3. design
4. architecture

William A. Tozier

5. implementation
6. testing
7. debugging

Very few scientists or engineers would admit that any *real* project has followed this narrative structure in practice, but in every phase of work life from fund-raising to published reports the story is framed in that way, especially in scientific work: Because of work gone before, an idea or insight was had. The idea was framed as a formal hypothesis. The hypothesis (and current Best Statistical Practices) suggested an experimental design obvious to any in the discipline. The experimental design was undertaken, the data collected, the hypothesis tested, and we can be confident of its veracity because, well, you just heard me say "Best Statistical Practices", right?

Under trivial term substitutions—"cost–benefit analysis" and "requirements document" for "hypotheses" and "experimental design", for example—the same narrative can be used to describe any institutional project management and public policy planning process as well. The flow in every case is essentially from *vision* to *plan*, *plan* to *implementation*, *implementation* to *verification*, and *verification* to *validation*.

Of course, nobody "really believes" this narrative who has ever done any of the work. It is a matter for another day to draw parallels with the social construction of religious belief.[1] There have been numerous philosophical challenges to this artificial narrative of course, from Peirce and Dewey nearly a century ago, to Kuhn and Lakatos in the 1970s, and more today.

Andrew Pickering's is the one I'd like to use here.

2 "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[2], but in this work I'll focus 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

[1] Paul Veyne's excellent *Did the Greeks Believe in Their Myths?* might be an interesting starting point, though.

[2] 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.

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.