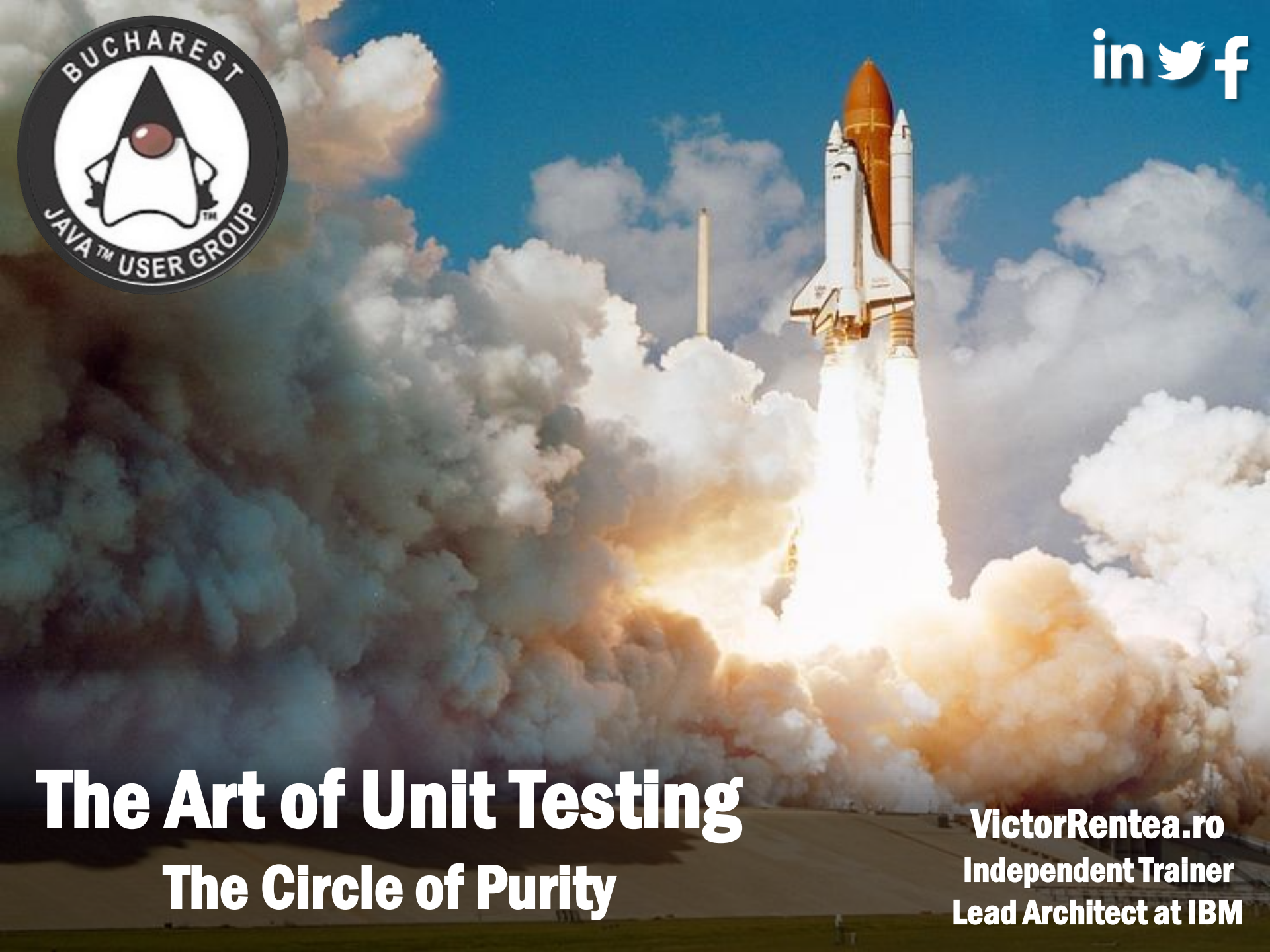


The Art of Unit Testing

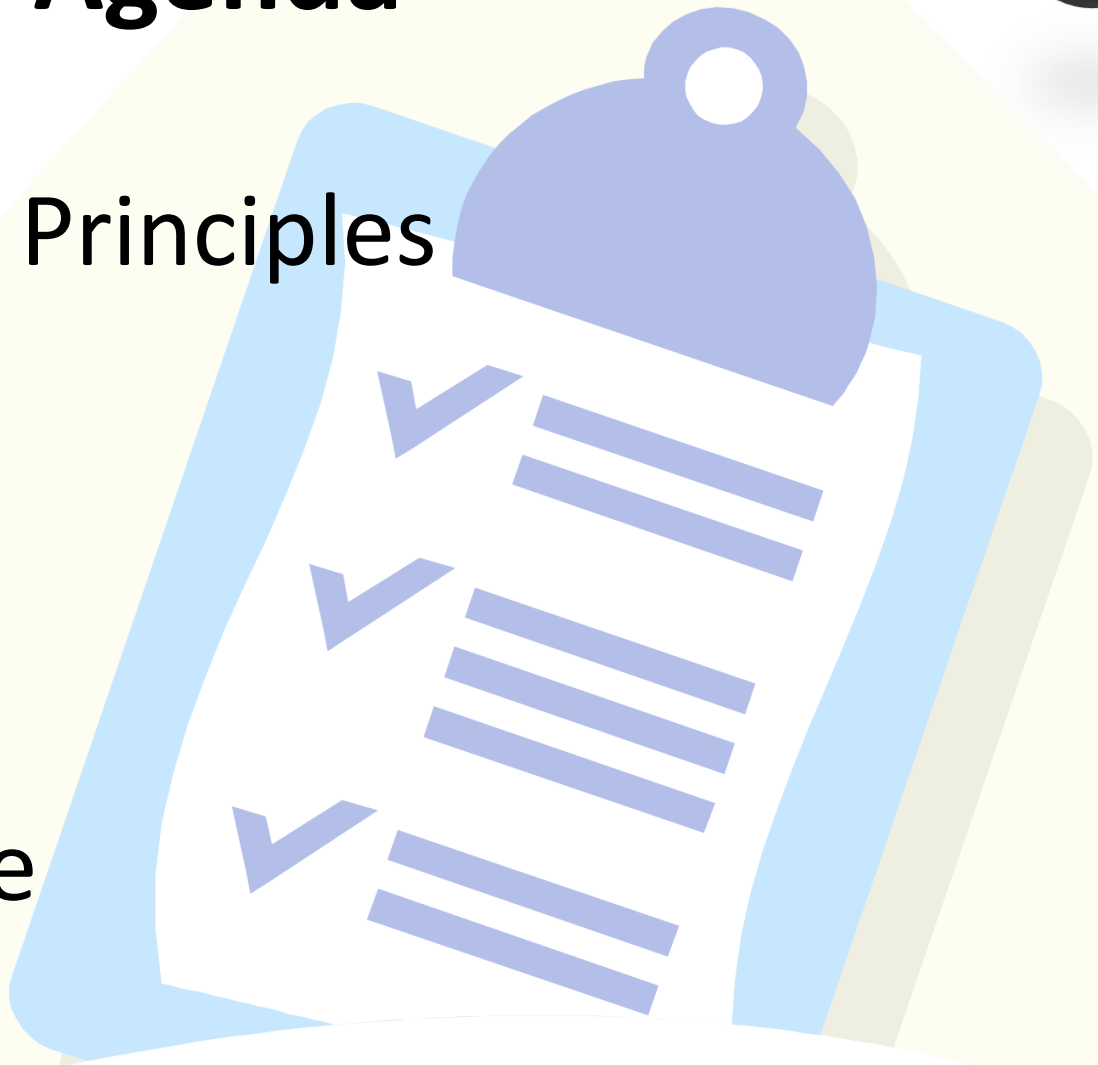
The Circle of Purity

VictorRentea.ro
Independent Trainer
Lead Architect at IBM

About me

Agenda

- Test Design Principles
- TDD
- Mocks
- Databases
- Legacy Code



Terms

Given

- **Arrange**: setup the test *fixture*

When

- **Act**: call the *production code*

Then

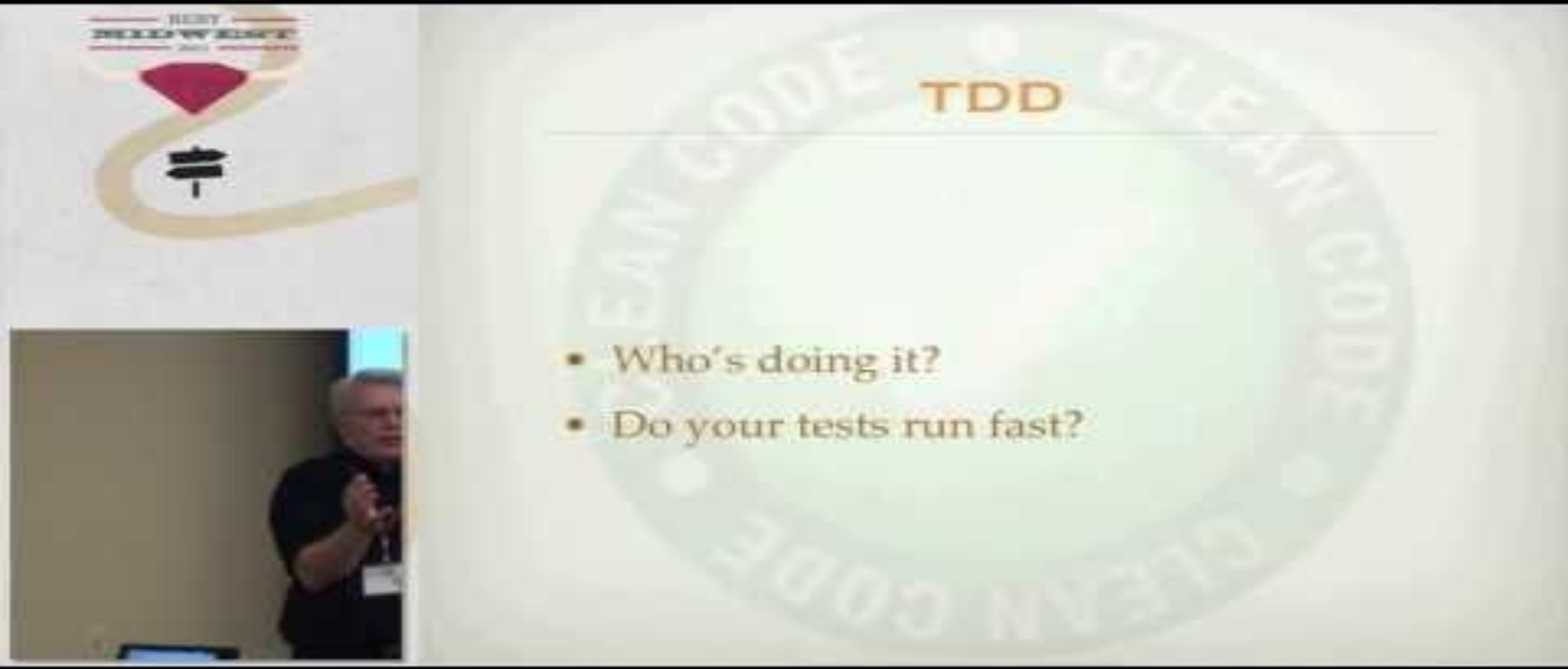
- **Assert** the outcomes

- **Annihilate**[opt]: clean up

Why do we Write Tests ?

Why do we Write Tests ?

Uncle Bob on TDD



1:48

Full: https://www.youtube.com/watch?v=GvAzrC6-spQ

Shorter: https://youtu.be/GvAzrC6-spQ?t=116

Why do we Write Tests ?

know when we're done

To Read-the-Spec™

(before we finish the implem☺)

executable spec

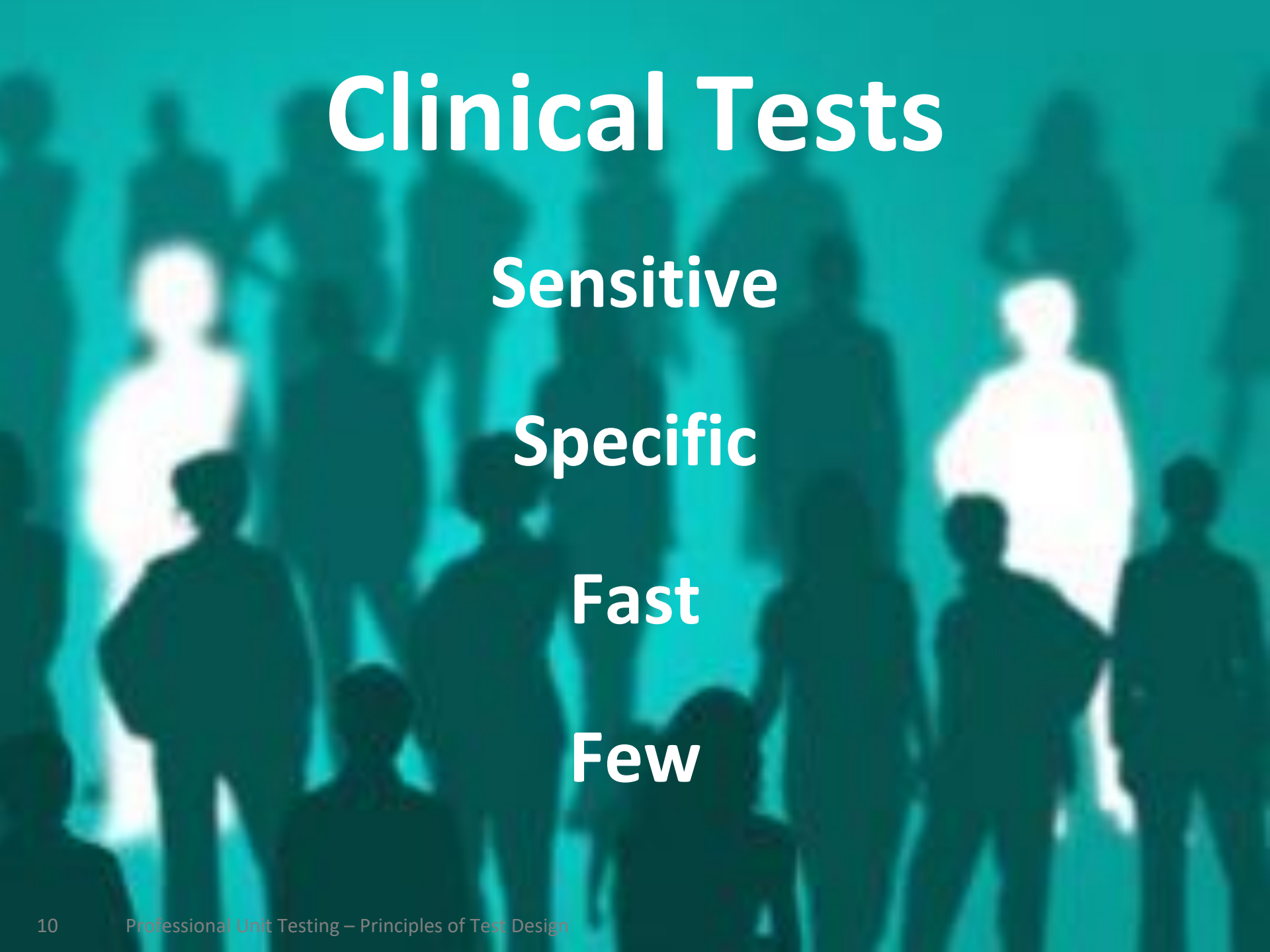
better design

(simple, loose, API focus)

~~**FEAR**~~

faster feedback

Medical Screening


Clinical Tests
Sensitive
Specific
Fast
Few

Test Design Goals

Sensitive — fail for any bug

High Coverage %

Correct Tests

Specific — precise failures

Expressive

Isolated & Robust

Low overlap

Fast

Few

Small, DRY tests

= Maintainable

Test code >> Prod code

Not maintainable?

Under pressure, devs might:

@Ignore

delete test

delete asserts

// comment test

invert asserts

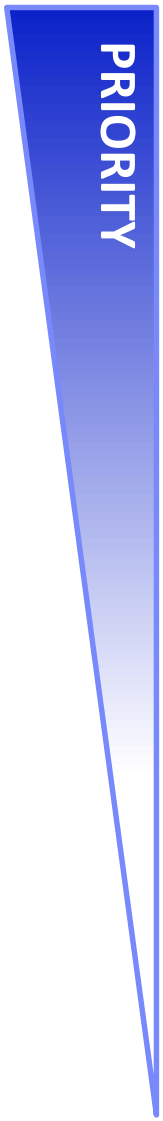
// @Test

Victor Rentea
@VictorRentea

Most developers hate testing. They tend to test gently, subconsciously knowing where the code will break and avoiding the weak spots. #TDD should fix that. #ruthless #mercyless #unittesting

♡ 7 9:52 AM - Feb 15, 2019

Testing Priorities



- Code that you fear 😱
  (hard to understand)
- A distant corner in the logic
  (takes 3 mins of UI clicks to reproduce)
- A bug (before fixing it) 🏆
- for/if/while ⚙
- Thrown exception
- A method that just calls two others
- Trivial code `a.setName(b.getName());`
- Legacy code with 0 bugs, 0 changes
  (over the last year)

We ♡ Green Tests!

Tests are **green**.

You implement a huge CR in Prod code.

Tests are **green**.

Evergreen Tests


Evergreen Tests

Evergreen Tests

```
public void activate() {
    status = Status.ACTIVE;
    activationTime = LocalDateTime.now();
    publishEvent(new CustomerActivated(id));
}
```

Coverage% = ?

```
@Test
public void activate() {
    customer.activate();
}
```

Can you

Trust

Such Tests?

How can you check that your test really covers what it says?

How can you check that your test really covers what it says?

Mutation Testing

Tests fail ⇔ Production code is altered

You make some change to the Production code

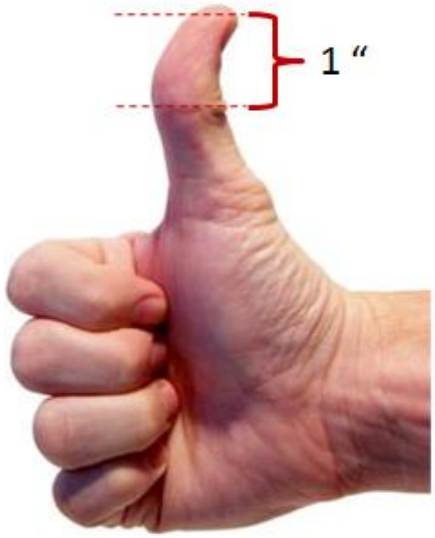
You run the Tests

You undo the change

You get some "mutant" code

The tests should squash the mutant by turning **red**

There are frameworks that play like this with your code, but their results might be depressing



Always see your test failing at least once

Tests, like laws, are meant to be broken

Why can a Test Fail ?

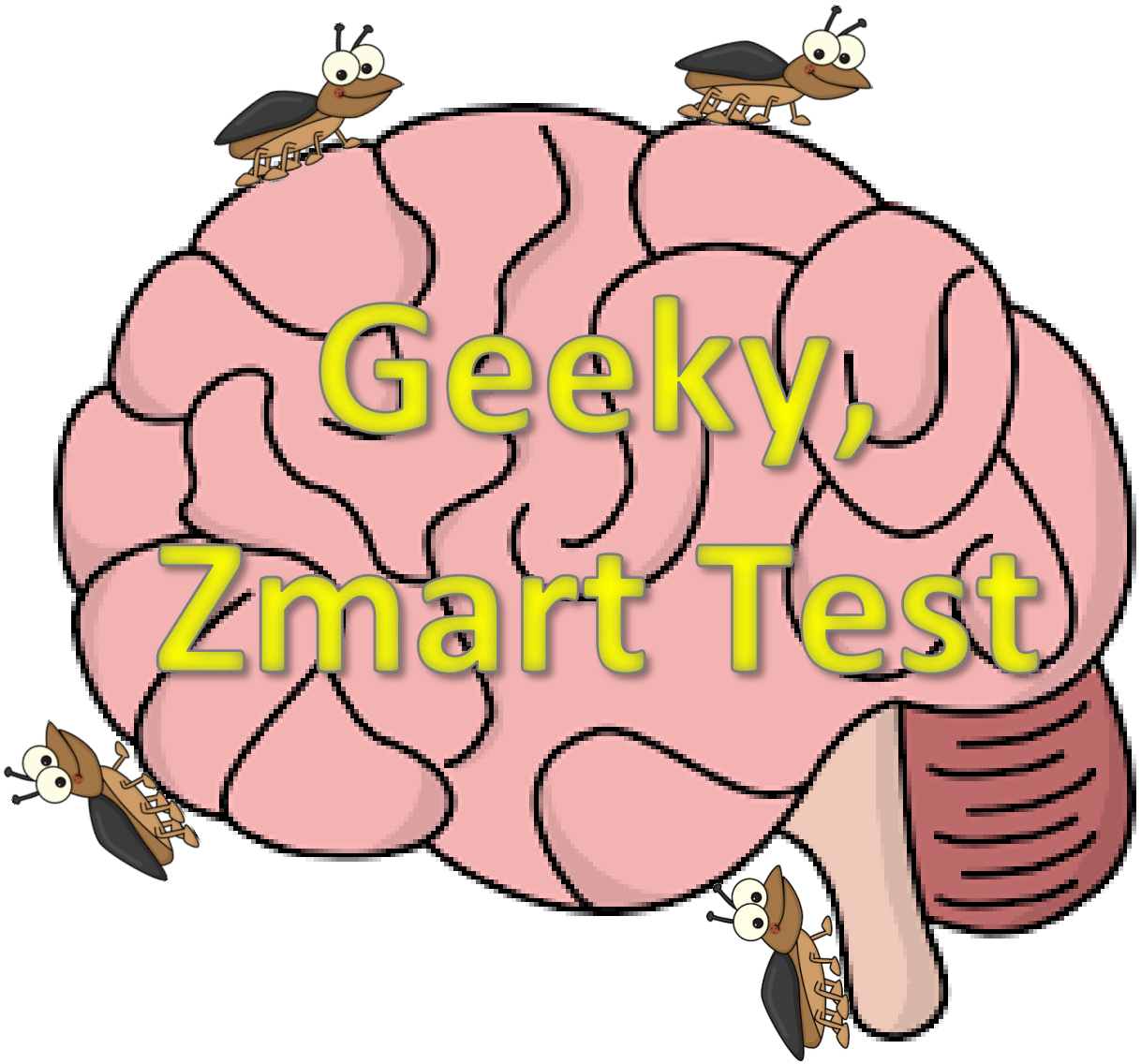
- Regression
  - Yeehaaa! ➔ fix it
- Change Request
  - Double Check ➔ Update/Delete it ?

- Iso-functional changes
  - **Superficial API Changes** ➔ adjust the test
  - **Refactoring** internals ➔ loosen the test coupling
- A Bug in a Test
  - Usually discovered via debugging ➔ fix the test

the usual suspect = production code

A Bug in a Test

A Bug in a Test


Geeky,
Zmart Test

You need

Explicit
Simple Tests !

Concatenation, arithmetic

```
assertEquals("Greetings" + name, logic.getGreetings(name));
```

Can you find the bug? it's missing a " "! production code

Why? The same Dev wrote both prod and test code! ☺ CPP

➔ **Prefer literals** ➔

```
assertEquals("Greetings John", logic.getGreetings("John"));
```

if, switch = logic

Did you also tested that logic? ("test the tests" ☺)

Simple Tests !

Simple Tests !

catch-assert

When you want to verify an exception is thrown

```
try {
   logicThatShouldThrow();
   fail("should have thrown");
} catch (MyException ex) {
   assert("msg", ex.getMessage());
}
```

Too easy to get it wrong

Can you find the bug?

Verify an Exception Type is Thrown

```
@Test(expected = IllegalArgumentException.class)
public void expectExceptionTypeThrown() {
   // throving code
}
```

For precision: let's create 20 new Exception classes !

Verify Exception Message/Code

```
@Rule
public ExpectedException exception = none();

@Test
public void expectExceptionWithDetails() {
  exception.expectMessage("contains this"); // or…
  exception.expect(exceptionWithCode(FILE_LOCKED));
  // throving code
}
```

Hamcrest

Junit 5

```
@Test
public void assertException() {
  IllegalArgumentException ex = assertThrows(
       IllegalArgumentException.class,
       () -> throwingProdCode());

  assertEquals(FILE_LOCKED, ex.getErrorCode());
}
```

Simple Tests !

for loop

WHY?!

Testing a set of input-output pairs?

➔ Parameterized Unit Tests ➔

➔ .feature files ➔

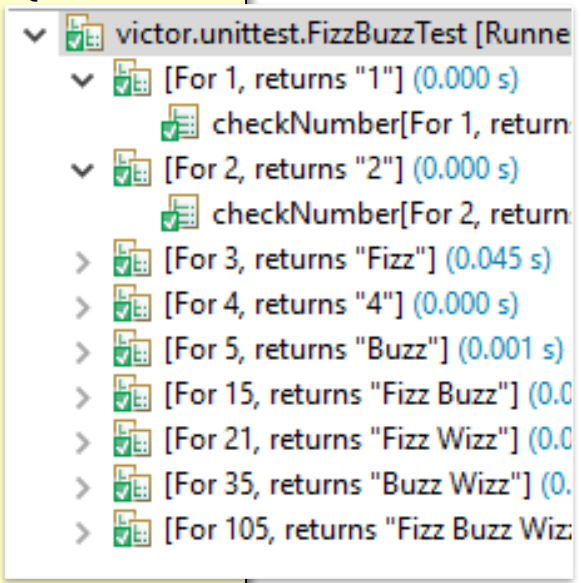
Parameterized Tests

```
@RunWith(Parameterized.class)
public class FizzBuzzTest {
  private final int number;
  private final String expected;

  public FizzBuzzTest(int number, String expected) {
    this.number = number;
    this.expected = expected;
  }

  @Parameters(name="For {0}, returns \"{1}\"")
  public static List<Object[]> getParameters() {
    return asList(
        new Object[] {1, "1"},
        new Object[] {2, "2"},
        new Object[] {5, "Buzz"},
        new Object[] {3*5, "Fizz Buzz"},
        new Object[] {3*7, "Fizz Wizz"},
        [...]
        );
  }

  @Test
  public void checkNumber() {
    assertEquals(expected, FizzBuzz.getNumber(number));
  }
}
```



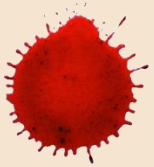
(instead of asserting in a loop)

.feature files

Can be signed directly by Biz guys !

```
Scenario Outline: Tennis Score is correct
    Given An empty game
    When Player1 scores <player1points> points
    And Player2 scores <player2points> points
    Then The score is "<expectedScore>"

    Examples:
        | player1points | player2points | expectedScore |
        |             0 |             0 | Love-Love     |
        |             1 |             0 | Fifteen-Love  |
```



```
@When("^Player(\\d+) scores (\\d+) points$")
public void player_scores_points(int playerNo, int points) {
    for (int i = 0; i<points; i++) {
        tennisScore.point("Player"+playerNo);
    }
}

@Then("^The score is \"([^\"]*)\"$")
public void the_score_is(String expectedScoreStr) {
```

Simple Tests !

random

Testing to find errors?

➔ replay real calls from production ➔

(Even more: Golden Master Technique)

multi-thread

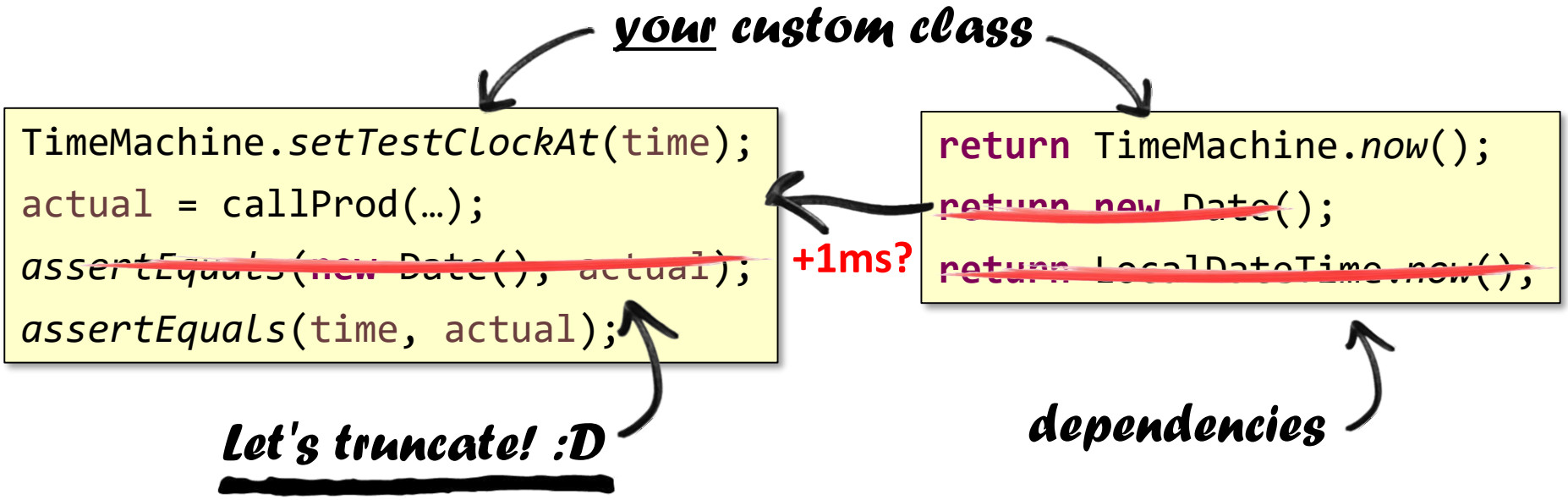
Looking for a race bug with sleep(random) ?

➔ Use-Your-Brain™ instead ➔

asserting time

asserting time

fails/passes randomly!



draconian

PowerMock?

Tests Can

Control the Time

Simple Tests !

If such a test fails, is it always a bug?

files

databases

queues

external web services

These are Integration Tests

Integration vs Unit Tests

Deep Testing

through many layers

Overlapping

is normal

GREEN ZONE

Slow

start-up
the container

Fragile

(DB, files, external systems)

Their failure is not always a bug

In-mem DB

Super-Fast

100 tests/sec
(some can be slower)

Test Design Goals

Sensitive

High Coverage %

Correct Tests

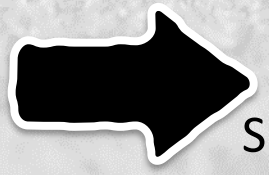
Specific

Expressive

Isolated & Robust

Low overlap

Fast



Few

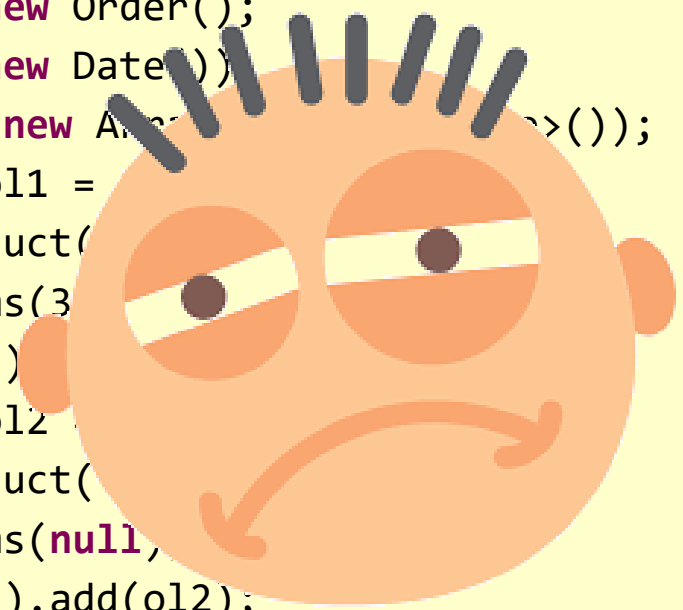
Small, DRY tests

Minimize Duplication

Data Initialization

```
public void myTestRaw() {
  Order o = new Order();
  o.setDate(new Date());
  o.setLines(new A                    >());
  OrderLine ol1 =
  ol1.setProduct(
  ol1.setItems(3
  o.getLines()
  OrderLine ol2
  ol2.setProduct(
  ol2.setItems(null);
  o.getLines().add(ol2);

  target.process(o);
}
```



Fluent Builder

```
public void myTestWithBuildersAndObjectMother() {
  Order o = anOrderWithOneLine()
              .with(anOrderLine().withItem(anItem()))
              .build();
  target.process(o);
}
```

Object Mother

DRY Tests

- Fluent Builder®
- Factory methods
  - **Object Mother** SHARED between tests ?
- Wrap production API
  - In a more testable API
- Keep common stuff in fields
- Shared @Before

Safe

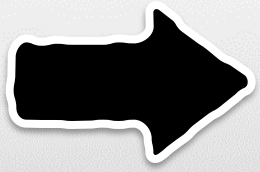
Each @Test runs on a
fresh instance of the Test class

Test Design Goals

Sensitive

High Coverage %

Correct Tests



Specific

Expressive

Isolated & Robust

Low overlap

Fast

Few

Small, DRY tests

Abusing @Before

==Practical Guide☺==

You want to shrink a large test

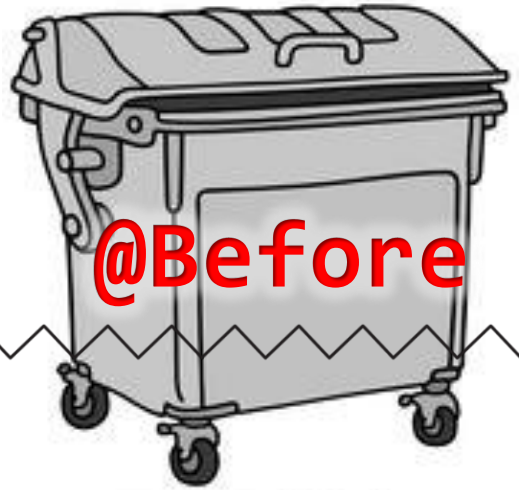
⇩

You move some code in the @Before

- Data initialization
- Mocks
- Infra setup: DB, ..

⇩

Other tests do the same:

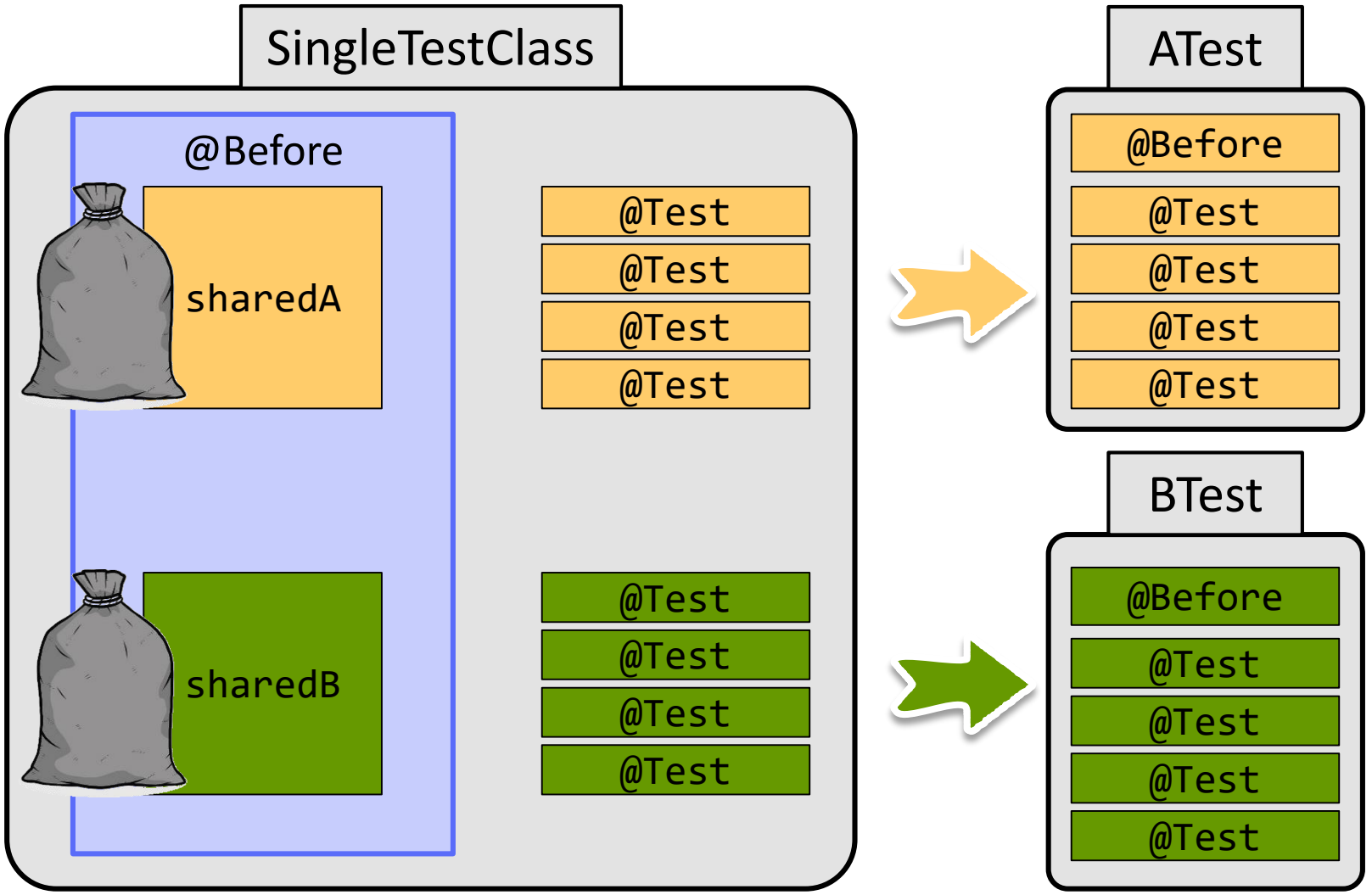


And time passes by

When you read a test:

- *Arrange* = method + @Before
- What part of it ? ↗

Fixture-Based Test Breakdown



Nested Test Fixtures in JUnit5:
https://junit.org/junit5/docs/current/user-guide/#writing-tests-nested

Start with one Test class/Prod class

`RecordRepositoryTest`

- If it grows (>300 lines?) or
- If many tests share a fixture

Split it

`RecordRepositorySearchForVersionTest`

Maybe also split the prod code ?

Test Methods

5-10 lines

Descriptive names

```
public void givenAnInactiveCustomer_whenHePlacesAnOrder_itIsActivated()

public void getRecordDeltasForVersionWithDeletedRecord()

public void givenAProgramLinkedTo2Records_whenUpdated_2AuditsAreInserted

RecordRowValidatorTest.invalidRecordStartDateFormat()
```

A Naming Convention

https://codurance.com/2014/12/13/naming-test-classes-and-methods/

"**Given**" = the class name

~ @Before

No need for method prefixes

```
AnInactiveCustomerShould
    .beActivatedWhenHePlacesAnOrder()
```

Test names start with "**Then**" part

the expectations

"**When**" follows


Superb Failures
Expressive Tests

Suggestive Literals

```
@Test
public void customerEmailIsUpdated() {
   Long customerId = createCustomer(new Customer("oldPhone"));
   updateCustomerPhone(customerId, "newPhone");
   Customer updatedCustomer = getCustomer(customerId);
   assertEquals("newPhone", updatedCustomer.getPhone());
}
```

d:<[new]Phone> but was:<[old]Phone>

Beautiful Failure Messages
(an example)

Assert a collection

```
assertEquals(1, names.size());
assertEquals("name", names.iterator().next());
```

expected:<1> but was:<2>

vs

```
assertEquals(Collections.singleton("name"), names);
```

expected:<[name]> but was:<[oldName, name]>

Single Assert Rule

one feature/test

NOT 1 assert/test !

```
assertEquals("FN", customer.getFirstName());
assertEquals("LN", customer.getLastName());
```

Try to name this test:

```
assertEquals(Status.SUCCESS, result);
verify(emailService).sendEmail(anyObject());
```

➔ Break it in 2 tests ➔

(perhaps break the Prod too? ~ CQS)

Test Design Goals

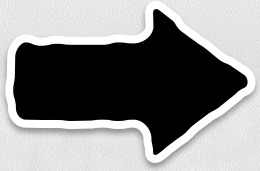
Sensitive

High Coverage %

Correct Tests

Specific

Expressive



Isolated & Robust

Low overlap

Fast

Few

Small, DRY tests

Should Tests Respect Encapsulation?

Fresh code:

Test through the public API

Tests won't break when implem changes

Practices the API

Legacy code:

Breaking encapsulation

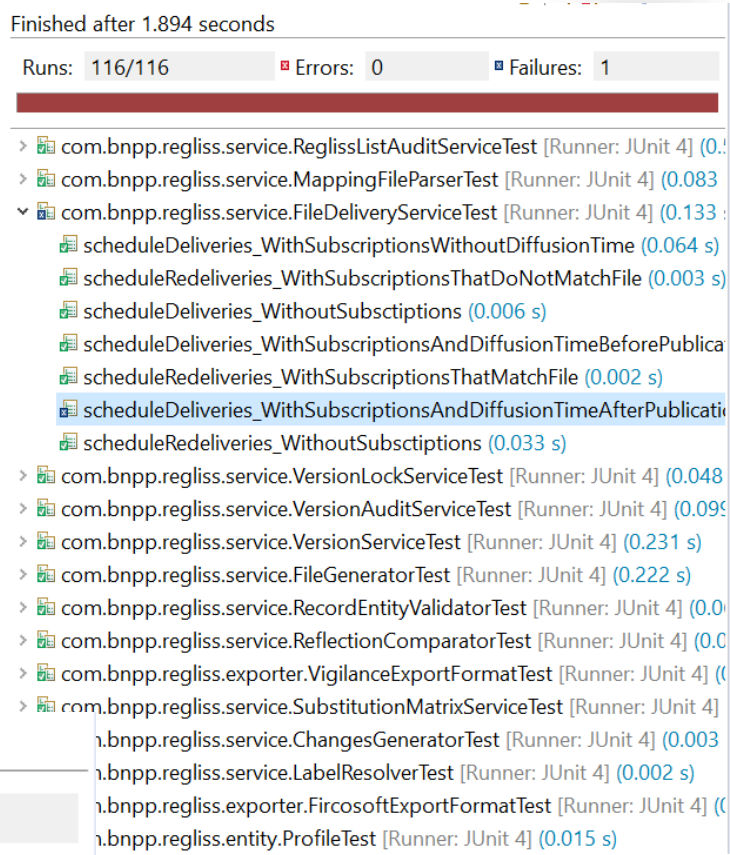
is a core technique

Temporarily use package/public protection

Yet Another Happy Day#2…

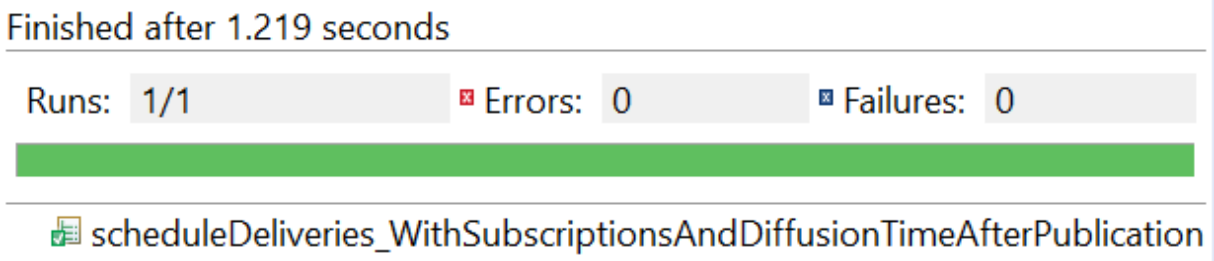
You run your test suite

Test #13 is **RED**



To debug it, you run it alone:

Test #13 is **GREEN**

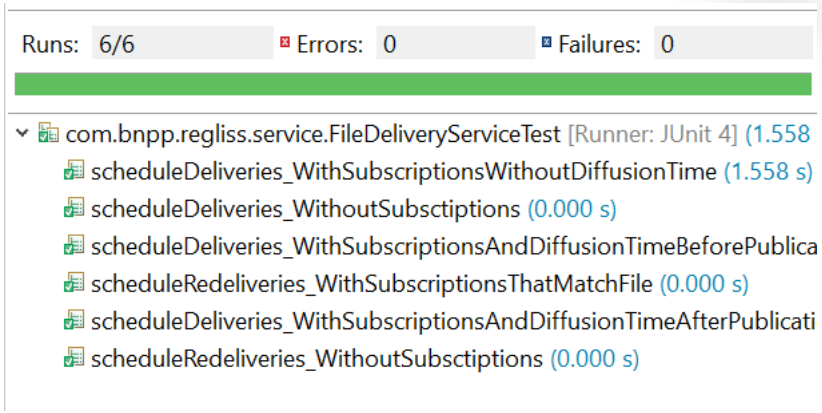


??!!!?!

Yet Another Happy Day#3…

You run your test suite

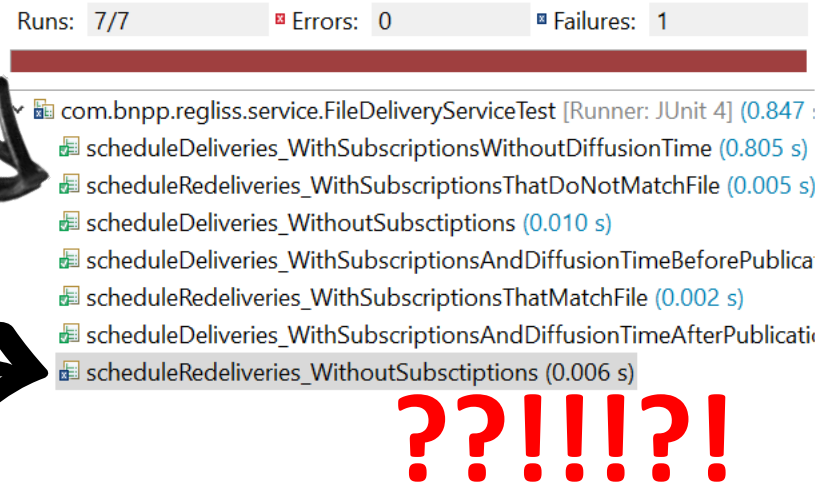
It's **GREEN**



You add Test #43

Test #13 turns **RED**

(unchanged!)



??!!!?!

You delete Test #43

Test #13 goes back **GREEN**

??!!!?!

Hidden Test Dependencies

- **In-memory Objects**
  - Static fields, singletons
  - PowerMocks
  - Thread Locals
  - Container shared by tests [Spring]
- **COMMITs to DB**
  - In-memory or external
- **External files/systems**

➔ Reset them in @Before ➔

To Ensure Independency

JUnit runs tests in unexpected order

(DON'T DO THIS)

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
```

Test Design Goals

Sensitive

High Coverage %

Correct Tests

Specific

Expressive

Isolated & Robust

Low overlapping

Fast

Few

Small, DRY tests

Agenda

- Test Design Principles
- **TDD**
- Mocks
- Databases
- Legacy Code

TDD

A **discipline** in which

you grow software in **small increments**

for which you write

the **tests before the implementation**

Credited to Kent Beck, 2003

TDD Mantra

Red-Green-Refactor

Overlapping? –
Incorrect? –

pass

Run Tests

fail

With Grace

If you get stuck or other tests break => revert and retry

Write a Failing Test
for a new feature

- Start with the simplest cases
- Start with the ☺ flows

Write prod code to
Make It Pass

1-10 min

Run Tests

fail

KISS, **YAGNI**, **Golf**:
Write as little prod code,
don't mind about quality

pass

pass

Run Tests

You can't break anything

Refactor
prod + tests

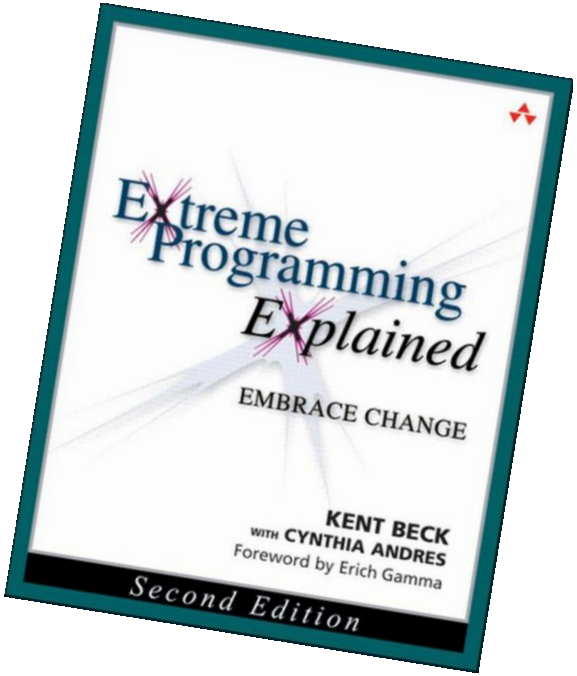
DRY, **Clean Code**

Rules of Simple Design

Rules of Simple Design

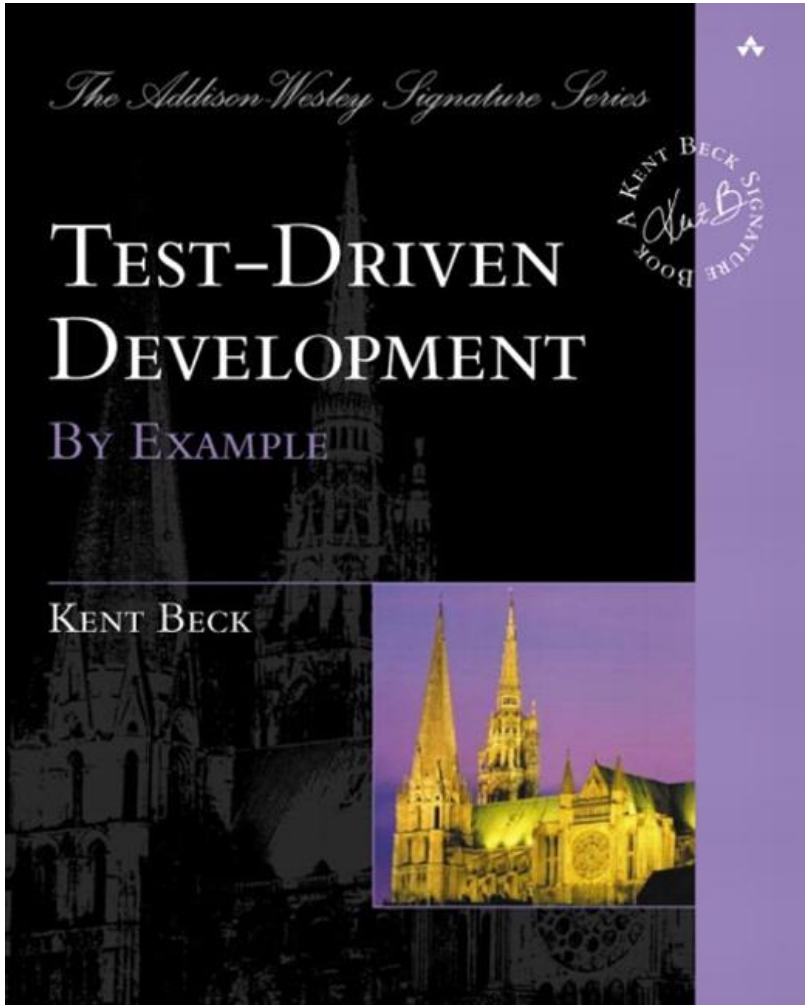
- Kent Beck

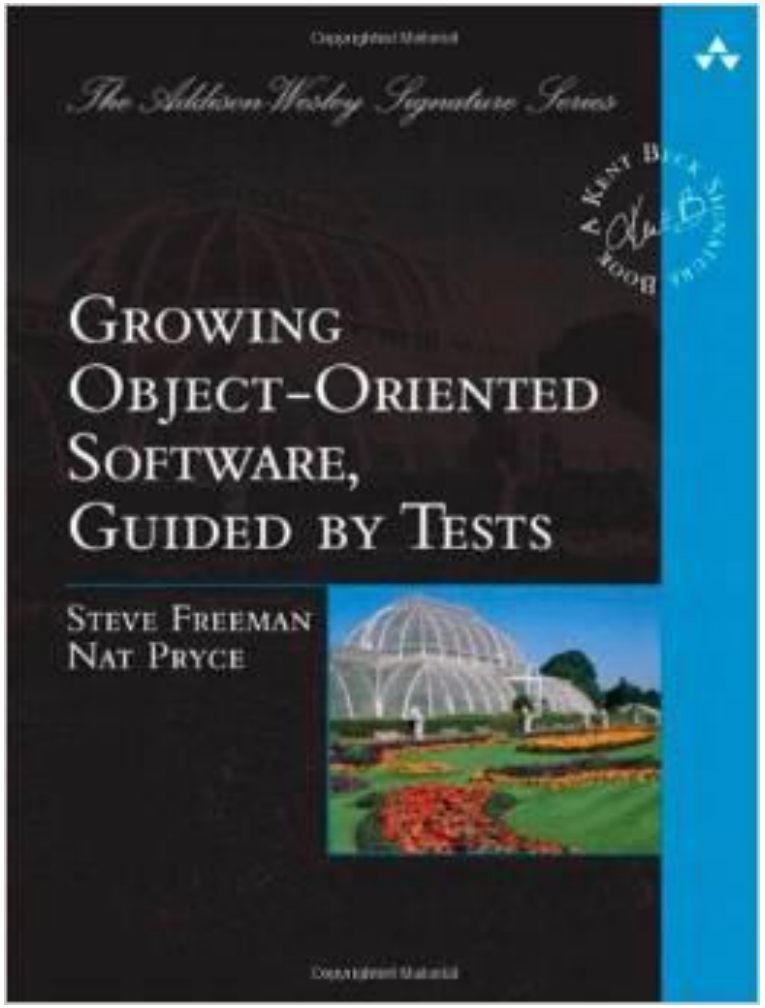
1. Passes the Tests
2. Reveals Intention
3. No Duplication
4. Fewest Elements



https://martinfowler.com/bliki/BeckDesignRules.html

TDD Schools of Thought

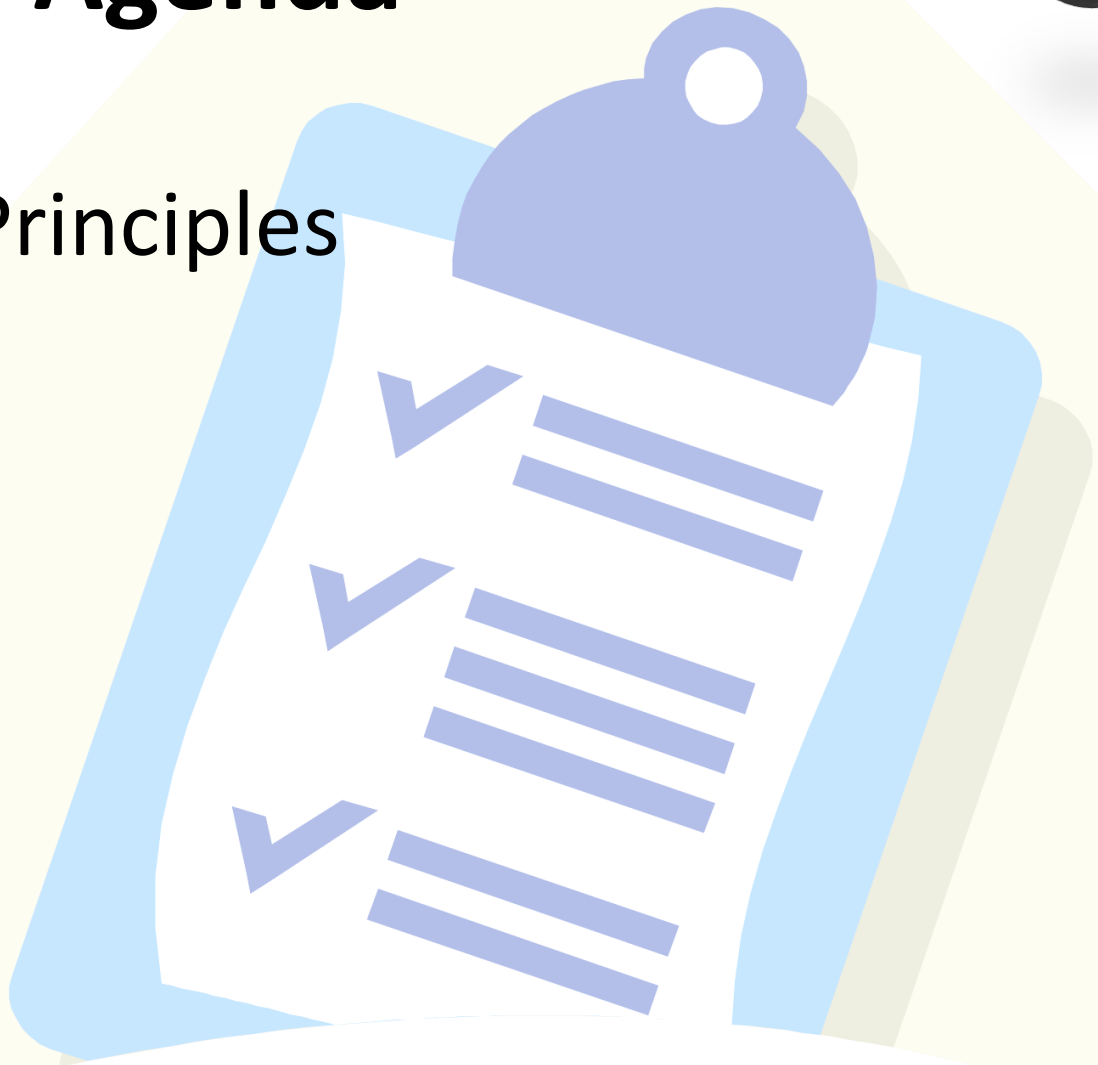




TDD as if you meant it

Agenda

- Test Design Principles
- TDD
- **Mocks**
- Databases
- Legacy Code



Fake Objects

WHY?!

Fast

~~DB~~ ☺

Repeatable

Isolation from external systems

Mentally Simpler

To test a layer, fake the layer bellow:

Fake

Stub

Respond to method calls

```
when(mock.method())
    .thenReturn(…);
```

Mock

Verify a method call

```
verify(repo).save(…);
```

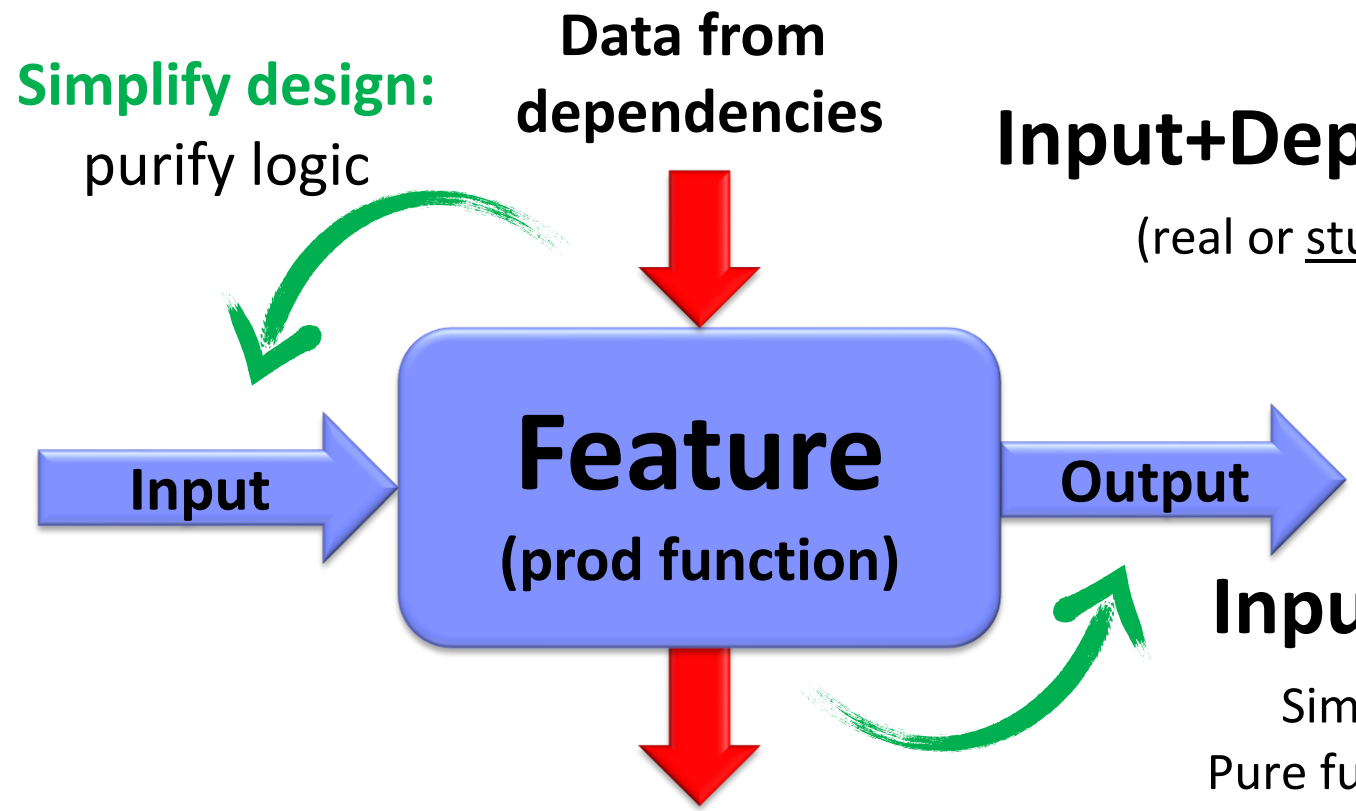
The confusion:

Mocking Frameworks

Mockito
EasyMock
PowerMock
jMock

The Circle of Purity

What can a Unit Test check?



Input+Deps ⟶ Output

(real or stubbed)

Input ⟶ Output

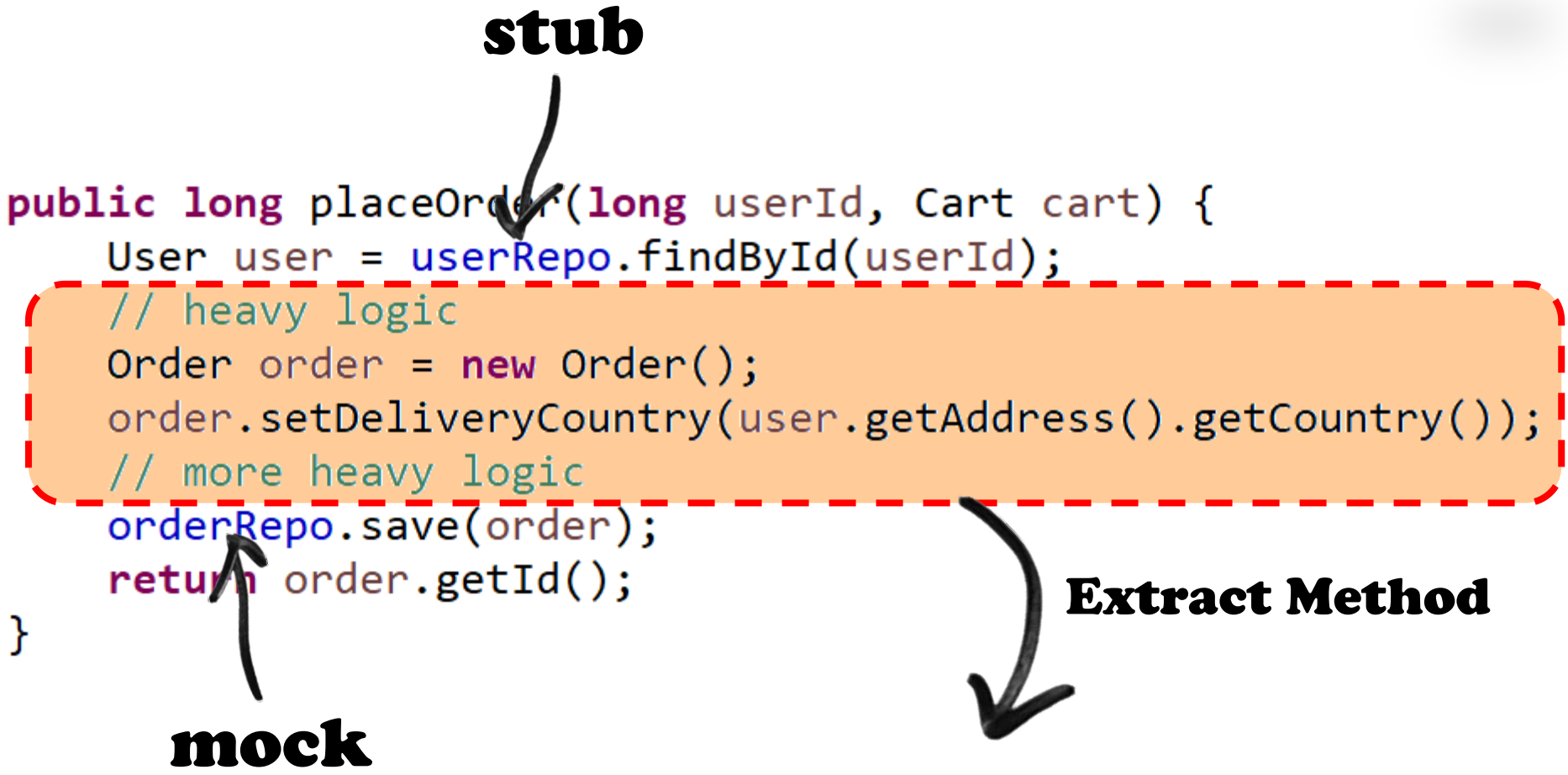
Simple and Elegant

Pure function: $f(x) = x^2$

Side-effects

1) On external systems:
   - a) **Real**: SELECT
   - b) **Mock**: verify repo.save() was called
2) On in-memory objects

➔ assert their state

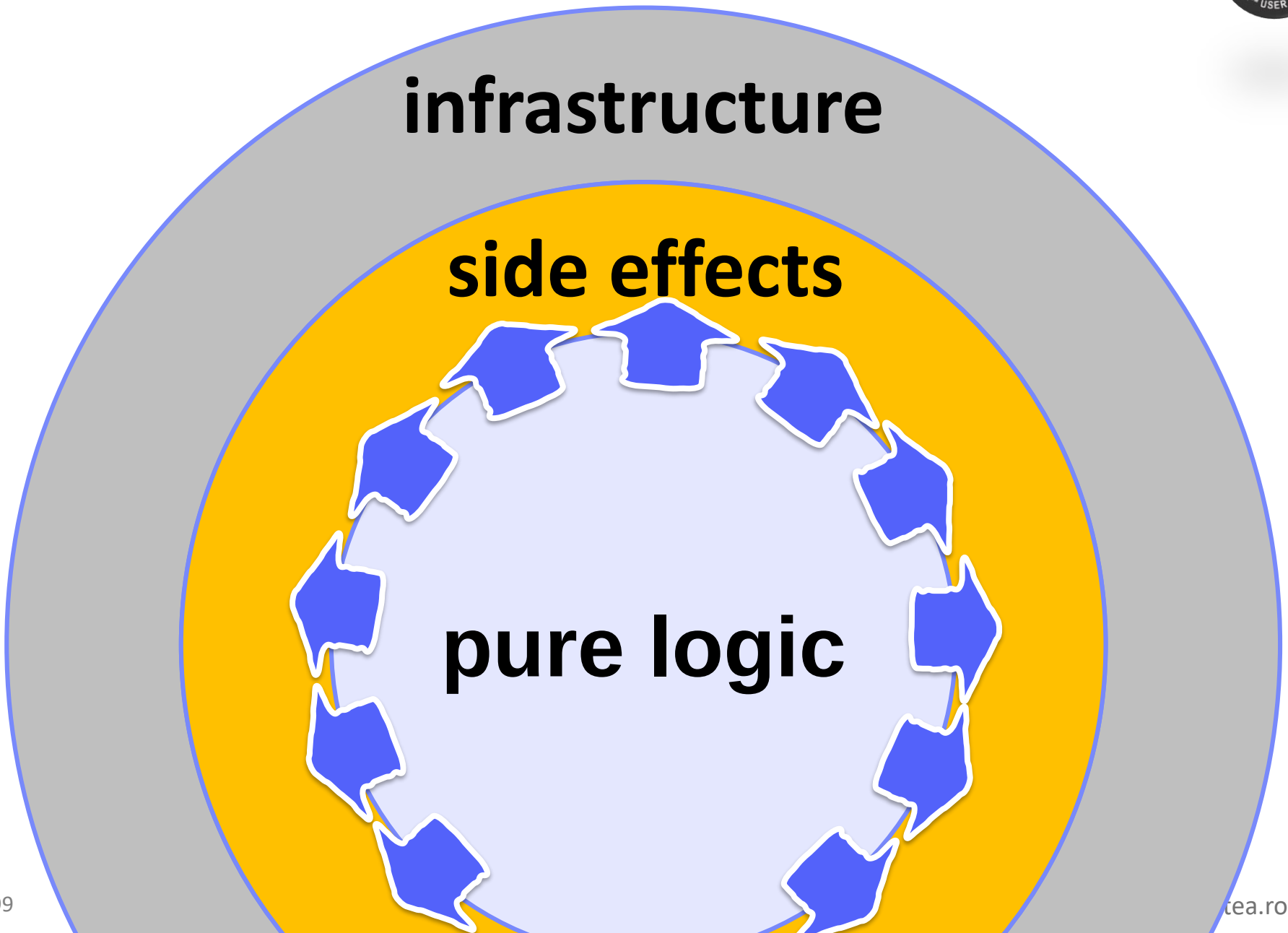

stub
public long placeOrder(long userId, Cart cart) {
User user = userRepo.findById(userId);
// heavy logic
Order order = new Order();
order.setDeliveryCountry(user.getAddress().getCountry());
// more heavy logic
orderRepo.save(order);
return order.getId();
}
Extract Method
mock


```
public long placeOrder(long userId, Cart cart) {
    User user = userRepo.findById(userId);
    Order order = createOrder(user, cart);
    orderRepo.save(order);
    return order.getId();
}

Order createOrder(User user, Cart cart) {
    // heavy logic
    Order order = new Order();
    order.setDeliveryCountry(
            user.getAddress().getCountry());
    // more heavy logic
    return order;
}
```

Mock-Free Unit Tests

= Pure Function


infrastructure
side effects
pure logic

Verify every method call

How many times calls happen

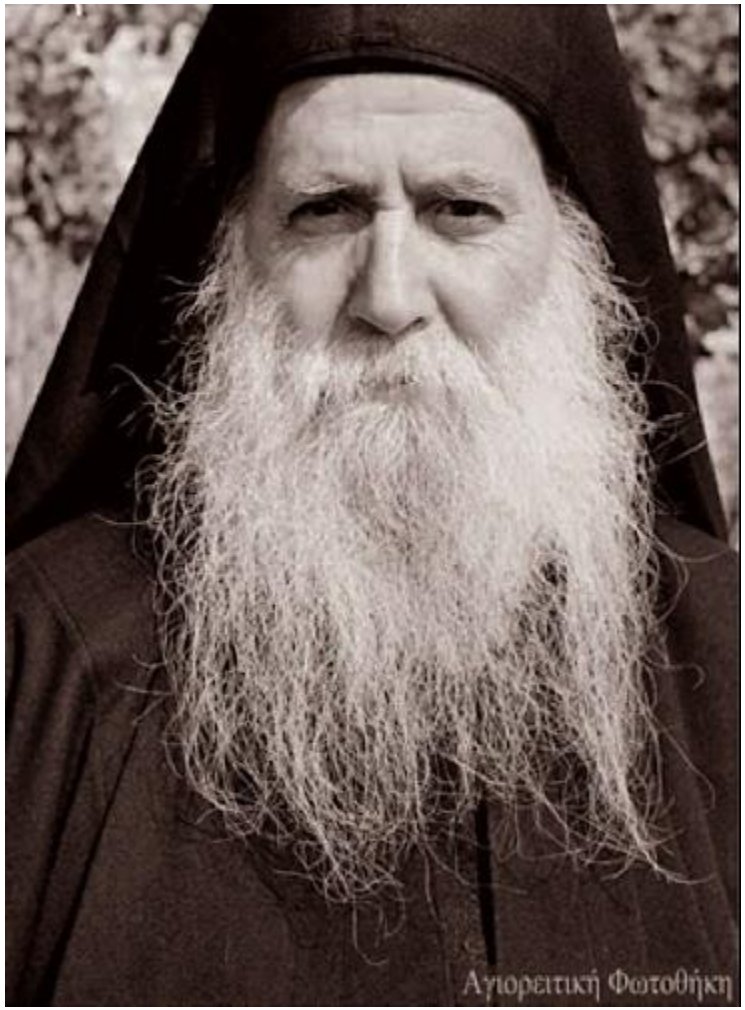
times( )

No extra calls happen

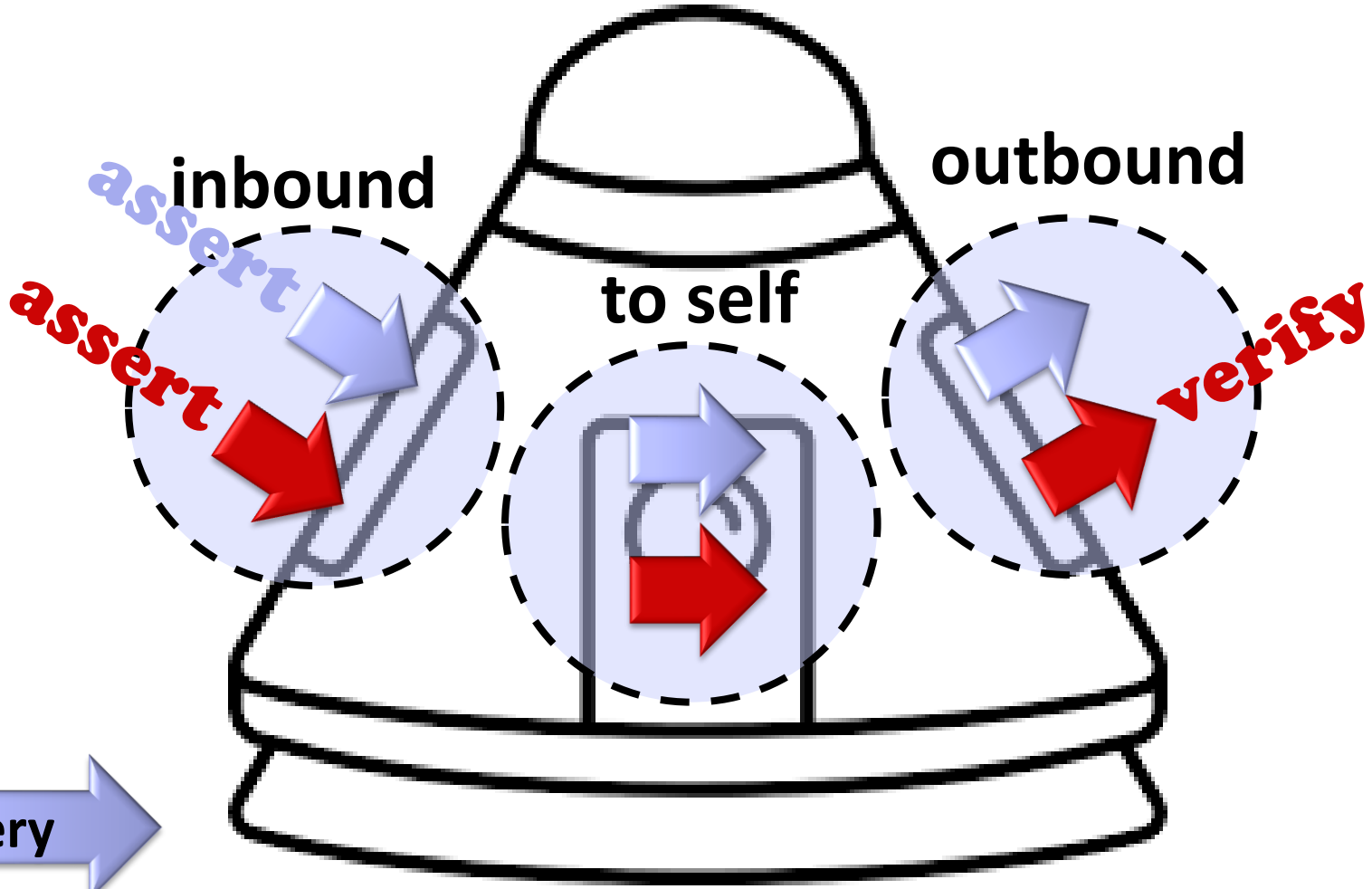
Capture-assert every argument

Terror-Driven Testing

Minimalistic Testing



What to test?



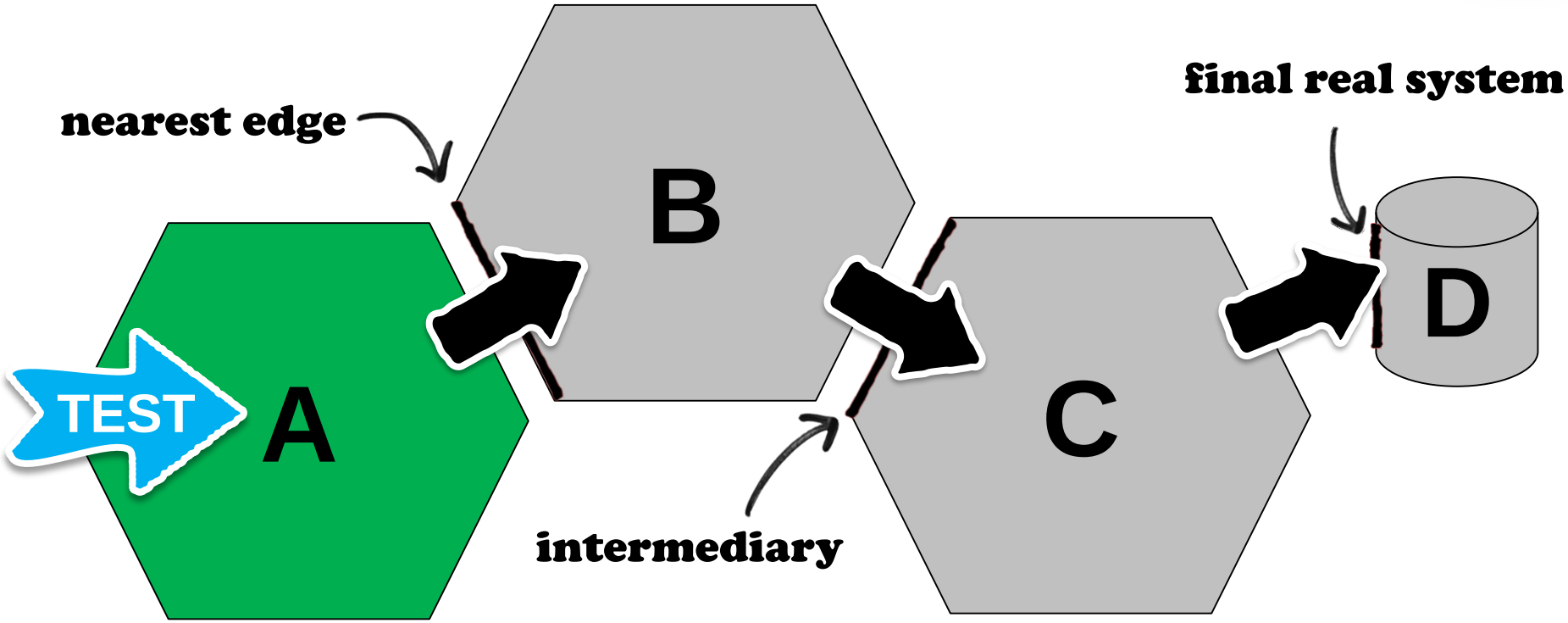
Inspired from: Sandi Metz - The magic tricks of testing https://www.youtube.com/watch?v=URSWYvyc42M

Minimalistic Testing

	query	command
inbound	assert	assert
to self	ignore	ignore
outbound	ignore	verify

Inspired from: Sandi Metz - The magic tricks of testing https://www.youtube.com/watch?v=URSWYvyc2UM

Where to Slice it?



Which API is stable?

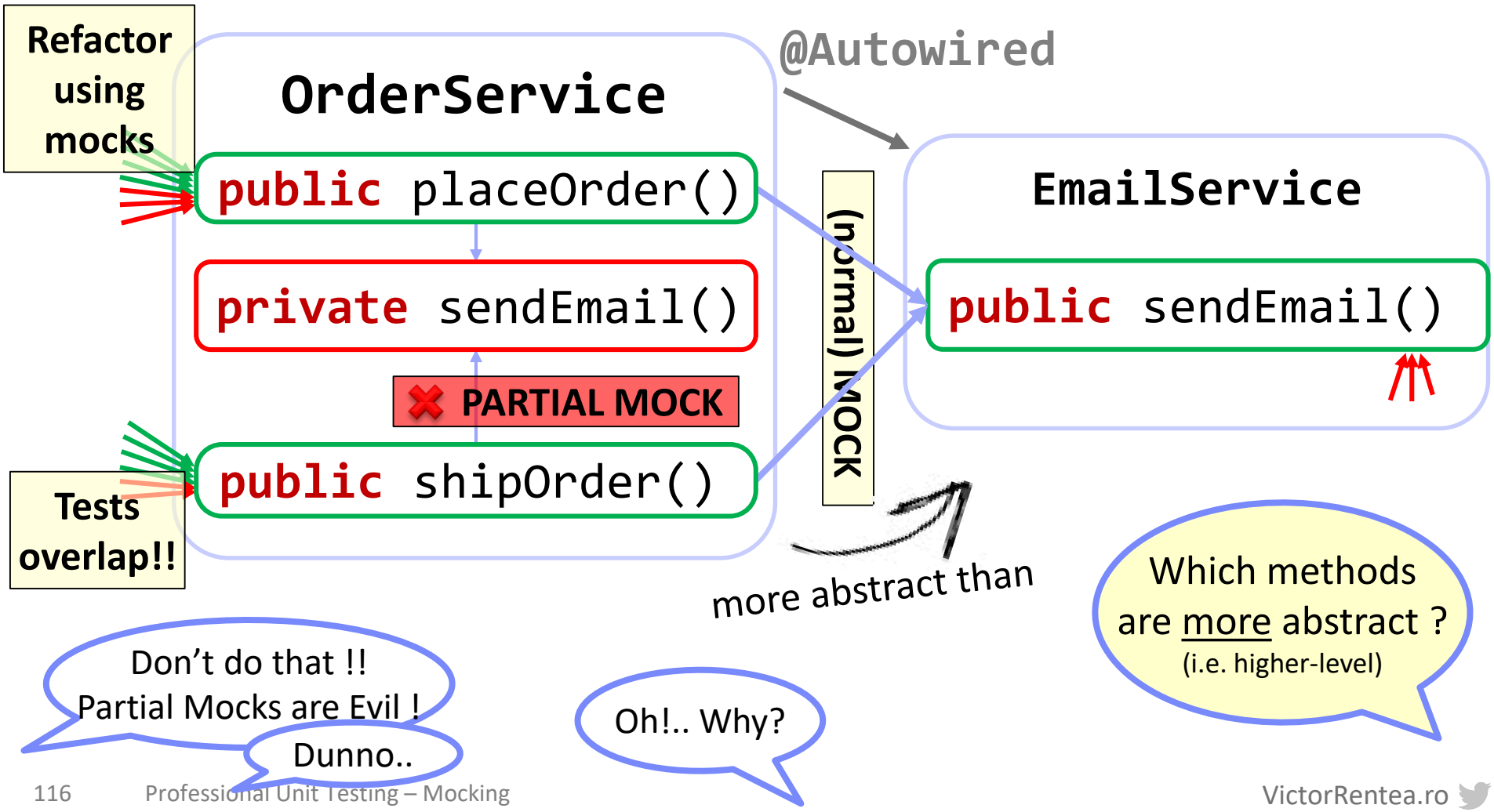
Yet reachable?

It may be simpler to test some classes together!

(versus testing them separately using mocks)

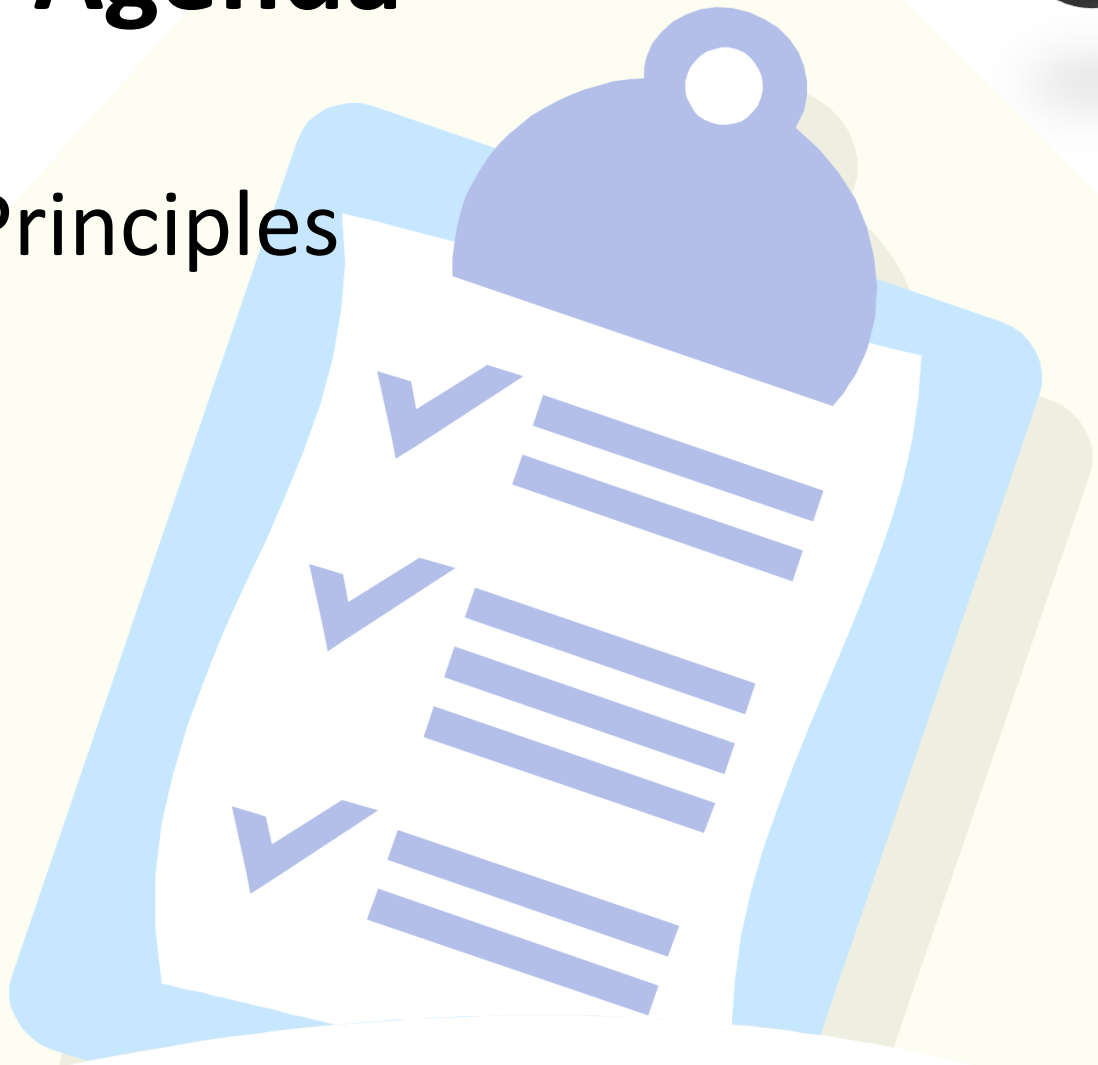
Separation by Layers of Abstraction

A Tale about the Evil Partial Mock... A.K.A. "SPY"

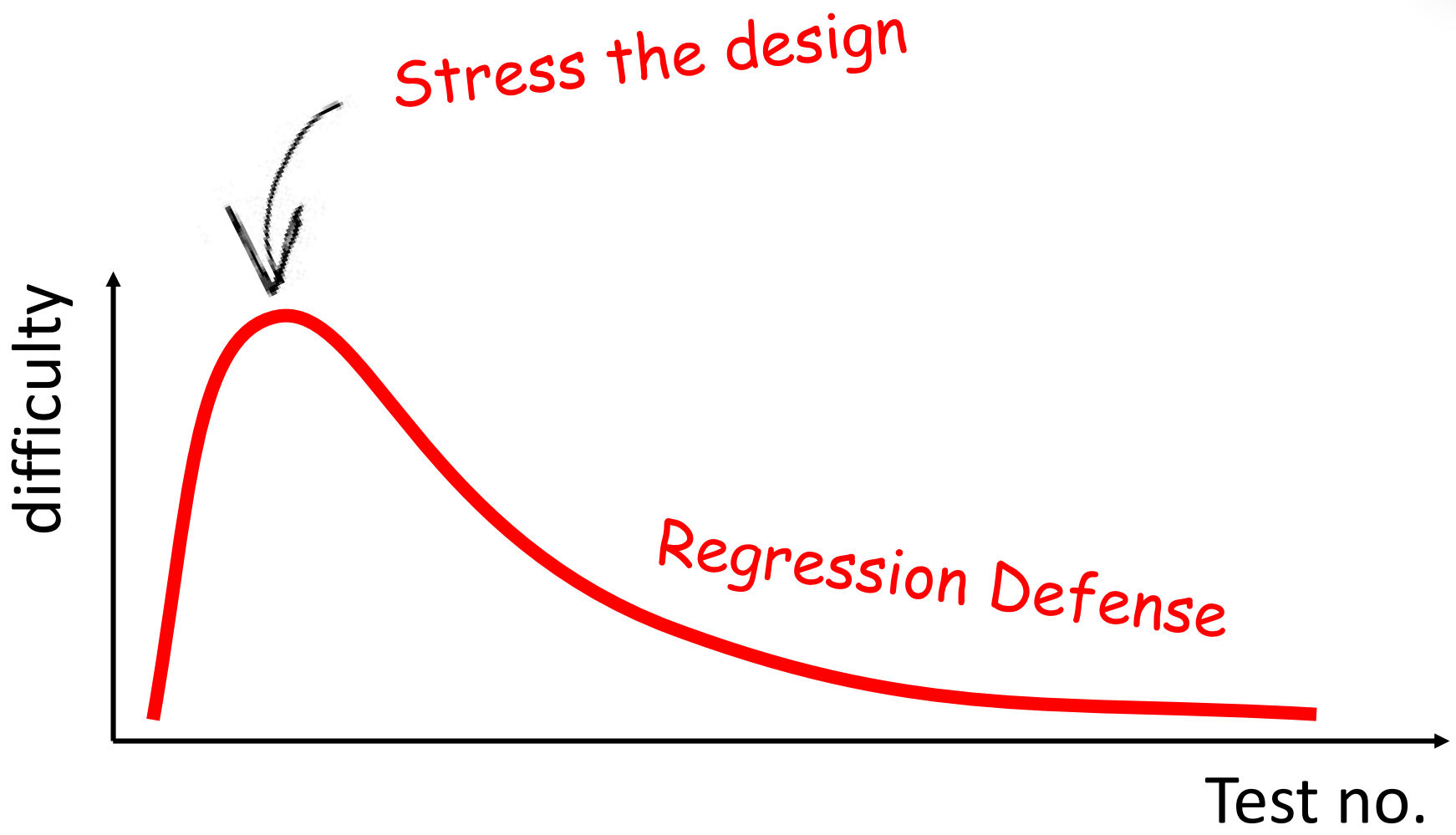


Agenda

- Test Design Principles
- TDD
- Mocks
- Databases
- **Legacy Code**



First couple of unit tests: Priceless



Unit Testing Legacy Code

The toughest task for any programmer, no matter how ferocious


Unit Testing Legacy is Hard
Requires Broad Vision
And Assuming Risks


A Legacy Unit Test is Long
Takes Strength to throw it

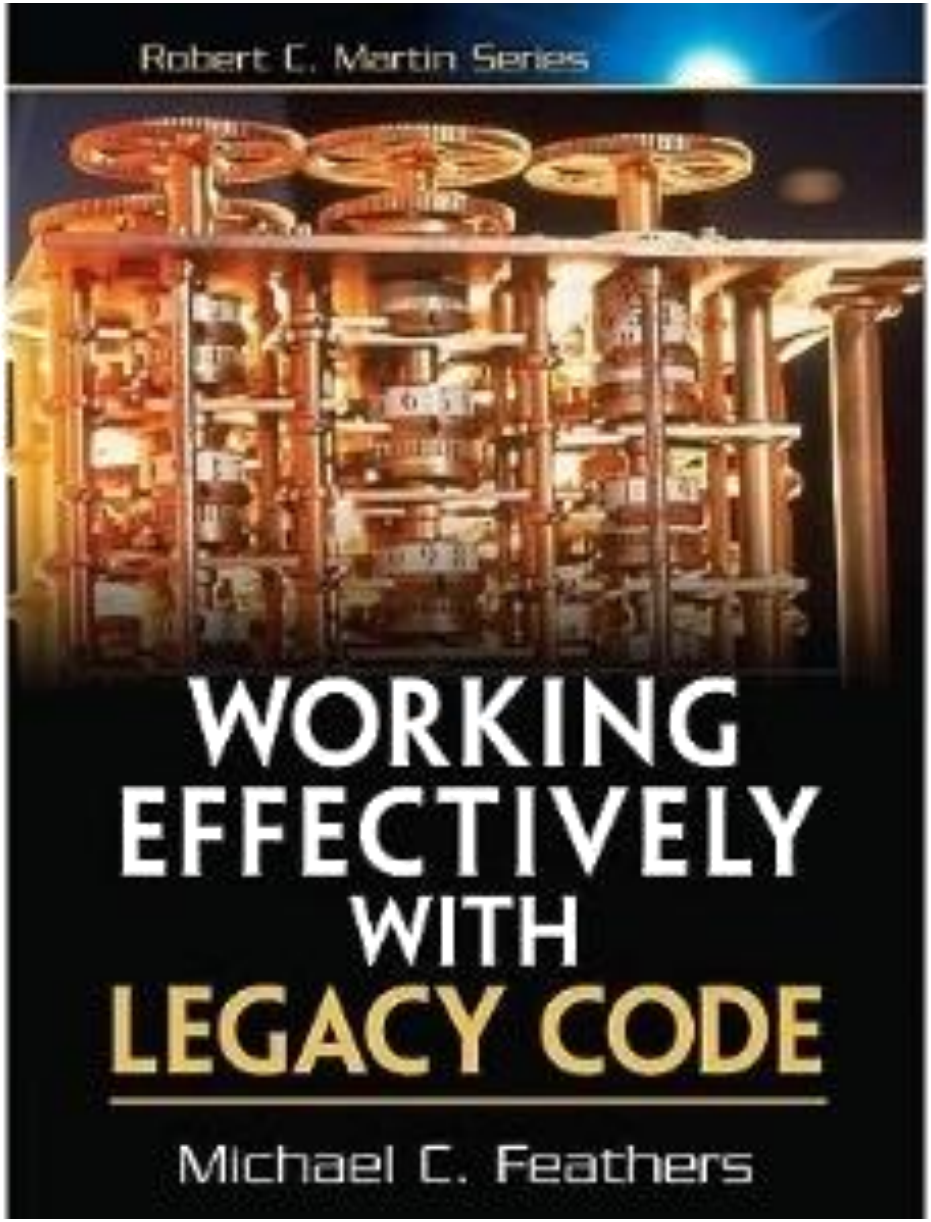

But can prove that
Even God Objects Can Bleed
When you slice a feature out to Unit Test it

Unit Testing Takes Practice

Some Techniques

for Testing Legacy Code

- **Extract-'n-test**
  - Then mock it out
- Break encapsulation
  - Expose internal state
  - Spy internal methods
- Hack statics
  - PowerMock methods
  - Control global state
- Introduce Seams
  - Decoupling places
- Exploratory refactoring
  - 30 mins that you always throw away
  - Search for ways to break the design to become more testable
- IDE Refactoring only
  - When not guarded by tests
  - Mob
- Temporary ugly tests
- Unit Test + **Refactoring**
  - Always together!

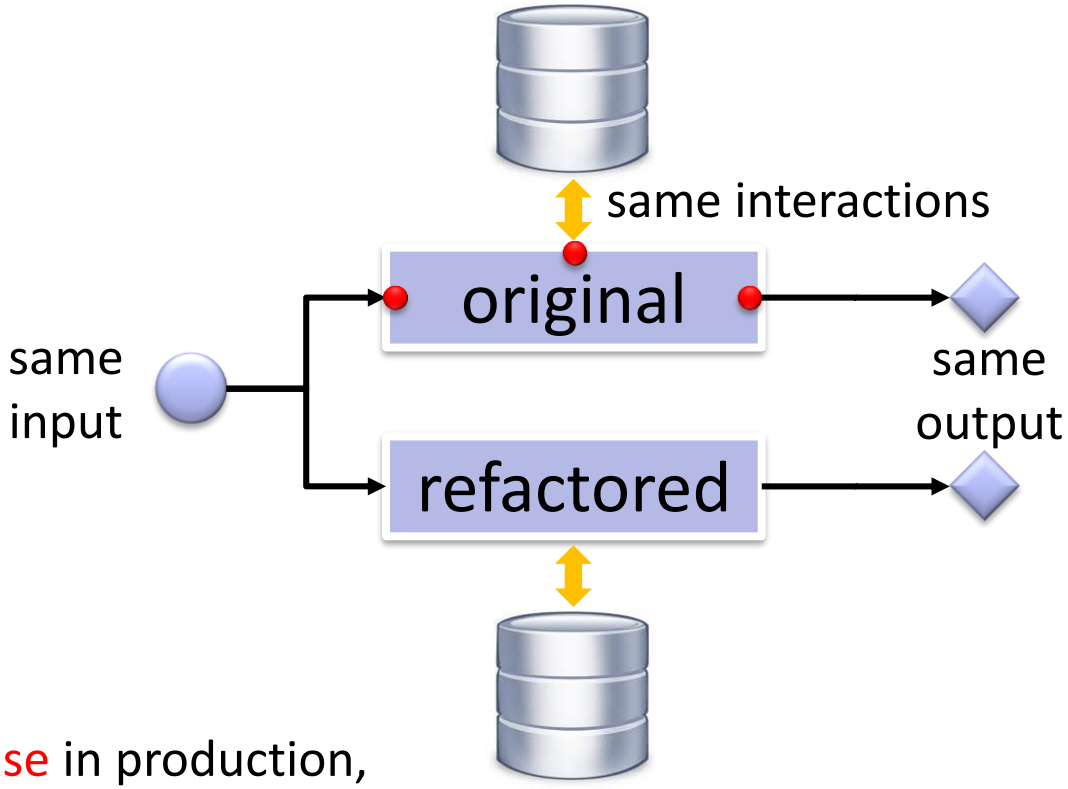

Robert C. Martin Series
WORKING EFFECTIVELY WITH LEGACY CODE
Michael C. Feathers

Testing Legacy Code: Feeling

https://www.youtube.com/watch?v=_gQ84-vWNGU

Testing Legacy Code

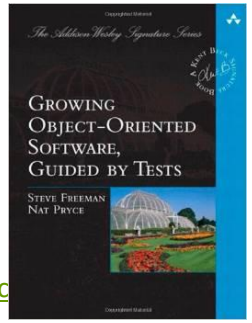
Golden Master Technique

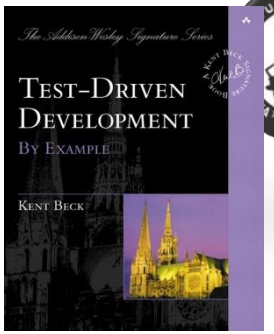


Idea: record these in production, then replay them on your refactor

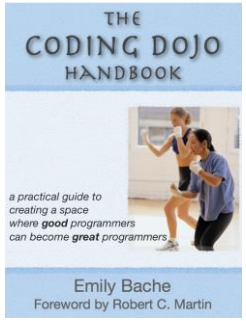
http://blog.adrianbolboaca.ro/2014/05/golden-master/

References

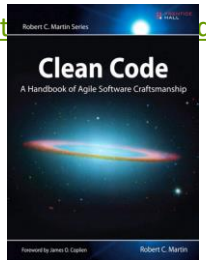


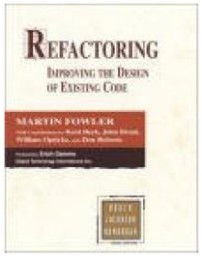


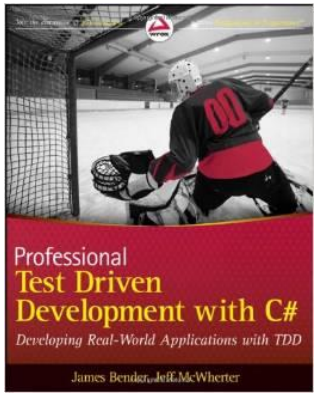
- **Clean Code + Clean Coders Videos** [CCEp#] http://www.cleanc
  Robert C. Martin (Uncle Bob)
- **(London-Style TDD): Growing Object-Oriented Software, Guided by Tests**
  Steve Freeman, Nat Pryce, 2009
- **(Classic-Style TDD): Test-Driven Development, by example**
  Kent Beck, 2000
- **The Art of Unit Testing**
  Roy Osherove, 2009
- **Coding Dojo Handbook**
  Emily Bache, 2009

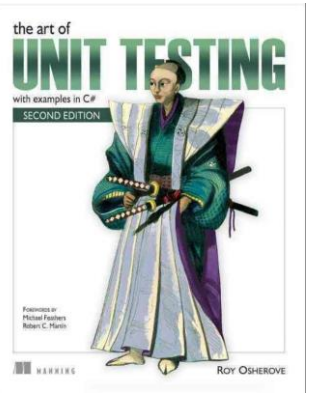


- https://springtestdbunit.github.io/spring-test-dbunit/faq.html
- **Professional TDD with C#: Developing Real World Apps with TDD**
  James Bender, Jeff McWherter, 2011
- **Agile in a Flash:** https://pragprog.com/book/olag/agile-in-a-flash
- **"Is TDD Dead?"** - on You Tube
- **Introducing BDD:** http://dannorth.net d/
  Dan North
- **Refactoring**
  Martin Fowler, 1999









Key Points

- Test where the **Fear** is
- Correct Tests fail ⇔ Prod mutates
- Explicit, Simple Tests
- TODO: Try TDD!
- Purify the heavy logic
- Listen to Tests: Testable Design

End of Part 1

What do you want after the break?

Proxies; Legacy Kata; TDD; Java8

Vote here: victorrentea.ro/vote

Come take 1 sticker:

Trainings, talks, goodies

Daily posts on

Too Fast!! Let's chat!

VictorRentea.ro

@VictorRentea