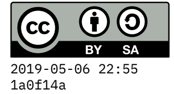
Testing


2019-05-06 22:55
1a0f14a

Contents

Purpose of testing

- QA = Quality Assurance
- "QA" leads us to believe that the purpose of testing is to **ensure it works.**
- A better focus/goal for testing is **to find the faults/bugs.**

  Testing can show the presence of bugs, not the absence of bugs.

  E.W. Dijkstra

Kinds of testing

There are lots of terms for different kinds of tests – we'll inventory some of them here.

Scope of what is tested

unit test unit means one isolated module within the software. Module = function, class, file. Mostly independent of other modules. **Mocking** = providing a fake dependency for a module under test.

integration test test how well 2+ (independently-tested) modules work together.

system test integration test that includes every module, entire system.

acceptance test a system test where the customer is determining whether it meets the requirements.

What info is used to construct tests

black box ignores the source code and design documents. Treats the system as a "black box."

white box (aka clear box) uses source code and design to create tests.

Tradeoff between these two: on one hand, more information seems better. But on the other, seeing the code can *bias* the kinds of tests you write.

regression test Whenever a bug is reported or discovered *outside* the testing mechanism, construct a corresponding test. That way, if we have a **regression** (aka move backwards, lose ground) on that issue, the test suite will discover it.

Testing non-functional requirements

performance test Goal is to understand efficiency

load testing Scaling the system to larger data sets

security test Test authentication, authorization, etc.

destructive test try to make system fail

A/B test two versions of the system against each other, often used for user interfaces to determine which is more effective. (Users are randomly placed in A group or B group, and see different versions of the system.)

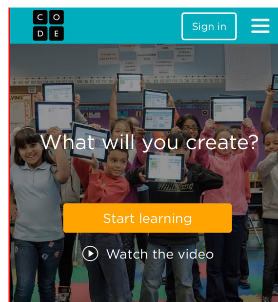


Figure 1: `code.org` home page – phrases like "Start learning" and "Watch the video" are often A/B-tested to determine what wording produces a better response rate.

Automation

- Ideally, running of the test suite is automated.
- Don't want to rely on human memory and consistency. Create programs to test programs.
- **Continuous integration** — there's a **suite** of tests that are automatically run on every commit or every push.

Test case selection/creation

Test "case" refers to a particular input paired with an expected output or result.

Example: maybe we coded a multiplication routine. For the inputs **3*4**, the expected output should be **12**.

It's impossible to have a **complete** set of test cases due to "state space explosion." Just for the simple multiplication program, if it works on 64-bit integers, there are 2×2^{64} = 36893488147419103232 possible inputs. Trying all of them, at a rate of one per nanosecond, could take more than a thousand years!

Moreover, two 64-bit integers is quite a simple input. For realistic, complex systems, the number of possible inputs is substantially larger.

So we must be selective about which tast cases are effective **at exposing faults.**

Manual test case selection

- Partition test case inputs into **equivalence classes.** Tests within the same equivalence class are likely to behave the same way.
- So pick just a few *representative* cases within each equivalence class.
- For integer multiplication, equivalence classes might be:

  two positive numbers `3*4 = 12`

  two negative numbers `-7 * -9 = 63`

  one negative, one positive `-5 * 8 = -40`

  smallish numbers As above

  large numbers `44050026 * 209384029 = 9223371921434754`

  overflow `209384029382 * 209384029382 = -6238902965364889308`

- Be sure also to focus on **boundary cases** (aka corner cases) — at the boundaries between two equivalence classes. For fixed-size integers: zero, one, negative one, maximum, and minimum values.
  - `4*0 = 0`
  - `4*1 = 4`
  - `44050026 * -1 = -44050026`
  - `-9223372036854775808 * 9223372036854775807 = -9223372036854775808`
- Another example: testing a user registration form with a **Last Name** field. What are the equivalence classes?
  - All lower-case
  - All upper-case
  - Empty name, “Null” vs NULL[1]
  - Non-letters (numbers, symbols, spaces) especially O’Reilly, Lin-Manuel, etc.
  - Different languages - é, ñ, ☒ (Unicode chars)
  - Really long (what should be the max size?)
  - Really short (what should be the minimum size?)

[1] `www.bbc.com/future/story/20160325-the-names-that-break-computer-systems`

Brenan Keller
@brenankeller

A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

4:21 PM - 30 Nov 2018

24,865 Retweets 62,028 Likes

Figure 2: @brenankeller on Twitter

Automatic test-case selection

property-based testing Encode a general property we want the software to have, the tool or library will generate test cases automatically to verify that property holds. Any test cases that cause the property to fail are reported as counterexamples.

fuzzing passing random inputs into a program. Could be random command-line arguments, random bits on standard input, simulate random clicks and keystrokes in a GUI. Goal is to see if it holds up or crashes, whether it gives access to sensitive data, etc.

How good are our tests?

coverage (related to white-box testing) indicates which lines of code were executed during a test suite. Any lines *not* executed are an opportunity to write more tests to exercise them.

fault injection If we intentionally plant bugs in the code, does the test suite expose them? If we automatically modify the code (think changing a single < to > or <=) — will a test fail? If not, was < the correct operator at all? When done automatically, this can be called **mutation testing.**

An example using `unittest` in Python

Wed 13 Feb

```python
import unittest

# Test-driven development (TDD)
# write failing test first, then code to pass the test.

# Suppose we want to convert hexadecimal numbers
# into base ten.  "2F" -> 47.
```

```
# This function deals with one digit at a time.
def hexDigitToInt(digit):
    if digit >= "A" and digit <= "F":
        return ord(digit) - 55
    else:
        if digit >= "0" and digit <= "9":
            return int(digit)
        else:
            raise ValueError("Invalid hex digit: " + digit)

# This function applies the conversion to a multi-digit string.
def hexToDec(hexString):
    hexString = hexString.strip()
    if len(hexString) == 0:
        raise ValueError("Empty string")
    columnValue = 1
    number = 0
    for i in range(len(hexString)-1, -1, -1):
        number += columnValue * hexDigitToInt(hexString[i])
        columnValue *= 16
    return number

class MyFirstTests(unittest.TestCase):
    def test_two_hex_digit_with_letter(self):
        self.assertEqual(hexToDec("2F"), 47)
        self.assertEqual(hexToDec("FC"), 252)

    def test_single_digit_letter(self):
        self.assertEqual(hexToDec("A"), 10)
        self.assertEqual(hexToDec("E"), 14)

    def test_invalid_single_digit(self):
        with self.assertRaises(ValueError):
            hexToDec("G")

    def test_two_hex_digits(self):
        self.assertEqual(hexToDec("19"), 25)
        self.assertEqual(hexToDec("99"), 153)
        self.assertEqual(hexToDec("A0"), 160)

    def test_multi_digit(self):
        self.assertEqual(hexToDec("3C1A"), 15386)
```

```
        self.assertEqual(hexToDec("5BD23546"), 1540502854)

    def test_single_digit(self):
        self.assertEqual(hexToDec("3"), 3)
        self.assertEqual(hexToDec("9"), 9)

    def test_extra_spaces(self):
        self.assertEqual(hexToDec("3CA "), 970)

    def test_empty_string(self):
        with self.assertRaises(ValueError):
            hexToDec("")
        with self.assertRaises(ValueError):
            hexToDec("     ")

    def test_one(self):
        self.assertEqual(1+1, 2)

# If this file is being executed directly,
# invoke the unittest.main() function.
if __name__ == "__main__":
    unittest.main()
```

Fuzz Testing

"Fuzz" = randomizing

Take a program and provide totally random inputs to it. This is automated by a tool.

Since inputs are totally random, we can't really specify what the outputs should be.

But at the very least, the program shouldn't crash or hang (not responding).

- Fuzzing Like It's 1989 | Trail of Bits Blog[2]

[2] blog.trailofbits.com/2018/12/31/fuzzing-like-its-1989/

Stress testing

Running a program under adverse conditions.

- Low memory
- Network outages
- Disk failures
- Power outages
- Low battery

- DoS attacks on network

Modern cloud-based networking may use Chaos Monkey (Netflix).

UI Testing

Tricky to test user interfaces, because inputs/outputs are not as simple, clearly-defined as with unit tests or command-line programs.

When there's something graphical and visual, the inputs/outputs are less simple to define and specify. Inputs may include mouse clicks, mouse motion, mouse drags, keystrokes – these can be simulated. Outputs can be harder, because we need to "read" positions on the screen, visual aspects like colors, frames.