frog hop

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

graydon2
My Website

November 2025

S	M	T	W	T	F	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30						

Most Popular Tags

capitalism - 1 use
gender - 1 use
history - 1 use
infrastructure - 1 use
justice - 1 use
management - 1 use
novelty - 1 use
politics - 1 use
rust - 1 use
software - 1 use
tech - 13 uses
writing - 1 use

Page Summary

ewen - Not rocket science
robbat2.livejournal.com - *(no subject)*
lindseykuper - *(no subject)*
elsmi.livejournal.com - *(no subject)*

Active Entries

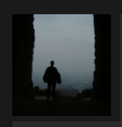
1: flywheels, again
2: A note on Fil-C
3: there absolutely are free lunches, among them "all life on earth"
4: snuffle / salsa / chacha
5: consensus
6: losing language features: some stories about disjoint unions

Style Credit

Base style: Crossroads by **branchandroot**
Theme: Green Light by **krja**

Expand Cut Tags

No cut tags



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).

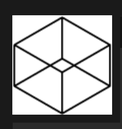
On some projects, test time is too long for this strategy to work on each revision, at least without defining an integration-test subset. This is certainly everyone's chief initial concern, and it's legitimate, but in my experience it's a bit like people objecting to testing (or typechecking) in the 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

Previous Memory Share Next 8 comments Reply

Flat | Top-Level Comments Only



Not rocket science

Date: 2014-02-03 10:10 am (UTC)
From: **ewen**

Thanks for sharing the history, and the example of how the tools work -- I particularly hadn't realised that was the origin of monotone. As you say "don't break the build" is such a simple concept (and one I try hard to maintain manually in any project I work on), but it's often left to social enforcement at best. It's odd that in a 100%-test-coverage and continuous-integration world that the tooling isn't also prioritising an "any trunk revision is a viable release candidate" test-first-then-merge-to-trunk.

Ewen

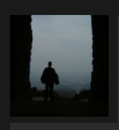
Link Reply

Date: 2014-02-04 02:07 am (UTC)
From: **robbat2.livejournal.com**

It's a shame that Not Rocket Science Rule didn't get a better early start, I sadly still see many cases where it would have save much time later on (possibly with the mandatory writing of non-trivial test cases before new code).

But also fun to run into somebody with time in Redhat's boom era, hats off to you for that - I have a signed copy of Bob Young's autobiography from November 1999, from my early start in open source.

Link Reply Thread Hide 1 comment

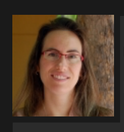


Date: 2014-02-25 02:11 am (UTC)
From: **graydon.livejournal.com**

Aegis has a delightful wrinkle on this: each changeset has to not just pass all existing tests, but also come with a new test, independently tracked and associated with the change, that must fail before the change, and pass after it.

I found working within this structure -- enforced and automated -- extremely pleasant, the few times I worked on aegis-managed projects. Wish it had spread more.

Link Reply Thread from start Parent



Date: 2014-02-04 04:39 am (UTC)
From: **lindseykuper**

Good stuff, Graydon. Thanks for writing this down.

Link Reply

Date: 2014-02-04 03:18 pm (UTC)
From: **elsmi.livejournal.com**

And the pull-request support in Travis-CI is so close, and yet so far... it tells you whether the PR passed when merged into master, at the time when the test was run. But if other PRs have been merged since then, then the result may be stale. One could trivially detect this, but who cares, right?

Link Reply Thread Hide 3 comments

Date: 2014-02-04 05:28 pm (UTC)
From: **graydon.livejournal.com**

Yeah, Bors actually uses the status hooks they put in place for Travis. I was seriously surprised how close, yet how far.

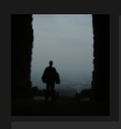
Link Reply Thread from start Parent Thread Hide 2 comments

Date: 2014-02-04 06:31 pm (UTC)
From: **elsmi.livejournal.com**

Though, looking at the bors-reviewed commits you linked, I will say that it is kind of nice that Travis goes ahead and tests everything immediately, so when reviewing you can refer to the test results (or skip reviewing things that are actually still broken). Best of both worlds would be to test immediately when the PR is submitted, and then when it passes review, check whether those test results are still valid and re-use them if so; or, if master has moved on, re-run.

(I guess this is probably more interesting for "slowish project that gets lots of PRs from random outsiders" than "fast-moving project under active development by small team".)

Link Reply Thread from start Parent Thread Hide 1 comment



Date: 2014-02-04 06:40 pm (UTC)
From: **graydon.livejournal.com**

No, it's more that we weren't (and still aren't, on that test farm) fully isolating test environments, so "run random code someone opened a PR containing without any review" would be a nice way for a stranger to take over one of the buildslaves.

In 2014 I'd probably start with docker and do container-per-PR on a public cloud, which I think is now all the new CI stuff works.

Link Reply Thread from start Parent

Previous Memory Share Next 8 comments Reply

Flat | Top-Level Comments Only