Testing, part 1

UNC COMP 523

October 19, 2020

Jeff Terrell

Announcements

- sign up for OpenClass.ai before Q5 this Wednesday (will include content from Wednesday)
- remember that mentor meetings need demos now
- grades are up to date as of 12pm today
- tech talk schedule and topics are finalized and posted; 6 teams go next week
- what counts as active learning in your tech talk?

After this semester

- client needs some bug fixed or feature added
- you graduated and moved to `<location>` to work for `<awesome company>`
- client's niece has a friend who knows how to code
- how will the friend learn the codebase and feel confident about their change?

After this semester (2)

- you graduate and moved to `<location>` to work for `<awesome company>`
- your first assignment is to add `<feature>` to the flagship app of `<awesome company>`
- how will you learn the codebase and feel confident that you aren't breaking something?

The point: an app with *automated tests* covering all of the important features is easier to contribute to.

After this semester (3)

- you graduate and moved to `<location>` to work for `<awesome company>`
- your first assignment is to add `<feature>` to the flagship app of `<awesome company>`
- you get a dev environment setup to see the effect of your changes
- you add some code for your feature
- *you can run the comprehensive test suite to be sure you haven't broken anything else*
- in fact, your code can't be deployed unless all the tests pass
- (and, ideally, your commit(s) won't be merged unless they're covered with tests you wrote, to pay it forward)

Before automated tests [were mainstream]

- without automated tests, we might break things more often
- some shops have a manual testing script
- manual testing is not fun, error prone, and tends to make releases infrequent (remember system agility?)
- I had a freelance gig once...
- some shops have a QA (quality assurance) team; see above

Aside: alternatives to testing

- static type systems can catch errors before deployment
- some type systems are very sophisticated (e.g. Haskell's) and catch many things
- (for more on this, I'll be teaching 524 programming languages next semester...)
- even there, many complex projects still have tests
- neither types nor tests give waterproof guarantees of working software

Outline

- ~~Introduction~~
- Why test?
- Why not test?
- Types of tests
- Test coverage
- Test-driven development (TDD)

Outline

- ~~Introduction~~
- **Why test?**
- Why not test?
- Types of tests
- Test coverage
- Test-driven development (TDD)

Why test? (1)

1. Tests give confidence

- see earlier stories: without tests, it's hard to know if you broke something
- also, with tests, you have more confidence that your own feature works right
- recall XP value of courage; tests enable courage
- confidence is huge! **This is the most important reason to write tests!**

Why test? (2)

2. You get the perspective of the caller

- this helps keep things taut and minimal
- remember YAGNI - You Ain’t Gonna Need It

Why test? (3)

3. Tests document developer intentions

- e.g. “I need to be sure this function works on these kinds of inputs”
- or, “How do I intend this function to be called/this object to be used?”

Why test? (4)

4. Tests reveal regressions

- did you break an existing feature?
- if it was tested, you'll know right away
- a test suite is a gift to future developers on the project, including yourself

Why test? (5)

5. Tests close the time gap

- expert programmers use tight feedback loops to iterate quickly
- this enables a laser focus
- with a tight feedback loop, why not trigger it often?
- then if something breaks, it was in code you just wrote
- if it works, then forget about it and focus on the next tiny thing
- key idea: **rapid tests enable rapid iteration**
- and slow tests help find bugs before users do

Why test? (6)

6. Tests enable refactoring

- remember XP practice of refactoring? “red, green, refactor”
- once you have passing tests (green), you can clean up
- decouples getting it *working* from getting it *simple*
- again, rapid tests give a tight feedback loop when refactoring
- in other words, decouple getting it /working/ from getting it /right/
- advice from Don Smith re: writing: braindump first, clean up later (or Hemingway: write drunk, edit sober)

Why test? (7)

7. Testing can be fun

- taking small steps and making steady progress feels good!
- helps you get into a mental state of *creative flow*, which is fun!
- worst experience in programming IMO: feeling stuck
- with good tests as a feedback loop, it's harder to get stuck

Why test? (summary)

1. **Tests give confidence**
2. You get the perspective of the caller
3. Tests document developer intentions
4. Tests reveal regressions
5. Tests close the time gap
6. Tests enable refactoring
7. Testing can be fun

Outline

- ~~Introduction~~
- ~~Why test?~~
- **Why not test?**
- Types of tests
- Test coverage
- Test-driven development (TDD)

Why not test? (1)

1. Tests add complexity to the project

- now you have to write not just code, but also tests
- sometimes you even have more test code than implementation code
- but, complexity is in a "sidecar" though, in separate directory
- also, getting perspective of caller and doing refactoring can pay for testing complexity

Why not test? (2)

2. Tests don’t catch every bug

- but it does catch many of them
- and the feedback loop and creative flow tests provide are useful

Why not test? (3)

3. Tests add inertia to the interface

- if you need to change the interface, you also have to change the tests
- this isn't technically "refactoring" at this point, it's a heavier change
- avoid changing the interface if possible
- but if it sucks, redo it! (remember XP value of simplicity)

Should I test?

My standard recommendation: if you're working on throwaway code or a short-lived experiment or exploration, don't test. Otherwise, test. (And COMP 523 projects should be considered long-lived!)

Outline

- ~~Introduction~~
- ~~Why test?~~
- ~~Why not test?~~
- **Types of tests**
- Test coverage
- Test-driven development (TDD)

Types of tests

- unit: test one component in isolation
- integration: test several components working together
- end-to-end: a scripted robot that operates as a user

Unit tests

- goal: test one component in isolation
- "component": typically a function or a class
- can use "test doubles" to achieve isolation that doesn't otherwise exist (more about this later)
- a pure function is dead-simple to test: did these arguments result in this returned value?
- example: `add(8, 7) => 15`
- OOP example: `new CsvData("col1,col2\nval11,val12\n").columns() => ['col1', 'col2']`

Integration tests

- goal: test several components working together
- "several" could be 2 or 500; there's a wide range here
- example: `new CsvDataFile('data.csv').columns() => ['col1', 'col2']`
- example: trying to login to the backend with invalid credentials results in a 403 Forbidden response

End-to-end (E2E) tests

- goal: test *the entire system as a user*
- typically involves fancy tools to navigate a user interface
- e.g. Cypress (for Javascript) lets you drive a browser through your app

Tradeoffs

- *implementation difficulty*: unit tests are easiest
- *execution speed*: unit tests are fastest (remember fast feedback loops)
- *confidence delivered*: e2e tests are very valuable!
- suggestion: **make an informed choice that makes sense for your project**
- suggestion: tackle variation where it lives
- rare to find a project that only has one type of test (though many lack e2e tests)
- most components aren't isolated, and test doubles might not be worth it
- understand these tradeoffs to navigate the religious wars about testing

Outline

- ~~Introduction~~
- ~~Why test?~~
- ~~Why not test?~~
- ~~Types of tests~~
- **Test coverage**
- Test-driven development (TDD)

Test coverage

- there's no perfect metric to measure health of a test suite
- but test coverage is a useful one
- measures percentage of codebase exercised by test suite
- 100% means every line of code is covered by a test
- 100% is usually not a goal: some code is boring and/or tests aren't worth the trouble
- (a measure of your engineering maturity: evaluating risk/reward in light of *context*)
- recall from syllabus that "an app getting a grade of A...has good test coverage (probably over 70%, although that can depend on the project)"

Test coverage challenges

- e.g. Team I is using Tableau for their app
- perhaps some code could be extracted into functions/classes, and unit/integration tests written
- also, they could use Cypress to drive a browser through the app
- e.g. Team C is creating a VR app
- see above about unit/integration tests for functions/classes
- is there a way to do e2e testing on VR apps? That might be too hard!
- e.g. Team N is modernizing a data-centric web site
- their program runs once to create the site; there is no long-lived app
- unit/integration tests still useful for the components
- also, e2e testing might be easier: the program's UI is the command line

Outline

- ~~Introduction~~
- ~~Why test?~~
- ~~Why not test?~~
- ~~Types of tests~~
- ~~Test coverage~~
- **Test-driven development (TDD)**

Test-driven development (TDD)

- we saw TDD as one of the XP practices
- this is still somewhat extreme
- alternative might be called TLD: test-last development
- I've done TDD at my last job, but it was hard to get to a place it felt comfortable
- my difficulty: fluency with testing framework vs. just using dev environment
- TDD and TLD **both provide current and future confidence**
- experienced devs will have good feedback loops with TLD like what TDD can give you

Topics for next time (maybe)

- Testing
  - Test doubles
  - Tips for test-friendly code
    - dependency injection
    - effects management and pure functions
- Tech debt
- Quiz 5!

TAQs

Go to pollev.com/jeffterrell