The Unit Tests Strike Back

Dave Steffen, Ph.D.
Principal Software Engineer
dsteffen@scitec.com

https://scitec.com/

Properties of Good Tests

Science:

1. Precise
2. Accurate
3. Reproducible

Software Engineering:

- Complete
- Maintainable
- Robust
- Reliable
- Readable
- Hermetic

How we Get There

The Good Advice!

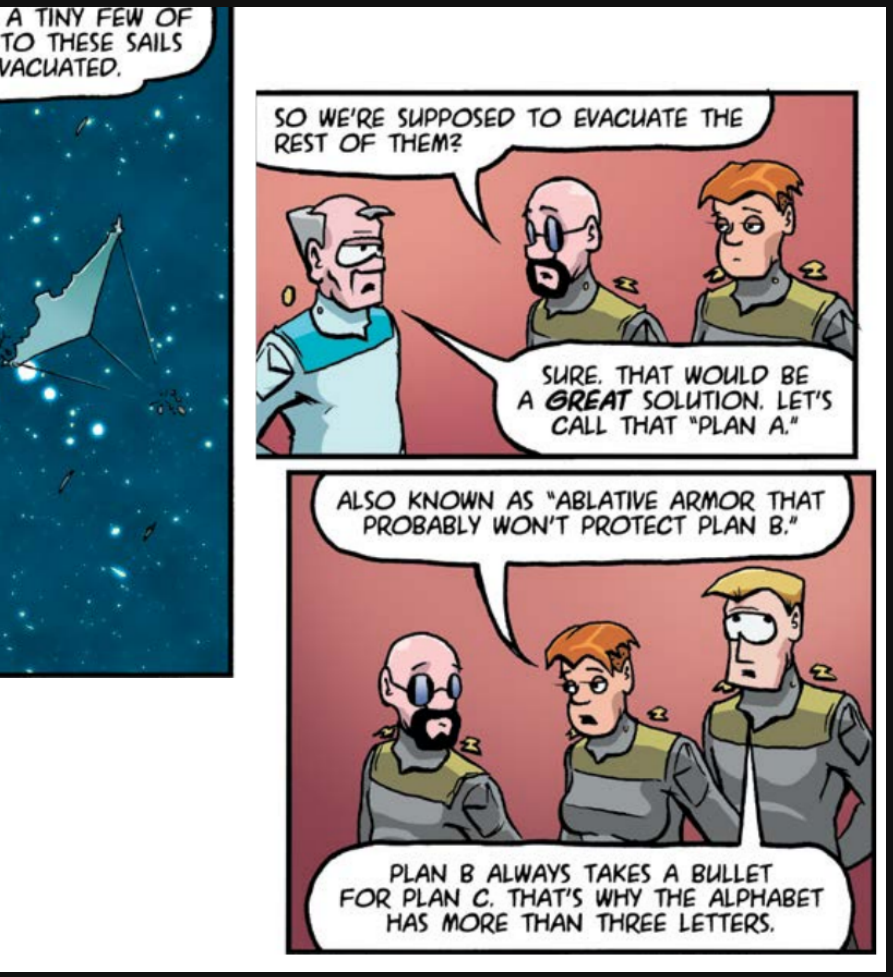
1. Use TDD Development
2. Use BDD Principles
3. Use only the public interface to test
4. Design for testability
5. Techniques for handling Legacy Code

Also see Brian Ruth's talk earlier this week

This is the plan for generating good unit tests

"No plan survives contact with the enemy."

- Helmuth von Molkte (1800 - 1891), translated by Correlli Barnett

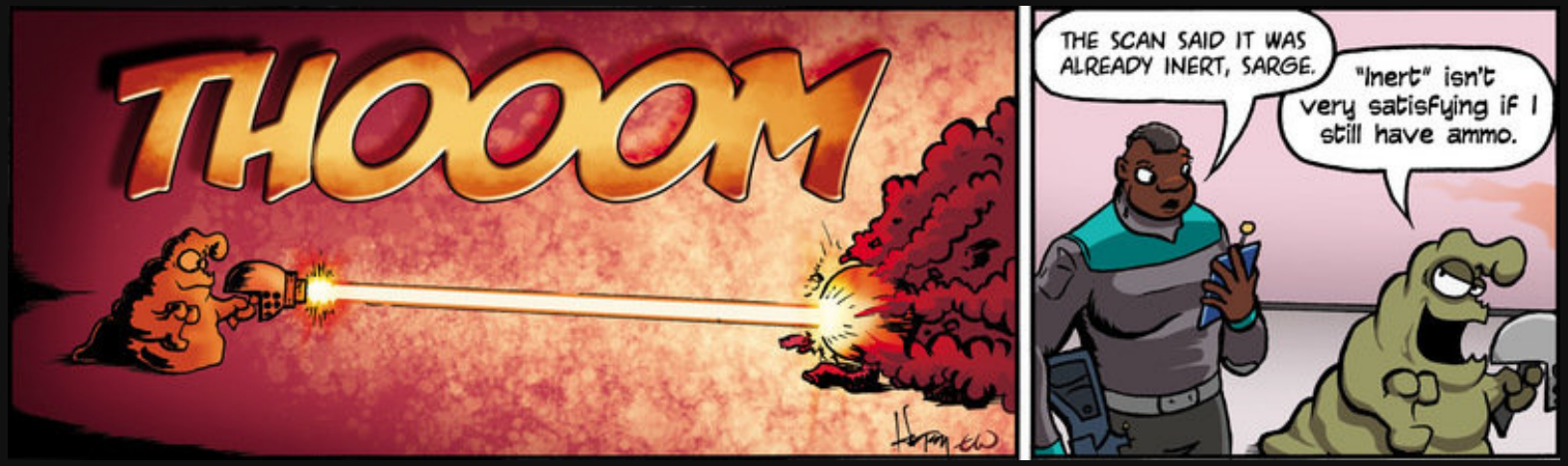


https://www.schlockmercenary.com/2015-09-01

If plan A is

"Follow the good advice",
what's plan B?

Good Advice is plan A



https://www.schlockmercenary.com/2017-05-30

Any test is better than no test.

If your backup plan is "we can't test that"
you need a better plan

Two kinds of hard tests:

- Part 1: The code is hard to **test**
- Part 2: The **code** is hard to test

Part 1:

The code is hard to **test**

The problem lies in the nature of what the tests are testing

1. Buggy nondeterministic code
2. Code with environmental dependencies

"Tests should fail because the code under test fails, and for no other reason" -- Titus Winters

Flaky Tests: tests that fail occasionally for no apparent reason

Fundamental Flakyness

Unit tests that fail occasionally for no apparent reason

Fundamentally Flaky tests stem from *buggy* nondeterminism (undefined behavior)

Plan A: Fix your code

Plan A Failure Modes

- UB is *hard* to track down
- Unit tests for "known-to-be-good" code have low return on investment
- Not annoying enough to justify the time

A Fix code

Fundamental Flakyness

Plan B: Ignore the problem

Accept occasional false failures

- Many of us live with flaky test anyway; maybe it's not worth the effort
- Incorporate into process (officially or otherwise)

Plan B Failure Modes

- May interfere with development process (at inopportune times)
- "Boy who cried wolf" syndrome
- *Developers become habituated to ignoring test failures*

A Fix B Ignore

Fundamental Flakyness

Plan C: Manage the Flake

1. Gather failure statistics
2. Define "pass" as no change to statistical behavior

- Rig testing framework to check for correct statistics, not individual results

Habituate your testing harness to flaky tests,
not your developers

Plan C Failure Modes

- Lack of resources

A Fix code B Ignore C Flake-aware tests

Fundamental Flakyness

Plan D: Abandon Unit Testing

This is Plan Ω

we will come back to this

A Fix code B Ignore C Flake-aware tests D Plan Ω

External / Environmental Dependencies

Testing non-hermetic code

- Sends data through communication interface
- Hits a database
- Writes to the filesystem
- Accesses custom hardware

Arguably, this is no longer *unit* testing

but we still need to do it somehow

Nonhermeticity is hard to deal with

Practically speaking, non-hermetic tests are just difficult, annoying, or flaky.

- Flaky / nonreproducible failures due to external causes, not the code under test
- Ties dev/build/CI environment to the external system
  - Might be limited by available hardware
  - Might be hard to arrange for unique instances

Non-hermetic tests

Plan A: Restore hermiticity with Mocks

Doubles / fakes considered useful:

- real objects are nondeterministic
- real objects are hard to set up
- real objects are hard to trigger behavior
- real objects are slow
- real objects have a user interface
- real objects don't exist (yet)
- test objects need to ask other real objects for information
- real objects are hardware or something not available

A Mocks

Non-hermetic tests

Plan A Failure Modes

- Mocks are expensive to make
- Mocks need independent testing
- Mocks are hard to hook in
- Mocks are unrealistic
- Mocks can mask problems

CppCon 2017: Peter Sommerlad “Mocking Frameworks considered harmful”

C++Now 2019: Kris Jusiak “Dependency Injection - a 25-dollar term for a 5-cent concept”

A Mocks

Non-hermetic tests

Plan B: Abandon hermeticity, embrace locality:

make the external system ubiquitous, local, and reliable.

1. Fold dependency into all build environs
2. Invest in dedicated resources

Assumption: if the external resource is reliable, the dependent unit tests are too

Plan B Failure Modes

- Resource is limited
- Only available at certain locations
- … see previous slide

A Mocks B Locality

Non-hermetic tests

Plan C: Accept and ignore occasional failures

- Many of us live with flaky test anyway; maybe it's not worth the effort
- Incorporate into process (officially or otherwise)

Plan C Failure Modes

- May interfere with development process (at inopportune times)
- "Boy who cried wolf" syndrome
- *Developers become habituated to ignoring test failures*

A Mocks B Locality C Ignore

Managing unreliable test resources

Plan D: Build independent sensors to detect outage

- Queue tests to run when the resource is back
- Flag tests as incomplete
- Habituate test framework to occasional failures, *not* your developers

Plan D Failure Modes

- Sensors must be highly reliable
  - Or represent an additional point of flaky
- Disruption to dev cycle

A Mocks B Locality C Accept D Detect outage

Fundamental Flakyness

Plan D: Abandon Unit Testing

This is Plan Ω

we will come back to this

A Fix code B Ignore C Flake-aware tests D Plan Ω

Part 1 Summary

Nondeterministic Code:

- Plan A: Fix the code
- Plan B: Accept flaky behavior
- Plan C: Rig to test for statistical success

Part 1 Summary

External and unreliable dependencies:

- Plan A: Mock / simulate external resources
- Plan B: Abandon hermeticity, retain locality; external resource is ubiquitous

Abandon hermeticity and locality; rely on external, possibly unreliable, resources

- Plan C: Accept false positives
- Plan D: Instrument to detect when tests can't run

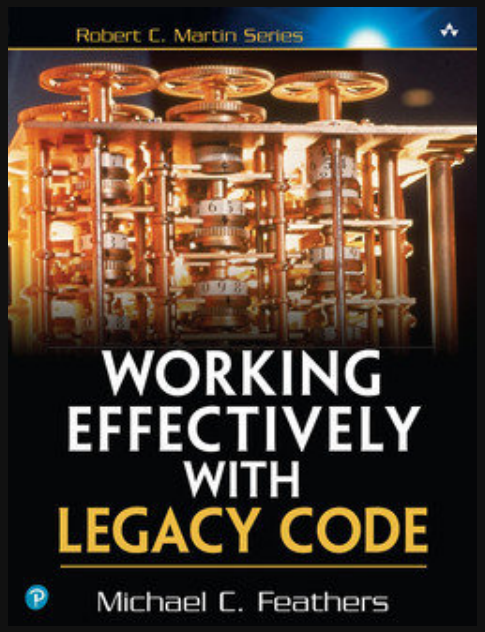
Either way, adjust processes to cope

Plans C and D provide data for requests to management for increased testing resources

Part 2:

The **code** is hard to test

The difficulty in testing lies in the stucture
or layout of the code itself.



Mostly but not always means legacy code.

The Simplest Example Ever:

```
1 class vector {
2
3   // no capacity access
4
5   private:
6
7     size_t capacity_;
8 }
```

You have designed std::vector

but you left out capacity()

You can't test resizing via the public interface

Plan A: TDD prevents this.

Plan A Failure Modes

- Legacy Code
- Someone else designed this.
- Added in a hurry during maintenance
- It's bad design: don't expose an internal detail

A The Good Advice (TDD, BDD, etc)

Plan B: Add Public Interface

```
1 class vector {
2
3   size_t capacity() const;
4
5   private:
6
7     size_t capacity_;
8 };
```

Just add the member function

Plan B Failure Modes

- "But that's hard to do"
  - probably symptomatic of other design problems
- "But that's a bad design"
  - you're probably wrong; good design is testable
  - is it worse than the alternatives?

Plan C: Refactor Untestable Behavior

```
1  class vector {
2
3    // still no capacity access
4
5  private:
6    // but this has been tested
7    array_storage data_;
8  };
```

```
1  class array_storage {
2  public:
3
4    size_t capacity() const;
5    size_t resize();
6    ...
7  private:
8    size_t capacity_;
9    int*     data_;
10 };
```

Plan C: Refactor the behavior into another class

- Possibly better than plan B?

Plan C Failure Modes

- "But that's really hard to do"
- More work, Riskier, More Intrusive

A B Add public interface C Refactor

Plan D: White Box (option 1)

```
1 class vector {
2
3   // no capacity access
4
5   protected:
6
7     size_t capacity_;
8 };
```

```
1 struct VecTester: public vector {
2   auto capacity() { return capacity_; }
3 };
4
5 TEST_CASE("Reserve increases capacity") {
6     VecTester v;
7     v.reserve(1000);
8     REQUIRE (v.capacity() >= 1000);
9 }
```

Plan D1: Change from private to protected, derive to test

Plan D1 Failure Modes

- Weakens encapsulation
- Changes source code
- Usual white-box testing issues

A B C Refactor D1 White Box (inheritance)

Plan D2: White Box (better)

```
1  class vector {
2
3    // no capacity access
4
5    private:
6
7      size_t capacity_;
8
9      friend class VecTester;
10 };
```

```
struct VecTester {
  VecTester (vector& v);
  auto capacity() {
    return v.capacity_;
  }
};

TEST_CASE("Reserve increases capacity") {
   vector v;
   VecTester tester(v);
   v.reserve(1000);
   REQUIRE (tester.capacity() >= 1000);
}
```

Plan D2: Give access to a trusted friend

If we are going to break encapsulation, do it correctly.

- Doesn't affect the design
- Breaks encapsulation but not in a way that matters ?

A B C D1 White Box (inheritance) D2 White Box (friendship)

White Box Failure Modes

Plan D2 Failure Modes

Maybe you can't change the source code

- Legal or regulatory issues
- Company or customer policy

Maybe you *don't have* the source code

- headers + precompiled library

A B C D1 White Box (inheritance)

D2 White Box (friendship)

Plan E: White Box In Anger

```
1 class vector {
2
3   // no capacity access
4
5   private:
6
7     size_t capacity_;
8 };
```

```
#define private public

#define class struct
```

```
#include "vector.h"

TEST_CASE("Reserve increases capacity") {
   vector<int> v;
   v.reserve(1000);
   REQUIRE (v.capacity_ >= 1000);
}
```

This is an even worse act of desperation

Plan E: Invoke undefined behavior

- No changes to source code
- Almost certainly works reliably in many compilers
  - … might change an available overload set

A B C D1 White Box (inheritance) D2 White Box (friendship) E define private public

Part 1 Summary

Code is hard to test because of its
interface and general non-testable-ness

white box {

- Plan A: Avoid this situation. (Use TDD, BDD, etc)
- Plan B: Redesign for testability: change the interface
- Plan C: Refactor that behavior out into another class
- Plan D1: Change access to protected, inherit and test
- Plan D2: Add a friend tester class
- Plan E: #define private public and damn the torpedoes

Plan Ω: abandon unit tests.

Plan Ω

Abandon unit testing

Software Testing is "Defence in Depth"

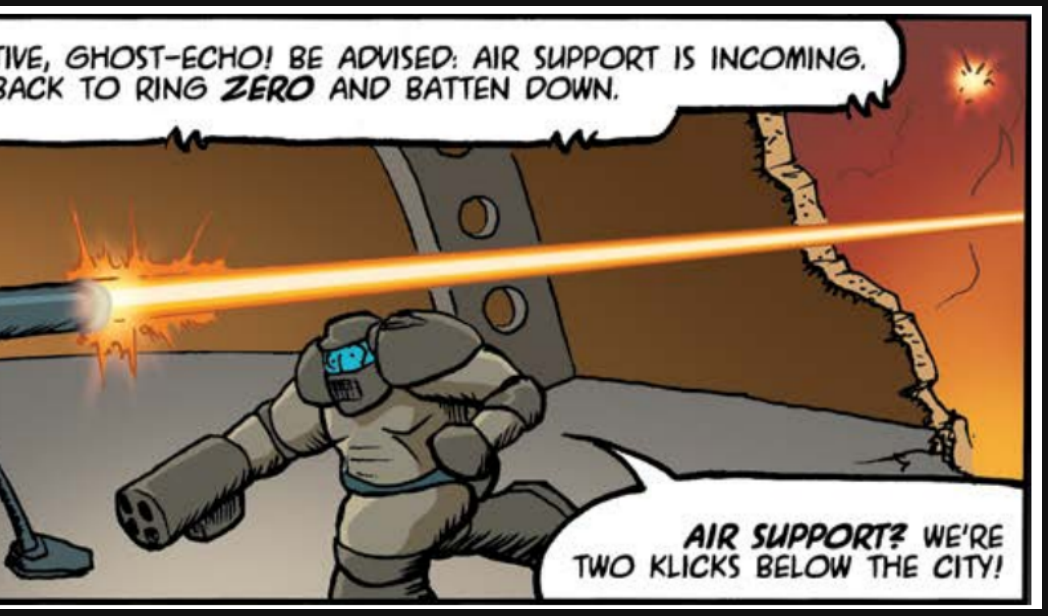
Unit Tests are just the first line

- Acceptance Testing
- System / Component testing (with sanitizers!)

Plan Ω

"We can't test this" is the path to failure.

But it's almost never true



https://www.schlockmercenary.com/2015-02-22

- A The Good Advice
- ...
- Ω Abandon unit testing
- Ø No Tests

Make your plans. Lots of them

References

All talks available on YouTube

- T. Winters and H. Wright, *All Your Tests Are Terrible...*

  CppCon 2015 https://youtu.be/u5senBJUkPc
- Fedor Pikus, *Back to Basics: Test-driven Development*

  CppCon 2019 https://youtu.be/RoYljVOj2H8
- Phil Nash, *Modern C++ Testing with Catch2*

  CppCon 2018 https://youtu.be/Ob5_XZrFQH0 (And see any number of other talks by Phil on the subject)
- Kevlin Henney: *Structure and Interpretation of Test Cases*

  NDC Conferences 2019 https://youtu.be/tWn8RA_DEic
- Kevlin Henney: *Test Smells and Fragrances*

  DevWeek 2014 https://youtu.be/wCx_6kOo99M

References

- Phil Nash "Test Driven C++ With Catch"
- Phil Nash "Modern C++ Testing with Catch 2"
- [Boost].µt

1