Programming Practices: Part 1 – Watching from a distance

by BRIAN HARRY MS, XOPOSHIY, SMACLELL, RON HARDING, CARL D, IDJI, CRAIG LEBOWITZ, LOÏC BAUMANN, KEVIN, TOMAS NILSSON, CHRIS RATHJEN, JALF, WALTER SASSANO, WHATKNOTT, BILL SORENSEN, JEROEN RITMEIJER | MARCH 9, 2010

Предложенные ответы I Звезда блогов Оценщик-новичок

I figured I'd start the series with a more abstract post about what watching me code looks like. It was interesting to observe myself doing it because I didn't really realize the degree to which I do some things. If I were to summarize my overall approach to coding with a simple description, I'd call it "Annealing" – from the Encarta dictionary: "metallurgy craft transitive and intransitive verb to subject an alloy, metal, or glass to a process of heating and slow cooling to make it tougher and less brittle." Or in more of a software sense, "Simulated Annealing" – from Wikipedia: "a technique for searching for a solution in a space otherwise too large for "ordinary" search methods to yield results".

What does that mean?

Let me try talking about it as a set of principles and techniques:

1) Working code every few hours – I find I can't stand to have code that is in pieces all over the floor. I don't ever like to be far from code that works. My coding is defined by a sequence of "stable" points that are no more than a few hours apart. A "stable point" is a state where all of the code compiles, it runs and it does something useful. It may not be anywhere near doing what the ultimate application is intended to do but it does some fraction of it. A quantum of work transitions from one stable point to the next.

I find that as I approach a problem, I think about how I am going to build the code in bite sized chunks. It's kind of funny to watch but if I get a few hours into something and I'm not very close to a "stable point", I actually get anxious. I start getting nervous and looking for ways to stub out or cauterize parts of the application so I can get back to something I can build and hit F5 on. I think part of this is related to why I got into programming in the first place – I'm a huge fan of the "instant gratification" that you get programming. I can think of no other profession where you can start with nothing and in a few short hours have built something really useful. But more practically, it's an important way to make sure you don't get lost in the weeds of a solution.

2) Prove the concept – When I'm getting ready to leave a stable point, I find the first thing I want to do is map my path to the next stable point. I choose some specific, containable goal – add a dialog, a menu item, move an operation to a background thread, add a control to a form, something. Before I enter the chaos between stable points I want to prove to myself that I know how to get to the next one. I often start, particularly if I'm working with API surface I'm unfamiliar with, by doing a rapid prototype of the core concepts that I need. I don't care what the code looks like. I don't modularize anything. I don't worry about comments, variable names, code organization, error handling, performance, anything. I just want to touch each of the core APIs that I need to use and prove that I can get access to the information that I need and understand roughly what I need to do with it.

I usually get it all the way to running code and step through it in the debugger so that I can see all of the information flow and ensure that I understand all of the side effects, etc. To a first approximation, I then throw it all away. Sometimes, I'll just comment out the routine(s) that I wrote and keep them around a while for reference and a source of code snippets. I step back and conceptualize an organization of the code that feels clean and I then start writing the code "for real".

At some level this might feel like a waste of time but I believe it is a huge time saver for me. It usually doesn't take me more than 15-20 minutes or so to explore the key concepts of the few hour leap from one stable point to the next and I often learn a ton in the process and make significant shifts in my approach as a result. It saves me from getting an hour or two into something and realizing that some assumption I made at the beginning was invalid and I need to redo everything I just did. I think it also helps ensure that I end up with better organized code when I'm done.

3) Refactor, Refactor, Refactor – I never really realized how much I just naturally do this. I'll bet the net effect is that I write any given stable point to stable point quantum 2 or 3 times in the few hours I work on it. I start with a conceptual picture of the algorithm but I don't try to plan out everything. I start writing the code organized in a way that feels the "cleanest" but as soon as I realize something's not right, I refactor. I'm constantly splitting methods, reordering code, reorganizing data structures, building abstractions, etc. This is the main reason I started by talking about annealing – because that's what the process feels like to me. I keep jiggling the code, gradually reducing the temperature until all of the crystals line up just right and I have a very clean, maintainable, reusable, fast, robust, … implementation.

I now realize that I used to do this even before "refactoring tools" were invented but I am finding some of the refactoring tools to be very useful in saving me some typing – though they have a long way to go to make me really happy. The VS ones, at least, don't automatically do everything I want and don't always have the flexibility for me to control it. Examples – Extract method often leaves me with a lot of clean up to do.

- It doesn't give me the ability to control parameter ordering (and as you'll see from this series, I'm anal about everything). I am very particular about the ordering of parameters.
- It doesn't allow me to control the types of parameters to the method. Every method should have an abstraction or contract of what it operates on and what it does. I might want to type a parameter as a base class of the passed object. I might want to pass a one or more members of an object that is used rather than the whole object (because the type of the object is not part of the method's contract).
- It doesn't allow me to extract to a different class.
- It rarely puts the method where I want it in the class (more on this in future posts).
- Lastly, I find that before I extract the method, I actually want to type the call to the extracted function. It's part of making sure I understand the factoring of the code and what the contract is. I wish I could type the call site, then select the code I want to extract and say – extract it such that the call matches what I just typed. But instead, of course, it extracts it, adds another call site, which I immediately delete and then go fix up the parameter types and order, etc.

As I write the code, I comment lightly. Because I refactor so much, I don't want to waste time writing and rewriting comments, method contract descriptions, etc. But once I've gotten the code just right, I polish it. I go back over all of it making sure I like the abstractions and the flow, I comment everything and I move all of the methods that I've written to their permanent resting place in the class – I'm very particular about the order of methods in a class (did I mention I'm anal about everything? :)).

4) Step through everything in the debugger – Before I start polishing a quantum of work, I walk through everything in the debugger. I conceive and write or manually execute test cases for every interesting scenario and I step through all of them in the debugger. It's not sufficient that the code works. I'm a firm believer that you don't actually understand your code if you don't understand exactly how and why it works and the only way to do that is to step through it in the debugger and see what it is actually doing.

You'd be stunned to realize how often the results of code match what you expect but the execution path is nothing like what you expect. This is one of my greatest secrets to getting high quality, high performance code that I never have to come back and revisit again. When it's done, it's done done :)

5) Paper and pencil – The last thing that I'll mention that's perhaps a bit odd and I think you'd notice if you sat and watched me code is that I always have a pencil and paper in front of me. As I'm coding, I'm constantly drawing pictures of data structures, writing out sample data, hand walking through the algorithm on paper

to see how things work. I suspect it's just a quirk of mine but there's no better way for me to think through the algorithms as I go. When I'm done with a quantum of work, I always have a sheet of paper full of scribbles. I generally extract some of the value and then just throw the paper away. Some of the scribbles become part of the documentation – pictures, algorithm descriptions, etc. Some become the source of test cases that I keep to serve as regression tests in the event that I ever need to come back to the code.

Summary

At a very high level, that's what watching me code looks like. Some of what I've written here, I think of as quirks of my personality – heavy use of pencil and paper, for example. But most of it I consider important practices to produce high quality code efficiently. I can imagine a lot of people looking at this and saying, "wow, that sounds expensive" and if you are banging out a one off project with a short lifetime, it's really not worth it. But if you are writing code that is part of a significant application, is going to live for years and potentially have other people needing to understand it, I believe these practices are invaluable.

I'm not sure I've captured everything. Perhaps in the discussions that follow and subsequent posts I'll think of a few more "high level" things. We'll see.

Brian
