Surviving Legacy Code: Microtechniques

A summary:

0. I'm not even going to mention Mikado Method, because that's obviously a great idea and you all need to read that book next.
1. Start with undo, and invest as much as it takes.
2. Reserve capacity, schedule the work, set a timer, start anywhere.
3. Subclass to Test, then Replace Inheritance with Delegation.
4. Extract Pure Functions, and let the chips fall where they may.
5. Play the long game, ignore "ROI", focus on controling cost, let good things happen.

Start with Undo, and Invest As Much As It Takes

- If you have undo, then nothing hurts "too much". Reduce the cost of failure because you probably can't reduce the probability of failure.
- **Safety matters more than flow and speed**, because mistakes could be *absolutely devastating* in a legacy environment. It could cost ridiculous amounts of money, time, energy, and reputation.
- Be careful: undo in the code is easy, but undo in the environment is difficult. In the worst case, you have to take snapshots of things or wait minutes for `git` to store things for you. It could take a lot of disk space. I've done this with Screenflow recordings, and sometimes it takes 2 minutes, and that's fine with me.
- If you don't have automated version control, then do it by hand. It feels like waste, but it absolutely isn't. If you knew which mistakes you'd make, then you'd have already won the lottery, so don't pretend that you *can* know nor that you're *expected* to know.

Reserve Capacity, Schedule the Work, Set a Timer, Start Anywhere

- Rescuing legacy code — and even writing new code that uses legacy code — amounts to unplanned, seemingly-unbounded work. You can't "call your shot" on which parts of it will deliver value. Make peace with that uncertainty and don't try to deny it. Therefore, treat it like "Research &

Development", meaning reserve some fixed portion of your capacity (example: 1/2 day per week) to invest in legacy code, then hope for the best. Review the results every month or so to decide whether you change the level of investment.

- Make a regular appointment with yourself and your team to work on legacy code, so that you actually remember to do it and you actually make time to do it. Protect that appointment like you would protect any other valuable appointment. Yes, put it on your calendar every single week.
- When you sit down to do the work, start doing it in small increments. Set a timer for 30 minutes, then take your hands off the keyboard when it sounds, then write down everything in your head, then stop.
- You can't predict where value will come from, so pick a place to start, then just start. Start anywhere. Follow the pain in the design, then it will lead you directly to the worst parts. Bad news: there's no good place to start; good news: there's no obviously good place to start, so just start!

Subclass to Test, then Replace Inheritance with Delegation

- Cracking open a class and changing it carries risk. In order to optimize for safety, I assume that I can't change the class yet, and so I create a "testable subclass" and use that as a staging area for sketching out prospective changes.
    - I probably have to make `private` parts of the class more visible. I usually just make things `public` quickly, but not with legacy code: I increase visibility only as little as I absolutely have to in order to get into place the tests I want to write.
- A tangled class usually leads to run-on tests where it's hard to distinguish the action from the (probably excessive) setup. This makes it easy to tangle ideas in the tests in precisely the same way that implementation details are tangled in the code.
    - In the tests, I try to move setup code into the constructor of the testable subclass. (In Java I use instance initializers for this, then decide when code should flow into the constructor.) This way I clarify which parts of my test are "arrange" and which are "act". This helps me see which parts are tangled that should be separated.
    - Each test initializes its instance a slightly different way. This code usually flows into new constructors and creation methods (the Named Constructor pattern). These help me understand and document which parts of the class we should pull apart from one another.

- If a testable subclass only changes data, then that tells me that I don't want hierarchy, but rather different ways of creating the class. Maybe this means a literal Factory and maybe it just means better-named constructors. Maybe it means that I have "configuration" and "operation" tangled in the same place and I need to separate them into "configure, then start". (This is an example of the Lifecycle/Operate tangle, which is an example of the Container/Item tangle. Have you seen JDBC's `ResultSet`?)
- Over time, replace inheritance with delegation/composition, for all the usual reasons. The tricky part is leaving behind a suitable abstraction.
    - First, don't panic. If you don't know a good abstraction, then extract a bad one.
    - Second, remember that abstraction simply means "hide details", so look for irrelevant details in the tests and take those as a signal that a class knows too much about its collaborator. Remove details one by one until you reach a minimum-knowledge interaction, in which a class knows the bare minimum it needs to interact with its collaborator. The more details you remove, the more general it gets, and the more reuse the pieces become!
- I don't know what the functional programming version of this technique is. Perhaps in functional programming languages, you don't have this problem?

Extract Pure Functions, and let the chips fall where they may

- The Killer Technique is extracting pure functions, because **nothing gets you to The Finish Line sooner** and **nothing hurts quite as much**.
- Give yourself plenty of extra time, space, and be prepared to throw your work away, because with this technique you will bite off some random amount of stuff, and you won't know how much you've bitten off until you start chewing.
- Bad news: it's "the long way" to reach a great design. Good news: it's "the sure way" to reach a great design. If you survive the technique, it rewards you.
- When I say *pure function*, I mean referentially transparent, meaning that you can fully determine the function's behavior from its input parameters and that the function stably computes the same output for the same input. It means that passing parameters by reference and by value always yields the same results. In OOP terms, the function does not change the state of any of its inputs (and, as is explicit in Python, the receiver of a method is

an input!).

- Extracting blocks of code into pure functions, rather than merely using the Composed Method pattern, forces us to confront every last detail of every tangled dependency and every run-on function. We are forced to see that our 1,274-line function really manipulates 39 separate pieces of data. We are forced to see which batches of statements have to happen before other batches of statements, which limits what we're allowed to reorder without changing the observable behavior. We do not get to hide these issues in fields on objects where we can pretend that the design isn't so bad. We see it all.
- Extracting pure functions helps us by creating "free agents", which are functions that are free to move anywhere, including specifically to the object on or the namespace in which they belong.
- Extracting pure functions helps us by freeing functions from system state, so that we can more easily write clear, unambiguous tests for them. This is horribly tedious work, but it's not *hard*. As we crank through the permutations of inputs, we start to notice patterns that show us exactly where to break things apart. For example, we see that *these* 7 inputs cause the output to change, but that *those* 2 inputs are mostly irrelevant, except for a few cases. Clearly, *these* inputs and *those* inputs belong on separate classes/modules. Once we break them apart, our single batch of 500 tests (300 of which look mostly the same) because two batches totaling about 120 tests (most of which capture important differences from the rest).
- Extracting pure functions helps us by separating code from its context, so that we can assemble smaller, more generic, more reusable modules that we can begin to treat as pure components. From here, we can take advantage of abstraction and modularity so that we can achieve the aims of good design: locality of change, avoiding the ripple effect, adding features by (mostly) adding variations instead of (mostly) changing existing code.
- Extracting pure functions can involve a lot of mechanical, repetitive, mind-numbing work. This makes it dangerous for the impatient programmer who wants to get to the point. Remember: it took years to create the mess, so you need to expect to spend the occasional hour filling spreadsheets with 790 permutations of 14 different inputs and computing the 790 corresponding sets of 6 different outputs.
- Extracting pure functions takes implicit/hidden duplication in the code and makes it impossible to ignore, thereby encouraging us to remove the duplication, creating the barely-sufficient structure that the code desperately needs.
- Pure functions satisfy the *substition model*. When programs break the substitution model, they are immensely more difficult to reason about. The more pure functions in your design, the more of your system you can clearly reason about and the more you isolate the "difficult parts" so that they mostly stay out of the way of the rest of the system. This strongly relates to the Dependency Inversion Principle.

Play the long game, ignore "ROI", focus on controling cost, let good things happen

- Normally, we Agilists focus on value over cost. Legacy code is one area where we probably can't measure value *at all*, except to measure loss of value from the high cost of making a mistake. For that reason, when working with legacy code, I focus on controling cost. This explains why I optimize for safety.
- Often, in legacy code, we don't know where the value will come from, so we stumble around in the dark until the situation becomes clearer. Over time, we hope that we can better identify areas of high value. This explains why we refactor so carefully and with small, reversible steps, so that when we notice a more valuable direction, we feel comfortable dropping everything to go that way.
- Often, in legacy code, even if we know where the value lies, we have no idea how much it costs to realize that value, so we can't measure what matters: profit. (We can focus on value when we have a way to understand burn rate, if not overall cost.) This explains why we control cost while we hunt value, and then prepare ourselves to change directions as needed. Value can come from unexpected places, so we plug away, protect against downside risk by limiting investment, then let good things happen.

References

Ola Ellnestam and Daniel Brolund, *The Mikado Method*. One of the now-classic texts on how to approach rescuing legacy code systematically and with high discipline.

Michael Feathers, *Working Effectively with Legacy Code*. Still *the* classic text on the topic.

J. B. Rainsberger, "Relative Include Paths and the Slow, Certain March Towards Legacy Code". One specific design choice that leads towards legacy code, and how to avoid it.

J. B. Rainsberger, "Brewing Espresso and Legacy Code". A real-life, non-software example of the difficulties involved in wrestling with a legacy system.

Special Offer

Would you like to see how I approach legacy code in more detail? Sign up for *Surviving Legacy Code* during the pre-release period and receive a ***significant discount***. I'm posting content through late 2016 and recording more in early 2017. Not only do you have the chance to take the course without paying full price, but if you have code that you want me to refactor, I will do it! All this for less than the normal price of the course.

(Dearest European friends: I apologize for this, but my distributor will add VAT to the price of the course when you check out. I beg you not to be annoyed by this.)