Software testing anti-patterns

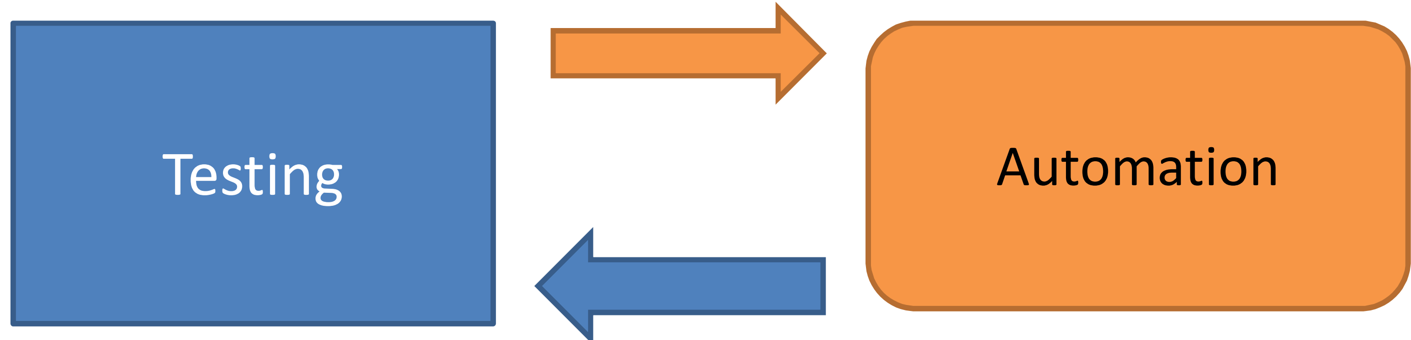
Make IT

October 2019

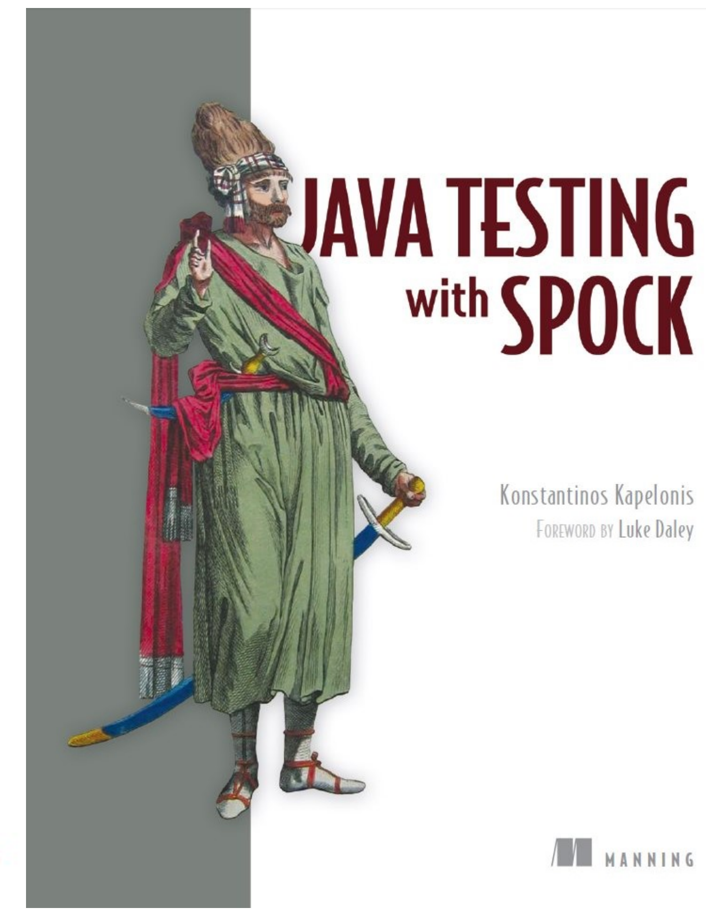
Kostis Kapelonis

Antipattern – common mistake

Things I love



Things I write



Roger

Changed how we do software testing

July 26, 2017

Format: Paperback | **Verified Purchase**

This book was great in teaching how and why to use Spock for testing. We have since built our testing methodologies around Spock based on techniques learned from this book. Our non-technical staff finds Spock tests much easier to understand than straight JUnit. This book was very readable and had very good examples.

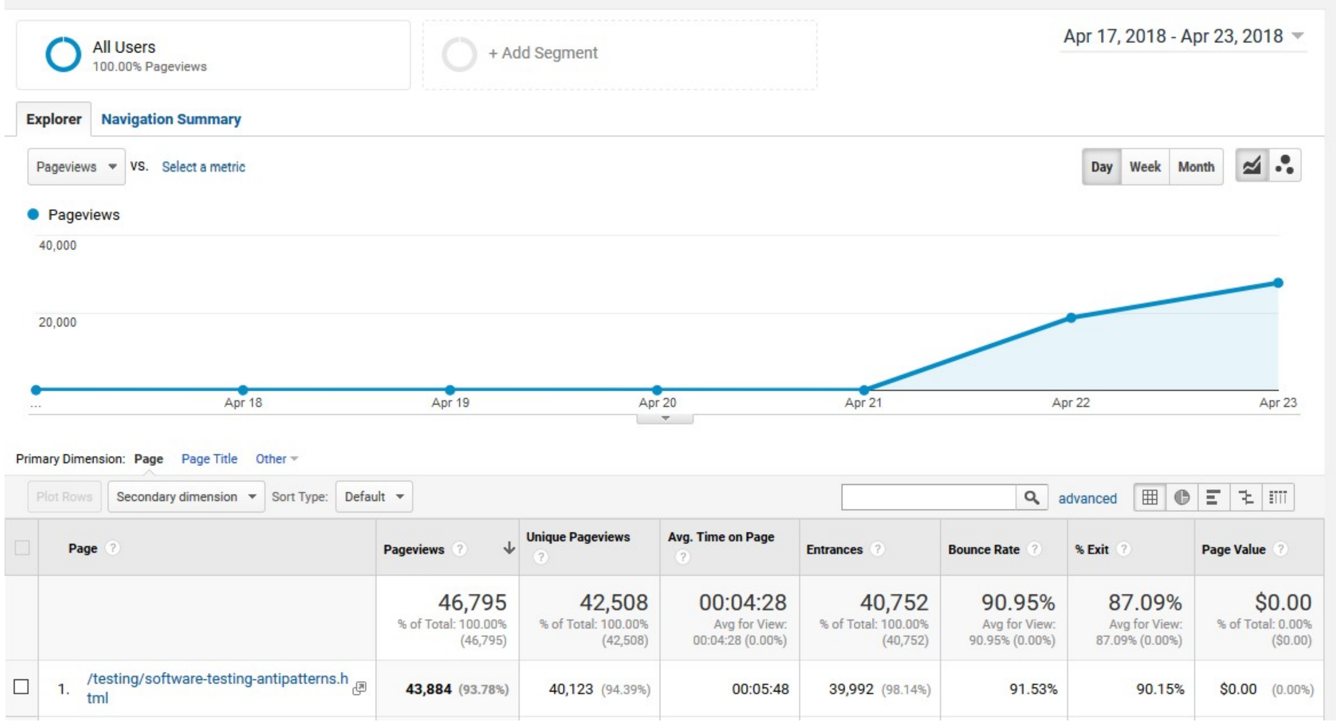
Things I blog

Software Testing Anti-Pattern List

1. Having unit tests without integration tests
2. Having integration tests without unit tests
3. Having the wrong kind of tests
4. Testing the wrong functionality
5. Testing internal implementation
6. Paying excessive attention to test coverage
7. Having flaky or slow tests
8. Running tests manually
9. Treating test code as a second class citizen
10. Not converting production bugs to tests
11. Treating TDD as a religion
12. Writing tests without reading documentation first
13. Giving testing a bad reputation out of ignorance

http://blog.codepipes.com/testing/software-testing-antipatterns.html

Things I blog

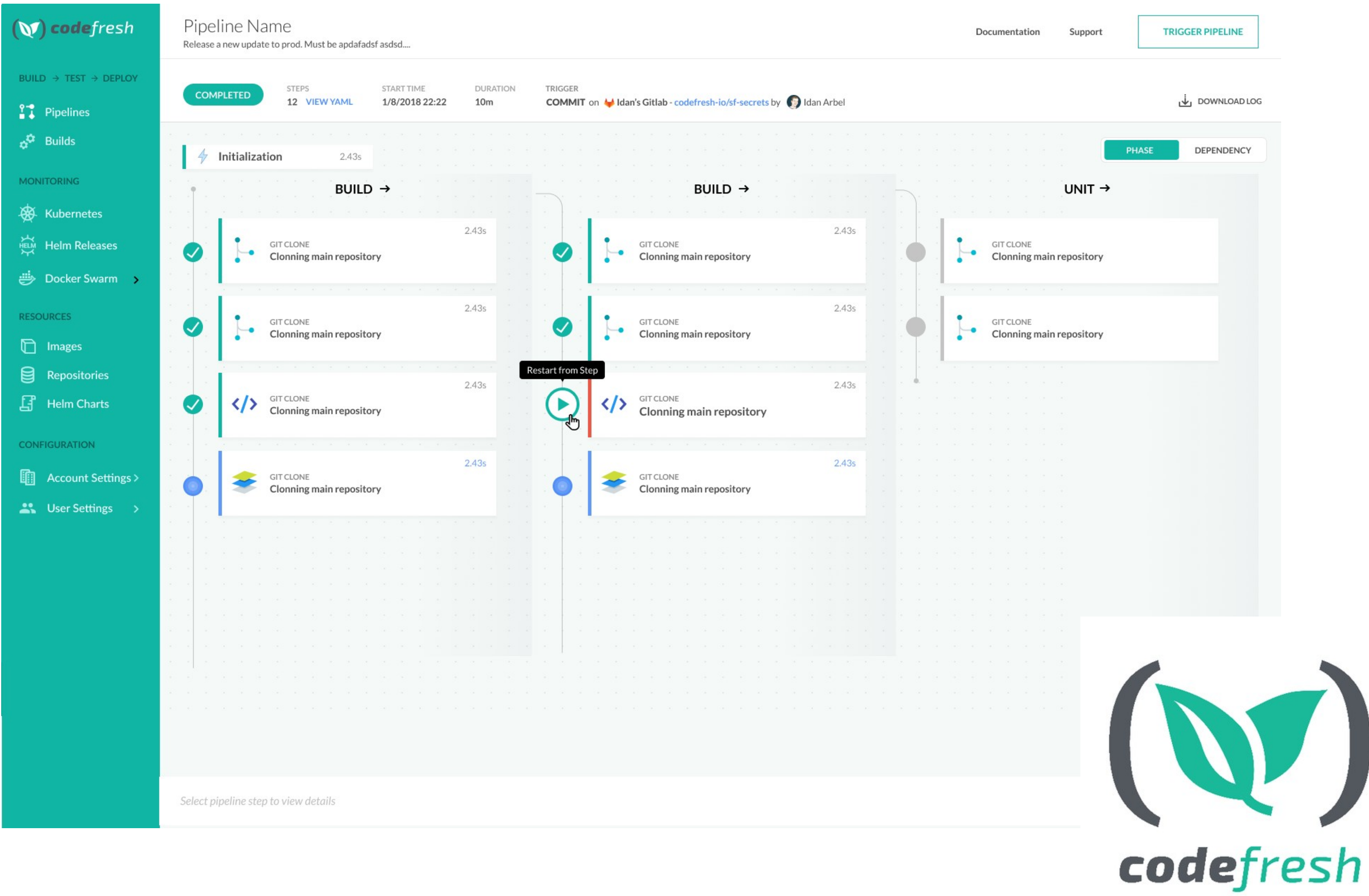


https://news.ycombinator.com/item?id=16894927

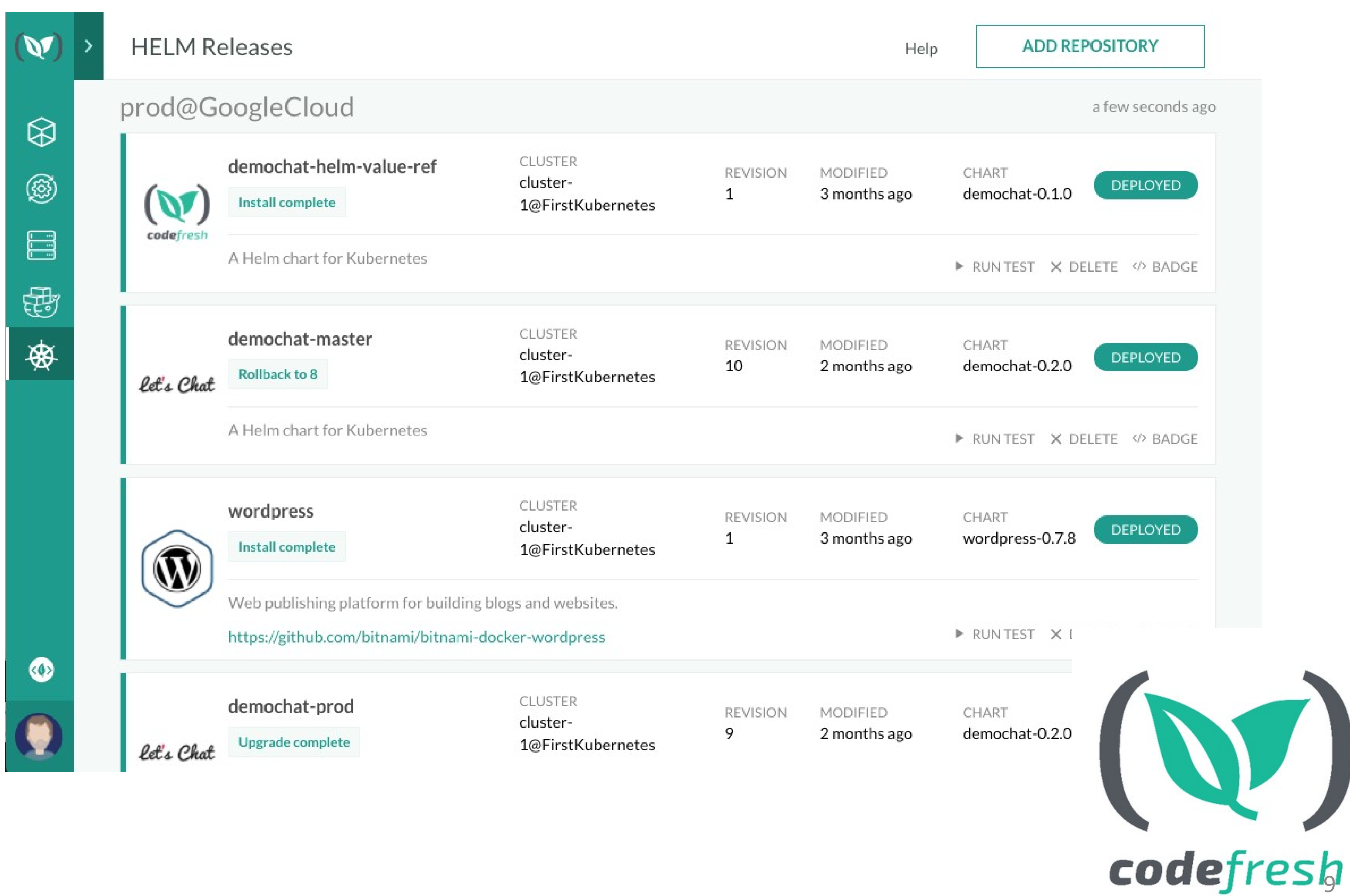
Current Work

Docker based CI/CD solution for Helm/ Kubernetes deployments

Current Work



Current Work



Current Work

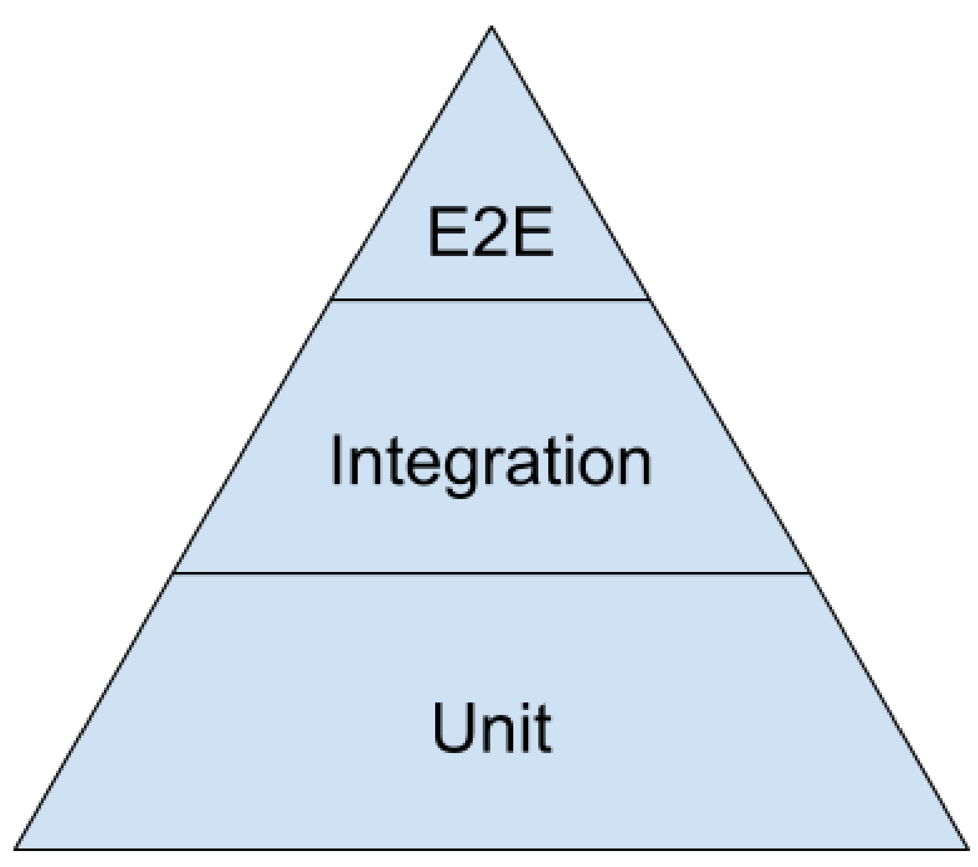
Docker Tutorial | June 20, 2018

Using Docker from Maven and Maven from Docker

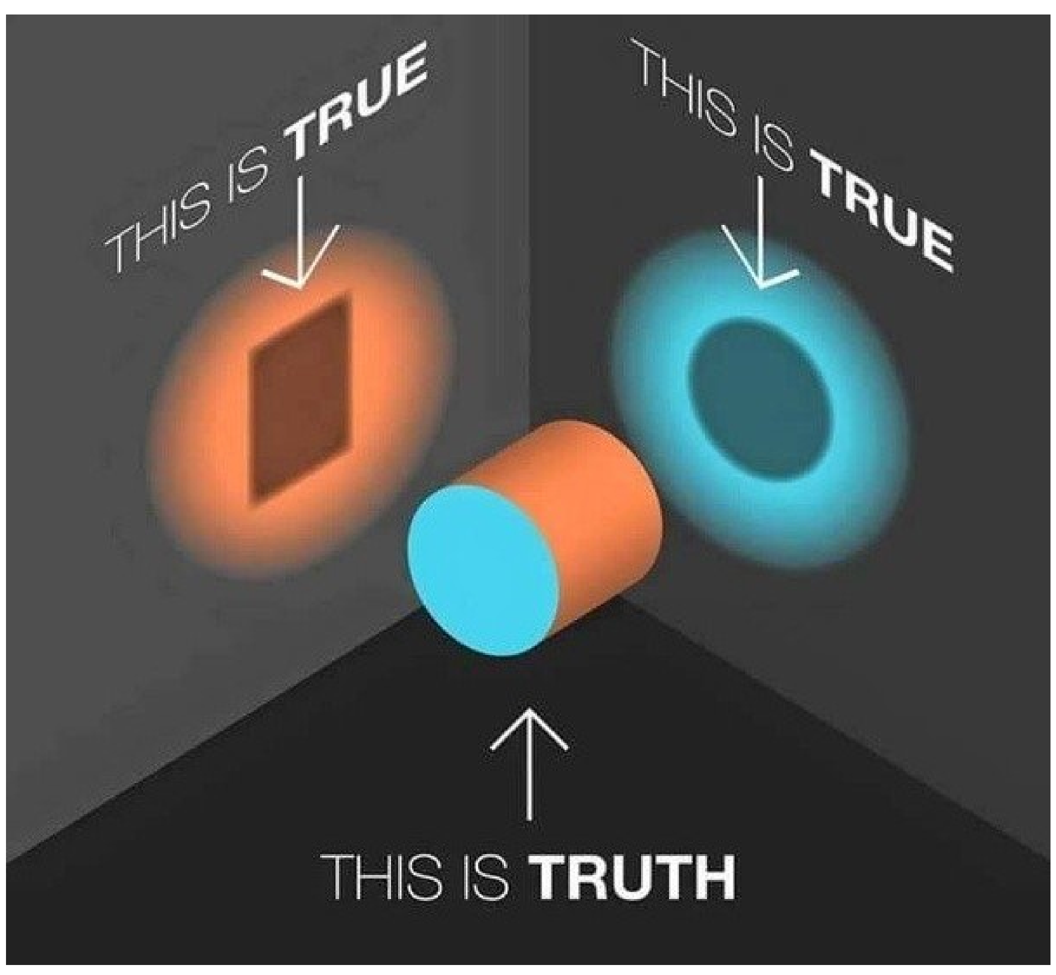
https://codefresh.io/blog/

https://codefresh.io/features/

Testing pyramid



Some definitions



Unit tests

- Require ONLY source code
- Everything that is external is mocked
- Mainly involve business logic testing
- Focus is on a single method/class
- Run with xUnit or similar framework
- Easy to setup and run
- Fast (20- 500ms)

Unit test example

```
Basket basket = new Basket()
basket.add("Samsung 4k TV", 600)
basket.add("Sony PS4", 300)


basket.getValue() == 900
```

Integration/Service/Component test

- Uses a database
- Uses the network to call another component
- Uses a queue/webservice
- Reads/writes files, performs I/O
- Needs the application to be deployed (even partially)
- Can be complex to setup and run
- Slow (seconds or even minutes)

Integration test example

```
Basket basket = new Basket(....)
Customer customer = new Customer(…)
customer.checkout(basket, cc, inventory)
```

Assert invoices, cc charge, inventory subtraction etc.

Maven lifecycle

`compile`	compile the source code of the project.
`process-classes`	post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.
`generate-test-sources`	generate any test source code for inclusion in compilation.
`process-test-sources`	process the test source code, for example to filter any values.
`generate-test-resources`	create resources for testing.
`process-test-resources`	copy and process the resources into the test destination directory.
`test-compile`	compile the test source code into the test destination directory
`process-test-classes`	post-process the generated files from test compilation, for example to do bytecode enhancement on Java classes. Fo
`test`	run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
`prepare-package`	perform any operations necessary to prepare a package before the actual packaging. This often results in an unpack 2.1 and above)
`package`	take the compiled code and package it in its distributable format, such as a JAR.
`pre-integration-test`	perform actions required before integration tests are executed. This may involve things such as setting up the require
`integration-test`	process and deploy the package if necessary into an environment where integration tests can be run.
`post-integration-test`	perform actions required after integration tests have been executed. This may including cleaning up the environment.
`verify`	run any checks to verify the package is valid and meets quality criteria.

Antipattern 1 – Only unit tests

Antipattern 1 – Only unit tests

- Usually in small companies
- Developers who have never seen integration tests
- Integration tests were abandoned
- Test Environment is “hard” to setup

We need integration tests

Type of issue	Detected by Unit tests	Detected by Integration tests
Basic business logic	yes	yes
Component integration problems	no	yes
Transactions	no	yes
Database triggers/procedures	no	yes
Wrong Contracts with other modules/APIs	no	yes
Wrong Contracts with other systems	no	yes
Performance/Timeouts	no	yes
Deadlocks/Livelocks	maybe	yes
Cross-cutting Security Concerns	no	yes

Antipattern 1 – Solution

- Dockerize your application
- Launch containers after every Pull Request
- Test features BEFORE merging to master
- Anybody should be able to launch all or part of the application with a single command

Antipattern 2 – Only integration tests

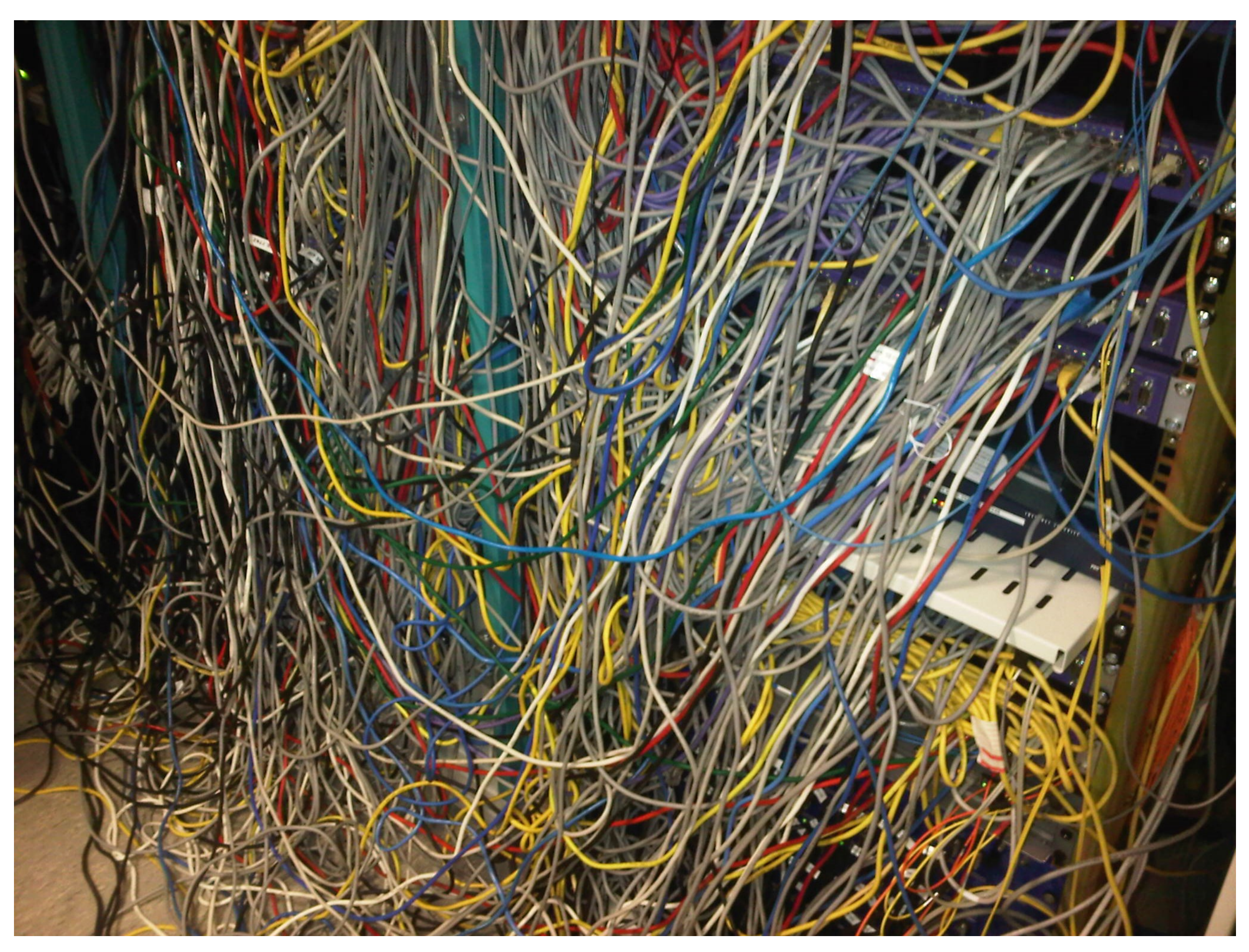
Antipattern 2 – Only integration tests

- Usually found in big companies
- "Unit tests are a waste of time"
- People were forced to write unit tests for code coverage requirements
- "Unit tests are useless, they never fail"
- "Value comes only from integration tests"

We need unit tests

1. Integration tests are complex
2. Integration tests are slow
3. Integration test are hard to setup and debug

Integration tests are complex



Let's test a service

```
public class UserService {

        public User findUser(Long id) {
                ...
        }

        public String serializeResponse(Payload payload) {
                ...
        }

        public void checkoutBasket(Basket basket) {
                ...
        }

        public boolean checkInventory(Basket basket) {
                ...

        }
}
~
~
~
```

method A

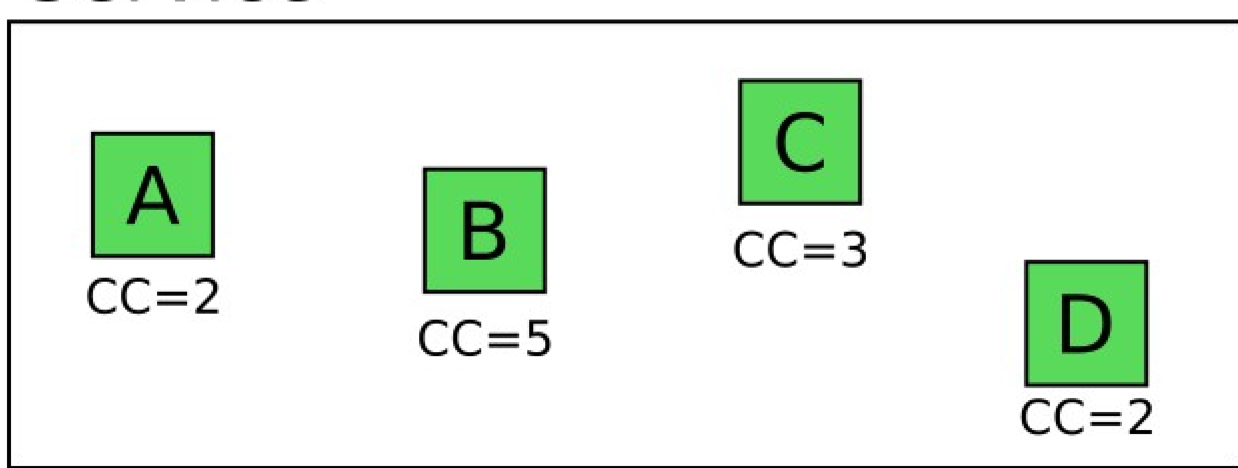
method B

method C

method D

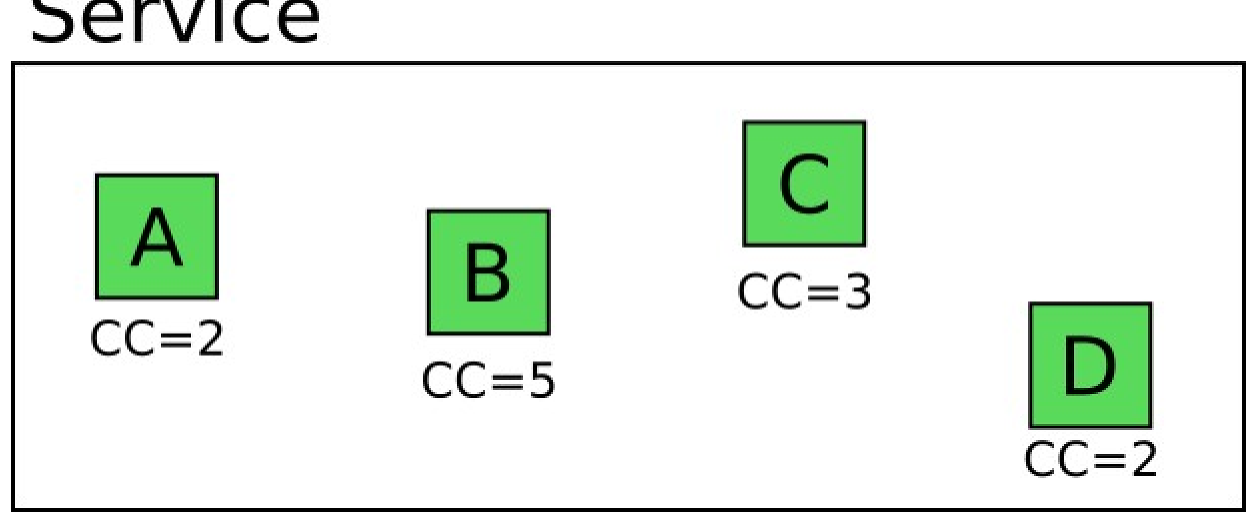
Let’s test a service

- A Java service
- 4 methods
- CC = cyclomatic complexity (number of code paths)



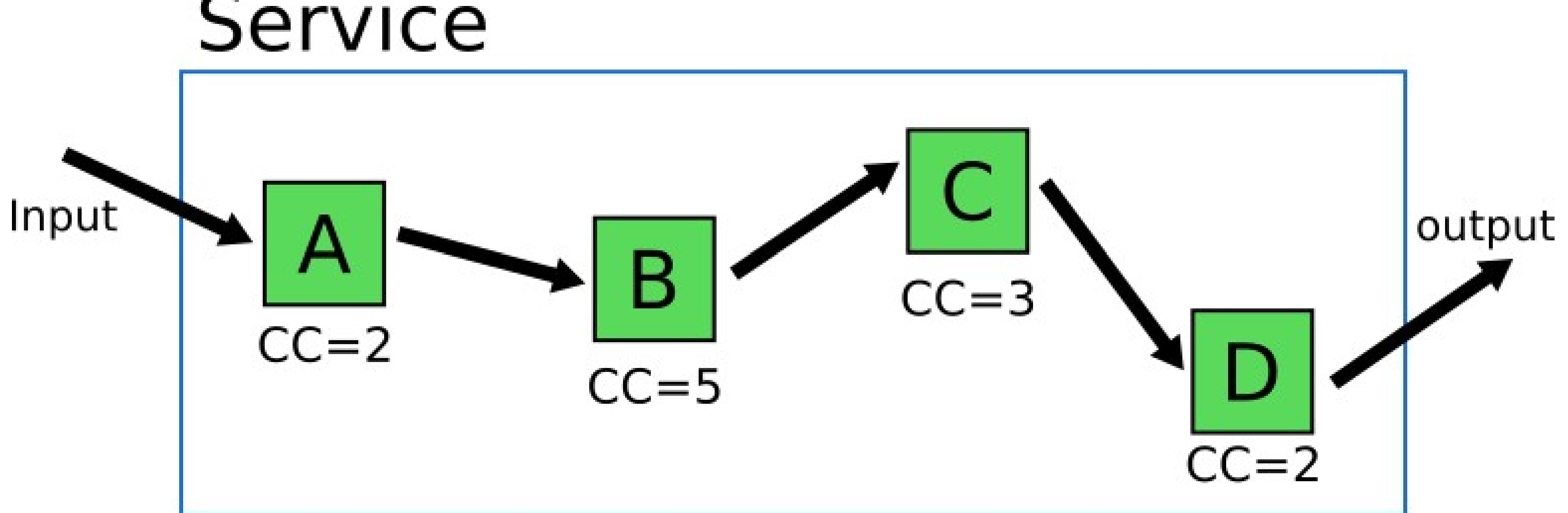
Only unit tests

- Can write 2 + 5 + 3 + 2 = 12 unit tests
- Get 100% of business logic
- The full application has other more services



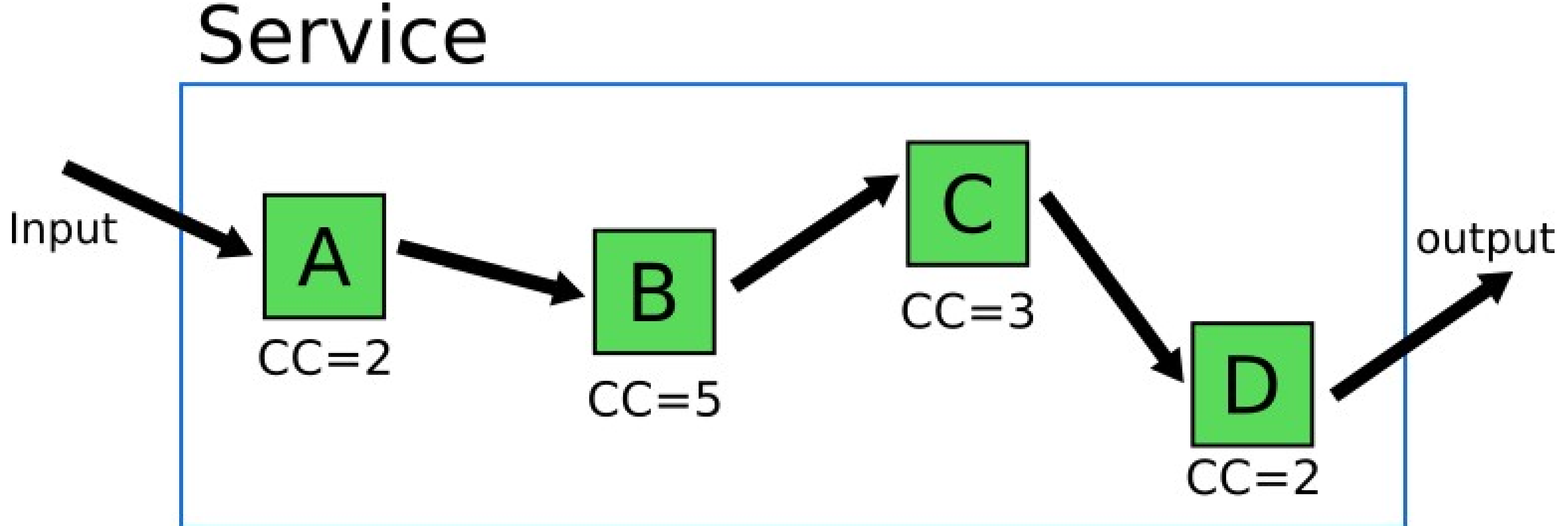
Only integration tests

- Should write 2 * 5 * 3 * 2 = 60 tests
- People cheat and only choose some "representative tests"
- Usually happy path scenarios



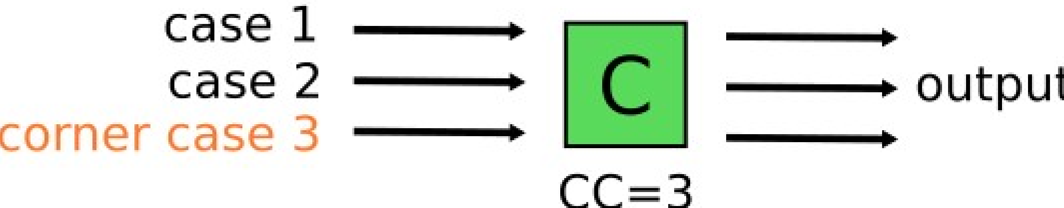
Hard to test corner cases

- A special scenario in C requires….
- A special scenario in B that requires…
- A special scenario in A



Easy to test corner cases

- With unit tests only a single method is focused
- Corner case can be created on the spot
- Very easy to test



Integration tests are slow

Integration tests are slow

- Two developers Mary and Joe
- Joe writes only integration tests
- Mary writes unit tests PLUS some integration tests

Test assumptions

- Each unit test takes 60ms (on average)
- Each integration test takes 800ms (on average)
- The application has 40 services like the one shown in the previous section
- Mary is writing 10 unit tests and 2 integration tests for each service
- Joe is writing 12 integration tests for each service

Speed comparison

- Joe waits 6 minutes after a commit
- Mary waits 1 minute

Time to run	Having only integration tests (Joe)	Having both Unit and Integration tests (Mary)
Just Unit tests	N/A	24 seconds
Just Integration tests	6.4 minutes	64 seconds
All tests	6.4 minutes	1.4 minutes

Integration tests are hard to debug

E-Shop application

- You write tests for the typical eshop applications
- Customers buy products
- Discounts on prices
- Warehouse inventory
- Credit card processing

Result from integration tests

Integration tests result

Customer registers an account

Customer buys a single item

Customer adds 2nd shipping address

Customer checks order history

Result from all tests

Integration tests result

- Customer registers an account
- Customer buys a single item
- Customer adds 2nd shipping address
- Customer checks order history

Unit tests result

- Basket Weight Test
- Special Discount Test
- Product Recommendation Test
- Credit card validation test
- Promo Code Test
- Wish List Test
- VAT Calculation Test

Anti-pattern 2 - summary

1. Integration tests are complex
2. Integration tests are slow
3. Integration test are hard to setup and debug

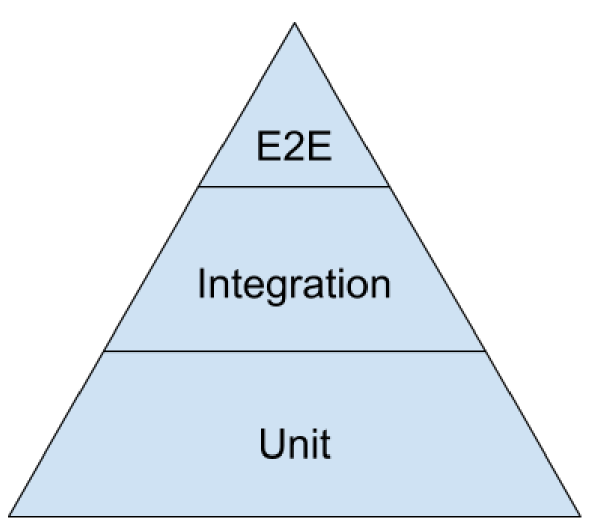
Corollary

- We need both unit and integration tests
- Having only one type is an anti-pattern

Antipattern 3 – Wrong kinds of tests

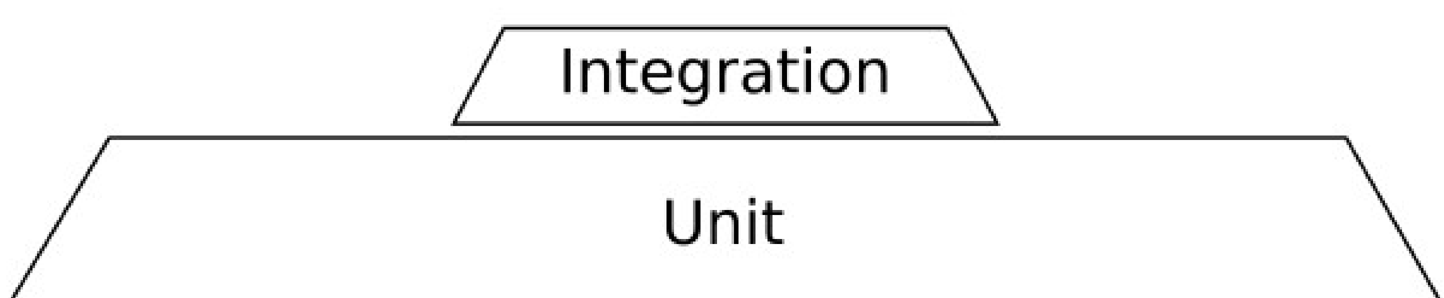
Amount of tests for each type

- Test pyramid is only a suggestion
- You need to decide what your application is doing
- Different applications have different needs



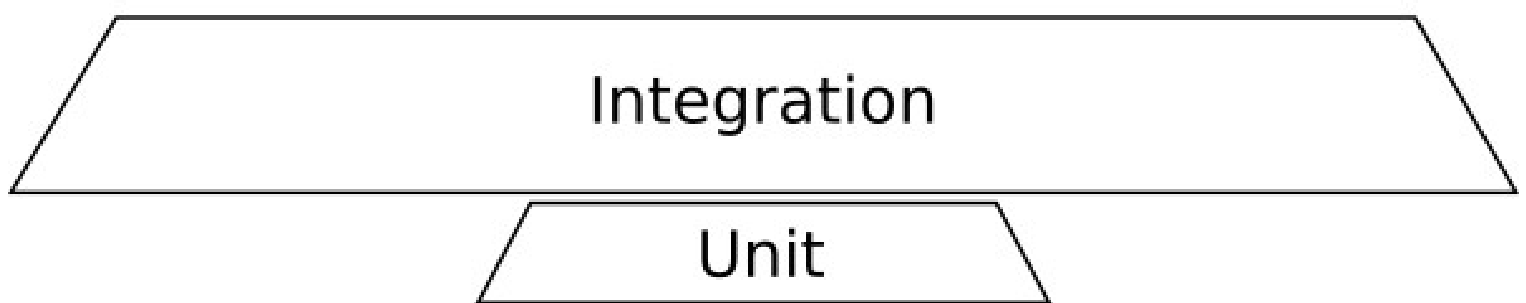
Example 1

Command Line Utility

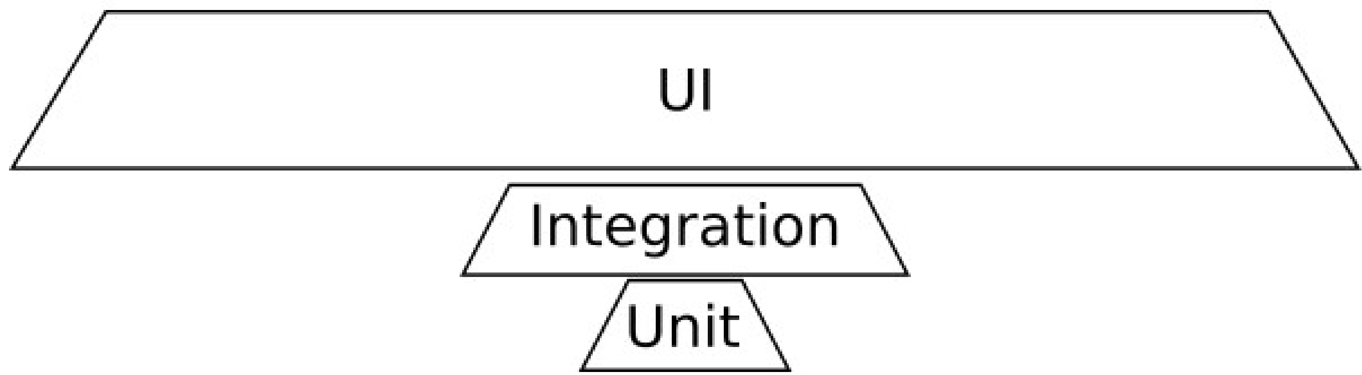


Example 2

Payment Gateway

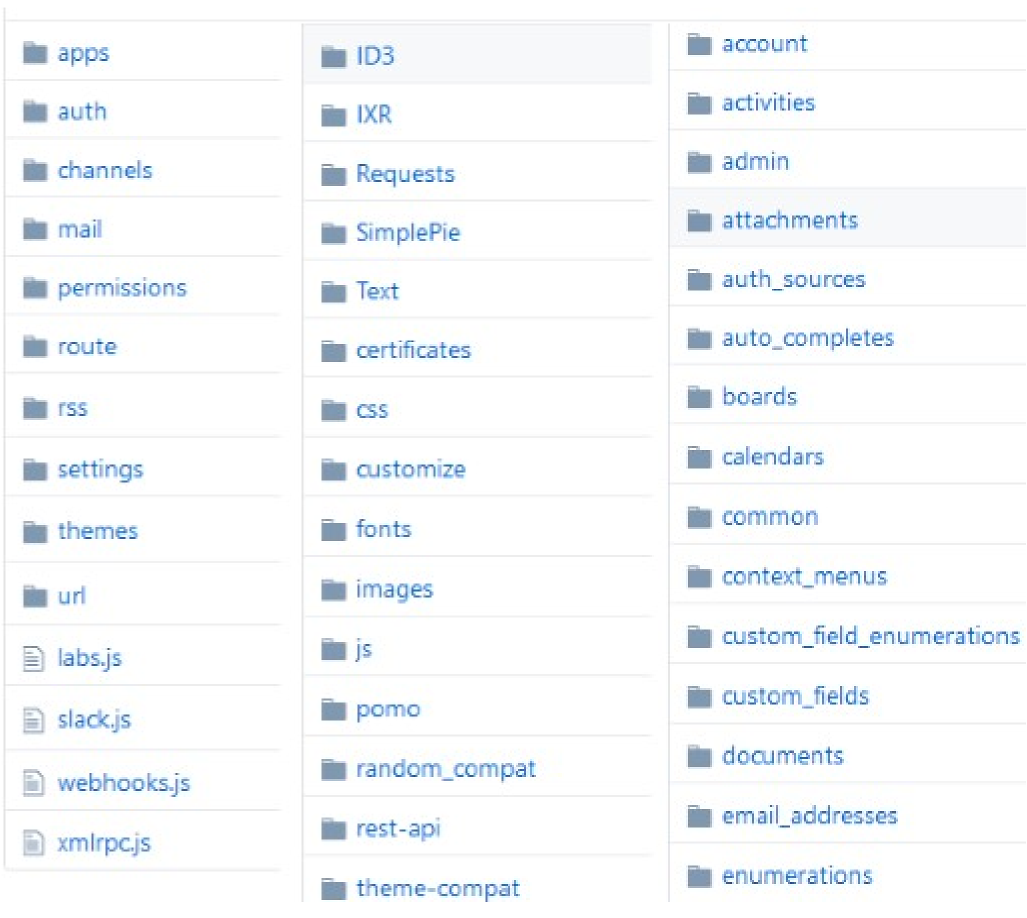


Example 3



Antipattern 4 – Testing the wrong functionality

Code != file folders



Deployment time

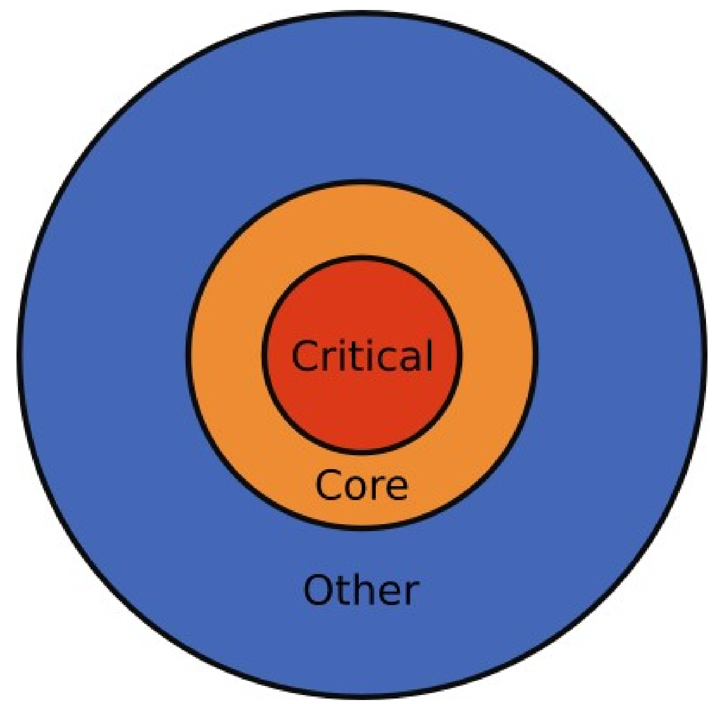
Two bugs after deployment

1. Customers cannot check-out their cart halting all sales
2. Customers get wrong recommendations when they browse products.

Obviously first one is critical, second one is not

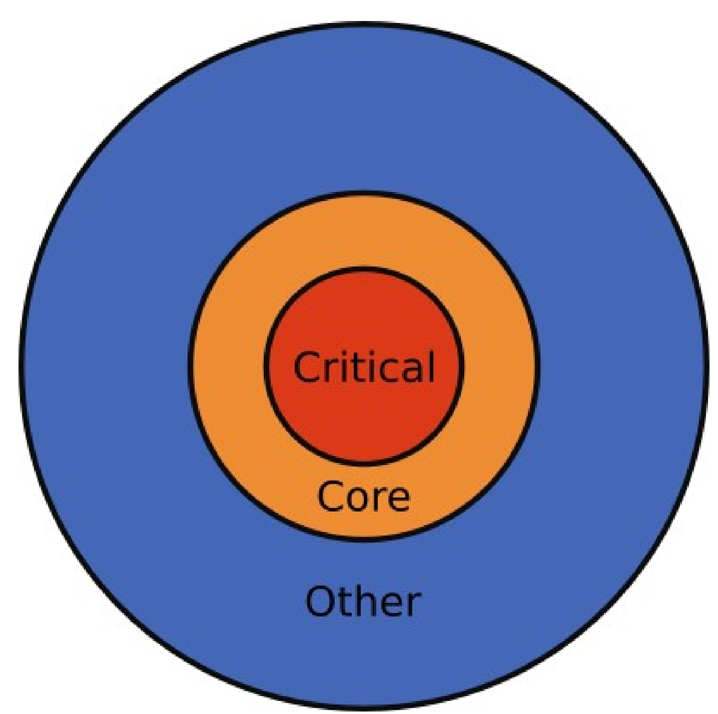
Code severity

Critical code - This is the code that breaks often, gets most of new features and has a big impact on application users



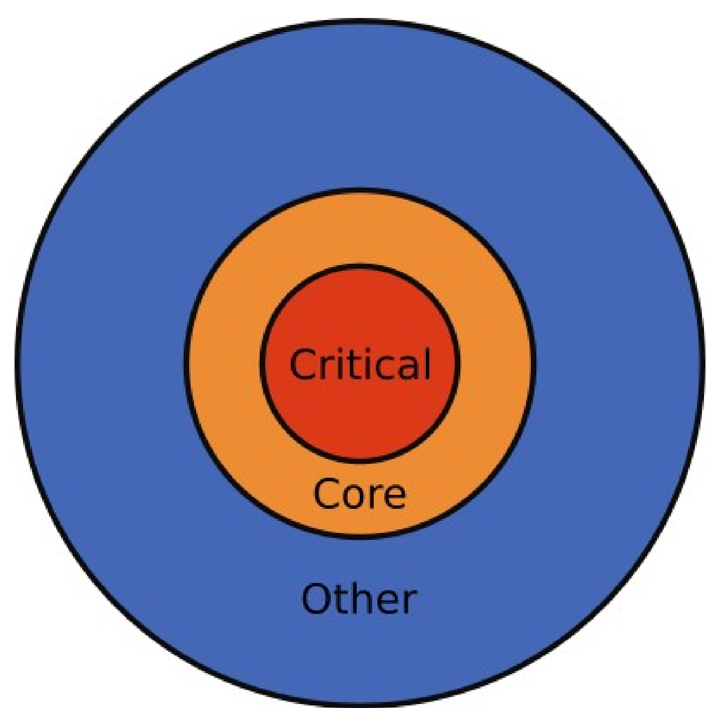
Code severity

Core code - This is the code that breaks sometimes, gets few new features and has medium impact on the application users



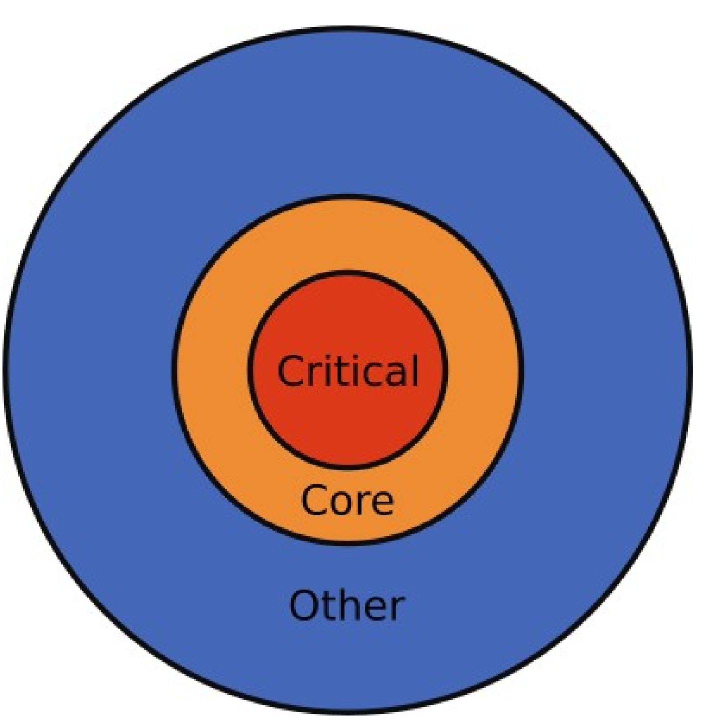
Code severity

Other code - This is code that rarely changes, rarely gets new features and has minimal impact on application users.



Write tests for code that

- Breaks often
- Changes often
- Is critical to the business



Antipattern 5 – Testing internal implementation

Antipattern 5 – Testing internal implementation

- Worse kind of tests
- Wasted time the first time they are written
- Wasted time when a new feature is added
- They give a bad name to unit testing
- Closely connected to antipattern 2 (no unit tests)
- Mostly relevant for unit tests

Rules of unit testing

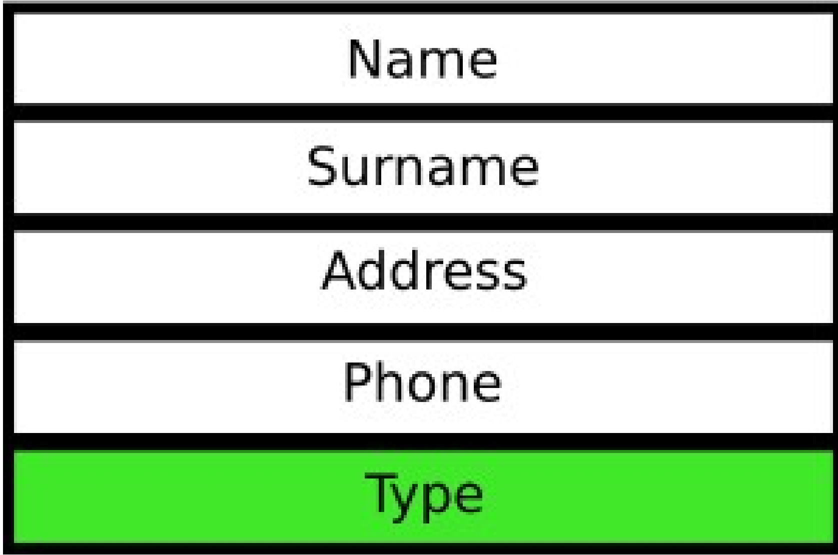
1. Test behavior and not state

2. Test behavior and not state

3. If this is your first unit test

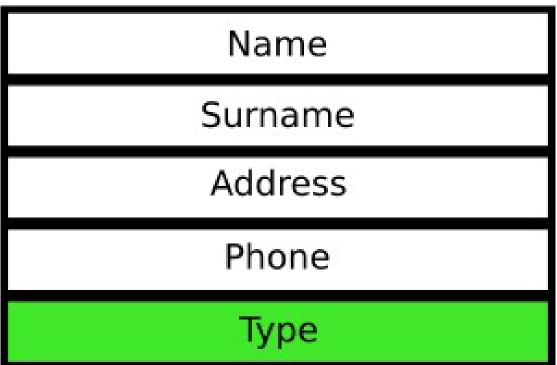
…test behavior and not state!

Testing state – bad Example



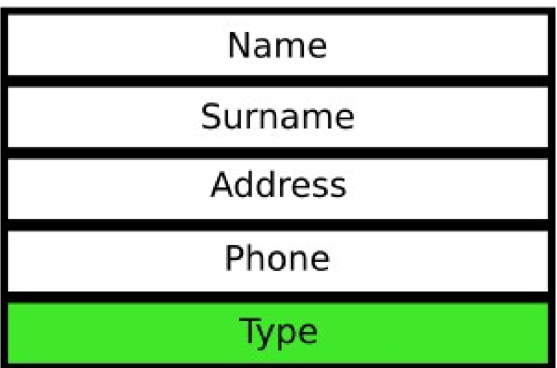
Testing state – bad Example

- Customer type 0 means "guest" and 1 means "registered user"
- 10 unit tests are written that verify this particular field



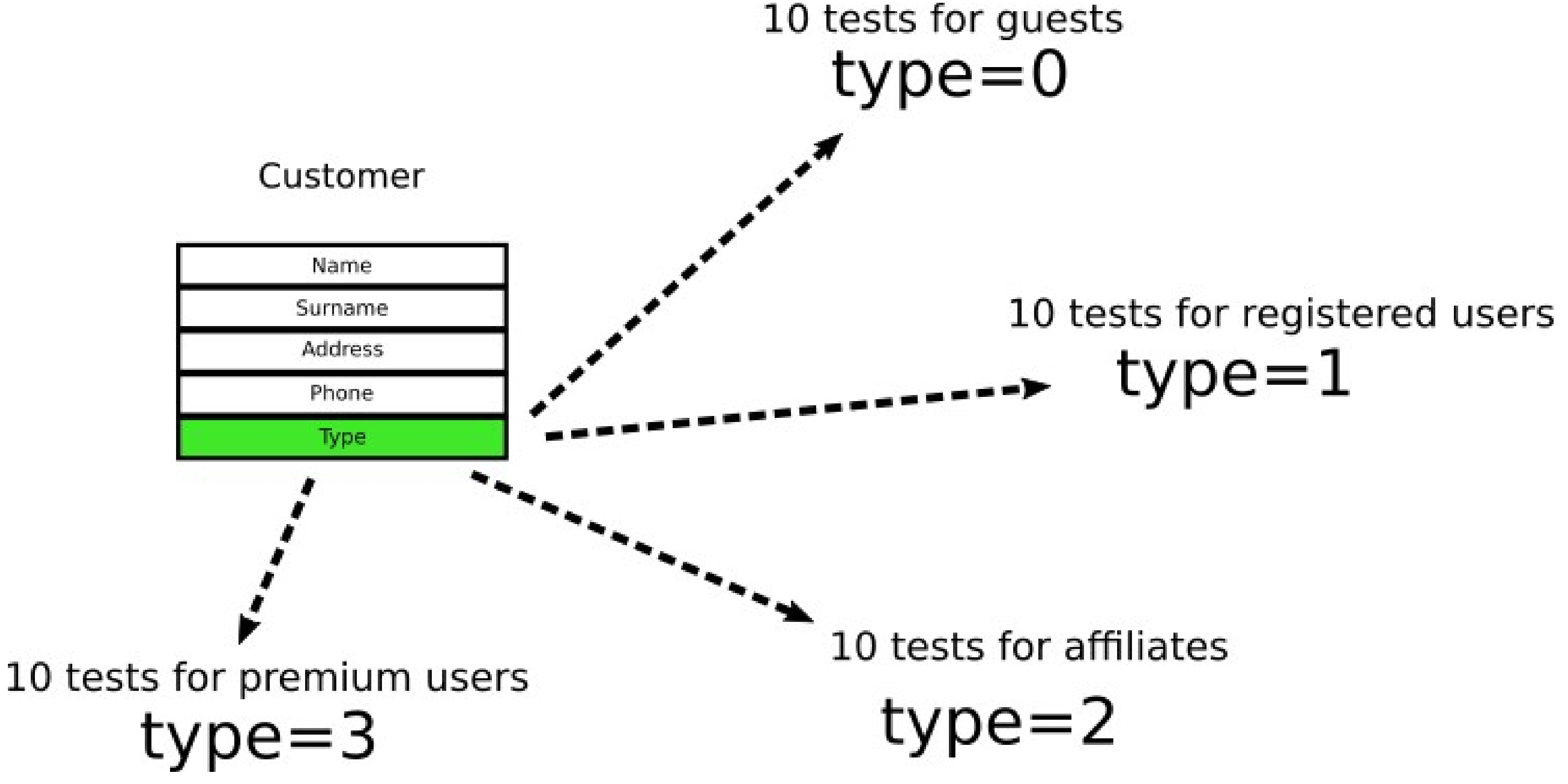
Testing state – bad Example

- Customer type 2 means “affiliate” and 3 means “premium user”
- 20 more unit tests are written that verify this particular field



Testing state – bad Example

40 tests in total, all looking at this field

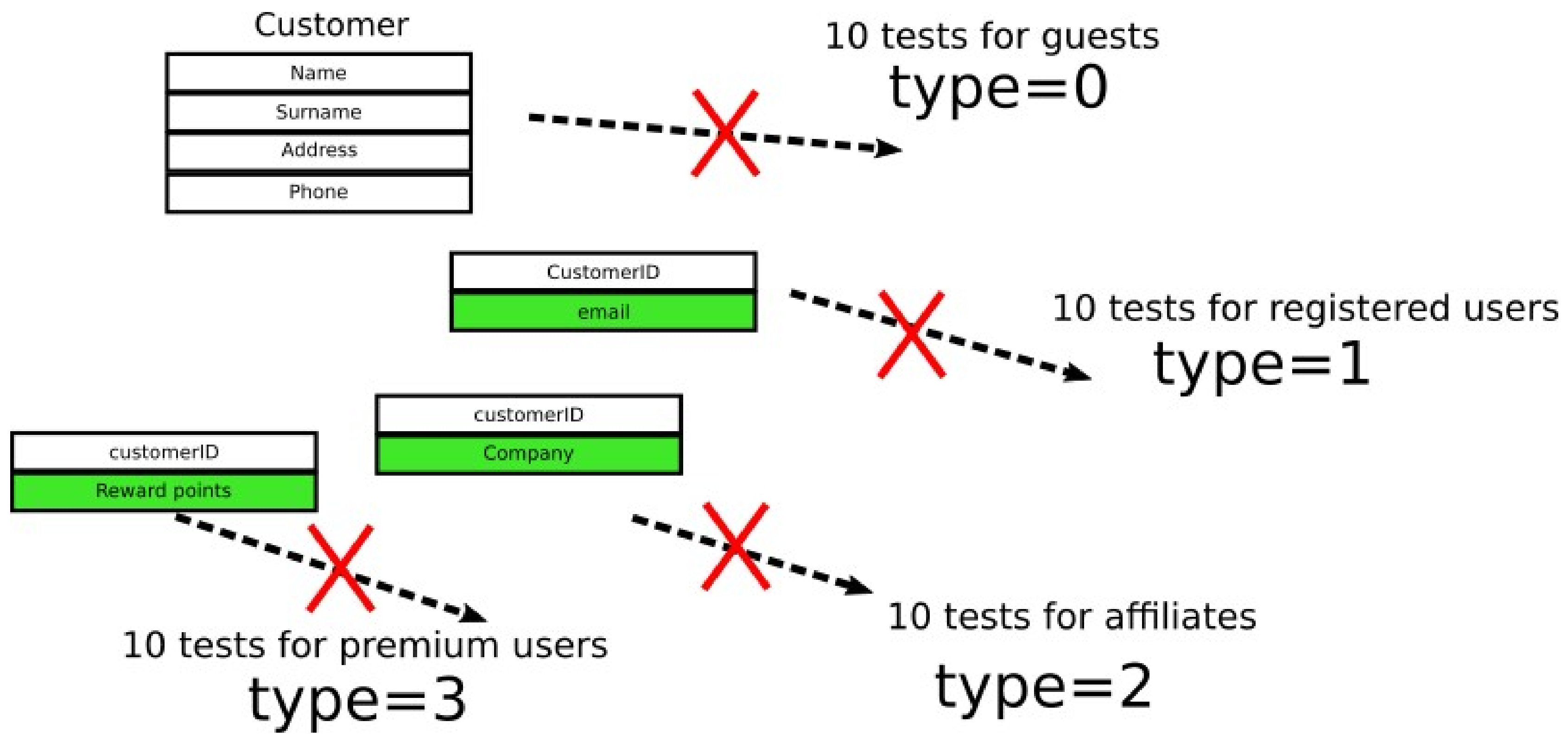


New feature from customers

New feature from customers

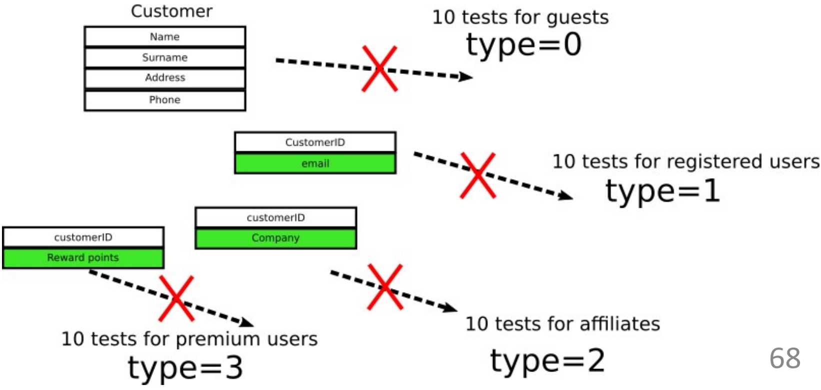
1. For registered users, their email should also be stored
2. For affiliate users, their company should also be stored
3. Premium users can now gather reward points.

40 tests are now broken

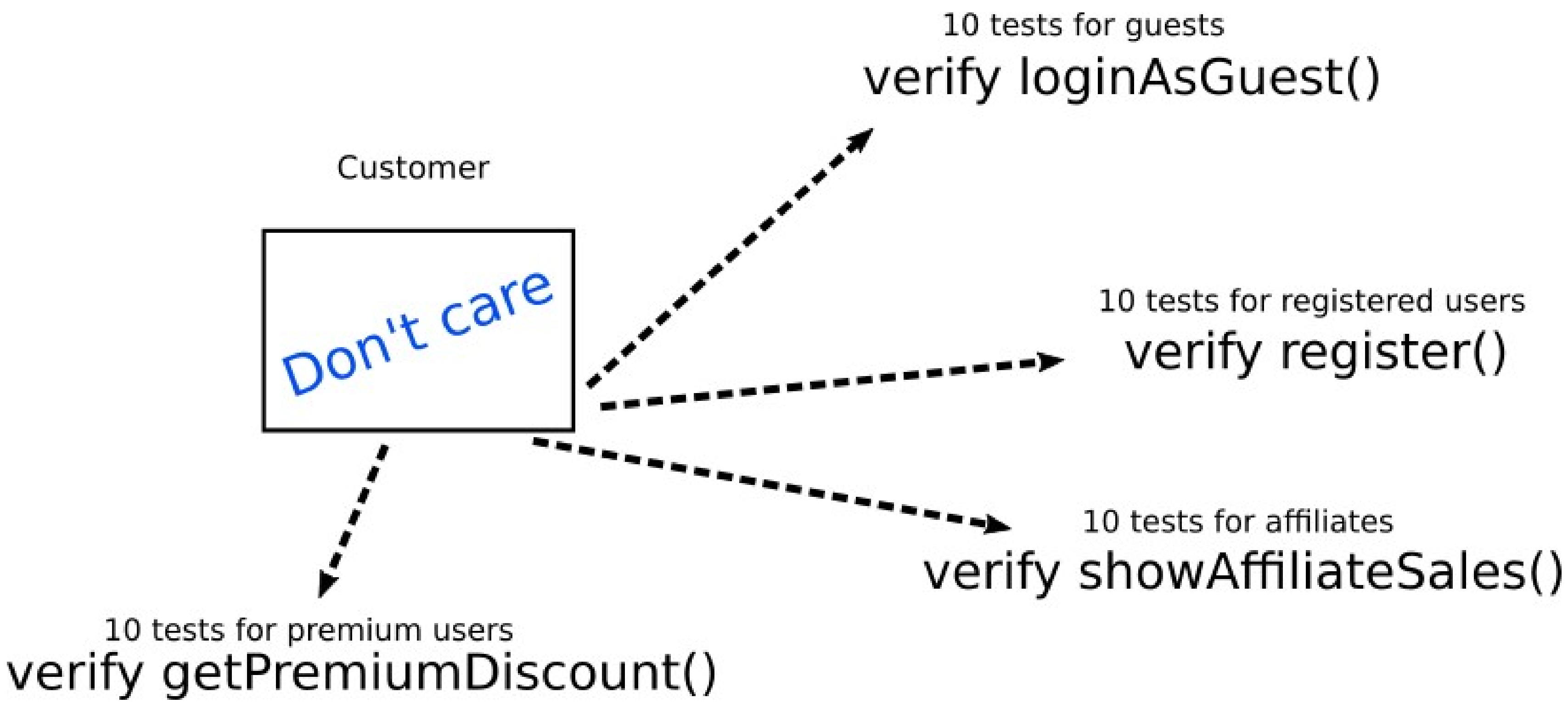


40 tests are now broken

- This is why some people hate unit tests
- “I try to implement a feature and all tests are broken”
- “I spend more time with tests than actual code”
- Damage is already done

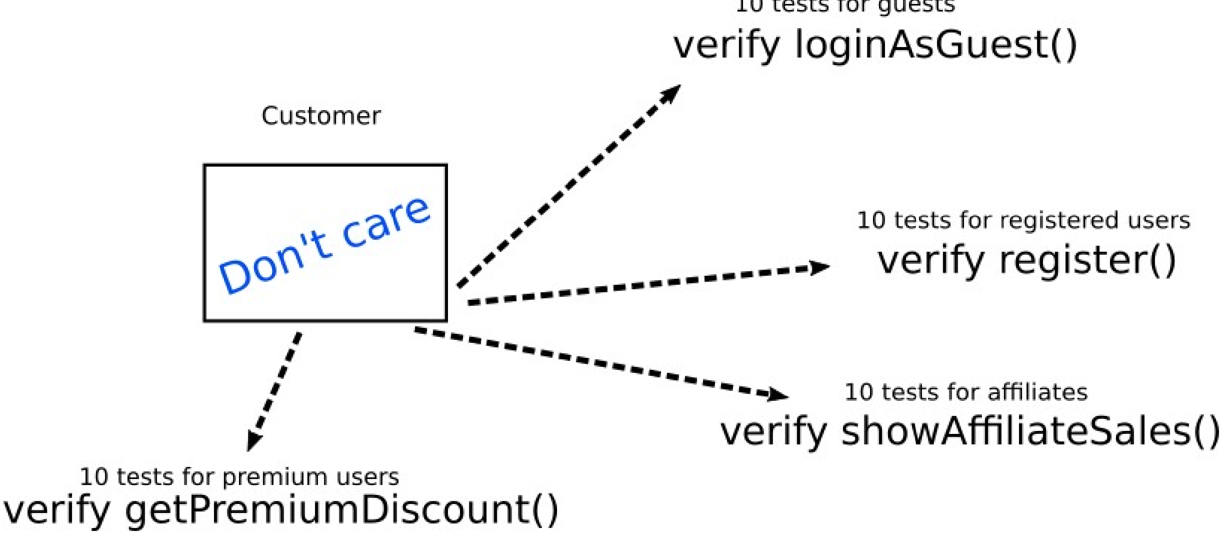


Testing behavior instead of state



Testing behavior instead of state

- Business needs do not affect tests
- At most 10 tests will break (not all of them)
- New fields can be added/removed in customer object



Good names of Test Methods

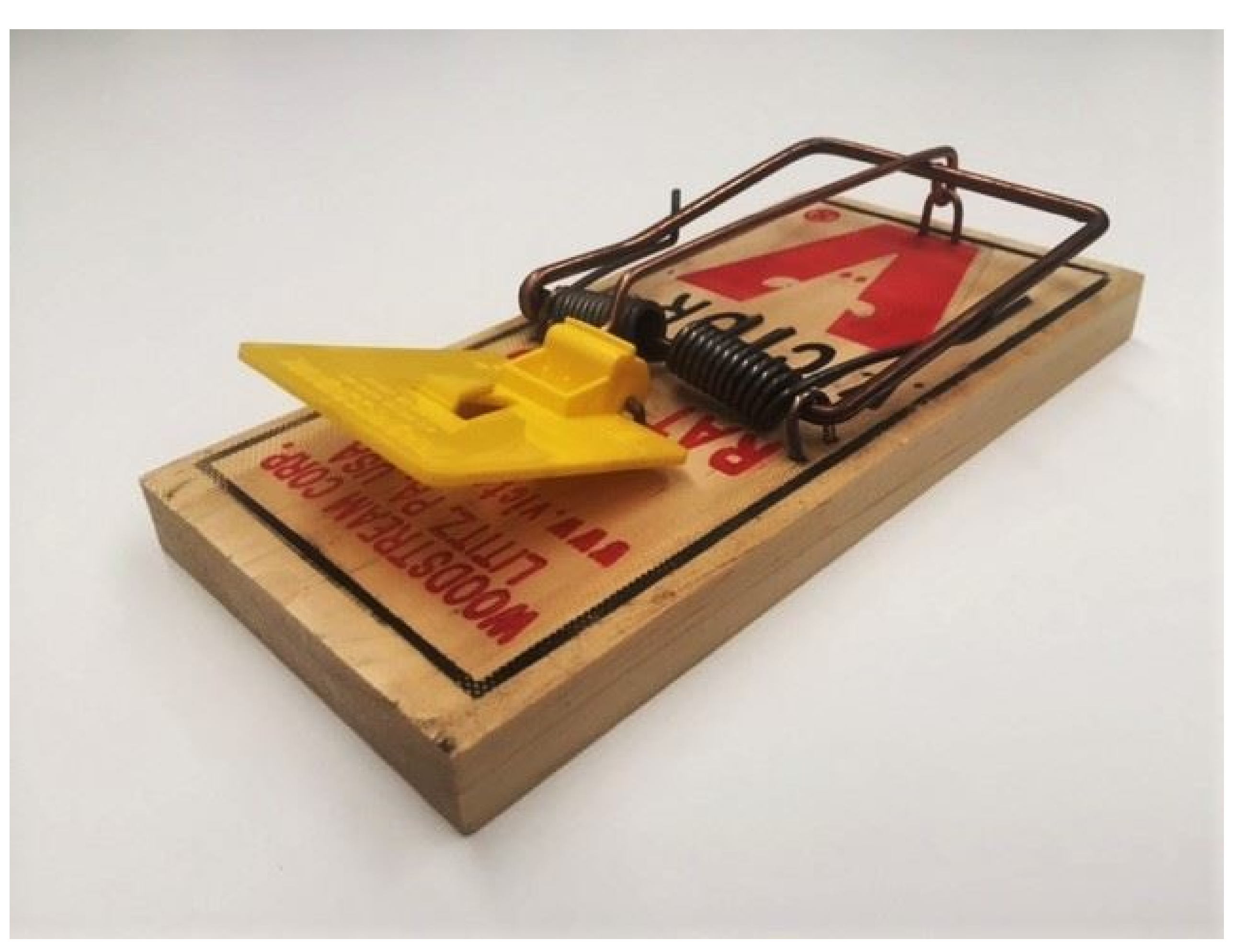
- guestUsersDoNotGetRewardPoints()
- premiumCustomerShouldGetDiscount()
- maximum100AwardPointPerBasket()
- customerWithoutEmailDoesNotGetNewsletter()
- digitalGoodsNeverHaveWeight()
- orderHistoryAlwaysIncludesShippingAddress()

Bad names of Test Methods

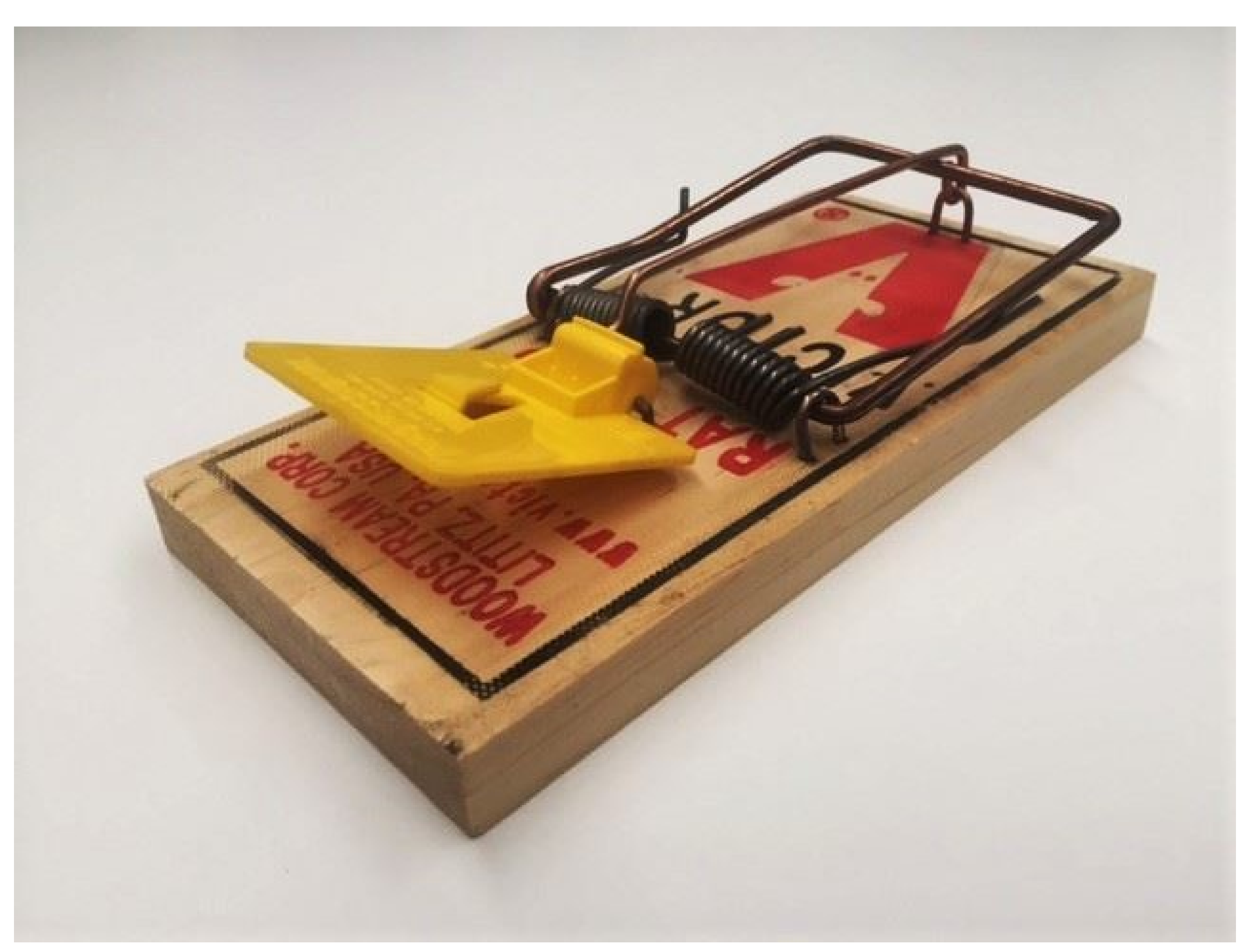
- checkInvoiceType()
- billingTest1()
- billingTest2()
- testJIRA2345()
- customerObjectHasCorrectFieldsTest()
- productsTypeCshouldBelongToCategoryA()

Antipattern 6 – Paying too much attention to code coverage

Code coverage is a trap



How much code coverage is enough?



Code coverage everywhere

- It is easy to understand
- It is easy to measure
- There are many tools for measuring it
- Also familiar to other project stakeholders
- Beloved by QA departments and managers

I will tell you a secret

I will tell you a secret

A project can be full of bugs and still have 100% code coverage

Sample application

```java
1 package gr.jhug.sample;
2
3 public class MyCalculation {
4
5     public int velocity(int angle, int direction ) {
6         return ((3 * (4* angle - direction))* 3) / (7 * (direction - (2 * angle)));
7     }
8 }
9
```

100% Coverage

```
14
15
16 public class MyCalculationTest {
17
18     @Test
19     public void simpleEntry() throws IOException
20     {
21
22
23         MyCalculation myCalc = new MyCalculation();
24
25         assertEquals("Expected an entry",-5,myCalc.velocity(3, 4));
26         assertEquals("Expected an entry",-2,myCalc.velocity(10, 2));
27         assertEquals("Expected an entry",-2,myCalc.velocity(8, 3));
28         assertEquals("Expected an entry",-3,myCalc.velocity(6, 6));
29         assertEquals("Expected an entry",-3,myCalc.velocity(5, 3));
30
31     }
32
33 }
34
```

Problems | Javadoc | Declaration | Console | Progress | Coverage

MyCalculationTest (Apr 9, 2019 11:50:28 AM)

Element	Coverage	Covered Instructi	Missed Instructio
sample-calculation	100.0 %	69	0
src/main/java	100.0 %	21	0
src/test/java	100.0 %	48	0

Bug if direction is double the angle

```
1 package gr.jhug.sample;
2
3 public class MyCalculation {
4
5     public int velocity(int angle, int direction ) {
6         return ((3 * (4* angle - direction))* 3) / (7 * (direction - (2 * angle)));
7     }
8 }
9
```

```
Failure Trace
java.lang.ArithmeticException: / by zero
at gr.jhug.sample.MyCalculation.velocity(MyCalculation.java:6)
at gr.jhug.sample.MyCalculationTest.simpleEntry(MyCalculationTest.java:30)
```

I will tell you a secret

Do not try to achieve a specific number (such as 100%)

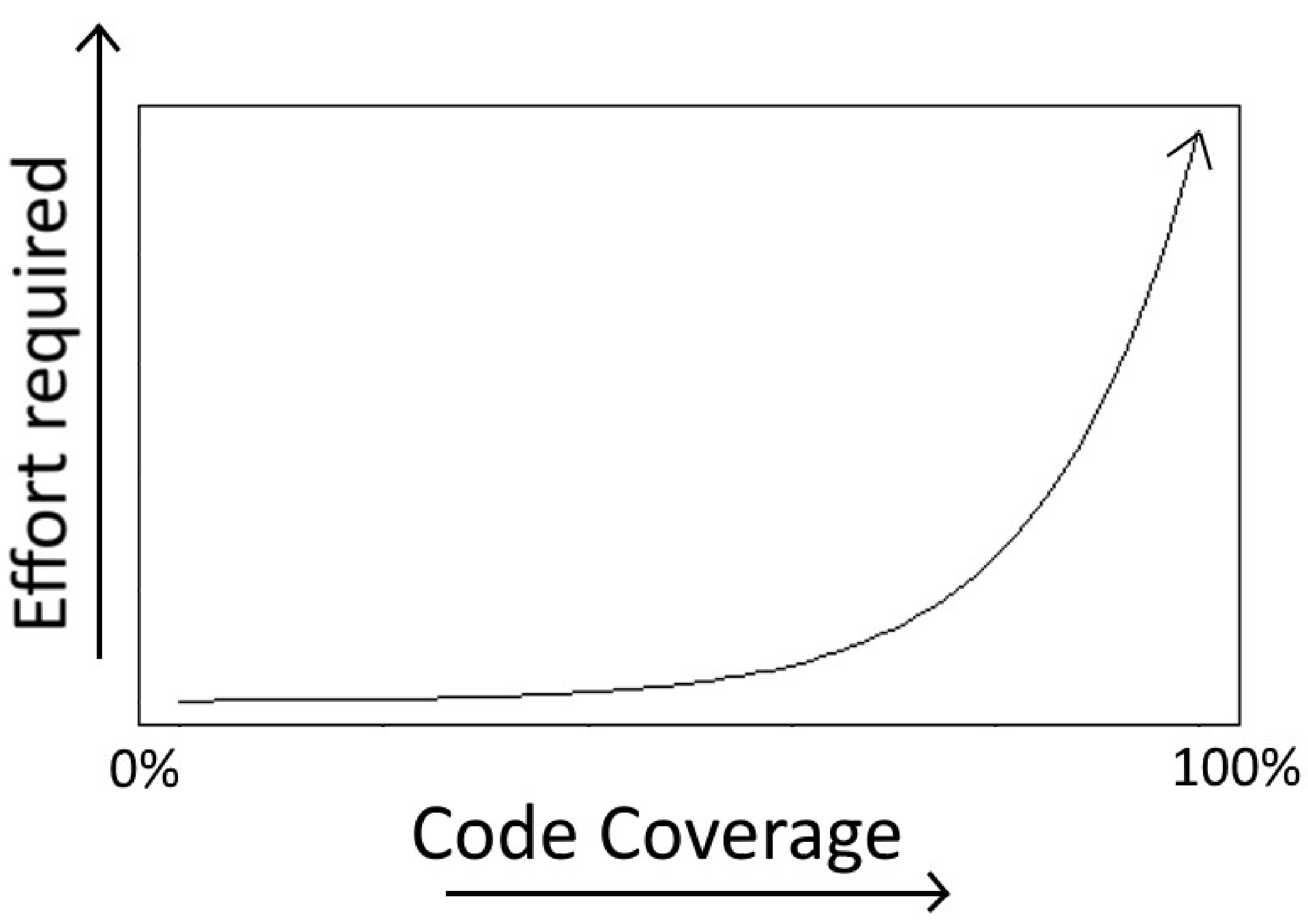
I will tell you a secret

Bigger numbers require more effort (logarithmic?)

I will tell you a secret

Getting from 80% to 100% is much more difficult than 0% to 20%

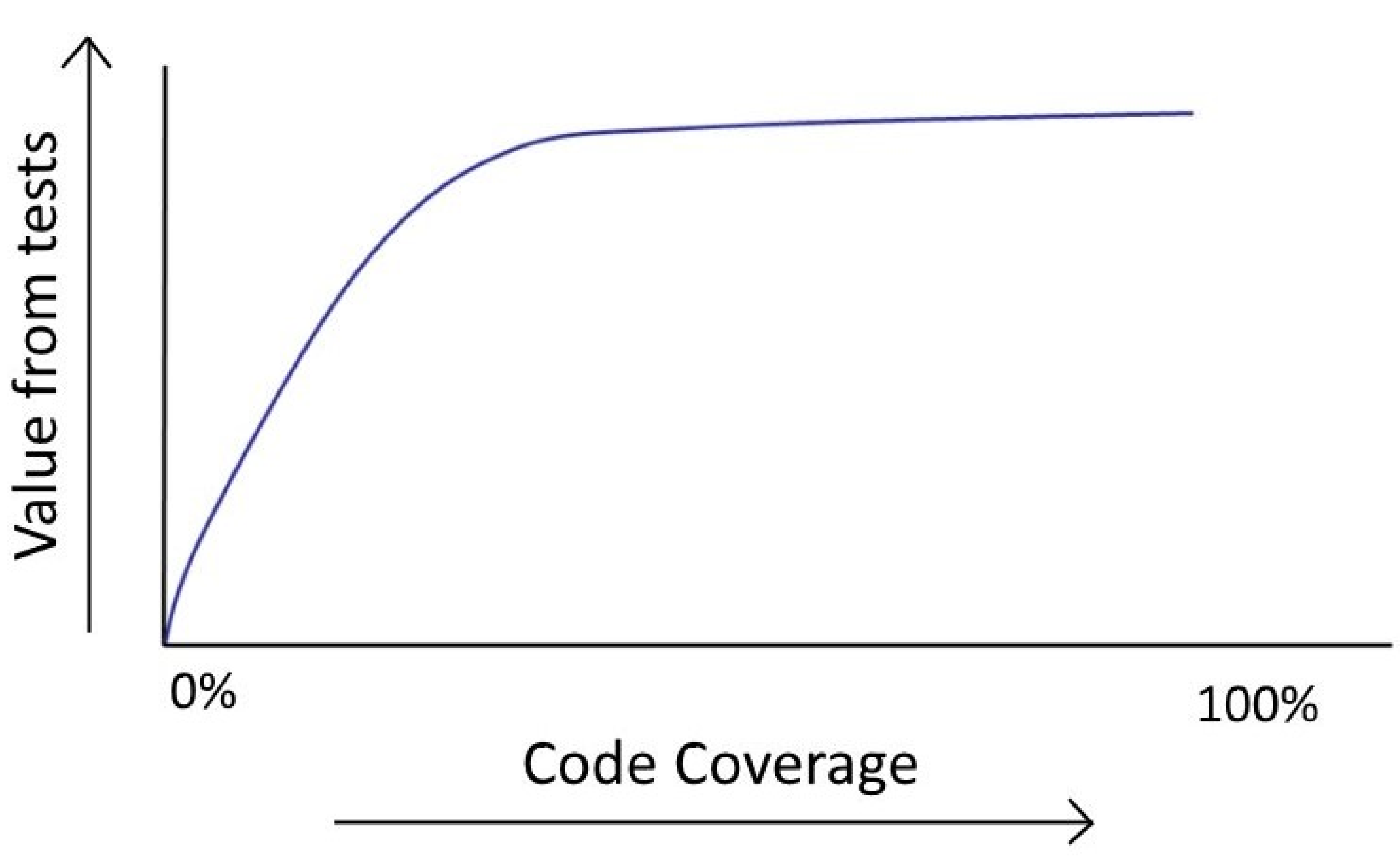
I will tell you a secret



I will tell you a secret

Increasing code coverage has diminishing returns

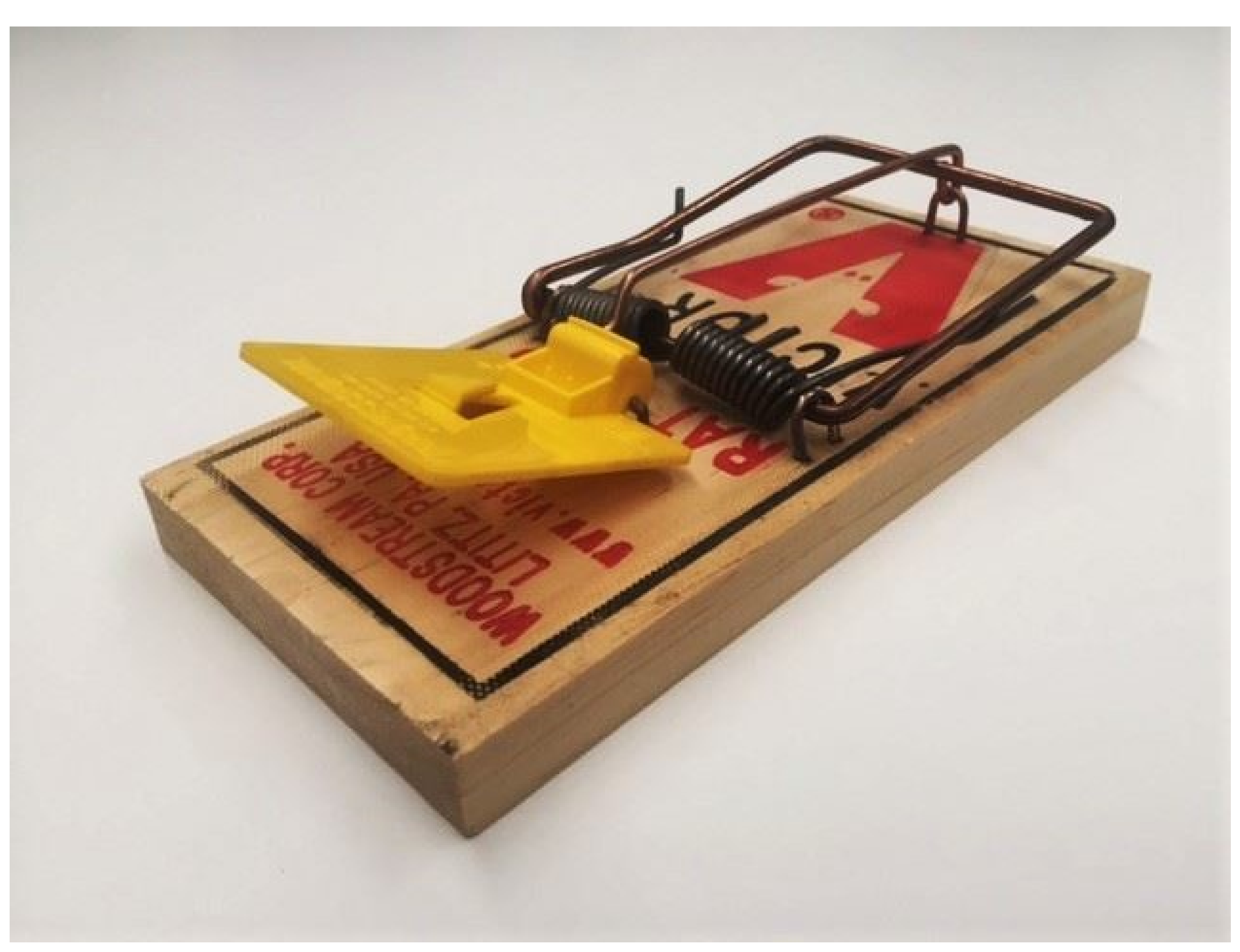
I will tell you a secret



I will tell you a secret

High code coverage !=
high code quality

Give me a number!



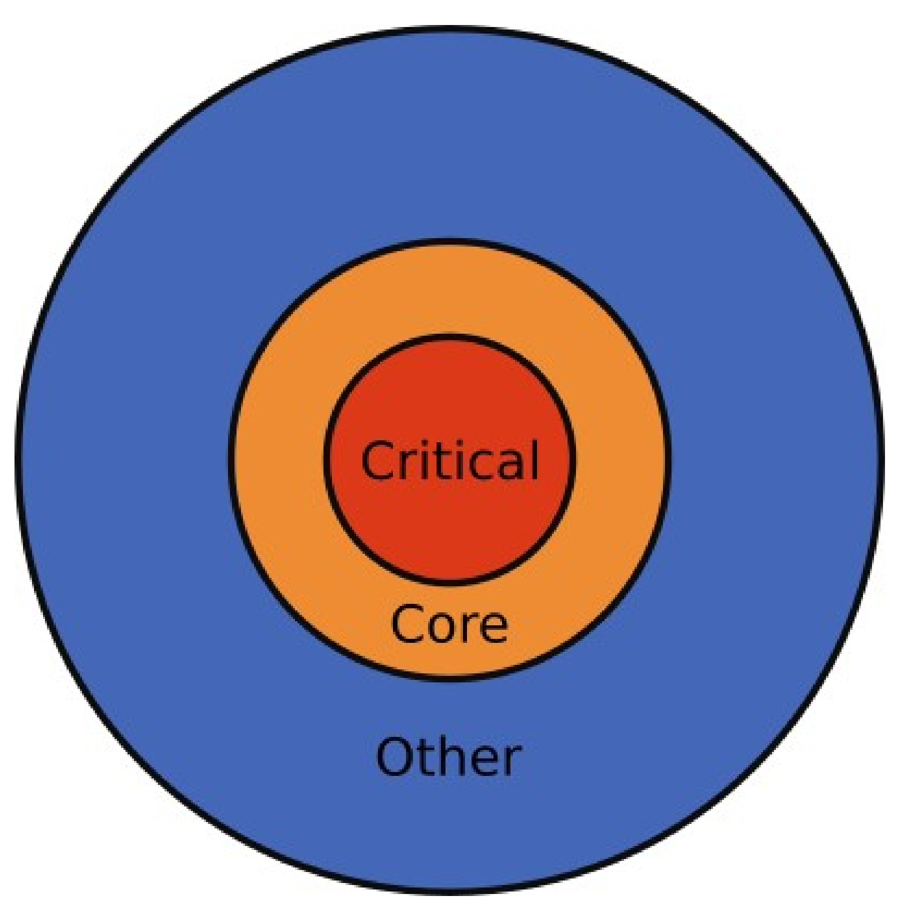
Best code coverage

20% is the magic number

Pareto principle

20% of your code is responsible for 80% of your bugs

Pareto principle



Pareto principle

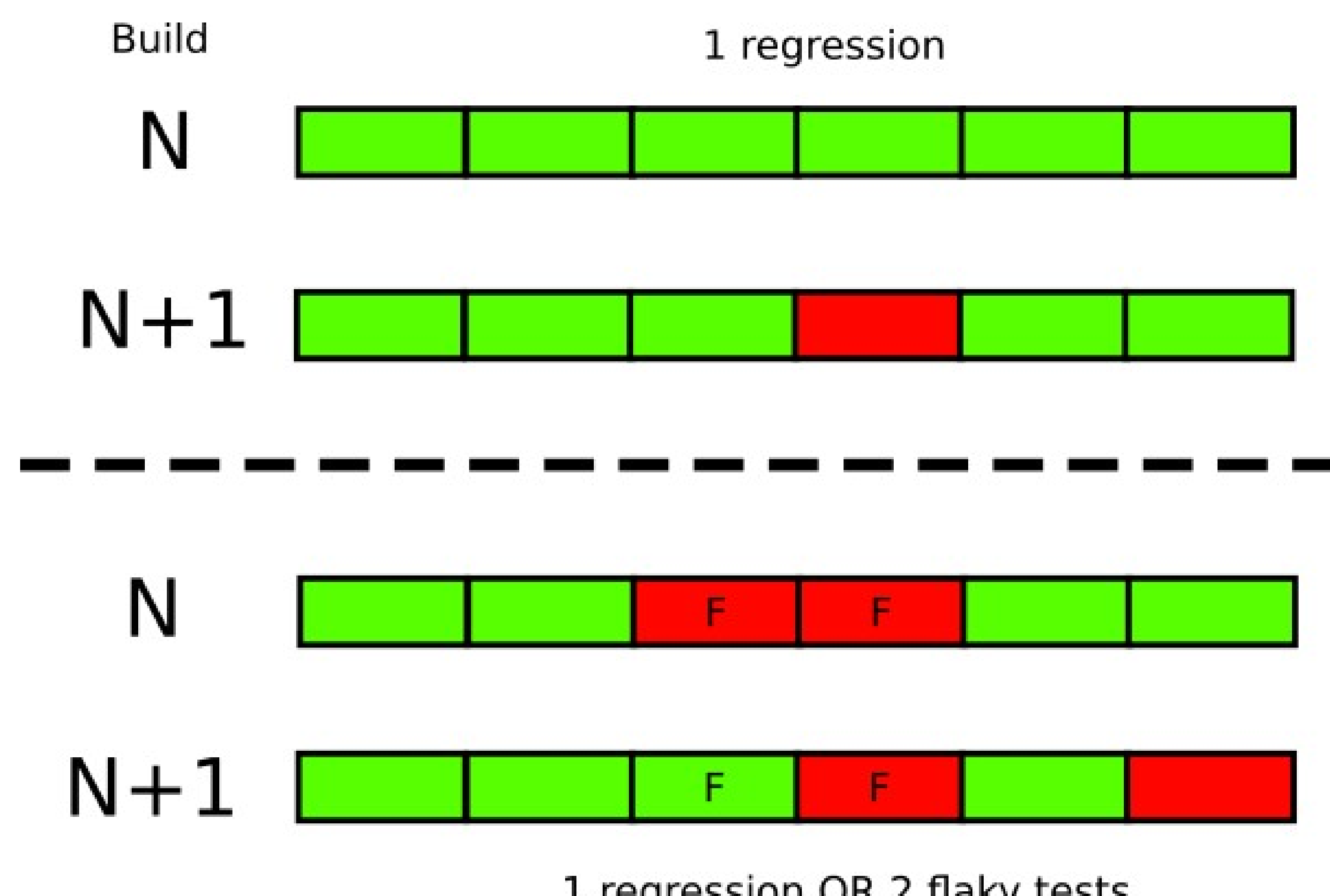
Try to achieve 100% coverage of your CRITICAL code, (which itself is probably 20% of total code)

Antipattern 7 – Flaky or slow tests

Antipattern 7 – Flaky or slow tests

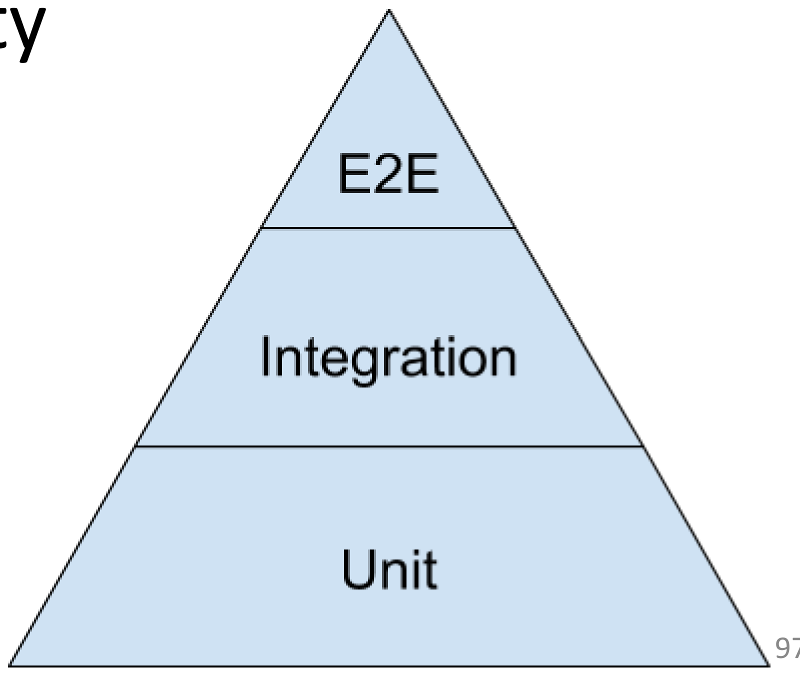
- Flaky tests are a well known problem
- They hide real bugs
- They make tests untrustworthy
- People start ignoring tests
- Everything goes downhill afterwards

Antipattern 7 – Flaky or slow tests



Antipattern 7 – Flaky or slow tests

- As we go up in pyramid tests become slow/flaky
- UI tests are notoriously problematic
- Test environments parity



Antipattern 7 – Solution

- Fix flaky tests
- Isolate them in a different test suite
- Tests should be rock solid
- Failure of test means immediate problem with code
- Exclude tests that are broken for a temporary reason

Antipattern 8 – Running tests manually

Quiz:

How many steps do you need to setup and run your whole test suite?

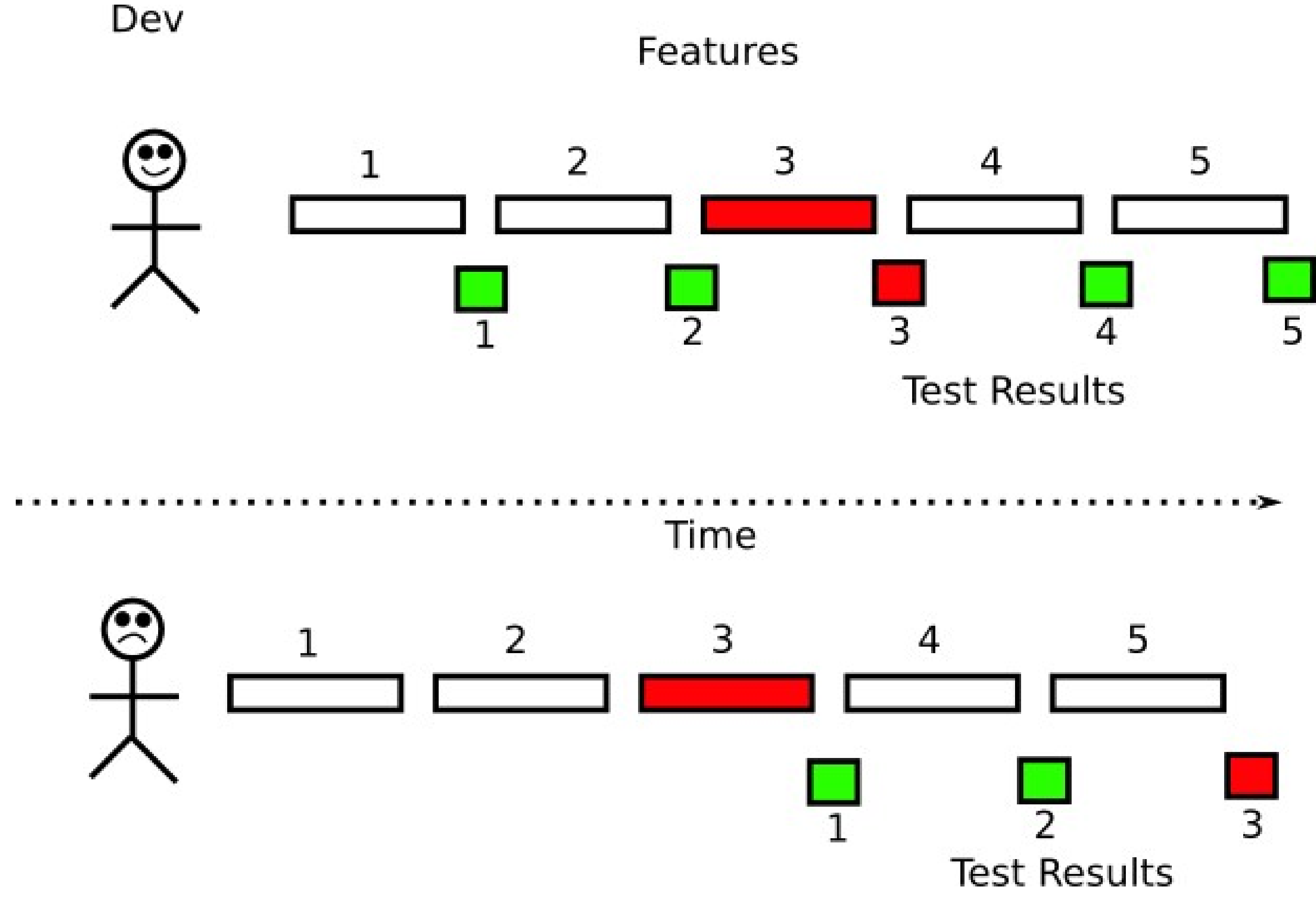
Wrong answers

1. Prepare database
2. Edit settings file
3. Prepare test environment
4. Run tests
5. Cleanup environment

Correct answer

- Before commit: single command to run tests
- After commit: Tests run automatically, with no human intervention

Correct answer



Quiz:

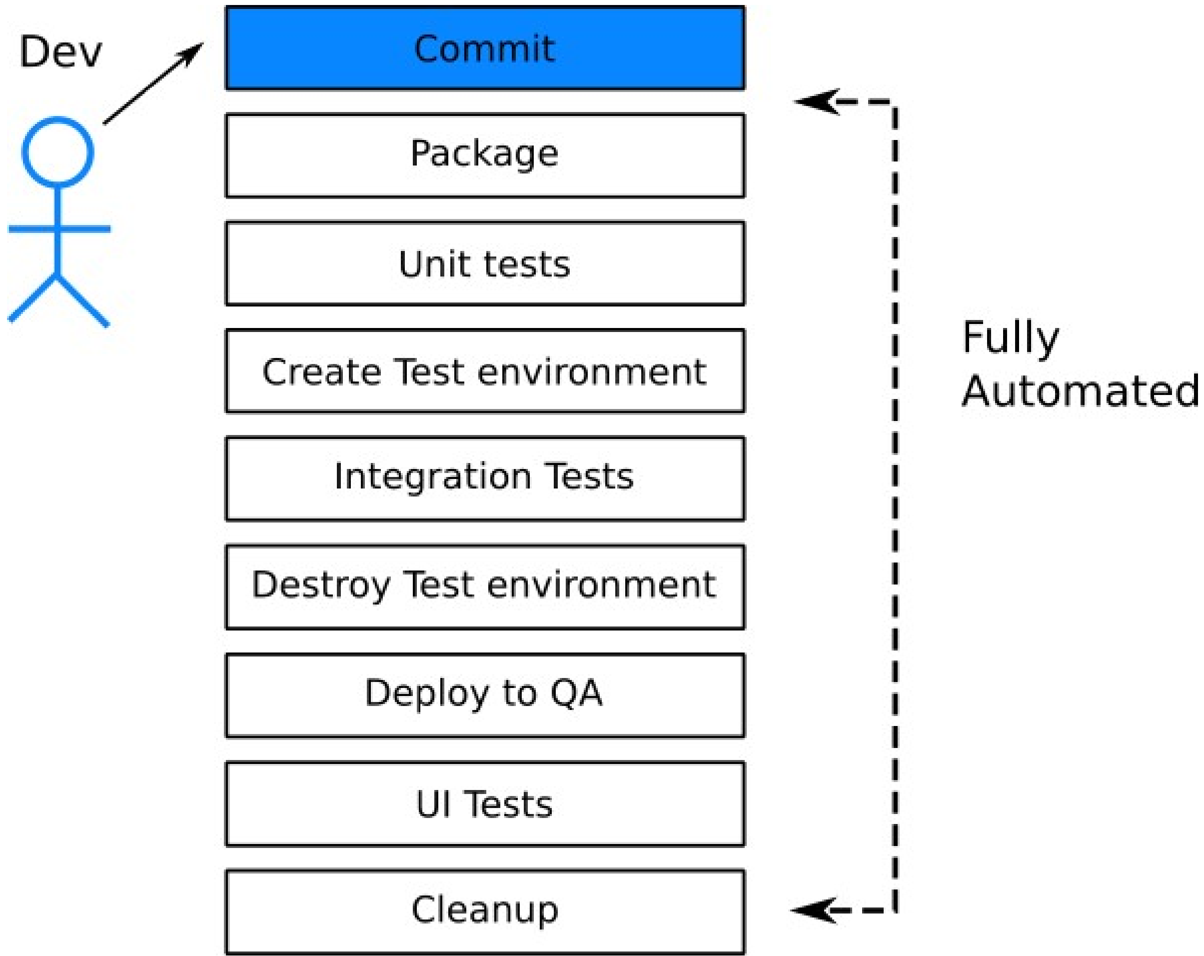
What is the role of the test engineer?

What is the role of the QA department?

Test engineers

- Test engineers should NOT run tests
- Test engineers should write NEW tests and add them in the automatic test suite
- QA department should NOT run tests
- QA department should only evaluate results from automatic test suites
- CI server actually runs 99% of tests
- 1% of smoke GUI tests run manually

Testing strategy



Antipattern 8 – Solution

- Automate everything
- Make local testing easy for developers
- CI server should run test for each feature branch in a transparent manner
- You should also have smoke/acceptance/production tests

Antipattern 9 – Not respecting test code

Antipattern 9 – Not respecting test code

- Developers pay great attention to main code
- They treat test code as second class citizen
- Test code is hacky and does not follow DRY, SOLID and KISS principles

I will tell you a secret

I will tell you a secret

Test code is as important as feature code

Antipattern 9 – Solution

- Create common abstractions for test data creation
- Centralize common assert code
- Refactor test code when needed
- Apply KISS, SOLID and DRY to test code
- Do not leave tech debt in test code

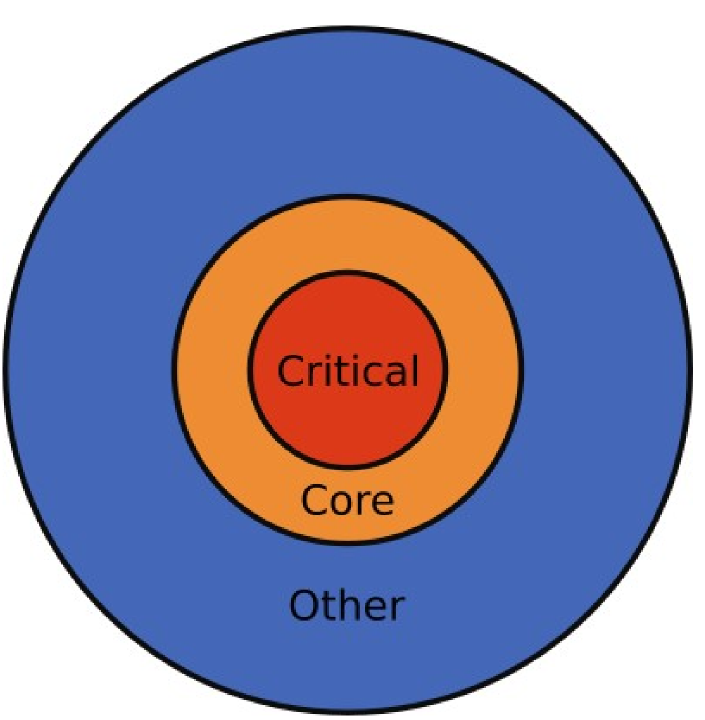
Antipattern 10 – Not converting production bugs to tests

Quiz:

You start working on an unknown project with zero tests. Where do you start testing?

Write test for code that

- Breaks often
- Changes often
- Is critical to the business



How do you find critical code

See what bugs appear in production

How do you find critical code

…and write unit/integration tests for them

Production bugs

- Have passed all QA gates (since they appeared in production already)
- Are great for regression testing

Production bugs

Should only happen once!

New project – zero tests

- Do NOT start testing code you understand
- Do NOT start testing code that requires easy tests
- Do NOT start testing the first folder in your file system
- Do NOT start testing what a colleague suggested

New project – zero tests

First test suite should be production bugs

Antipattern 11 – TDD madness

Antipattern 11 – TDD madness

- Test driven development says that tests are written before code
- Add test, run test, refactor, repeat

I will tell you a secret

You can write tests

- Before the feature implementation
- During the feature implementation
- After the feature implementation
- Never (see “Other” code severity)

TDD requires a spec

If you have no spec TDD is a waste of time

TDD is not needed

- For research code
- For throw away code
- For quick spikes/POCs
- For weekend projects
- For startups that pivot all the time

Antipattern 12 – Not reading test framework documentation

A professional is..

…somebody who knows the tools of the trade

Antipattern 12 – Not reading test documentation

- Do not re-invent the wheel
- Do not write new test utilities
- Do not create “smart” test solutions
- Do not copy paste test code
- Do not write “helper” test methods
- Do not ignore off-the-self test libraries

Research and learn

Your test framework and its capabilities

Learn about

- Parameterized tests
- Mocks and stubs (and spies)
- Test setup and tear down
- Test categorization
- Conditional running for tests
- Assertion grouping

Learn about

- Test data creators
- Http client libraries
- HTTP mock libraries
- Mutation/fuzzy testing
- Db cleanup/rollback
- Load testing

Assume that your “smart” solution

…is already invented and available on the internet

The end

Software Testing Anti-Pattern List

1. Having unit tests without integration tests
2. Having integration tests without unit tests
3. Having the wrong kind of tests
4. Testing the wrong functionality
5. Testing internal implementation
6. Paying excessive attention to test coverage
7. Having flaky or slow tests
8. Running tests manually
9. Treating test code as a second class citizen
10. Not converting production bugs to tests
11. Treating TDD as a religion
12. Writing tests without reading documentation first
13. Giving testing a bad reputation out of ignorance

http://blog.codepipes.com/testing/software-testing-antipatterns.html