This presentation is based upon a 3 day course I took from Jared Richardson. The examples and most of the tools presented are Java-centric, but there are equivalent tools for other languages or you can use the same tools to do non-Java automated testing as well.

1

Outline

Contents

2 Introduction

Why test automation?

- programmers are lazy
- Automated system finding bugs is better than people
- Need a reproducable test system

Being lazy isn't a bad thing. Automate things that you can. It keeps people from making mistakes. Making sure you have a reproduceable test system allows you to automate tests.

Test automation setup

- Make sure you reproduce the running system
- May need virtual machines
- Do whatever is necessary to get your test data

Make sure you're running as close to the real system as possible. This sometimes means building up a virtual machine that has all of the same tools as the production system. Get real test data. This may mean you suck down the database off the production server to get a reasonable test set. Although you may want to sanitize it before you use it too much.

Ant

- Least common denominator
- Make this the gold standard
- Don't make developers always wait too long for tests, otherwise they'll skip
- use multiple targets
  - one for quick unit tests
  - one for longer integration tests
  - short test for current task

The really nice thing about ant is that all you need is Java to run it. If you're working on a Java project this is great, if not, think about what tools make sense here. The key here is that it's not tied to the IDE. You can build everything and run your tests outside of the IDE so the devlopers can pick the IDE that works best for them.

3 Continuous Integration

Continuous Integration (CI)

- Hudson or CruiseControl
- Once you can run automated tests - run them in CI!
- Will send emails on status of builds

Continuous Integration is the idea that on each checkin to source control you kick off a build and some set of tests and then the results of that are reported back to the developers. The tests can be as long or as short as you'd like, however it's best to have a short set of tests so that the devlopers find out quickly if they've broken something.

Any tests that you can automate, you can run from CI. Hudson and CruiseControl are the two popular tools these days. Both are written in Java and have nice hooks for Java, but are definitely not limited to Java. I have successfully used Hudson on C and C++ projects. Personally I like the extra plugins in Hudson and the ease of setup.

Metrics & CI

- Don't display too much
- Find out what numbers are important
- Play to the developers egos
- Don't point at problems bugs, point at buggy code

If you display too much information, people will ignore it. Play to the devleopers egos. Start by sending the emails to just the people that broke the build. If that isn't working, try sending the messages to all developers on the team. At one point Jared ended up sending the emails to the whole company - in a small company with CEO approval. Since it's an automated system, it's not a personal attack. It's just a system making an objective statement about the code.

Some Metrics

- McCabe Cylomatic
- http://en.wikipedia.org/wiki/Cyclomatic_complexity
- Code coverage

Pick some metrics that are useful. McCabe Cylomatic is a common one. Scores between 0 and 28 are ok. Above that the number of bugs increases linearly with the score. You can find out more details about it on Wikipedia.

Code coverage is another good metric, although remember that it should be used as a guide, not a goal. 100% test coverage as a goal cuases people to write useless tests that test things that don't need testing.

static code analysis

- run against development code
- run against test code too
- Start with FindBugs
- Move to PMD once FindBugs is clean enough
- Open Tasks
- Copy Paste Detector

Static code analysis can be useful to find standard bugs and it doesn't require you to write more test code to find them. Remember lazy is good.

Run your static code analysis against the test code as well. I haven't done this in the past, but Jared pointed out that bugs in tests result in bugs in production.

FindBugs and PMD are two good tools for Java. You can even run them as a Java WebStart app, so you don't need to install it or hook it into your build system just to test. FindBugs will find lots of good things. Once you've cleaned up the ones that you care about (and it may not be all), then try PMD. Don't start with PMD otherwise you'll be overwhelmed with everything that it finds.

A metric that that is sometimes used is the number of TODO and FIXME notes in a codebase. This can be useful to see if things are getting finished, in addition to the bug tracking information.

Example Metric - Risk

- risk = (McCabe's * call count) * coverage percentage
- Example risk = (70 * 74) * 50
- List top 10 classes by risk
- Watch and see the results

An example metric that Jared ended up using was risk. It's computed based on McCabe's cylomatic number and code coverage and the number of times that a method is called. As call count decreases or coverage increases, risk goes down. List the classes with the top 10 risk values on the CI server page and see what happens. People will start finding things that need to be fixed in that code to get "their" class of the risk list.

The example is a McCabe value of 70, 74 calls to the method and 50% code coverage of the method.

4 Writing Tests

Writing Tests

- Don't open the Kimono
- Only test the public API

Don't open the Kimono. You don't test the private parts of the code. Stick to testing the public API and it's behaviors. This way as the developers decide to change the implementations, as long as the behavior is still the same, the tests pass.

Characteristics of a good test

Right-BICEP

- Are the Results Right
- Boundary Conditions
- Check Inverse Relationships

- Cross-check using Other Means (Test the Oracle)
- Force Error Conditions (Attacks)
- Performance Characteristics

Did you test it right, did you test the boundary conditions? Did you test relationships going both ways. Make sure your computing the answer in a different way, otherwise you're likely to make the same mistakes twice. For instance testing that adding 1 to max int results in 1+max int is bad. What you should test for is that the result is greater than max int.

Test error conditions and how the system behaves, this can be big for security. Test the performance constraints of the system.

Characteristics of a good test (cont.)

- abstract away the test tools
- keep test methods short
- longer than a page and it needs to be shortened

Abstract away the test frame work, as it allows one to change it out later. Keep test methods short, just like production code. You're code shouldn't be longer than a page as then you'll forget what was above and scrolled off the screen.

Mocks & stubs

- Sometimes the terms are interchangble
- stubs are usually written by people
- mocks are automatically created, just count method calls
- If you use Continuous Integration you shouldn't need mocks

Some people use mocks to isolate code that is being tested so that you only break 1 test and you can easily find what is wrong. However, if you're checking in regularly then CI will tell you where the error is. Use stubs where you can't test reliably, such as testing a thermometer.

Integration vs. Unit tests

- Unit Tests
  - good for starting from new code
- Integration Tests
  - good for legacy code
  - best use of time (for legacy code)
  - will end up with low code coverage percentage
  - but it's the right percentage

It depends on what kind of code you're writing. You can write unit tests for legacy code, but it's usually slow and doesn't have much payoff. Now adding new features to legacy code can be unit tested and should be, if possible.

Test Driven Design (TDD)

- Write one test then write code to make the test pass
- Causes you to really think
- Use for new code
- will end up with more stable code
- will end up with high code coverage (not goal though)
- Pair programming

This coding methodology causes you to really think about the requirements for the code that you're writing. Doing pair programming here can help to brainstorm ideas. Extra eyeballs on the code are always good.

Defect Driven Testing (DDT)

- Find a bug
- Add a test
- Jazz it up
  - add tests with variations
  - never write 1 test for a bug

This testing is great for legacy code, code that doesn't have tests written for it to start with. For each bug, write a test case that exposes the bug. Then write a couple of variations on that test case. Don't just write one test, write multiple tests to help catch bugs that haven't been discovered in the same area.

Testable code

- Good testable code does 1 thing and then returns

Methods that do lots of things are really hard to test. You have multiple behaviors going on inside the method and it's hard to write tests for complicated things like that. Keep it simple.

Testing Multi-tier architecture

- Mock everything up to start (tracer bullets)
- Test everything with canned data
- Allows you to find out if the architecture works early

Tracer bullets are used to see what you're shooting at. When building a multi-tier architecture, mock up the architecture. Put some canned data through it and make sure the architecture makes sense. Then start implementing pieces of the system and continue to run the same tests with the canned data and start adding more real data as the system matures.

5 Tools

Picking Tools

- Have 1 person pick the tools and go with their choice
  - otherwise end up with too many options
  - Have them write the test templates
  - Make everyone use it

When picking tools, have one person pick the tools and make everyone use them. Make the person that picks the tools write up test templates for everyone to use. That avoids writers block. You've always got a test to start with and to modify. You never hear of Editors Block.

Database Tools

- Liquibase
- ruby database migrations

Some tools for keeping track of database schema changes are Liquibase and ruby database migrations. If you're building this kind of system, look into them. Liquibase is XML based, ruby is code based. Depends on what you like to use.

Web Testing

- Selenium
  - Firefox plugin for IDE to create tests
  - Can call multiple browsers
- YSlow
  - static web page analysis tool
  - Gives performance tips

Selenium is a really useful tool for writing web tests. To avoid writers block, start by recording a test using the selenium IDE (firefox plugin). Then export the test in your language of choice and start editing. While the recording can only be done in firefox, the playback can be done in any browser. Safari, IE and firefox have full support with other browsers having limited support.

For testing performance of web pages, use YSlow. It's an addon for firefox that builds on Firebug (which is really helpful for web development as well). YSlow will analyze your web page and provide suggestions on how to speed up the page load time.

UI testing

- Test at the controller and model layer
- Use something like selenium or AWT Robot

When doing UI testing, try and test at the controller and model layer and you avoid the issues of having to figure out how to push buttons. However when you need to, there are tools like selenium and the AWT robot class. UISpec is one that I've looked into, but wasn't mentioned in class. There are other ones out there as well.

Resources

- Pragmatic Programmer
- Pragmatic Unit Testing
- Buildix - can download everything for CI in a vm

Some resources that can be useful. I'm told the "Pragmatic" books are great resources, especially Pragmatic Programmer and Pragmatic Unit Testing.

For getting started with CI you can goto Buildix.thoughtworks.com and get a VM that contains everything you need for CI as well as source control and bug tracking.