Who Tests the Tests?

Daniela Petruzalek

Principal Software Engineer @ JPMorgan Chase

Google Developer Expert - Go & GCP

@danicat83

Disclaimers

Any similarities with real code, including all names, types, and functions portrayed in this presentation is a mere coincidence.

No actual production code was harmed during the making of these slides.

Why do we write tests?

Tests are our “insurance policy”

The code does what is supposed to do

The code doesn’t do what it is not supposed to do

Tests are also design tools

They are the first clients of our code

There is a strong correlation between code that is **easy to test** and code that is **easy to read / maintain**

Measuring tests

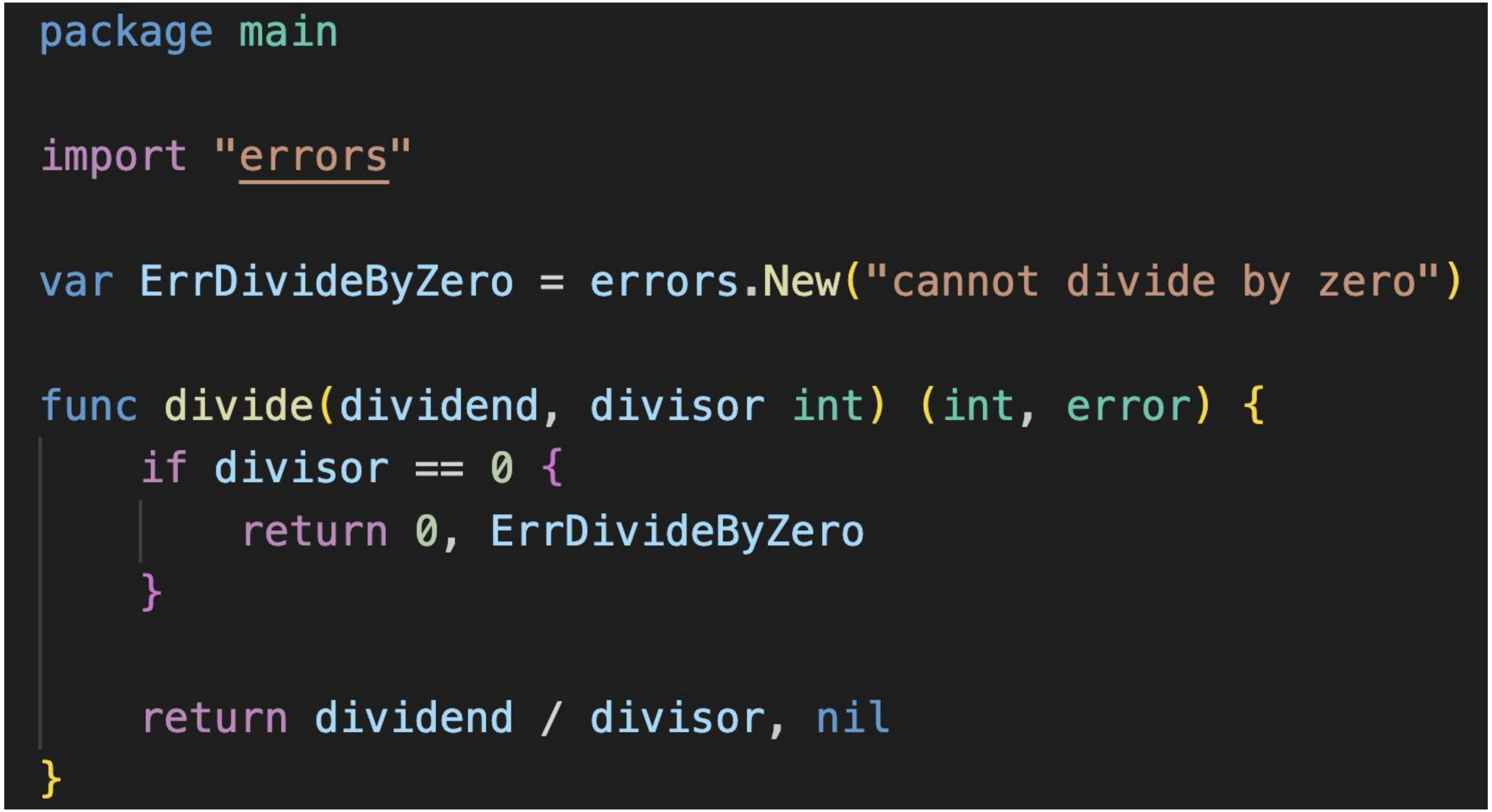
Code coverage: % of the code touched by the tests

Test coverage: % of the functionality tested

```
$ go test -cover          # unit tests

$ go build -cover         # integration tests
```

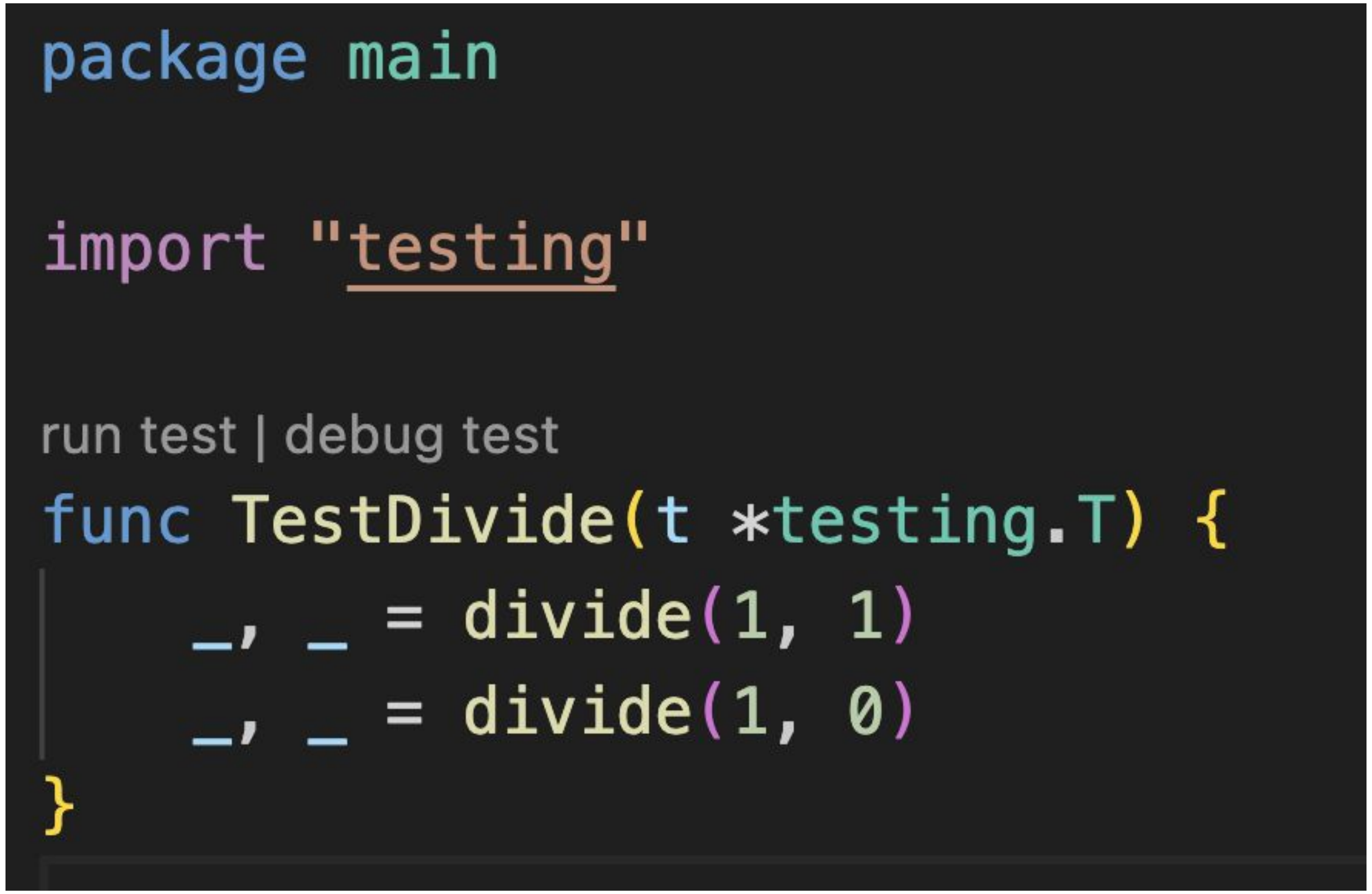
A metric that becomes a goal stops being a good metric

Goodhart’s Law



```go
package main

import "errors"

var ErrDivideByZero = errors.New("cannot divide by zero")

func divide(dividend, divisor int) (int, error) {
    if divisor == 0 {
        return 0, ErrDivideByZero
    }

    return dividend / divisor, nil
}
```

Coverage can be easily **faked**



```
package main

import "testing"

run test | debug test
func TestDivide(t *testing.T) {
    _, _ = divide(1, 1)
    _, _ = divide(1, 0)
}
```

=> 100% code coverage!

Mutation Testing

What is Mutation Testing?

Mutation testing is a technique to find flaws in tests by introducing **small modifications** (mutations) to a program before running the test suite.

A few possible mutations

Reversing boolean operators: < to >, == to !=, || to &&

Changing operators: * to /, + to -

Force boolean expressions to true or false

Statement deletion: remove else, remove function body

Value mutation: small to big, big to small

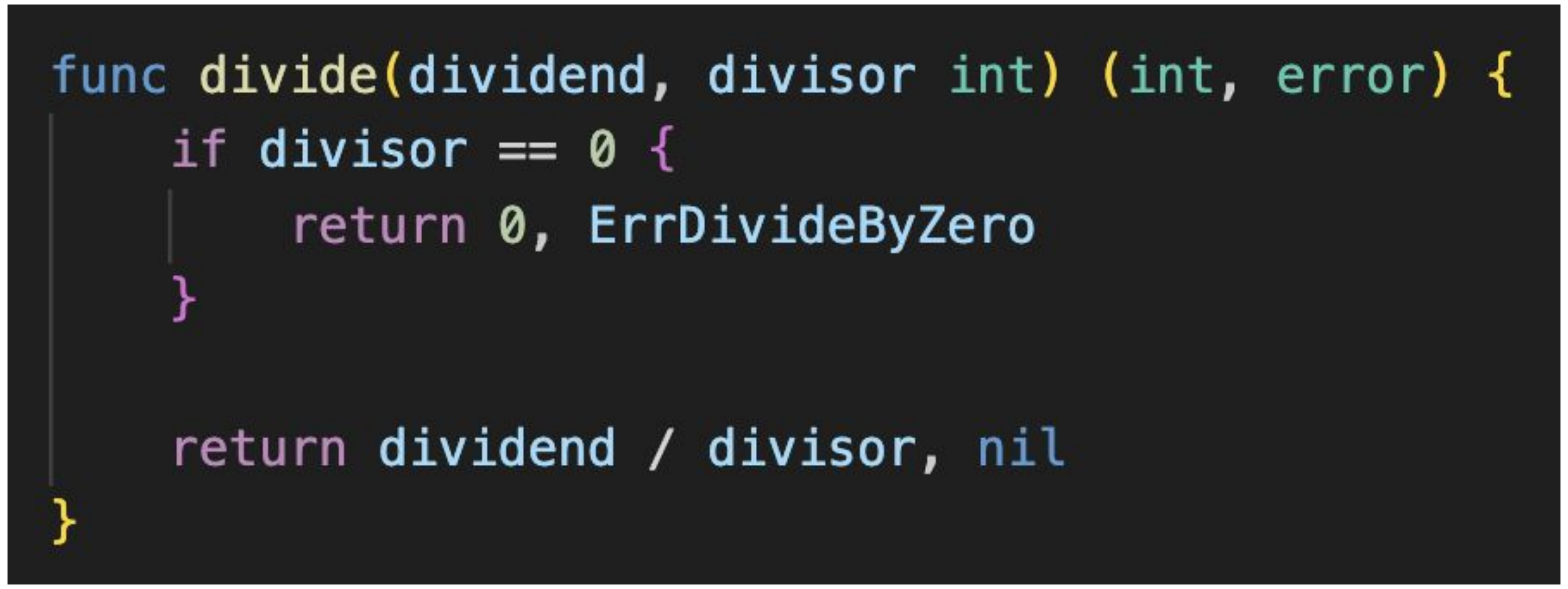
Testing mutants

If, after the mutation, the test:

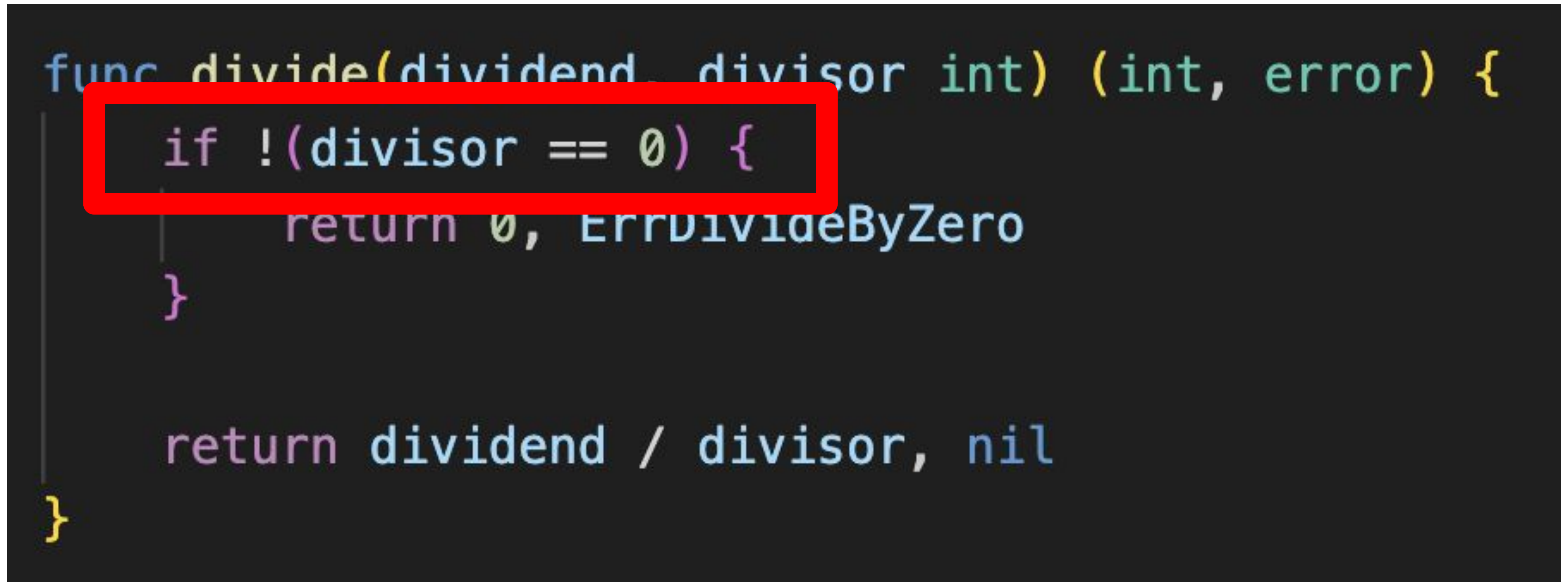
- **FAIL**, we say the mutant was **KILLED**
- **PASS**, we say the mutant **SURVIVED**

The quality of the test suite is measured by **how many mutants it kills**

Example #1: reverse if conditional

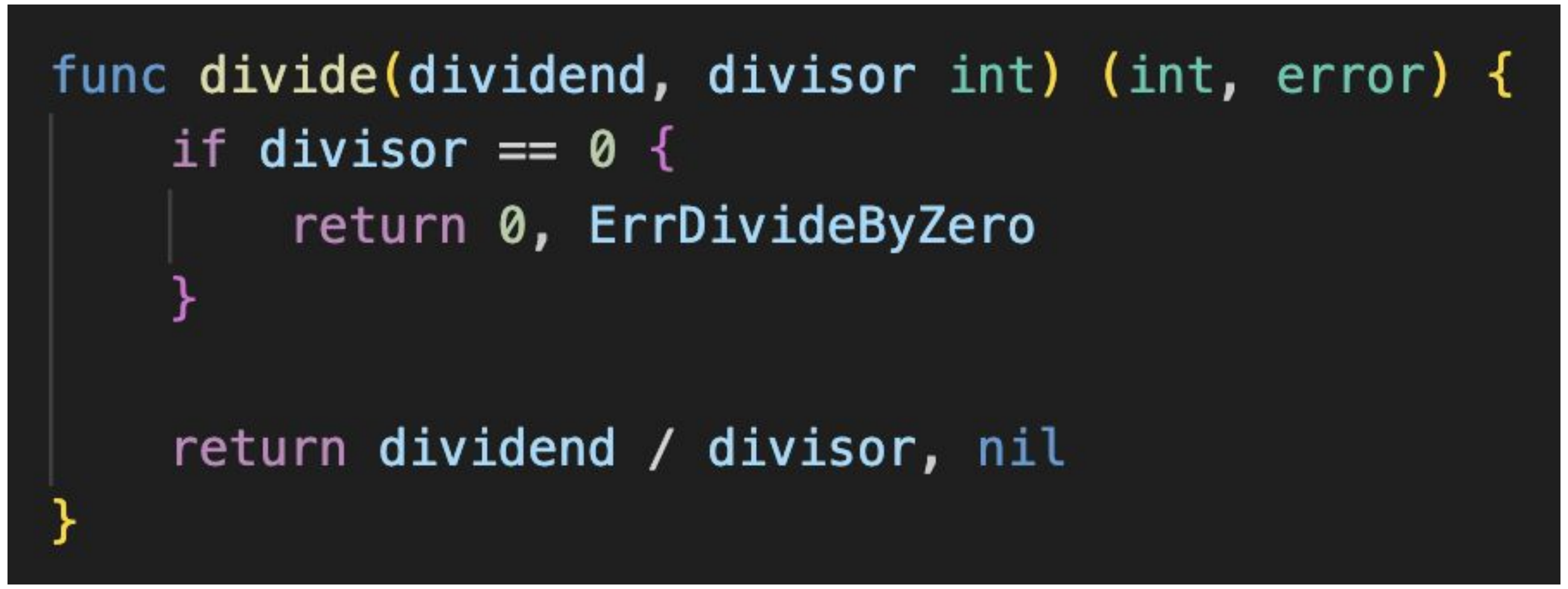


```go
func divide(dividend, divisor int) (int, error) {
    if divisor == 0 {
        return 0, ErrDivideByZero
    }

    return dividend / divisor, nil
}
```

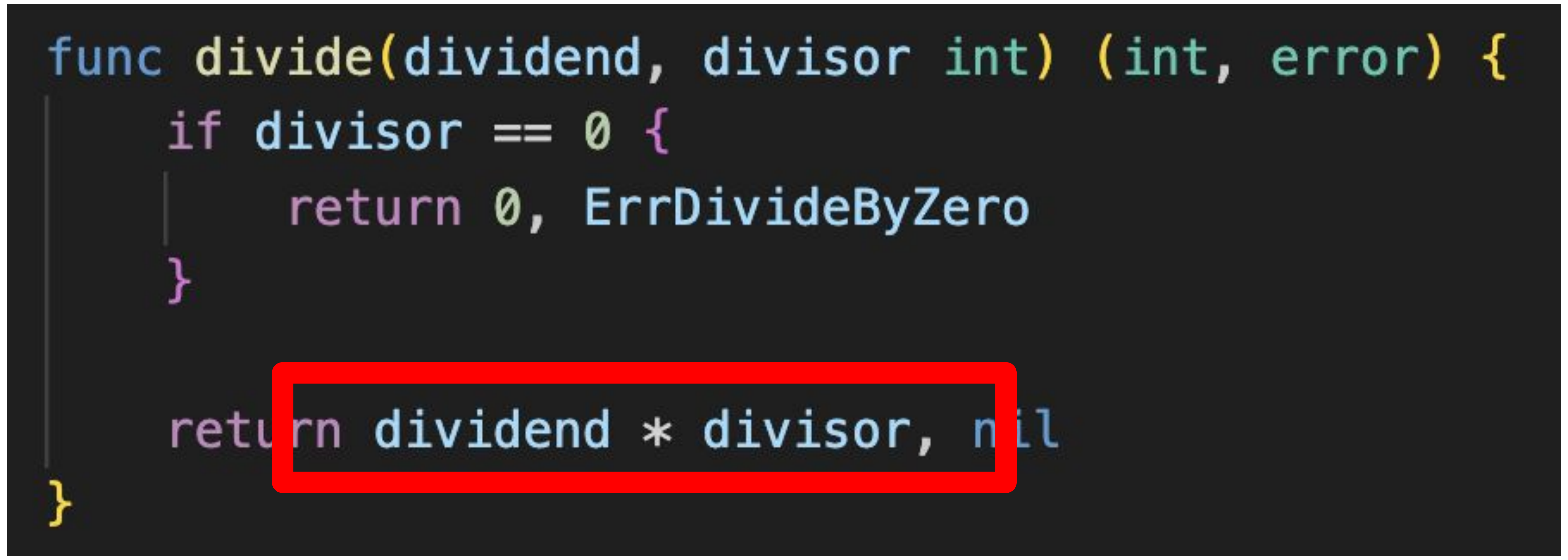


```go
func divide(dividend, divisor int) (int, error) {
    if !(divisor == 0) {
        return 0, ErrDivideByZero
    }

    return dividend / divisor, nil
}
```

Example #2: change binary operator



```
func divide(dividend, divisor int) (int, error) {
    if divisor == 0 {
        return 0, ErrDivideByZero
    }

    return dividend / divisor, nil
}
```



```
func divide(dividend, divisor int) (int, error) {
    if divisor == 0 {
        return 0, ErrDivideByZero
    }

    return dividend * divisor, nil
}
```

Mutation test results

Mutation #1: reverse if conditional

- TestDivide: **KILLED**

- TestDivideByZero: **KILLED**

Mutation #2: change binary operator / to *

- TestDivide: **SURVIVED**

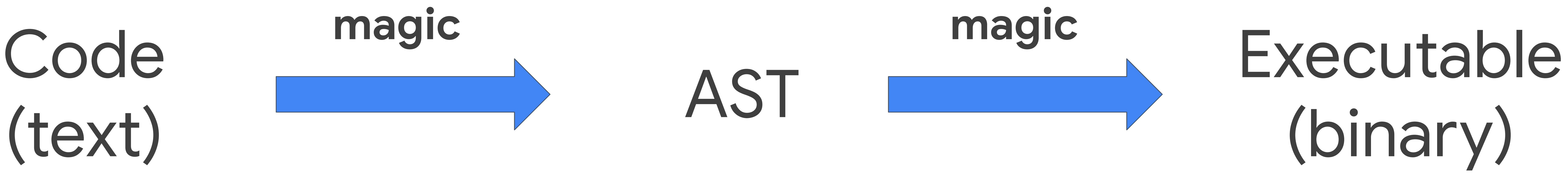
- TestDivideByZero: **SURVIVED**

Mutation Score
# Mutants Killed / Total Mutants = 50%

Can we do this in Go?

Abstract Syntax Trees (AST)

Intermediate representation of the code, sitting in between text and binary formats.



Code
(text)

magic

AST

magic

Executable
(binary)

AST Representation

Binary expression: **A + B**

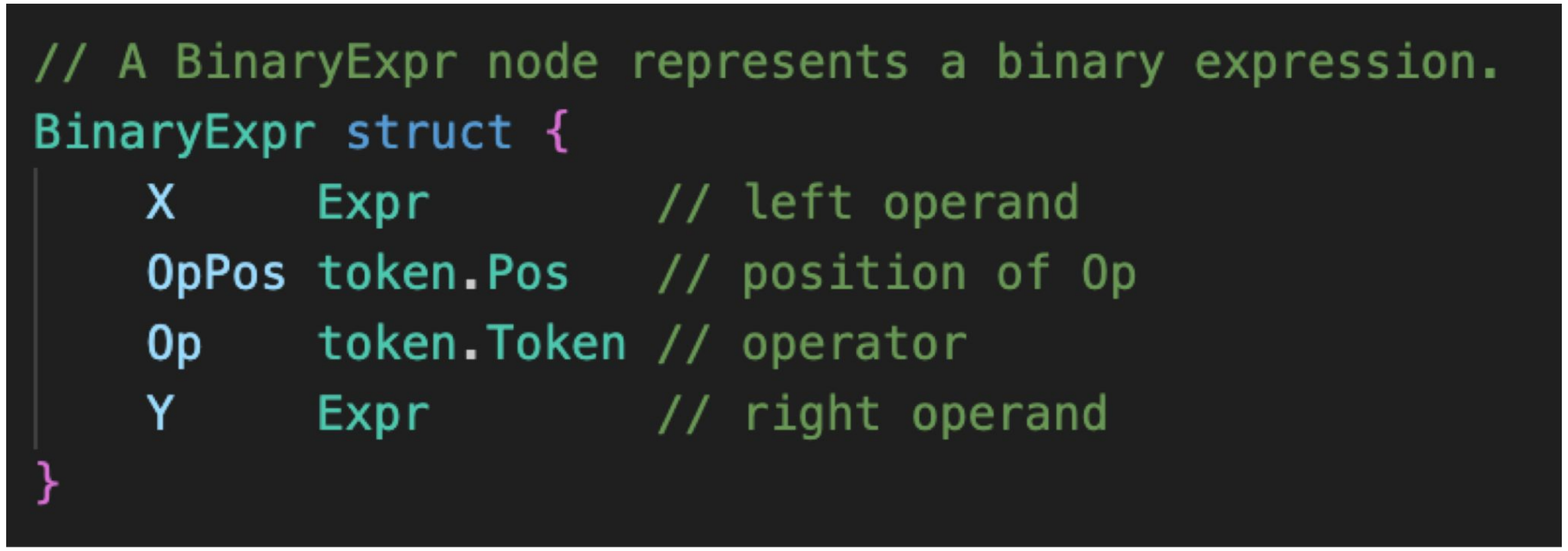
ast.BinaryExpr{

X: **A,**

Y: **B,**

Op: **token.ADD**

}



```
// A BinaryExpr node represents a binary expression.
BinaryExpr struct {
        X     Expr        // left operand
        OpPos token.Pos   // position of Op
        Op    token.Token // operator
        Y     Expr        // right operand
}
```

AST makes easy to apply mutations

```
ast.BinaryExpr {
  X: exprA,
  Y: exprB,
  Op: token.ADD,
}
```

→

```
ast.BinaryExpr {
  X: exprA,
  Y: exprB,
  Op: token.SUB,
}
```

How to kill mutants with Go

Load code from disk

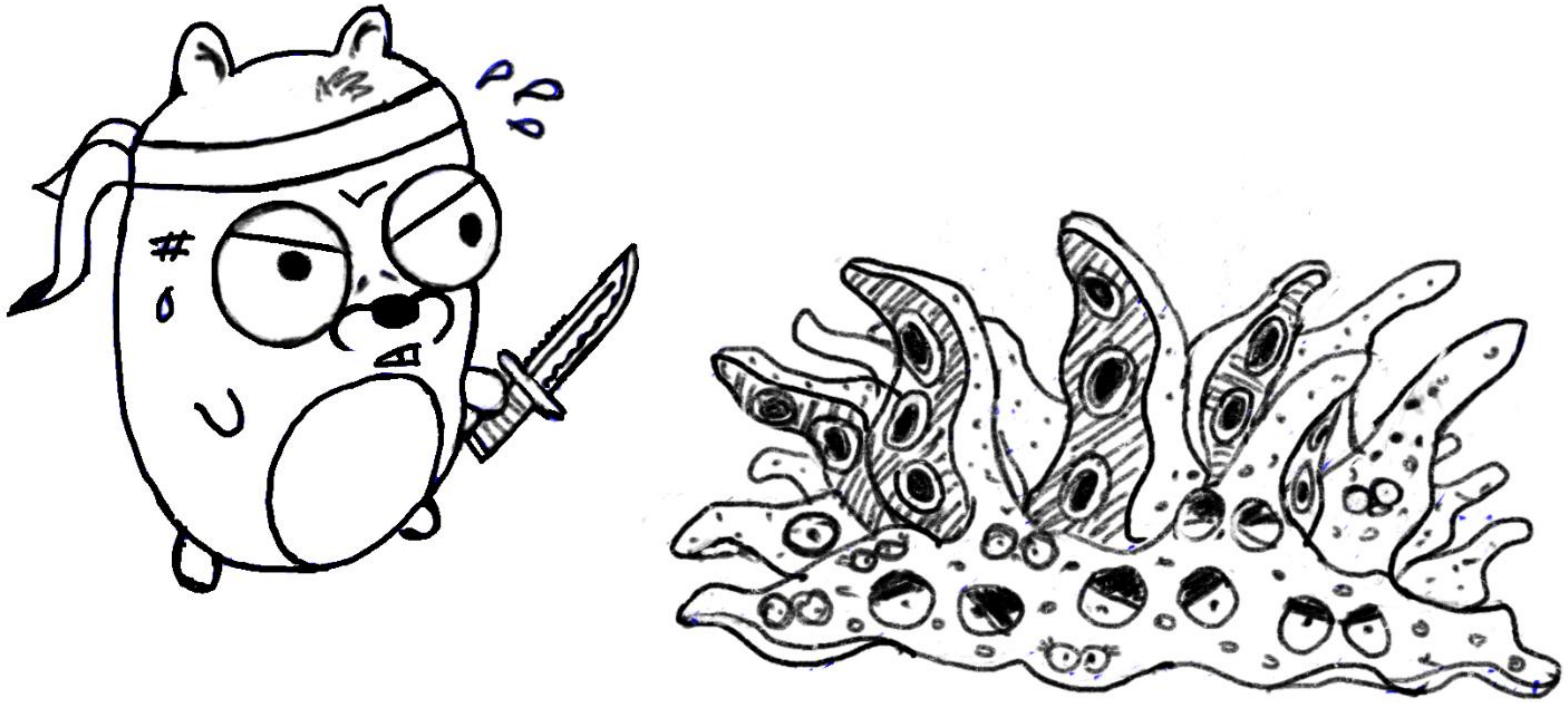
Parse code to AST

Apply mutation(s)

Write code back to disk

Run go test

Profit!



Mutation test tools

Gremlins: https://github.com/go-gremlins/gremlins

Go Mutesting: https://github.com/zimmski/go-mutesting

Go Mutate: https://github.com/zabawaba99/gomutate

Selene: https://github.com/danicat/selene

Takeaways

Write tests for the **right reasons**

Choose your test inputs wisely

One mutation alone is not enough to tell if a test is good or bad

But performing all possible mutations will be expensive

Mutation testing in Go is not yet mature, but we can make it happen daniela.petruzalek@gmail.com

@danicat

@danicat83

https://github.com/danicat/public-speaking