My experience with automated testing for Rails

Evan Silberman

June 11, 2007

(In lieu of the 4th quarter iteration report, I suppose.)

Last summer, I began developing a Ruby on Rails-based quiz bowl tournament statistics and organization suite I dubbed "Taft." After several abortive attempts, I finally managed to understand how to use Rails, but what I still didn't understand was testing. It seemed boring and pointless. I could just test my program in the browser, right?

Well, as I went on, and read various articles in Rails blogs about testing, I realized that knowing for sure that the components of my program were working correctly at all times. So a few months ago, I hesitantly wrote my first test. And I realized how inspiring it was when you didn't just *think* that your program worked, but when the computer tells you it *knows* your program works. Tests, simply put, ensure program correctness, as long as the tests are written well. It's an extremely helpful when after every change, you can reassure yourself that you haven't broken anything.

Testing for Ruby on Rails is managed by a Ruby testing library called Test::Unit. The purpose of unit tests is to ensure that all of an object's methods produce the correct output or change the object's state in the proper way. A test can consist simply of a call to a method and an assertion that the return value is equal to what we expect. Then, repeat so every method and line of code in the program is covered. Rake, a Ruby build tool, allows all the tests for Taft to be run simultaneously and any failures are immediately apparent.

Tests are great, but for the test suite to be truly useful, it must cover as much of the code as possible. Currently, about 50% of the system is covered by the tests. My goal is to eventually reach 90% coverage (I don't know if I can manage 100%). I analyze my code coverage using another testing tool called rcov. Rcov runs all the tests for the project, then produces an HTML report showing the lines of code for each file that were executed when the tests ran. This is incredibly useful when trying to figure out what tests I need to write and whether my tests are testing useful things.

Many people advocate a test-first, or behavior-driven, development process. In other words, write a test, which will fail because you haven't written the code yet. Write just enough code so the test will pass, then write more tests and more code iteratively until you're satisfied with your program. Since I wrote a whole lot of code before I started testing, I can't really adopt this practice at this stage for Taft, but I may try it in the future when starting new projects.