Auterion

Automated testing as part of PX4 development

Julian Kent
Auterion

PX4 Dev Summit 2020

Overview

01.

What is automated testing?

Automated testing as part
of PX4 development

Auterion

Test hierarchies

01
Unit testing

One function at a time

The lowest level of testing, to make sure each class or function works, completely independently of everything else.

02
Functional testing

A single library

Making sure that multiple functions and classes work together. In PX4 this includes uORB messaging.

03
Integration testing

Full system simulation

This should be as end-to-end as possible. In the PX4 world, this would be automated SITL testing.

04
User acceptance testing (UAT)

Manual testing

Finally the code reaches a real human, who manually interacts with it. Particularly when run on real hardware, bugs discovered here can be *expensive*.

02.

It’s extra work, why automate?

Automated testing as part
of PX4 development

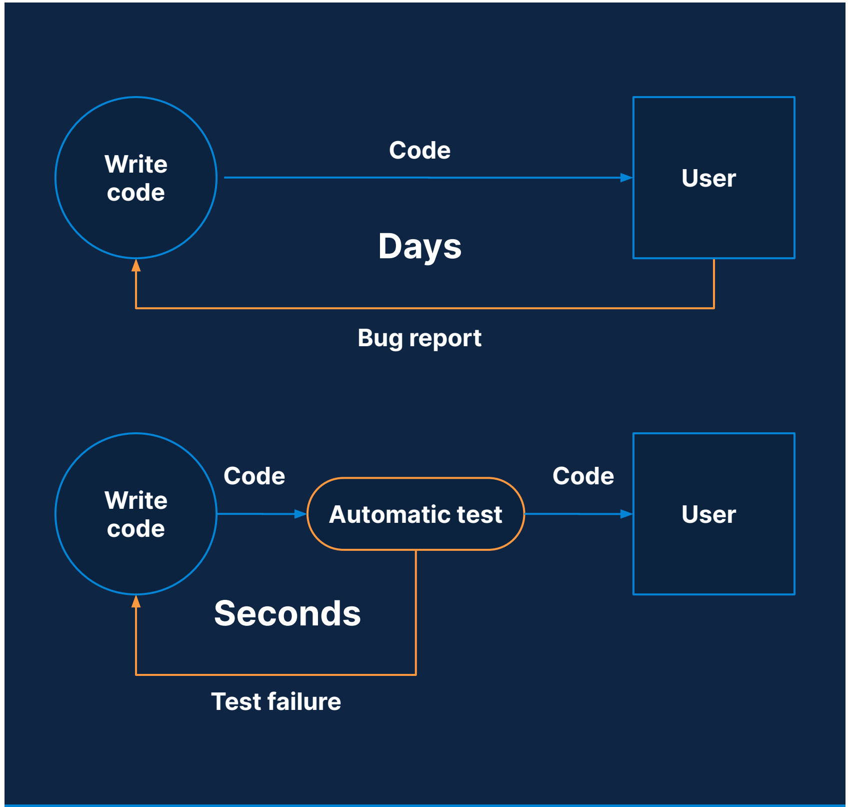
Auterion

Faster development cycles

Waiting for the code to get all the way to the user before checking for bugs can means days of waiting. By this time the developer has probably completely forgotten the details of what they were working on.

Faster development cycles leads to:

- Less overall development time
- Less new bugs introduced during fixes
- Happier users who find less bugs
- Happier, more focused developers



Using automated testing finds bugs faster, providing feedback to developers while code is fresh in their minds

Auterion

Don’t make humans do computer work

Computers are good at

- Running code
- Comparing outputs

Humans are good at

- Pattern matching
- Creative thinking
- Problem solving

Checking if code runs correctly

- Repeatedly running code
- Comparing outputs

→ **Computer work**

Designing tests

- Creating specifications
- Writing code

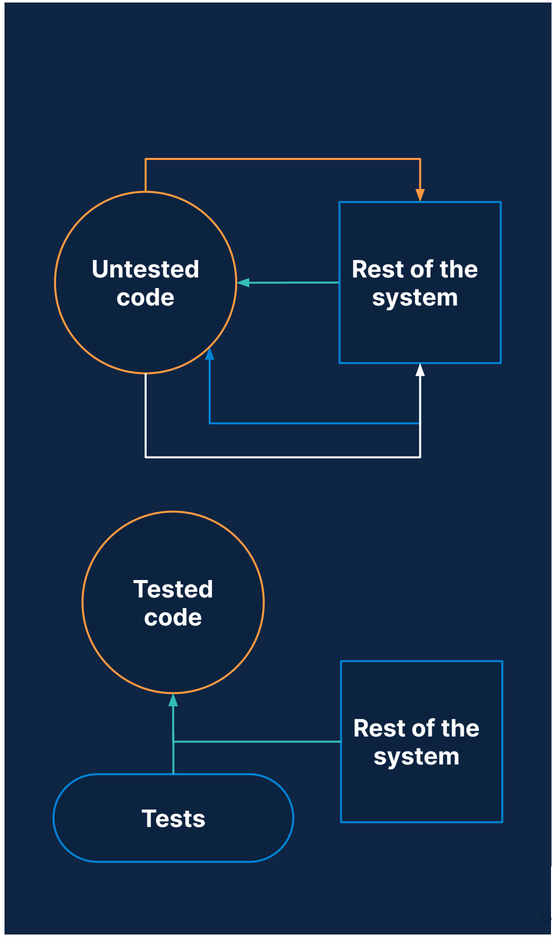
→ **Human work**

Tested code is reusable code

Major causes of non-reusable code are overly-specific APIs and circular dependencies

- Having unit and functional tests forces the API to be used from 2 places, both application code and the tests. If it is usable in 2 places, it is probably also usable in 3.

- Code cannot depend on the tests, so this forces the dependency graph to be non-circular, or at least outwards dependencies need to be generic.



Auterion

Low risk refactoring

Refactoring is often necessary, but also a major source of bugs.

Automated tests reduce bugs from refactoring in two main ways.

- Tests fail if the refactored code is broken.
- Code with tests is better specified and less likely to have surprising side effects.

Self maintaining modules

A

Untested

Under active development

More new bugs

As the module grows, the entire module needs to be held in the mind of the developer to keep it consistent and bug free.

B

Untested

Not under development

Hard to maintain

Changing code in other modules can cause unexpected breakage

C

Tested

Under active development

Fewer new bugs

The developer only needs to focus on the part they are working on, since other components will fail tests if they are broken.

D

Tested

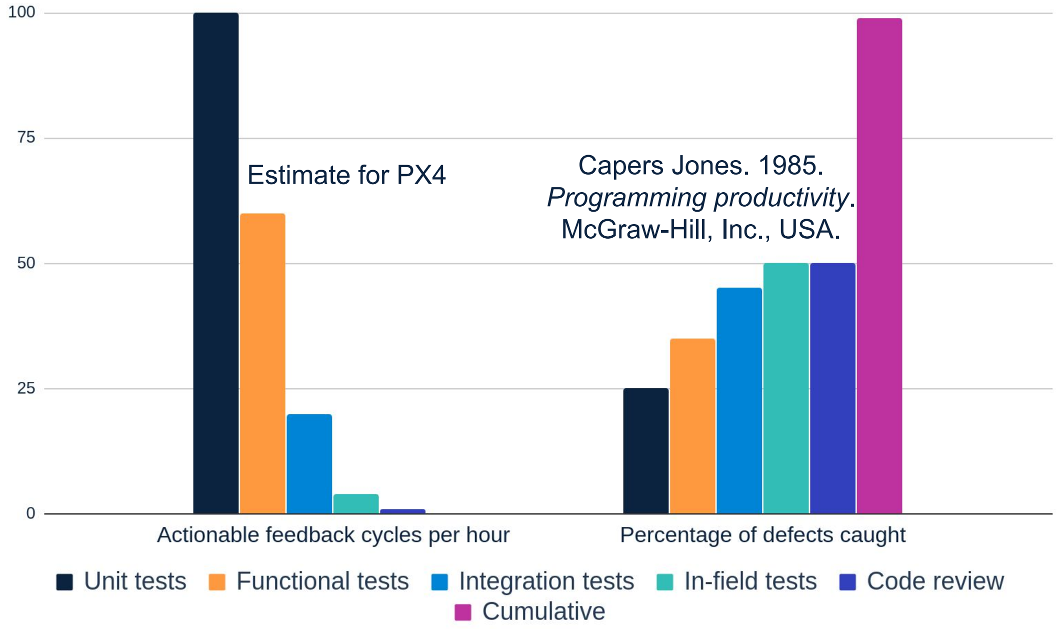
Not under development

Easy to maintain

Changes in other modules which would cause breakage will fail a test. Anything which gets through can have a test added, so it won’t happen again.

Industry outcomes

- Low level tests point to exact locations where bugs are, leading to very rapid debugging.
- Higher level tests catch more errors, but help less with identifying *where* the error is.
- Code review catches the most bugs, but has very limited bandwidth
- Each test / review type catches different classes of bugs. Eg. bugs in low-level code may just manifest as reduced performance at high level, but still pass tests.



Auterion

Who benefits

Community contributors

- Less likely to introduce bugs
- Faster review / merge process
- Less likely to break in future releases

Core developers / maintainers

- Easier refactoring
- Low stress maintenance - more bugs surface themselves before reaching users
- Faster development cycles

Manufacturers

- Less bug reports
- Faster custom feature development
- Safer airframe bringup and testing

End users

- Less bugs
- Faster bugfixes
- More features

03.

Automated testing in practice

Automated testing as part
of PX4 development

Where automated testing makes sense

New code

Get a minimal, empty API for the feature you want, then write the tests. As you complete the functionality you can get a ‘progress indicator’ based on how many of the tests pass.

Characterize untested code

Does it have bugs? Does it really do what you expect it to do? How about after someone changes it to add a feature or fix a bug?

Code to be debugged

Often bugs are located in very difficult to reproduce edge cases. Setting up a test for this edge case lets you verify that this is what the bug actually is, and also lets you verify that your solution fixes it.

Where automated testing $\underbrace{\text{doesn’t}}_{\text{sometimes}}$ work

Exploring / unknown specs

Tests act as specifications for your code. If you don’t yet know what your code is meant to do, then a test doesn’t make sense. However, a higher level test (where hopefully you *do* know what is meant to happen) can still be valuable to get quick feedback on whether the idea is working.

Glue code

Your code just sticks other pieces together. In this case the amount of dependencies of your code make it near impossible to test (in an automated manner) whether the behavior is as expected. Consider in this case just a basic ‘does it think it is running’ test to catch the most obvious bugs.

Test / build code

Do you need to test your tests? Probably not, unless you have a big library for generating test / simulation data or something similar that you want to make sure is behaving as it should

Z.O.M.B.I.E.S.

If you need an acronym for what tests to write: **Z.O.M.B.I.E.S.**

- **Z**ero - no data, null, and 0.0
- **O**ne - a single non-zero/null input
- **M**any - build it up until all of your inputs are good
- **B**oundaries - catch your off-by-1 errors
- **I**nterfaces - what does the API promise
- **E**xceptions - what you expect to give bad results
- **S**imple - simple setup with simple results - try to do this with all of your tests

Behavior Driven Development (BDD)

- Use your tests as specifications for your code
- If you know what the code is meant to do, write the tests first
- This can be used at any level: unit, functional, integration

Test structure:

```
// GIVEN: the initial conditions of the test
// WHEN: we do the thing we want to test
// THEN: we should get the output state we were expecting
```

Behavior Driven Development (BDD)

- Use your tests as specifications for your code
- If you know what the code is meant to do, write the tests first
- This can be used at any level: unit, functional, integration

```
TEST(Common, pointOutsideAzimuth) {
  // GIVEN: a regular FOV and a point outside
  FOV fov(34.0f, 12.0f, 90.0f, 60.0f);
  PolarPoint p_outside_azimuth(-1.0f, 126.0f, 2.0f);

  // WHEN: we check whether it is inside the FOV
  bool inside = pointInsideFOV(fov, p_outside_azimuth);

  // THEN: it should lie outside
  EXPECT_FALSE(inside);
}
```

04.

Adding a test to existing PX4 code

Demo
- edit: code here:
https://github.com/PX4/ecl/pull/864

Automated testing as part
of PX4 development

Auterion

Questions

Automated testing as part of PX4 development

More about Auterion on auterion.com