Testing, part 2

UNC COMP 523

October 21, 2020

Jeff Terrell

Announcements

- this is the last lecture from me this semester
- final quiz (#5) is today *on OpenClass*
- remember that you can get technical help in the App Lab (applab.unc.edu)
- be aware of upcoming assignment deadlines; I might not remind you henceforth
    - tech talk ☕☕☕☕
    - test coverage report ☕☕, due week of Nov. 1
    - developer README ☕☕, due week of Nov. 8
    - client-oriented overview ☕☕, due week of Nov. 15
    - APPLES reflection 2 ☕, due week of Nov. 15
    - personal report ☕, due (email to me) on Wed. Nov. 18

Outline

- tech debt
- testing demo (incl. test doubles and dependency injection)
- tips for test-friendly code
- quiz 5 on OpenClass

Outline

- **tech debt**
- testing demo (incl. test doubles and dependency injection)
- tips for test-friendly code
- quiz 5 on OpenClass

Tech debt

- I worked on a project recently...
- **key idea: like debt begets debt, complexity begets complexity**
- both cause a kind of "drag" on progress
- so, spend time refactoring, before your messy code gets repeated and compounded
- (remember that refactoring is a practice pursuing the XP value of *simplicity*)
- notable refactoring principle: DRY, a.k.a. Don't Repeat Yourself

Tech debt - positives

- remember how engineers like to say "it depends"?
- like credit cards, tech debt can be useful
- for example, if you have a deadline to ship a feature, might be worth it to leave it sloppy or skip writing tests
- if you do, though, don't consider the feature done yet
- might need to communicate to client or manager that you need to circle back and finish that before doing other work
- if you don't circle back, that sloppy or untested code can come back to bite you

Outline

- ~~tech debt~~
- **testing demo (incl. test doubles and dependency injection)**
- tips for test-friendly code
- quiz 5 on OpenClass

Testing demo

- simple example of tests in Javascript/Node.js
- illustrates various tradeoffs of different testing strategies
- includes test doubles and dependency injection
- based on commits in a public git repo
- you can follow along by cloning https://gitlab.com/jeff.terrell/testing-tutorial/ (then say `npm install`)
- (assuming you have node.js installed, that is)
- I will summarize take-aways at the end

Testing demo

https://gitlab.com/jeff.terrell/testing-tutorial/

Testing demo take-aways

- mocks are a kind of test double
- test doubles enable you to isolate a class/module/function from its dependencies, turning it from an integration test into a unit test
- integration tests can be noisy when there's a bug in a widely depended-upon unit
- but test doubles help point you directly to the broken code
- but test doubles can hide *integration bugs* (though less of a problem in some languages)
- unit tests can run faster (and create tighter feedback loops) because they short-circuit the depended-upon units
- dependency injection gives callers (incl. in the test suite) control over the dependency, at the cost of increased complexity in the caller

Outline

- ~~tech debt~~
- ~~testing demo (incl. test doubles and dependency injection)~~
- **tips for test-friendly code**
- quiz 5 on OpenClass

Tips for test-friendly code

- **most important source of complexity: effects**
- effects can include network requests, database writes, file system writes, standard I/O, etc.
- fingerprint of an effect: a statement that doesn't contribute to a return value
- tip 1: isolate your effects into single-purpose, specific functions
- example: need to save a post to database? have a `savePost` function that does only that
- then you can either mock that function when testing or use dependency injection
- then you don't have to worry about managing the effect when testing; you skip it (and save time)
- (ugly alternative: reset the database after every test)

Tips for test-friendly code (2)

- these effects change the shared state of the program (e.g. file system or database)
- then any function that reads from the shared state is non-deterministic
- tip 2: use dependency injection when you must read from shared state
- example: rather than passing a file path or file handle to a function, pass the file contents
- then the caller has to do the I/O, but the function can remain deterministic
- testing is easier here, too: the tests don't have to mess with the file system at all
- side benefit: learn/develop a trusted library of very simple effect-only functions (e.g. `slurp` in Clojure)
- caveat: if file is too big to read into memory, this won't work (but a lazy sequence of lines might)

Tips for test-friendly code (summary)

1. isolate effects into single-purpose, specific functions, like `savePost`
2. use dependency injection when you must read from shared state, like passing a file's contents instead of a file's path

Outline

- ~~tech debt~~
- ~~testing demo (incl. test doubles and dependency injection)~~
- ~~tips for test-friendly code~~
- **quiz 5 on OpenClass**

Quiz 5

see https://openclass.ai