Hudson for Everybody Else

Joe McMahon
mcmahon@blekko.com
Twitter: @joemcmahon

Project problems

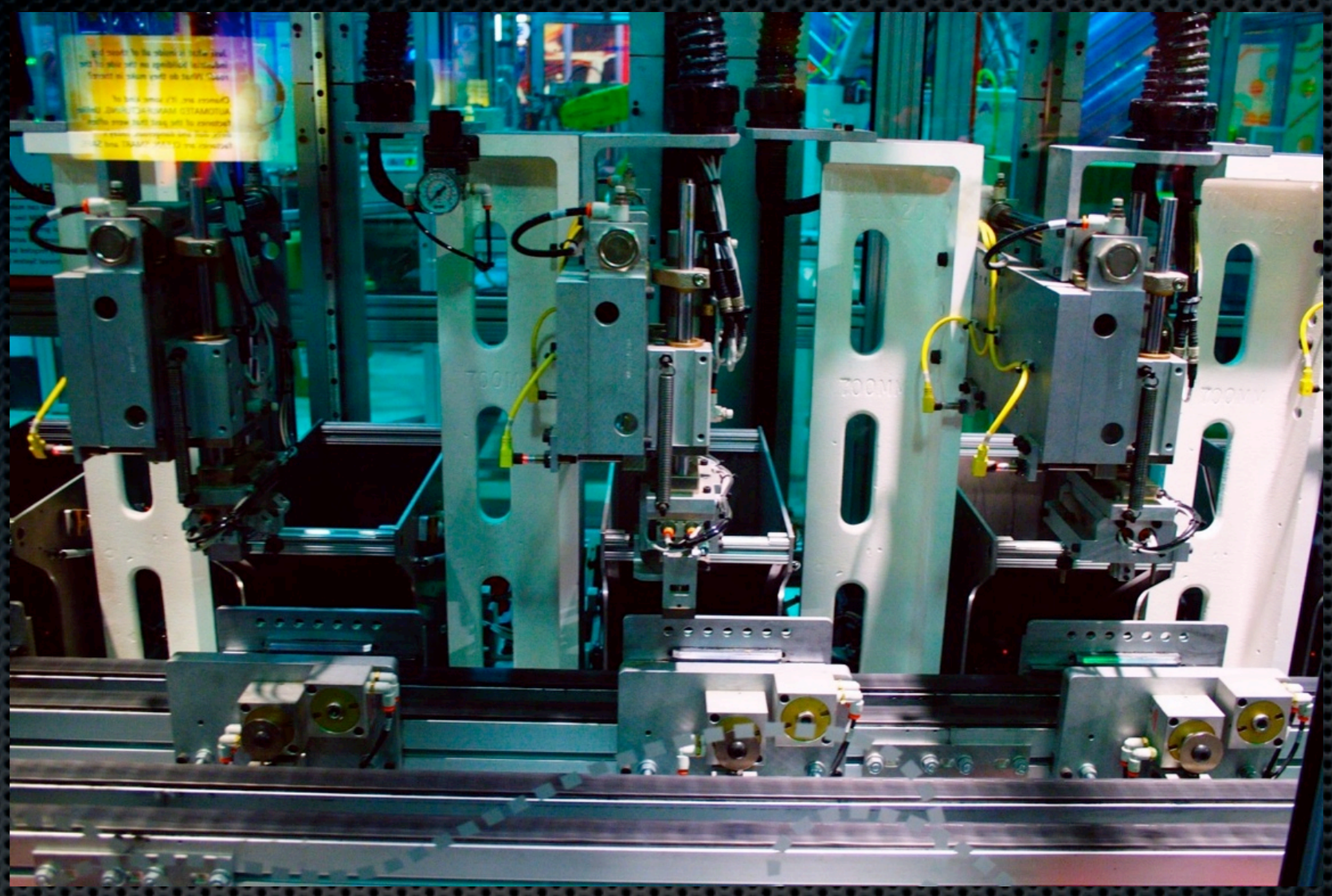
- project sometimes builds and runs, sometimes doesn't

- folklore builds: if you know how to build it, it builds, and if you don't it will never build

- checked-out code is broken: is it me or is the code?

- tests are failing: is this a real problem or is it bad tests?

- tend to happen because of lack of exposure



Continuous integration

Code always gets built and tested when it’s checked in

Never “goes bad” without it being detected: build fails, tests fail

Being built by machine, so there has to be a way to build it that doesn’t require “knowing how”

This is a great idea, and of course I don’t want to have to write something to do this

Hudson

Simple to install, configure and use - everything from the web API, no XML config files

Monitors source code control, schedules and runs builds, records logs and test results

Can automatically archive and track build products (e.g., tar or zip file), associated with build that produced them (not required, but can be useful)

Trend analysis of tests, memory usage, disk space usage

Plug-in architecture; many provided by the user community, not hard to write

Free-form builds allow you to set up and run a series of shell commands to do a build

Makefiles or scripts equally acceptable and usable to do a build

Tests can be tracked and analyzed automatically if you can produce JUnit XML (more later)

What do we need?

Reliable source for code

Revision control system best alternative - many (14 or so) supported

In particular, CVS and Subversion support is built in; git (and github), Mercurial, and others available via plugin

Even a plugin that lets you treat a directory as a source for code

Dependable tests

Need some kind of a test suite - anything is OK as long as it passes completely

No “oh ignore that one” tests allowed! TODO, skip, fix, or remove failing tests. (See Test::More perldoc for more details)

Again, has to output JUnit XML - which we can do

Skipping definitely-bad tests

```
SKIP: {
    skip “These tests are broken”, 2
        unless $ENV{RUN_BROKEN_TESTS};
    is $sometimes_wrong, $right_answer, “this sometimes fails”;
    like $messy_answer, qr/strict requirement/, “as does this”;
}
```


Causes the tests to not run at all

Use cautiously! Blocking off swathes of tests may lead you to trust a test suite that’s actually not doing much testing.

Making this conditional allows you to try the tests again when you want to

Your goal should be to not have *any* skipped tests: try to make them at least TODO-able

TODO for tests that will pass in the future

```
# These tests will run and fail, but the script won’t fail
TODO: {
    local $TODO = “Reason these are failing now”
        if $ENV{ENABLE_TODOS};
    is $sometimes_wrong, $right_answer, “this sometimes fails”;
    like $messy_answer, qr/strict requirement/, “as does this”;
}
```


TODO actually does try to run the tests - make sure that the tests running and failing will not compromise code following them.

TAP::Formatter::JUnit must be told that you’re OK with it if TODOs pass ($ENV {ALLOW_PASSING_TODOS} = 1); otherwise JUnit flags them as errors!

TODOs preferable because you’re actually exercising the code

Process

Some kind of a working build process
Doesn't have to be done via a specific build tool, but it does need to be

- orderly (no "fix a lot of exceptions then this works")

- repeatable (if you follow the instructions, it should always work, assuming the code is good)

- executable by a non-human (ideally just an unconditional series of commands)

Perl test integration

Hudson can look for files in the build tree, so globbable names will work

Leading Java-style double-star says “look everywhere in the tree for a file like” (e.g. **/TEST-*.xml looks for any file in the build directory that matches TEST-*.xml)

All we have to do is get the TAP translated into JUnit so Hudson can use it

TAP

```
$ prove -v -Iblib/lib t/007itersmode.t
t/007itersmode.t ..
1..22
ok 1 - use Tie::Hash::MultiValue;
ok 2 - The object isa Tie::Hash::MultiValue
ok 3 - in iterators mode
ok 4 - empty hash has no keys
ok 5 - no iterators yet
ok 6 - one item, one key
ok 7 - keys are as expected
ok 8 - two items, two keys
ok 9 - keys are still as expected
ok 10 - two items (one multiple), two keys
ok 11 - keys are still as expected
ok 12 - first multivalue is as expected
ok 13 - last multivalue is as expected
ok 14 - no more multivalues is as expected
ok 15 - single value as expected
ok 16 - no more multivalues is as expected
ok 17 - empty hash element as expected
ok 18 - one item, one key again
ok 19 - keys are as expected
ok 20 - deleted item is as expected
ok 21 - single value as expected
ok 22 - out of values as expected
```


Test Anything Protocol

Originated in the Perl tests, but formalized into a specific protocol via Test::More

ultra-simple: ok / not ok, small extensions for TODO, skipping tests, etc.

testanything.org is the TAP Wiki - worthwhile reading

Building

Standard build tools work fine - Hudson allows you to add as many “steps” as you want to a build

Each step of the build acts like an independent invocation of the shell

Anything you can type at the command line will work (make, ./Build, even xcodebuild); multiple lines OK

Getting JUnit XML

We've got TAP output from the tests - with prove we even can get timing info as well

All we need to do is transform this into JUnit XML

Painful text sausage-making, so let's avoid it

TAP::Formatter::JUnit

Wednesday, June 23, 2010

Original version was written by Graham TerMarsch (YAY GRAHAM!) - lot of thankless and not intrinsically interesting parsing work

I added a bunch of code to clean up the produced XML

- Hudson's XML parser is picky, and follows the XML parser standard of "fall over when you see the first error"

- Changes needed to make sure that no odd characters were found outside of CDATA sections and that only the basic entities (& < > ") were present (&#hexdigit; OK, and we translate the weird stuff into that.)

- Use with the --formatter option of `prove` and we automatically get all the tests run by that invocation recorded in one JUnit XML file - easy to get a whole test suite as one file

```
#!/usr/bin/perl
print "test_hudson:\n";

# Note: CPAN distros with no tests will need to have test.pl added via patch.
if (-e 't/TEST') {
  # Special case for libwww.
  print expanded_template(qw(t/base t/html t/robot t/local));
}
elsif (glob('t/*.t')) {
  # Normal test suite.
  print expanded_template('t/');
}
else {
  # Assume test.pl in all other cases. If the distro has no tests at all, you
  # should add at least a test.pl via a patch.
  print expanded_template('test.pl');
}

sub expanded_template {
  my(@tests) = @_;
  my $cover_opts = $ENV{COVER}
    ? q[PERL5OPT='-MDevel::Cover=-db,$(BLEKKO_DIR)/cover,+ignore,\\bTest\\b,+ignore,\\bprove$
$,+ignore,\\bt$$'] : '';
  return <<EOF;
        $cover_opts PERL_DL_NONLAZY=1 prove -v --time \\
                -I\$(INST_LIB) -I\$(INST_ARCHLIB) \\
                -I\$(BLEKKO_DIR)/perl --formatter TAP::Formatter::JUnit \\
                @tests > TEST-\$(DISTVNAME).xml
EOF
}
```


- This script adds a test_hudson target at the end of the Makefile (AFTER perl Makefile.PL has run).

- lets us add a specific target to run the test suite and get JUnit out

- Only works for Makefile.PL-based modules; Module::Build modules can be forced to create a makefile via Module::Build::Compat - haven't needed this yet

- If you're doing this yourself, it's simpler to just add it

Demo

Wednesday, June 23, 2010

[INSERT LENGTHY HUDSON DEMO HERE :)]

Particular things to note include the ability to take a build product (say a tar.gz file) and map it back to the build where it was created - invaluable in making sure the version you have is the version you wanted

Optimizing for speed

Target getting any single start-to-finish chunk of the build under 5 minutes
 - quick enough to make feedback useful, but enough time to do some reasonable building and testing

Make your individual targets as small as possible; better to have many fast, small dependent builds than one gigantic slow one with no dependencies

use Hudson dependent builds to supply build products to one another

make -j

http://developers.sun.com/solaris/articles/parallel_make.html

Parallel make - huge timesaver (e.g., 8-way multicore machine done with -j9 can drop a 2-hour build to like 20 minutes)

If you are using a makefile, try to support this if you can

May be necessary to have some subtargets re-run make with -j1 if they can't handle it (e.g. ImageMagick's Makefile doesn't work except with -j1)

As a rule of thumb, you'll get around the best performance with -j n+1, where n is the number of processors you have

Can greatly speed up a build, but getting it worked out is hard; trying to convert is even harder

Biggest key is remembering that

```
foo: bar baz quux
```

means all three are candidates for simultaneous building; if there are interdependencies, state them and make will figure it out

```
foo: bar baz quux
baz: quux
bar: baz quux
```

URL on this page is THE best resource on doing this

Don’t build it at all

- If it’s a static requirement that doesn’t change as quickly as the codebase, consider just upgrading the system copy on the machines you’re building and running on

- use Hudson dependent builds - if the dependent piece is done, you won’t need a rebuild

- builds can trigger other builds on success

- (with a plugin) wait for a set of builds to complete before starting

- more and shorter builds = earlier feedback

- try to make your makefiles a series of small targets; for developer convenience you can create an “everything” target that runs all the others

- don’t forget to explicitly state the dependencies if you want the devs to be able to -j too!

Build visibility

None of this is useful if it’s not being paid attention to

Have to get it in front of the developers, and as soon as possible - but without being so noisy they start ignoring it

Build messaging

Wednesday, June 23, 2010

IM plugin allows us to announce to IRC, IM, Jabber, Twitter, ...

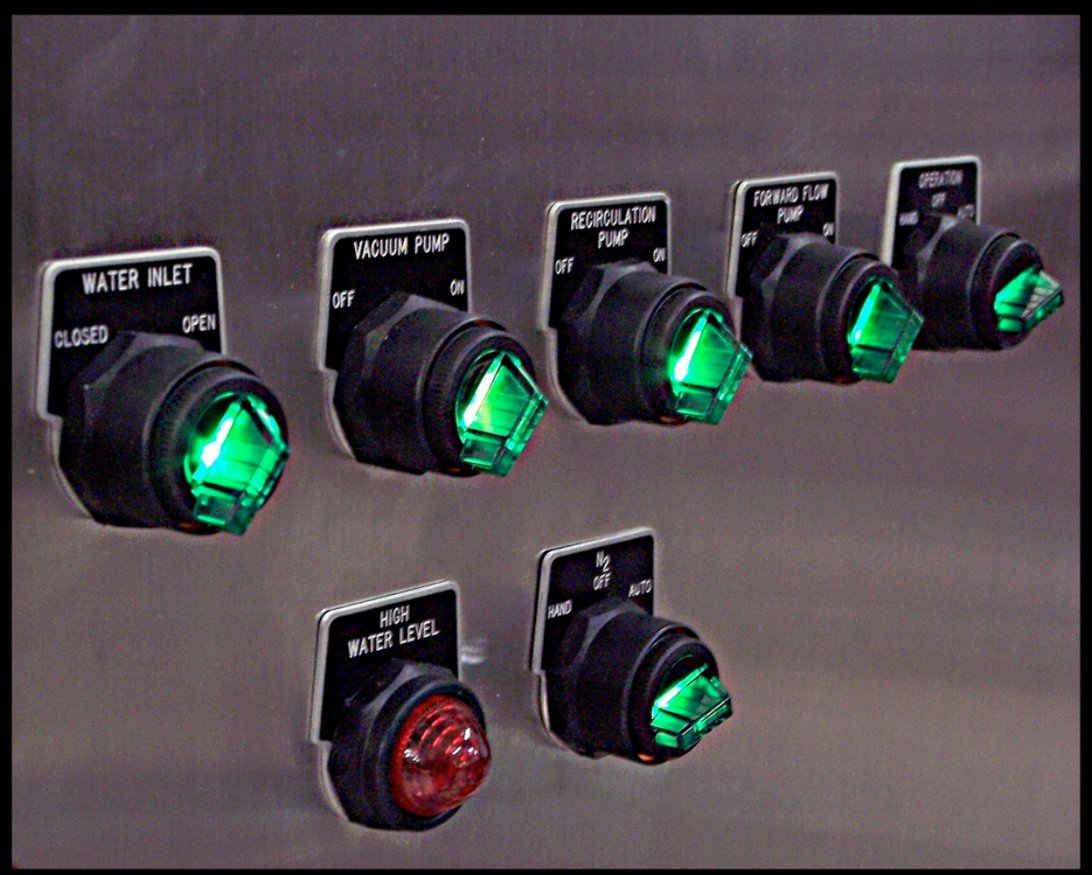
Figure out where your developers hang out most, and have Hudson post notifications there

This is best suited to the incremental/smoke builds, as it posts the changelog as well

BUILD FAILED plus one or two lines generally gets a positive response

20 lines of changelog generally gets “can we turn that off”?

You want to avoid “oh, I didn’t look at X for a few hours ...”



Ambient feedback

Wednesday, June 23, 2010

Several different plugins and features for this

Meant to go up on a monitor in everyone's eyeshot

Big, bold indications of success or failure, status of all builds

I like the Extreme Feedback plugin best - also shows how far along a build is, and who broke it if it's broken

Nobody wants their name up on the big screen with the big red bar!

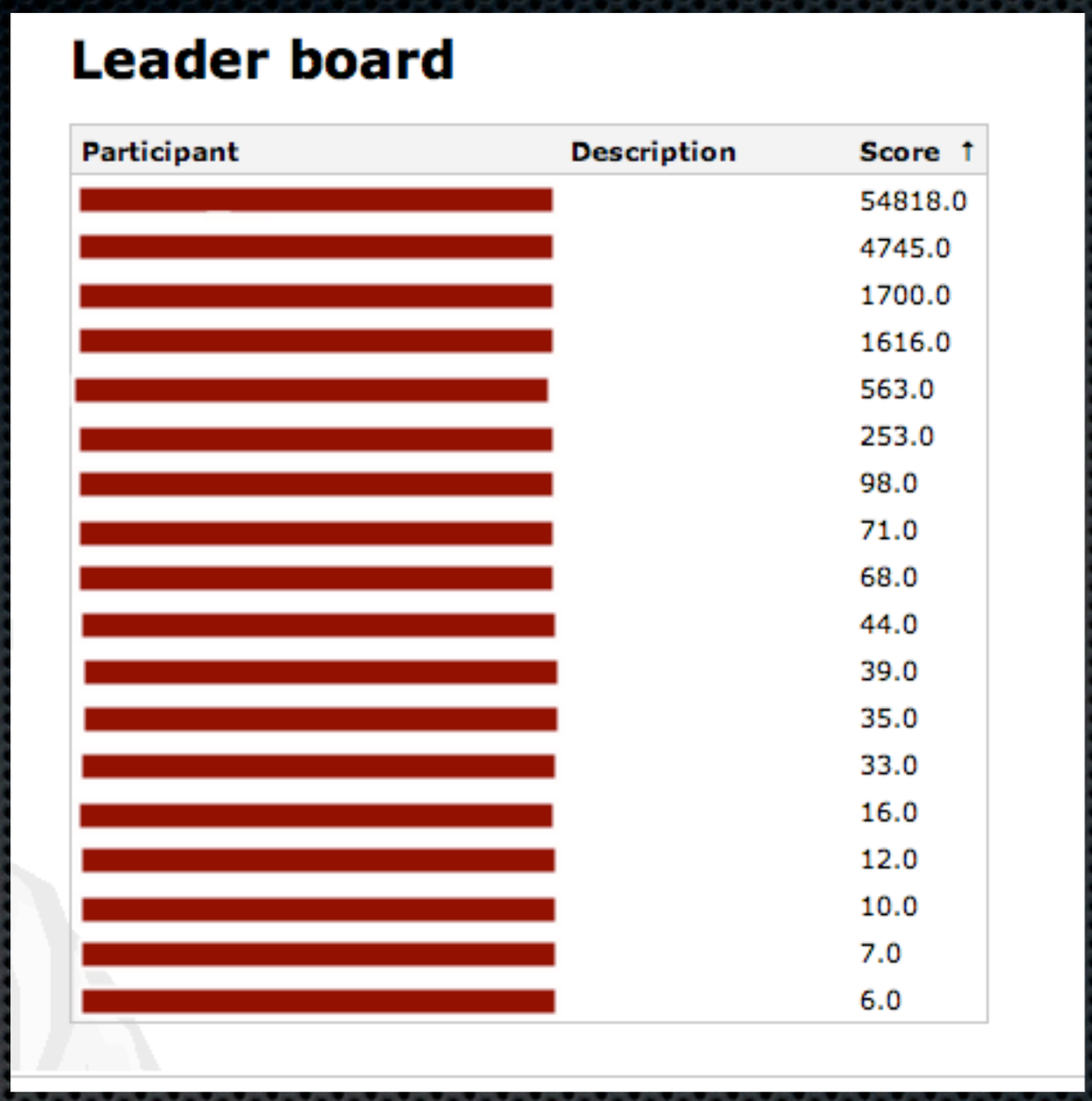
Good old email

Of course we can send mail

Best suited for “big” builds that will have large changelogs

More ignorable than most notification methods; last resort if you get too much pushback on build messaging

And you can always make sure it mails the project managers on all the broken builds if you have to ...



The CI game

Wednesday, June 23, 2010

Depending on your developers, this may be the best way to get them involved

Score points for adding successful tests, fixing a broken build

Lose points for breaking the build

No score for not fixing a broken build

We use an iterative build as noted above, so successful changes in modules that others depend on score more points (sometimes MANY more points) – this is a weak point BUT makes it useless as a programmer performance metric – way too easy to game.

Can stay “fun” – and maybe puts more attention on the modules that are most core to the project – so win/win?

Devel::Cover

Wednesday, June 23, 2010

You will need to add something to run with Devel::Cover, whether a target or a script fragment

Hudson's HTML publisher can be used to put a link to the last build's coverage on the job page

This doesn't get archived the way build products do unless you mark it as a build product as well

```
ifeq ($(COVER),)
	cd $(dir $@); \
	PERL_DL_NONLAZY=1 prove -r -I$(BLEKKO_DIR)/perl -v --time $(if $
(HUDSON),--formatter TAP::Formatter::JUnit > TEST-$(subst /,,$(dir $@)).xml,);
else
	cover --delete $(subst /,,$(dir $@))_cover_db
	PERL5OPT='-MDevel::Cover=-db,$(subst /,,$(dir $@))_cover_db,+ignore,\\bp
rove$$,+ignore,\\.t$$,+ignore,\\bAlgorithm\\b,+ignore,\\bSync\\b,+ignore,\\bFile
sys\\b,+ignore,\\bTest\\b,+ignore,\\bTemplate\\b,+ignore,\\bClass\\b,+ignore,\\b
Crypt\\b,+ignore,Data/Dumper,+ignore,Date.*?/?.*,+ignore,\\bException\\b,+ignore
,\\bHTTP\\b,+ignore,\\bIPC\\b,+ignore,\\bImage\\b,+ignore,\\bList\\b,+ignore,\\b
Math\\b,+ignore,\\bInterface\\b,+ignore,Params/Validate,+ignore,\\bRegexp\\b,+ig
nore,\\bSVG\\b,+ignore,Set/Infinite,+ignore,Uplevel,+ignore,Mmap,+ignore,Syscall
,+ignore,Text/.*,+ignore,Time/Zone,+ignore,Tree,+ignore,common,+ignore,parent,+i
gnore,\\bGeo\\b,+ignore,\\bJSON\\b,+ignore,\\bNet/DNS\\b' PERL_DL_NONLAZY=1 prov
e -r -I$(BLEKKO_DIR)/perl -v --time $(if $(HUDSON),--formatter TAP::Formatter::J
Unit $(dir $@)/t > TEST-$(subst /,,$(dir $@)).xml,$(dir $@)/t);
endif
```


Get aggressive with the exclusions to get a better estimate of coverage

CPAN and core modules can distort the coverage because they aren't being called to get test coverage; they're just utilities.

Merging coverage

```
COVERED_MODULES = My-Module Another-Module Still-Another
cover: *_cover_db
        cover --delete
        cover -write cover_db \
              $(patsubst %, %_cover_db,$(COVERED_MODULES))
        mkdir -p cover_db/structure
        for f in $(patsubst %, %_cover_db,$(COVERED_MODULES)); do \
              cp $$f/structure/* cover_db/structure; \
        done
        cover
```


If you have multiple modules, you'll have several cover_db files

This will merge them

Individual coverages targets have custom names

the structure/ dir has the critical data to measure coverage

We just create a directory for each thing we covered individually, and cover will combine them for us

The downside

Hudson isn't perfect

Everything runs under the JVM's control, so you have to make sure the JVM has enough resources to handle it.

Rough rule of thumb is that you'll need a GB of memory per 100K tests when it does its summary.

Documentation is sometimes minimal.

- VirtualBox cloud plugin supposedly lets you set up to automatically launch VMs as slaves

- The form says you need a name, a URL, and an user ID and password, but doesn't say what the last three are (ID on the box running VirtualBox? On the VM? What port are we using? what URL scheme? If you know, you know, but if you don't you don't)

Things are quirky

- $WORKSPACE IS an absolute path to the job's workspace on a master node

- on a slave node, it's relative to the directory the slave was launched from (at least if you use the 'javaws' method of starting the slave)

- this is not documented anywhere

What we're doing now

Iterative build (make check) on each checkin; runs in 2 minutes or so, with IRC alerts

2x daily, run a clean build from scratch with extra integration testing (2+ hours)

Perlcritic run over every script and module with customized ruleset

Plans

Add coverage reporting; currently too slow, since our CPANish modules and real CPAN modules get built and covered (need to pull the “constant” CPAN modules out into RPMs)

Selenium tests currently being developed; have made some fixes to Test:WWW::Selenium (and now have commit bit). Developing Selenium Grid tests.

Considering pushing to dev when build succeeds.

What to remember

- Hudson is easy-install, customizable cron on steroids
- Need: safe source, dependable process, good tests
- Get: trackable products, history, analysis

Questions?

Thank you!

Hudson for Everybody Else

Joe McMahon
mcmahon@blekko.com
Twitter: @joemcmahon

/Library/LaunchAgents/hudson.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//
EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
 <key>UserName</key>
 <string>mcmahon</string>
 <key>Label</key>
 <string>Hudson</string>
  <key>EnvironmentVariables</key>
    <dict>
      <key>HUDSON_HOME</key>
      <string>/Users/mcmahon/.hudson</string>
    </dict>
  <key>ProgramArguments</key>
  <array>
  <string>/usr/bin/java</string>
  <string>-jar</string>
  <string>/Users/mcmahon/Hudson/hudson.war</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
 </dict>
 </plist>
```

```
sudo launchctl load -w /Library/LaunchAgents/hudson.plist
```


Bonus: this creates a LaunchAgent that starts Hudson as soon as you log in on OS X

Change the username to yours so you can fiddle with the workspaces if necessary (can use a special ‘hudson’ user – do not leave out, as this means ‘run as root’!)

launchctl unload will stop Hudson – may need to do it more than once if it complains it couldn’t stop Hudson

Linux install seems to enable a restart option in Hudson itself – need to look into that some more to see if it can be made to work on OS X as well