Software Quality

Respond to requirement change with confident

Testing purpose

- Validate that our application will work as intended.
- Ensure that we don’t have mistakes in our logic.
- Ensure new changes don’t break.

Go Testing

Package testing provides support for automated testing

Convention

- Ending a file's name with "**_test.go**"
- Put the test file in the same package as the one being tested
- func with a signature "**func Test**Xxx(t ***testing.T**)"

Test Functions

```go
import "testing"

func TestName(t *testing.T) {
   // ...
}

func TestMultiply(t *testing.T) { /* ... */ }

func TestSum(t *testing.T) { /* ... */ }

func TestMinus(t *testing.T) { /* ... */ }
```

Test Command

- `go test  .`
- `go test  ./...`
- `go test -v      .`
- `go test -run TestName .`

Sample Test

- `sum.go`
- `sum_test.go`

Create Go Project

- Create new folder "softQ"
- cd softQ
- git init
- go mod init github.com/<username>/softQ
- go mod tidy

Signal failure

```
func TestName(t *testing.T) {
        got := sum(1,2)

        if got != 3 {
        t.Error("it not 3")
    }
 }
```

```
func (c *T) Error(args ...any)

func (c *T) Errorf(format string, args ...any)
```

Subtests

```
func TestSum(t *testing.T) {
        // <setup code>
        t.Run("should return 3 when input 1 and 2", func(t *testing.T) {
                got := sum(1, 2)
                if got != 3 {
                        t.Error("it not 3")
                }
        })
        t.Run("should…", func(t *testing.T){
                // TODO
    })
    // <teardown code>
}
```

Arrange, Act, Assert pattern

```go
func TestSum(t *testing.T) {
	t.Run("should return 3 when 1 and 2", func(t *testing.T) {
		// Arrange
		want := 3

		// Act
		got := sum(1, 2)

		// Assert
		if got != want {
			t.Errorf("sum(1, 2) = %d; want %d", got, want)
		}
	})
}
```

Testing Techniques

Minimize numbers of test case

Ticket System

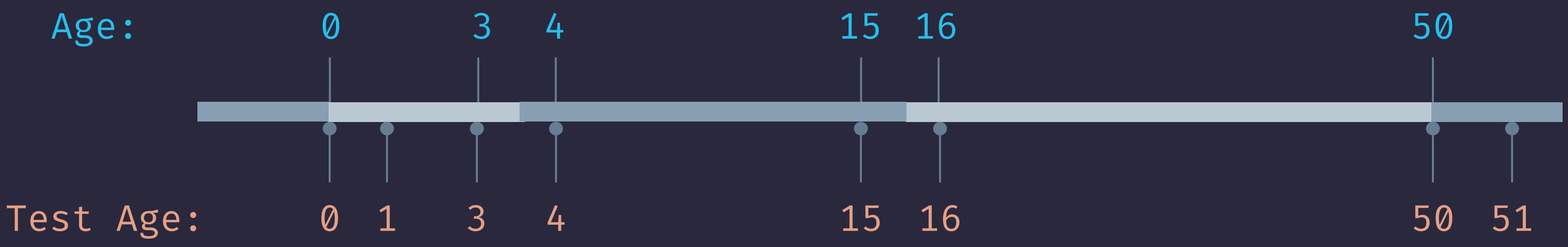
```
GIVEN we have price age ranges based
WHEN the user selects a range
THEN  the price is the correct one
```

Age	0 to 3	4 to 15	16 to 50	>50
Price	Free ticket	$15	$30	$5

Boundary values

- Minimum
- Just above the minimum
- A nominal value
- Just below the maximum
- Maximum

Age	0 to 3	4 to 15	16 to 50	>50
Price	Free ticket	$15	$30	$5



Decision table

Combinations of different input. All possible combination it easy to see.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Username	False	True	False	True
Password	False	False	True	True
Output	Error	Error	Error	Log in

Error guessing

It requires a good knowledge of the application

- Divide by zero
- Entering blank spaces in the text fields
- Pressing the submit button without entering values
- Uploading files exceeding maximum limits
- Null pointer exception
- Invalid parameters
- And many more …

Test Price Ticket

```go
func Price(age int) float64 {
	if age <= 3 {
		return 0
	}
	if age <= 15 {
		return 15
	}
	if age <= 50 {
		return 30
	}
	return 5
}
```

Test Sum all

```go
package sum

func sum(xs ...int) int {
        var total int
        for _, num := range xs {
                total += num
        }
        return total
}
```

Test Pyramid

metaphor grouping software test cases into group of different granularity

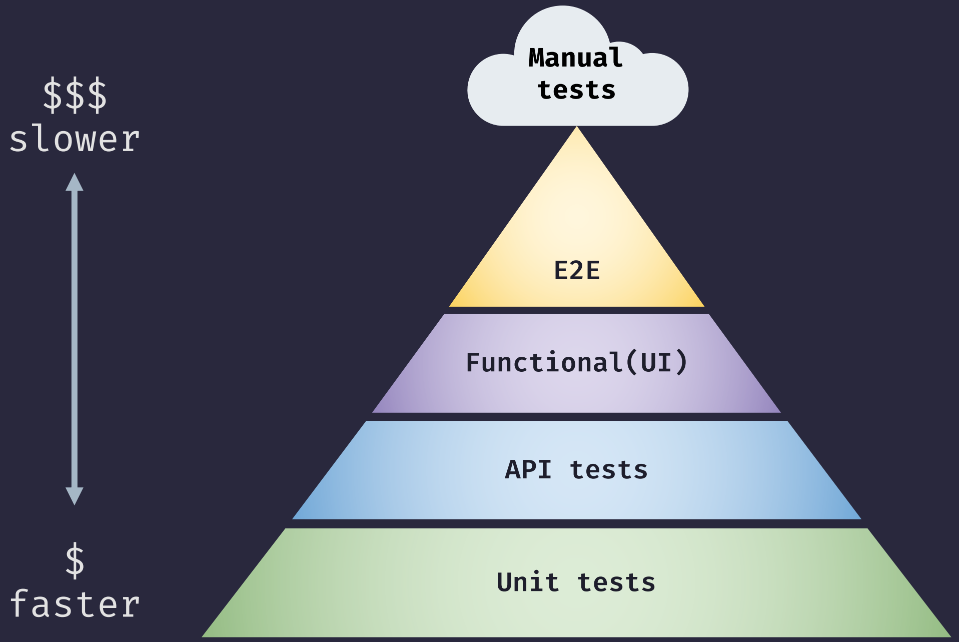
UNIT TEST

- WHAT: Tests that individual units of code (class, method, function) works as intended.
- HOW: In isolation. Typically replaces external collaborators with Test Doubles (mocks, stubs). Tests at least test the public interface of the class.
- WHY: Gives developers confidence that changes (refactoring) didn’t break anything. Enables TDD.

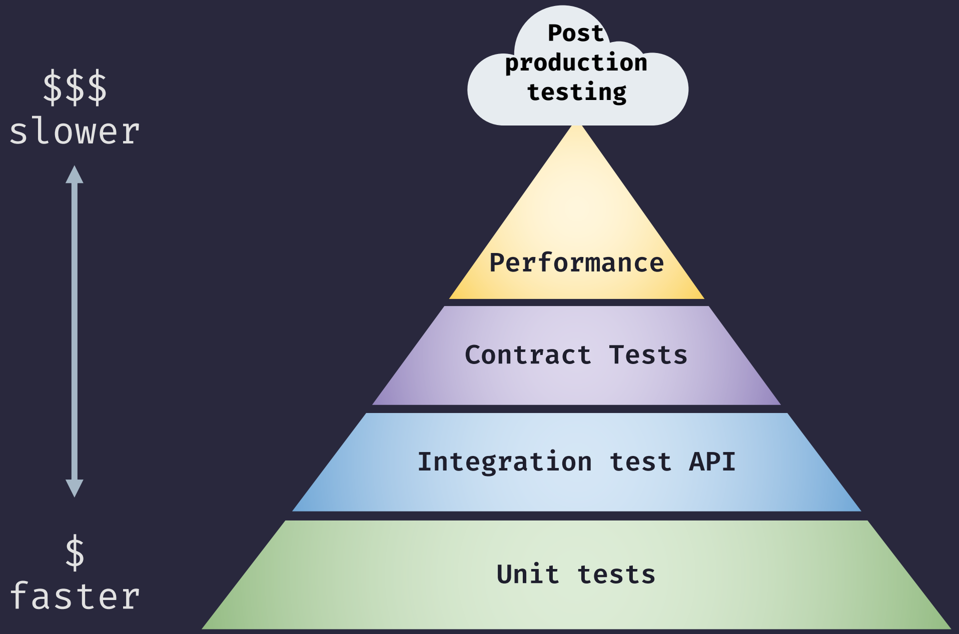
INTEGRATION TEST

- WHAT: Tests whether independent software units work correctly when they are connected.
- HOW: Activate multiple units and perform higher level tests against them all to ensure that they operate together.
- WHY: Tests if many separate units (classes, modules) work together as expected.

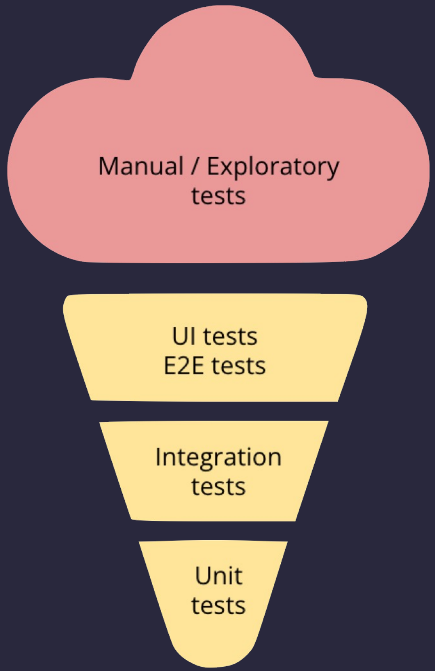
TEST PYRAMID: WEB SYSTEM



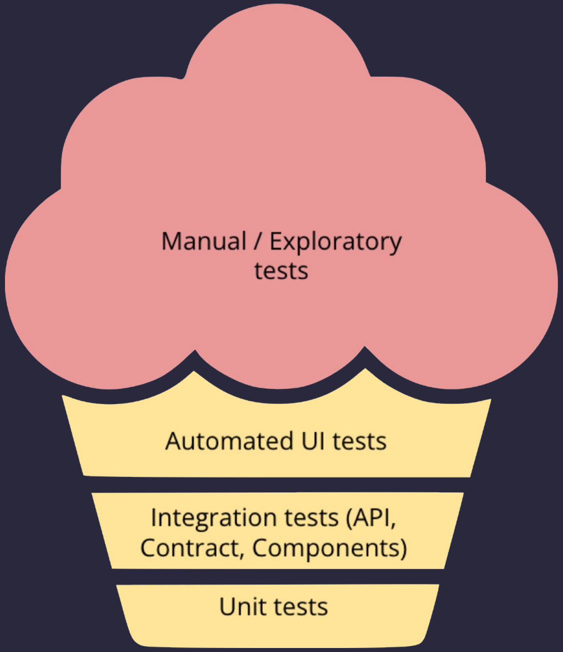
TEST PYRAMID: REST API



ANTI PATTERNS: ICE CREAM CONE



ANTI PATTERNS: CUPCAKE

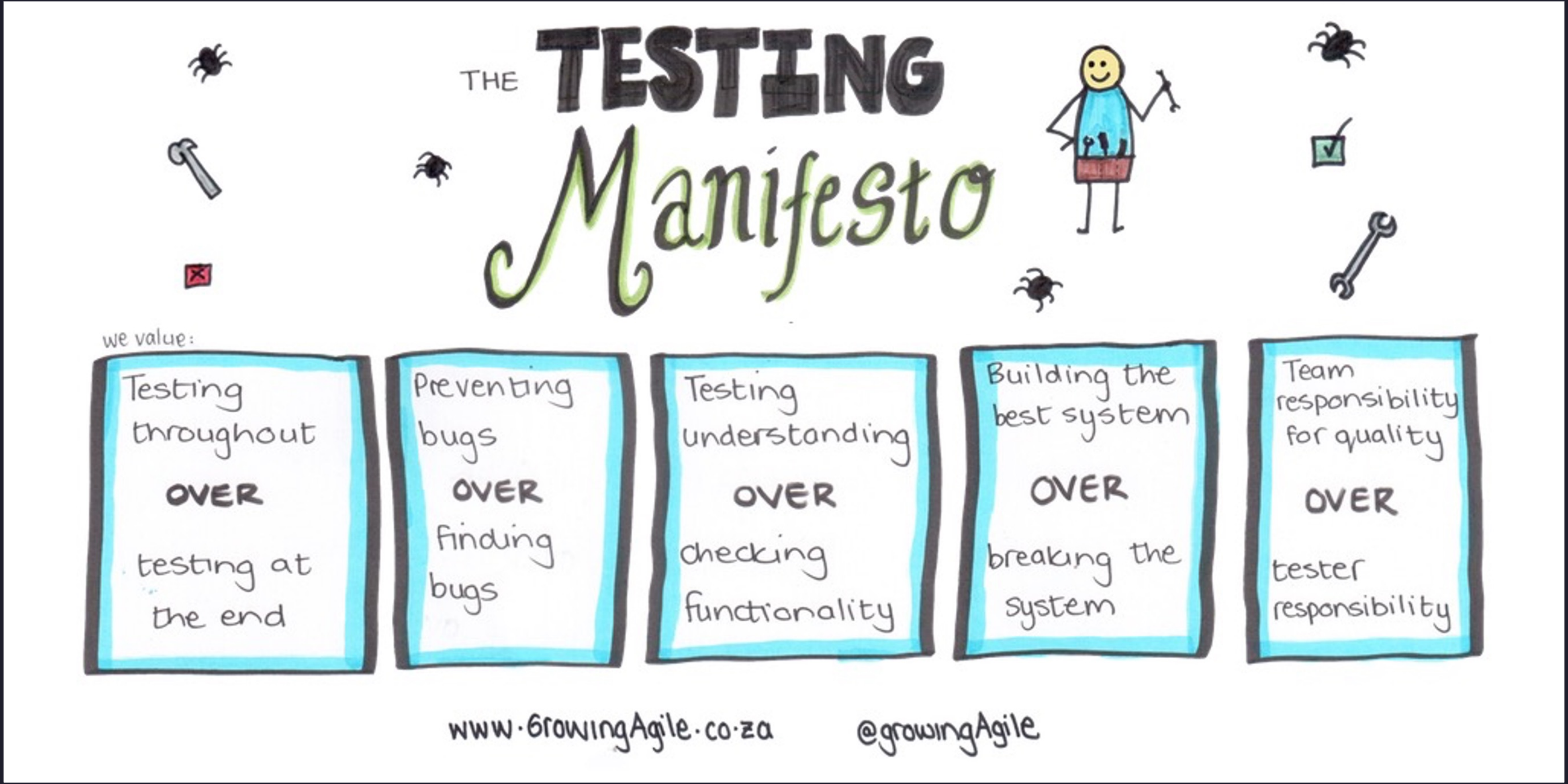

Manual / Exploratory tests
Automated UI tests
Integration tests (API, Contract, Components)
Unit tests

HOW TO ACHIEVE THE IDEAL PYRAMID?

There is no Perfect Pyramid! Use the pyramid to stimulate discussion about tradeoffs!

- How can we push this test down the pyramid?
- What needs to be higher up the pyramid?
- How much value are we getting by putting this higher up the pyramid?
- How much effort are we willing to invest to get faster feedback?
- Always remember the whole team is responsible for quality.

TESTING ≠ QUALITY



THE TESTING Manifesto

we value:

Testing throughout **OVER** testing at the end

Preventing bugs **OVER** finding bugs

Testing understanding **OVER** checking functionality

Building the best system **OVER** breaking the system

Team responsibility for quality **OVER** tester responsibility

www.GrowingAgile.co.za @growingAgile

Go testing trick

Empower way writing test in Go

Testable

```
func TestTicketPrice(t *testing.T) {
            tests := []struct {
                        name string
                        age  int
                        want float64
            }{
                        {"Free Ticket when age under 3", 3, 0.0},
                        {"Ticket $15 when age at 4 year old", 4, 15.0},
                        {"Ticket $15 when age is 15", 15, 15.0},
            }

            for _, tt := range tests {
                        t.Run(tt.name, func(t *testing.T) {
                                    got := Price(tt.age)
                                    if got != tt.want {
                                                t.Errorf("Price(%d) = %f; want %f", tt.age, got,
tt.want)
                                    }
                        })
            }
}
```

Coverage

Run test with coverage

- `go test -cover`

Generating an HTML coverage report

- `go test -cover -coverprofile=c.out`
- `go tool cover -html=c.out -o coverage.html`

Black box testing

```go
package ticket_test

import (
	"testing"

	"github.com/anuchito/ticket"
)

func TestTicket(t *testing.T) {
	t.Run("should return 0 when age is 3", func(t *testing.T) {
		want := 0.0
		got := ticket.Price(3)
		if got != want {
			t.Errorf("Price(3) = %f; want %f", got, want)
		}
	})
}
```

Setup/Teardown

```go
package teardown

import "testing"

func setup(t *testing.T) func() {
	t.Log("setup")

	return func() {
		t.Log("teardown")
	}
}

func TestTeardown(t *testing.T) {
	teardown := setup(t)
	defer teardown()                    // t.Cleanup(teardown)

	// test...
}
```

Static code Analysis

- go fmt : formats go code
- **go vet** : reports suspicious constructs (https://staticcheck.io/docs/getting-started/)
- golint : reports poor coding style

staticcheck

- `go install -v honnef.co/go/tools/cmd/staticcheck@latest`
- `staticcheck ./…`

Environment

- mac/linux

```
export GOROOT=~/go1.x.x
export GOBIN=$GOPATH/bin
export PATH=$GOBIN:$GOROOT/bin:$PATH
```

- windows

```
set GOROOT=C:\go1.x.x
set GOBIN=%GOPATH%\bin
set PATH=%GOBIN%;%GOROOT%\bin;%PATH%
```

Go Document

Go Doc Example

Document Code as well as Test your code

    https://pkg.go.dev/strings@go1.17.6#ToUpper

https://pkg.go.dev/testing#hdr-Examples

```
func Example() { ... }

func ExampleF() { ... }

func ExampleT() { ... }

func ExampleT_M() { ... }
```

ExampleMinus

```go
package sum

import (
	"fmt"
)

func ExampleMinus() {
	fmt.Println(Minus(5, 2))
	// Output:
	// 3
}
```

- `go install -v golang.org/x/tools/cmd/godoc@latest`
- `godoc -http=:6060`

Test Http Server

Http request

```go
import (
	"encoding/json"
	"io/ioutil"
	"net/http"
)

type Response struct {
	ID	int	`json:"id"`
	Name	string	`json:"name"`
	Info	string	`json:"info"`
}
```

```go
func MakeHTTPCall(url string) (*Response, error) {
	resp, err := http.Get(url)
	if err != nil {
		return nil, err
	}
	body, err := ioutil.ReadAll(resp.Body)
	if err != nil {
		return nil, err
	}
	r := &Response{}
	if err := json.Unmarshal(body, r); err !=
nil {
		return nil, err
	}
	return r, nil
}
```

Local Server

```
func handler(w http.ResponseWriter, r *http.Request) {
        w.WriteHeader(http.StatusOK)
        w.Write([]byte(`{"id": 1, "name": "AnuchitO"}`))
}



server := httptest.NewServer(http.HandlerFunc(handler))
defer server.Close()

resp, err := MakeHTTPCall(server.URL)
```

Test Double

replace a production object for testing purposes

Test Double

- Dummies
- Stubs
- Spies
- Fakes
- Mocks

Dummies

- Objects are passed around but never actually used. usually they are just used to fill parameter lists.

Stubs

- Provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.

Spies

- Spies are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
- That’s what spy is - a stub that keeps track of invocations of its methods.

Fakes

- Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an InMemoryTestDatabase is a good example).

Mocks

- Some think of stubs as mocks; others do not even think of mocks as types of instances.
- It’s generally accepted to use "mocking" when thinking about creating objects that simulate the behavior of real objects or units.
- They have the same characteristics as the stubs & spies, with a bit more
- Mocks are pre-programmed with expectations which form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting.

Dependency Injection

- DI is a Technique not a library.

```go
import "database/sql"

type DB interface {
	Exec(query string, args ...interface{}) (sql.Result, error)
}

func execQuery(db *sql.DB, query string, args ...interface{}) (int64, error)
{
	res, err := db.Exec(query, args...)
	if err != nil {
		return 0, err
	}

	return res.RowsAffected()
}
```

Test library

Third-party library for testing utilities

testify

```
import (
	"testing"
	"github.com/stretchr/testify/assert"
)



func TestSomething(t *testing.T) {
	t.Run("equal", func(t *testing.T) {
		want := 555
		got := 555

		assert.Equal(t, want, got, "they should be equal")
	})
}
```

Matryer : is

```go
import (
	"strings"
	"testing"

	"github.com/matryer/is"
)

func Binary(b string) (bool, error) {
	return true, nil
}
```

```go
func TestSomething(t *testing.T) {
	is := is.New(t)

	b, err := Binary("0")

	is.NoErr(err)
	is.Equal(b, true)
	is.Equal([]string{"a", "b"}, []string{"a",
"b"})


	got := "anuchito is gopher"
	is.True(strings.Contains(got, "anuchito"))
}
```

Build Tag

A build constraint condition; which file should be included

- `go build -tags=name`

- `go test -tags=integration`
- `go test -v -tags integration`
- `go test -v -tags integration,db`

One Tag

```go
//go:build integration

package tag

import "testing"

func TestBuildTags(t *testing.T) {
	t.Log("build tags")
}
```

OR tag

```go
//go:build integration || db

package tag

import "testing"

func TestTagOR(t *testing.T) {
	t.Log("tag integration || db")
}
```

AND tag

```
//go:build integration && db

package tag

import "testing"

func TestTagAND(t *testing.T) {
	t.Log("tag integration && db")
}
```

NOT tag

```go
//go:build !integration

package tag

import "testing"

func TestBuildTagsNotIntegration(t *testing.T) {
	t.Log("build tags NOT integration")
}
```

F.I.R.S.T principles

Test should be F.I.R.S.T.

F.I.R.S.T

- **Fast**

  Each tests should be as fast as possible
- **Isolated**

  Each Test should not depend on one another
- **Repeatable**

  Each tests should be to run in every envs and result should be the same
- **Self-validating**

  Each tests should be able to auto-detect if it passed or not
- **Timely**

  Tests should be written in the correct time; follow TDD

End