Programming 6

Testing

Ubiquitous language: testing types

- Unit testing
- Integration testing
- Smoke testing
- Functional testing
- Performance testing
- E2E testing
- Acceptance testing
- Stress testing
- Regression testing
- Accessibility testing
- White/blackbox testing
- Dry testing
- Security testing
- PEN testing
- Chaos testing
- …

What tests are needed?

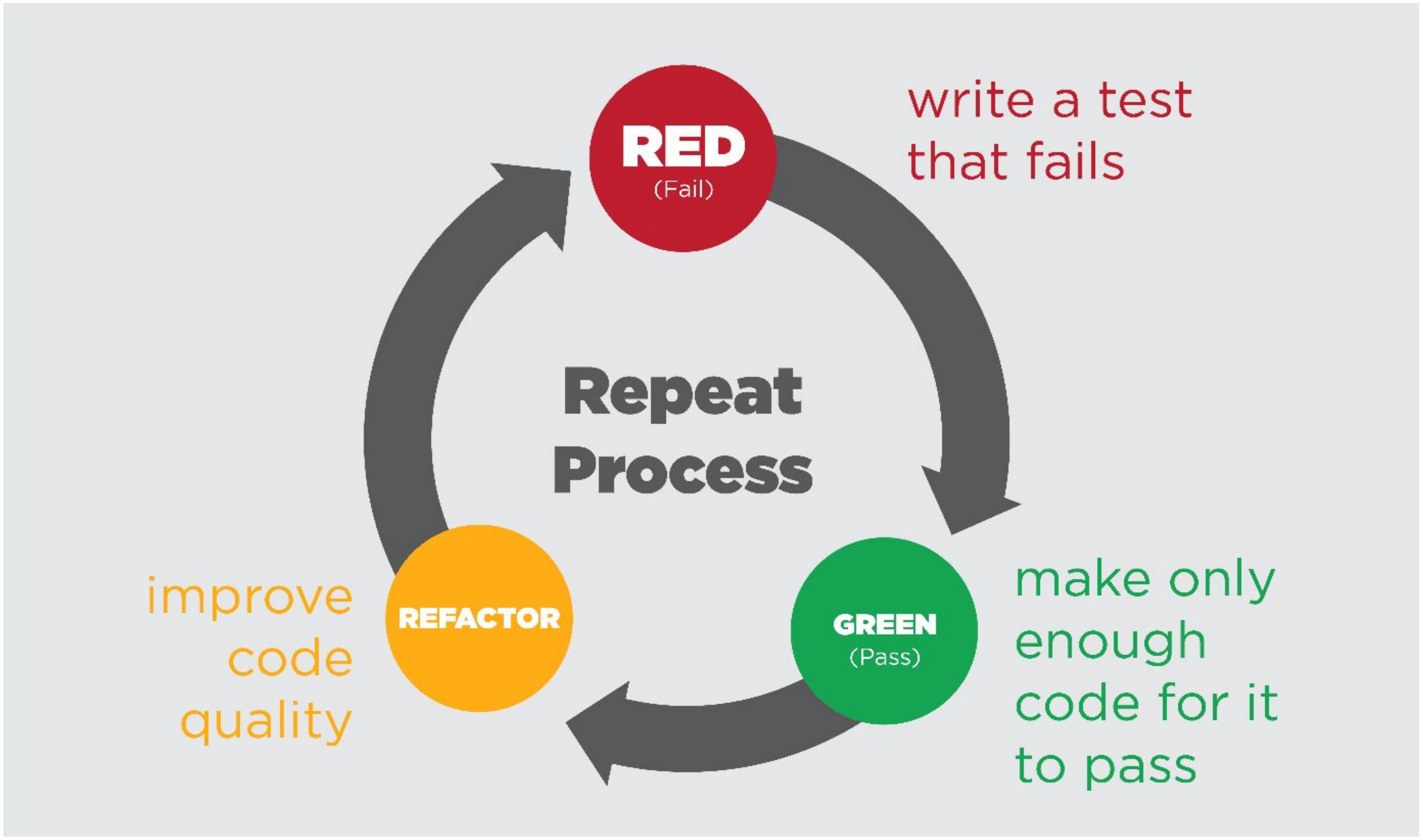
When writing a test you should ask yourself a couple of questions:

- Who will write test, maintain the test, run the test?
- What components do we cover in our test?
- Why are we testing?
- How do we know if our test was successful or not?
- What if we find a problem?

Wrong answers:

- That's the tester's job
- Don't know, as much as possible
- To get coverage up
- Tomorrow if the nightly build hangs… we know what's up.
- Some test might fail sometimes, just run it again, if it keeps failing log a ticket

TDD


RED
(Fail)
write a test that fails
GREEN
(Pass)
make only enough code for it to pass
REFACTOR
improve code quality
Repeat Process

Unit testing vs integration testing

Unit test	Integration test
Tests a small unit	Spans multiple units, are tested as a group
Should run fast, and easy to write	Slower to run
Low maintenance	High maintenance
White box testing	Black box testing
Easy to pinpoint problem	More difficult to pinpoint problem
Runs instantly at any time	Typically more planned
Behavior of a single unit	Tests integration of units
Should be able to run in parallel	Probably best to run them in order
No infrastructure or framework integration	Infrastructure and integration with framework

MockMVC -> integration or unit test?

A good unit test

- Easy to write
  - If your test case is not easy to write this is a smell!
  - Make your code testable!
- Documenting
  - Your test case documents this part of the code, so make sure it’s readable, understandable,
  - 3 A’s (Arrange - Act - Assert)
- Reproducible
  - Not flaky!
- Fast
  - They will run instantly, every time, so make them fast.
- Isolated
  - Do not make them integration tests!

Some implementation techniques

<u>“Test doubles”</u>

- Stubbing
  - Creating real objects as stand-ins
  - Order not really important
  - Easier to reuse
- Mocking
  - Not real objects
  - Mimic actions
  - Order matters
- Spying
  - Records the calls it gets
  - Counting, registering arguments
- (Fakes, Fixtures, …)

In hexagonal architecture: ports or dependencies in adapters that’s it!

Only application stuff gets its test double, no domain objects!!

Stubbing

- Extend or implement a class
- Use the concrete classes as test doubles.

- Reusable when creating a separate class
  - `PiggyBankLoadPortStub`

- Can hold state
  - `PiggyBankUpdatePortStub` holds new piggy bank which you can assert

- Can be much easier to read then mocking

- Sometimes easier to stub the class under test to make it more testable
  - static calls
  - fixed values etc.

Mocking

- Creates empty proxy and intercepts behavior
- You’ll need a framework in order to do this
- Can be messy
- If something cannot be mocked (or stubbed) this can indicate a smell of bad design.
- Mockito is the most known framework:

```
mock
verify
```

Spying

- “Special kind of mock”
- Creates a proxy around a concrete class:

```
ActivityWindow activityWindow = Mockito.spy(new ActivityWindow());
```

- Intercepts usages of that spy.
- Interesting when you inject a real Spring bean:

```
@Autowired
@Spy
private final MyRealSpringBean myRealSpringBean;
```

Testcontainers

- Start a container with a specific image
- Set your entire (or partial) environment up
- Do a real integration test with some of your infrastructure included

Annotations

```
@Mock

@Captor

@MockBean

@InjectMocks

@Spy

@ExtendWith(MockitoExtension.class)

@ExtendWith(SpringExtension.class)

@SpringBootTest

@TestConfiguration

@ContextConfiguration

@Import

@TestContainers

@AutoConfigureMockMvc
```

Testing your architecture

https://www.archunit.org

ArchUnit is a tool that makes it possible to test our architecture

For instance:

```
@ArchTest
static final ArchRule domainShouldNotDependOnAnyOtherLayerRule =
      noClasses().that().resideInAPackage( DOMAIN_LAYER)
            .should().dependOnClassesThat().resideInAnyPackage(
                  ADAPTER_LAYER,
                  PORT_LAYER,
                  CORE_LAYER
            )
            .because( "This conflicts with hexagonal architecture: Domain
should not depend on other layers." );
```

Testing your architecture

https://www.archunit.org

It makes it possible to test code guidelines

```
@ArchTest
static final ArchRule doNotUseJunit3StyleOfTests =
     noMethods().that().areAnnotatedWith(Test.class).and()

.areDeclaredInClassesThat().haveSimpleNameEndingWith("Test")
              .should().haveNameStartingWith("test")
              .because("prefixing tests with 'test' does not have
an added value.");
```

The project

Write at least:

- 1 architecture test
- 1 integration test
- 1 unit test using mocking
- 1 unit test using stubbing
- 1 test using Testcontainers
  - Smoke test 1 infrastructure component.
  - The test using Testcontainers does <u>not</u> count as the "1 integration test" mentioned above.

Questions?