Getting started with

Testing

Summary

- Intro
- Functions
- Testing
- Examples
- Best Practices

```
test("Distance for long range.”, () => {
  const a = Gps(49.87811, -119.46441)
  const b = Gps(49.86079, -119.49099)
  const distance = distanceTo(a, b)
  expect(distance).toEqual(2708.9)
}

function distanceTo(a: Gps, b: Gps): number {
    const x1 = toRadians(a.latitude)
    const x2 = toRadians(b.latitude)
    const y = toRadians(b.latitude - a.latitude)
    const z = toRadians(b.longitude - a.longitude)
    const q = sin(y / 2) * sin(y / 2) + cos(x1) *
              cos(x2) * sin(z / 2) * sin(z / 2)
    const c = 2 * atan2(sqrt(q), sqrt(1 - q))
    return (6371000 * c).roundAt(1)
  }
```

Intro

What is testing?

- Given the function input, is the output what you expected?
- function (input) => output

```
test(“Description of test”, () => {

  // Arrange input and expected output.
   const a = Gps(49.87811, -119.46441)
   const b = Gps(49.87921, -119.46553)
   const expected = 146.3

  // Act
   const distance = distanceTo(a, b)

  // Assert
  expect(distance).toEqual(expected)

})
```

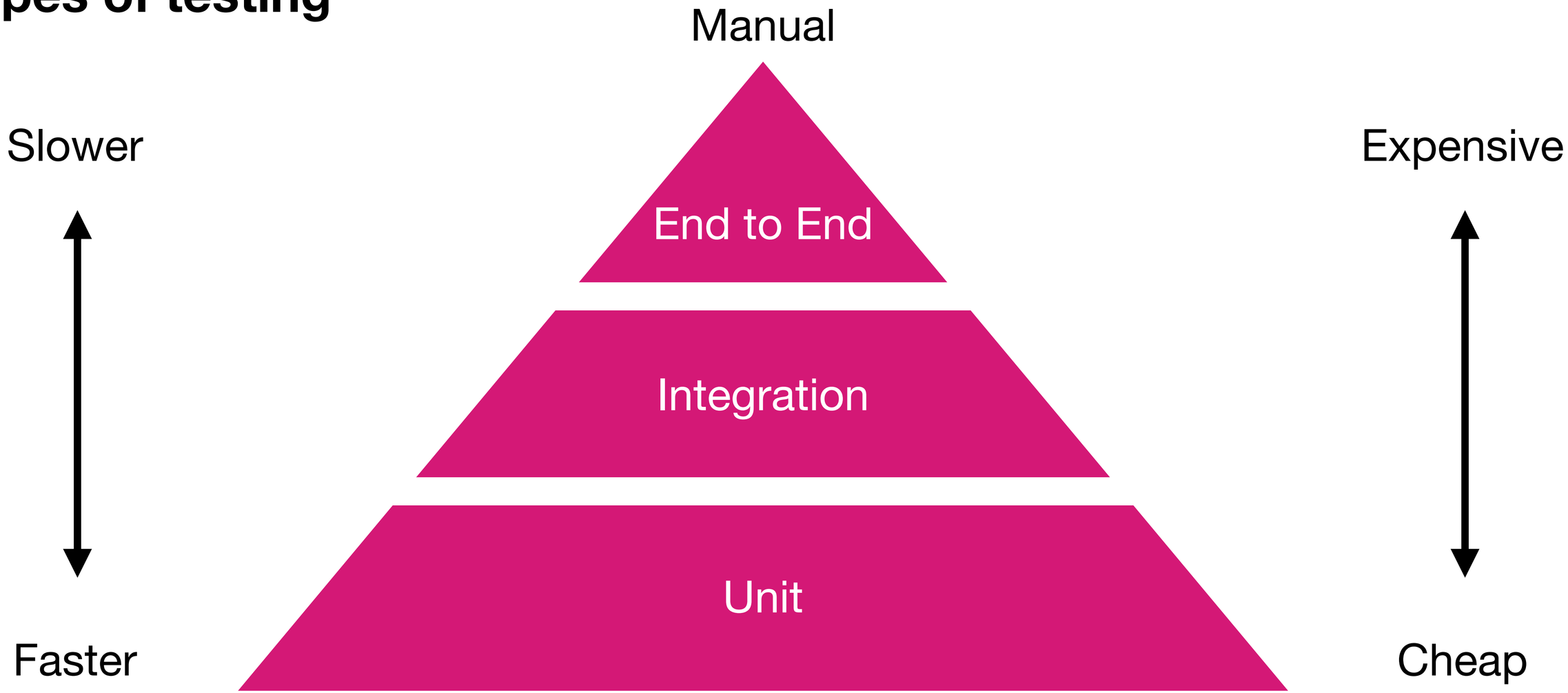
Intro

Why do we test?

- Confidence in the code
- Demonstrate a bug is fixed, or a feature is complete
- Bugs hide in untested code
- Safety net for refactoring
- Knowledge transfer
- Fast Feedback

Intro

Types of testing



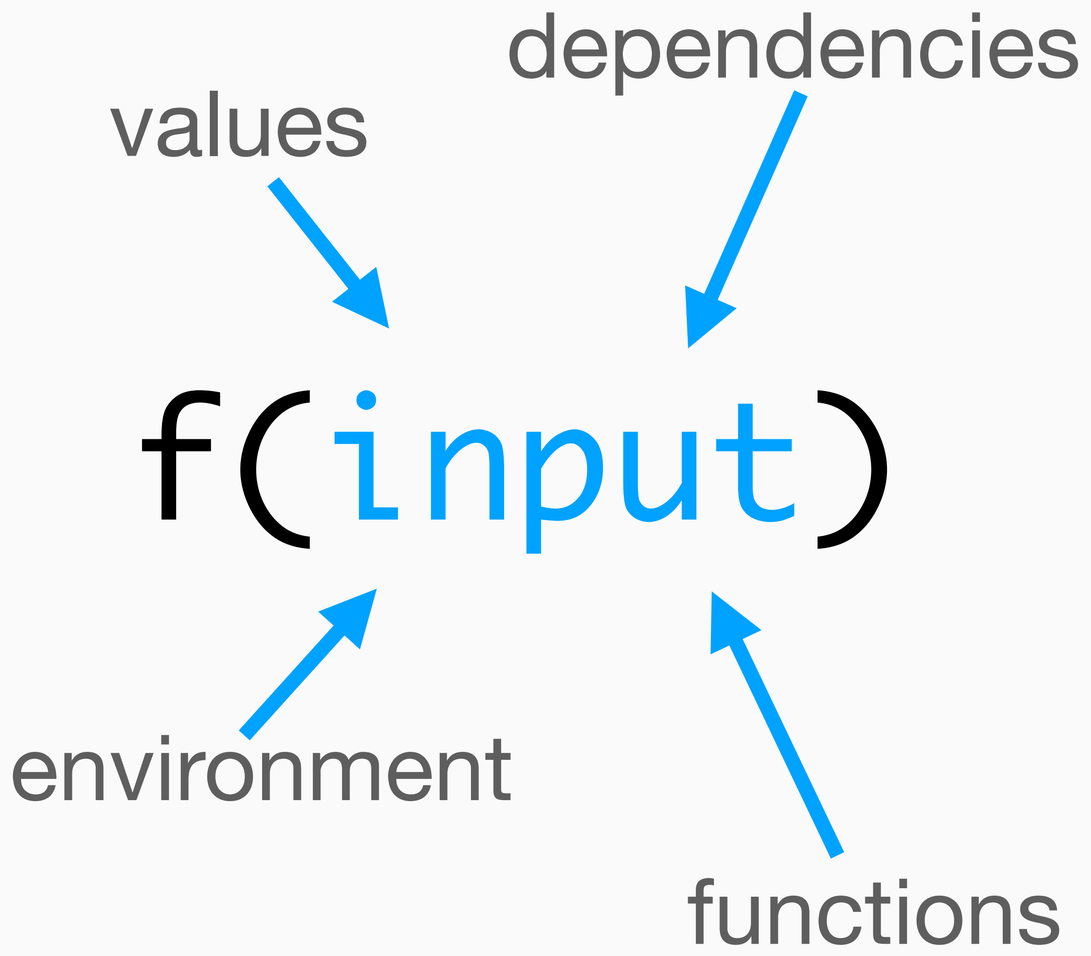
Functions

Favour a **complex network of simple objects**,
over a simple network of complex objects

Functions

Input

- Input is the only thing you can control during a test
- Provide various values to check all scenarios and edge cases
- Use ***test doubles*** (mock, stub, fake, dummy, spy) to replace production implementations of dependencies for controlling execution flow



Functions

Unexpected input

- Input not explicitly passed into the function
- Hidden dependencies
- Global and/or environment variables
- Produces unexpected output, or untestable
- Rework code to make these inputs explicit parameters

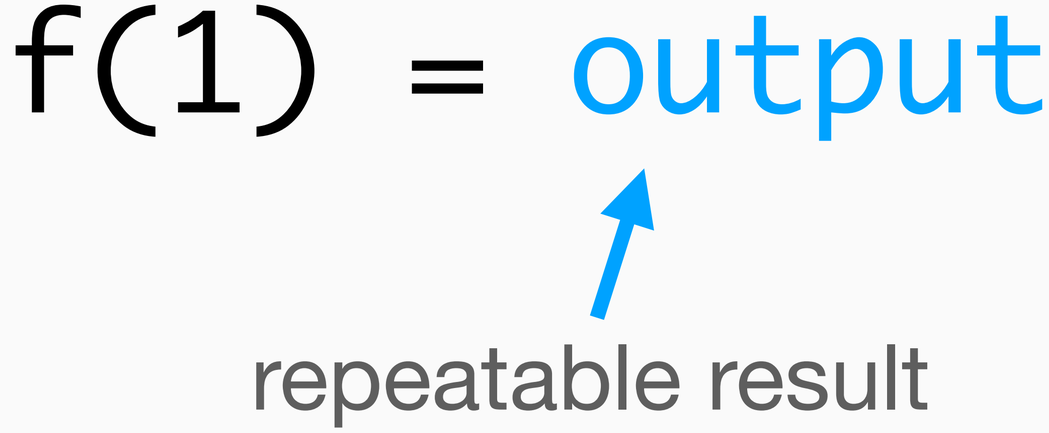
```
function isReady(
  a: number,
  b: number,
): Promise<boolean> {
   const x = Window.getScrollX()
  if (a > b) {
    return checkerService.check(a * b, x)
  } else {
    return new Promise(r => r(false))
  }
}

class CheckerService {
  public check(c: number, x: number) {
    return database.queryItemsWith(c)
  }
}
```

Functions

Output

- Repeatable, same result for given input



Functions

Side Effects

- User input/output
- Read/write to files, database, network
- Clock
- Changing variables outside of scope
- **Rework your code to wrap side effects** — dependencies

```
class MyServiceDefault(
  users: UserRepository,
  auth: AuthService,
  env: Environment,
  log: Logger,
  getNow: DateTimeNow
) implements MyService {
  function updateProcess(id: UserId) {
    this.log.info('Update process started')
    const key = this.env.get(‘key’)
    const user = this.users.withId(id)
    const now = this.getNow()
    const result = this.auth.set(user, key, now)
    return this.users.update(result.hello)
  }
}
```

Functions

SOLID Principles

- Single: one responsibility, one reason to change
- Open: for extension, closed for modification
- Liskov substitution: swap with subtypes and get the same results
- Interface segregation: don’t force implementation of interfaces that aren’t used — keep small interfaces
- Dependency Inversion: depend on abstractions not implementations

Makes testing easier

Testing

Testing is not only about proving something right,
but also **discovering where things go wrong**.

Testing

What to test?

- Public methods
- Helper functions
- Try to break your function
- Write tests to cover all possible combinations of input parameters

Hint: Writing ***console.log*** or ***println*** in the code base can be an indicator that a test would be helpful in that area

Testing

Ask yourself…

How could I test this?

→ What inputs would I use?

→ What output would I expect?

Testing

AAA

- **Arrange**: prepare the input parameters and output expectations
- **Act**: execute the function with the parameters
- **Assert**: Check the output against your expectations

```
// Arrange input params and expectations
const userId = UserId.init
const data = makeSamples(userId)
const users = new UserRepositoryStub(data)
const auth = new AuthServiceFake(userId)
const env = new EnvironmentStub({ key: ‘123’ })
const logger = new LoggerConsole()
const dateTime = () => DateTime.now()
const service = new MyServiceDefault(
  users, auth, env, logger, dateTime)
const expected = Either.right({…})

// Act on the function you want to test
const result = await service.updateProcess(userId)

// Assert the result is what you expected
expect(result).toEqual(expected)
```

Testing

Legacy Code

- You might have to start with integration tests depending on code coupling
- Often easier to isolate the function you want to test
- If your classes/functions have too many dependencies or input parameters, it’s a code smell that it might be doing too many things, and will make testing it more difficult
- Tightly coupled test code is brittle to changes

Examples

It’s hard enough to find an error in your code when you’re looking for it; it’s even harder when you’ve assumed your code is error-free.
— Steve McConnell, author, Code Complete

Examples

Pure Function

- Simplest to test
- Explicit dependencies and/or input
- No side effects

```
describe('Haversine Calculation', () => {

  test('Distance between two points',() => {
    const a = Gps(49.87811, -119.46441)
    const b = Gps(49.86079, -119.49099)
    const distance = distanceTo(a, b)
    expect(distance).toEqual(2708.9)
  })

  test('Zero distance',() => {
    const a = Gps(49.87811, -119.46441)
    const distance = distanceTo(a, a)
    expect(distance).toEqual(0.0)
  })

})
```

Examples

Impure Function

- Manage side effects by wrapping them
- DateTime

```
type GetNow = () => Date

test(“Shift forward”, () => {
  const now: GetNow = () =>
    new Date(“2024-04-28T08:30:00”)
  const expected = new Date(“2024-05-08T08:30:00”)
  const duration = 10.days
  const shifter = new Shifter(now)
  const result = shifter.add(duration)
  expect(result).toEqual(expected)
})
```

Examples

Database Query

- Simplified
- ***Integration test*** to verify a query works as expected
- Requires ***setup before*** test can run
- Requires ***cleanup after*** the test is run

```
describe('Activity Repository', () => {
  beforeAll(async () => {
    await migrateDatabase()
  })

  afterEach(async () => {
    await truncateData(['activities'])
  })

  test('Users who like tennis', async () => {
    expect.assertions(1)
    const db = new Database(loadTestConfig())
    db.insert(sampleData)
    const act = new ActivityRepository(db)
    const results = await act.findUsersFor('tennis')
    expect(results.length).toEqual(34)
  })
```

Examples

Api Request

- ***Integration test*** to verify an external service
- Load test keys/secrets from environment
- Sandbox actions
- Be mindful of limits; how often can you hit the end-point

```
describe('Clever Api', () => {
  const clever = new CleverApi(cleverTestConfig())

  test('Import students', async () => {
    expect.assertions(1)
    const expectedJson = { . . . }
    const request = new CleverImport('id')
    const result = await clever.import(request)
    expect(result).toContain(expectedJson)
  })

  test('Update student id', async () => {
    expect.assertions(1)
    const request = new CleverUpdate(sampleData)
    const result = await clever.update(request)
    expect(result.status).toEqual(200)
    expect(result.data).toEqual({ . . . })
```

Examples

End to End

- Requires ***setup before*** test can run
- Prebuilt virtual machine with fast start up time
- Contains everything needed to run - from database to front-end
- Requires ***cleanup after*** the test is run

```
describe('User Account', () => {
  beforeEach(async () => {
    await startVirtualMachine(config)
  })

  afterEach(async () => {
    await destroyVirtualMachine()
  })

  test('Signup', async ({ page }) => {
    await gotoSignup(page)
    await setEmail('hello@danicabrown.com', page);
    await expectSignupSuccess(page);
  })
})
```

Best Practices

Write more tests

Best Practices

Fast tests

- Blazing fast unit tests — milliseconds/seconds
- Unlikely to run tests if slow
- Run automatically on save for instant feedback
- Group slow tests together (eg, integration, slow)
  - Databases, external services, network calls are all “slow” compared to executing a function.

Best Practices

Fixing Bugs

- Recreate bug and write a failing test to demonstrate bug
- Fix the code where the bug is occurring
- Tests should now pass

Happy Testing!

Message me if you have any questions.

danicabrown.com