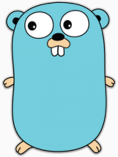
Programming in Go

Matt Holiday
Christmas 2020



Testing in Go

Go test features

Go has standard tools and conventions for testing

Test files end with `_test.go` and have `TestXXX` functions

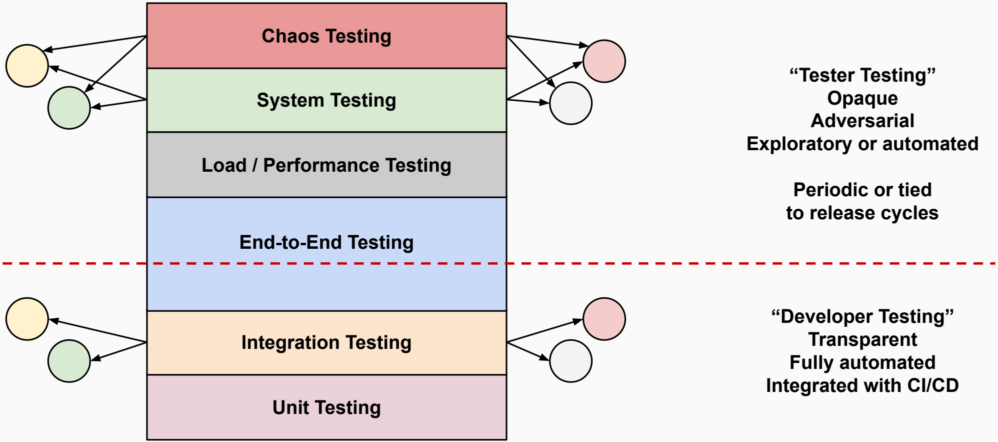
(they can be in the package directory, or in a separate directory)

You run tests with `go test`

```
go test ./...
ok   xyz/test        56.841s
ok   xyz/pkg/acedb (cached)
```

Tests aren’t run if the source wasn’t changed since the last test

Layers of testing



“Tester Testing”
Opaque
Adversarial
Exploratory or automated

Periodic or tied
to release cycles

“Developer Testing”
Transparent
Fully automated
Integrated with CI/CD

Goals

Things to test for

- extreme values
- input validation
- race conditions
- error conditions
- boundary conditions
- pre- and post-conditions
- randomized data (fuzzing)
- configuration & deployment
- interfaces to other software

Test functions

Test functions have the same signature using `testing.T`

```
func TestCrypto(t *testing.T) {
    uuid := "650b5cc5-5c0b-4c00-ad97-36b08553c91d"
    key1 := "75abbabc1f9f8d28d55200b43fd95962"
    key2 := "75abbabc1f9f8d28d66200b43fd95962"

    ct, err := secrets.MakeAppKey(key1, uuid)

    if err != nil {
        t.Errorf("make failed: %s", err)
    }
    . . .
}
```

Errors are reported through parameter `t` and fail the test

Table-driven tests

```
func TestValueFreeFloat(t *testing.T) {
    table := []struct {
        v float64
        s string
    }{
        {1, "1"},
        {1.1, "1.1"},
    }

    for _, tt := range table {
        v := Value{T: floater, V: tt.v, m: &Machine{}}

        if s := v.String(); s != tt.s {
            t.Errorf("%v: wanted %s, got %s", tt.v, tt.s, s)
        }
    }
}
```

Table-driven subtests

We can run *subtests* under the parent using `t.Run()`

```
func TestGraphqlResolver(t *testing.T) {
    table := []subTest{
        name string
        . . .
    }{
        name: "retrieve_offer",
        . . .
    }

    for _, st := range table {
        t.Run(st.name, func(t *testing.T) {     // closure
            . . .
        })
    }
}
```

A complex unit test example

```go
func TestScanner(t *testing.T) {
    scanTests := []struct{
        name  string
        input string
        want  []token.Token
    }{
        {
            name:  "simple-add",
            input: "2 1 +",
            want: []token.Token{
                {Type: token.Number, Line: 1, Text: "2"},
                {Type: token.Number, Line: 1, Text: "1"},
                {Type: token.Operator, Line: 1, Text: "+"},
            },
        },
        . . .
    }
```

A complex unit test example

```
    for _, st := range scanTests {
        t.Run(st.name, func(t* testing.T) {
            b := bytes.NewBufferString(st.input)
            s := NewScanner(ScanConfig{}, st.name, b)

            var got []token.Token

            for tok := s.Next(); tok.Type != token.EOF; tok = s.Next() {
                got = append(got, tok)
            }

            if !reflect.DeepEqual(st.want, got) {
                t.Errorf("line %q, wanted %v, got %v", st.input, st.want, got)
            }
        })
    }
}
```

A complex unit test refactored

```
type scanTest struct {
    name  string
    input string
    want  []token.Token
}
```

A complex unit test refactored

```go
func (st scanTest) run(t *testing.T) {
    b := bytes.NewBufferString(st.input)
    c := ScanConfig{}
    s := NewScanner(c, st.name, b)

    var got []token.Token

    for tok := s.Next(); tok.Type != token.EOF; tok = s.Next() {
        got = append(got, tok)
    }

    if !reflect.DeepEqual(st.want, got) {
        t.Errorf("line %q, wanted %v, got %v", st.input, st.want, got)
    }
}
```

A complex unit test refactored

```
var scanTests = []scanTest{
    {
        name:  "simple-add-comma",
        input: "2 1 +, 3+",
        want: []token.Token{
            {Type: token.Number, Line: 1, Text: "2"},
            {Type: token.Number, Line: 1, Text: "1"},
            {Type: token.Operator, Line: 1, Text: "+"},
            {Type: token.Comma, Line: 1},
            {Type: token.Number, Line: 2, Text: "3"},
            {Type: token.Operator, Line: 2, Text: "+"},
        },
    },
    . . .
}
```

A complex unit test refactored

```
func TestScanner(t *testing.T) {
    for _, st := range scanTests {
        t.Run(st.name, st.run)       // method value = func(*testing.T)
    }
}
```

More refactoring

```
type checker interface {
    check(*testing.T, string, string) bool
}

type subTest struct {
    name       string
    shouldFail bool
    checker    checker  // parameterize how we check results
    . . .
}

// we can now define different checker types
type checkGolden() struct { . . . }

func (c checkGolden) check(t *testing.T, got, want string) bool {
    . . .
}
```

Mocking or faking

```
type DB interface {
    GetThing(string) (thing, error)
    . . .
}

type mockDB struct {
    shouldFail bool
}

var errShouldFail = errors.New("db should fail")

func (m mockDB) GetThing(key string) (thing, error) {
    if m.shouldFail {
        return thing{}, fmt.Errorf("%s: %w", key, errShouldFail)
    }
    . . .
}
```

Main test functions

You can define a root function for all testing; it will then run all tests from this point

```go
func TestMain(m *testing.M)
    stop, err := startEmulator()

    if err != nil {
        log.Println("*** FAILED TO START EMULATOR ***")
        os.Exit(-1)
    }

    result := m.Run()  // run all UTs

    stop()
    os.Exit(result)
}
```

Special test-only packages

If you need to add test-only code as part of a package, you can place it in a package that ends in `_test`

That package, like `XXX_test.go` files, will not be included in a regular build

Unlike normal test files, it will only be allowed to access *exported* identifiers, so it's useful for "opaque" or "black-box" tests

```go
// file myfunc_test.go

package myfunc_test
// this package is not part of package myfunc, so
// it has no internal access
```

See this StackOverflow answer

Philosophy of Testing

Testing culture

“Your tests are the contract about what your software does and does not do. Unit tests at the package level should lock in the behaviour of the package’s API. They describe, in code, what the package promises to do. If there is a unit test for each input permutation, you have defined the contract for what the code will do in code, not documentation.”

“This is a contract you can assert as simply as typing `go test`. At any stage, you can know with a high degree of confidence, that the behaviour people relied on before your change continues to function after your change.” — Dave Cheney

Testing culture

You should assume your code doesn't work *unless*

- you have tests (unit, integration, etc.)
- they work correctly
- you run them
- they pass

Your work isn't done until you've added or updated the tests

This is basic code hygiene: **start clean, stay clean**

Psychology of computer programming

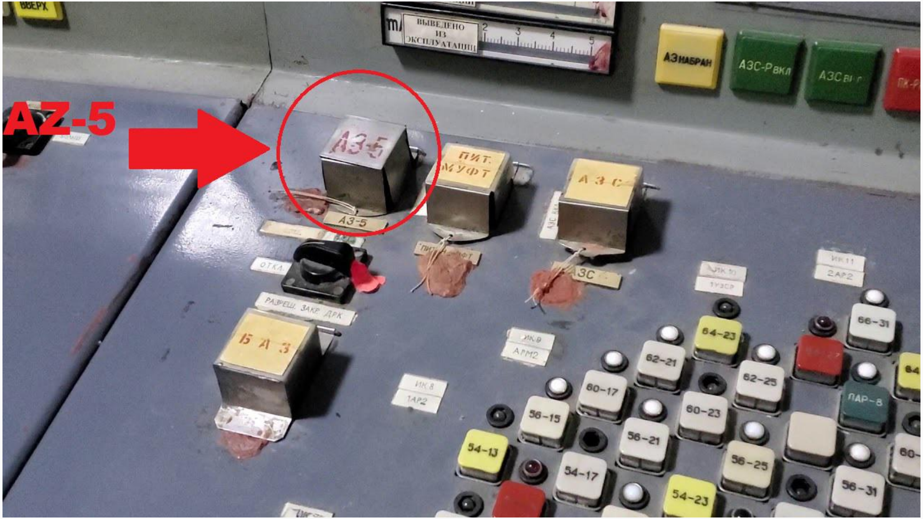
“The hardest bugs are those where your mental model of the situation is just wrong, so you can’t see the problem at all.” — Brian Kernighan

This issue applies to testing also

In general, **developers test to show that things are done & working** *according to their understanding of the problem & solution*

Most difficulties in software development are *failures of the imagination*

“I never thought they would press it”



Program correctness

There are eight levels of correctness “in order of increasing difficulty of achievement” (Gries & Conway)

1. it compiles [and passes static analysis]
2. it has no bugs that can be found just running the program
3. it works for some hand-picked test data
4. it works for typical, reasonable input
5. it works with test data chosen to be difficult
6. it works for all input that follows the specifications
7. it works for all valid inputs and likely error cases
8. **it works for all input**

“It works” means it produces the desired behavior or fails safely

Program correctness

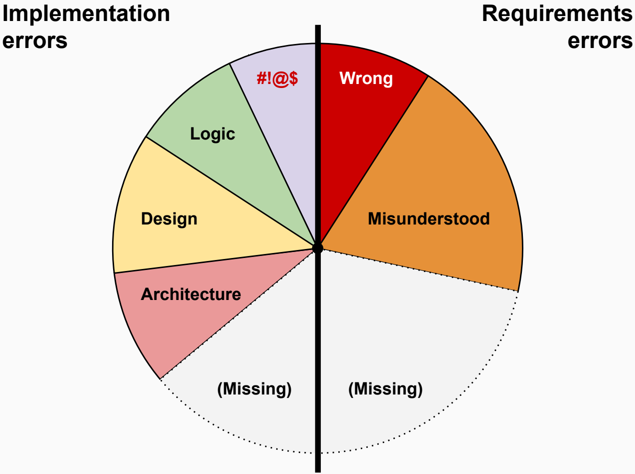
There are four distinct types of errors (Gries & Conway):

1. errors in understanding the problem requirements
2. errors in understanding the programming language
3. errors in understanding the underlying algorithm
4. errors where you knew better but simply slipped up (even experienced developers make mistakes)

> “Type 1 errors tend to increase as problems become larger, more varied, and less precisely stated.”

Even worse, some requirements may just be **missing**

Sources of errors



Developer testing is necessary

You should aim for 75–85% code coverage

- unit tests
- integration tests
- post-deployment sanity checks

Developers **must** be responsible for the quality of their code

They shouldn’t just “throw crap over the wall”

Tests can be part of your documentation

Testing is not “quality assurance”

Confusing “test” and “QA” is a basic mistake

- QA is a different discipline in software development
- we’re not dealing with a manufacturing process
- you can’t “test in” or prove quality

Testing is not about running “acceptance” tests to show that things work

It’s about surfacing defects by causing the system to fail (breaking it)

The *wrong testing mindset* leads to inadequate testing

Developer testing isn’t enough

You can have 100% code coverage and still be wrong

- the code may be bug-free, but not match the requirements
- the requirements may not match expectations
- you can’t test code that’s missing

Testers test to show that things *don’t* work

But they can’t test your system well if the requirements aren’t documented
(this is a major limitation of the agile method *as practiced* *)

Code & unit tests are simply not enough documentation

Reality check

Pick any two

- good
- fast
- cheap

You can’t have all three in the real world

Effective and thorough testing is hard & expensive

Software is annoying because most orgs pick fast and cheap over good