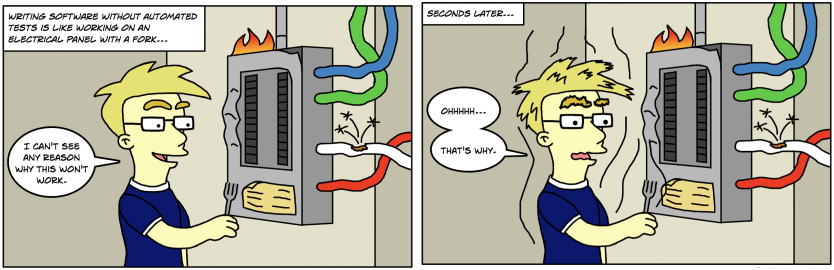
A Case for Automated Tests

Outline

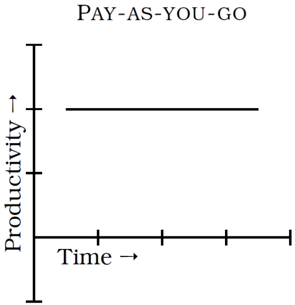
Security

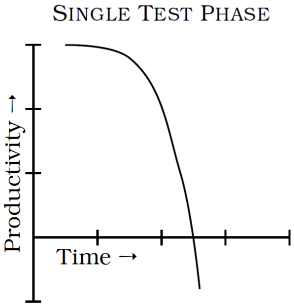

WRITING SOFTWARE WITHOUT AUTOMATED TESTS IS LIKE WORKING ON AN ELECTRICAL PANEL WITH A FORK...
I CAN'T SEE ANY REASON WHY THIS WON'T WORK.
SECONDS LATER...
OHHHHH...
THAT'S WHY.

Writing software without automated tests...[1].

[1] http://www.leonardscomic.com/comic/68/unit-testing/

Productivity

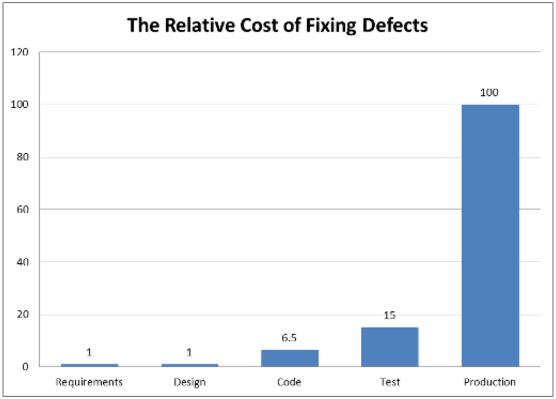




Constantly investing time on tests vs. having a single testing phase[1].

[1] Andrew Hunt, David Thomas: Pragmatic Unit Testing in Java with JUnit

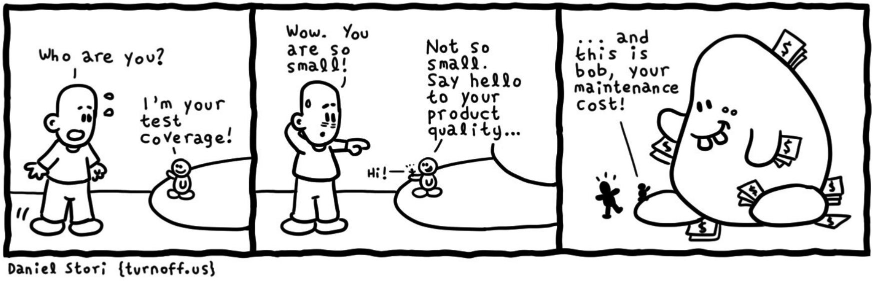
Money



The relative cost of fixing defects by product phase[1].

[1]IBM Systems Sciences Institute

Productivity, Quality, Money


Who are you?
I'm your test coverage!
Wow. You are so small!
Not so small. Say hello to your product quality...
Hi!
... and this is bob, your maintenance cost!
Daniel Stori {turnoff.us}

Why automated tests?

Automated tests

- are reproducible (in contrast to manual tests)
- provide fast feedback
- all tests are run automatically for every code change
- tests are documentation and provide examples
- avoid "works on my machine"

Why automated tests?

Automated tests are like a safety net

- new features
- updates
- refactoring

Example: The Fear Factor

```
for (int i = 0; i < size; j++) {
  // do stuff here
}
```

Is this a bug?
Or the reason that keeps the software running?

Time issues

I do not have the time

- It takes too long
- It's not a factor in the project plan

Time issues

I do not have the time

- It takes too long
- It's not a factor in the project plan

Why you should still do it

It **costs** time to write good automated tests.
It **saves** time later on to have them.

Time issues

I do not have the time

- It takes too long
- It's not a factor in the project plan

Why you should still do it

It **costs** time to write good automated tests.
It **saves** time later on to have them.

Bug hunting

The more code your write without testing, the more paths you have to check for errors.

Complexity issues

It is not possible

- too many dependencies
- initialising the required environment is cumbersome
- the method does too much

Complexity issues

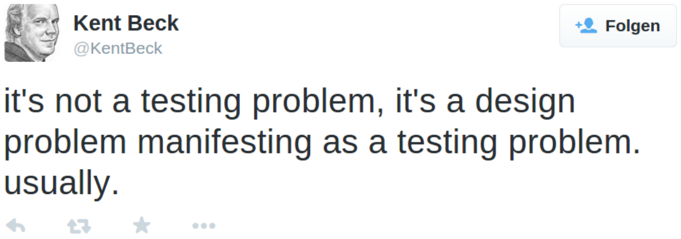
It is not possible

- too many dependencies
- initialising the required environment is cumbersome
- the method does too much

Why you should still do it

If a method is hard to test it is probably also difficult to use and maintain
$\Rightarrow$ Refactoring (Collective Code Ownership)

Complexity Issues? Or Design Issues?



Kent Beck
@KentBeck

it's not a testing problem, it's a design problem manifesting as a testing problem. usually.

Maintenance

Test code needs maintenance

- not flexible enough
- refactoring

Maintenance

Test code needs maintenance

- not flexible enough
- refactoring

Why you should still do it

Test code is not throw-away code.
Test code may be **more important** than production code.

Maintenance

Test code needs maintenance

- not flexible enough
- refactoring

Why you should still do it

Test code is not throw-away code.
Test code may be **more important** than production code.

Safety. Again.

If you want to refactor, the **essential precondition** is having solid tests[1].

[1] Martin Fowler, Kent Beck: Refactoring: Improving the Design of Existing Code

Benefits

Automated tests are absolutely necessary for

- Fearless Refactoring
- Continuous Integration
- Collective Code Ownership

Pitfalls

Tests are not a silver bullet.

Pitfalls

Automated tests

- do not magically reveal all errors
- do not fix badly written code
- if carelessly written, may create a false illusion of safety.

Take away message

The novice says: "I do not strive for 100% line coverage in tests; I only write tests for the code that is important."

The master says: "If the code is not important, why is it there at all? I will strive to test every line I write; if a line is not important, it should be removed."

[1] Paul M. Jones, Live Coverage in Unit Tests

Basic properties of a good unit test

A unit test should

- test exactly one component
- test exactly one behaviour
- be able to fail
    - logging statements are not sufficient.
    - use assertions. a lot.

Basic properties of a good unit test

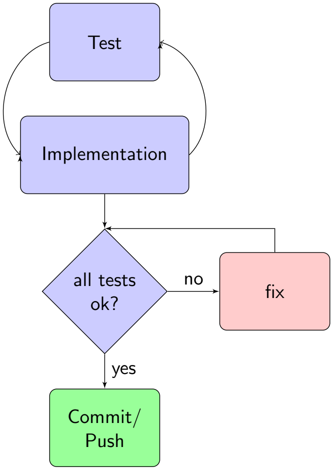
A unit test should

- test exactly one component
- test exactly one behaviour
- be able to fail
  - logging statements are not sufficient.
  - use assertions. a lot.

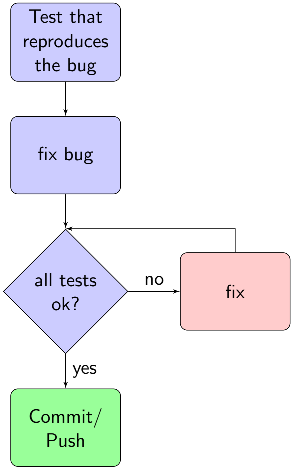
Best practice

Avoid happy path testing. Try to break the code.

Implementing a new feature

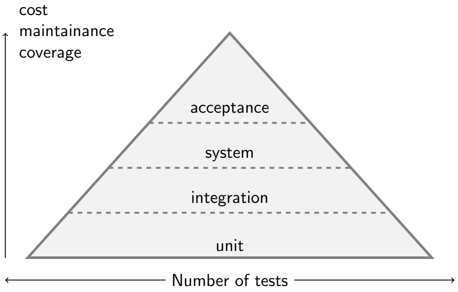


Fixing a bug


Test that reproduces the bug
fix bug
all tests ok?
no
fix
yes
Commit/ Push

Types of tests

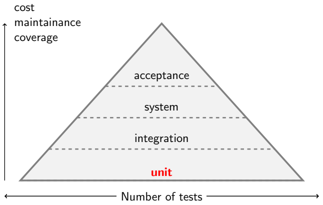
- unit tests
- integration tests
- system tests
- acceptance tests
- ...



Unit Tests

Make sure that **one particular unit** of the software does what it should.

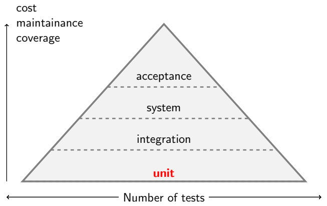
- Test my code, mock everything else.



Unit Tests

Make sure that **one particular unit** of the software does what it should.

- Test my code, mock everything else.



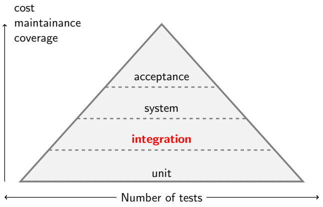
Keep in mind

Unit tests will not catch integration errors or broader system-level errors.

Integration Tests

Make sure that **some modules** of the software work together properly.

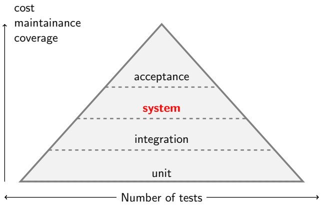
- my code + your code.
- code + database.



System Tests

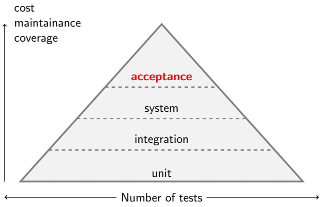
Make sure that **everything** works together properly

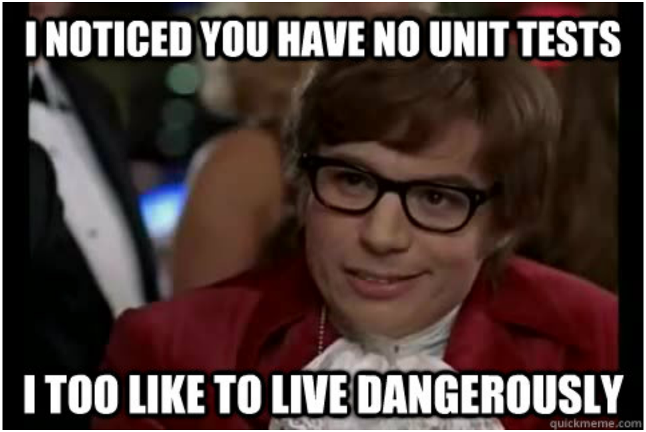
- my code + your code + database + server + client + $\cdots$



Acceptance Tests

- So far: did we build it right?
- Acceptance test: did we build the right thing?




I NOTICED YOU HAVE NO UNIT TESTS
I TOO LIKE TO LIVE DANGEROUSLY
quickmeme.com

So. Up to you...