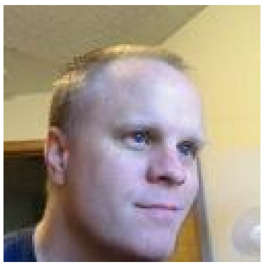
My Ruby Regrets

My Ruby Regrets

This guest post is by **Jeff Langr**, who has developed software for thirty years, mastering many other languages (including Smalltalk, C++, Java, and currently C#), but just not Ruby and Python… yet. (Ever?) He owns the consulting and training company Langr Software Solutions, and codes full-time as an employee of GeoLearning. Jeff is the author of close to a hundred articles on software development and three books, including Agile Java and the very-soon-to-be-published Pragmatic Programmers "book," Agile in a Flash.



Ruby guru? Hardly. Even though I first "learned" Ruby about nine years ago, perpetual Ruby newbie is a far more correct term. In those nine years, I've coded here and there on a number of throwaway scripts. For each separate effort, I would get past novice struggles to the point where I felt reasonably comfortable with the language. But just as I started to enjoy high levels of productivity, the job was done and it was back to "enterprisey" Java coding. Months later, sometimes more than a dozen, I'd work another Ruby script. The cycle would start again at a slightly higher proficiency level than the last time I started. My feeble brain would struggle to recall any remnants of my Ruby memory. It's impossible to become a guru this way!

I'm once again working on a side effort in Ruby, pairing with a few other developers to build a testing framework. It's intended for inbetweeners (my newly coined term for QA people who are willing to get a little technical) to easily put together Watir-based tests. As usual, it's a small effort, a few days at most. As usual, we (me and my pair partners) decided to just "hack at it." Never mind adhering to good OO design concepts, and never mind test-driving the code. Why?

- it's a small effort that should stay small over the long haul.
- we're (re)learning Ruby, which means we're experimenting in irb and pasting over code that appears to work.
- it's a test tool itself. Do we really need to write tests for a test tool?

I'd also paired to develop a couple comparably scoped (that is, small) Python scripts in the earlier few months. For similar reasons, we eschewed test-driven development (TDD) and good OO design on those efforts too.

Fortunately, our constant companion Humility is one of the best teachers. It doesn't take long to generate legacy code (code without tests) to the point of regret. In the case of my three most recent scripting efforts, the point of regret was somewhere after a half day of code cranking. What happened? Looking more closely at our three arguments for hacking it out, we can easily find flaws.

We quickly slammed out a couple hundred lines of Ruby code. Small, yes, but we quickly found ourselves often unsure about what code was where, and we knew that we had a good amount of unnecessary duplication. But our only verification mechanism was to manually run the test framework against the handful of scripts that it drove–a cycle that took about 90 seconds. We couldn't make the rapid, ten-second changes that we wanted to–no one wants to wait more than a minute to verify every two-line code change.

As far as the learning Ruby part, we know that there's a Ruby way to express code, but our novice Ruby brains tend to code it a little more familiarly first. Without unit tests, we've been a lot slower to transform the ugly procedural-isms into tight Ruby-way constructs. Unit tests are great for letting you slap together a method's implementation, and then safely play with improving its expressiveness.

From the design standpoint, it’s a little more work to create proper classes, but inevitably we found that the lack of good design just made for crappier code and more duplication. We also found that mixins, while very beneficial, can create some interesting challenges and confusion if you’re not careful with them. Moving to a more OO solution simplified our codebase and gave us more flexibility.

We quickly wished we had built more tests. As we got more frustrated with dumb mistakes that took a while to pin down, we started building some TAD (test-after development) tests. At least these tests let us put a stake in the ground, but it’s unlikely that we’ll have the time to go back and completely cover the code. Had we started with TDD, we would have avoided getting bogged down in the defects. We also would have been able to keep the code base small, and exhibiting minimal duplication and confusion.

I didn’t do things the right way the past three times I’ve built small scripts. But next time I code in Ruby–no doubt enough months away that I’ll have forgotten much of what I re-learned–I hope my feeble brain won’t have forgotten, once again, this important lesson. Don’t wait to test-drive, and don’t hack just because you can.

Feel free to ask questions and give feedback in the comments section of this post. Thanks and Good Luck!