Writing Profitable Tests in Go

by @kinbiko
on 2024-09-19

» Who am I?

* Writing Go for 7 years.
* Some of it was good.
* Love testing.

Outline and assumptions

What you will learn

- A new way of thinking about testing.
- A rule for deciding whether or not to write a test.
- 5 techniques on writing Go tests you might not know.

Not covered

- Go testing basics.
- What mocking is.

» Why do we write tests?

- Verify code works (with important limitations):
  - Only test cases you can think of.
    - Ignore fuzz-testing for now.
  - Can't test all real-world cases.
    - Diminishing returns on more test cases after a point.
- **Confidence that future code changes are OK.**
  - Change && test failure $\Rightarrow$ either expectations (tests) or new code is wrong.
- These ideas are abstract, so let's make them concrete through the idea of profitability.

» Definition

A test is **profitable** iff for the lifetime of the test,

[the "revenue" of the test] **is greater than** [the cost of the test].

» Example:

Lifetime ”revenue”:

- Avoided 15 minutes of debugging, 3 times
- Saved 1 hour of responding to an incident
- Avoided ¥ 100,000 loss in revenue due to the incident
- = 105min eng time + ¥ 100,000

Lifetime cost:

- 20 minutes to write
- 20 minutes to update by 5 engineers
- 1 minute of CI runtime to execute
- = 120min eng time + 12 sec runtime

Profit:

- If you assume an engineer is paid ¥ 5000/hr and runtime costs are negligible, that becomes:
- (100,000+8,750)-(10,000+ε)
- **= ¥98,750***

» *Engineering isn’t accounting …and I can only experience one timeline

- You can’t tell if an incident would happen ahead of time!
  - So multiply relevant revenue by your Change Failure Rate (CFR); the percentage of deployments that cause a production issue.
    - If your CFR is 10%, then the profit estimate is ¥3,750.
  - Profit estimate *could be negative*.
- For more about CFR, read *Accelerate* by Gene Kim, Jez Humble, and Nicole Forsgren.

» In a product, tests don't directly provide value. Address indirectly.

- Increase ratio of time spent on production code instead of test code.
- Increase quality to reduce impact/frequency of incidents.
- (Reduce the cloud bill by reducing resources required for CI.)

» Tip 1: Empathy is a skill you can grow.

* Assume every line of code is read 5 times.
* Make it easy to read, even if it's harder to write.
* Within reason.

» Write helpful assertion messages (bad case)

```
got, err := somepkg.SomeFunc()
assert.NoError(t, err)
assert.Equal(t, got, want)
```

» Write helpful assertion messages (good case)

```go
got, err := somepkg.SomeFunc()
if err != nil {
    t.Fatalf("unexpected error when calling SomeFunc %s", err.Error())
}
if got != want {
    t.Errorf("expected SomeFunc to return %+v but got %+v", want, got)
}
```

» Other uses of empathy in writing test code

* Name your test functions and variables well.
* Code comments, when test code is complex.
* Verify that tests fail when they should and read the output.
    * Remember to call `t.Helper()`
* Maintain same or greater quality of test code compared to production code.

» Tip 2: High-fidelity tests can be fast in Go, and super valuable.

* Starting the entire app in a test, with DBs in docker/in-memory service fakes etc. is not that hard.
    * In general you can't run these tests in parallel, but integration tests are very often just as fast as unit tests.
* Creating a test harness for reuse pays dividends:
    * The harness is a type that wraps common functions specific to your application, and provides test helpers for integration testing.

» Example test case: Complicated test case, easy to read

```go
func TestHarness(t *testing.T) {
    it := harness.NewIT(t)
    user := it.RegisterNewTestUser()
    it.Post("/api/endpoint_under_test", `{"some_data": "%s"}`, user.Name)
    it.Expect(401, `{"err": "unauthenticated"}`)

    it.Login(user)
    it.Post("/api/endpoint_under_test", `{"some_data": "%s"}`, user.Name)
    it.Expect(201, `{"status": "created"}`)
}
```

» Tips for implementing a test harness

* Best for integration tests, not unit tests.
* Write test doubles for observability tools, for testing async protocols like pubsub.
* Wrap AND embed `*testing.T` to avoid changing the harness itself too often.
* The constructor should do the hard stuff. Make it fast and lazy.
* Prioritise helpful error messages if anything goes wrong in the harness itself.
    * "Did you remember to run `docker compose up`?"

» Tip 3: Be careful to not overuse mocks

- **Mocks** (e.g. gomock)
  - Typically generated
  - Include expectations
  - Compile-time dependency, maybe a separate CLI tool too.
- **Stubs**
  - Lightweight, handwritten. Very simple
  - Return a specific value.
- **Fakes** (e.g. in-memory DB/filesystem)
  - Complex – don't create yourself unless absolutely necessary.
  - Great fidelity. Use these if you can.

» Tip 4: Make your table driven tests look like a *table*.

- Tables have columns and rows.
- Table rows contain **similar** sets of data
- Columns should be scannable.
- Code smell: big changes to the table/test body to add a new test case
    - It's probably a different **sub-test** in this case
    - Common unnecessary complexity:
        - Unnecessary nesting: keep the table "flat"
        - Error cases and happy path cases in the same table
        - Mock constructors or other complex functions in each row
        - Repetition (e.g. `ctx: context.Background(),`)

» Before vs After

Before:

```
tests := []struct {
    name    string
    input   string
    ctx     context.Context
    want    *string
    wantErr bool
}{
    {
        name: "success: SS",
        input: "SS",
        ctx: context.Background(),
        want: pstr("80"),
    },
    {
        name: "success: S",
        ctx: context.Background(),
        input: "S",
        want: pstr("120"),
    },
    // 10 more similar cases...
    {
        name: "success: G",
        ctx: context.Background(),
        input: "G",
        want: pstr("450"),
    },
```

After:

```
t.Run("error case", func(t *testing.T) {
    if got, err := MyFunc("H"); err == nil {
        t.Fatal("expected error, got %s", *got)
    }
})

t.Run("default case", func(t *testing.T) {
    got, err := MyFunc("")
    if err != nil {
        t.Fatal("unexpected error %s", err.Error())
    }
    if got != nil {
        t.Errorf("expected nil value but got %s", got)
    }
})

tests := map[string]*string{
    "SS": pstr("80"),
    "S":  pstr("120"),
    // 10 more similar cases...
    "G": pstr("450"),
}
```

» Tip 5: Golden files are a high-fidelity and low-maintenance testing strategy.

* Idea:
  * Store "expected" data as a file, e.g. pretty-printed json, in your repository.
    * Tip: use `testdata/` – a special directory ignored by go.
  * The test writes/asserts e.g. API responses against golden files.
  * If there's a diff the test fails, and updated files are reviewable.
* Don't actually need Go code for assertions – bash scripts work for static APIs.
* For more dynamic JSON APIs you can use a tool like `jsonassert`.

» Practice writing tests that optimize for profitability.

- Remember:
  - Use CFR and engineering time to estimate profitability of tests.
  - Maximise ratio of time spent maintaining production code vs test code.
  - Ideas to achieve a good ratio: Empathy / test harness / mock alternatives / proper test tables / golden files.
- Thank you for your time.