technicalities: "not rocket science" (the story of monotone and bors)

Feb. 2nd, 2014 10:25 pm

graydon2

A technical note about a program I wrote last year called bors and some of its ancestry. This is *excruciatingly boring* unless you happen to build software for a living, in which case I recommend taking a minute to read it.

Thirteen years ago I worked at Cygnus/RedHat on a project with a delightful no-nonsense Australian hacker named Ben Elliston. We were faced with a somewhat nightmarish multi-timezone integration-testing scenario that carried a strong penalty for admitting transient bugs to the revision control repository (then CVS). To try to keep the project under control, Ben, Frank, and possibly a few other folks on the team cooked up a system of cron jobs, rsync, multiple CVS repositories and a small postgres database tracking test results.

The system had a simple job: *automatically maintain a repository of code that always passes all the tests*. It gave us peace of mind (customers only pulled from that repository, so never saw breakage) and had the important secondary benefit that engineers could do their day's work from known-good revisions, without hunting someone else's bug that accidentally slipped in. Because the system was automated, we knew there was no chance anyone would rush to commit something sloppy. You could commit all the sloppy stuff you wanted to the *non-automated* repository; it would just be declined by the automated process if it broke any tests. Unforgiving, but fair. At the time I felt I'd learned an important engineering lesson that any reasonable shop I went to in the future would probably also adopt. It turns out that if you set out to engineer a system with this as an explicit goal, it's not actually that hard to do. Ben described the idea as "not rocket science". Little did I know how elusive it would be! For reference sake, let me re-state the principle here:

> ***The Not Rocket Science Rule Of Software Engineering:***
>
> *automatically maintain a repository of code that always passes all the tests*

Time passed, that system aged and (as far as I know) went out of service. I became interested in revision control, especially systems that enforced this Not Rocket Science Rule. Surprisingly, only one seemed to do so automatically (Aegis, written by Peter Miller, another charming no-nonsense Australian who is now, sadly, approaching death). At the time Aegis was not a distributed system, and while I was using Aegis on my own time I knew there were interesting possibilities if one went into the area of *distributed* revision control, as I saw my friends using bitkeeper. I wanted to blend the two ideas, so I got to work on a system of my own called *monotone*.

Monotone was named as it was because I wanted to really pursue this concept of "monotonically increasing test coverage". Revision branch-inclusion was based on certificates that could easily be published out-of-order by testing robots. Etc. Etc. I figured this was the future. Of course, numerous adventures followed eventually resulting in mercurial and git, monotone was swept aside, the world of revision control changed, and so forth. Hurrah.

Along the way something weird happened. "Continuous integration" became a standard practice, and eventually "continuous deployment", but *very few sites seemed to be following the Not Rocket Science Rule* in their own codebase. That is, everywhere I saw "continuous integration" in practice, it was being done in the wrong order: code being accepted *before* testing (leaving a potentially broken tree), or tested in isolation and then integrated on the basis of that test (with no guarantee that the integrated combination works). Continuous integration seemed everywhere to be used only to learn (rapidly) when the tree was broken, not prevent it breaking in the first place. I was dumbfounded, annoyed, saddened.

So when it came time, a few years later, to scale up contribution beyond a few of us on Rust, I decided I needed to enforce the Not Rocket Science Rule in order to keep the code under control. I repeatedly explained it but nobody else was biting at the idea, so as "technical lead" I wound up implementing it myself. The result was a small script called *bors*.

Bors implements the Not Rocket Science Rule against a combination of a buildbot test farm and a github repository: it monitors pull requests, waits for reviewers to approve them, then for each approved revision, makes a *temporary integration revision* which extends your integration branch by the proposed changes. It tests that temporary revision and advances your integration branch to point to the integration revision if and only if the tests pass. If the tests fail, the integration revision is discarded and a comment in the pull request is left showing the failures.

You can see the Rust instance of its work queue here and the Servo instance here. An example successful integration is here and an example rejected integration (at least at the time of writing) is here. This rule is, I reiterate, not rocket science. I'm banging on about this because it's *so amazing* to me how little tool support seems to exist for it. I had to write it myself! Its effect on a fast-moving, delicate project is extremely beneficial. Trunk is never broken. It does mean that your integration cycle time is bounded by your test cycle time, and it means that some changes take a number of attempts to integrate (bors supports priority markers to help you manually tune the integration order).

On some projects, test time is too long for this strategy to work on each revision, at least without defining an integration-test subset. This is certainly everyone's chief initial concern, and it's legitimate, but in my experience it's a bit like people objecting to testing (or typechecking) in the first place because writing the tests and types will take too long: you'll spend much more time fighting the bugs if you wait and discover them later. I *strongly* recommend anyone who works in software *try this arrangement* to see what a difference it makes.

Crossposts: http://graydon.livejournal.com/186550.html

Tags: tech